Chapter 4

Disjoint Support Decompositions

We introduce now a new property of logic functions which will be useful to further improve the quality of parameterizations in symbolic simulation. In informal terms, a function has a Disjoint Support Decomposition (DSD) when it can be expressed by composing two other functions such that there is no sharing between the variables in the support of each function. Moreover, in general a function may have multiple distinct DSD.

This chapter defines and characterizes the Disjoint Support Decomposition (DSD) of logic functions. We provide two novel contributions to the theory of DSD. First, we define a canonical form to represent simulataneously all the DSD of a logic function and we show that any Boolean function has a unique representation within this canonical form. As we discuss in the next section dedicated to the previous work, Ashenhurst had shown that, given a decomposition for a function, there is a unique way of assigning its variables to the support of the component functions in the decomposition (except when associative operators are involved). Our second result proves that the component functions are also uniquely determined, once we apply the rules of our canonical form.

The next chapter will make this theory and its properties applicable to any Boolean function that can be represented through a ROBDD by providing a novel algorithm that automatically exposes all the decompositions of a function by generating our canonical form . This algorithm takes as input a ROBDD representation of a function and returns a tree graph that represents its decompositions and has worst-case complexity that is quadratic in the size of the ROBDD of the function.

4.1 Introduction

Disjoint support decomposability is an intrinsic property of Boolean functions. Given a Boolean function $F(x_1,\ldots,x_n)$, it is often possible to represent F by means of simpler component functions. When F can be represented by means of two other functions, say K and J, such that the inputs of J and K do not intersect, $F = K(x_1,\ldots,x_{j-1},J(x_j,\ldots,x_n))$, then we say that F has a *simple disjoint support decomposition*.

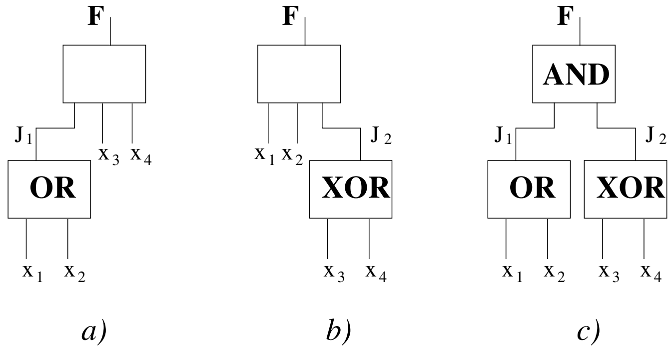

Figure 4.1: Decompositions for Example 4.1

Example 4.1. *The function $F = (x_1+x_2)(x_3 \oplus x_4)$ has a simple disjoint support decomposition where $K = J_1(x_3 \oplus x_4)$ and $J_1 = x_1 + x_2$ as in Figure 4.1.a. The decomposition $K = (x_1+x_2)J_2$ and $J_2 = x_3 \oplus x_4$ is also a simple disjoint support one (Figure 4.1.b). Note that it is possible to combine these two decompositions and represent the function as $F = K(J_1,J_2)$ where $K = j_1 j_2$ (Figure 4.1.c).*

A disjoint support decomposition can also be seen as a way of partitioning the inputs of a function, each element of the partition being the set of inputs to one of the component functions. For instance, with reference to the previous example, the decomposition in Figure 4.1.c corresponds to the partition $\{\{x_1,x_2\},\{x_3,x_4\}\}$. If we consider each of the inputs x_i of F, they can belong to the support of at most one of the component functions, otherwise we would violate the hypothesis of non-intersection. We can also guarantee that they belong to no less than one function; if that was not the case, the resulting function would not have x_i in its support and thus it could not be equal to F.

When a function can be decomposed in more than one way, there is always a decomposition of maximal granularity, that is, a decomposition that imposes a finer partition on the support of F and such that the elements of this partition can be composed to generate all the other decompositions. The last decomposition of Example 4.1 is a maximal decomposition.

We discuss now some previous work on the subject and introduce some formal definitions related to disjoint support decompositions, before we present our contributions.

4.2 Related work on Disjoint Support Decompositions

Algorithms for extracting disjunctive decompositions are a classic research subject of switching theory. Ashenhurst and Singer [3, 63] developed the first theoretical framework in the '50s. In particular, Ashenhurst presented in [3] a classification of the various types of disjoint decompositions. They also introduced an algorithm to detect all the simple decompositions of a function based on *decomposition charts*. The method consists in partitioning the support variables of a function in two sets A and B and detecting if there exist a decomposition such that $F(A,B) = L(P(A),B)$. The method is efficient for functions of up to six variables and it is exponential in the number of variables in the support since it needs to try all the possible partitions of the variable support set. Ashenhurst showed in [3] how simple decompositions can be combined to obtain complex ones and proved that the partition of the support variables induced by decomposition is unique, with the exception of functions representing associative operations. Curtis and Karp explore applications for the theory in the area of synthesis of digital circuits in [30, 44].

In the early '70s, Shen *et al.* , [62], presented an algorithm based on the Jacobian that quickly rules out some partitions as candidates for a disjunctive decomposition. This method achieves good performance when used on undecomposable functions. However, it requires even more computation time for functions which have a decomposition. This algorithm has been implemented recently in

[60].

Alternative simplified techniques, such as algebraic factorization [14], have been extremely successful in transforming large two-level covers in multiple-level representations, and have been extended in various ways to include other forms of decomposition. Algebraic factoring [14] is a form of disjunctive decomposition. In algebraic factorization, one attempts to decompose a 2-level cover of F into a product $G * H$, where G and H have no variables in common. Factoring is a powerful step in passing from a Boolean cover to a multiple-level representation in multiple-level logic synthesis [15]. Logic synthesis have been also attempted starting directly from BDD representations: In [28] it was shown, for instance, that all implicants of a function could be implicitly represented in a BDD. A two-level synthesis algorithm, finding an optimal cover of a function from its BDD, was also developed in [53].

Links between BDDs and multiple-level logic have been explored in [46, 45, 47]. In [47], in particular, it is shown that particular BDD topologies may lead to the identification of particular decompositions. For instance, the presence of a *two-cut* (a partition of the BDD with only two boundary nodes) leads to the identification of disjunctive, MUX-based decompositions. On the other hand, the presence and aspect of two-cuts depends on the variable order of the BDD. Therefore, topological approaches must rely on tailored ordering algorithms.

