Chapter 5

A novel algorithm for Disjoint Support Decompositions

This chapter introduces a new algorithm to expose the maximal disjoint support decomposition of a Boolean function. The most relevant aspect of this algorithm is that its complexity is only worst-case quadratic in the size of the BDD representation of a function. Previously known algorithms had complexity exponential in the size of the support of the function to be decomposed. It is a well known fact that there exists functions for which the BDD representation is exponential in the size of their support – for instance, the functions representing the outputs of an integer multiplier [18]: in such situations our algorithm does not present any mayor complexity benefit. However, most complex functions that arise in the design and verification of digital circuits have BDD representations that are sub-exponential, hence the widespread use BDDs. For all these complex functions the algorithm introduced here below is the first that can find a disjoint support decomposition in sub-exponential time. Moreover, our algorithm finds the maximal disjoint support decomposition, and consequently all the other decompositions, since they can be all derived from it, as we showed in the previous chapter; on the other hand, previous algorithms could only find one or a few decompositions for a function, not necessarily the maximal DSD.

The algorithm traverses the BDD representation of a function in a bottom up fashion. At each

node it constructs the decomposition of the function rooted at the node based on the type of decompositions of each of the two cofactors of the node. The algorithm works by identifying all the possible situations that may occur at a BDD node and constructing the proper decomposition. The central part of this chapter analyzes the cases that may arise and shows what the resulting decomposition should be for each of them. We present implementation details at the end of the chapter and results we found by decomposing Boolean functions derived from the functionality of industrial digital circuit testbenches.

5.1 Building the decomposition bottom-up

The algorithm to expose the maximal Disjoint Support Decomposition starts from a BDD representation of the function F – see Section 2.3.1 – and finds all its disjoint support components by traversing the BDD tree recursively in a bottom up fashion.

Since we are presenting a recursive approach, we assume to know the disjoint support decomposition of the two cofactors F_0, F_1 with respect to a variable z. This chapter describes how to build the decomposition tree $DT(F)$ from the decomposition of the cofactors, $DT(F_0)$ and $DT(F_1)$.

In principle, one could build $DT(F)$ by running a case analysis based on the decomposition type of F_0, F_1. Example 5.1 below, however, indicates that this information alone may be not enough, and additional comparisons need be carried out on $DT(F_0), DT(F_1)$:

Example 5.1. *Let G, H, J denote three functions, with pairwise disjoint supports. Suppose they all have a PRIME kernel. Suppose also that the two cofactors of F w.r.t. z are as follows:*

$$F_0 = G$$
$$F_1 = G + H$$

That is, F_0 has a PRIME decomposition, while F_1 has an OR decomposition. The decomposition for

F can be found as follows:

$$F = \overline{x}G + xG + xH = G + xH = OR(xH, G)$$

and K_F is an OR function.

Consider now the case where F_1 is as above, while $F_0 = J$. Again we have a situation where F_0 and F_1 decompose through a PRIME and an OR function, respectively. However the decomposition of F results as:

$$F = \overline{x}J + xG + xH = MUX(x, G + H, J)$$

and K_F is a PRIME function.

Thus, functions with different decomposition types can have cofactors whose decomposition types are identical.

In practice, in order to build the decomposition, it is necessary to take the specific actual lists of F_0 and F_1 into consideration. The resulting analysis involves additional comparisons on the actual lists that are often numerous and complex. Therefore, we present here a different solution, based on the following observation:

Example 5.2. *Suppose that F has a decomposition with K_F a PRIME function:*

$$F = K_F(A_1(z), A_2, \cdots, A_l). \tag{5.1}$$

The two cofactors will then have decomposition

$$\begin{aligned} F_0 &= K_F(A_1(z=0), A_2, \cdots, A_l) \\ F_1 &= K_F(A_1(z=1), A_2, \cdots, A_l). \end{aligned} \tag{5.2}$$

If neither $A_1(z=0)$ nor $A_1(z=1)$ is a constant, the kernels of F_0 and F_1 coincide, and the two actuals lists differ in exactly one element. It will be shown below in Section 5.2 that also the inverse is true: if F_0 and F_1 have the same prime kernel and very similar *actuals lists, then F will have the same kernel as F_0 and the actuals list can be readily constructed. A similar observation holds also if F has OR, AND, or XOR decomposition.*

Example 5.2 suggests that we may subdivide the problem by distinguishing the case where both $A_1(z=0)$ and $A_1(z=1)$ are constants, from the case when at most one of them is constant. In fact, the former will requires a simpler analysis to identify the decomposition of the function F. For both the two cofactors of A_1 to be constant, A_1 must have a single variable in its support and it must be $A_1 = z$ or $A_1 = \overline{z}$. We refer to this situation as a *new decomposition*, since in this case we are starting a new decomposition block that contains the single variable at the top of our bottom-up decomposition construction. We refer to the other situation as an *inherited decomposition*, since in this case we are, generally speaking, expanding a block that exists already in the decomposition of the cofactors of F by "adding" the variable z to it.

Definition 5.1. *We say that the decomposition of F: $\langle K_F,\ F/K_F\rangle$ is* **inherited** *if $|\mathcal{S}(A_1)| \geq 2$. It is termed* **new** *otherwise.*

It will be shown that in an inherited decomposition, F shares the kernel (and some actuals) with at least one of its cofactors. In a new decomposition, this is not guaranteed to happen.

Let $A_{10} = A_1(z=0)$ and $A_{11} = A_1(z=1)$, respectively. We further classify *inherited decompositions* as follows:

1. Neither A_{10} nor A_{11} is constant, $A_{10} \neq \overline{A_{11}}$, and

   (a) F has *PRIME* decomposition;

   (b) F has *AND*, *OR* , or *XOR* decomposition;

2. Exactly one of A_{10}, A_{11} is constant (*i.e.* A_1 is the *OR* or *AND* of z – or $\overline{z}$ – with a suitable function); and

(a) F has *PRIME* decomposition;

(b) F has *AND*, *OR*, *XOR* decomposition.

3. $A_{10} = \overline{A_{11}}$ and A_{10} is not a constant (*i.e.* A_1 is the *XOR* of z with a suitable function); and

(a) F has a *PRIME* decomposition;

(b) F has *AND* or *OR* decomposition.

Notice that since A_1 has a *XOR* decomposition, F cannot have a *XOR* decomposition.

Notice that, in the first type of inherited decompositions, A_1 is essentially an arbitrary function of three or more variables. A_1 may of course have a *XOR*, *OR*, or *AND* decomposition, we just exclude the situation where z (or $\overline{z}$) appears as an element of its actuals list. The three scenarios are mutually exclusive, and together they cover all the possibilities for inherited decompositions.

Given this classification of decomposition types, we proceed now as follows: Sections 5.2 to 5.4 cover all the three subtypes of inherited decompositions, Section 5.5 analyzes new decompositions. Each Section shows how to determine which scenario a Shannon decomposition belongs to, and how to construct $DT(F)$ from $DT(F_0)$ and $DT(F_1)$.

5.2 Case 1. Neither A_{10} nor A_{11} is constant and $A_{10} \neq \overline{A_{11}}$

This case was implicitly described in Example 5.2. We need to distinguish the two subcases where F is prime and where F is decomposed by an associative operator. The two subcases are addressed separately by the two Lemmas below:

Case 1.a - *PRIME* decomposition

Lemma 5.1. *A function F has a PRIME decomposition with arbitrary function $A_1(z, \cdots)$ in its actuals list if and only if :*

1. *F_0 and F_1 both have PRIME decompositions;*

2. *the actuals lists F_0/K_{F_0} and F_1/K_{F_1} have the same size, and they differ in exactly one element, called G and H, respectively;*

3. *either*

$$F_0(G=0) = F_1(H=0) \quad \textit{and} \quad F_0(G=1) = F_1(H=1) \tag{5.3}$$

or

$$F_0(G=0) = F_1(H=1) \quad \textit{and} \quad F_0(G=1) = F_1(H=0) \tag{5.4}$$

must hold.

Moreover, if Eq. 5.3 holds, then F/K_F is obtained from F_0/K_{F_0} by replacing G with $A_1 = \overline{z}G + zH$, else by replacing G with $A_1 = \overline{z}G + z\overline{H}$.

Notice that Lemma 5.1 does not require any explicit comparison between K_{F_0} and K_{F_1}. These comparisons are replaced by the comparison of generalized cofactors. We can thus avoid building explicit representations of K_{F_0}, K_{F_1}.

Proof. To prove the *only if* part, notice that Eq. 5.2 indicates that K_F divides F_0 and F_1. Since we assumed K_F to be *PRIME*, K_F will be NP-equivalent to K_{F_0}, K_{F_1}. All elements of F/K_F have positive BDD polarity, hence, $A_2, \cdots, A_l$ will appear with the same polarity in F_0/K_{F_0} and F_1/K_{F_1}. One or both of A_{10}, A_{11}, however, may have negative BDD polarity. Therefore, F_0/K_{F_0} will actually contain either A_{10} or $\overline{A_{10}}$. The same reasoning obviously applies to F_1/K_{F_1}. We indicate with G, H the functions actually appearing in F_0/K_{F_0} and F_1/K_{F_1}, respectively. To verify the third point, consider taking the generalized cofactors of F_0 and F_1 with respect to G and H. If A_{10} and A_{11} have the same polarity (say, positive), then $K_{F_0} = K_{F_1}$ and we have:

$$F_0(A_{10}=0) = K_{F_0}(G=0, A_2, \cdots, A_l) = K_{F_1}(H=0, A_2, \cdots, A_l) = F_1(A_{11}=0) \tag{5.5}$$

$$F_0(A_{10}=1) = K_{F_0}(G=1, A_2, \cdots, A_l) = F_1(A_{11}=1). \tag{5.6}$$

If A_{10} and A_{11} have opposite polarity (say, A_{10} has negative polarity), then

$$F_0(A_{10}=0) = K_{F_0}(G=0,A_2,\cdots,A_l) = K_{F_1}(H=1,A_2,\cdots,A_l) = F_1(A_{11}=1) \quad (5.7)$$

$$F_0(A_{10}=1) = K_{F_0}(G=1,A_2,\cdots,A_l) = F_1(A_{11}=0). \quad (5.8)$$

Hence, Eqs. 5.5 and 5.7 reduce to Eq. 5.3 and 5.4.

