# Chapter 6

# Exact Parameterizations for Symbolic Simulation

The theory of disjoint support decompositions provides important insights on the structure of a Boolean function and on the role and influence of each of its support variables. Moreover, the algorithm presented in the previous chapter allows us to take advantage of such insights very efficiently. We saw at the end of Chapter 3 that the parameterization technique of Cycle-Based Symbolic Simulation is computationally very efficient, but not exact. Often we may need to compromise by exploring only a subset of the possible set of states of the design under verification in order to maintain simulation efficiency. In some cases, this trade off produces simulation performance that is comparable to plain logic simulation in terms of vectors simulated per second.

Here we present a new parameterization technique that, by exploiting the disjoint decomposition properties of the functions in the state vector, can produce an exact parameterization, that is a new set of functions spanning the exact same state set as the original state vector. These new functions have smaller support than the original ones, and thus a simpler BDD representation.

In other words, if the parameterization of CBSS was building a function $\mathbf{PS}_{@\mathbf{k}}$ such that

$$\mathcal{R}(\mathbf{PS}_{@\mathbf{k}}) \subseteq \mathcal{R}(\mathbf{S}_{@\mathbf{k}}),$$

the algorithm presented in this chapter builds a parameterized vector function such that

$$\mathcal{R}(\mathbf{PS_{@k}}) = \mathcal{R}(\mathbf{S_{@k}}).$$

We call the new algorithm *DSD-based Symbolic Simulation*, (DSD-SS).

## 6.1 Re-encoding the state function

This new technique for improved scalability and robustness in symbolic simulation is similar to CBSS in that it inserts a parameterization phase in the feedback look of symbolic simulation as indicated in Figure 3.1. However, now we take a completely different approach to how we perform the parameterization.

In order to generate the parametric state vector for DSD-SS, at each step of simulation we start by generating the disjoint support decomposition representation for each of the component functions of the state vector. While each element of the vector has a tree decomposition with no reconvergence, as described in Chapter 4, it is now possible that two or more elements intersect at some intermediate node of their trees.

Figure 6.1 shows an example of a decomposed state vector for a small design with only four memory elements. The dashed line delimits the decomposition of component $s_1$ to show that each single component function is represented by a tree. We call this structure a **decomposition graph**.

The decomposed representation is generated dynamically during the simulation. We then use this representation to generate a parameterization of the state vector. The parameterization we propose is based on the observation that at each symbolic simulation step $k$, it is possible to substitute the state function $\mathbf{S_{@k}} : \mathcal{B}^{mk} \rightarrow \mathcal{B}^n$ with a new function $\mathbf{PS_{@k}}$ such that $\mathcal{R}(\mathbf{S_{@k}}) = \mathcal{R}(\mathbf{PS_{@k}})$ without affecting the results of the simulations; namely: 1) The set of outputs that can be generated by the circuit and 2) the set of states the circuit can reach at each cycle. If we can find a suitable function $\mathbf{PS_{@k}}$ that also has a smaller BDD representation (*i.e.*, fewer BDD nodes), then we can control the size of the Boolean expression and improve the performance of symbolic simulation.

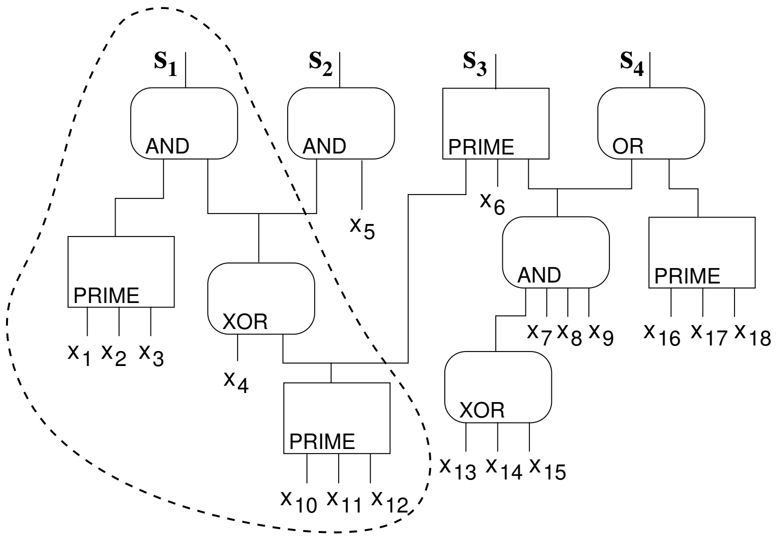

Figure 6.1: A decomposed state vector for a small design

The relationship between the set of states spanned by the new $\mathbf{PS}_{@\mathbf{k}}$ vector function versus the original state vector and the entire search space is reported in Figure 6.2. It is worth comparing it with the corresponding Figure 3.5 of the CBSS parameterization of Chapter 3.

In the following sections we present various transformations that we apply to the decomposition graph to accomplish the objective of producing an exact parameterization with a more compact representation than the original state vector. For each of these transformations, we show that the function vectors before and after the transformation span the same identical range. The first technique, called *reduction at free points*, is independent of the type of decomposition node it applies to. *Prime function elimination* is specific to *PRIME* nodes, while *non-dominant variable removal* refers to variable inputs that fan out to associative operators nodes.

In presenting the techniques, we will refer to the generic vector function $\mathbf{F}$ instead of $\mathbf{S}_{@\mathbf{k}}$ since such transformations can be applied to any Boolean vector function. Moreover, we will use the terms *decomposition graph* $\mathbf{F}$ and *function* $\mathbf{F}$ interchangeably to refer to the multiple output function $\mathbf{F}$.

