STACCATO: Disjoint Support Decompositions from BDDs through Symbolic Kernels

Stephen Plaza and Valeria Bertacco

Advanced Computer Architecture Lab
The University of Michigan – Ann Arbor, MI

Abstract— A disjoint support decomposition (DSD) is a representation of a Boolean function F obtained by composing two or more simpler component functions such that the component functions have no common inputs. The decomposition of a function is desirable for several reasons. First, it's a method to obtain a multiple-level implementation of a function. It leads to a partition in simpler blocks that easily results in smaller areas and fewer interconnects. Moreover, it exposes a parallelism in the computation of the function that can be exploited by hardware as well as during simulation.

In this paper we present a novel algorithm, STACCATO, that generates a DSD decomposition starting from the BDD of a function. STACCATO is novel because 1) it provides a complete description of each decomposition, that is, it computes the "kernel" function K relating the elements of each decomposition, and 2) it has better performance than previously known algorithms. Experimental results run on both IWLS and industrial testbenches show that STACCATO's performance is in most cases three times as fast or more than previously known solutions.

I. Introduction

The disjoint support decomposition (DSD) of a Boolean function $F(x_1, \cdots, x_n)$ consists in representing F by means of simpler component functions J and K, such that the inputs of J and K do not share any input variable, and:

$$F = K(x_1, \cdots, x_{j-1}, J(x_j, \cdots, x_n)). \tag{1}$$

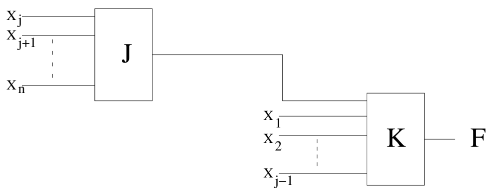


Fig. 1. A Disjoint Support Decomposition of F

In general, a function has several disjoint support decompositions, which can be superimposed to obtain decompositions with finer granularity. Moreover, it is possible to recursively search for disjoint support decompositions for functions J and K to produce even smaller components. At the limit, F can be represented as a tree of functions, with the inputs x_i being the leaves of the tree.

Techniques for solving disjoint decompositions have been studied for the past 40 years. Traditional approaches are based on using decomposition tables to partition the inputs into disjoint sets as in [1, 2], while more recent techniques proceed by computing the DSD starting from a BDD representation [3, 4, 5]. Decompositions have been applied to many domains including *Synthesis* applications, *placement and routing* applications, and *verification* [6, 7, 8, 9, 10, 11].

In this paper we present STACCATO, a novel algorithm that computes efficiently all the disjoint support decompositions of a function. Previous algorithms developed with this objective include [3, 12]. This previous work focused on finding all the partitions of the support of F for which Eq. 1 holds. The novelty of our approach rests in the additional insightful description of the function's structure that we can provide. In generating the complete set of disjoint-support decompositions for a function, we provide the support partition for each decomposition, and we compute the root function K, also called *kernel*, that relates together the disjoint components in F – with reference to Fig. 1, all the functions like J – through a novel technique that generates the kernel functions in compact form and at no extra computational cost. We achieve this by using a kernel-dependent optimization for calculating generalized cofactors called, *symbolic generalized cofactors*. The benefits of generating the kernel functions span multiple domains: from technology mapping where the set of functions represents an optimal gate library for minimal-cardinality routing, to general applications in synthesis and verification, where the knowledge of the root function enables computations to be performed directly on the decomposed form of a function.

In the remainder of the paper, we first review the previous work in this area and provide the necessary background on disjoint support decompositions. We then present our novel contributions of symbolic kernels and symbolic generalized cofactors. We conclude by analyzing the performance of our technique in the experimental results section and by discussing future research directions.

II. Previous work

Ashenhurst was first to propose a theory and an algorithm for constructing disjoint support decompositions in [1]. This algorithm discovers a decomposition by partitioning the support of a function in two disjoint sets and then verifying that the chosen partition is a valid disjoint support decomposition. The decomposition can be applied recursively to each of the two sets to find finer granularity decompositions. This approach requires an exponential amount of time due to the exponential number of trial partitions of the support of a function that need to be checked. As a consequence, practical uses of this approach are confined to functions of a few variables.