Decomposition has also been considered in the context of technology mapping [55] and function representation [8, 7]. Bertacco and Damiani proposed in [7] a new function representation that merges BDD and a restricted type of decomposition. In that paper, it was shown that a special decomposition, using only NOR functions, is indeed canonical. If a function F can be decomposed into the NOR of disjoint-support components:

$$F = (f_1 + \cdots + f_n)' \tag{4.1}$$

then, provided that no component function f_i is itself the OR of other disjoint-support functions, the functions f_i are uniquely determined, up to a permutation.

This result was used to develop a hybrid normal form (MLDDs) for logic functions based on Shannon and disjoint-support NOR decompositions. Algorithms for translating BDDs into MLDDs and for the direct manipulation of MLDDs were also presented. This algorithm is capable of identifying a NOR-tree decomposition regardless of the variable ordering selected. The ability of discovering decomposition and the efficiency of the representation, however, are impaired by the restriction to NOR gates. Finally, a preliminary version of the material presented here and in the next chapter was developed by Bertacco and Damiani in [9].

4.3 Terminology

This section covers first a few background definitions that are required for the reminder of the presentation and then provide a formal definition of Disjoint Support Decompositions. For background definitions on Boolean functions the reader is referred to Section 2.3.

We introduce here a special class of functions which will be helpful for our purposes. It consists of those functions whose components are the permutations and/or complementations of the input variables $x_1, \cdots, x_n$ [22, 51].

Definition 4.1. *A function* $\mathbf{F}(x_1, \cdots, x_n)$*:* $\mathcal{B}^n \to \mathcal{B}^n$ *is termed a NP-function if for each of its components* F_i *either* $F_i = x_j$ *or* $F_i = \overline{x_j}$ *for some* j *and* $\mathcal{S}(F_i) \cap \mathcal{S}(F_k) = \emptyset, i \neq k$.

Definition 4.2. *Two functions* $F(x_1, \cdots, x_n)$ *and* $G(x_1, \cdots, x_n)$ *are said to be* **NP-equivalent** *if there is a NP-function* $\mathbf{NP}(x_1, \cdots, x_n)$ *such that*

$$F(x_1, \cdots, x_n) = G(\mathbf{NP}(x_1, \cdots, x_n)) \tag{4.2}$$

that is, F *is obtained by composing* G *with* **NP**.

We can now define the operation of decomposition of a function F as finding other, simpler functions $L : \mathcal{B}^k \rightarrow \mathcal{B}$ and $A_1, \cdots, A_k$ such that

$$F(x_1, x_2, \cdots, x_n) = L(A_1(x_1, \cdots, x_n), A_2(x_1, \cdots, x_n), \cdots, A_k(x_1, \cdots, x_n)) \tag{4.3}$$

The following definition classifies the decomposing function L as a *divisor* of F and introduces the concept of *prime* function. In informal terms, a prime function is any function for which no disjoint support decomposition exists: for instance $F = a + b$ is prime, since it cannot be decomposed by any simpler function. Another example of prime function is the majority function: $F = ab + bc + ca$ has no decomposition through disjoint support components.

Definition 4.3. *A function $L(y_1, \cdots, y_k) : \mathcal{B}^k \rightarrow \mathcal{B}$ is said to* **divide** *a function $F(x_1, \cdots, x_n)$, $n \geq k \geq 2$ if there are k non-constant functions $A_1, \cdots, A_k : \mathcal{B}^n \rightarrow \mathcal{B}$ such that*

$$\begin{aligned} F(x_1, \cdots, x_n) &= L(A_1(x_1, \cdots, x_n), A_2(x_1, \cdots, x_n), \cdots) \qquad (4.4) \\ \mathcal{S}(A_i) \cap \mathcal{S}(A_j) &= \emptyset; \qquad i \neq j \end{aligned}$$

If $n > k$, we say that L **divides** *F* **properly**.

F is said to be **prime** *if it cannot be divided properly by any L.*

Note that, based on the above definition, any function F can always be divided by itself, although improperly. We indicate by F/L any ordered list of functions $(A_1, A_2, \cdots)$ satisfying Eq. 4.4. The list of variables $y_1, \cdots, y_k$ and F/L will be termed **formals list** and **actuals list** of L, respectively.

Definition 4.4. *We call a* **disjunctive decomposition** *of F any pair $(L, F/L)$ that satisfies Eq. 4.4. We distinguish two situations to define* **maximal** *decompositions:*

- *If $L = y_1 \otimes y_2 \otimes \ldots y_n$, and $\otimes$ is one of the associative operators: OR, AND, XOR, the decomposition is said to be maximal iff none of the F/L can be further divided by the same operator, that is the cardinality of the inputs of L is maximal;*

- *Otherwise the decomposition is maximal iff L is a prime function.*

If a function L divides F, then any other function L' that is NP-equivalent to L will also divide F: The actuals list F/L' will be a permutation of the original ones, possibly with some functions $A_i \in F/L$ complemented.

Example 4.2. *Consider the function $F = \overline{x_1}x_2 + \overline{x_1}x_3 + x_1x_4x_5$. It can be divided by $L(y_1, y_2, y_3) = y_1y_2 + \overline{y_1}y_3$. The formals list is (y_1, y_2, y_3), while the actuals list is $(x_1, x_4x_5, x_2 + x_3)$. It can also be divided by $L'(y_1, y_2, y_3) = \overline{y_2}\,\overline{y_3} + y_2y_1$. In this second case the formals list is the same as before , while the actuals list is $(x_4x_5, x_1, \overline{x_2 + x_3})$. Notice that L and L' are NP-equivalent.*

4.3.1 Decomposition trees.

As each function in the actuals list F/L may be itself decomposable, the lists associated with the decomposition of F and of its actuals, form a tree, hereafter called a **decomposition tree** for F.

Leaves of a decomposition tree of a function F are labeled by variables x_i or their complements $\overline{x_i}$. Nodes of the decomposition tree are labeled by a function L that divides the subfunction rooted at that node of the tree.

Example 4.3. *Consider the function $F = \overline{h}((a \oplus b) \cdot MAJ(c, d, e + f) \cdot g) + h(k + j \cdot m)$. Figure 4.2. represents its decomposition tree. The node labeled MUX corresponds to the function $L(y_1, y_2, y_3) = y_3y_1 + \overline{y_3}y_2$ with indexes of the formals list are increasing left to right for the edges in the picture. The node labeled MAJ corresponds to the function majority. The other nodes corresponds to AND, OR and XOR functions.*

4.4 The unique maximal Disjoint Support Decomposition

We can now introduce a characterization of decompositions and decomposition trees. In particular, we show in this Section our first novel result, that, under simple restrictions, every logic function has a unique decomposition tree, and that, much like its BDD, this decomposition tree is also unique. In general, it may be expected that any nontrivial function F can be divided by many functions.