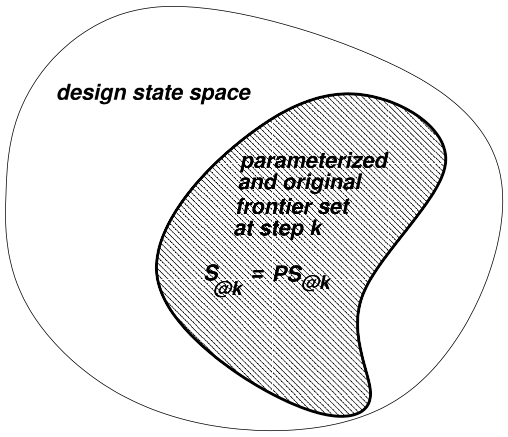

Figure 6.2: The parameterized frontier set $\mathbf{PS}_{\mathbf{@k}}$

## 6.2 Reduction at Free Points

The first transformation, called *reduction at free points*, aims to simplify the decomposition graph by finding nodes which constitute a single cut-point. In other words, the output of such nodes is only affected by a set of variables which don't influence any other portion of the graph.

We first provide the definition of a *free point* and we show an example transformation. Then we provide proof that the transformation does not affect the range of the vector function.

The following definition is also illustrated in Figure 6.3:

**Definition 6.1.** *A* **free point** *p in a decomposition graph of* $\mathbf{F}$ *is a function corresponding to an output of a block in the graph. It has the property that, if we substitute the sub-graph rooted at the point with a new input variable w, the new function* $\mathbf{G}$ *has disjoint support with the function rooted at p:*

$$\mathbf{F}(x_1,\cdots,x_m) = \mathbf{G}(w,x_{p+1},\cdots,x_m) \circ p(x_1,\cdots,x_p) \tag{6.1}$$

*and* $\mathcal{S}(\mathbf{G}) \cap \mathcal{S}(p) = \emptyset$*.*

Figure 6.3 shows three free points with darkened circles. Note that the output of $p$ is a free point since none of the variables in the support of $p$ appears in the support of other parts of the graph. On

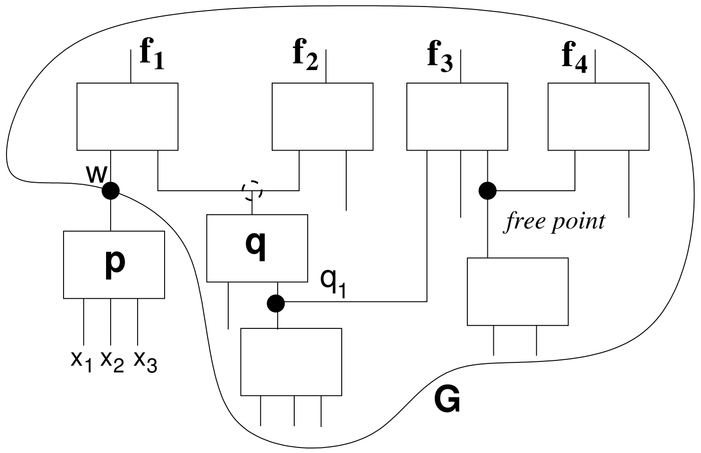

Figure 6.3: A vector function and its free points

the other hand, the dashed circle at the output of $q$ is not a free point since, if we split the graph at that node, the two functions obtained, $\mathbf{H}$ and $q$ with $\mathbf{F} = \mathbf{H} \circ q$, would still share the input $q_1$.

The following theorem shows that we can use free points to simplify the decomposition graph:

**Theorem 6.1.** *Given a decomposition graph for a multiple output Boolean function $\mathbf{F}(x_1, \cdots, x_m) : \mathcal{B}^m \to \mathcal{B}^n$, a free point $p(x_1, \cdots, x_p) : \mathcal{B}^p \to \mathcal{B}$ in it, and the function $\mathbf{G}(p, x_{p+1}, \cdots, x_m) : \mathcal{B}^{m-p+1} \to \mathcal{B}^n$, obtained by substituting the function $p()$ with the new input variable $p$ in the graph of $\mathbf{F}$, $\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{G})$.*

*Proof.* Consider the function $\mathbf{F}(x_1, \cdots x_m)$ and compute its range by splitting on the input variables [27]:

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{x_1=0}) \cup \mathcal{R}(\mathbf{F}_{x_1=1})$$

By applying this equation recursively over all the variables $(x_1, \cdots x_p)$ in the support of $p$, we obtain:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{(i_1, \cdots, i_p) \in \mathcal{B}^p} \mathcal{R}(\mathbf{F}_{x_1=i_1, x_2=i_2, \cdots, x_p=i_p}) \tag{6.2}$$

Using Equation 6.1:

$$\mathbf{F}_{x_1=i_1,x_2=i_2,\cdots,x_p=i_p} = \mathbf{G}_{p=i_w} \quad \text{where} \quad i_w = p(i_1,\cdots,i_p) \in \{0,1\}$$

since $p$ evaluates to a constant. Substituting in Eq. 6.2 we finally obtain:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{i_w \in \{0,1\}} \mathcal{R}(\mathbf{G}_{p=i_w}) = \mathcal{R}(\mathbf{G})$$

$\square$

Thus, we can substitute all the free points with new variables and generate a new state function **G** with a smaller representation.

A simple traversal of the graph is sufficient to discover all the free points with maximal support, that is, all the free points whose support is not contained in any other free point of the decomposition graph:

**Definition 6.2.** *A* **free point** $p()$ *is said to have* maximal support *if its support* $\mathcal{S}(p)$ *is not a proper subset of any other free point in the graph.*

The transformation of free sub-graphs with new variables produces a new function **G**, with $|\mathcal{S}(\mathbf{G})| \leq |\mathcal{S}(\mathbf{F})|$.

