Restructuring multi-level networks by using function approximations

J. Cortadella
Universitat Politècnica de Catalunya
Barcelona, Spain

M. Kishinevsky
SCL, Intel Corp.
Hillsboro, USA

A. Mishchenko
University of California
Berkeley, USA

***Abstract*— This paper presents a technique for multi-level minimization using implicit don't cares. One of the major problems in minimization is the calculation of the don't care set. Conventional approaches project the global don't cares onto the local support of each node to keep minimization cost low. This prevents restructuring multi-level networks. The approach presented in this paper enables the extension of the support for the projection of the don't care set, while keeping a low computational complexity. The extension of the support is based on the idea of function approximations. The approach has shown to produce tangible improvements in multi-level networks when compared with existing techniques.**

I. Introduction

Optimization of combinational multi-level networks is achieved by exploring flexibility of the network using logic simplification and restructuring operations.

The concept of implicit (or internal) don't cares is crucial for the node simplification. For each node n_i in a network, we can use the following don't care for simplification:

$$DC_i = EDC + SDC_i + ODC_i$$

where EDC is the external don't care and SDC_i and ODC_i are the satisfiability and observability don't cares, respectively, for node n_i. DC_i can be represented in terms of the primary inputs of the network (global don't cares) and projected, by image computation, on any subset of signals of the network (local don't cares).

Practical implementations of don't care-based minimization project DC_i onto the local support of the node, thus allowing the simplification of the node function and, possibly, the elimination of some signals in the support. However, the local projection of DC_i never permits restructuring the network by incorporating *new signals* to the support of the node. Some methods, such as `full_simplify` in `SIS` [1], partially lift this limitation by greadily considering a few candidate nodes whose support is a *subset* of the node being simplified.

In other words, powerful Boolean simplification and restructuring are in large decoupled in the current methods. Restructuring techniques are typically based on adding redundant connections to some gates and then deleting a different set of redundant connections. The implementations of such techniques are typically based on ATPG [2], global flow analysis [3], transductions on the decomposed network [4], or SPFD-based rewiring [5].

In this paper we describe a method that *couples simplification and restructuring together*:

- New potential candidates for the support of a node under minimization are computed.
- Then simplification selects actual fan-in nodes based on the results of the minimization and cost function estimation.

In this paper we focus on a single method for computing new fan-in candidates for the simplified node, which is based on function approximation (although other methods are possible). The basic idea is as follows: if a fan-in node, n, implements a global function, f_n, that is close enough to a global function, f_m, of the node m under minimization, then by incorporating n as a fan-in for m there are good chances to significantly simplify implementation of m. That is, nodes with functions that are good approximations of a minimized functions are plausible candidates to be included in the support as restructuring candidates.

Using the argot of SPFDs [6], the rationale is the following:

> *The logic function of a node distinguishes the on-set from the off-set. When optimizing the representation of the node, we rely on the distinguishing power of other nodes in the network whenever possible. By taking good approximations into the support of the node, the requirement for the node to distinguish is reduced, and as a result, the logic function of the node is simplified.*

A. An example

Let us analyze an example of `SIS` optimization with the following network (inputs: a, b, c, d; outputs: x, y, z):

$$\begin{aligned} x &= ac + \overline{a}\,\overline{c} + \overline{b}(c+d) \\ y &= b(ac + \overline{a}\,\overline{c}) + \overline{a}\,\overline{b}c \\ z &= a + b + d \end{aligned}$$

After running the SIS command `full_simplify`, the expression for x is simplified as follows:

$$x = y + \overline{a}\,\overline{c} + \overline{b}(c+d)$$

The simplification is possible because the SDC of node y,

$$y \oplus (b\,(\overline{a \oplus c}) + \overline{a}\,\overline{b}\,c)$$

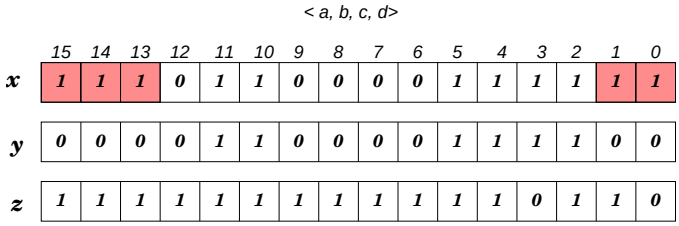


Fig. 1. Look-up tables for nodes x, y, and z.

(a negation of the second equation) is used during the minimization of x. Considering y for simplifying x was possible, since the support of y, $\{a, b, c\}$ is a subset of the support of x, $\{a, b, c, d\}$.

However, this approach is not able to incorporate x or z into the fanin of y, since their support is not a subset of the support of y.