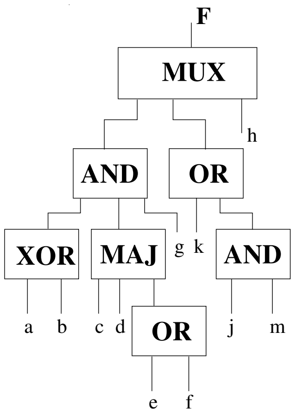

Figure 4.2: A decomposition tree for Example 4.3.

Moreover, for a given divisor L, one may expect that many different functions could contribute to F/L. This section provides a characterization of the divisors of F: we prove that there is actually a unique (modulo NP-equivalence - see Section 4.3) prime function L maximally dividing F, possibly coinciding with F.

We then show an important property of such prime function, namely, that it divides any other function M that divides F. This result leads to a partial ordering of Boolean functions based on the maximal divisor of any function F and is key to the definition of decomposition tree that is presented in Section 4.4.3.

In order to show the uniqueness of the maximal DSD, we now need to introduce the concept of *kernel* function. Notice first that all the Boolean functions with only two inputs can only be one of the associative operators: *AND*, *OR*, *XOR* or their complement or one of their NP-equivalent variants. These functions are also always prime, since they cannot be properly divided by any other function. If a function F can be divided by a prime function L that is a 2 inputs associative operator: AND_2, OR_2, XOR_2, we call *kernel* that function K_F that: 1) divides F, 2) is the same associative operator as L, but 3) has the maximum number of input operands. For instance if $F = a + be + cf$, it can be divided by $L = x_1 + x_2$, but its kernel function is $K_F = x_1 + x_2 + x_3$. In the case where the prime function L has more than 2 inputs, $|\mathcal{S}(L)| > 2$, the kernel function is L itself: $K_F = L$. We show in this section that for a given F, there is a unique K_F, Section 4.4.2 proves that F/K_F

is unique. The reason why we refer to K_F in our presentation is to disambiguate among the many similar functions that can divide a function F that is an associative operator (similar since they differ only in the number of input operands): we choose to use the one with maximum granularity because that is the only one that can impose a unique partitioning on the actuals list of F, F/K_F.

In order to prove the results, we need to introduce some auxiliary terminology.

Definition 4.5. *Given a set of variables $\mathcal{S}$, a* **partition** *$\mathcal{P}$ of $\mathcal{S}$ is a collection of disjoint subsets of $\mathcal{S}$:*

$$
\begin{aligned}
&\mathcal{P} = \{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\} && && (4.5)\\
&\mathcal{S}_i \neq \emptyset; && i = 1, \cdots k &&\\
&\mathcal{S}_i \cap \mathcal{S}_j = \emptyset, && i, j = 1, \cdots k; \qquad i \neq j &&\\
&\bigcup \mathcal{S}_i = \mathcal{S} && &&
\end{aligned}
$$

Given a partition $\mathcal{P}$ of $\mathcal{S}$ into k subsets $\mathcal{S}_1, \cdots \mathcal{S}_k$, we call a **selection** *of $\mathcal{S}$ a subset $\mathcal{S}^{\mathcal{P}}$ of $\mathcal{S}$ containing exactly k variables $x_1, \cdots, x_k$, where $x_i \in \mathcal{S}_i$.*

In other words, $\mathcal{S}^{\mathcal{P}}$ contains one representative variable for each subset $\mathcal{S}_i$ in the partition $\mathcal{P}$.

4.4.1 Decomposition by prime functions.

We first show that any function is decomposed by a unique prime function L. Among all the functions that divide a function F, the prime function L is the one with the smallest number of inputs. We prove here that there is a unique such function L for any F and that any other divisor M with a larger number of inputs, can be further divided. The following Lemma demonstrates this last statement and it is used to show the uniqueness of the prime function L in Theorem 4.2.

Lemma 4.1. *Consider an arbitrary function $F(x_1, \cdots, x_n)$, $n \geq 2$, and let L denote a function dividing F. Then, for any other function M dividing F, if $|\mathcal{S}(M)| > |\mathcal{S}(L)|$, M is decomposable.*

Proof. Let $A_1, A_2, \cdots, A_{|\mathcal{S}(L)|}$ denote the functions in F/L. Recall that such functions are all non-constant and share no support variables. These properties must also hold for the functions in F/M, hereafter listed as $P_1, P_2, \cdots, P_{|\mathcal{S}(M)|}$.

The starting point of the proof is the presumed equality

$$F = L(A_1, A_2, \cdots) = M(P_1, P_2, \cdots). \tag{4.6}$$

The sets $\mathcal{S}(P_1), \cdots, \mathcal{S}(P_{|\mathcal{S}(M)|})$ form a partition of $\mathcal{S}(F)$. Consider building a selection from this partition. We indicate with $x_{P_1}, x_{P_2}, \cdots$ the selected variables, and with $\mathcal{X}_M$ the selection $\{x_{P_1}, x_{P_2}, \cdots\}$ just constructed. Notice in particular that $|\mathcal{S}(M)| = |\mathcal{X}_M|$.

The selection must satisfy one additional property: $\mathcal{X}_M$ must be such that for at least two functions in F/L, say, A_1 and A_2,

$$\mathcal{X}_M \cap \mathcal{S}(A_1) \neq \emptyset \qquad \text{and} \qquad \mathcal{X}_M \cap \mathcal{S}(A_2) \neq \emptyset. \tag{4.7}$$

It is always possible to construct $\mathcal{X}_M$ so that Eq. 4.7 holds, as follows. If Eq. 4.7 is not satisfied by a current selection $\mathcal{X}_M$, then it must be (say) $\mathcal{X}_M \cap \mathcal{S}(A_2) = \emptyset$. Select a variable x_{A_2} from $\mathcal{S}(A_2)$. Notice that x_{A_2} also belongs to the support of some function in F/M, say, to $\mathcal{S}(P_1)$. By replacing x_{P_1} with x_{A_2} in $\mathcal{X}_M$ the new set is still a valid selection, and it satisfies Eq. 4.7.

Consider assigning constant values to the variables not belonging to $\mathcal{X}_M$, in the following way. Since $x_{P_i} \in \mathcal{S}(P_i)$, it is always possible to assign values to the remaining variables of $\mathcal{S}(P_i)$ in such a way that, under this assignment, $P_i = x_{P_i}$ or $P_i = \overline{x_{P_i}}$. For our purpose, complementation is irrelevant. Hence, we will assume for simplicity that this assignment results in $P_i = x_{P_i}$.

