Finding complex disjunctive decompositions of logic functions.

Maurizio Damiani
Synopsys, Inc.
Mountain View, CA 94043
maurizio@synopsys.com

Valeria Bertacco
Computer Systems Lab.
Stanford University
valeria@stanford.edu

Abstract

This paper extends the previous work [1] on simple disjunctive decompositions to the important case of complex disjunctive decompositions. We pinpoint the sources of additional complexity of handling this decomposition style, and relate it to the existence of encoding don't care *conditions [2]. We define two classes of complex decompositions,* **almost-** *and* **perfect decompositions***, respectively. For these decompositions, the decomposition problem can be separated from the encoding problem. From a theoretical standpoint, we prove that a logic function has a* **unique** *perfect decomposition. For a given function, both types of decomposition can be found by an efficient decomposition algorithm, that contains the one presented in [1] as special case. Preliminary experimental results are extremely encouraging.*

1 Introduction.

The decomposition of a logic function F is the process of passing from a lumped representation of F (such as a BDD [3, 4] or a cover) to a multiple-level representation [5, 6, 7, 8, 9, 10].

A *disjunctive* decomposition seeks explicitly a tree-like multiple-level representation, so that no two logic blocks (the tree nodes) share inputs. Disjunctive decompositions are desirable because input wires are clustered into smaller blocks, thereby generally resulting in simpler routing. Moreover, the designer (or design tools) can focus optimization on smaller individual blocks. Disjunctive decompositions are further classified into *simple* and *complex*. In simple disjunctive decompositions, each logic block appearing in the tree is a single-output function. In complex decompositions, instead, each logic block can output multiple functions.

Fig. (1) exemplifies simple and complex decomposition topologies for a function. $k_1, k_2, \cdots k_m$ denote bus widths. The functions A_i are single-output functions for simple decomposition, and multiple-output blocks for complex decompositions.

Because of their importance, procedures for decomposing logic functions have been a central subject of research in CAD and switching theory. Factorization can be regarded as a special case of decomposition [11], and it represents the cornerstone of algebraic multiple-level logic optimization techniques [12].

Recently, an algorithm was presented in [1] for deriving efficiently a simple disjunctive decomposition of a function from a BDD representation. The exactness and efficiency of the procedure stem from several desirable properties of the BDD representation as well as from the uniqueness of the

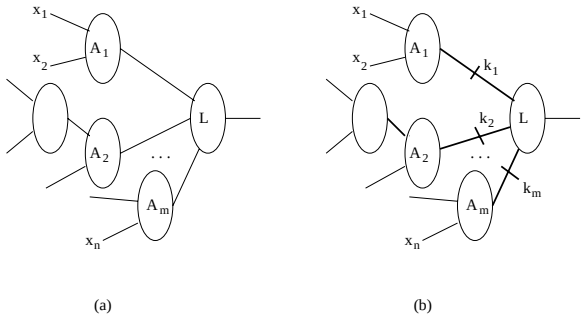

Figure 1. Simple disjunctive decomposition (a) and complex disjunctive decomposition (b) tree structures

decomposition. For a n-input function F, the complexity is $O(n|V|^2)$, where $|V|$ denotes the number of nodes in the BDD representation of F.

Complex decompositions are clearly more attractive than simple decompositions. Yet, as will be illustrated later, the number and nature of the degrees of freedom available for decomposing a function make it very difficult to find a general, efficient, and exact decomposition technique.

In this paper we consider two special classes of complex decompositions, namely, **almost-perfect** and **perfect** . We show that perfect decompositions have all the desirable properties of simple decompositions. In particular, we prove that a function has a **unique** perfect decomposition. We show that this decomposition can be found efficiently. Moreover, it is possible to provide an accurate upper bound on the complexity of the algorithm, and trade off CPU time for quality of results.

The rest of the paper is organized as follows. First, we consider the problem of finding general complex decompositions, and point out the sources of difficulty. We then introduce the two new decomposition classes, and prove the uniqueness result. Section (4) contains a description of the decomposition algorithm. Eventually, we present experimental results.