Let us assume now that the support limitation is removed and we can project the DC for y onto other existing signals in the network, e.g. x. To check the effect of this with `SIS` we can simply "cheat" and artificially introduce x in the support of y by using a redundant form for y:

$$y = b(ac + \overline{a}\,\overline{c}) + \overline{a}\,\overline{b}cx + \overline{a}\,\overline{b}c\overline{x}$$

After executing `full_simplify`, the following simpler (in literal count) network is obtained:

$$\begin{aligned} x &= ac + \overline{a}\,\overline{c} + \overline{b}d + \overline{z} \\ y &= x(\overline{a}c + b) \\ z &= a + b + d \end{aligned}$$

B. Using function approximation for the example

Fig. 1 shows look-up tables for functions x, y, and z of the example. Let us compute the distance between the functions as a number of minterms in their boolean difference. E.g., functions x and y differs in five cells of the table (shaded boxes). The distances between all pairs of functions are:

$$\delta(x, y) = 5/16;\ \delta(x, z) = 7/16;$$
$$\delta(y, z) = 10/16;\ \delta(y, z') = 6/16$$

Note, that the invertion of z is closer to y, than z itself. Selecting between x and $\overline{z}$ as candidates for the support of y, we can conclude that x is better since it is closer to y (smaller distance). Moreover, x is an over-approximation of y, i.e., each time x is "1", y is "1". Hence, y can be implemented as an `AND` of y and some other function. Thus, x can be considered as a promising candidate to be incorporated in the support of y. Indeed, after minimization, the expression

$$y = x(\overline{a}c + b)$$

is obtained. The reader can check that no improvements would be obtained when trying to include z in the support of y. Other methods of computing distances between two functions can be explored as well.

Iterative restructuring techniques have good potential for area reduction at the cost of delay due to potential for sharing of nodes on the critical path [7]. As our experimental resluts show (cf. V) we have managed to avoid this drawback by special provisions when selecting restructuring candidates. In particular,

- There is a mode in which new candidates for the support cannot increase the logic depth of a node
- There is a mode restricting sharing on the new support signals that promotes tree-like structures friendly to the technology-mappers.

The numbers reported in our result tables are obtained with both modes on.

The rest of the paper is organized as follows. Section II presents basics of don't care computation. Function approximations are considered in Section III. Section IV describes methods for restructuring. Experimental results are reported abd discussed in Section V.

II. Internal don't-cares in muti-level networks

A. Previous Work

Flexibility in the network can be captured using different techniques: don't cares, Boolean relations, SPFDs, etc. In this paper we use don't cares as the most widely used and computationally feasible. Both Boolean Relations and SPFDs have power to incorporate simplification and restructuring in a single framework, however Boolean Relations are too computationally expensive for even small circuits [8] and similarly SPFDs implementations [9] did not combine support restructuring with simplification so far due to the large computational cost.

The well-known algorithm for don't-care-based simplification of binary networks, developed in [10], uses a combination of satisfiability don't-cares (SDCs) and compatible observability don't-cares (CODCs). It was implemented as command `full_simplify` in SIS [1]. Recently, it was shown that CODCs can be computed independently of the node's implementation [11].

A new scheme for computing the internal don't-cares of a node in an MV network (binary network is a particular case of MV network) is proposed in [12] and implemented in MVSIS [13]. The don't-cares computed are complete but not compatible. Therefore, they should be used to simplify the node before any other nodes are changed. Because the don't-cares are complete, they allow for a more thorough simplification, compared to compatible or partial don't-cares derived by other methods, which are subsets of the complete don't-cares [14].

Although in the past complete don't-cares have been derived for Boolean networks where each node has a single binary output [15] or several outputs (Boolean relations) [1], they did not find practical applications until recently because of the inherent computational complexity.

For network restructuring described in this paper, we use a new efficient implementation of the complete don't-cares computation in the SIS [1] environment.

B. Terminology

We assume the reader to be familiar with the concept of Boolean functions and Binary Decision Diagrams [16].

Definition 2.1: The Boolean function of a node (completely or incompletely specified) expressed in terms of the node's inputs is the *local function* of the node. The same function expressed in terms of the primary inputs (PIs) of the network is the *global function* of the node. The node function can also be expressed in terms of any set S of variables of the network (not necessarily fanins of the node or PIs of the network). This representation is called the projection of the node's function onto S.

The global functions are computed by visiting all nodes of the network in a topological order from the PIs to the PIs. The global functions of the PIs are set to be equal to the elementary variables. When a node is visited, the global functions of the fanins are already computed. To compute the global function of the node, its local function is composed with the global functions of the fanin nodes.

The projection of the node's function onto an arbitrary subset of nodes in the network can be computed by the image computation [17].

C. Don't-cares

The external don't-cares (EXDCs) for a primary output (PO) expresses the condition when the PO can take any value. Besides EXDCs, the sources of don't-cares at a node are the limited satisfiability and observability of the node, illustrated in Example 2.1 and Example 2.2, respectively.

Example 2.1: Let N be the network containing node n. Let x and y be fanins of n, expressed in terms of PIs of the network, a and b, as follows:

$$x = a \ \vee \ b \qquad y = a \ \wedge \ b.$$

It is easy to see that the combination ($x = 0$, $y = 1$) is never realized at the inputs of node n. This combination is an example of a local satsfiability don't-care of node n, because it is expressed using the fanin variables of node n.