**Example 6.1.** *Consider the decomposition graph of Figure 6.4. Figure 6.4.a shows all the free points of the graph with filled circles. The free points surrounded by a dashed circle are also maximal and we can substitute the portion of the graph rooted at these nodes with a new parameter, without affecting the range of the graph. Figure 6.4.b shows the new, reduced graph obtained.*

Note that, anytime we perform a free point reduction we remove a set of input variables from the support of the vector function **F**. Thus, we can reassign any of these variables from a combinational input variable role to a parameter variable role and use it as the parameter assigned to the free point.

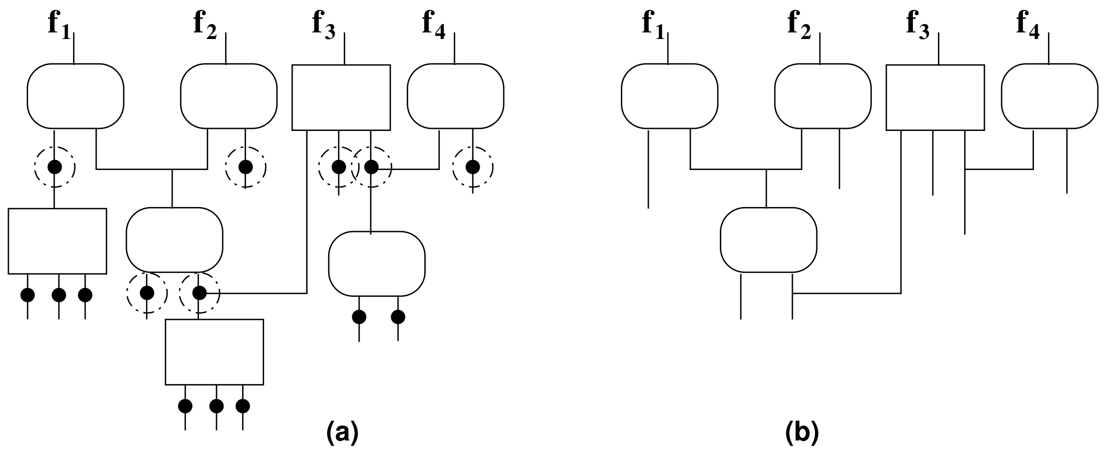

Figure 6.4: Free points elimination for Example 6.1

## 6.3 Elimination of Prime functions

As mentioned in Section 4.4.3, each block of a decomposition is either termed a *PRIME* function or it is an associative operator. We found that, if a *PRIME* function satisfies certain conditions, we can remove it from the decomposition graph, along with all of its sub-graph and substitute it with a fresh input variable.

In order for the substitution to be acceptable, the output node of the *PRIME* block has to be *almost* a free point, in the sense that up to one input of the *PRIME* block can be a node shared with rest of the decomposition graph. As the proof shows, in this special case, the tree rooted at the *PRIME* block can still be removed. In fact, *PRIME* blocks inherently guarantee that their output cannot be kept constant by assigning any single one of their input signals. It follows that, no matter what is the value for the node that is shared with the rest of the decomposition graph, the output of *PRIME* block can still assume both values 0 and 1, and thus has full range.

**Theorem 6.2.** *Given a prime function $r(r_1, \cdots, r_r)$ in a decomposition graph* $\mathbf{F}$*, if all of its inputs, except at most one, are free points, than the decomposition graph G obtained by substituting the new variable r for function $r()$,*

$$\mathbf{F}(x_1, \cdots x_m) = \mathbf{G}(r, \cdots x_m) \circ r(r_1, \cdots r_r)$$

*is such that*

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{G}).$$

*Proof.* We distinguish two cases:

1. All the inputs of the prime block are free points. Then the output of the free block is also a free point and the theorem reduces to the hypothesis of Theorem 6.1.

2. The prime block $r$ has one input that it is not a free-point, say $r_1$, without loss of generality. All the other inputs to the prime function: $(r_2, \cdots, r_r)$ are still free points and we can assume that have been reduced to input variables by Theorem 6.1.

   In the most general case, $r_1$ is a single output function of other input variables that are in the support of both $\mathbf{G}$ and $r$: $\mathcal{S}(r_1) = (a_1, \cdots, a_p)$. The function $\mathbf{F}$ has then the form:

$$\mathbf{F}(a_1, \cdots a_p, r_2, \cdots r_r, \cdots x_m) = \mathbf{G}(r, a_1, \cdots a_p, r_1, \cdots x_m) \circ r(r_1, \cdots r_r) \circ r_1(a_1, \cdots a_p) \quad (6.3)$$

   Let's proceed again by computing the $\mathcal{R}(\mathbf{F})$ by recursively splitting on the input variables:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{(i_1,\cdots,i_p)\in\mathcal{B}^p} \mathcal{R}(\mathbf{F}_{a_1=i_1, a_2=i_2, \cdots, a_p=i_p}) \quad (6.4)$$

   For each different assignment $(i_1, \cdots, i_p)$, $r_1$ evaluates to a constant value:

$$i_r = r_1(i_1, \cdots, i_p) \in \{0, 1\}.$$

   Substituting the expansion of $\mathbf{F}$ as in Eq. 6.3, we obtain:

$$\mathbf{F}_{a_1=i_1,\cdots,a_p=i_p} = \mathbf{G}_{a_1=i_1,\cdots,a_p=i_p,r_1=i_{r_1}} \circ r_{r_1=i_{r_1}} \quad (6.5)$$

   Note that we cannot drop the cofactors w.r.t. the $a_i$ in $\mathbf{G}$ because $r_1$ is not a free point and thus its inputs fan out to other nodes of the graph.