To prove the *if* part, recall that F_0 and F_1 both have PRIME decompositions and that their actuals list differ in exactly one element (G vs. H). The cofactors of F_0 and F_1 with respect to G and H are then well defined. Suppose first Eq. 5.3 holds:

$$F_1 = \overline{H}F_1(H=0) + HF_1(H=1) = \overline{H}F_0(G=0) + HF_0(G=1). \quad (5.9)$$

Using the decomposition of F_0 in Eq. 5.9 :

$$F_1 = \overline{H}K_{F_0}(G=0,A_2,\cdots,A_l) + HK_{F_0}(G=1,A_2,\cdots,A_l) = K_{F_0}(H,A_2,\cdots,A_l). \quad (5.10)$$

Eq. 5.10 indicates that K_{F_0} divides F_1 as well, hence F_0 and F_1 have the same decomposition type. From Eq. 5.3 it also follows that

$$F = \overline{z}F_0 + zF_1 = \overline{z}K_{F_0}(G,A_2,\cdots,A_l) + zK_{F_0}(H,A_2,\cdots,A_l) = K_{F_0}(\overline{z}G + zH,A_2,\cdots,A_l) \quad (5.11)$$

Eq. 5.11 indicates precisely that F has *PRIME* decomposition (its kernel being K_{F_0}), and that $F/K_F = \{\overline{z}G + zH,A_2,\cdots,A_l\}$, which is what we needed to prove.

The case where Eq. 5.4 holds can be handled in the same way, just by replacing H with $\overline{H}$. □

Example 5.3. *The function $F = azb + e\overline{z}b + cb \oplus d$ has kernel $K_F(x_1,x_2,x_3,x_4) = x_1x_2 + x_3x_2 \oplus x_4$ and actuals list $(az + e\overline{z},b,c,d)$. By computing the cofactors w.r.t. z, we obtain F_0 and F_1 with kernel identical to K_F and actuals lists: $F_0/K_{F_0} = (e,b,c,d)$ and $F_1/K_{F_1} = (a,b,c,d)$, respectively. Since the two cofactors satisfy all the three conditions of Lemma 5.1, we can find the decomposition of F*

from their kernels and actuals lists.

Case 1.b - Associative decomposition

The case where F is decomposed by an associative operator is slightly more complex. Therefore, we first provide the intuition, and then prove formally a criterion for identifying such a case. Suppose F has a (say) *OR* decomposition:

$$F = OR_k(A_1(z), A_2, \cdots, A_k)$$

The two cofactors will also have *OR* decomposition:

$$F_0 = OR_k(A_{10}, A_2, \cdots, A_k) \quad \text{and} \quad F_1 = OR_k(A_{11}, A_2, \cdots, A_k)$$

Notice, however, that one or both of A_{10}, A_{11} may have a *OR* decomposition as well. Let

$$A_{10} = OR_l(B_1, B_2, \cdots, B_l) \quad \text{and} \quad A_{11} = OR_m(C_1, \cdots, C_m) \quad \text{where} \quad l, m \geq 1.$$

Therefore, $K_{F_0} = OR_{k-1+l}$, $F_0/K_{F_0} = \{B_1, \cdots, B_l, A_2, \cdots, A_k\}$ and $K_{F_1} = OR_{k-1+m}$, $F_1/K_{F_1} = \{C_1, \cdots, C_m, A_2, \cdots, A_k\}$. Notice that all the functions B_i must differ from all of the C_j, and that the two actuals lists still have at least one element in common $(A_2, \cdots, A_k)$. These observation are formalized in Lemma 5.2 below:

Lemma 5.2. *A function F has an OR decomposition with arbitrary function $A_1(z, \cdots)$ in its actuals list if and only if:*

1. *both F_0 and F_1 have OR decompositions;*
2. *the set of common actuals $\mathcal{A}_c = \{A_2, \cdots, A_k\}$ is not empty;*
3. *$F_0/K_{F_0} - \mathcal{A}_c \neq \emptyset$ and $F_1/K_{F_1} - \mathcal{A}_c \neq \emptyset$.*

Proof. The *only if* part of the proof follows immediately from the previous observations. For the *if* part, let $B_1, \cdots, B_l$ denote the functions in $F_0/K_{F_0} - \mathcal{A}_c$, and $C_1, \cdots, C_m$ those in $F_1/K_{F_1} - \mathcal{A}_c$. Then ,

$$F_0 = OR_{k-1+l}(B_1, \cdots, B_l, A_2, \cdots, A_k) \quad \text{and} \quad F_1 = OR_{k-1+m}(C_1, \cdots, C_m, A_2, \cdots, A_k)$$

Hence,

$$F = z' F_0 + z F_1 = OR_k(\overline{z} OR_l(B_1, \cdots, B_l) + z OR_m(C_1, \cdots, C_m), A_2, \cdots, A_k) \tag{5.12}$$

We need to show now that $\overline{z} OR_l(B_1, \cdots, B_l) + z OR_m(C_1, \cdots, C_m)$ does not have an *OR* decomposition. Suppose, by contradiction, that it has an *OR* decomposition. Then, some of the terms $(B_1, \cdots, B_l)$ would coincide with some of the $(C_1, \cdots, C_m)$, against our assumptions. Hence, Eq. 5.12 indicates that F has a OR_k decomposition. □

Identical results can be shown for the *AND* and *XOR* cases.

5.3 Case 2. Exactly one of A_{10}, A_{11} is constant

We now assume that exactly one of A_{10}, A_{11} is a constant. We consider only the case $A_{10} = 0$, so that effectively $A_1 = zA_{11}$. The other cases can be handled similarly. In this scenario we need to consider separately the case where F will have a *PRIME* decomposition, and the case where F will be decomposed by an associative operator.

Case 2.a - *PRIME* decomposition

In this case :

$$F = K_F(A_1, A_2, \cdots, A_l)$$

where K_F is a *PRIME* function. Recalling that $A_1 = zA_{11}$, the two cofactors are:

$$F_0 = K_F(0, A_2, \cdots, A_l) \quad \text{and} \quad F_1 = K_F(A_{11}, A_2, \cdots, A_l) \tag{5.13}$$

Eq. 5.13 indicates that:

1. K_F is also the kernel of F_1;

2. F_1/K_F differs from F/K_F in exactly one element (A_{11} va. A_1).

3. K_F **is not** the kernel of F_0.

Again, the following Lemma helps us avoid comparing kernels explicitly:

Lemma 5.3. *A function F has a PRIME decomposition with $A_1 = zG$ in its actuals list, for a suitable non-constant function G if and only if :*

1. *F_1 has a PRIME decomposition;*

2. *there exists a function $G \in F_1/K_{F_1}$ such that $F_1(G = 0) = F_0$.*

Proof. For the *only if* part, recall that Eq. 5.13 indicates that F_1 has the same kernel as F. The second point also follows immediately from Eq. 5.13, using $G = A_{11}$.

For the *if* part, notice that, since $F_0 = F_1(G = 0)$,

$$F = z'F_0 + zF_1 = z'K_{F_1}(0, A_2, \cdots, A_l) + zK_{F_1}(G, A_2, \cdots, A_l) = K_{F_1}(zG, A_2, \cdots, A_l)$$

indicating precisely that K_{F_1} is also the kernel of F , and that $A_1 = zG$. □

It is worth noticing that Lemma 5.3 does not indicate which function in F_1/K_{F_1} needs be chosen for the cofactoring. Indeed, all functions $A_i \in F_1/K_{F_1}$ such that $\mathcal{S}(A_i) \cap \mathcal{S}(F_0) = \emptyset$ are candidates.

Case 2.b - Associative decomposition

Since we assumed at the beginning of Section 5.3 that A_1 has an *AND* decomposition, zA_{11}, F can have only *OR* or *XOR* decomposition. We focus here on *OR* decompositions, the *XOR* case being conceptually identical.

Again, we need to consider the case where A_{11} itself may have an *OR* decomposition.

Let

$$A_{11} = OR_l(B_1, \cdots, B_l) \quad l \geq 1.$$

The case where A_{11} does not have a *OR* decomposition is implicitly addressed by $l = 1$. The decomposition of F can then be written as :

$$\begin{aligned} F &= OR_k(zA_{11}, A_2, \cdots, A_k) \quad k \geq 2 \\ F_0 &= OR_{k-1}(A_2, \cdots, A_k) \qquad (5.14) \\ F_1 &= OR_k(A_{11}, A_2, \cdots, A_k) = OR_{k+l-1}(B_1, \cdots, B_l, A_2, \cdots, A_k) \qquad (5.15) \end{aligned}$$

Equation 5.15 indicates that F_1 will also have an *OR* decomposition. F_0, however, may have a different decomposition: in fact, in the special case $k = 2$, Eq. 5.14 simplifies to $F_0 = A_2$ and A_2 does not have an *OR* decomposition by hypothesis. In the general case, all the actuals of F_0/OR_{k-1} will belong to F_1/OR_{k+l-1}. In the special case $k = 2$, F_0 itself will be an element of F_1/OR_{k+l-1}. These observations are formalized below:

Lemma 5.4. *A function F has an OR_k decomposition with $A_1 = zG$ in its actuals list, for a suitable non-constant function G if and only if:*

1. *F_1 has an OR_{k+l-1} decomposition with $k \geq 2$ and $l \geq 1$;*

2. *either $k > 2$ and F_0 has an OR_{k-1} decomposition and $F_0/K_{F_0} \subset F_1/K_{F_1}$; or $k = 2$ and $F_0 \in$*

F_1/K_{F_1}.

Proof. The *only if* part follows directly from the above observations. For the *if* part, suppose:

$$F_1 = OR_{k-1+l}(B_1,\cdots,B_l,A_2,\cdots,A_k)$$

and

$$F_0 = OR_{k-1}(A_2,\cdots,A_k).$$

Consequently, we have:

$$\begin{aligned} F &= \overline{z}F_0 + zF_1 = OR_2(OR_{k-1}(A_2,\cdots,A_k), zOR_l(G_1,\cdots,G_l)) \\ &= OR_k(zOR_l(G_1,\cdots,G_l),A_2,\cdots,A_k) \end{aligned}$$

which is what we needed to show. Notice that the algebra holds also for the corner case $k = 2$. □

5.4 Case 3. $A_{10} = \overline{A_{11}}$ and A_{10} is not a constant

In this scenario A_1 has *XOR* decomposition : $A_1 = z \oplus A_{10}$. It is not restrictive to assume that A_{10} has positive BDD polarity. Again, we need to address the case where F has a *PRIME* decomposition separately from the other cases.