Example 2.2: Let N be the network containing node n. Let node n have exactly two fanouts, u and v, specified using formulas:

$$u = n \ \vee \ a \qquad v = n \ \vee \ b$$

where a and b are the PIs of the network. It is easy to see that combination ($a = 1$, $b = 1$) makes the output of node n immaterial because it never reaches the POs. This combination is an example of a global observability don't-care of node n, because it is expressed using the PIs of the network.

Flexibility can be measured by the number of don't-care minterms in the corresponding ISF. The larger the don't-cares, the more freedom is available to optimize the node. For practical applications, it is important to have efficient don't-care computation procedures, e.g., similar to the one described below.

Definition 2.2: The don't-cares are *complete* if it is not possible to add another don't-care minterm, without affecting the functionality of the network.

D. Don't-care computation

The following definition is required to describe the procedure for computing different representations of the complete don't-cares of a node in the network.

Definition 2.3: For a given node n, the *cut (at n) network* is obtained from the original network by replacing n in the local functions of the n's fanout nodes by an additional PI of the network.

The cut network is a model used to detect the influence of the node on the global functions of the network. The actual representation of the original network is not modified when the cut network is used in the computation.

Theorem 2.1 (Global Don't-Care Computation [12]): Let the global functions and EXDCs of the POs of the network be, respectively, $F_i(X)$ and $DC_i(X)$, where i is the index of a primary output, and X is the set of PIs. Let the global function of the POs of the cut network be $F_i(X, z)$, where z is the additional PI representing a node. The complete global don't-cares of node z are computed as follows:

$$DC_z(X) = \forall z \bigwedge_i [(F_i(X) \oplus F_i(X, z)) \implies EXDC_i(X)]$$

Interpretation. The global don't-cares for the node is derived assuming that, for all output values of the node and all outputs of the network, the difference between the global function of the original network and global function of the cut network $(F_i(X) \oplus F_i(X, z))$ is contained ($\implies$) in the EXDCs. If there are no EXDCs, this formula requires that there were no difference between the global functions.

The next Theorem extends the local don't care computation from [12] in that the latter uses only the mapping of PIs into the local variables of the node, while the former allows for an arbitrary mapping.

Theorem 2.2 (Projected Don't-Care Computation): Let z be a node with the global don't-care function $DC_z(X)$. Let X be the set of PIs and $S = \{y_j\}$ be an arbitrary subset of nodes with the global functions, $Y_j(X)$. Then, the *don't-cares of node z projected onto S* are computed as follows:

$$DC_z(Y) = \forall X \ [M_S(X, Y) \implies DC_z(X)] \qquad (1)$$

where

$$M_S(X, Y) = \bigwedge_j [y_j \equiv Y_j(X)] \, .$$

Interpretation. The expression $M_S(X, Y)$ is the transition relation [17], which defines the mapping between the PIs, X, and the variables from the set S. The projected don't-cares are computed assuming that, for all PI minterms, either the mapping is false, $M_S(X, Y) = 0$, or the global don't-care is true, $DC_z(X) = 1$. When the mapping is false, the corresponding local minterms cannot be produced under any assignment of the PIs. This corresponds to the satisfiability don't-care of the node. When the global don't care is true, it means that the given assignment of PIs is in the scope of external and observability don't-cares. Taking all types of don't-cares into account, this formula derives the complete don't-care of the node z projected on set S.

III. Function approximations

The proposed selection of nodes to participate in the support of other nodes is based on the concept of function approximation. We first present some definitions required in this section.

Definition 3.1 (Satisfying fraction): Given a Boolean function f, the *satisfying fraction* of f, denoted by $|f|$, is the fraction of assignments that satisfy f.

The *satisfying fraction* is a value in the interval $[0, 1]$. As examples, the functions $f = a$, $g = ab$ and $h = a + b$, have satisfying fractions of $|f| = 1/2$, $|g| = 1/4$ and $|h| = 3/4$. The satisfying fraction can be recursively calculated as follows:

$$|f| = \begin{cases} 1 & \text{if } f = 1 \\ 0 & \text{if } f = 0 \\ \frac{|f_a|+|f_{\overline{a}}|}{2} & otherwise \end{cases}$$

This definition suggests a linear-time calculation of $|f|$ when f is represented as a BDD.

Definition 3.2 (Distance between two functions): The distance between two Boolean functions f and g, denoted by $\delta^*(f, g)$, is the fraction of assignments x, such that $f(x) \neq g(x)$. The distance between two functions can be calculated as

$$\delta^*(f, g) = |f \oplus g|$$

Usually, we are interested in the distance between f and g, or the complement of any of them. Thus, the concept of distance is extended as follows:

$$\delta(f, g) = \min(\delta^*(f, g), \delta^*(f, \overline{g}))$$

The following statements are easy to prove:

$$\delta^*(f, g) = \delta^*(\overline{f}, \overline{g}) \in [0, 1]$$
$$\delta^*(f, \overline{g}) = \delta^*(\overline{f}, g) = 1 - \delta^*(f, g)$$
$$\delta(f, g) \in [0, 1/2]$$

Distance is a measure of *similarity* between functions. If $f = g$ or $f = \overline{g}$, then the distance between both functions is 0. The reader will not find difficult to prove that the distance between two functions that have disjoint support is always $1/2$.