We indicate with f^* a function resulting from another function f after this partial assignment.

By applying the same partial assignment to the left-hand side of Eq. 4.6, we obtain

$$L(A_1^*, A_2^*, \cdots) = M(x_{P_1}, x_{P_2}, \cdots). \tag{4.8}$$

Notice that the support of the right-hand side of Eq. 4.8 is precisely $\mathcal{X}_M$. Because $|\mathcal{X}_M| = |\mathcal{S}(M)| > |\mathcal{S}(L)|$, the support of at least one of the functions $A_1^*, A_2^*, \cdots$ must contain 2 or more variables. Because of the partial assignment, some of the functions A_i^* may actually be constants, and the function L may simplify to a different function L^*. From Eq. 4.7, however, at least two functions A_i^* are not constant, hence $|\mathcal{S}(L^*)| \geq 2$ and L^* is a function of at least two inputs. Eq. 4.8 then indicates that L^* divides M and M/L^* is the set of non-constant A_i^*. □

We can now prove that the the prime function L dividing F is unique:

Theorem 4.2. *Let L denote a prime function dividing F. Then, L divides any other function M that divides F.*

Proof. Consider a selection $\mathcal{X}_M$ of $|\mathcal{S}(M)|$ variables and a sensitizing assignment as in the Proof of Lemma 4.1. Eq. 4.6 then reduces to:

$$L(A_1^*, A_2^*, \cdots) = M(x_{P_1}, x_{P_2}, \cdots). \tag{4.9}$$

(remember that f^* is the function resulting from a function f after a partial assignment).

By the way of choice of the variables x_{P_j}, we know that at least two of the functions A_i^* are not constant. We now show that, because of the primality of L, actually none of them can be constant. If, by contradiction, any of the A_i^* were a constant, then L could be replaced in Eq. 4.9 by a function L^* such that $|\mathcal{S}(L^*)| < |\mathcal{S}(L)|$. L^* would also have at least two inputs because at least two A_i^* are not constant. Hence, Eq. 4.9 would indicate that L^* divides M, and therefore it would divide F. We would then have two functions, namely, L and L^*, with $|\mathcal{S}(L)| > |\mathcal{S}(L^*)|$, that divide F. From Lemma 4.1, L would then be decomposable, against the assumption. None of the A_i^* of Eq. 4.9 can then be constant.

Suppose first $|\mathcal{S}(M)| > |\mathcal{S}(L)|$. At least one of the functions A_i^* must have support size larger than one. Hence, Eq. 4.9 indicates that L divides M, and $M/L = \{A_i^*\}$.

If $|\mathcal{S}(M)| = |\mathcal{S}(L)|$, each of the A_i^* must be either a variable x_j from the selection or its complement. In other words, $(A_1, \cdots, A_{|\mathcal{S}(L)|}) = \mathbf{NP}(x_1, \cdots, x_{|\mathcal{S}(M)|})$. Therefore M is NP-equivalent to L. $\square$

Example 4.4. *Consider the function $F(x_1,x_2,x_3,x_4) = x_1x_2x_3 + x_1x_2x_4 + x_3x_4$. It can be decomposed as $F = MAJORITY(x_1x_2, x_3, x_4)$. It is easy to verify that $MAJORITY$ is prime. From Theorem 4.2, F cannot be decomposed with any prime function L other than $MAJORITY$, while maintaining arguments with disjoint support. It follows that $MAJORITY$ is the kernel of F.*

The following result is a direct application of the previous Theorem. It has relevant applications, however, in the rest of the present work.

Corollary 4.3. *If a function F can be decomposed into the 2-input OR (AND, XOR) of two disjoint-support functions, then it cannot be decomposed using any of the other two operators.*

4.4.2 A characterization of F/K_F.

We complete the uniqueness results by providing a characterization of the functions in F/K_F. Theorem 4.4 below, in particular, considers the case where the prime L is not an associative operator. It follows that $|\mathcal{S}(L)| > 2$. As we discussed in the previous Section, in this case $K_F = L$. We show here that, in this situation, the functions in F/K_F are unique. The case where $|\mathcal{S}(L)| = 2$ and $K_F \neq L$ is then considered in Theorem 4.5.

Theorem 4.4. *Let L denote a prime function, with $|\mathcal{S}(L)| > 2$, and let $\mathcal{S}_A = \{A_1, A_2, \cdots, A_{|\mathcal{S}(L)|}\}$, $\mathcal{S}_B = \{B_1, B_2, \cdots, B_{|\mathcal{S}(L)|}\}$ denote two sets of disjoint-support functions such that*

$$L(A_1, A_2, \cdots) = L(B_1, B_2, \cdots) \tag{4.10}$$

Then, there exists a NP-function **NP** *such that*

$$(A_1, A_2, \cdots) = \mathbf{NP}(B_1, B_2, \cdots). \tag{4.11}$$

In other words, $A_1, A_2, \cdots$ *coincides with* $B_1, B_2, \cdots$ *or their complements.*

Proof. We prove the result by contradiction: We assume that two sets of functions exist that satisfy Eq. 4.10 but violate Eq. 4.11, and draw the conclusion that L is not prime.

The proof mechanism is again based on building a selection from the supports of $B_1, B_2, \cdots$.

We need consider two cases. In the first case, the support partition induced by $A_1, \cdots$ coincides with that of $B_1, \cdots$. In the second case, it does not.

First case.

It is not restrictive to assume that $\mathcal{S}(A_1) = \mathcal{S}(B_1); \mathcal{S}(A_2) = \mathcal{S}(B_2), \cdots$. Consider any assignment that is complete for the variables in $\mathcal{S}(A_2), \mathcal{S}(A_3), \cdots$, but that does not assign any values to those variables in $\mathcal{S}(A_1)$. In this case, all functions except A_1 and B_1 reduce to constants. Let $a_2, a_3, \cdots$ denote the constant values $A_2^*, A_3^*, \cdots$. Eq. 4.10 reduces to

$$L(A_1, a_2, a_3, \cdots) = L(B_1, b_2, \cdots) \ . \tag{4.12}$$

Notice that we can always choose the values $a_2, a_3, \cdots$ in such a way that

$$L(A_1, a_2, \cdots) = A_1 \qquad \text{or} \qquad L(A_1, a_2, \cdots) = \overline{A_1}. \tag{4.13}$$

In this case, Eq. 4.12 becomes

$$A_1 = L(B_1, b_2, \cdots) \qquad \text{or} \qquad A_1 = \overline{L}(B_1, b_2, \cdots). \tag{4.14}$$

Since A_1 is, by construction, not a constant, Eq. 4.14 indicates that $L(B_1, b_2, \cdots)$ is also non-constant. On the other hand, L has only one non-constant argument, namely, B_1, and therefore

$L(B_1, b_2, \cdots)$ coincides with either B_1 or with $\overline{B}_1$. Hence, Eq. 4.14 ultimately implies that

$$A_1 = B_1 \qquad \text{or} \qquad A_1 = \overline{B}_1. \tag{4.15}$$

By repeating the same reasoning for all functions in S_A, eventually $(A_1, A_2, \cdots) = \mathbf{NP}(B_1, B_2, \cdots)$ for some NP function.