Case 3.a - *PRIME* decomposition

If F has *PRIME* decomposition, then

$$F_0 = K_F(A_{10},A_2,\cdots,A_l) \quad \text{and} \quad F_1 = K_F(\overline{A_{10}},A_2,\cdots,A_l) \tag{5.16}$$

Again, K_{F_0} and K_{F_1} are NP-equivalent to K_F, hence, F_0 and F_1 have *PRIME* decompositions. Moreover, F_0/K_{F_0} and F_1/K_{F_1} are identical (because of the definition of normal Decomposition Tree - see

also Section 4.4.3). Another consequence of Eq. 5.16 is that:

$$F_0(A_{10}=0) = F_1(A_{10}=1) \qquad F_0(A_{10}=1) = F_1(A_{10}=0)$$

The following Lemma provides necessary and sufficient conditions for identifying this case:

Lemma 5.5. *A function F has a PRIME decomposition with $A_1 = z \oplus G$ in its actuals list, for a suitable non-constant function G if and only if:*

1. *F_0 and F_1 have PRIME decompositions;*
2. *$F_0/K_{F_0} = F_1/K_{F_1}$;*
3. *there exists a function H in F_0/K_{F_0} such that:*

$$F_0(H=0) = F_1(H=1) \quad \textit{and} \quad F_0(H=1) = F_1(H=0) \tag{5.17}$$

In this case, either $G = H$ or $G = \overline{H}$.

Proof. The *only if* part follows directly from the introduction to this case. For the *if* part, observe that if Eq. 5.17 holds, then:

$$\begin{aligned} F_1 &= \overline{H}F_0(H=1) + HF_0(H=0) \\ F &= \overline{z}F_0 + zF_1 = \overline{z}\overline{H}F_0(H=0) + \overline{z}HF_0(H=1) + z\overline{H}F_0(H=1) + zHF_0(H=0) \\ &= (\overline{z}\overline{H} + zH)F_0(H=0) + (z\overline{H} + \overline{z}H)F_0(H=1) = F_0(H = z \oplus H). \end{aligned}$$

Hence, F has the same kernel as F_0, and its actuals list coincides with that of F_0, except for one element, namely, H, which is being replaced by either $z \oplus H$ or by $\overline{(z \oplus H)}$, depending on the polarity of the BDD representation. □

Notice that Lemma 5.5 does not indicate which function of F_0/K_{F_0} needs to be *XOR*-ed with z. Unfortunately, there is no way of knowing other than checking each function until Eq. 5.17 is verified.

Case 3.b - Associative decomposition

The difference from Case 3.a lies again in the fact that the candidate H may have the same decomposition type (*AND*, *OR*) as F. The way to handle this difference has been described already in Sections 5.2 and 5.3 for the other cases. Therefore, we omit it from the present analysis.

5.5 New decompositions

We now consider the case where $A_1 = \overline{z}$ or $A_1 = z$. We need to distinguish three subcases, namely,

(a) F has an *AND* or *OR* decomposition;

(b) F has an *XOR* decomposition;

(c) F has a *PRIME* decomposition.

These cases will be handled separately in the three paragraphs below.

Case a - *AND* or *OR* decomposition

$F = OR(z,G)$, then the two cofactors are $F_{\overline{z}} = G$ and $F_z = 1$. Conversely, if $F_z = 1$, then $F = F_{\overline{z}}\overline{z} + 1z = OR(z,F_{\overline{z}})$. Hence the decomposition is inferred by verifying that one of F_z is the constant 1. Since $z \notin S(F_G)$, F has a *OR* decomposition with $z \in F/OR$. The second case can be treated similarly showing that F has an *OR* decomposition with $\overline{z} \in F/OR$ if and only if the cofactor $F_{\overline{z}}$ is the constant 1. The case of *AND* decomposition is symmetrical, with the constant 0 replacing the constant 1. In summary, a new *AND* or *OR* decomposition is discovered if one of the two cofactors F_0 or F_1 is a constant:

- $F_1 = 1 \rightarrow F = z + F_0$.
- $F_0 = 1 \rightarrow F = \overline{z} + F_1$.
- $F_0 = 0 \rightarrow F = z \cdot F_1 = \overline{\overline{z} + \overline{F_1}}$.
- $F_1 = 0 \rightarrow F = \overline{z} \cdot F_0 = \overline{z + \overline{F_0}}$.

Case b - *XOR* decomposition

If $F = XOR(z, G)$, then $F_{\overline{z}} = G, F_z = \overline{G}$, and conversely, if $F_z = \overline{F_{\overline{z}}}$, then F has XOR decomposition with $z \in F/XOR$. For this case, the decomposition is inferred by checking that $F_z = \overline{F_{\overline{z}}}$.

Case c - *PRIME* decomposition

This case is by far the most complex of all. There are no necessary and sufficient conditions for identifying this case : It is determined by failing to construct any other type of decomposition. As mentioned, we do not need to keep track of the particular *PRIME* function used in the decomposition. Therefore, the task at hand is just to identify the actuals list F/K_F. Unlike the previous cases, in order to build this list, we will need to compare not just the actuals lists F_0/K_{F_0}, F_1/K_{F_1}, but the entire trees. Fortunately, this comparison can still be carried out efficiently. The rest of this section contains the details of this construction and the theoretical justification.

Consider once again the Shannon decomposition of a function F with disjunctive decomposition $F = K_F(z, A_2, A_3, \cdots, A_l)$:

$$F_{\overline{z}} = K_F(0, A_2, A_3, \cdots) \qquad F_z = K_F(1, A_2, A_3, \cdots) \tag{5.18}$$

Let $L_{\overline{y_1}}(y_2, \ldots, y_m)$ and $L_{y_1}(y_2, \ldots, y_m)$ denote the functions $K_F(0, y_2, .., y_m)$ and $K_F(1, y_2, .., y_m)$, respectively. Eq. 5.18 can then be written as

$$F_{\overline{z}} = L_{\overline{y_1}}(A_2, A_3, \cdots) \qquad F_z = L_{y_1}(A_2, A_3, \cdots) \tag{5.19}$$

In general, $L_{\overline{y_1}}$ and L_{y_1} may be further decomposable. Moreover, they may depend on only a subset of $y_2, \cdots, y_m$. For this reason, in order to determine the decomposition of F, it is not sufficient to compare the actuals list of $F_{\overline{z}}, F_z$. However, from Eq. 5.19, $L_{\overline{y_1}}$ divides $F_{\overline{z}}$. From Lemma 4.7, the set of functions $\{A_2, A_3, \cdots\}$ forms a cut of $DT(F_{\overline{z}})$ and thus F/K_F also contains a cut of the same decomposition tree. Similar reasoning applies to F_z.

The definition of uniform-support is needed to identify which functions from the two decomposition trees of the cofactor we need to select as components of $DT(F)$:

Definition 5.2. *Given a function F and a variable $z \in \mathcal{S}(F)$, a function A appearing in $DT(F_{\overline{z}})$ or in $DT(F_z)$ is said to have* **uniform-support** *if it has positive polarity and exactly one of the following is true:*

1. $\mathcal{S}(A) \subseteq \mathcal{S}(F_{\overline{z}}) \cap \overline{\mathcal{S}(F_z)}$ *and A appears in $DT(F_{\overline{z}})$ only;*
2. $\mathcal{S}(A) \subseteq \mathcal{S}(F_{\overline{z}}) \cap \mathcal{S}(F_z)$ *and A appears in both $DT(F_{\overline{z}})$ and $DT(F_z)$;*
3. $\mathcal{S}(A) \subseteq \overline{\mathcal{S}(F_{\overline{z}})} \cap \mathcal{S}(F_z)$ *and A appears in $DT(F_z)$ only.*

A is also termed **maximal** *if for no other uniform-support function B appearing in $DT(F_{\overline{z}})$ or $DT(F_z)$, we have $\mathcal{S}(A) \subset \mathcal{S}(B)$.*

For a given pair of decomposition trees $DT(F_z)$, $DT(F_{\overline{z}})$, we denote by $Max(F_{\overline{z}}, F_z)$ the set of maximal uniform support functions. It is this set of functions, togheter with the top variable z, that we will use as the actuals list for the decomposition of F. Theorem 5.6 shows that this is the correct set of functions for F/K_F.

Example 5.4. *Consider the function F of Figure 5.1. The decomposition of the two cofactors F_0 and F_1 is shown by its normal Decomposition Tree (which includes signed edges to indicate complementation of the function rooted at the signed node). The set $Max(F_{\overline{z}}, F_z)$ for this function is $\{x_1 + x_2, x_3, x_4x_5, x_6\}$. Notice that $x_1 + x_2$ appears implicitly in $DT(F_0)$ by rule (2) of Definition 4.7, while it appears implicitly in $DT(F_1)$ by rule (3) of the same Definition since the first element of F_1/K_{F_1} is $A_1 = \overline{x_1 + x_2 + x_6}$.*

The first three elements of the maximal set satisfy condition 2 of the definition of uniform support, while the last element satisfies condition 1.

As we mentioned, the set $Max(F_{\overline{z}}, F_z)$ effectively represents the actuals list of F. This is stated by the following Theorem:

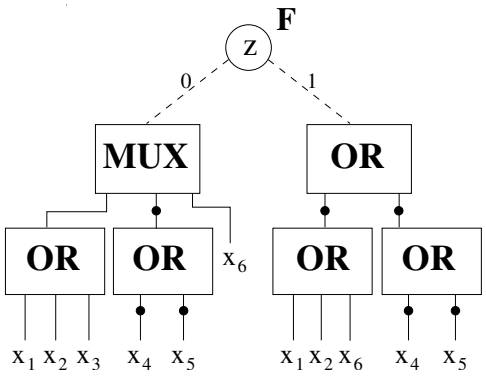

Figure 5.1: PRIME decomposition.

Theorem 5.6. *For a function F with decomposition $F/K_F = \{z, A_1, \cdots, A_l\}$, the actuals list is given by $\{z\} \cup Max(F_{\overline{z}}, F_z)$.*

We first illustrate the result with an example and then prove the Theorem.

Example 5.5. *Based on Theorem 5.6, the actuals list for the decomposition of the function in Figure 5.1 is given by $\{z, x_1 + x_2, x_3, x_4x_5, x_6\}$. The kernel function can then be easily derived by substituting the corresponding element of the formals list for each element of the actuals list. The formals list is $\{y_1, y_2, y_3, y_4, y_5\}$ and the kernel is $K_F = \overline{y_1} MUX(y_2 + y_3, y_4, y_5) + y_1((\overline{y_2 + y_5}) + y_4)$.*

The proof of Theorem 5.6 requires the proof of some properties of uniform-support functions.

Lemma 5.7. *Any two maximal uniform-support functions of $DT(F_{\overline{z}})$ or $DT(F_z)$ have disjoint support.*

Proof. We prove the Lemma by contradiction by showing that if two uniform-support functions A_1, A_2 share support variables, then at least one of them is not maximal. Notice, first of all, that there must be at least one decomposition tree where both functions appear. In fact, if one function only appeared in $DT(F_{\overline{z}})$ and the other only appeared in $DT(F_z)$, then, by definition of uniform-support, they would also be disjoint support. For sake of simplicity, we assume that both functions appear in $DT(F_z)$.

We need now to distinguish a few cases.