2 Classification of decompositions.

This section introduces the basic terminology.
We denote by $\mathcal{B}$ the Boolean set $\{0, 1\}$.

Definition 1. *A* **disjunctive decomposition** *of a n-input function F is defined as follows:*

1. *a* **root** *function $R(x_1, \cdots, x_m)$; the inputs x_i are termed* formal inputs *of R;*
2. *a list of disjoint-support multiple output functions $\mathbf{A}_i : \mathcal{B}^{n_i} \rightarrow \mathcal{B}_i^k$, termed* **input functions.**. *The quantities*

k_i, n_i have the property:

$$\sum_i k_i = m; \quad \sum_i n_i = n \tag{1}$$

3. *a* **port binding** *from the outputs of each component of $\mathbf{A}_i$ to the formal inputs of R : $\mathbf{A}_i \Rightarrow (x_{i_1}, \cdots, x_{i_{k_i}})$.*

□

In practice, it is convenient to assume that the inputs of F are given a total order, for example the BDD order. The functions $\mathbf{A}_i$ can then be listed according to the relative ranking of their top variable, and a default port binding can be assumed. A synthetic representation of a decomposition of F can then be $(R; \mathbf{A}_1, \cdots, \mathbf{A}_l)$.

Fig. (1.b) shows pictorially the type of disjunctive decompositions considered here. Simple disjunctive decompositions are defined by $k_i = 1; i = 1, \cdots$. A decomposition is termed **complex** if $k_i > 1$ for some i.

Intuitively, the space of complex decompositions for a function is much larger than that of simple decompositions, and probably far less structured. Indeed, the following example shows that a function may have many "essentially different " complex decompositions:

Example 1. Consider a function F, with decomposition

$$R = MAJORITY(x_1, x_2, x_3); \tag{2}$$
$$x_1 = a \oplus b; \tag{3}$$
$$x_2 = bc; \tag{4}$$
$$x_3 = d; \tag{5}$$

It can be cast in the framework of Definition (1) by taking $R = MAJORITY(x_1, x_2, x_3)$; $\mathbf{A}_1 = (a \oplus b, bc)$, and $\mathbf{A}_2 = d$. Since b is shared by two outputs, it is a complex decomposition.

The function $\mathbf{A}_1$ can assert all four possible combinations $00, 01, 10, 11$. On the other hand, because of the symmetries of $MAJORITY$, for $\mathbf{A}_1$ the output combination 01 is equivalent to 10. Therefore, $\mathbf{A}_1$ could be replaced by a different function $\mathbf{AA}_1$, for instance producing only combinations $00, 01, 11$. □

In the above example, we were able to change "significantly" the behavior of $\mathbf{A}_1$ because of the equivalences of the function $MAJORITY$. The function $MAJORITY$, however, may now be modified as well:

Example 2. Consider the new decomposition, obtained after replacing $\mathbf{A}_1$ with $\mathbf{AA}_1$. The new function $\mathbf{AA}_1$ cannot assert the output combination 10. Therefore, the input combinations $10-$ are impossible at the input of $MAJORITY$, and correspond to input don't cares for it. The function $MAJORITY(x_1, x_2, x_3)$ could then be replaced by a different function. For instance, if the output corresponding to the don't care entries is set to 1, then $MAJORITY$ can be replaced by $AO(x_1, x_2, x_3) = x_1 + x_2x_3$. □

A root of the multiplicity of decompositions in the above examples can be traced to the fact that $MAJORITY$ is a "lossy" function. We now show **why** this multiplicity makes it difficult to decompose a function from its BDD representation.