Now, the function $r_{r_1=i_{r_1}}(r_2,\cdots,r_r)$ is a free point and as such it can be substituted by a new free variable $r$. We show now that it is not possible that $r_{r_1=i_{r_1}}(r_2,\cdots,r_r)$ reduces to a constant for any value of $i_{r_1}$. In fact, if that was the case, $r$ could be expressed as $r = r_1 \otimes r_{res}(r_2,\cdots,r_r)$, where $\otimes$ is either *AND* or *OR* and $\mathcal{S}(r_1) \cap \mathcal{S}(r_{res}) = \emptyset$. $r$ would then have a disjoint support decomposition through an associative operator and would not be a *PRIME* function.

By carrying on the substitution $r = r_{r_1=i_{r_1}}(r_2,\cdots,r_r)$, Eq. 6.5 reduces to:

$$\mathbf{F}_{a_1=i_1,a_2=i_2,\cdots,a_p=i_p} = \mathbf{G}_{a_1=i_1,a_2=i_2,\cdots,a_p=i_p}$$

which substituted into Eq. 6.4 proves the theorem.

$\square$

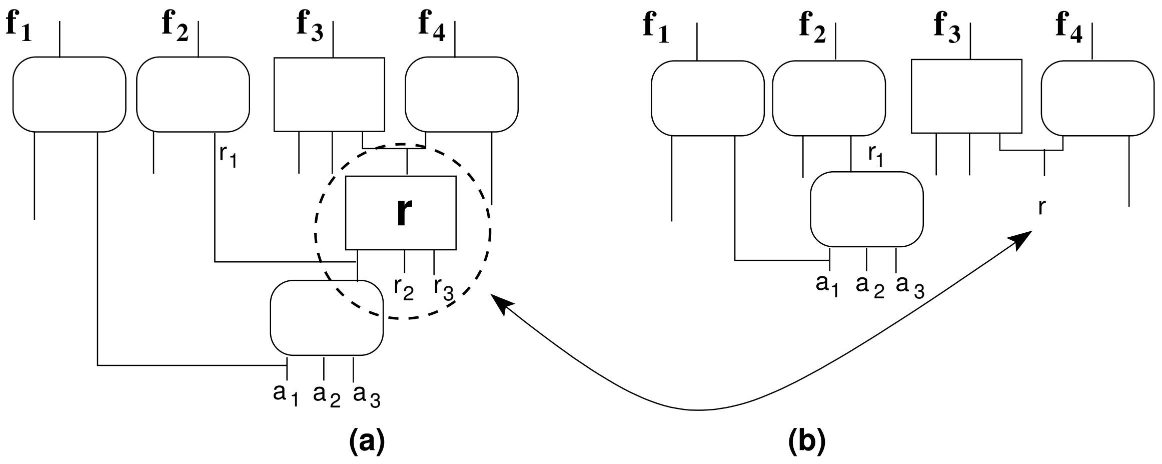

Figure 6.5: General case for prime function elimination: (a) before and (b) after

A possible structure for the graph **F** is represented in Figure 6.5.a: All the inputs to block $r$ are free points, except for $r_1$. We can then remove the block $r$ and substitute it with a new input variable obtaining the graph in Figure 6.5.b without affecting the range of the function. Note that input variables $r_2$ and $r_3$ are not needed anymore.

**Example 6.2.** *The testbench s1196 from the IWLS suite contains the blocks reported in Figure 6.6 in its next state function at step 10 of symbolic simulation. The variables names are just indices corresponding to the variables in the support of the state function. Since the prime function r has the two inputs $x_{35}$ and $x_{39}$ that are free points and only one input that has multiple fanout, we can completely eliminate this portion of the graph and just substitute it with the input variable r.*

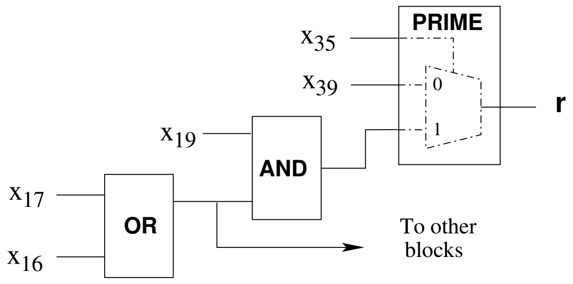

Figure 6.6: Prime elimination for Example 6.2.

## 6.4 Removal of non-dominant variables

Under certain conditions, an input variable can be removed from the decomposition graph without affecting its range.

**Example 6.3.** *Consider the following 3-outputs function:*

$$\begin{aligned} f_1 &= AND(b,e) \\ f_2 &= AND(e,OR(a,b,d)) \\ f_3 &= XOR(a,c) \end{aligned}$$

*The range of this function is* $\mathcal{B}^3 \setminus \{101,100\}$. *We can remove the variable a from the function, by cofactoring all the components w.r.t.* $a = 0$ *without changing the range spanned by* $\mathbf{F}$. *The result* *is:*

$$\begin{aligned} f_1 &= AND(b,e) \\ f_2 &= AND(e, OR(b,d)) \\ f_3 &= c \end{aligned}$$

*and it still has range* $\mathcal{B}^3 \setminus \{101, 100\}$.