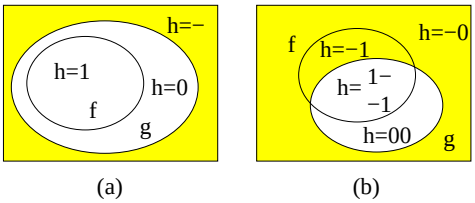
Definition 3.3 (Over- and under-approximations): f is said to be an *over-approximation* of g if $g \implies f$. In this case, g is said to be an *under-approximation* of f.

In this paper, we are interested in the minimization of Boolean networks. When doing so, using the flexibility provided by the don't-care space of the network is crucial. The concepts presented so far can be extended when a don't-care space is considered.

Definition 3.4 (Extensions with don't-cares): The previous definitions can be extended when a don't-care space, represented by a Boolean function d, is considered. The extensions are as follows:

$$\begin{aligned} \delta^*(f, g, d) &= |(f \oplus g) \cdot \overline{d}| \\ \delta(f, g, d) &= \min(\delta^*(f, g, d), \delta^*(f, \overline{g}, d)) \end{aligned}$$

and f is an over-approximation of g (or g an under-approximation of f) if $g \cdot \overline{d} \implies f$.



(a) (b)

Fig. 2. Minimization with function approximations.

For simplicity, we will omit the argument d when it is implicit from the context.

We say that a function f is a good approximation of another function g when the distance between them is small. We will next show that good approximations are promising candidates to participate in the support of a function under minimization. This is the basis of the restructuring approach presented in this paper.

Even though it can be argued that the size of the don't-care is not always a good indication of the flexibility to minimize a Boolean function, it is reasonable to assume that the minimization with a larger don't-care space tends to deliver better results, in general. The heuristics presented in this paper are based on this arguable fact. The results corroborate that the heuristics are useful to minimize Boolean networks.

A. Minimization with over- and under-approximations

Let us start with a simple example. Assume that f and g are nodes of a Boolean network and g is an over-approximation of f (see Fig. 2(a)). Thus, g can be a Boolean factor of f such that

$$f = g \cdot h$$

where h can be defined as an incompletely specified function as follows:

$$h^{on} = f; \quad h^{off} = \overline{f} \cdot g; \quad h^{dc} = \overline{g}$$

If we measure the flexibility to minimize h as the size of its don't-care set, we have that $|h^{dc}| = |\overline{g}|$. In other words, $|h^{dc}|$ becomes larger when $\delta^*(f, g)$ becomes smaller. Note that, in the ideal case in which $\delta^*(f, g) = 0$, we have that $h^{off} = 0$ and the trivial implementation $h = 1$ is valid. This indicates that f and g are equivalent and f can be substituted by g in the Boolean network.

The previous reasoning can be done when g is an under-approximation, with

$$f = g + h$$

and

$$h^{on} = f \cdot \overline{g}; \quad h^{off} = \overline{f}; \quad h^{dc} = g$$

with $|h^{dc}| = |g|$.

Finally, the extension for the case in which a don't-care space exists is straightforward.

B. Minimization with general approximations

Let us assume now that g is neither an over- nor an under-approximation of f. Thus, g can be a Boolean divisor of f according to the following expression[1]

$$f = g \cdot h_1 + h_2.$$

The flexibility for minimization is now distributed between h_1 and h_2 and can be specified as a Boolean relation as shown in the following table (see Fig. 2(b)):

Region	f	g	h_1	h_2
R_{00}	0	0	–	0
R_{01}	0	1	0	0
R_{10}	1	0	–	1
R_{11}	1	1	1	–
			–	1

In this case, the similarity between f and g determines the flexibility for $h = (h_1, h_2)$. In order to increase the flexibility we need to:

- reduce the region R_{01}, thus moving assignments to the region R_{00} and increasing the flexibility for h_1, or
- reduce the region R_{10}, thus moving assignments to the region R_{11} and increasing the flexibility for h_2.

Ideally, we would like to have $f = g$, thus allowing $h_1 = 1$ and $h_2 = 0$ be constant functions. In general we would like g to be a good approximation of f.

The approximation-based minimization can be also studied from the point of view of SPFDs [6], [9]. Each function f must perform an effort to *distinguish* the on-set from the *off*-set vertices. The support of the function provides enough information to do that, and the function combines that information to produce the result. If one of the inputs in the support, say g, is a good approximation of the function, it means that g has *already* performed a significant effort to distinguish the on-set from the *off*-set of f. Thus, the effort left for f is small and the complexity of the function can be low.

IV. Algorithms

The main contribution of this paper is the selection of the set of nodes S in equation 1 for the calculation of the local don't-cares. By defining S to include nodes that are not in the local fanin, the possibility or restructuring is introduced.

Two complementary approaches for restructuring are presented. The first one aims at reducing the complexity of each node individually. The second one attempts to eliminate nodes that can be substituted by other nodes.

