Recursive Decomposition of Sparse Incompletely-Specified Functions

Robert Brayton and Alan Mishchenko
EECS Dept.
UC Berkeley

Abstract

A sparse function is one with only a few onset and offset minterms, compared to the entire input space. The paper examines effective use of the don't cares to synthesize a small logic network. An algorithm is proposed, implemented and tested on known functions where only a small sample of care minterms is given. The algorithm is top-down recursive looking for decompositions based on two-input Boolean functions and multiplexors.

I. Introduction

We examine the problem of implementing a multi-level multi-output logic function in the form of a completely specified Boolean network, when only a small sample of its onset and offset minterms is given. The goal is to find a relatively small network that distinguishes between the given onset and offset minterms. Minterms of the Boolean input space outside of the sample set, are treated as don't cares. The challenge of sparse functions is that there is an enormous number of don't cares. Examples of applications are in the domain of character recognition, image reconstruction and, in general, learning the smallest logic function consistent with a given training set.

Occam's Razor Principle states that "the simpler explanations for available observed data have the greater predictive power", so it serves as a motivation for finding a simple network. In this work, a simpler explanation is the smaller Boolean network that distinguishes the provided data. In another sense, we want to generalize from a given set of examples. Since this problem is known to be NP-hard [Gary and Johnson'79], our approach is heuristic. In addition, we restrict our selection of the nodes to two-input functions and multiplexors, which can be further optimized by standard logic synthesis tools.

We develop an algorithm which produces such a network and measure its efficacy for the case when one known good network is available. In our experiments, completely-specified Boolean networks are sampled for a given percentage of their onset and offset minterms. This training set is thus given as two sets of minterms for each output, i.e. we are given a number of incompletely-specified Boolean functions (ISFs) $\{(f_i, g_i)\}$, one for each output. In this work, (f_i, g_i) is given as a pair of truth tables. Our algorithm produces a generalization network, which for each output *i*, $h_i(x)$ as a function of the inputs *x*, satisfies $f_i(x) \subseteq h_i(x) \subseteq \overline{g}_i(x)$. This is compared to the original completely-specified network after both have been synthesized by ABC. The comparison is in terms of the number of AIG nodes.

A viable algorithm should produce a AIG count at least as good as the "golden model", because the golden model gives the existence of at least one circuit with that count, but the don't cares provide additional flexibility for circuit generalization. An example is the *alu4* benchmark from *mcnc91*, which has 14 inputs and 8 outputs. If the initial completely-specified circuit is optimized by iterating the ABC command *dc2*, it can be reduced to 631 AIG nodes. On the other hand, our algorithm produces a circuit, based on a 10% sampling of the onset and offset minterms of the 8 outputs, which has only 429 nodes after iterative synthesis with command *dc2* in ABC.

The problem of synthesizing incompletely-specified functions with many don't-care has been the topic of previous research. In particular, [Chang et al '10] proposes an elaborate portfolio solution and an in-depth discussion of industrial applications.

II. The Algorithm

Our algorithm decomposes a sparse ISF into a network of two-input ANDs, ORs, XORs, and MUXes. At each step it tries to use the don't cares in a helpful way but at the same time preserving as many as possible don't cares for later use in the recursion. It can also choose to implement the complement if it is estimated to give better results. The input is an ISF for each output given as a pair (ON, OFF) of truth tables or sets of minterms. Since the input is specified using a set of minterms, we restrict the number of inputs to 16 in our current implementation, which was done in Python. The pseudo-code of the algorithm is given in Fig. 1.

```
decomp_network({(f_i,g_i)})
   Net = []
   for each i:
          R = decomp(f_i,g_i)
          Verify(R,(f,g))
          Net = Net + R
   return dc2(Net)

decomp (f,g):
   a.  fm = min(f,g)
   b.  gm = min(g,f)
   c.  sign =  |fm| <= |gm|:
   d.  if not sign:
          f,g) = (g,f)
   e.  v_m = choose_var_cof(f,g)
   f.  v_x = choose_lit_xor(f,g)
   g.  mx = estimate_size('mux',v_m,(f,g))
   h.  ex = estimate_size('xor',v_x,(f,g))
   i.  fx = estimate_size('factor',f)
   j.  decomp_type =
       select(mx,ex,fx,('mux','xor','factor'))
   k.  if decomp_type == 'mux':
          x = v_m

          N = (x decomp(f_x,g_x) +
               xbar decomp(f_xbar,g_xbar))
    l.  if decomp == 'xor':
          x = v_x
          N = (x xor decomp(xor (x,
               (f,g))
    m.  else:
        if sign:
          N = fm
        else:
          N = gm
    n.  if sign:
          return N
        else:
          return ~N
```