Second case.

Suppose the support of one function of $\mathcal{S}_A$ (say, A_2) overlaps with the support of (at least) two functions B_j (say, B_1 and B_2). Consider constructing a selection $\mathcal{X}_B$ from $\mathcal{S}_B$ containing at least two variables from $\mathcal{S}(A_2)$. We impose one more requirement on $\mathcal{X}_B$, namely, that for at least another function $A_i, i \neq 2$ $\mathcal{X}_B \cap \mathcal{S}(A_i) \neq \emptyset$.

It is always possible to construct such a selection. If, for a current selection $\mathcal{X}_B$, $\mathcal{X}_B \cap \mathcal{S}(A_i) = \emptyset; i \neq 2$, it would imply $\mathcal{X}_B \subseteq \mathcal{S}(A_2)$. Choose then a variable from, say, $\mathcal{S}(A_3)$. This variable must belong to the support of some B_j. Replace then x_{B_j} with this new variable. For the new selection, $\mathcal{X}_B \cap \mathcal{S}(A_3) \neq \emptyset$, and, since $|\mathcal{X}_B| \geq 3$, at least two variables still belong to $\mathcal{S}(A_2)$.

Notice that, since at least two variables belong to $\mathcal{S}(A_2)$, for at least another function of $A_1, A_2, \cdots$ (say, A_1) $\mathcal{X}_B \cap \mathcal{S}(A_1) = \emptyset$.

Consider applying a partial assignment, such that all functions $B_1, B_2, \cdots$ reduce to variables in $\mathcal{X}_B$ or their complements: $B_i^* = x_{B_i}$. Since no variables of A_1 are included in $\mathcal{X}_B$, A_1 reduces to a constant a_1, and Eq. 4.10 becomes

$$L(a_1, A_2^*(x_{B_1}, x_{B_2}), A_3^*, \cdots) = L(x_{B_1}, x_{B_2}, \cdots) \tag{4.16}$$

Notice that, since the left-hand side of Eq. 4.16 must have support $\mathcal{X}_B$:

1. A_2^* is not a constant;

2. A_3^* is not a constant;

Because a_1 is a constant, however, we can replace L by a simpler function L^* :

$$L^*(A_2^*(x_{B_1}, x_{B_2}), A_3^*, \cdots) = L(x_{B_1}, x_{B_2}, \cdots) \tag{4.17}$$

Eq. 4.17 then indicates that we have been able to decompose L using L^*, and $L/L^* = \{A_2^*, A_3^*, \cdots\}$. This contradicts the assumption that L be a prime function. Hence, this second case is impossible, and F/L is unique. □

Notice that the constraint $|\mathcal{S}(L)| > 2$ is essential to the proof, for if $|\mathcal{S}(L)| = 2$, Eq. 4.16 reduces to

$$L(a_1, A_2^*(x_{B_1}, x_{B_2})) = L(x_{B_1}, x_{B_2}) \tag{4.18}$$

indicating only that A_2^* coincides with L or its complement.

Example 4.5. *Consider again the function $F(x_1, x_2, x_3, x_4) = x_1x_2x_3 + x_1x_2x_4 + x_3x_4$. Its kernel function is $MAJORITY$. From Theorem 4.4, the only possible elements of the actuals list are $A_1 = x_1x_2$, $A_2 = x_3$ and $A_3 = x_4$ or any other NP-quivalent set.*

Example 4.6. *Consider the function $F = x_1 + x_2 + x_3$. It can be decomposed using the function $OR_2(a,b)$ at the root. From Corollary 4.3, no other prime function can be used. The functions F/OR_2, however, are not uniquely identified, as $A_1 = x_1 + x_2$, $A_2 = x_3$ and $A_1 = x_1$, $A_2 = x_2 + x_3$ are both legitimate choices.*

We now address the case where the prime function L dividing F is a 2-input function. It is convenient to restrict our attention to the associative operators OR, AND, XOR: All other 2-input functions are in fact NP-equivalent to one of these operators. Example 4.6 already showed that the inputs of L are not identified uniquely. Because the operators are associative, however, instead of decomposing F using only two arguments A_1, A_2, we allow the number of inputs to the divisor function to be as large as possible and in this case we call K_F such maximum-inputs divisor function.

To this end, we report here a result derived from [7] :

Theorem 4.5. *Suppose a function F is decomposable using one binary associative operator $\otimes$ (where $\otimes = AND, OR, XOR$) as*

$$F = A_1 \otimes A_2 \otimes \cdots \otimes A_n \tag{4.19}$$

and suppose further that none of the component functions A_i is further decomposable using $\otimes$; then the set of functions $\{A_1, \cdots, A_n\}$ is :

- *unique in the case of AND, OR decompositions.*
- *unique modulo complementation for XOR decompositions.*

Proof. The proof follows by contradiction. Assume that there exist two distinct sets of component functions that decompose F, namely, $\{A_1, \cdots, A_n\}$ and $\{B_1, \cdots, B_q\}$; we show that this leads necessarily to the violation of some properties of the functions A_i or B_i.

Consider first the case where $\otimes = OR$. Since the two sets are distinct, at least one of the functions B_j (say, B_1) must differ from any of the functions A_i. Since $\{A_i\}, \{B_j\}$ are both actuals lists for the decomposition of F, it must be :

$$A_1 + \cdots + A_n = B_1 + \cdots + B_q \ . \tag{4.20}$$

Since all functions B_i have disjoint support, it is possible to find a partial assignment of the variables such that $B_j = 0, j = 2, \cdots, q$. Notice that the variables in $\mathcal{S}(B_1)$ have not been assigned any value. Corresponding to this partial assignment, Eq. 4.20 becomes:

$$A_1^* + \cdots + A_n^* = B_1 \tag{4.21}$$

In Eq. 4.21, A_i^* denotes the residue function obtained from A_i with the aforementioned partial assignment.