- Both A_1 and A_2 appear in $DT(F_z)$ *explicitly*. It is easy to see that, in order for the supports to overlap, either A_1 appears as a node in the subtree $DT(A_2)$, or A_2 appears as a node in the subtree $DT(A_1)$. In the first case, $\mathcal{S}(A_1) \subset \mathcal{S}(A_2)$, while in the second case $\mathcal{S}(A_2) \subset \mathcal{S}(A_1)$. In either case, one of the two functions is not maximal, as we intended to show.

- One of the two functions (say, A_1) appears only *implicitly*, while A_2 appears *explicitly*. Then it must be:

$$A_1 = \otimes_{i=1}^{k} B_i \qquad k \geq 2 \tag{5.20}$$

where $\otimes$ is one of *AND*, *OR*, *XOR* , and B_i are disjoint-support functions. Moreover, there is a function Q_1 appearing explicitly in $DT(F_z)$ such that

$$Q_1 = \otimes_{i=1}^{m} B_i \quad 2 \leq k < m \tag{5.21}$$

Notice that Q_1 does not need to be uniform-support. A_2 shares support variables with A_1, thus, either A_2 appears explicitly in $DT(Q_1)$ or Q_1 appears explicitly in $DT(A_2)$. If Q_1 appears explicitly in $DT(A_2)$ or if $A_2 = Q_1$, however, $\mathcal{S}(A_2) \supset \mathcal{S}(A_1)$, and A_1 is not maximal.

A_2 must then appear explicitly in $DT(Q_1)$, *i.e.*in exactly one of $DT(B_i)$, $i = 1 \ldots m$. But if A_2 appears explicitly in any $DT(B_i), i = 1, \cdots, k$, then $\mathcal{S}(A_2) \subset \mathcal{S}(A_1)$ and A_2 is still not maximal. Finally, if A_2 appears explicitly in any $DT(B_i), i = k+1, \cdots, m$, then $\mathcal{S}(A_2) \cap \mathcal{S}(A_1) = \emptyset$, against the hypothesis.

- Finally, suppose that both A_1 and A_2 appear *implicitly*. Then there must be an associative operator $\oslash = AND$, *OR* or *XOR* such that

$$A_2 = \oslash_{i=1}^{l} C_i. \tag{5.22}$$

Moreover, there must be a function Q_2 appearing explicitly in $DT(F_z)$ such that

$$Q_2 = \oslash_{i=1}^{n} C_i \quad 2 \leq l < n. \tag{5.23}$$

As both Q_1 and Q_2 appear explicitly in $DT(F_z)$, exactly one of the following must hold :

1. $\mathcal{S}(Q_1) \cap \mathcal{S}(Q_2) = \emptyset$. But then $\mathcal{S}(A_1) \subseteq \mathcal{S}(Q_1) \cap \mathcal{S}(Q_2) \supseteq \mathcal{S}(A_2) = \emptyset$, against the hypothesis.

2. Q_1 appears in $DT(C_i)$ for one of the functions $C_i, i \leq l$. But, from Eq. 5.22, $\mathcal{S}(A_1) \subseteq \mathcal{S}(C_i) \subseteq \mathcal{S}(A_2)$, and again one of the functions (A_1) is not maximal.

3. Q_1 appears in $DT(C_i)$ for some $C_i, l < i \leq n$. This case is also impossible since it would be $\mathcal{S}(A_1) \subseteq \mathcal{S}(C_i) \cap \mathcal{S}(A_2) = \emptyset$.

4. $Q_1 = Q_2$. Then, the operator $\otimes$ of Eq. 5.20 must coincide with $\oslash$, and the functions C_i in Eq. 5.23 must coincide with the functions B_i in Eq. 5.21. Hence, A_2 can be written as

$$A_2 = \otimes_{i=j}^{l} B_i \quad \text{for} 1 \leq j \leq k < l \leq m. \tag{5.24}$$

Consider then the function

$$U = \otimes_{i=1}^{l} B_i. \tag{5.25}$$

U contains all the functions in the decomposition of $A_1/\otimes$ and of $A_2/\otimes$. Hence, U has uniform support, and $\mathcal{S}(U) \supset \mathcal{S}(A_1), \mathcal{S}(U) \supset \mathcal{S}(A_2)$, showing again that at least one of A_1, A_2 is not maximal.

In summary, in all cases, the assumption that A_1, A_2 share variables leads to the conclusion that at least one of them is not maximal, as we intended to prove. □

Lemma 5.8. *The set $Max(F_{\overline{z}}, F_z)$ contains a cut of $DT(F_{\overline{z}})$ and of $DT(F_z)$.*

Proof. We only prove that $Max(DT(F_{\overline{z}}), DT(F_z))$ contains a cut of $DT(F_z)$, the second part being entirely symmetrical.

Consider the collection C of functions $A_i \in Max(DT(F_{\overline{z}}), DT(F_z))$ such that $S(A_i) \cap S(F_z) \neq \emptyset$. From the definition of uniform support, for each such function, $S(A_i) \subseteq S(F_z)$ and they appear in $DT(F_z)$. From Lemma 5.7, they are disjoint-support. Therefore,

$$\bigcup_{A_i \in C} S(A_i) \subseteq S(F_z). \tag{5.26}$$

It remains to be shown that the containment relation 5.26 is actually an equality. To this regard, notice that for each variable $x_i \in S(F_z)$, the function x_i is trivially uniform-support. Either it is maximal, or there exist a maximal uniform-support function X_i appearing in $DT(F_z)$ whose support contains x_i. This function must then belong to $Max(DT(F_{\overline{z}}), DT(F_z))$ and therefore x_i must belong to the left-hand side of Eq. 5.26. This completes the proof. □

We define now a *bi-cut* as a set of uniform-support functions that provides a cut for the cofactors' decomposition trees:

Definition 5.3. *Given a function F and a variable $z \in \mathcal{S}(F)$, a collection of uniform support functions (not necessarily maximal) $\mathcal{C}_2 = \{A_i\}$ is termed a* **bi-cut** *if the following holds:*

1. $\mathcal{S}(A_i) \cap \mathcal{S}(A_j) = \emptyset \quad$ *for* $i \neq j$;
2. $\mathcal{C}_2$ *contains a cut of* $DT(F_{\overline{z}})$ *and of* $DT(F_z)$.

Example 5.6. *Consider a function F such that $F_{\overline{z}} = (x_1 + x_2)x_4$ and $F_z = (x_1 + x_2 + x_3)x_5$ as in Figure 5.2 (we present a non-normal decomposition tree for improved readability). A possible bi-cut for such function is $\mathcal{C}_2 = \{x_1 + x_2, x_3, x_4, x_5\}$. Note that the set $\mathcal{C} = \{x_1 + x_2 + x_3, x_4, x_5\}$ is not a bi-cut since it does not contain a cut of $DT(F_{\overline{z}})$.*

From Lemma 5.8, $Max(DT(F_{\overline{z}}), DT(F_z))$ is a bi-cut. It is also straightforward to verify that $Max(DT(F_{\overline{z}}), DT(F_z))$ has minimum size among bi-cuts. We now show that bi-cuts have a one-to-one correspondence to decompositions. These facts will be enough to prove Theorem 5.6.

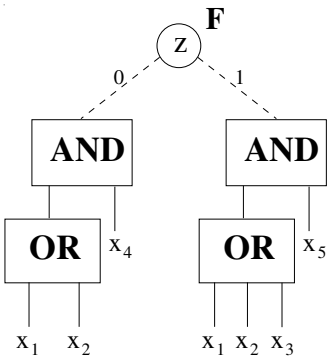

Figure 5.2: Function for Example 5.6.

Lemma 5.9. *Let M denote any function dividing F, such that $F/M = \{z, A_2, \cdots, A_m\}$. Then, the subset $\mathcal{C}_2 = \{A_2, \cdots, A_m\}$ is a bi-cut of F w.r.t. z. Conversely, for each bi-cut $\mathcal{C}_2$ there exists a function M such that $F/M = \{z\} \cup \mathcal{C}_2$.*

Proof. Eq. 5.19 shows that $\mathcal{C}_2$ contains a cut of $DT(F_{\overline{z}})$ and of $DT(F_z)$. The functions A_i are all disjoint-support, and each of them appears in at least one of $DT(F_{\overline{z}}), DT(F_z)$ (or else F would be independent from the variables in $\mathcal{S}(A_i)$). We also need to show, however, that each A_i has uniform support. To this end, suppose, for the sake of contradiction, that the support of one of the functions (say, $\mathcal{S}(A_2)$) is not uniform. It is not restrictive to assume that A_2 appears in $DT(F_{\overline{z}})$. Then $\mathcal{S}(A_2) \subseteq \mathcal{S}(F_{\overline{z}})$. Since we take A_2 to be not uniform, it must be

$$\mathcal{S}(A_2) \cap \mathcal{S}(F_z) \neq \emptyset \tag{5.27}$$

otherwise A_2 would be uniform by condition 1 of the definition of uniform-support; and

$$\mathcal{S}(A_2) \cap \overline{\mathcal{S}(F_z)} \neq \emptyset \tag{5.28}$$

otherwise A_2 would be uniform by condition 2. Let $\mathcal{C}$ indicate a subset of $\mathcal{C}_2$ forming a cut of

$DT(F_z)$; it follows that

$$\mathcal{S}(\mathcal{C}) = \bigcup_{A_i \in \mathcal{C}} \mathcal{S}(A_i) = \mathcal{S}(F_z); \tag{5.29}$$

The last equality being valid by definition of cut.

From Eq. 5.28, if $A_2 \in \mathcal{C}$, then $\mathcal{S}(\mathcal{C}) \cap \overline{\mathcal{S}(F_z)} \neq \emptyset$, contradicting Eq. 5.29. Hence, A_2 cannot belong to $\mathcal{C}$. We now show that A_2 cannot be left out of $\mathcal{C}$ either: From Eq. 5.27, there is a variable x in $\mathcal{S}(A_2) \cap \mathcal{S}(F_z)$. Since all functions of $\mathcal{C}_2$ are disjoint-support, x cannot be in the support of any other function of the bi-cut $\mathcal{C}_2$. Hence, if A_2 is left out of $\mathcal{C}$, $x \notin \mathcal{S}(\mathcal{C})$ and $\mathcal{C}$ is not a cut of $DT(F_z)$. In summary, A_2 could not be in a cut of $DT(F_z)$, but it could not be left out, a contradiction. Hence, A_2 must have uniform support and $\mathcal{C}_2$ is a bi-cut of F w.r.t. z.