Figure 1: Recursive function to decompose an ISF.

The algorithm *decomp_network* works top-down recursively calling *decomp*(f_i, g_i) to obtain a decomposition tree for each output given the ISF (f_i, g_i). These are accumulated into a single multi-output network Net.

At each call to *decomp*, there is given a pair (f, g) as either a pair of minterm sets or a pair of truth-tables. The first step is to estimate if it would be better to implement the onset or the offset (lines a-d), so at each step we can implement either a function (sign = '+', see Fig. 2 below) or its complement (sign = '-'). This is done by quickly estimating the decomposition of (f, g) and (g, f) separately and choosing the "simplest". Then it is tested if a MUX/XOR at the top could lead to a better decomposition, or a simple algebraic factoring (lines e-j). For the first two, we need to select a best variable (pivot). For a MUX the ISF is $(f, g) = x(f_x, g_x) + \overline{x}(f_{\overline{x}}, g_{\overline{x}})$ so we recur by calling decomp on two ISFs, (f_x, g_x) and $(f_{\overline{x}}, g_{\overline{x}})$. For XOR, we use the fact that $(f, g) = x \oplus x \oplus (f, g) = x \oplus (x(g, f) + \overline{x}(f, g))$, so we recur by calling decomp on the single ISF $(xg + \overline{x}f, xf + \overline{x}g)$, using the fact that $\overline{(f, g)} = (g, f)$ for an ISF.

For making these choices, we need to estimate, which of the inputs would be the best variable for co-factoring or for XORing. As a measure of which to choose, each variable *x* is tested by measuring how well the SOPs created by the associated decomposition, can be factored if *x* is chosen. The SOPs are produced using Espresso minimizer or a recursive algorithm for Irredundant Sum-Of-Product (ISOP) computation [Minato'00]. Each of these algorithms is applied to (f_x, g_x) and $(f_{\overline{x}}, g_{\overline{x}})$ in the case of a MUX or $(xg + \overline{x}f, xf + \overline{x}g)$ in the case of an XOR, and the best result is chosen (lines e-f) and compared against a simple algebraic factoring of the SOP for the given ISF (line j). If the multiplexor wins, we recur by applying decomp to (f_x, g_x) as well as $(f_{\overline{x}}, g_{\overline{x}})$ (line k). If the XOR wins we recur by applying decomp to $(x \oplus (f, g))$ (line *l*). Otherwise, the recursion terminates by returning the factoring of the minimized (f, g) (line m). Finally the variable sign is used to complement the resulting network or not (line n).

Minimization of an ISF is done with both Espresso and ISOP, both of which can use don't cares. This depends on whether the minimization is being used just for estimation (ISOP only is used for speed), e.g. in choosing the pivot variable *x* for the multiplexor or XOR, or for choosing whether to implement the complement of the ISF (both ISOP and Espresso are used to get a better estimation).

As an example of the kind of decomposition possible, we consider the benchmark *alu4* in the *mcnc91* benchmark set (which has 14 inputs and 8 outputs) and consider part of the decomposition tree of the second output, which is PO = 1. The decomposition of that part is printed by the algorithm as follows:

```
[[-13, 'xor', '+', '+'],
   [[9, 'shannon', '+'],
    ['+', ['-------------0', '-----------0-',
           '----------0-1-']],
    ['-', ['----------1-1-', '0-------1-----',
           '----1-----0---']]]]]
```

Figure 2: A sample of part of a decomposition produced by the algorithm.