We need now to distinguish several cases, depending on the assumptions on the structure of the left-hand side of Eq. 4.21.

1. The left-hand side reduces to a constant. Hence, B_1 is a constant, against the assumptions.

2. The left-hand side contains two or more terms. Since these terms must have disjoint support, B_1 is further decomposable by *OR*, against the assumptions.

3. The left-hand side reduces to a single term. It is not restrictive to assume this term to be A_1^*. If $A_1 = A_1^*$, then we have $B_1 = A_1$, against the assumption that B_1 differs from any A_i. Hence, it must be $A_1^* \neq A_1$, and

$$\mathcal{S}(B_1) = \mathcal{S}(A_1^*) \subset \mathcal{S}(A_1) \qquad \text{strictly}. \tag{4.22}$$

We now show that also this case leads to a contradiction.

Consider a second assignment, zeroing all functions $A_i, i \neq 1$. Eq. 4.20 now reduces to

$$A_1 = B_1^* + \cdots + B_q^* \tag{4.23}$$

By the same reasonings carried out so far, the r.h.s. of Eq. 4.23 can contain only one term. We now show that this term must be B_1.

In fact, if $A_1 = B_j^*, j \neq 1$, then by Eq. 4.22 one would have

$$\mathcal{S}(A_1) = \mathcal{S}(B_j^*) \supset \mathcal{S}(B_1) \tag{4.24}$$

against the assumption of B_1, B_j being disjoint-support. Hence, it must be $A_1 = B_1^*$. In this case, by reasonings similar to those leading to Eq. 4.22, we get

$$\mathcal{S}(A_1) = \mathcal{S}(B_1^*) \subset \mathcal{S}(B_1) \qquad \text{strictly} \tag{4.25}$$

which contradicts Eq. 4.22. Hence, B_1 cannot differ from any A_i.

The case where $\otimes = AND$ is derived similarly, with the only difference that we choose partial assignments such that the component functions evaluate to 1.

Finally, for the case $\otimes = XOR$, we choose partial assignments such that the component functions evaluate to 0. Case 1) and 2) are still analogous to the previous derivation above. For case 3), we may reduce to either $B_1 = A_1^*$ or $B_1 = \overline{A_1^*}$. However, in both cases the relation between the supports still holds, in particular Eq. 4.22 and 4.24 are still valid. From the two equations we can then derive the contradiction. □

The Corollary below indicates how the decomposition of Theorem 4.5 is the common denominator of all the other decompositions through an associative operator:

Corollary 4.6. *Suppose F is divided by an associative operator $\otimes$, and let $B_1, \cdots, B_q$ denote a collection of disjoint-support functions such that*

$$F = B_1 \otimes B_2 \otimes \cdots \otimes B_q. \tag{4.26}$$

Then each function B_j can be expressed using terms from the actuals list F/K_F:

$$B_j = A_{k_j+1} \otimes \cdots A_{k_{j+1}} \qquad with \qquad k_1 = 0, k_q = k. \tag{4.27}$$

In other words, F/K_F forms a base *for expressing all possible ways of decomposing F using $\otimes$.*

Proof. We prove the results only for $q = 2$, $\otimes = OR$, the generalizations being straightforward.

Consider the equality

$$F = B_1 + B_2 = A_1 + \cdots + A_k. \tag{4.28}$$

Consider two distinct partial assignments leading to $B_1 = 0$ and to $B_2 = 0$, respectively. Eq. 4.28

becomes

$$B_2 = A_1^* + \cdots + A_k^*; \tag{4.29}$$

and

$$B_1 = A_1^{**} + \cdots + A_k^{**}. \tag{4.30}$$

By computing the *OR* of the two components,

$$F = B_1 + B_2 = A_1^* + A_1^{**} + A_2^* + A_2^{**} + \cdots + A_k^* + A_k^{**} \tag{4.31}$$

From Theorem 4.5, there can be at most k terms in the right-hand side of Eq. 4.31. For each function A_i, at least one of A_i^*, A_i^{**} must be nonzero (or otherwise $\mathcal{S}(A_i) \cap \mathcal{S}(F) = \emptyset$). Hence, for each A_i, either $A_i^* = 0$ and $A_i^{**} = A_i$, or $A_i^* = A_i$ and $A_i^{**} = 0$. Each term in the right-hand sides of Eq. 4.29 is then either 0 or coincides with some A_i. It is not restrictive to assume that the first k_1 terms are nonzero. Hence, Eqs. 4.29 and 4.30 reduce to Eq. 4.27. □

In summary, a function can be decomposed in exactly one of the following ways :

1. By the binary associative operators *AND* or *OR*. In this case, hereafter K_F denotes the *AND* or *OR* function with the largest support size and K_F is unique in the sense of Theorem 4.5.

2. By an *XOR* operator. Also in his case F/K_F will be taken to denote the finest-grain decomposition. F/K_F is unique modulo complementation of an even number of its elements.

3. By a PRIME function of three or more inputs. F/K_F is unique modulo complementation of some of its elements.

Note that the complement of a function has a decomposition that can be derived immediately from the decomposition of the function: If a function is OR-decomposable, its complement has an

AND-decomposition where the inputs are complemented and conversely. The complement of functions with XOR-decompositions are also XOR-decomposition with one of the inputs complemented. Finally, PRIME-decompositions have complements which are PRIME-decompositions with the kernel function K_F complemented.

4.4.3 The normal Decomposition Tree

In Sections 4.4.1 and 4.4.2, we showed that for a given function F, the kernel K_F and the actuals F/K_F are unique, up to complementations and permutations. We now establish some conventions so as to choose a unique representative of all the decomposition trees corresponding to the same function F. These conventions will lead to the definition of the normal Decomposition Tree.

In representing decomposition trees, we reference the tree by a pointer to the root node. Moreover, we use *signed* edges, much like common ROBDD representations ([13]). If a decomposition tree represents the function F, the tree obtained by complementing the edge to the root node represents the function $\overline{F}$.

Definition 4.6. *Given a function, its* **normal Decomposition Tree** *is a tree graph and it is denoted* $\mathbf{DT}(F)$.

It is defined recursively as follows.

The root node represents F, and it is labeled by the type of decomposition. The root node has $|F/K_F|$ outgoing edges, each edge pointing to the root of $DT(A_i)$. In order to resolve permutation ambiguities, the elements of F/K_F are ordered according to the order of their top variable in their ROBDD representation.