We could do the simplification in the example because the range of the function for $a = 1$ is a subset of the range for $a = 0$. The following definition formalizes the situation:

**Definition 6.3.** *An input variable of a decomposition graph has a* **non-dominant value 0** *iff it fans out only to blocks that are decomposed through OR or XOR associative operators. It has a* **non-dominant value 1** *iff it fans out only to blocks that are AND or XOR decompositions. Otherwise it does not have a non-dominant value.*

Note in particular that a variable may have a non-dominant value 0 and a non-dominant value 1 simultaneously if it fans out only to XOR decompositions. The theorem below shows that in the most general case, a variable that fans out only to associative operators can be removed from the decomposition graph if it has a unique non-dominant value for the whole graph.

**Theorem 6.3.** *If a decomposition graph F has an input variable v with non-dominant value* $k \in \{0,1\}$*, and each of the blocks (i.e., intermediate single-output functions) that have v in their fanin have at least one other input in their fanin that is a free point, then:* $\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{v=k})$

*Proof.* For a generic function $\mathbf{F}$, we have:

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{v=k}) \cup \mathcal{R}(\mathbf{F}_{v=\bar{k}}) \tag{6.6}$$

We now show that under the conditions specified:

$$\mathcal{R}(\mathbf{F}_{v=\bar{k}}) \subseteq \mathcal{R}(\mathbf{F}_{v=k}) \tag{6.7}$$

and Eq. 6.6 reduces to $\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{v=k})$.

Let's consider first the case where $k = 0$ and let's label each of the functions that have $v$ in their fanin $x(v, p, x_1, \cdots x_x)$, $y(v, q, y_1, \cdots y_y)$, $w(v, r, w_1, \cdots w_w)$ ... where $p$, $q$, $r$ ... are the free points in each of them and $x_i$, $y_i$, $w_i$ ... are other variables the functions depend on. The $x()$, $y()$, $w()$, ... functions by hypothesis can only be *OR* or *XOR* decompositions.

We can then express $\mathbf{F}$ using the composition of these functions:

$$\mathbf{F} = \mathbf{G}(x, y, w, \cdots, x_1 \cdots x_x, y_1 \cdots y_y, \cdots w_w, \cdots) \circ x(v, p, x_1, \cdots x_x) \circ y(v, q, y_1, \cdots y_y) \cdots$$

Note that, in general, $x_i$, $y_i$, $w_i$ ... are also in the fanin of $\mathbf{G}$. Let's now compute the two cofactors of $\mathbf{F}$ w.r.t. $v$:

$$\mathbf{F}_{v=0} = \mathbf{G}(x, y, w, \cdots, x_1 \cdots x_x, y_1 \cdots y_y, \cdots w_w, \cdots) \circ x(0, p, x_1, \cdots x_x) \circ y(0, q, y_1, \cdots y_y) \circ \cdots$$
$$\mathbf{F}_{v=1} = \mathbf{G}(x, y, w, \cdots, x_1 \cdots x_x, y_1 \cdots y_y, \cdots w_w, \cdots) \circ x(1, p, x_1, \cdots x_x) \circ y(1, q, y_1, \cdots y_y) \circ \cdots$$

In order to show the inclusion of the ranges of Eq. 6.7, we are going to represent each range as a union of ranges by cofactoring the variables in the support of $x$, $y$, $w$, ... one function at a time starting with $x()$:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \bigcup_{(x_1 \cdots x_x) \in \mathcal{B}^x} \mathcal{R}(\mathbf{G}(x, y, \cdots x_1 \cdots, y_1 \cdots) \circ x(0, p, x_1 \cdots)_{x_1 = i_{x_1}, \cdots})$$
$$\mathcal{R}(\mathbf{F}_{v=1}) = \bigcup_{(x_1 \cdots x_x) \in \mathcal{B}^x} \mathcal{R}(\mathbf{G}(x, y, \cdots x_1 \cdots, y_1 \cdots) \circ x(1, p, x_1 \cdots)_{x_1 = i_{x_1}, \cdots})$$

We distinguish two cases for each $x$, $y$, $w$, ... function:

1. **x is a OR decomposition**. When all the $(x_1, \cdots, x_x)$ are zero, for $\mathbf{F}_{v=1}$, $x$ evaluates to the constant value 1. For $\mathbf{F}_{v=0}$, $x = p$. In all the other cases $x$ evaluates to 1. By grouping all the component ranges so that to distinguish the special case from all the others , we can simplify

the expressions:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \mathcal{R}(\mathbf{G}(p,y,\cdots 0 \cdots 0,\cdots)) \bigcup_{(x_1,\cdots,x_x)\neq \mathbf{0}} \mathcal{R}(\mathbf{G}(1,y,\cdots x_1 \cdots x_x \cdots)_{x_1=i_{x_1},\cdots}$$
$$\mathcal{R}(\mathbf{F}_{v=1}) = \mathcal{R}(\mathbf{G}(1,y,\cdots 0 \cdots 0,\cdots)) \bigcup_{(x_1,\cdots,x_x)\neq \mathbf{0}} \mathcal{R}(\mathbf{G}(1,y,\cdots x_1 \cdots x_x,\cdots)_{x_1=i_{x_1},\cdots}$$

It can be easily seen that the first range for $\mathbf{F}_{v=1}$ is a subset of the corresponding range for $\mathbf{F}_{v=0}$, while the rest of the expression is identical.