We now show that for any given bi-cut $\mathcal{C}_2$ we can construct a decomposition of F. Consider the subset $\mathcal{C}_0 = \{A_2, \cdots, A_{c_0}\}$ of $\mathcal{C}_2$ forming a cut of $F_{\overline{z}}$. From Lemma 4.7, there exists a function $L_0(y_2, \cdots, y_{c_0})$ such that $F_{\overline{z}}/L_0 = \mathcal{C}_0$. Let also $\mathcal{C}_1 = \{A_{c_1}, \cdots, A_m\}$ denote the subset of $\mathcal{C}_2$ forming a cut of $DT(F_z)$. There exists then a function $L_1(y_{c_1}, \cdots, y_m)$ such that $F_z/L_1 = \mathcal{C}_1$. It is then easy to verify that the function $L(y_1, \cdots, y_m) = \overline{y_1} L_0(y_2, \cdots, y_{c_0}) + y_1 L_1(y_{c_1}, \cdots, y_m)$ satisfies

$$L(z, A_2, \cdots, A_m) = F, \tag{5.30}$$

that is, we have constructed a decomposition of F from $\mathcal{C}_2$. □

Finally, the proof of Theorem 5.6 follows:

Proof. - Theorem (5.6) - From Lemma 5.9, a function L can be found such that $F/L = \{z\} \cup Max(F_{\overline{z}}, F_z)$. Then, from Theorem 4.2, F/K_F cannot contain more elements than $\{z\} \cup Max(F_{\overline{z}}, F_z)$. Since $Max(F_{\overline{z}}, F_z)$ is a bi-cut of minimum size, F/K_F cannot contain fewer elements either, and consequently F/K_F and F/L must have the same size. In this case, however, from Theorems 4.2 and 4.4, K_F must be NP-equivalent to L and F/K_F must coincide with $\{z\} \cup Max(F_{\overline{z}}, F_z)$, modulo NP-equivalence. □

5.6 Putting it all together: The DSD procedure

We detail now the decomposition procedure. This description sets the stage for the complexity analysis presented in Section 5.7.

The algorithm traverses the nodes of the BDD of F in a bottom-up fashion. During the sweep, each node is inspected, and the decomposition tree of the function rooted at this node is determined from the decomposition of its cofactors and the top variable using the results presented earlier in this chapter.

The BDD node is then labeled with a *signed* – see Section 4.4.3 – pointer (`DEC *`) to the root of its decomposition tree.

```
void decompose_node(BDD* node) {
   node = NodeRegular(node);
   if (node->dec != NULL) return;
   var z = node->topVar;
   BDD *cof0 = node->cofactor0;
   BDD *cof1 = node->cofactor1;
   decompose_node(cof0);
   decompose_node(cof1);
   DEC *dec0 = GetDecomposition(cof0);
   DEC *dec1 = GetDecomposition(cof1);
   DEC *res = decompose(z,dec0, dec1);
   node->dec = res;
   return
}
```

The function `GetDecomposition` simply extracts the `DEC` pointer from a BDD node, and complements it if the BDD node was complemented. The call to `decompose` is the decomposition procedure proper:

```
DEC *decompose(var z, DEC* dec0, DEC* dec1) {
   res = decompose_INHERITED(z, dec0, dec1); // cases 1 2 3
   if (res) return(res);
   res = decompose_NEW(z, dec0, dec1);
   return(res);
}
```

We attempt the decomposition as an inherited or new decomposition. Each subroutine then considers all the corresponding cases from the previous section.

A `DEC` node contains a `.type` field and an `.actuals` list. The type field has four possible values: `VAR` (for simple variables), `OR`, `XOR` and `PRIME`; and it represents the decomposition type of the function rooted at that node. The actuals list is a list of signed pointers to BDD nodes. Each pointer represents a function in F/K_F.

It is worth noting that `decompose_INHERITED`, `decompose_NEW` are just switches, activating other procedures. In addition, since we must succeed with at at least one type of decomposition, the return value of `decompose` is guaranteed to be non-null. Finally, when two or more cases require a similar analysis, we group them in the same procedure so that portion of the computation can be shared; this is especially exploited in building inherited decompositions:

```
DEC* decompose_INHERITED(var z, DEC* dec0, DEC* dec1) {
   // case 1.b 2.b 3.b for AND/OR dec.
   res = decompose_INHERITED_OR_123.b(z, dec0, dec1);
   if (res) return(res);
   // case 1.b 2.b for XOR dec.
   res = decompose_INHERITED_XOR_12.b(z, dec0, dec1);
   if (res) return(res);
   //case 1.a 2.a 3.a
   res = decompose_INHERITED_PRIME_1.a(z, dec0, dec1);
```

```
    if (res) return(res);
    res = decompose_INHERITED_PRIME_2.a(z, dec0, dec1);
    if (res) return(res);
    res = decompose_INHERITED_PRIME_3.a(z, dec0, dec1);
    return(res);
}

DEC* decompose_NEW(var z, DEC* dec0, DEC* dec1) {
    res = decompose_NEW_OR(z, dec0, dec1); //case 4.a
    if (res) return(res);
    res = decompose_NEW_XOR(z, dec0, dec1); //case 4.b
    if (res) return(res);
    res = decompose_NEW_PRIME(z, dec0, dec1); //case 4.c
    return(res);
}
```

Since the maximal decomposition is unique, the calling order of the various subprocedures is irrelevant; with the following exception: since we only detect a new *PRIME* decomposition by failing all other cases, the procedure that builds a new *PRIME* decomposition, `decompose_NEW_PRIME`, must be kept last. In practice, we exploit this level of freedom by ordering the procedures based on the amount of analysis that they require, the fastest ones first; and disregarding even the grouping of new decompositions and inherited ones. For instance, Cases 4.a and 4.b are the fastest, and our implementation of `decompose_node` executes first of all `decompose_NEW_OR` and `decompose_NEW_XOR`.

In the reminder of this section, we will not discuss complement edges for BDD and DEC nodes any further. In particular, the segments of pseudo-code consider only nodes with positive polarity for simplicity, the extensions to include also nodes with negative polarity being straightforward.

We now analyze the subprocedures of `decompose` in detail.

5.6.1 Inherited decompositions

OR decompositions

`decompose_INHERITED_OR_123.b` groups the constructions described in Sections 5.2, 5.3 and 5.4 for identifying OR decompositions. For all of the three cases, we need to consider the actuals lists of the two cofactors and identify the common elements, which will be part of the resulting actuals list. To this list, we need to add a new element obtained by calling the second prototype of `decompose_node` with the node's top variable and the reminder *OR* decompositions as cofactors. Notice that this new element must be the first element of the resulting actuals list, based on the definition of normal Decomposition Tree from Section 4.4.3.

This procedure is successful as long as at least one of the two cofactor has an *OR* decomposition and there is at least one element in common between the actuals lists of F_0 and F_1. If, the actuals list of one cofactor is a proper subset of the other, then we have a Case 2.b decomposition. Otherwise we have a Case 1.b or 3.b decomposition.

Moreover, if one of the cofactors does not have a *OR* decomposition, for the purpose of this analysis, we consider its actuals list to have only one element, the cofactor function itself: Lemma 5.4 shows how to treat this situation in its special case of $k = 2$.

```
DEC* decompose_INHERITED_OR_123.b(var z, DEC* dec0, DEC* dec1) {
   DEC* res, dec0_residue, dec1_residue;
   list  common = list_intersect(dec0->actuals, dec1->actuals);
   if ( list_size(common) > 0 &&
       dec0->type == dec1->type == OR ) {
       dec0_residue = buildDecNode( OR,  dec0->actuals - common);
       if ( list_size(dec0_residue->actuals) == 0)
            dec0_residue = CONST_0;
       if ( list_size(dec0_residue->actuals) == 1)
            dec0_residue = getFirst(dec0_residue->actuals);
```

```
        // equivalently for right_residue
        G = decompose(z, dec0_residue, dec1_residue);
        res = buildDecNode(OR, { G, common}); //constructs node
        return res;
    }
    else if (list_intersect(dec0->actuals, dec1) ||
             list_intersect(dec1->actuals, dec0) )
        // build resulting decomposition
        // similar to above case
    }
    else return 0;
}
```

XOR decompositions

Inherited *XOR* decompositions can arise only from Cases 1.b and 2.b of Section 5.2.

Similarly to what has been discussed in the previous section, we need once again to check that at least one of the two cofactors is an *XOR* decomposition and that there is at least one element in common between the two actuals lists. The rest of the construction corresponds to the one for inherited *OR* decompositions.

PRIME decompositions

The first type of inherited *PRIME* decomposition is Case 1.a. The conditions for that case require that the two cofactors be both *PRIME* decompositions, the actuals lists differ in exactly one element and the cofactors w.r.t. those two elements match.

Example 5.7. *Consider again the function of Example 4.7 and assume that the top variable in its*

BDD representation was g. We consider available the decompositions of the cofactors w.r.t. g:

$$\begin{aligned} F_0 &= MAJORITY(G,H,i); \\ & G = a \oplus b; \\ & H = L + e; \\ & L = cd; \\ F_1 &= MAJORITY(G,H,N); \\ & N = ITE(f,h,i); \end{aligned}$$

Both F_0 and F_1 are decomposed by the same PRIME function MAJORITY. Their actuals lists are (G,H,h) and (G,H,N), respectively. They differ in exactly one element, namely, N instead of h.

We then check if Eq. 5.3 or 5.4 holds. This check can be carried out by computing $F_0(i=0)$, $F_0(i=1)$, $F_1(N=0)$, $F_1(N=1)$, and verifying that $F_0(i=0) = F_1(N=0)$, $F_0(i=1) = F_1(N=1)$. We then form a representation of the function $I = g'i + gMUX(f,h,i)$ and construct the decomposition of F as $MAJORITY(G,H,I)$. Note that, unless the decomposition of I is already known, we need to build that, too using the second prototype of `decompose_node`.

The following pseudocode checks if Eq. 5.3 or 5.4 hold. It returns the decomposition of F if the tests are successful:

```
DEC* decompose_INHERITED_PRIME_1.a (var z, DEC* dec0, DEC* dec1) {
   DEC* res;
   BDD* left_el, right_el, l0, r0;
   if (dec0->type != dec1->type != PRIME) return 0;
   if (list_size(dec0->actuals) != list_size(dec1->actuals))
    return 0;
   common = list_intersect(dec0->actuals, dec1->actuals);
   if (list_size(common) != size(dec0->actuals) -1) return 0;
```

```
    // the two functions differ in exactly one argument
    left_el = dec0->actuals - common;
    right_el = dec1->actuals - common;
    l0 = cofactor(dec0, left_el, 0);
    r0 = cofactor(dec1, right_el, 0);
    // compute also l1 and r1

    if ( ((l0 == r0) && (l1 == r1)) ||
         ((l1 == r0) && (l0 == r1))  ) {
       G = decompose (z, left_el->dec, right_el->dec);
       res = buildDecNode( PRIME, { G, common } );
       return res;
     }
     else return 0;
}
```

Case 2.a has a more complex set of comparisons. As the reader may recall from Section 5.3, Lemma 5.3 does not indicate precisely which is the function G to use to cofactor F_1. Instead we have a pool of candidates which are all the functions $A_i \in F_1/K_{F_1}$ such that $\mathcal{S}(A_i) \cap \mathcal{S}(F_0) = \emptyset$.