This represents the function $\overline{x}_{13} \oplus [x_9(\overline{x}_{13} + \overline{x}_{11} + \overline{x}_{10}x_{12}) + \overline{x}_9\overline{(x_{10}x_{12} + \overline{x}_0x_8 + x_4\overline{x}_{12})}]$. The example illustrates choosing an XOR decomposition at the top level using the pivot literal $\overline{x}_{13}$ (`-13`). The initial ISF (f, g) is not complemented and the sub-call to decompose $(xg + \overline{x}f, xf + \overline{x}g)$ is also not complemented ('+','+'). The recursion on the ISF $\overline{x}_{13} \oplus (f, g)$ chooses a Shannon decomposition with the co-factoring variable x_9 and chooses to implement the ISF as un-complemented (`[9, 'shannon', '+']`). The first cofactor is chosen to be un-complemented (`'+'`), while the second is complemented (`'-'`). Finally the algorithm terminated by estimating that each of the "shannon" cofactors are best implemented as a factored form.

As a final step of the loop in `decomp_network`, the circuit for the i^{th} output is (1) implemented as an AIG, `R`, (2) verified to be correct i.e. $f \subseteq R \subseteq \overline{g}$, and (3) added to `Net` as another output of `Net`. Finally, `Net` is minimized by iterating the *abc* command *dc2* (`dc2(Net)`).

The Python code for ISOP of an ISF is given in Figure 3. The ISF is given as a lower bound truth table (L) and an upper bound truth table (U).

```
def isop(self, L, U, i):
        if L.is_contradiction():
            return ([], L) # return False
        if U.is_tautology():
            return ([set([])], U) # return True
        x = min( L.min_variable(i), U.min_variable(i) )
            # next dependent variable after i
        fx = self.var(x, 1) # truth table for variable x
        (L0,L1) = L.cofactors(x) #ISFs for cofs of onset
        (U0,U1) = U.cofactors(x) #ISFs for cofs of offset
        (c0, f0) = self.isop(L0 & ~U1, U0, x+1)
        (c1, f1) = self.isop(L1 & ~U0, U1, x+1)
        Lnew = L0 & ~f0 | L1 & ~f1
        (cstar, fstar) = self.isop(Lnew, U0&U1, x+1 )
        cres =  [c.union(set([x+1])) for c in c0] +
             [c.union(set([-(x+1)])) for c in c1] + cstar
        fres = f0&fx | f1&~fx | fstar
        return ( cres, fres )
```

Figure 3. Python code for ISOP.

III. Related Work

In general, if we decompose with a two-input function at the top, e.g. an AND, the two inputs provide don't cares for each other. For an AND, when one input is 0, the other is a don't care and vice versa. In general, this gives rise to a Boolean relation. For example for an AND, if the output is to implement an ISF (f,g) = (onset, offset), and the two inputs of the AND are (u,v), then the Boolean relation is $fuv + g(\overline{u} + \overline{v}) + \overline{fg}$. In tabular form, the Boolean relations for AND, XOR, and MUX look as shown below.

Input minterms	AND (u,v)	XOR (u,v)	MUX (u,v,w)
onset	(1,1)	(0,1), (1,0)	(1,1,-), (0,-,1)
offset	(0,-), (-,0)	(0,0), (1,1)	(1,0,-), (0,-,0)
don't care	(-,-)	(-,-)	(-,-,-)

Thus for the AND and for any offset minterm, we have a choice to have (u,v) = (0,0) or (0,1) or (1,0) but not (1,1). Finding a minimum implementation for the *AND*(u,v), *XOR*(u,v) or *MUX*(u,v,w) requires a Boolean relation minimizer, such as *BREL* [Bañeres et al '04] (see also [Bernasconi et al '15]), which seems to be the best available. In the present paper to avoid Boolean relation minimization, we opted for a decomposition where one of the inputs was a primary input variable, called the pivot variable.