In order to resolve complementation ambiguities, the following rules are adopted:

- *If F has PRIME or XOR decomposition, the set F/K_F contains functions with positive ROBDD polarity. In the case of XOR decomposition, the root node will be referred to through a complement edge if necessary.*

- *If F has an AND decomposition, DeMorgan rule is applied: the root node is labeled OR and* *will be referred to through a signed edge. The fanout edges of the root node point to the complements of F/AND_k.*

From Theorems 4.2 and 4.4, it follows trivially that with this set of conventions and with the full labeling of *PRIME* nodes, there is a one-to-one correspondence between normal decomposition trees and logic functions.

Example 4.7. *Consider the function $F = MAJORITY(a \oplus b, cd + e, ITE(fg, h, i))$. F has the following disjoint-support representation:*

$$\begin{aligned} F &= MAJORITY(G, H, I); \\ G &= a \oplus b; \\ H &= L + e; \\ I &= ITE(M, h, i); \\ L &= cd; \\ M &= fg; \end{aligned}$$

The data structure of its normal decomposition tree is reported in Figure 4.3. Notice that the representation is normalized by representing each AND decomposition with it's dual OR and using complement edges. Moreover, since the function I is decomposed through a PRIME, all of its actuals list element must have positive polarity. Thus, the first element A_1 for the decomposition of I is the function $\overline{fg}$ instead of fg and the kernel we use for I is $K_I = \overline{x_0}x_1 + x_0x_2$ which takes into account the polarity change.

The decomposition tree represents concisely all possible disjunctive decompositions of F. This property will be useful to the decomposition algorithm presented in the next chapter. In order to extract a decomposition from a normal Decomposition Tree, we need to define the concept of a *cut* of a DT. Given a generic tree graph, a cut is any set of nodes that separates each leaf of the tree from the root and such that in any path from the root to the leaves there is only one node that

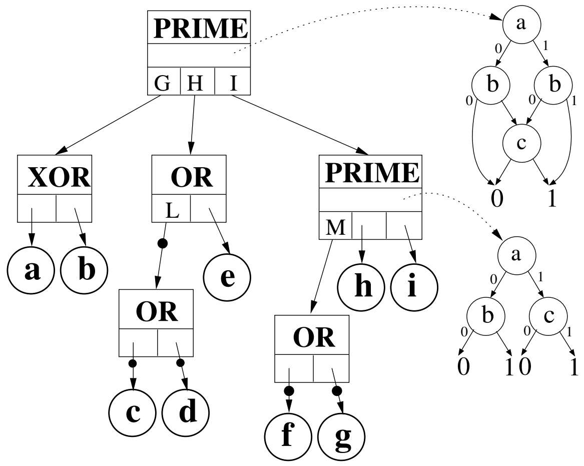

Figure 4.3: Decomposition representation of the function of Example 4.7

belongs to the cut. Since for our Decomposition Trees each node corresponds to a function, we define cuts as a collection of functions. The last lemma of this chapter shows that there is a one to one correspondence between divisors of a function F and cuts through its normal Decomposition Tree.

To formalize this aspect of decomposition trees, we need to introduce a few definitions, which will be used again when we describe our decomposition algorithm.

Definition 4.7. *We say that a function $G(x_1, \cdots, x_n)$* **appears explicitly** *in $DT(F)$ if one of the following holds:*

1. *$G = F$, or*

2. *G appears explicitly in $DT(A_i)$ for some A_i in F/K_F.*

In other words, functions appearing explicitly in $DT(F)$ correspond to tree nodes.

We say G **appears implicitly** *in $DT(F)$ if one of the following holds :*

1. *$G = \overline{F}$*

2. *$F = \otimes(A_1, \cdots, A_n)$ and $G = \otimes(B_1, \cdots, B_m)$, where $\otimes$ is OR, XOR, and where each $B_i \in \{A_1, \cdots, A_n\}$*

3. *F and B_i are as above and* $G = \overline{\otimes(B_1, \cdots, B_m)}$

4. *G appears implicitly in one of the subtrees $DT(A_i)$.*

Finally, we say G **appears** *in $DT(F)$ if it appears explicitly or implicitly.*

Example 4.8. *Consider the function $F = x_1x_2x_3 + x_4x_5$. A decomposition tree is reported in Figure 4.4.a. Figure 4.4.b depicts the equivalent normal Decomposition Tree (the Figure uses the* signed edges *convention to represent the complementation of a node in the tree). The functions $G_1 = F$, $G_2 = x_1x_2x_3$, $G_3 = x_4x_5$, and all the functions corresponding to a simple input variable, appear explicitly in $DT(F)$ and are indicated in the Figure. Functions $G_4 = \overline{x_1} + \overline{x_2}$, $G_5 = \overline{x_1} + \overline{x_3}$ and $G_6 = \overline{x_2} + \overline{x_3}$ appear implicitly in the decomposition tree by rule (2) on implicit appearance of Definition 4.7. Moreover, functions $G_7 = x_1x_2$, $G_8 = x_1x_3$ and $G_9 = x_2x_3$ also appear implicitly by rule (3) of the Definition.*

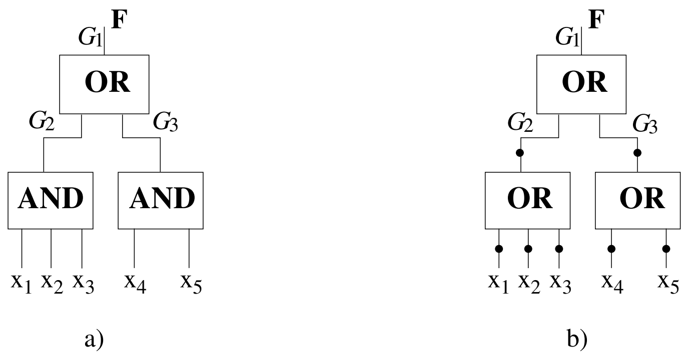

Figure 4.4: Decomposition tree for Example 4.8.

Notice that if a function G appears in $DT(F)$, then every function in $DT(G)$ will also appear in $DT(F)$.

Definition 4.8. *A set of functions $C = \{A_i\}$ is called a* **cut** *of $DT(F)$ if the following hold:*

1. *each function A_i appears in $DT(F)$.*

2. $\mathcal{S}(A_i) \cap \mathcal{S}(A_j) = \emptyset; \qquad i \neq j$

3. $\bigcup_{A_i \in C} \mathcal{S}(A_i) = \mathcal{S}(F)$.