Subsequent research efforts in this area developed in different directions. One focused on simplified, approximated solutions of manageable algorithmic complexity, such as algebraic factorization [13]. The other direction involves the use of heuristics to limit the number of trial partitions in the original approach [2, 12, 14, 15]. In recent years, the widespread acceptance of BDDs to represent Boolean functions has motivated additional research in constructing decompositions directly from a BDD representation. In [3], an algorithm is presented that generates all disjoint support decompositions of a function by traversing a BDD from the bottom up in a recursive fashion. The complexity of this algorithm is quadratic with respect to the size of the BDD which, in most cases, represents a significant improvement over the algorithm developed by Ashenhurst.

III. Background

This section provides the necessary background on the theory of disjoint support decompositions and a BDD-based recursive algorithm presented in [3, 4, 5], for computing them. First, however, we provide general definitions.

III-A. Definitions

A scalar function F is a mapping $F : \mathcal{B}^m \rightarrow \mathcal{B}$, where $\mathcal{B}$ denotes the set $\{0, 1\}$. Lower case letters will be used for variables and upper case letters will be used for functions. The *1-cofactor* of a functions F w.r.t. a variable v is the function $F_{v=1}$ obtained by substituting 1 for v in F. Conversely, the *0-cofactor* is obtained by substituting 0 for v in F. The *generalized cofactor* of a function F is an operation that restricts the relevant portion of the co-domain of F to a care-set, specified by the second operand [16]:

DEFINITION 1. *Given two functions F and G, the* **generalized cofactor** *of F w.r.t. G is the function F_G such that for each input combination satisfying G the outputs of F and F_G are identical.*

Section IV-C shows how STACCATO can reduce the operation of generalized cofactor to a simple cofactor w.r.t. a single Boolean variable when applied to a DSD.

The *support* of a logic function is the set of variables F depends on and is indicated by $\mathcal{S}(F)$.Two functions F and G are *disjoint-support* if they share no support variables, *i.e.*, $\mathcal{S}(F) \cap \mathcal{S}(G) = \emptyset$. We indicate the size of the support set of F with $|\mathcal{S}(F)|$.

BDDs are the underlying structure we used to represent functions. An introduction to BDDs can be found in [17].

III-B. Disjoint Support Decompositions (DSD)

In the most general case, the disjoint support decomposition of a scalar function F, consists of finding other, simpler functions L and A_i such that:

$$F(x_1, .., x_n) = L(A_1(x_1, .., x_{A_1}), A_2(x_{A_1+1}, .., x_{A_2}), ..) \quad (2)$$

with $\mathcal{S}(A_i) \cap \mathcal{S}(A_j) = \emptyset, \forall i, j$.

The decomposition can be applied recursively to each of the A_i components leading to a **decomposition tree** representation for the Boolean function F. Figure 2.a shows an example of decomposition tree: each block represents a function with a single output and input functions that have pairwise disjoint support. Referring to the naming convention and canonical form defined in [3, 4], each block is characterized by its **kernel function**, K_F, which defines the inputs/output relation of the block. The list of functions that are inputs of K_F is called the **actuals list**. Finally, the leaves of the decomposition tree are the input variables of the function F.

The kernel K_F of a block can be either an **associative operator** (AND, OR, XOR) or a **prime function**. Prime functions are simply functions that cannot be decomposed any further with disjoint support inputs; examples of primes are the *majority* and the *multiplexor* functions. A prime function must contain at least three inputs; if a kernel function depends on less than three variables, then it must be trivially an associative operator.

III-C. Construction of DSDs from a BDD representation

In recent years, researchers have considered generating decompositions starting from the BDD of a Boolean function [14, 3]. In particular, the algorithm presented in [3] generates the decomposition tree of a function in a recursive fashion by traversing its BDD bottom-up. At each node F, a number of conditions on the cofactors F_0 and F_1 is checked, and, based on the results of this analysis, one of a number of different constructions is applied. The algorithm presented maintains only the actuals lists of each decomposition, since they prove that the kernel K_F can be uniquely determined from the actuals list. This approach allows them to achieve a low algorithmic complexity of worst-case quadratic in the size of the BDD representing the function being decomposed.