Example 3. Consider the function F, decomposed as :

$$R = MAJORITY(x_1, x_2, x_3); \tag{6}$$
$$(x_1, x2) = (a \oplus b, b \oplus c); \tag{7}$$
$$x_3 = d; \tag{8}$$

Suppose the root variable of F is a. The two cofactors of A would have decomposition :

$$R = MAJORITY(x_1, x_2, x_3); \tag{9}$$
$$(x_1, x_2) = (b, b \oplus c); \tag{10}$$
$$x_3 = d; \tag{11}$$

and

$$R = MAJORITY(x_1, x_2, x_3); \tag{12}$$
$$(x_1, x_2) = (b', b \oplus c); \tag{13}$$
$$x_3 = d; \tag{14}$$

respectively. A local search algorithm could recognize the topological similarities of the two cofactors and infer a decomposition from F from these similarities. This is precisely what happens in the case of simple decompositions [1]. In this case, however, $MAJORITY$ may be replaced by AO in one of the cofactors, or the input functions somehow altered. This makes comparisons substantially more difficult. □

3 (Almost) Perfect decompositions.

We now introduce perfect functions and decompositions, and show their univocity.

3.1 Almost perfect (AP) decompositions.

The notion of AP decompositions stems directly from the examples in Section (2). We want to avoid decompositions where the root function has "equivalences" and where the input functions are allowed not to assert all possible output values. We formalize this notion of "equivalences" as follows.

Definition 2. *Consider a decomposition $R; \mathbf{H}_1, \cdots, \mathbf{H}_h$. Let $x_1, \cdots, x_m$ denote the formal inputs linked to, say, $\mathbf{H}_1$. We say that the root function R is a* **perfect root** *if each of the 2^m partial assignments $(x_1^*, \cdots, x_m^*)$ of $x_1, \cdots, x_m$, results in a distinct cofactor $R(x_1^*, x_2^*, \cdots . x_m^*, x_{m+1}, \cdots)$.* □

Definition 3. *A n-input, k-output function $\mathbf{A}$ is termed* **full-range** *(FR) if it asserts all possible 2^k output configurations.* □

Definition 4. *A decomposition $R; \mathbf{H}_1, \cdots, \mathbf{H}_h$ is termed* **almost-perfect** *if R is perfect and all functions $\mathbf{H}_i$ are FR.* □

Notice that the decompositions of Examples (1-2) are not almost perfect, because the root function $MAJORITY$ is not a perfect root. This resulted in us being able to replace a FR function $(\mathbf{A}_1)$ with a non-FR function $(\mathbf{AA}_1)$.

Although almost-perfect decompositions allow us to get rid of most of the ambiguities, it may be difficult to obtain them by "local" algorithms. The reason is exemplified below:

Example 4. Consider the function $F = (xy + z)w' + (x + yz)w$. It can be decomposed as $F = MUX(a_0, a_1, w)$; $a_0 = xy + z; a_1 = x + yz$. Fig. (2.a) shows the circuit and the underlying decomposition. The BDD of the function MUX is shown in Fig. (2.b). It indicates that the four

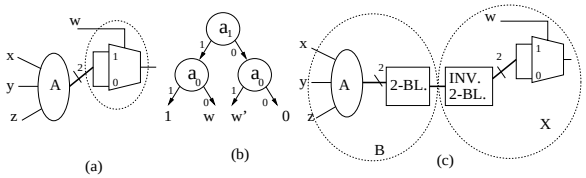

Figure 2. BDD and decomposition for the function of Example (4)

cofactors with respect to a_0, a_1 are distinct,hence the root is perfect. Moreover, the two functions a_0, a_1 can assert all possible output combinations, and therefore the decomposition is AP . Suppose, however, that x is the top variable of the BDD of F, and consider $F(x = 0)$. It can be decomposed as $MUX(z, yz, w)$. On the other hand, the two-output function (z, yz) is not full-range. Hence the decomposition could not have been found on a local paradigm. □

The example indicates that a AP decomposition may be derived by non-AP decompositions at the cofactors. If we discard non-AP decompositions at the cofactors, we will be unable to discover the AP one at the root. This motivates the following definitions.

3.2 Perfect functions.

The definition of perfect functions is recursive:

Definition 5.