In both approaches, the selection of candidates for restructuring a node is identical.

A. Selection of restructuring candidates

The set of candidates S to minimize a node n includes nodes from four different categories. The order of the categories is important since, for each category, only those candidates not included in the previous categories are eligible.

- `LF` (local fanin): the nodes in the local fanin of n.
- `SS` (same support): nodes whose local fanin is a subset of the local fanin of n.
- `OA` (over-approximation): nodes not in `SS` that are over- or under-approximations of n.
- `GA` (general approximation): nodes not in `SS` that are general approximations of n.

The calculation of approximations is done by using the global functions of the nodes and the global don't care of n. Note that the previous categories define a partition of the network since any node is a general approximation of another one.

The nodes in the transitive fanout of n are excluded from the sets `SS`, `OA` and `GA`. Otherwise, combinational cycles would be created in case they would be selected for restructuring.

After having classified all nodes, the best approximations are selected for each of the categories `SS`, `OA` and `GA`. In particular, the selection is tuned by three parameters K_{SS}, K_{OA} and K_{GA} that indicate how many nodes must be chosen within each category. The selected nodes are those with the smallest value of $\delta(m, n, DC_n)$ calculated in the global space.

B. Local restructuring

The approach used for local restructuring is identical to the one used for multi-level minimization, with the exception that the set S for the calculation of local don't-cares is extended with approximate candidates, as previously described. In our case, we use as a basis the approach presented in [12] restricted to binary multi-level networks.

C. Global restructuring

The minimization of each node is typically done by a two-level minimizer that aims at reducing the number of literals of a cover. With a greedy approach, restructuring would never be produced if that involves increasing the number of literals of an individual node. However, increasing the complexity of a node may have a positive influence in the rest of the network if some other nodes can be simplified or eliminated.

We say that a node n_1 is a *dominator* of another node n_2 if n_1 belongs to all paths from n_2 to any primary output. We also say that n_2 *is dominated* by n_1.

The strategy for global restructuring aims at eliminating nodes with single fanout and the nodes dominated by them (see Fig. 3). Thus, for each arc $m \rightarrow n$ in the network such that n is the only fanout of m, the following procedure is executed:

[1] A division as $(g + h_1) \cdot h_2$ is also possible, leading to similar conclusions.

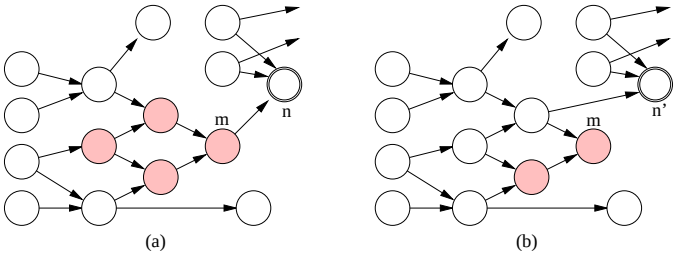


Fig. 3. Multi-level restructuring.

S := candidates for minimizing n;
dc := $DC_n(S)$; {calculation of the local DC with S}
on := "local function of n" $\cdot$ $\overline{dc}$; {local on-set with S}
off := $\overline{on + dc}$;
{exclude m from the support and check whether it is essential}
(on, off, dc) := $(\exists m\ on, \exists m\ off, \forall m\ dc)$;
`if` $on \cap off = \emptyset$ `then` { m is not essential }
  n' := minimize (on, off, dc);
  $\mathcal{D}$:= $\{m\} \cup$ set of nodes dominated by m if n would be substituted by n';
  `if` lits(n') - lits(n) $\leq$ lits($\mathcal{D}$) `then`
    Substitute n by n';
    Remove nodes in $\mathcal{D}$;
  `endif`
`endif`

Initially, the ISF for n using the extended support S is calculated. Next, m is excluded from the ISF by performing the exsitential abstraction in the on- and off-set, and the universal abstraction in the dc-set. The variable is not essential if the on- and off-set do not intersect after the abstractions. Next, the cost of substituting n by n', in which m is not present, is calculated. In case the increased cost is compensated by the removal of m and the dominated nodes, the restructuring is accepted. The nodes dominated by m are shadowed in Fig. 3. It is important to realize that some of the nodes dominated by m before restructuring (shadowed nodes in Fig. 3(a)), may not be dominated after restructuring (see Fig. 3(b)).

V. Results

The method for multi-level combinational logic optimization presented in this paper has been implemented as a `fsapp` command in `SIS`. The primary goal of the experiments was to compare the new method with

- a classical `full_simplify` command that uses the satisfiability and compatible observability don't cares and
- an `mfs` command that uses roughly the same strategy as `full_simplify`, but with complete don't cares (as explained in section II-A).

Our `fsapp` has two modes of operation corresponding to the global and local restructuring as explained in Sections IV-C and IV-B. The comparison strategy is to call `fsapp` for one iteration of global restructuring followed by an iteration of local restructuring in place of a call to `full_simplify` or `mfs`, correspondingly. The quality of results was evaluated after technology mapping using the `map -n1 -AFG` command of `SIS` and the `lib2.genlib` library. We also confirmed consistency of the results using another technology mapper and another library available to us. The experiments have been run on 69 circuits from the IWLS'91 benchmark set of multi-level combinational circuits [18] and its smaller subset of 34 circuits (for fast comparison).