IV. DECOMPOSITIONS WITH SYMBOLIC KERNELS

We present now STACCATO, our algorithm for disjoint support decompositions. STACCATO computes the disjoint decomposition of a function and generates its decomposition tree from the BDD. Moreover, it constructs at the same time a representation of the kernel function for each block using BDDs. We call this representation *symbolic kernel*. For simplicity of notation, from hereon, we use the symbol K_F to denote specifically the "symbolic kernel of F", in contrast with its generic meaning of "kernel of F" used up to this point. In the following, we overview the decomposition tree structure used in STACCATO. We then discuss the properties of symbolic kernels and explain how they can be used to simplify the computation of the generalized cofactors that are used throughout the DSD algorithm. Finally, we present a technique to efficiently calculate symbolic kernels that results in minimal overhead in STACCATO.

IV-A. Structure of a decomposition node

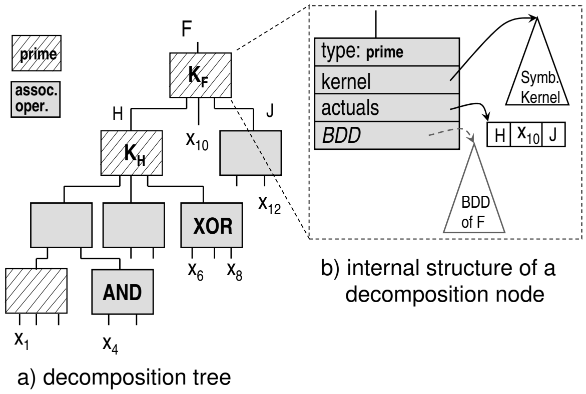


Fig. 2. Decomposition tree and structure of a decomposition node

Figure 2 shows an example decomposition. Part a) shows the decomposition tree for a function F where blocks representing prime decompositions are hashed and blocks representing associative operators are grayed. The leaf nodes are connected to the function's input variables. Figure 2.b) shows the structure used to represent a decomposition node. In addition to the decomposition type and actuals list, the block contains a field referencing a BDD that represents the *symbolic kernel*, as defined below. The last field is a pointer to the BDD of F. This field is used as key in a hash table to determine whether a given function has been previously decomposed.

IV-B. Symbolic Kernels

STACCATO represents the kernel of a DSD using a structure called *symbolic kernel*. The use of symbolic kernels enables us to simplify the computation of the generalized cofactors required in the DSD algorithm.

A symbolic kernel is a BDD where each variable in the support represents a distinct disjoint component of the DSD of F. In STACCATO, we call these variables *representative variables*.

DEFINITION 2. *Given a function $F = K_F(A_1, A_2, A_3, ...)$, a* **representative variable** *for the member function A_i is a variable x such that $x \in \mathcal{S}(A_i)$.*

The set of representative variables for the DSD of F constitutes the support of K_F. Note that each member function is always mapped to a distinct representative variable since member functions have disjoint supports. Moreover, there are in general multiple variables that can be selected as the representative variable of a list member; we exploited this freedom to optimize implementation performance by always selecting the bottom variable in the BDD ordering. We provide below a formal definition of symbolic kernels:

DEFINITION 3. *The* **symbolic kernel** *of F, K_F, is a kernel function whose support consists exactly of the set of representative variables determined by the bottom variable of each actuals list element in the decomposition of F, based on the BDD variable ordering used to represent the decomposition.*

Example 1. $F = \mathrm{MAJ}(b, \mathrm{MUX}(a, cd, fg), e)$ where the variable order from top to bottom is a, b, c, d, e, f, g. To construct the symbolic kernel according to Definition 3, a representative variable needs to be chosen for each actuals list member $\{b, \mathrm{MUX}(a, cd, fg), e\}$. By choosing the bottom variable in each actuals list member, the following mapping is achieved: $\{b, g, e\}$. The resulting symbolic kernel would be $\mathrm{MAJ}(b, g, e)$.□

Symbolic kernels are maintained for each intermediate decomposition during the execution of STACCATO. The basic construction step involves substituting the proper representative variable for the corresponding actuals list member in F.