1. *a k-bit constant is a perfect function;*
2. *A n-input, k-output function $\mathbf{P}$ is* **perfect** *if :*

(a) *it can assert* **exactly** 2^m *output combinations, for some $m, 1 \leq m \leq k$;*
(b) *for each input variable x_i, both cofactors $\mathbf{P}(x_i = 0), \mathbf{P}(x_i = 1)$ are perfect functions.*

□

Example 5. Consider the two-output function $\mathbf{A} = (A_1, A_0) = (xy + z, x + yz)$. It can assert all 2^2 output combinations. Consider its cofactors with respect to x: for $x = 1$, one of the outputs is constant, and therefore $\mathbf{A}(x = 0)$ asserts 2^1 outputs. On the other hand, $\mathbf{A}(x = 1) = (z, yz)$, which can assert 3 output combinations. Therefore, $\mathbf{A}$ is not perfect.

Consider now the function $\mathbf{B} = (B_1, B_0) = (x \oplus a, x \oplus b)$. Both cofactors with respect to x are trivially perfect. It can similarly be verified that the cofactors with respect to any of a, b are perfect functions. Therefore, $\mathbf{B}$ is perfect. □

Definition 6. *A decomposition is termed* **perfect** *if :*

1. *The root is a perfect root;*
2. *the input functions are full-range and perfect;*
3. *The root does not have any perfect decomposition.*

In this case, we also say that R is a **prime** *function.* □

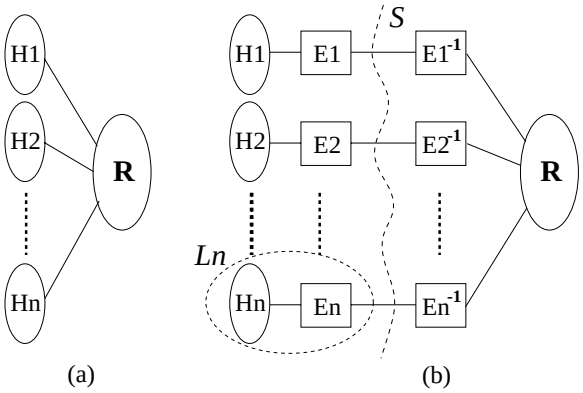

Figure 3. Graphical representation of equivalent decompositions

3.3 Equivalence and uniqueness of perfect decompositions.

Definition 7. *A E-block is any 1-1 function $\mathbf{E} : \mathcal{B}^k \rightarrow \mathcal{B}^k$.* □

The role of E-blocks is that of re-encoding k-bit words into other k-bit words. Because E-blocks are 1-1, this encoding is lossless. It trivially follows that if $\mathbf{A}$ is a perfect function, also $\mathbf{E}(\mathbf{A})$ is a perfect function.

Definition 8. *We say that $\mathbf{A}$ and $\mathbf{B}$ are E-equivalent ($\mathbf{A} \approx \mathbf{B}$ if there exists an E-block $\mathbf{E}$ such that $\mathbf{B} = \mathbf{E}(\mathbf{A})$.* □

Definition 9. *A decomposition $(R; \mathbf{H}_1, \cdots, \mathbf{H}_h)$ is termed* **perfect** *if R is a perfect root and each input function $\mathbf{H}_i$ is a full-range perfect function.* □

Consider a decomposition $(R; \mathbf{H}_1, \cdots, \mathbf{H}_h)$. By suitably using E-blocks, it is possible to transform this decomposition in a different one. Figure (3) illustrates graphically this process.

Definition 10. *We say that two decompositions $(R; \mathbf{H}_1, \cdots, \mathbf{H}_h)$ and $(S; \mathbf{L}_1, \cdots, \mathbf{L}_l)$ are* **equivalent** *if and only if:*

1. *$h = l$;*
2. *there exist h E-blocks $\mathbf{E}_i$ (possibly degenerating in identity functions) such that*

$$\mathbf{L}_i = \mathbf{E}_i(\mathbf{H}_i); \tag{15}$$

$$S = R(\mathbf{E}_1^{-1}(x_1, \cdots), \mathbf{E}_2^{-1}(), \cdots) \tag{16}$$

□

We are now able to state a theoretical result of this work.