Decomposition from the bottom up can be done also. In [Kravets and Sakallah'98] a bottom up method is proposed which directly decomposes a Boolean network into a set of gates from a library. This combines the technology -independent and -dependent steps. At each step a new node is created using a library element and current inputs. At subsequent steps, these new nodes are treated as inputs and can be fanins to the next node created, using Boolean division. However, if there were any external don't cares initially, they are also all consumed by creating the initial Boolean network. A related method used for restructuring the network is given in [Kravets and Kudva'04].

Another work is on bi-decomposition [Kravets and Mishchenko'09]. It is applicable to ISFs. At the top, a choice of either a two-input AND or two-input XOR is

made for the decomposition (other two-input functions are obtained by inversion and deMorgan). Then, bi-decomposition tries to minimize the set of variables, which are in the overlap of support dependencies between the two fanins; the remaining variables are in the support of only one fanin or the other, but not both. This choice is balanced by wanting the sides to have reasonably the same complexity. Once this choice of variable separation is made, some don't cares are used up but the remaining don't cares are propagated to one fanin and it is again bi-decomposed. At each node in the recursion, after one fanin is implemented as a CSF, it furnishes additional don't cares which can be added to the initial don't cares given to the remaining fanin during variable separation. Thus its strategy is to use the don't cares to separate the variable dependencies on each fanin before recurring on one of the fanins.

The *BREL* minimizer [Bañeres et al '04] can be applied to a top-level decomposition method. This has some similarity with bi-decomposition, in that after a top-level Boolean function is chosen (but not limited to only two fanins), it derives a corresponding Boolean relation BR_0, whose outputs are the respective fanins. Then it focuses on each fanin separately by projecting BR_0 onto that fanin. This gives an ISF for that fanin which is minimized first using don't cares to reduce the number of dependent variables and then using ISOP. These functions, one for each fanin, are then composed into a "functional" Boolean relation (composed only of functions), BR_f, for which a cost is computed. This is a lower bound, *b*, on the cost of any final implementation. If BR_f is compatible with BR_0 then this is returned; else an incompatible minterm/fanin pair (*m,u*) is chosen and BR_0 is split into two BRs: BR_1, where $u = 0$ for *m* and BR_2, where $u = 1$ for *m*. These are explored separately. This splitting is done recursively. In the end, the algorithm returns a set of functional BRs, and any one of them can be chosen as the final implementation depending on a given cost function. *BREL* uses branch-and-bound using the bound, *b*, to restrict the number of branches visited. Other heuristics are used to limit the number of functional BRs returned.

Another approach uses Occam's razor as a motivation to find a Boolean function to do image reconstruction and hand written character recognition. It proposes two approaches, one that uses a gradient method to construct a network in a bottom up manner; the other uses a decision tree approach. (See [Oliveira et al '93] for more details.)

In contrast to the previous works, our algorithm is restricted to choosing the top-level Boolean function at each step to have one of its fanins to be a primary input. If such a choice seems to be worse than minimizing SOP and then factoring, the recursion is stopped and the factoring solution is returned. In doing this, most of the initial don't cares are reserved for the bottom-up phase, i.e. the SOP minimization and factoring. This is sub-optimal, because the don't cares are all used up by the SOP minimizer and none are left for the factoring.

IV. An SPFD Approach

Another approach for recursive decomposition of ISFs, which as far as we know has not been tried, is based on SPFDs [Yamashita et al '00]. Consider an ISF $f = (f_{on}, f_{off})$ and a three-level structure with a binary operation, *op*, on top. This structure implements a cover, $c = f_1\ op\ f_2$, of f (denoted $c>f$) where f_1 and f_2 are SOPs. Given a cost function based on, say, number of BDD nodes, number of factored-form literals, etc., the problem is to find an *op* and a pair of SOPs (f_1, f_2) such that cost(f_1, f_2) is minimized.