2. **x is an XOR decomposition**. For the 1-cofactor, $\mathbf{F}_{v=1}$, $x = XNOR(p,\, x_1,\cdots,x_x)$. In the case of the 0-cofactor, $\mathbf{F}_{v=0}$, $x$ evaluates to the complement: $x = XOR(p,x_1,\cdots,x_x)$. We can again group all the component ranges so that to distinguish the cases where $XOR(x_1,\cdots,x_x) = 0$ from the ones where $XOR(x_1,\cdots,x_x) = 1$:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \bigcup_{XOR(x_1,\cdots x_x)=0} \mathcal{R}(\mathbf{G}(p,y,\cdots x_1 \cdots)_{x_1=i_{x_1},\cdots} \bigcup_{XOR(x_1,\cdots x_x)=1} \mathcal{R}(\mathbf{G}(\bar{p},y,\cdots x_1 \cdots)_{x_1=i_{x_1},\cdots}$$
$$\mathcal{R}(\mathbf{F}_{v=1}) = \bigcup_{XOR(x_1,\cdots x_x)=0} \mathcal{R}(\mathbf{G}(\bar{p},y,\cdots x_1 \cdots)_{x_1=i_{x_1},\cdots} \bigcup_{XOR(x_1,\cdots x_x)=1} \mathcal{R}(\mathbf{G}(p,y,\cdots x_1 \cdots)_{x_1=i_{x_1},\cdots}$$

And it can be observed that the two components of each expression match. It follows: $\mathcal{R}(\mathbf{F}_{v=0}) = \mathcal{R}(\mathbf{F}_{v=1})$.

This procedure can be applied recursively for each of the other functions $y$, $w$, $\ldots$, by computing and grouping all the cofactors for the sets of input variables $(y_1 \cdots y_y)$, $(w_1 \cdots w_w)$, $\ldots$.

For the case where $k = 1$, the functions $x$, $y$, $w$, $\ldots$ can now only be *AND* or *XOR* decompositions. The corresponding proof can be obtained by substituting *AND* for *OR* and 1 for 0 in the proof just discussed. Finally, for the case where the input variable $v$ has both a non- dominant value 0 and 1, we can just use any of the two value- specific proofs. $\square$

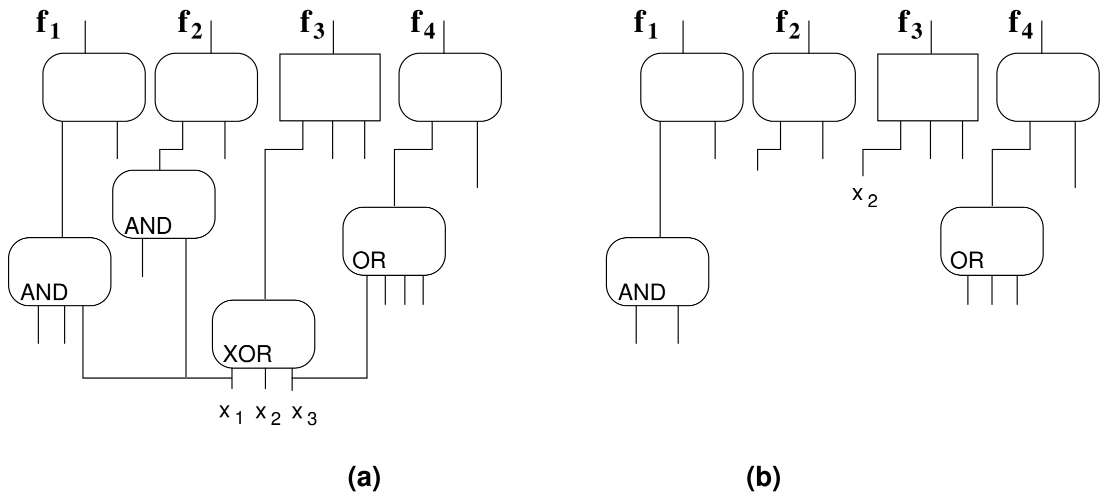

Figure 6.7: Non-dominant variable removal for Example 6.4

**Example 6.4.** *Figure 6.7.a shows a system with two non-dominant variables: $x_1$ has a non-dominant value 1, since it only fans out to AND and XOR nodes, while $x_3$ has a non-dominant value 0, since it fans out to OR and XOR. After removing of these two non-dominant variables and eliminating the nodes left with only one input, we obtain the system in Figure 6.7.b. Note that at this point we can apply the free point reduction technique to the graph of* **F**.

## 6.5 DSD-SS Implementation

Our implementation of the Disjoint Support Decomposition based Symbolic Simulator performs the parameterizations at the end of each symbolic simulation step. We first generate the decomposition graph for the state vector $\mathbf{S_{@k}}$ and then attempt the three transformations described above. Often, the graph produced by applying one of the transformations enables further simplifications through some of the other transformations.

Even when all of the transformations fail, we still want to maintain a compact representation for the state function $\mathbf{S_{@k}}$, so that we can make further progress with the simulation. Thus, when the state function exceeds a threshold value, we choose a variable to set to a constant value. The variable with fanout to the maximum number of blocks is selected becuase by simplifying this variable we eliminate the largest interdependency among the nodes of the graph and thus we maximize the likelyhood of creating a graph where our techniques can be applied in future simulation steps. When computing the fanout of a primary variable that is candidate for elimination, we only consider those decomposition blocks which have other input variables in their fanin. The intuition behind this choice is that those blocks are closer to become free points, since some of their inputs are already free points.

We found experimentally that often, after eliminating a variable by setting it to constant as described, we could discover additional free points or variables with non-dominant values.