Theorem 1. *If a function F has two distinct perfect decompositions, namely $R; \mathbf{H}_1, \cdots, \mathbf{H}_h$ and $S; \mathbf{L}_1, \cdots, \mathbf{L}_l$, these two decompositions are equivalent.* □

The proof, although not difficult, is somewhat laborious. It is therefore omitted from this version of the paper, but it will be made available upon request to the authors.

Unlike perfect decompositions, AP-decompositions may be not unique: **Example 6.** Consider the function $F =$

$x'y'a + xy'b + x'yc + xyd$. It can be decomposed as

$$F = MUX(x_1, x_2, x_3) \quad (17)$$

$$(x_1, x_2) = (x'a + xb, x'c + xd); \quad (18)$$

$$x_3 = y; \quad (19)$$

as well as

$$F = MUX(x_1, x_2, x_3) \quad (20)$$

$$(x_1, x_2) = (y'a + yc, y'b + yd); \quad (21)$$

$$x_3 = y; \quad (22)$$

Both decompositions are AP. □

As mentioned previously, the non-uniqueness of AP-decompositions implies that we may miss opportunities for discovering them. On the other hand, we will be sure of finding perfect decompositions, but such decompositions may be less frequent.

4 Finding (almost) perfect decompositions.

The procedures for finding AP-decompositions and truly perfect decompositions are very similar. Therefore, they will be outlined together, marking the point where the two procedures actually differ.

4.1 Overview of the algorithm.

The decomposition algorithm proposed here works its way up a BDD graph, and annotates each node with a decomposition information. This decomposition information contains a list of pointers to the BDDs of the functions $\mathbf{H}_i$. For reasons of memory efficiency, no information is stored about the root function R. This decomposition information is constructed as follows. When a BDD node ν is visited, it contains no information. A procedure checks the similarity of the decomposition of two cofactors of each node ν. The actual comparison is decribed below. If the decompositions are deemed "sufficiently" similar, then a decomposition is inferred for the function rooted at ν and ν is annotated with its own decomposition information.

We now describe how the decompositions of the cofactors are compared, and how the decomposition of the node is constructed.

4.2 Comparing decompositions.

The following observation is central when identifying decompositions: If a function $F(z, x_1, \cdots, x_n)$ has a non-trivial decomposition, then also its cofactors with respect to , say, z, must have a "similar" decomposition. "Similar", in this context, means that :

1. The decomposition of each cofactor F_0, F_1 is similar to that of F, and
2. the decomposition of the two cofactors are "similar" among themselves.

Indeed, consider a function F, with decomposition

$$F = R(\mathbf{A}(z, x_1, \cdots, x_A), \mathbf{B}(x_{A+1}, \cdots, x_B), \cdots). \quad (23)$$

Consider its cofactors with respect to the top variable z:

$$F_0 = R(\mathbf{A}_0(x_1, \cdots, x_A), \mathbf{B}(x_{A+1}, \cdots, x_B), \cdots) \quad (24)$$

and

$$F_1 = R(\mathbf{A}_1(x_1, \cdots, x_A), \mathbf{B}(x_{A+1}, \cdots, x_B), \cdots) \quad (25)$$

Eqs. (23-25) indicate that superficially the two cofactors have a decomposition similar to that of F: the root function is the same, and most of the input functions are the same. In practice, we need to analyze three decomposition "types" separately :

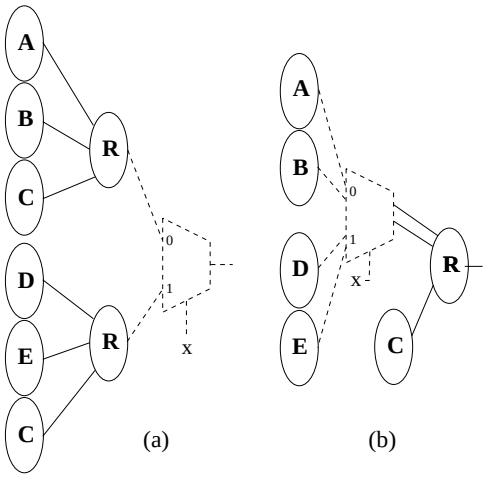

Figure 4. Decomposition of a complex function.