Example 4.9. *A possible cut for the function of Example 4.8 is given by the set* $\{G_3, G_4, x_3\}$*, that is,* $\{x_4x_5, x_1x_2, x_3\}$*. Notice that* G_4 *appears only implicitly.*

Lemma 4.7. *For every function* M *dividing* F*,* F/M *is a cut of* $DT(F)$*. Conversely, for any cut* $\mathcal{C}$ *of* $DT(F)$*, there is a function* M *such that* $F/M = \mathcal{C}$*.*

Proof. The first part of the theorem is proved by induction on the number of variables in $\mathcal{S}(F)$.

The base of the induction (when $|\mathcal{S}(F)| = 2$) is trivial. For the generic induction step, let $m = |\mathcal{S}(M)|$ and let $y_1, \cdots, y_m$ denote formal inputs to M.

Since M divides F, there exist m functions $P_1(x_1, \cdots), P_2, \cdots$ (the actuals list of F/M) such that:

$$F = M(P_1, P_2, \cdots, P_m). \tag{4.32}$$

By assumption, $P_1, \cdots, P_m$ are disjoint support and their support must coincide with $S(F)$. Thus, we need to show only that $P_1, \cdots, P_m$ appear in $DT(F)$.

From Theorem 4.2, the prime function L that divides F, divides also M. Therefore, there exist $l = |S(L)|$ disjoint-support functions $B_1, B_2, \cdots, B_l$ of $y_1, \cdots, y_m$ such that

$$M = L(B_1, \cdots, B_l). \tag{4.33}$$

It is not restrictive to assume that the support variables y_i are numbered so that

$$\mathcal{S}(B_i) = \{y_{b_{i-1}+1}, \cdots, y_{b_i}\} \qquad i = 1, \cdots, l \tag{4.34}$$

for suitable integers b_i, with $b_0 = 0$ and $b_l = m$.

We now need to distinguish whether L is a 2-input function (i.e. an associative operator), or a prime function with three or more inputs. The latter case is simpler and we carry it out first.

By composing Eqs. 4.32 and 4.33 one obtains

$$F = M(P_1, \cdots, P_m) = L(B_1(P_1, \cdots, P_{b_1}), \cdots, B_l(P_{b_{l-1}+1}, \cdots, P_m)) \tag{4.35}$$

Since L divides F and it is a prime function, from Theorem 4.2 it must be

$$A_i = B_i(P_{b_{i-1}+1}, \cdots, P_{b_i}) \qquad i = 1, \cdots, l \tag{4.36}$$

where $\{A_1, \cdots, A_l\} = F/L$.

Notice that by definition of decomposition tree, all A_i appear in $DT(F)$.

For each function A_i, B_i is either a single-input function, or a multiple-input function. In the first case, $A_i = P_{b_i}$ (modulo complementation), and therefore P_{b_i} appears in $DT(F)$. In the second case, Eq. 4.36 indicates that A_i is decomposed by B_i. Since $|\mathcal{S}(A_i)| < |\mathcal{S}(F)|$, by induction each of $P_{b_{i-1}+1}, \cdots, P_{b_i}$ must appear in $DT(A_i)$, hence in $DT(F)$.

Consider now the case where L is an associative operator $\otimes$. In this case, one can write

$$M = G_1(y_1, \cdots, y_{m_1}) \otimes G_2(y_{m_1+1}, \cdots, y_m) \tag{4.37}$$

for suitable functions G_1, G_2. By substituting the formals y_i with the actuals P_i in Eq. 4.37 , and taking into account Theorem 4.5,

$$F = G_1(P_1, \cdots, P_{m_1}) \otimes G_2(P_{m_1+1}, \cdots, P_m) = A_1 \otimes A_2 \otimes \cdots \otimes A_k \tag{4.38}$$

where $\{A_1, \cdots, A_k\} = F/K_F$. We now focus on G_1, the same reasoning being then applicable to G_2. Equation 4.38 is the subject of Corollary 4.6: Either G_1 coincides with some A_i (in which case it appears explicitly in $DT(F)$), or it is expressible as the sum of some of the A_i. In this second case, we have

$$G_1(P_1, \cdots, P_{m_1}) = A_1 \otimes \cdots \otimes A_{k_1} \quad 2 \le k_1 < k. \tag{4.39}$$

Let F_2 denote the function $A_1 \otimes \cdots \otimes A_{k_1}$. Notice that F_2 appears implicitly in $DT(F)$. Hence, every function appearing in $DT(F_2)$ will appear in $DT(F)$. By the inductive assumption, for every function M dividing F_2, F_2/M appears in $DT(F_2)$, hence in $DT(F)$. Eq. 4.39 states precisely that G_1 divides F_2, thus $P_1, \cdots, P_m$ appear in $DT(F_2)$ and consequently in $DT(F)$.

The second statement of the theorem can be trivially proved by building the function M corresponding to the decomposition tree obtained from $DT(F)$ by substituting a distinct variable y_i for each node A_i of the cut $\mathcal{C}$. □

Example 4.10. *Consider the function F of Example 4.7. The function MAJORITY*$(x_1 \oplus x_2, x_3, x_4) = (x_3 + x_4)(x_1 \oplus x_2) + x_3 x_4$ *is a divisor of F and the cut $\mathcal{C}$ of $DT(F)$ with reference to the Example is* $\mathcal{C} = \{a, b, H, I\}$.

4.5 On the decomposability of Boolean functions

Shannon proved in [61] that for a sufficiently large support size, $|\mathcal{S}()|$, almost all Boolean functions require an exponential number of elements for their representation. He also showed in the same paper that the fraction of all functions of a given size support, $|\mathcal{S}()|$, that are decomposable, approaches 0 as $|\mathcal{S}()|$ approaches infinity. Sasao provided some quantitative results on how fast this limit is approached in [59]. He reports there that at $|\mathcal{S}()| = 5$, the percentage of functions that are undecomposable is already 99.9%. However, common experience indicates that most functions representing the functionality of digital systems can be represented by logic networks with much less than an exponential number of elements. The reason lies in the fact that most functions used in digital designs are not randomly picked at all, but instead are usually the results of the designers' natural choice of building complex systems by "adding" together simpler components.

The next chapter presents a new algorithm to expose the disjoint support decomposition components of a Boolean function. Because of its scalability, the algorithm has made possible to compute the disjoint decomposition of many complex functions, showing that most functions used in industrial testbenches are in fact decomposable. It is natural to think that functions that can be represented by a sub-exponential network are more prone to be decomposable because inputs are less intertwined together in their path to produce the output value.