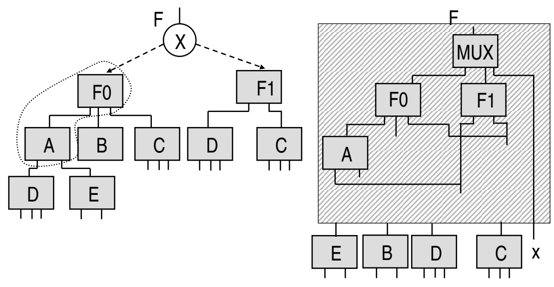
IV-C. Symbolic Generalized Cofactor

The symbolic generalized cofactor is a cofactoring operation performed on the symbolic kernel. Since each representative variable is mapped to a disjoint support function, taking the cofactor of K_F with respect to a representative variable corresponds to computing the generalized cofactor of F with respect to an actuals list function.

The DSD algorithm defined in [4] makes extensive use of generalized cofactors to identify special cases of decomposition that occur in the bottom-up algorithm. By using symbolic generalized cofactors, STACCATO simplifies the computation necessary to identify these decompositions: Each generalized cofactor operation applied to a complex F function reduces to a simple cofactor computation applied to a much smaller K_F.

IV-D. Tree-Intersection and Symbolic Merging

The computation of a new decomposition for a function $F = \overline{x}F_0 + xF_1$, given the decomposition of the two cofactors F_0 and F_1, involves computing a special "tree-intersection" between the decomposition trees of F_0 and F_1 to generate the actuals list for F. The intersection algorithm produces two lists. The first one contains blocks whose support only occurs in one of the trees. These blocks are referred to as *exclusive blocks*. For instance, a block in the decomposition tree of F_0, whose support variables are disjoint from the support of F_1, is an exclusive block. Additionally, exclusive blocks whose parent block is also an exclusive block do not appear in the list. The second list produced by the intersection algorithm consists of blocks that are common to F_0 and F_1. These blocks are called *common blocks*. Similarly, this second list does not contain common blocks whose parent block is also a common block. The union of the two lists and the top variable x constitutes the actuals list for the decomposition of F. The example below illustrates this tree-intersection operation:



a) decompositions of F_0 and F_1 b) New kernel of F *(hatched area)*

Fig. 3. Kernel construction for NEW decompositions

Example 2. Consider the function F of Figure 3.a) where the decomposition trees of the two cofactors are as illustrated. By performing the tree-intersection operation we find that functions B and E are exclusive blocks and are placed into the first list, while functions C and D are common blocks and are placed into the second list. Note that none of the other blocks are either exclusive or common. The final actuals list for the new decomposition is then: $\{x, B, C, D, E\}$. Figure 3.b) shows the new actuals list along with a hashed area that denotes the kernel of F. $\Box$

In addition to computing the actuals list, STACCATO needs to generate a corresponding new symbolic kernel. *Symbolic merging* is the algorithm used for symbolic kernel construction on-the-fly during the tree-intersection operation. Both symbolic kernels K_{F_0} and K_{F_1} are expanded during tree-intersection until their supports include only representative variables that map to exclusive or common blocks. The resulting symbolic functions are called SF_{F_0} and SF_{F_1}. In general, SF_{F_0} and SF_{F_1} are functions that can be further decomposed, in contrast with K_{F_0} and K_{F_1}, which are prime functions or represent associative operators. The two symbolic functions are then composed with a multiplexor block to produce the new symbolic kernel, K_F, as indicated pictorially in Fig. 3.b). The decomposition algorithm guarantees that K_F is, in fact, a kernel function even when SF_{F_0} and SF_{F_1} are not. The symbolic merging algorithm is computationally much simpler than generating the kernel function completely after the tree-intersection operation, because the former involves only the smaller symbolic kernels, K_{F0} and K_{F1}, the latter requires computations over F.

Example 3. With reference to Figure 3, when constructing the symbolic kernel for F, the inputs of F_0 are examined first. K_{F0} contains representative variables for the functions A, B, and C. However, because A is neither exclusive nor common, it must be composed into K_{F0}. The resulting symbolic function, SF_{F0}, has representative variables for the functions B, C, D, and E. The construction of SF_{F0}, is simply obtained by substituting K_A for the representative variable of A in K_{F0}. F_1 is then analyzed. F_1 is neither exclusive nor common, thus its inputs must be examined. Because each input function is a common block, we have $SF_{F1} = K_{F1}$. Since at this point SF_{F0} and SF_{F1} have consistent variable mappings by construction, the symbolic kernel of F can simply be computed with the following operation: $K_F = \text{MUX}(x, SF_{F0}, SF_{F0})$. $\Box$

V. Experimental Results

We implemented STACCATO and ran experiments on circuits derived from the combinational portions of IWLS, ISCA - LogicSynthesis, and VIS suites, as well as units from the publicly released picoJava processor by SUN Microsystem [18]. STACCATO is a library that links to the CUDD package [19], which we use for BDD construction and manipulation. First, we build a BDD for each primary output function using CUDD, and then call STACCATO to decompose each individual output.