## 6.6 Experimental results

The algorithm presented in this chapter was implemented in a C++ program called Disjoint Support Decomposition based Symbolic Simulator (DSD-SS). We tested this approach on the largest sequential circuits from the Logic Synthesis Benchmarks suite [68] and the ISCAS'89 Benchmark Circuits [16], including their 1993 additions, as we did for the previous CBSS technique in Chapter 3. Table 6.1 reports results on all but the smallest testbenches of the two suites (we excluded from the table the circuits with less then 20 memory elements). The testbenches are grouped by benchmark suite. 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. We linked the DSD-SS to the CUDD package [29] as the underlying BDD manipulation library for the combinational portion of the simulation and a proprietary BDD package for the parameterizations. We set the reordering threshold in CUDD to 80,000 nodes. Each testbench is run for 100 simulation steps and, at the end of each step, DSD-SS performs the decomposition of the next state symbolic vector and applies the transformations described in Sections 6.2-6.4. Whenever the transformations are not sufficient to provide an exact small representation for the state vector, we resort to pick a variable to evaluate to a constant value, in order to guarantee a compact representation. The variable is chosen based on the criteria described in the previous section. After a few experiments, we chose 2,500 nodes as a reasonable value to use for the upper limit for the size of the state vector. We noticed that, generally speaking, this value can be used to trade-off simulation breadth vs. time.

For each circuit, the table reports first the same relevant metrics that we presented before in Table 3.1: the number of inputs In, outputs Out, memory elements FF, and internal network gates Gates.

The next three columns report how many times we were able to apply our transformations: FP is the cumulative number of free point substitutions, PE is the number of prime function eliminations, NVD the number of non-dominant variables removals over all the symbolic simulation steps. The next column of this group, Null, counts the cumulative number of times where no exact transformation could be applied, but the state vector was within the limit size (of 2,500 nodes), and thus DSD-SS advanced to the next step of simulation without applying any parameterization. Note that during a single simulation step we may apply more than one technique until we reduce the state vector within limits or until no additional exact parameterization is possible. The values of Table 6.1 indicate that the conditions that allow an exact parameterization of the state vector are frequently met in almost all the circuits. In particular, in most cases the transformations can be applied successfully multiple times during each same simulation step. Free point elimination is the parameterization that achieves the best results across all the testbenches producing a total 2,417 exact simplifications over 4,200 simulation steps (42 testbenches, each run for 100 steps). The second most successful technique appears to be the non-dominant variable removal, which was applied for a total of 1,243 times, while prime function elimination satisfied the necessary conditions for exact parameterization only 139 times.

The purpose of the next group of columns is to compare the breadth of the state exploration between DSD-SS and a pure symbolic simulator that does not include parameterization. To this end, we built a plain symbolic simulator and we constrained it to have the same upper bound for the size of the state vector at the end of each simulation cycle as DSD-SS. While the only reduction technique available to the plain symbolic simulator was approximation of the state vector by evaluating symbolic variables to constant values, DSD-SS would attempt first exact parameterization, and default to approximation only as a backup method. The number of variables approximated to constant provides an indication of how much the search breadth has been restricted: every time a variable is set to constant, we cut in half the amount of equivalent simulation traces checked by the exploration. Thus, in this section of the table, a bigger value indicates a more aggressive approximation and a smaller breadth of search. DSD-SS greatly outperformed a pure symbolic simulator in all but three testbenches. The situation of a test such as *s635*, can arise because DSD-SS chooses the variable to approximate so to maximize the chance of being able to perform additional exact parameterizations. This may not be the choice that leads to the smallest BDD vector size with the least number of approximations. However, in all the other cases, even with this disadvantage, DSD-SS avoids the elimination of many symbolic variables and propagates through the simulation a factor of 2 to 10 times more symbols over a plain symbolic simulator, when the same amount of memory is available. The situation of test *bigkey* is exceptional in this sense: because of the exact parameterizations, DSD-SS could avoid the evaluation to constant of more than 11,000 symbols over a plain symbolic simulator.

The last column reports the execution times of DSD-SS. The current implementation of DSD-SS at this point is fairly poor, since we need to transfer the data back and forth between the two BDD packages many times during the simulation. The proprietary BDD package that we currently use to perform the parameterizations has special functionalities for linking to the Disjoint Support Decomposition library. Execution times are also penalized by multiple variable reorderings in the CUDD package that are triggered by many of the testbenches. We hope in the near future to be able to directly link the DSD library to the CUDD package; we expect this connection to provide great improvements in the performance of DSD-SS. At this point, the plain symbolic simulator executes faster than DSD-SS since it can rely simply on the usage of the CUDD package. Still, in a few cases DSD-SS can gain enough advantage from a compact representation to be faster than the plain simulator, for instance, in the case of test *bigkey*.

Finally, the testbenches with a “-” mark indicate that either the plain symbolic simulator or DSD-SS run out of the allotted time of one hour of execution. For these testbenches we only report