Thus we can detect such decomposition by considering the generalized cofactors (see Definition 2.3) of F_1 with respect to a subset of its actuals list elements and compare the result with F_0 to check if there is an element that satisfies the condition 2 of the Lemma.

It is important to note that each of these cofactor operations have complexity that it is only linear in the size of the BDD of F_1 (instead of quadratic). The reason for this simplified operation lies in the fact that the functions that we use in the cofactor operation are one in the decomposition of the other. To see this, consider a function $F = L(G,\ldots)$ and suppose we want to compute the cofactor w.r.t. $G = 1$. Then, $F_{G=1} = K_F(1,\ldots)$. To compute the last expression, we just need to consider any

combination of inputs of G such that $G = 1$, for instance a cube that satisfies G. We can then take the cofactor of F w.r.t. this cube to obtain our result, which is a linear time operation.

In general, we need to identify all the candidate $A_i \in F_1/K_{F_1}$ functions, and for each of those compute two generalized cofactors: $F_1(A_i = 0)$ and $F_1(A_i = 1)$ until we find a match. In the worst case, this entails the computation of $2 \times n$ cofactors, where n is the number of candidate elements.

Example 5.8. *Consider the functions $F_{\overline{z}} = ITE(A, CD, B + C), F_z = CD$. The actuals list of $F_{\overline{z}}$ contains A, B, C, D, of which only A and B are disjoint support from F_z.*

We observe that by assigning $B = 1$, however, $F_{\overline{z}} = A + CD \neq F_z$, and that assigning $B = 0$ results in $F_{\overline{z}} = C(A + D) \neq F_z$. The function B is then discarded. Assigning $A = 1$ instead results in $F_{\overline{z}} = ITE(1, CD, B + C) = CD = F_z$. A new function $Z = A + z$ is constructed, and F is decomposed as $ITE(Z, CD, B + C)$.

The following pseudocode reflects the observations above:

```
DEC* decompose_INHERITED_PRIME_2.a (var z, DEC* dec0, DEC* dec1) {
   DEC* res;
   BDD* l0,l1;
   tree_tag(dec1->actuals);
   // find the untagged elements in the left actuals list
   tryset = list_untagged(dec0->actuals);
   foreach(BDD* argument in tryset) {
      l1 = cofactor(dec0, argument, 1);
      l0 = cofactor(dec0, argument, 0);
      if ( l1 == dec1 ) {
         G = decompose (z, argument->dec, CONST_1);
         list actuals = dec0->actuals - argument + G;
         res = buildDecNode( PRIME, actuals );
         return res;
```

```
      } else if ( l0 == dec1 ) {
          // similar to above.
      }
   }
   // if unsuccessful, repeat by labeling the left tree
}
```

Case 3.a can be carried out analogously to case 1.a, with the difference that now instead of checking that the lists differ in exactly one element, we expect them to be identical. Once again the candidate function H with reference to Lemma 5.5 can be any of the actuals list elements.

5.6.2 New decompositions

OR and *XOR* decompositions

`decompose_NEW_OR` and `decompose_NEW_XOR` implement the checks of Sections 5.5 and 5.5. In the general case we create a new decomposition tree node of type `OR` or `XOR` and with an actuals list of length 2. However, note that it is possible that the non-constant cofactor has already a decomposition of the same type. If we detect this situation, the decomposition node will have an actuals list that is the same of its cofactor with the new element z prepended.

PRIME decompositions

In order to implement the construction of a new *PRIME* decomposition, we need to construct the set $Max(F_0, F_1)$ as shown in Theorem 5.6.

Construction of $Max(G,H)$

This operation allows us to find the set of maximal uniform support functions of two functions G, H whose decomposition is known. We show now how to construct a decomposition tree whose

root node has as its actuals list precisely the set of functions $Max(G,H)$. We call this tree also $Max(G,H)$.

Given two normal Decomposition Trees DT_G and DT_H, representing the decomposition of two functions G and H, respectively, the tree $Max(G,H)$ is the tree obtained as follows:

1. $Max(G,H)$ contains each node appearing in both DT_G and DT_H;

2. $Max(G,H)$ contains each arc appearing in both DT_G and DT_H;

3. if a node N of DT_G represents a function F_N, such that $\mathcal{S}(F_N) \cap \mathcal{S}(H) = \emptyset$, then the tree rooted at N belongs to $Max(G,H)$. Similarly for nodes of DT_H.

4. there is a node N labeled OR (XOR) for each pair of nodes $N_G \in DT_G, N_H \in DT_H$ labeled OR (XOR) and such that $\mathcal{S}(F_{N_1}) \cap \mathcal{S}(F_{N_2}) \neq \emptyset$. The actuals of N are the actuals common to N_G and N_H. The node N is suppressed if it has fewer than two actuals.

5. a root node is added. There is an arc from the root node to each node with no ancestors.

The construction above takes trivially time linear in the size of the two trees.

Example 5.9. *Figure 5.3 illustrates two decomposition trees DT_G and DT_H and the construction of $Max(G,H)$. In the graph we represent AND nodes as* `AND` *instead of complemented* `OR` *only for readability.*

The node `OR` *and node l belong to the intersection by rule 3. The tree rooted at* `PRIME` *by rules 1 and 2. The two nodes* `AND` *follow rule 4 producing the* `AND` *in the $Max(G,H)$ tree.*

To build a new *PRIME* decomposition, we simply need to build the $Max(F_0,F_1)$ tree and label the root node with type `PRIME`.

Example 5.10. *Consider the case $F_0 = ITE(abc, d+e+f, g \oplus h)$, $F_1 = ITE(ab, e+f+g, h \oplus c)$.*

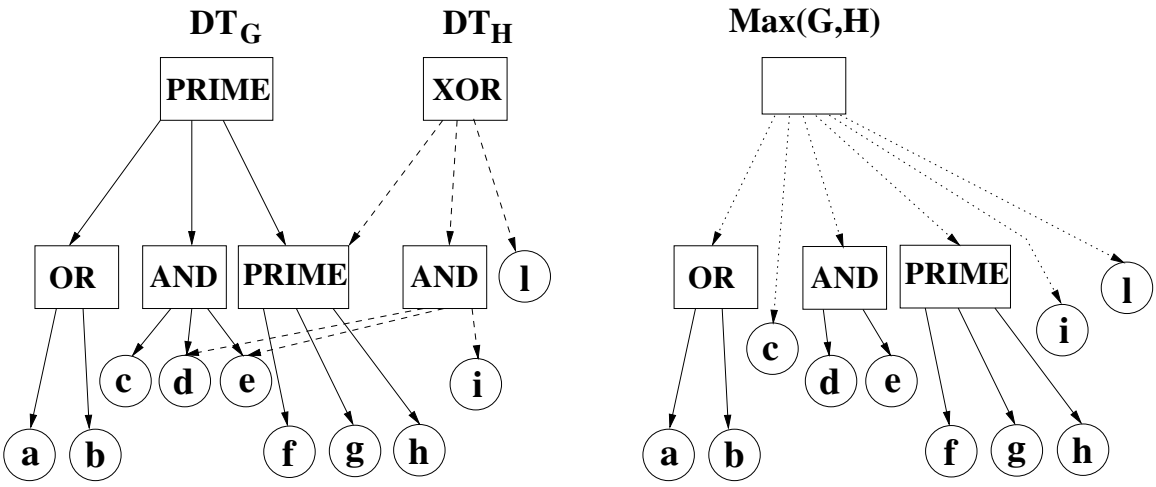

Figure 5.3: Two functions and the construction of their $Max(G,H)$ tree.

The set $Max(F_0,F_1)$ *is given by:*

$$\begin{aligned} A &= ab; \\ E &= e+f; \\ Max(F_0,F_1) &= \{A,E,c,d,g,h\}. \end{aligned}$$

Thus, the decomposition of $F = \overline{z}F_0 + zF_1$ *is given by* $F = K_F(z,A,E,c,d,g,h)$.

5.7 Complexity analysis and considerations

This section analyzes the complexity of the algorithm, given a function F whose BDD representation has **#BDD** nodes and whose support $|\mathcal{S}_F|$ has **#VAR** variables.

Notice, first of all, that the length of any actuals list in DT_F is bound by the number of variables in the support of the function, $|F/K_F| \leq |\mathcal{S}_F|$. We now analyze the complexity of each procedure in `decompose`.

The new decomposition procedures `decompose_NEW_OR` and `decompose_NEW_XOR` require only constant time operations: $O(k)$. `decompose_NEW_PRIME` requires only building the set

$Max(F_0, F_1)$. As pointed out previously, the complexity of this operation is linear in the size of the decomposition trees involved. The number of nodes in a decomposition tree is bound by #**VAR**; thus the complexity for this procedure is $O(\#\mathbf{VAR})$.

Inherited decomposition procedures involve recursive calls to `decompose`. The inherited procedures for *OR* and *XOR* decompositions require intersecting two actuals lists, operation linear in their length, and performing a recursive decomposition call. Note that, at each recursive call, the support of the function to decompose has at least one fewer variable, since the `common` portion of the final actuals list must have at least a support of size one. In conclusion, for these two procedures, we can write a recursive equation of their complexity: $O(|\mathcal{S}_F|) = O(\#\mathbf{VAR}) + O(|\mathcal{S}_F| - 1)$.

`decompose_INHERITED_PRIME_1.a` has a similar treatment, with two differences: 1) In addition of intersection the actuals lists, we need to compute also 4 generalized cofactors. As we showed in Section 5.6.1, these are special cofactors operations whose complexity is linear with the size of the BDDs involved. 2) At each recursive step, the support of the function to decompose now has at least two fewer variables, since we are dealing with `PRIME` nodes which have at least three inputs. The recursive operation for this procedure is thus: $O(|\mathcal{S}_F|) = O(\#\mathbf{VAR}) + 4 \cdot O(\#\mathbf{BDD}) + O(|\mathcal{S}_F| - 2)$.

`decompose_INHERITED_PRIME_2.a` and `decompose_INHERITED_PRIME_3.a` require a list intersection, a number of cofactors operations, up to twice the length of the actuals lists and a recursive call to `decompose`. However, in this case the call is guaranteed to be terminated by a new *OR* decomposition whose complexity, as we saw, is constant: $O(\#\mathbf{VAR}) + 2 \cdot O(\#\mathbf{BDD} \cdot \#\mathbf{VAR}) + O(k)$.