Table I reports relevant metrics for the circuits considered. The *in* and *out* columns represent the number of primary inputs and outputs respectively. Column *dec.out* describes the fraction of outputs in each test that are decomposable. The number of blocks in the decomposition is given by the field *blocks*.

V-A. Resource Requirements: CUDD vs. STACCATO

The memory required by STACCATO is compared to the memory required by CUDD to construct the initial BDDs. The last two columns of Table 1, %+BDD and % DEC, refer to the overhead required to maintain the symbolic kernels and the decomposition tree of the circuits, respectively, compared to the total memory needed for STACCATO and CUDD. For almost every circuit, the results clearly show that the memory necessary to represent the symbolic kernels and the decomposition trees is minimal. The BDDs that represent the symbolic kernels often produce minimal overhead because they overlap with the initial BDDs that represent the circuit's functionality. As for the decomposition trees, in the worst case, the number of DSD nodes is linearly related to the number of variables in the support of a function, in contrast with BDDs, which can potentially be exponential in the size of the support.

For the sake of brevity, we have excluded information comparing the runtime performance of STACCATO and CUDD. In general, we found that for most of the circuits the time required by CUDD to build the initial BDDs is comparable to the time required by STACCATO to construct the decomposition trees from these BDDs. More specifically, for circuits that have many decomposable outputs, STACCATO usually constructs decomposition trees faster than it takes CUDD to build the BDDs of circuits. For circuits that have no decomposable outputs, STACCATO performs comparatively slower.

V-B. Symbolic Generalized Cofactor and Merging

Circuit	in	out	dec.out.	blocks	%+BDD	% DEC
C1355	41	32	0	32	0.0	2.7
C1908	33	25	7	94	8.7	6.0
C2670	233	140	119	453	5.1	7.2
C3540	50	22	14	56	0.4	1.8
C499	41	32	0	32	0.0	2.7
C7552	207	108	107	518	41.6	4.2
FIFO	149	285	279	1027	4.9	12.5
am2901	95	150	138	280	22.9	0.5
dcu	700	265	231	3118	32.7	22.9
freecell	486	957	879	9963	16.4	1.9
hwb30	30	1	0	1	0.0	1.0
i10	257	224	224	2241	59.3	2.1
s15850.1	611	684	651	8877	30.4	10.0
s38417	1664	1742	1484	7215	30.4	1.5
s38584.1	1464	1730	1611	6153	33.2	33.3
s4863	153	88	66	302	3.1	1.0
s6669	322	269	194	1119	5.8	6.0
smu	435	142	104	709	16.2	17.5
vcordic	73	144	142	1749	1.8	1.1

TABLE I- BENCHMARKS.

Figure 4 evaluates the performance impact of the various components of our algorithm. The first bar from the left in the graph represents the STACCATO implementation described in this paper. This is compared to three limited-feature variants to evaluate the individual performance contribution of symbolic generalized cofactor and symbolic merging. The execution time of these variants is normalized to STACCATO. The first variant, corresponding to the second bar in the graph, *No Symbolic Kernel Computation*, does not compute any symbolic kernels and therefore cannot compute generalized cofactors symbolically, which is equivalent to the algorithm in [3]. The second variant of STACCATO, denoted by the third bar in the graph, *STACCATO -Symbolic GC*, computes symbolic kernels, but does not exploit them for symbolic generalized cofactor computations. This variant illustrates the performance impact of symbolic generalized cofactors alone. The third variant, corresponding to the fourth bar, *STACCATO -Symbolic Merging*, uses symbolic kernels and symbolic generalized cofactors, but it does not implement the on-the-fly symbolic merging algorithm described in Section IV-D, instead it builds the kernels from the resulting decomposition tree as discussed in Section IV-B, which involves substituting a representative variable for each actuals list member.