| Circuit | In | Out | FF | Gates | Par.techniques FP | PE | NDV | Null | Symbol reductions DSD-SS | PlainSym. | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Logic Synthesis ‘91 - FSM tests** | | | | | | | | | | | |
| ex1 | 9 | 19 | 20 | 622 | 0 | 0 | 0 | 100 | 0 | 0 | 0.3 |
| s1423 | 17 | 5 | 74 | 830 | 5 | 0 | 3 | 18 | 156 | 659 | 48.78 |
| s838 | 35 | 2 | 32 | 596 | 0 | 1 | 1 | 98 | 0 | 0 | 7.98 |
| s953 | 16 | 23 | 29 | 658 | 67 | 0 | 1 | 6 | 555 | 677 | 108.37 |
| **Logic Synthesis ‘91 - Addition ‘93** | | | | | | | | | | | |
| bigkey | 262 | 197 | 224 | 9211 | 28 | 0 | 47 | 1 | 178 | 11781 | 30.17 |
| clma | 382 | 82 | 33 | 24482 | 9 | 0 | 20 | 69 | 12 | 10 | 17.46 |
| dsip | 228 | 197 | 224 | 3893 | 24 | 0 | 0 | 1 | 395 | 13043 | 428.48 |
| mm9a | 12 | 9 | 27 | 639 | 0 | 0 | 14 | 27 | 110 | 71 | 11.93 |
| mm9b | 12 | 9 | 26 | 786 | 2 | 0 | 3 | 7 | 211 | 277 | 62.18 |
| mult16b | 17 | 1 | 30 | 284 | 63 | 0 | 78 | 1 | 1185 | 1229 | 107.55 |
| mult32a | 33 | 1 | 32 | 715 | 0 | 0 | 0 | 1 | 2003 | - | 9507.86 |
| s38417 | 28 | 106 | 1465 | 23771 | 49 | 1 | 13 | 6 | 148 | 867 | 7.75 |
| s38584 | 38 | 304 | 1426 | 20281 | 138 | 1 | 34 | 9 | 516 | 1755 | 86.18 |
| s5378 | 35 | 49 | 163 | 3232 | 136 | 0 | 30 | 1 | 636 | 1145 | 615.63 |
| s838 | 34 | 1 | 32 | 618 | 0 | 0 | 0 | 51 | 52 | 61 | 66.21 |
| s9234 | 36 | 39 | 135 | 3019 | 156 | 1 | 117 | 0 | 297 | 477 | 48.69 |
| sbc | 40 | 56 | 27 | 1143 | 183 | 1 | 28 | 1 | 1086 | 1314 | 244.3 |
| **ISCAS ‘89 - FSM tests** | | | | | | | | | | | |
| s13207.1 | 62 | 152 | 638 | 9539 | 53 | 1 | 12 | 8 | 607 | 1080 | 35.1 |
| s13207 | 31 | 121 | 669 | 9539 | 27 | 0 | 2 | 31 | 96 | 189 | 26.15 |
| s1423 | 17 | 5 | 74 | 830 | 5 | 0 | 3 | 18 | 156 | 637 | 52.17 |
| s15850.1 | 77 | 150 | 534 | 11316 | 103 | 23 | 100 | 0 | 994 | 2615 | 326.22 |
| s15850 | 14 | 87 | 597 | 11316 | 2 | 0 | 98 | 0 | 55 | 120 | 9.93 |
| s35932 | 35 | 320 | 1728 | 23085 | 0 | 0 | 0 | 16 | 245 | 1183 | 18.68 |
| s38417 | 28 | 106 | 1636 | 27648 | 47 | 1 | 12 | 6 | 155 | 1293 | 6.52 |
| s38584.1 | 38 | 304 | 1426 | 24619 | 124 | 0 | 51 | 9 | 500 | 1624 | 61.15 |
| s38584 | 12 | 278 | 1452 | 24619 | 21 | 0 | 0 | 9 | 141 | 458 | 19.27 |
| s5378 | 35 | 49 | 179 | 3973 | 150 | 0 | 58 | 1 | 680 | 1027 | 388.27 |
| s838 | 34 | 1 | 32 | 626 | 0 | 0 | 0 | 51 | 52 | 61 | 72.76 |
| S9234.1 | 36 | 39 | 211 | 6585 | 204 | 1 | 104 | 3 | 682 | 1096 | 110.46 |
| s9234 | 19 | 22 | 228 | 6585 | 88 | 0 | 11 | 18 | 311 | 437 | 41.78 |
| s953 | 16 | 23 | 29 | 658 | 38 | 0 | 1 | 7 | 547 | 764 | 110.97 |
| **ISCAS ‘89 - Addition ‘93** | | | | | | | | | | | |
| prolog | 36 | 73 | 136 | 1845 | 130 | 6 | 10 | 0 | 459 | 1201 | 169.99 |
| s1269 | 18 | 10 | 37 | 771 | 13 | 0 | 3 | 2 | - | 1306 | - |
| s1512 | 29 | 21 | 57 | 990 | 136 | 95 | 113 | 0 | 278 | 931 | 34.54 |
| s3271 | 26 | 14 | 116 | 2166 | 0 | 0 | 1 | 16 | 714 | 1469 | 23.37 |
| s3330 | 40 | 73 | 132 | 2020 | 144 | 5 | 51 | 0 | 472 | 1460 | 83.77 |
| s3384 | 43 | 26 | 183 | 1734 | 61 | 2 | 3 | 4 | 1551 | 2565 | 297.18 |
| s4863 | 49 | 16 | 104 | 2492 | 163 | 0 | 149 | 0 | - | 2400 | - |
| s635 | 2 | 1 | 32 | 382 | 31 | 0 | 0 | 35 | 82 | 5 | 55.74 |
| s6669 | 83 | 55 | 239 | 3272 | 17 | 22 | 0 | 1 | - | 6262 | - |
| s938 | 34 | 1 | 32 | 626 | 0 | 0 | 0 | 51 | 52 | 61 | 70.76 |
| s967 | 16 | 23 | 29 | 677 | 0 | 0 | 50 | 50 | 0 | 731 | 2.56 |

Table 6.1: Disjoint Support Decomposition-based simulation results

the number of transformations that we were able to complete.

## 6.7 Summary

This chapter introduced a new parameterization technique for symbolic simulation, DSD-SS. This work was published in [11]. Its core contribution is in exploiting the disjoint support decomposition properties of the state vector to generate a compact parameterization during symbolic simulation.

The major advantage of this approach is that it is a loss-less transformation, that means we can generate a compact representation of the state vector, without losing any of the information it carries between simulation steps. Results show that, within a fixed amount of memory resources dedicated to represent the frontier set, we can keep a much broader search space than pure symbolic simulation.