A general way to do this, is to partition f_{on} into two parts, (*A, B*), and f_{off} into two parts, (*C, D*). The sets of SPFD edges to be distinguished are (*A*-*C*), (*A*-*D*), (*B*-*C*), and (*B*-*D*). SPFD theory states that if two functions f_1 and f_2 cumulatively distinguish all these edges, then there exists an *op* such that $c = f_1\ op\ f_2 > (f_{on}, f_{off})$.

This can be done in three ways:

$$[f_1, f_2] = \begin{matrix} [\tilde{f}_{A(CD)}, \tilde{f}_{B(CD)}] \\ [\tilde{f}_{(AB)C}, \tilde{f}_{(AB)D}] \\ [\tilde{f}_{(AD),(BC)}, \tilde{f}_{(AC)(BD)}] \end{matrix}$$

where ~ means taking the function or its "complement" and $f_{A(CD)}$ denotes a cover for $(A, C+D)$, i.e. $f_{A(CD)} > (A, C+D)$ and its "complement" would be $f_{(CD)A} > (C+D, A)$. One can verify that, if the inputs to *op* are *u* and *v*, then the *op* is $\overline{uv}$ or $\overline{u}+\overline{v}$ or $uv+\overline{uv}$ respectively for the three brackets in the case when none of the ~ functions are complemented.

As an example, consider the third bracket and the pair $g = [f_{(AD),(BC)}, f_{(BD)(AC)}]$ Then $g(a) = (1,0)$, $g(b) = (0,1)$, $g(c) = (0,0)$ and $g(d) = (1,1)$, where $a \in A, b \in B, c \in C, d \in D$. Note that minterms of A are distinguished from C and D and minterms of B are distinguished from C and D as required. Since we need the output of *op* to be 1 for A and B, $op = u \oplus v$.

As far as we can determine given partitionings of the onset and offset, these are the only ways of implementing a pair of functions such that $f_1\ op\ f_2 > (on, off)$.

Thus, there are many ways to implement $c = f_1\ op\ f_2 > f$. In the bracketed pairs above, there are four choices within each bracket, and there are three brackets, leading to a total of 12 choices of pairs to implement. This corresponds to the total of 12 completely-specified Boolean functions that depend on exactly two variables. Each choice will determine what is implemented in the AND plane of the PLA and the amount of sharing that can be achieved between f_1 and f_2. This is in addition to the flexibility offered by the many ways of partitioning the onset and offset.

Note this is different than using Boolean relations when we first choose *op*, and then derive a relation to be minimized.

This approach is also different from bi-decomposition [Kravets and Mishchenko'09] when we search among a set of binary operators for the top level *op*. For example, suppose the ISF to be covered is given as (*on, u*), instead of (*on, off*), where *u* stands for the upper bound of the ISF, $u = \overline{off}$. Then for example the *OR* operator, we look for a partition of the inputs $x = (x_a, x_b, x_c)$ and solve for ISFs where $f_1 > (\exists_{x_a} on, \forall_{x_a} u)$ and $f_2 > (\exists_{x_c} on, \forall_{x_c} u)$. The partition of *x* is made so that f_1 and f_2 depend on as few of the inputs as possible, i.e. $supp(f_1) + supp(f_2)$ is minimized. For *op = OR,* we need to make sure that the interval $(on, (\forall_{x_a} u + \forall_{x_c} u))$ is not empty. A more complicated condition is stated if *op = XOR* and can be seen as an SPFD-type condition, namely, in order for $f_1\ XOR\ f_2 > (on, u)$ to be possible, it must be that all care minterms that cannot be distinguished by f_1 must be distinguished by f_2. Thus, we see that the bi-decomposition strategy is to partition the inputs optimally first, but not fix a partitioning of the onset (or offset). In the SPFD approach, we bi-partition the onset (or offset), determine a best SOP pair using the 12 choices and then let these determine the operator *op*.