By solving the recursive equation of `decompose_INHERITED_PRIME_1.a`, we obtain a complexity of $O(\#\mathbf{BDD} \cdot \#\mathbf{VAR})$, which cannot be made worse even by terminating any of the recursive steps. with a `decompose_INHERITED_PRIME_3.a` call. Thus this is also the worst complexity of `decompose`.

Since we need to call this procedure for each BDD node in the representation of F, the overall complexity of our algorithm is: $O(\#\mathbf{BDD}^2 \cdot \#\mathbf{VAR})$.

Previously known algorithms – see Section 4.2 – had exponential complexity in the size of $\mathcal{S}_F$ and would compute only one of the many decompositions of a function. The complexity of our algorithm is dominated by the size of the BDD that represents the function F, not by the number of variables in its support. Moreover, it has the advantage of computing the finest granularity decomposition, from which all others can be derived.

For those functions whose BDD representation has size exponential in the number of the input variables, our algorithm has no better complexity than previously known ones. However, it is known that most functions representing digital circuit have corresponding BDDs whose size is much more compact and thus it is possible to build such BDDs even for some very large functions. Using our algorithm it is practically always possible to find the maximal disjunctive decomposition of a function, once a BDD has been built.

5.8 Experiments on the decomposability of industrial testbenches

The algorithm described in this chapter was implemented in a C++ program and tested on the circuits from the Logic Synthesis Benchmarks suite [68] and the ISCAS'89 Benchmark Circuits [16], including their 1993 additions. We report results on all the testbenches of the two suites. The testbenches are grouped by benchmark suite and by group within the suite: the Logic Synthesis suite includes two-level combinational circuits, multi-level combinational networks, sequential circuits and the tests added in '93. The ISCAS '89 suite includes a set of core sequential testbenches and additional circuits from '93. For all the sequential circuits, we considered only the combinational portion of the tests, we created an additional primary output for each latch input net and an additional primary input for each latch output. For each testbench, we first built the ROBDDs representing each output node as a function of the primary inputs, and then we attempted the decomposition of this functions.

The decomposition results are reported in 5.1. Next to the testbench's name we indicate how many of the output functions we could decompose: Output corresponds to the number of outputs of

the circuit, DEC reports how many of this output functions have a disjoint support decomposition. Output functions that are constant or a copy of a single input signal are considered decomposable. When not all of the outputs could be decomposed, we also looked at the non-decomposable outputs and checked if any of the two cofactors w.r.t. the top variable were decomposable. Column Dec Cof reports in how many cases at least one of the two cofactors resulted decomposable. We report a "-" in this column when all the outputs of the circuits were decomposable and thus the decomposability of the cofactors is not meaningful. By just glancing at the table, it's easy to notice that the column Dec Cof has a - for most of the circuits, meaning all of the outputs for that testbench are found to be decomposable.

Often, if a function is not decomposable, its cofactors are, and thus it is still possible to obtain a representation that has almost all the advantages and properties of disjoint support decompositions, except for a non-disjoint multiplexer corresponding to the node with the top variable of the specific BDD. Notice that even fairly big functions have a disjoint decomposition in most cases. For visual reference to the more complex testbenches, the table reports in boldface those circuits whose BDD construction, before starting the decompostion, required building more than 10,000 nodes.

The following two columns report the number of inputs of the circuit (Inputs) and the maximum number of inputs to any block in the decomposition tree of the output functions for that testbench (FanIn). It is worth noticing that in many cases, even the most complex, decomposition can reduce considerably the largest fanin to any block in the network's representation, while keeping each block disjoint support from the others. Then we indicated the total number of blocks in the normal decomposition trees. These latter two values are helpful in giving an indication of the amount of partitioning possible in the routing of the benchmark circuit.

The last four columns provide performance information. The first time/memory pair reports the time in seconds and the amount of memory in kilobytes required to produce the ROBDDs of all the output functions of a testbench. The second pair indicates the *additional* time and memory required to construct the normal decomposition trees from the ROBDDs. All the experiments were run on a Linux PC equipped with a Pentium 4 processor running at 2.7Ghz and 2GB of memory

and 512Kb of cache. In running the tests, we used a proprietary ROBDD package. In particular, our ROBDD package records the support of the functions associated to each ROBDD node. While this feature is convenient because of the number of support operations and tests we need to perform, its efficiency could be optimized. Moreover, our decomposition package has also room for implementation improvements.

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
Logic Synthesis '91 - Two level tests										
5xp1	10	9	0	7	7	20	0.00	13	0.00	1
9sym	1	0	0	9	9	1	0.00	59	0.00	1
alu4	8	1	0	14	14	10	0.04	417	0.01	22
apex1	45	43	0	45	30	224	0.02	373	0.02	37
apex2	3	3	-	39	29	16	0.17	1384	0.02	22
apex3	50	39	2	54	42	200	0.01	399	0.02	23
apex4	19	5	0	9	9	22	0.01	384	0.01	23
apex5	88	88	-	117	14	463	0.03	377	0.01	59
bw	28	15	4	5	5	57	0.00	11	0.00	3
clip	5	0	2	9	9	5	0.00	62	0.00	3
con1	2	0	2	7	6	2	0.00	2	0.00	0
duke2	29	24	3	22	17	91	0.00	108	0.00	13
e64	65	65	-	65	2	2080	0.01	83	0.00	5
misex1	7	1	0	8	7	8	0.00	4	0.00	1
misex2	18	17	1	25	7	105	0.00	10	0.00	3
misex3c	14	2	5	14	14	20	0.01	186	0.00	11
misex3	14	2	1	14	14	16	0.07	410	0.00	15
o64	1	1	-	130	2	129	0.00	85	0.00	8
rd53	3	1	1	5	5	6	0.00	7	0.00	0
rd73	3	1	0	7	7	8	0.00	41	0.00	1
rd84	4	2	0	8	8	16	0.01	84	0.00	1
sao2	4	4	-	10	8	12	0.00	41	0.00	2
seq	35	35	-	41	33	198	0.08	385	0.02	41
vg2	8	8	-	25	24	23	0.00	95	0.00	4
xor5	1	1	-	5	2	4	0.00	5	0.00	0
Logic Synthesis '91 - FSM tests										
daio	6	5	1	6	3	4	0.00	1	0.00	0
ex1	39	39	-	30	23	677	0.00	67	0.01	54
ex2	21	21	-	22	18	340	0.00	37	0.00	8
ex3	12	12	-	13	10	98	0.00	10	0.00	9
ex4	23	23	-	21	8	283	0.00	14	0.00	5
ex5	11	11	-	12	10	78	0.00	10	0.00	4
ex6	16	12	4	14	13	58	0.00	15	0.00	8
ex7	12	12	-	13	11	97	0.00	13	0.00	3
s1196	32	24	6	33	21	68	0.01	59	0.00	34
s1238	32	24	6	33	21	68	0.01	64	0.00	35
s1423	79	77	2	92	32	330	0.01	376	0.06	348

Table 5.1: Disjoint Support Decomposition results - *continued on next page*

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
s1488	25	23	2	15	12	59	0.01	77	0.00	11
s1494	25	23	2	15	12	59	0.01	77	0.00	11
s208	10	10	-	20	3	53	0.00	10	0.00	3
s27	4	4	-	8	2	11	0.00	1	0.00	0
s298	20	17	3	18	8	32	0.00	9	0.00	3
s344	26	23	0	25	7	37	0.00	11	0.00	3
s349	26	23	0	25	7	37	0.00	10	0.00	3
s382	27	27	-	25	7	70	0.00	11	0.00	5
s386	13	13	-	14	9	67	0.00	12	0.00	3
s400	27	27	-	25	7	70	0.00	11	0.00	4
s420	18	18	-	36	3	113	0.00	25	0.00	10
s444	27	27	-	25	7	70	0.00	25	0.00	6
s510	13	5	3	26	19	24	0.00	25	0.00	9
s526n	27	24	2	25	8	66	0.00	16	0.00	4
s526	27	24	3	25	8	66	0.00	15	0.00	4
s641	42	42	-	55	18	150	0.00	37	0.01	25
s713	42	42	-	55	18	150	0.00	42	0.01	28
s820	24	22	1	24	17	109	0.00	29	0.00	7
s832	24	22	1	24	17	109	0.00	30	0.00	7
s838	34	34	-	68	3	233	0.00	46	0.01	67
s953	52	47	2	46	17	97	0.01	54	0.00	13
Logic Synthesis '91 - Multi level tests										
9symml	1	0	0	9	9	1	0.00	37	0.00	1
alu2	6	4	0	10	10	8	0.00	78	0.00	5
alu4	8	4	0	14	14	14	0.01	199	0.00	11
apex6	99	99	-	135	14	369	0.00	77	0.00	24
apex7	37	37	-	49	9	155	0.00	44	0.00	13
b1	4	3	1	3	3	2	0.00	1	0.00	0
b9	21	21	-	41	8	54	0.00	15	0.00	4
C1355	32	0	0	41	41	32	0.23	1545	73.57	41689
C17	2	1	1	5	4	4	0.00	0	0.00	0
C1908	25	7	0	33	32	93	0.05	754	2.40	5787
C2670	140	119	1	233	78	187	0.05	666	1.36	8017
C3540	22	14	0	50	50	49	0.53	2301	2.05	16348
C432	7	1	1	36	36	23	0.01	329	0.07	489
C499	32	0	0	41	41	32	0.16	1406	100.61	40187
C5315	123	80	10	178	66	186	0.04	371	0.12	1032
C7552	108	107	1	207	118	295	0.18	1148	0.25	1899
C880	26	26	-	60	41	96	0.02	373	0.41	3374
c8	18	10	8	28	3	69	0.00	19	0.00	2
cc	20	20	-	21	4	32	0.00	7	0.00	2
cht	36	36	-	47	3	74	0.00	12	0.00	4
cm138a	8	8	-	6	2	40	0.00	1	0.00	1
cm150a	1	1	-	21	20	2	0.00	8	0.00	2
cm151a	2	2	-	12	11	2	0.00	3	0.00	1
cm152a	1	0	0	11	11	1	0.00	2	0.00	1
cm162a	5	5	-	14	4	19	0.00	5	0.00	1
cm163a	5	5	-	16	3	26	0.00	3	0.00	1