1 The two functions $\mathbf{A}(z = 0)$ and $\mathbf{A}(z = 1)$ are both full-range. The two cofactors will have decompositions with the same root and similar decomposition lists.

2 Exactly one of the two functions (say, $\mathbf{A}(z = 0)$ is full-range. The other cofactor is equivalent to the parallel composition of a constant and a full-range k'-output function, where $k' < k$. In this case, R will no longer be the root function for the decomposition of F_1. The decomposition tree of F_1, however, will contain the functions $\mathbf{H}_i$.

3 Neither cofactor is full range. The observations of Type 2 apply to both cofactors. The functions $\mathbf{H}_i$ will appear in both decomposition trees.

When we inspect a node ν, we first verify if a function has a decomposition of Type 1. If the test suceeds, we construct the decomposition information. If it fails, we move on to check if it has a decomposition of Type 2, etc ...

We now show how to verify decompositions of Type 1, and how to construct the decomposition list. For reasons of space, the full description of the other cases is deferred to a more complete version of this work.

4.3 Type 1.

We inspect the decomposition lists of the two cofactors. If they differ by more than one element, then the decomposition of ν cannot be of type 1. Otherwise, let $\mathbf{H}$ and $\mathbf{L}$ denote the two differing elements in the decomposition lists.

Before asserting that node ν has a decomposition of type 1, We need then to check whether the two root functions are the same. While this check is conceptually simple, its formalization is notationally cumbersome. Therefore, it is convenient to describe it by examples.

Example 7. Consider the situation of Fig. (4.a). A, C, D denote two-bit busses, encoding positive integers, and $F_0 = LARGER(A, C)$, and $F_1 = LESS\_OR\_EQUAL(C, D)$. A, C, D may be primary inputs or functions of other variables. The only relevant information is that the support of A is disjoint from that of C, and that the supports of C and D are disjoint as well.

We now show that the root of the two decompositions is essentially the same, and that the comparator function $LARGER$ can be shared. [1].

To this end, the following actions are taken:

1. The inspection of the two fanin lists shows that they differ only in one argument, namely, A vs. D. The case would be rejected as "non-Type-1" if the fanin lists differ by more than one element.
2. We replace A with two independent Boolean variables, a_1, a_0. We then replace D with two independent variables, d_1, d_0.
3. The four cofactors of F_0 ($F_{0,0}, F_{0,1}, F_{0,2}, F_{0,3}$) with respect to a_1, a_0 are computed. The four cofactors of F_1 ($F_{1,0}, F_{1,1}, F_{1,2}, F_{1,3}$) with respect to d_1, d_0 are also computed.
4. The two sets of cofactors are compared against each other. In this case, each $F_{0,i}$ expresses the condition $C < i$. Each cofactor $F_{1,i}$ expresses the condition $C < i$ as well. The two sets of cofactors match exactly pairwise. Therefore, the root function can be re-written as
$$F = LARGER(zA + z'D, C)$$
whose circuit is shown in Fig. (4.b).

□

In the above example, we had $F_{0,i} = F_{1,i}; i = 0, \cdots, 3$. An input function $H = z'A + zD$ has been constructed. Notice that H is full-range, because both A and D are full range. Even if A and D were perfect, however, we would not be able to assert the perfection of H. If we are interested in perfect decompositions, we need to test the perfection of H. If H is perfect, then the decomposition is listed as perfect. Otherwise it is rejected. Currently, our test is based on the definition of perfect functions. This test is laborious, but we hope to find more efficient tests in the future. At this stage, we do not know of an efficient test that would help

In the previous example, we created a 1-1 matching between cofactors. To address general cases, however, an arbitrary 1-1 matching between cofactors must be considered. We illustrate this by means of the following example.

Example 8. Consider the case $F_0 = ADD\_MSB(C, A)$ and $F_1 = SUB\_MSB(C, B)$, that is, the most significant bit of the two arithmetic operations. The operations are on two-bit quantities, addition is modulo-4, and subtraction is in two's complement[2].