The SPFD approach has a larger search space because, for most functions, there are many more ways of partitioning the onset and offset minterms, compared to partitioning variables. However, it is not clear how this larger space can be efficiently searched. One approach might be to put minterms that are closer into the same part. Future work may concentrate on finding good heuristics to leverage the generality of the SPFD approach and produce better quality of decompositions.

V. Experimental Results

Table 1 shows some experimental results. The first column contains the name of the benchmark, the second lists the number of inputs, outputs and `aig` nodes in the initial description. The third column gives the number of `aig` nodes after iterating synthesis with command `dc2` in ABC until no change, and the fourth column gives the number of `aig` nodes in the decomposed network before and after synthesis. Sampling density was 10% for all benchmarks.

Table 1: Quality comparison for initial version vs decomposed version of MCNC benchmarks.

Name	ins/outs/aig	init aig	dec aig	time, s
9symml	9/1/211	186	10/9	0.7
alu2	10/6/401	346	76/68	5.0
alu4	14//8/735	617	646/484	53.8
cm163a	16/5/36	31	9/9	4.7
cmb	16/4/47	37	21/14	6.0
cu	14/11/55	38	21/15	5.8
f51m	8/8/139	96	20/19	2.7
parity	16/1/45	45	34/34	1164.0
pml	16/13/47	30	11/8	7.7
t481	16/1/1874	159	830/556	139.0
test	12/1/590	436	70/53	3.8
x2	10/7/54	38	28/22	2.9
z4ml	7/4/47	24	9/8	1.2

Observations:

1. Most of the sampled decompositions were small if the original was small. The exception was t481 where the original was large (1874) but synthesized impressively small (159) while the decomposition of the sampled function did not synthesize as well (830 -> 556). This is because t481 has a simple disjoint-support decomposition structure found by synthesizing the original.

2. The two examples with 16 inputs take longer to find good pivot variables.
3. In some cases, sample size of 10% minterms was not enough to sufficiently distinguish onset from offset minterms. It seemed in some cases, an output had minterms in only the onset or the offset, so the minimization resulted in a constant. Better sampling strategies should be tried.
4. All the runtimes were reasonable except for parity. However, the algorithm did discover that XOR decomposition was effective (see Fig. 4 below). Note that with XOR there are three opportunities to implement the complement; use the complemented literal, complement the initial ISF, and complement the argument of the recursive call to decomp.

```
Parity output:
[[[2, 'xor', '+', '+'],
   [[0, 'xor', '+', '-'],
    [[1, 'xor', '+', '-'],
     [[3, 'xor', '+', '-'],
      [[4, 'xor', '+', '-'],
       [[5, 'xor', '+', '-'],
        [[-15, 'xor', '+', '-'],
         [[-6, 'xor', '-', '-'],
          [[-7, 'xor', '-', '-'],
           [[8, 'xor', '-', '-'],
            [[9, 'xor', '+', '-'],
             ['-', ['-----------0----',
                    '----------0-----']
                                  ]]]]]]]]]]]]]
```

Figure 4: Printout of algorithm for sampled 16-input parity function.

VI. Discussion and Conclusions

Other decompositions were tried where one of the inputs was a primary input, in particular, the positive and negative Davio expansions, but these tended to use up the don't cares rather quickly.

The positive Davio expansion is $f = f_{\bar{x}} \oplus x(f_x \oplus f_{\bar{x}})$. For an ISF, this yields

$$(f,g) = (f_{\bar{x}}, g_{\bar{x}}) \oplus x((f_x, g_x) \oplus (f_{\bar{x}}, g_{\bar{x}})).$$

However the XORs tend to squander don't cares. So instead of using XOR decomposition at the output, XORing of an input variable with an ISF was used, $(f,g) = x \oplus (x \oplus (f,g))$.