Algorithm	Before Mapping			After Mapping	
	Levels	Fact. Lit.	CPU	Delay	Cell Area
full_simplify	240	7543	3.50	273.26	4865504
mfs	247	7457	5.30	277.58	4884528
mfs -r	236	11841	4.10	262.83	7488960
fsapp	218	7320	19.50	240.02	4757392

TABLE I
Comparing standalone application of `full_simplify`, `mfs`, and `fsapp`.

Script	Before Mapping			After Mapping		
	Levels	Fact. Lit.	CPU	Delay	Cell Area	Gates
sweep	794	111007	0.60	862.22	69422752	46496
rugged	831	45521	4317.30	946.84	30850432	21716
mfs	838	46979	245.90	955.27	31607680	22153
mfs -r	835	49238	241.00	950.87	33182960	23281
fsapp	808	39604	871.60	915.62	26806208	18917

TABLE II
Comparing application of `full_simplify`, `mfs`, and `fsapp` inside `script.rugged`.

First we compared the three methods taken standalone, i.e., by running the commands under comparison after the `sweep` on the original circuits from a subset of 34 circuits. Table I illustrates the comparison results. `fsapp` obtains an overall reduction of 12% in delay and 2% in area comparing to `full_simplify` and a bit larger reduction comparing to the `mfs`. The third line of the table corresponds to the application of `mfs` with no restructuring, i.e. when support of the node cannot be changed by incorporating other nodes from the "same support" category. The results in this case are much worse in area than the default `mfs`.

The second set of experiments compared `fsapp`, `full_simplify`, and `mfs` within the `rugged` script, a popular `SIS` script that uses `full_simplify`. We replaced `full_simplify` inside `rugged` with `mfs` and `fsapp`, correspondingly. A larger set of 69 circuits was used for this comparison. The results are presented in Table II. The first line gives as a reference the resulting numbers for the original circuits (after `sweep`), the second line is for the `script.rugged`. A large run time for the `script.rugged` is primarily due to one example, `too_large`. Overall, we can conclude that even after significant amount of optimization by the ealier commands of `script.rugged`, `fsapp` could deliver about 13% area reduction simultaneously with 3% delay reduction (after technology mapping) as compared with the `rugged` and even bigger delay and area reduction comparing with the `mfs`.

Table III gives comparison on the individual examples between the `script.rugged` (the right part of the table) and it's version with `fsapp` (the left part of the table). Only

			rugged with fsapp					rugged classic (with full_simplify)				
example	PI	PO	LIT	DPTH	GATE	AREA	DELAY	LIT	DPTH	GATE	AREA	DELAY
alu4	14	8	1293	27.0	579	840304	30.63	1256	26.0	623	873712	30.89
apex6	135	99	1288	14.0	569	825456	15.55	1344	14.0	599	868608	15.76
apex7	49	37	384	11.0	206	279328	11.82	402	12.0	223	301600	11.73
b1	3	4	16	3.0	7	10208	3.85	17	4.0	10	13456	4.30
b9	41	21	198	7.0	106	145232	7.71	265	8.0	129	180960	8.35
C17	5	2	13	3.0	7	9280	3.04	14	3.0	10	12528	3.58
C1908	33	25	798	27.0	412	567472	31.55	810	27.0	424	582320	31.36
C2670	233	140	1281	28.0	612	859792	27.24	1512	28.0	704	999920	28.61
C3540	50	22	2551	35.0	1221	1732576	37.22	3900	36.0	1747	2538544	40.68
C432	36	7	409	33.0	255	333152	33.80	561	32.0	283	396720	35.72
C5315	178	123	2883	27.0	1129	1710304	27.55	2950	28.0	1190	1783152	28.22
c8	28	18	239	7.0	110	157760	8.47	238	7.0	112	159616	8.29
cc	21	20	82	4.0	52	68208	6.53	84	5.0	55	71456	6.29
cm151a	12	2	53	7.0	19	30624	5.86	55	6.0	25	36656	6.25
cm162a	14	5	76	7.0	41	54752	6.80	102	8.0	54	73776	8.15
cm85a	11	3	81	6.0	37	52896	6.96	84	7.0	41	58000	8.04
comp	32	3	170	9.0	99	131776	10.31	174	10.0	103	135952	11.82
cordic	23	2	118	8.0	54	77488	6.95	134	9.0	57	83984	7.60
cu	14	11	94	6.0	46	64960	8.67	95	6.0	48	67280	8.61
frg1	28	3	383	9.0	164	242208	11.83	390	10.0	166	245920	12.06
frg2	143	139	1269	15.0	618	863968	19.66	1352	14.0	679	942848	17.42
i10	257	224	4123	32.0	1891	2710224	38.84	6064	35.0	2729	3944000	45.84
i1	25	16	71	7.0	45	58000	7.37	74	8.0	47	60784	7.36
i3	132	6	256	5.0	178	223648	4.60	620	7.0	306	427808	6.37
i8	133	81	2111	10.0	864	1300128	15.10	2132	10.0	917	1355808	14.91
i9	88	63	1050	9.0	402	628720	12.77	1054	9.0	567	782304	13.19
lal	26	19	171	6.0	88	122032	7.29	179	7.0	94	128992	9.03
pair	173	137	2931	23.0	1246	1835120	24.42	3062	27.0	1326	1953440	28.35
pm1	16	13	71	7.0	40	52432	9.58	75	7.0	44	58464	9.12
rot	135	107	1207	16.0	570	807360	19.42	1376	16.0	651	925216	21.62
t481	16	1	210	10.0	142	181424	11.33	1362	15.0	662	932640	16.95
term1	34	10	377	9.0	174	248704	9.97	371	9.0	170	245456	10.41
ttt2	24	21	371	10.0	167	238960	12.57	408	10.0	187	268656	13.79
x1	51	35	628	7.0	284	409248	8.56	642	7.0	287	415744	9.01
x3	135	99	1380	12.0	608	882992	13.23	1534	12.0	608	927536	13.24
x4	94	71	602	11.0	358	470032	10.62	628	10.0	355	476992	10.58
alu2	10	6	743	20.0	363	512256	24.10	668	22.0	341	475136	23.80
9symml	9	1	442	12.0	203	293712	14.90	432	12.0	192	282576	13.90
too_large	38	3	918	11.0	403	589280	13.88	758	12.0	353	507616	13.58