The two operand lists differ in only one component (namely, A vs. B). The four cofactors of ADD_MSB with respect to A are $F_{0,i} = MSB(C + i); i = 0, \cdots, 3$. Notice that the four cofactors are all distinct. The four cofactors of SUB_MSB with respect to B are $F_{1,i} = MSB(C - i); i = 0, \cdots, 3$, also all distinct. Moreover, from the properties of modulo-4 arithmetic and two's complement notation, one gets $C - 0 = C + 0; C - 1 = C + 3; C - 2 = C + 2; C - 3 = C + 1$. Consequently, $F_{0,0} = F_{1,0}; F_{0,1} = F_{1,3}, F_{0,2} = F_{1,2}, F_{0,3} = F_{1,1}$. The four cofactors can be matched pairwise. Therefore, the adder can be shared, provided that a suitable function of A, B and z is created. This function is as follows. When $z = 0$, the function should just replicate A to the adder inputs. When $z = 1$, the function should transform the value of B according to the mapping established by the cofactors of z : when B is, say, 01, then the value 11 should be presented to the adder inputs, and so on. This is accomplished by means of an N-block with a table implementing the cofactor mapping:

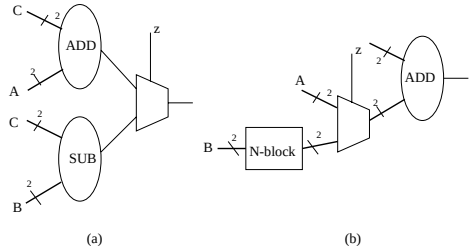

Figure 5. Decomposition of the adder/subtracter circuit.Note the added E-block.

A	B	
00	10	(26)
01	11	(27)
10	10	(28)
11	01	(29)
		(30)

Figure (5) shows the circuit. □

4.4 Clustering wires.

The previous section has shown how to push a variable z when the two cofactor functions are structurally similar. The "bussing" structure into the root function is unchanged. Each bus is the output of a multiple-output, lossless function.

Initially, however, typically only single-output functions are generated. These functions are then clustered into larger, multiple-output functions. In this paragraph, we show the mechanisms for creating these functions. For the sake of conciseness, only two simple cases are described, by means of examples.

Example 9. Consider the case where $F_0 = MUX(a, b, x)$, x being the MUX control, and $F_1 = MUX(c, d, x)$. All variables are primary input variables, and therefore the support lists for F_0 and F_1 are a, b, x and c, d, x, respectively. Variable z cannot be "pushed" anywhere, because the two lists differ by more than one element. Consider, however, making a multiple-output function of a and b on one side, and of c and d on the other side. The new lists now differ by just one element. Moreover, the inspection of the cofactors shows that the grouping is legal (because the cofactors of MUX with respect to all combinations of a and b are distinct), so that z can now be pushed. □

Example 10. Consider now the case where $F_0 = MUX(a, b, x)$, and $F_1 = d + x$. This represents a corner case where the number of input functions differ. There is one element in common, namely, x. To asses whether we can push the top variable z, we group a, b into a multiple-output function $A = (a, b)$. Now, we have a situation where the two lists differ in just one element, but this element is the 2-output function A on one side, and a 1-output func-

[1] Strictly speaking, the function $LARGER$ is not prime, and therefore it could not be a root. The choice of $LARGER$ is just to simplify notations

[2] Again, the choice of functions was motivated by readability

Circuit	Inputs	Outputs	DEC	CPU
k2	45	45	36	0.47s
apex1	45	45	36	0.45s
apex7	49	37	36	0.03s
seq	41	35	35	0.79s
s1423	91	79	74	3.58s
s953	45	52	23	0.08s

Table 1. Decomposability of some benchmark circuits.