In our implementations, we did not use Espresso, but instead wrote some Python code to mimic the algorithms of Espresso (expand, irredundant, essential, reduce etc). We also used a Python package `pyaig` which includes procedures `truth_tables`, developed by Baruch Sterin, and allows for fast manipulation of truth tables, such as cofactoring, comparison, Boolean operations, and conversion to SOP using the ISOP algorithm. For creating a network, we used a Python package `pyzz`, developed by Niklas Een and Baruch Sterin, which can manipulate Boolean functions to create a new internal variable, add it to a network, and finally write the network out as an AIG.

As mentioned, our implementation is limited to no more than 16 inputs. Therefore, the number of non-trivial examples we were able to study was limited. Clearly, a more efficient implementation can be done, for example by using the real Espresso package, which would overcome the 16 inputs limitation. Most of the run-time is spent on estimating which variable to choose as the pivot variable. This is done in the current implementation by enumerating over all variables. Therefore the implementation does not scale well right now, although optimizations were done to mitigate some of these limitations. Obviously, much of the algorithm can be implemented using concurrency.

A slightly different experiment would be to sample all the outputs at once so that the set of minterms occurring for each output is the same, i.e. the ISFs only differ in which minterms are in the onset and which are in the offset. This would better emulate an application where a single sample provides values for all outputs at once, as would be the case in character recognition where a minterm is a character and a sampling produces a set of values of characteristics for that character.

Another point is that common divisor sharing among different outputs is done only after the decomposition for each output is completed, by using `dc2` and `fx` on the final multi-output network. Another approach would be to convert the output ISFs to a Boolean relation [Bernasconi et al '15] and use *BREL* [Bañeres et al '04]. Some of the ideas used in bi-decomposition should be tried, as well as the SPFD approach as outlined in Sec. IV.

Nevertheless, this research shows that it is possible to find a reasonably small implementation by sampling a small network sparsely. Pending sufficient motivation and practical examples, a more efficient implementation can be developed. This might include trying many approaches in parallel and choosing the best one at each stage. The same can be said for finding the best pivot variable.

References

[Bañeres et al ’04] D. Bañeres, J. Cortadella, and M. Kishinevsky, “A recursive paradigm to solve Boolean relations”, Proc. DAC’04, pp. 416-421.

[Bernasconi et al ’15] A. Bernasconi, R. Brayton, V. Ciriani, T. Villa, “Complemented circuits”, submitted to IWLS’15.

[Chang et al ‘10] K.-H. Chang, V. Bertacco, I. L. Markov and A. Mishchenko, ``Logic synthesis and circuit customization using extensive external don't-cares" ACM Trans. on Design Autom. Elec. Sys (TODAES), 15 (3), May 2010.

[Garey and Johnson ‘79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979, ISBN 0-7167-1045-5.

[Kravets and Kudva’04] V. N. Kravets and P. Kudva, “Implicit enumeration of structural changes in circuit optimization”, Proc. DAC’04, pp. 438-441.

[Kravets and Mishchenko’09] V. N. Kravets and A. Mishchenko, “Sequential logic synthesis using symbolic bi-decomposition”, Proc. DATE’09, pp. 1458-1463.

[Kravets and Sakallah’98] V. N. Cravats and K. A. Sakallah, “M32: A constructive multilevel logic synthesis system”, Proc. DAC’98, pp. 336-341.

[Minato’92] S. Minato, “Fast generation of irredundant sum-of-products forms from binary decision diagrams. Proc. SASIMI'92, pp. 64-73.

[Oliveira et al ’93] A. Oliveira and A. Sangiovanni-Vincentelli, “Learning complex Boolean functions: Algorithms and applications”, in J. D. Cowan, G. Tesauro, J. Alspector, ed., *NIPS*, Morgan Kaufmann, pp. 911-918.

[Yamashita, et al ’00] S. Yamashita, H. Sawada, and A. Nagoya, "SPFD: A new method to express functional flexibility," IEEE TCAD, vol. 19, no. 8, pp. 840-849, Aug. 2000.