TABLE III

COMPARING `script.rugged` WITH `fsapp` AGAINST `script.rugged` ON INDIVIDUAL EXAMPLES.

selected examples for which there was a significant difference in results are shown. On other examples the results have been identical or close. The top (large) part of the table shows examples for which `fsapp` performed better, while the bottom part - three examples on which the classic script was superior. The columns from left to right shows the name of the example, number of primary inputs and outputs, and then the number of literals in the multi-level network, the depth of the technology-independent network, the gate count, area and delay after technology mapping for the `fsapp` and the classic `rugged`, correspondingly. Note that for some of the examples `rugged` with `fsapp` could get a dramatic improvement (all optimization results are verified by the equivalence checking). E.g., in case of `t481` the `fsapp` achieved more than 5x area reduction with the simultaneous 33% reduction in delay. We could not get anywhere close to this result with other existing `SIS` scripts.

Table IV shows how often local and global restructuring have been used for some of the examples. The top part of the table reports numbers for the local node restructuring. Only examples for which more than ten nodes have been changed by `fsapp` are shown. The top part of the table ends with a line `LTotal` that lists numbers for all the examples. The bottom part of the table reports numbers for global restructuring on some of the example (with more than one node changed) and summarizes overall numbers in the line `GTotal`. The columns from left to right reports the following statistics.

- The number of changed nodes, *Ch*, e.g. `alu4` has 27 nodes changed.
- The number of restructured nodes, *Res*, i.e. changed nodes with new fan-ins due to changes in the support of the node. `alu4` has 19 restructured nodes.
- The number of restructuring inputs, *rIn*, further partitioned into three categories: `SS`, for the same support, `OA` for over- and under-approximations, and `GA` for general approximations. E.g. if restructuring of a node involved the incorporation of three new fan-in nodes, two of which have the subset of the support for the current node, and one is a general approximation, then these new fan-in nodes will be counted in each corresponding category.

	nodes		new inputs				nodes with k new inputs				in TFI?	
example	Ch	Res	rIn	SS	OA	GA	0	1	2	> 2	TFI	nTFI
alu4	27	19	35	11	9	15	8	12	2	5	22	13
apex6	22	9	11	5	0	6	13	7	2	0	4	7
apex7	15	8	8	4	1	3	7	8	0	0	1	7
C1908	11	3	5	2	0	3	8	1	2	0	5	0
C2670	19	2	5	4	0	1	17	0	1	1	3	2
C3540	43	19	25	9	0	16	24	14	4	1	13	12
C432	25	10	16	1	0	15	15	4	6	0	16	0
C5315	27	18	28	15	2	11	9	10	6	2	4	24
comp	16	0	0	0	0	0	16	0	0	0	0	0
frg2	50	32	61	51	1	9	18	4	27	1	9	52
i10	105	43	56	35	5	16	62	32	10	1	28	28
i8	16	11	15	8	2	5	5	7	4	0	6	9
pair	85	24	39	35	1	3	61	10	13	1	9	30
rot	31	15	19	18	0	1	16	12	2	1	13	6
t481	167	12	14	4	5	5	155	10	2	0	4	10
term1	13	3	3	1	1	1	10	3	0	0	1	2
ttt2	16	9	18	10	1	7	7	2	5	2	5	13
x3	23	13	18	11	1	6	10	9	3	1	8	10
x4	12	6	7	2	1	4	6	5	1	0	1	6
LTotal	820	307	465	275	37	153	513	180	102	25	184	281
C3540	2	2	3	1	0	2	0	1	1	0	3	0
C5315	2	2	4	2	0	2	0	0	2	0	3	1
i10	7	5	13	8	1	4	2	1	2	2	11	2
rot	2	2	2	2	0	0	0	2	0	0	1	1
t481	7	6	14	5	0	9	1	2	1	3	12	2
GTotal	30	23	53	27	1	25	7	7	8	8	43	10

TABLE IV

STATISTICS FOR LOCAL (THE TOP PART) AND GLOBAL (THE BOTTOM PART) RESTRUCTURING ON THE SELECTED EXAMPLES.