Table 5.1: Disjoint Support Decomposition results - *continued on next page*

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
cm42a	10	10	-	4	2	30	0.00	1	0.00	1
cm82a	3	3	-	5	3	6	0.00	2	0.00	1
cm85a	3	3	-	11	3	20	0.00	6	0.00	1
cmb	4	4	-	16	2	33	0.00	10	0.00	1
comp	3	3	-	32	3	63	0.00	24	0.00	19
count	16	16	-	35	3	168	0.00	7	0.00	2
cu	11	11	-	14	6	43	0.00	4	0.00	1
decod	16	16	-	5	2	64	0.00	2	0.00	1
des	245	245	-	256	14	560	0.07	373	0.02	77
example2	66	49	17	85	11	281	0.00	25	0.00	12
f51m	8	8	-	8	7	13	0.00	17	0.00	1
frg1	3	3	-	28	19	12	0.00	77	0.00	4
frg2	139	139	-	143	17	519	0.01	336	0.01	60
k2	45	43	2	45	30	224	0.01	353	0.02	38
lal	19	19	-	26	2	89	0.00	11	0.00	3
ldd	19	18	1	9	5	60	0.00	8	0.00	2
majority	1	1	-	5	4	2	0.00	1	0.00	0
mux	1	1	-	21	20	2	0.00	13	0.00	2
my_adder	17	17	-	33	3	48	0.00	67	0.00	9
pair	137	137	-	173	28	724	0.03	374	0.06	275
parity	1	1	-	16	2	15	0.00	5	0.00	1
pcler8	17	17	-	27	3	99	0.00	12	0.00	4
pcle	9	9	-	19	3	62	0.00	5	0.00	2
pm1	13	13	-	16	3	46	0.00	5	0.00	1
rot	107	104	3	135	42	351	0.04	621	0.25	2114
sct	15	14	1	19	3	63	0.00	11	0.00	2
tcon	16	8	8	17	3	8	0.00	2	0.00	1
term1	10	10	-	34	10	66	0.00	69	0.00	5
too_large	3	3	-	38	29	16	0.06	587	0.01	16
ttt2	21	18	2	24	8	66	0.00	31	0.00	3
unreg	16	16	-	36	3	48	0.00	7	0.00	3
vda	39	29	10	17	17	81	0.00	121	0.01	15
x1	35	35	-	51	17	181	0.01	192	0.00	16
x2	7	7	-	10	6	24	0.00	4	0.00	1
x3	99	99	-	135	14	369	0.01	121	0.00	22
x4	71	71	-	94	8	207	0.00	48	0.00	13
z4ml	4	4	-	7	3	9	0.00	14	0.00	1
Logic Synthesis '91 - Addition '93										
b12	9	8	0	15	8	31	0.01	58	0.00	2
bigkey	421	194	3	487	10	232	0.19	371	0.02	83
clma	115	115	-	416	36	532	24.25	373	0.01	38
cordic	2	2	-	23	8	18	0.09	349	0.00	3
cps	109	109	-	24	15	1147	0.03	210	0.01	66
dalu	16	15	1	75	31	227	0.12	373	0.02	33
dsip	421	194	3	453	12	232	0.18	372	0.03	85
ex4p	28	28	-	128	15	46	0.06	363	0.01	15
ex5p	63	54	2	8	8	271	0.04	195	0.00	8
i10	224	224	-	257	74	1098	0.64	3294	14.21	102333

Table 5.1: Disjoint Support Decomposition results - *continued on next page*

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
i1	13	13	-	25	3	43	0.00	4	0.00	2
i2	1	1	-	201	6	187	0.00	154	0.00	17
i3	6	6	-	132	2	126	0.00	28	0.00	8
i4	6	6	-	192	2	186	0.00	53	0.00	37
i5	66	66	-	133	2	132	0.01	19	0.00	7
i6	67	1	29	138	5	69	0.00	45	0.01	8
i7	67	3	64	199	6	72	0.01	64	0.00	12
i8	81	18	63	133	17	93	0.05	372	0.04	35
i9	63	0	0	88	13	63	0.01	93	0.01	58
mm4a	16	16	-	20	13	52	0.00	60	0.00	8
mm9a	36	36	-	40	28	117	0.03	386	0.26	2110
mm9b	35	35	-	39	29	293	0.03	384	0.35	1892
mult16a	17	17	-	34	3	64	0.00	61	0.00	7
mult16b	31	31	-	48	3	61	0.01	13	0.00	4
mult32a	33	33	-	66	3	128	0.04	381	0.01	105
s208	9	9	-	19	3	46	0.00	5	0.00	4
s38584	1730	1611	113	1465	36	5146	9.92	946	0.17	887
s5378	212	211	0	199	52	784	0.11	242	0.02	154
s838	33	33	-	67	3	562	0.01	33	0.00	65
s9234	174	169	5	172	40	509	0.10	280	0.01	59
sbc	83	83	-	68	21	404	0.02	110	0.01	44
sqrt8ml	4	4	-	8	5	11	0.00	11	0.00	1
sqrt8	4	4	-	8	7	7	0.00	11	0.00	1
squar5	8	4	2	5	5	14	0.00	8	0.00	1
t481	1	1	-	16	2	15	0.02	190	0.00	1
table3	14	0	2	14	14	14	0.01	176	0.02	152
table5	15	3	2	17	17	25	0.01	169	0.02	130
ISCAS '89 - FSM tests										
s1196	32	24	6	32	21	68	0.00	59	0.00	34
s1238	32	24	7	32	21	68	0.00	69	0.01	56
s13207.1	790	783	7	700	42	1805	1.01	371	0.03	97
s13207	790	783	7	700	42	1805	1.04	371	0.03	96
s1423	79	77	2	91	32	330	0.01	191	0.01	148
s1488	25	23	2	14	12	59	0.01	77	0.00	11
s1494	25	23	2	14	12	59	0.01	77	0.00	11
s15850.1	684	651	33	611	148	2074	1.9	1059	0.62	2606
s15850	684	651	33	611	148	2074	2.13	813	0.58	2349
s208	9	9	-	18	3	46	0.00	5	0.00	7
s27	4	4	-	7	2	11	0.00	1	0.00	0
s298	20	17	2	17	8	32	0.00	7	0.00	2
s344	26	23	0	24	7	37	0.00	10	0.00	3
s349	26	23	0	24	7	37	0.00	10	0.00	3
s35932	2048	2048	-	1763	6	3371	8.00	371	0.01	135
s382	27	27	-	24	7	70	0.00	10	0.00	3
s38584.1	1730	1611	113	1464	36	5146	12.53	801	0.16	959
s38584	1730	1611	113	1464	36	5146	11.32	824	0.14	912
s386	13	13	-	13	9	67	0.00	14	0.00	4
s400	27	27	-	24	7	70	0.00	11	0.00	5

Table 5.1: Disjoint Support Decomposition results - *continued on next page*

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
s420	17	17	-	34	3	154	0.00	16	0.00	11
s444	27	27	-	24	7	70	0.01	23	0.00	6
s510	13	5	5	25	19	24	0.00	23	0.00	4
s526n	27	24	2	24	8	66	0.01	15	0.00	4
s526	27	24	3	24	8	66	0.01	15	0.00	4
s5378	213	212	0	214	51	795	0.10	288	0.03	211
s641	42	42	-	54	18	150	0.00	54	0.01	55
s713	42	42	-	54	18	150	0.01	46	0.00	41
s820	24	22	1	23	17	109	0.00	30	0.00	7
s832	24	22	1	23	17	109	0.00	30	0.00	7
s838	33	33	-	66	3	562	0.00	37	0.00	100
s9234	250	245	4	247	48	707	0.42	372	0.05	250
s953	52	47	2	45	17	97	0.01	58	0.00	12
ISCAS '89 - Addition '93										
prolog	158	152	4	172	67	424	0.03	175	0.00	101
s1196	32	24	6	32	21	68	0.01	59	0.00	34
s1269	47	30	9	55	35	97	0.01	231	0.03	115
s1512	78	78	-	86	18	285	0.01	136	0.00	33
s3271	130	102	28	142	15	353	0.03	198	0.01	38
s3330	205	199	5	172	67	424	0.03	199	0.02	159
s3384	209	172	37	226	39	373	0.03	151	0.01	202
s344	26	23	0	24	7	37	0.00	10	0.00	3
s4863	88	66	2	153	22	190	1.96	4231	9.03	43400
s499	44	44	-	23	5	423	0.00	41	0.00	9
s635	33	33	-	34	2	591	0.00	16	0.00	4
s6669	269	194	44	322	16	380	0.92	2237	1.66	7848
s938	33	33	-	66	3	562	0.01	37	0.00	100
s967	52	47	2	45	17	97	0.01	59	0.00	11
s991	36	36	-	84	54	53	0.01	102	0.00	21

Table 5.1: Disjoint Support Decomposition results

In most cases the additional time to decompose a function is small compared to the time required to build the initial ROBDDs. However, there are a few cases where this is not the case: specifically *C1355* and *C499* of the Logic Synthesis suite cannot find a decomposition for any of the primary outputs, yet the algorithm is very time consuming. These circuits are very similar, they have the same number of inputs and outputs and they are both error correcting circuits as reported in [68]. By inspecting the two circuits we found that the intermediate nodes of these circuits up to about half way in the bottom-up construction were often decomposable; then the repetitive application of the algorithm `decompose_NEW_PRIME`, Section 5.6.2, made so that the top half of the construction produces almost invariably a PRIME decomposition with a kernel identical to the function itself.

Circuit *i10* from Logic Synthesis '91 - Addition '93 instead, requires times and memory resources above average because of the long actuals lists that are produced during the computation. Table 5.1 reports decomposition results for all the circuits in the test suites mentioned above with two exceptions: we could not apply the decomposition algorithm to circuit C6288 (a 16-bit multiplier) since we run out of memory building the initial ROBDDs for it; circuit s38417 runs out of memory during the decomposition because of its large support size and long intermediate actuals lists involved. We hope to be able to tackle this latter testbench with a more clever implementation of the decomposition algorithm. We summarized the results and found that we could decompose 16,472 functions out of a total of 18,584. The total time spent constructing the ROBDDs was 79.63s, while the time spent after that to attempt the functions' decompositions was 209.11s.

5.9 Conclusion

We presented in this chapter a novel algorithm that can generate the maximal disjoint support decomposition of a Boolean function represented by its BDD. The worst case complexity of this algorithm is only quadratic in the size of the BDD representation, while previously proposed algorithms has exponential complexity on the number of variables in the support of the function. We found it very fast in practice as we were able to obtain the decomposition of most testbenches in time comparable to the construction of their BDD. Experimental results indicate that the majority of functions representing the behavior of digital systems are indeed decomposable and the maximal disjoint decomposition has a fine granularity, as indicated by the support size of the biggest component block.

The next chapter will exploit these encouraging results in devising a new type of parameterization for symbolic simulation. This time the parameterization is exact, meaning that we generate a set of parameterized functions whose range matches exactly the frontier set of represented by the state vector.