Circuit	Inputs	Outputs	Dec Lit.	CPU	SIS lit
9symml	9	1	74	0.00s	223
alu2	10	6	346	0.01s	357
cm150	21	1	23	0.01s	51
cm151	12	2	17	0.00s	26
cm152	11	1	34	0.00s	22
cu	14	11	75	0.01s	59
mux	21	1	26	0.01s	51

Table 2. Literals count of some benchmark circuits.

tion d on the other side. We handle this situation as follows. We first determine the four cofactors of MUX with respect to the two clustered inputs. The four cofactors are , again, $F_{0,0} = 0, F_{0,1} = x, F_{0,2} = x', F_{0,3} = 1$. Consider now the two cofactors of F_1 with respect to the differing function d: $F_{1,0} = x; F_{1,1} = 1$. There is then a 1-1 mapping from the cofactors of F_1 into the cofactors of F_0:

$$\begin{array}{cc} R_1 & R_0 \\ 0 & 01 \\ 1 & 11 \end{array} \tag{31}$$

This mapping is used to generate the 2-input function, as follows. When $z = 0$, the input function should present (a, b). When $z = 1$, the input function should always present only one of the right-hand side entries of Table (31), that is, $(d, 1)$. Notice that this is a 2-output function obtained by juxtaposing a constant to a lossless function. The final two-output function is then $A = (a_1, a_0) = (z'a + zd, z'b + z1)$, and F can be expressed as $MUX(a_1, a_0, x)$. Notice that the new decomposition is also lossless: both cofactors of A with respect to z are either lossless or the pairing of a lossless function with a constant. Similarly for the cofactors of F with respect to the other variables. □

5 Experimental results.

We are interested in two types of results. First, we are interested in discovering which benchmark functions are decomposable, and , second, what kind of improvement is made possible by decomposition, in terms of, for instance, literal counts.

We report in Table (5) the decomposability of some benchmark circuits. The results are incomplete because, as mentioned, not all corner cases have been implemented. Therefore a function may still be listed as non-decomposable when it actually could be decomposed.

It is interesting to contrast these results against those of [1]. Table (5) lists the functions for which complex decompositions are already found to be strictly better than simple decompositions.

Table (5) shows some literal counts. For these functions, we already know that complex decomposition produces better results than simple decomposition. For instance, benchmark `cm150` was reduced from 47 literals down to 23 literals. We expect further improvements once all corner cases are addressed.

6 Conclusions.

Decomposition is important during optimization because it reduces wiring complexity by identifying functions of minimal clusters of variables. Moreover, heuristically it also reduces area as functions of smaller support are identified whenever possible. In this paper, we have identified some reasons for the difficulty in deriving efficient, exact algorithms for complex decompositions. This analysis allowed us to define two decomposition classes, namely, almost-perfect and perfect decompositions. We have proved that a function has a unique perfect decomposition. Moreover, the "nice" properties of these decompositions allowed us to design a decomposition procedure with predictable run-time behavior. The number of corner cases makes its full implementation complex, but the first experimental results are definitely encouraging.

One byproduct of this work is the definition of a boundary between the proper decomposition process and an encoding step, the latter belonging strictly to logic optimization. Moreover, it further articulated the area of complex decompositions.

References

[1] V. Bertacco and M. Damiani. The disjunctive decomposition of logic functions. In *Proc. ICCAD*, pages 78–82, November 1997.

[2] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *Proceedings 27th ACM/IEEE Design Automation Conference*, pages 620–625, June 1990.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, August 1986.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. DAC*, pages 40–45, June 1990.

[5] R. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, pages 74–116, April 1957.

[6] H. A. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, N.J., 1962.

[7] J. P. Roth and R. M. Karp. Minimization over boolean graphs. *IBM Journal*, pages 661–664, April 1962.

[8] Kevin Karplus. Representing boolean functions with if-then-else dags. Technical Report UCSC-CRL-88-28, Baskin Center for ComputerEngineering & Information Sciences, 1988.

[9] R.K. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *ISCAS, Proceedings of the International Symposyium on Circuits and Systems*, pages 49–54, 1982.

[10] T. Sasao (Ed.). *Logic Synthesis and Optimization*. Kluwer Academic, 1993.

[11] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[12] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Trans. on CAD/ICAS*, 6(6):1062–1081, November 1987.