`alu4` has 35 restructuring inputs, 11 of which are from the same support, 9 are over/under approximations, and 15 are general approximations.

- The next four columns reports statistic relating changes of the nodes with the number of new inputs incorporated: $k = 0$ - how many nodes were changed without incorporating any new input, equivalent to *Ch-Res* (8 for `alu4`), $k = 1$ - number of nodes changed incorporating 1 new input (12), $k = 2$ - number of nodes changed incorporating 2 new inputs (2), and $k > 2$ - number of nodes changed incorporating more than 2 inputs (5).
- The last two columns indicate that for the `alu4` from the 35 new inputs that were incorporated in changed nodes, 22 were in the transitive fanin of a node under minimization (*TFI* column) and 13 were out of the transitive fanin (*nTFI* column).

This table indicates that `fsapp` restructuring may lead to a significant area or/and delay optimization. E.g. for the `t481` global restructuring of 6 nodes and local restructuring of 12 nodes further led to changes in 156 other nodes. We can conclude that restructuring of multi-level networks based on changing node support is a powerful tool for optimizing area and delay. We have demonstrated that function approximations is a good method for selecting candidates for such restructuring. We have explored a few methods for function approximations and plan to do a more careful investigation of the approximation metrics and methods in the future.

VI. CONCLUSIONS

This paper presented a method combining multi-level network simplification based on complete don't cares with restructuring based on support extension using function approximations. The results demonstrates that typically we can get significant area reduction without compromising (or improving) the delay as compared to the known methods of multi-level simplification.

A few avenues can be explored to further improve the results:

- different methods of function approximation,
- using layout-oriented measures for restructuring,
- this approach provides a logic synthesis framework for generating of constructive cyclic circuits,which can improve area or delay. This is an interesting (although not practical since no or little tools in the design flow support combinational cycles) research topic.

REFERENCES

[1] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," U.C. Berkeley, Tech. Rep., May 1992.

[2] S.-C. Chang and M. Marek-Sadowska, "Perturb and simplify: multi-level boolean network optimizer," *IEEE Transactions on Computer-Aided Design*, vol. 15, no. 12, pp. 1494–1504, Dec. 1996.

[3] C. L. Berman and L. H. Trevillyan, "Global flow optimization in automatic logic design," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 5, pp. 557–564, May 1991.

[4] Y. Matsunaga and M. Fujita, "Multi-level logic optimization using binary decision diagrams," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1989, pp. 556–559.

[5] J. Cong, Y. Lin, and W. Long, "SPFD-based global rewiring," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, Feb. 2002, pp. 77–84.

[6] S. Yamashita, H. Sawada, and A. Nagoya, "A new method to express functional permissibilities for LUT-based FPGAs and its applications," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1996, pp. 254–261.

[7] R. Rudell, "Tutorial: Design of a logic synthesis system," in *Proc. ACM/IEEE Design Automation Conference*, 1996, pp. 191–196.

[8] Y.Watanabe and R. Brayton, "Heuristic minimization of multiple-valued relations," *IEEE Transactions on Computer-Aided Design*, vol. 12, no. 10, pp. 1458–1472, October 1993.

[9] S. Sinha and R. K. Brayton, "Implementation and use of spfds," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1998, pp. 103–110.

[10] H. Savoj and R. Brayton, "The use of observability and external don't-cares for the simplification of multi-level networks," in *Proc. ACM/IEEE Design Automation Conference*, 1990, pp. 297–301.

[11] R. Brayton, "Compatible observability don't-cares revisited," in *Proc. International Workshop on Logic Synthesis*, 2001, pp. 121–126.

[12] A. Mishchenko and R. Brayton, "Simplification of non-deterministic multi-valued networks," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 2002, pp. 557–562.

[13] MVSIS Group, "MVSIS," UC Berkeley, www-cad.eecs.berkeley.edu/mvsis.

[14] A. Mishchenko, "An experimental evaluation of algorithms for computation of internal don't-cares in Boolean networks," Portland State University, Dept. ECE, Tech. Rep., Sept. 2001, www.ee.pdx.edu/~alanmi/research/net/DCcomparison.pdf.

[15] H. Savoj, "Don't cares in multi-level network optimization," Ph.D. dissertation, UC Berkeley, May 1992.

[16] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[17] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovannni-Vincentelli, "State enumeration of finite state machines using BDDs," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1990, pp. 130–133.

[18] IWLS'91 benchmark set, `http://www.cbl.ncsu.edu/CBL_Docs/lgs91.html`.