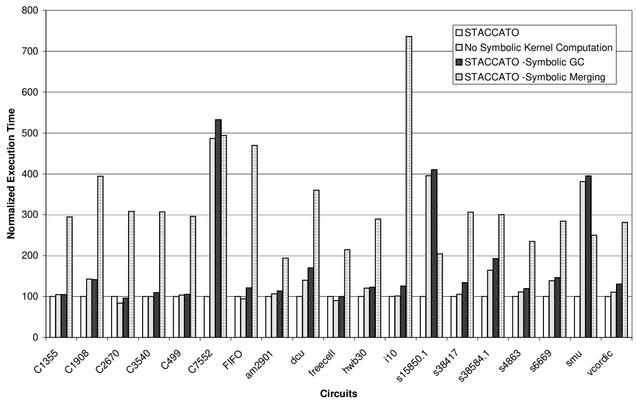


Fig. 4. Execution Time of Different Designs Normalized by STACCATO

The results show that STACCATO outperforms the first variant even though it does not need to compute any kernel. The savings stem from the symbolic generalized cofactoring that requires symbolic kernels. Accordingly, the second variant directly illustrates what type of performance degradation occurs when symbolic generalized cofactoring is not used. The results without symbolic merging show that the approach to construct kernels used in this paper far exceeds the performance of computing the kernel as discussed in Section IV-B.

VI. CONCLUSIONS

We presented a novel algorithm, STACCATO, that computes all the decompositions of a Boolean function while generating the symbolic kernels, the internal component functions of the decompositions, at practically no additional computational cost. We showed that computing symbolic kernels for all the blocks in the decomposition tree can speed up the disjoint support decomposition algorithm while involving minimal memory overhead, as we showed in our experimental results. Moreover, symbolic kernels find interesting application in technology mapping where they constitute an optimal library for minimal-interconnect synthesis. They could also enable Boolean manipulation directly on the decomposition tree, thus eliminating the need for the initial BDD construction.

VII. REFERENCES

[1] Robert L. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, Part I 29, pages 74–116, 1957.

[2] V. Yun-Shen Shen, Archie C. McKellar, and Peter Weiner. A fast algorithm for the disjunctive decomposition of switching functions. *IEEE Transactions on Computers*, C-20(3):304–309, 1971.

[3] Valeria Bertacco and Maurizio Damiani. The disjunctive decomposition of logic functions. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 78–82, November 1997.

[4] Valeria Bertacco. *Achieving Scalable Hardware Verification with Symbolic Simulation*. PhD thesis, Stanford University, 2003.

[5] Yusuke Matsunaga. An exact and efficient algorithm for disjunctive decomposition. In *SASIMI*, pages 44–50, October 1998.

[6] Kevin Karplus. Using if-then-else dags to do technology mapping for field-programmable gate arrays. Technical Report UCSC-CRL-90-43, Baskin Center for Computer Engineering & Information Sciences, 1990.

[7] Tsutomu Sasao. Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays. *IEEE Transactions on Computers*, C-30(9):635–643, September 1981.

[8] Rajeev Murgai, Yoshihito Nishizaki, Narendra V. Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *DAC, Proceedings of Design Automation Conference*, pages 620–625, June 1990.

[9] Thomas Kutzschebauch and Leon Stok. Layout driven decomposition with congestion consideration. In *DATE, Design, Automation and Test in Europe Conference*, pages 672–676, March 2002.

[10] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proceedings of Design Automation Conference*, June 2002.

[11] Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, and Stefano Quer. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *DAC, Proceedings of Design Automation Conference*, pages 728–733, June 1997.

[12] Tsutomu Sasao and Munehiro Matsuura. DECOMPOS: An integrated system for functional decomposition. In *International Workshop on Logic Synthesis*, pages 471–477, 1998.

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

[14] Kevin Karplus. Using if-then-else dags for multi-level logic minimization. In *Proceedings of Advanced Research in VLSI*, pages 101–118, 1989.

[15] Yung-Te Lai, Kuo-Rueih R. Pan, and Massoud Pedram. Obdd-based functional decomposition: algorithms and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):977–990, August 1996.

[16] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 126–129, November 1990.

[17] Randal E. Bryant. Symbolic boolean manipulation with ordered binary–decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[18] Sun Microsystems. PicoJava technology. *http://www.sun.com-/microelectronics/communitysource/picojava*.

[19] CUDD-2.3.1. *http://vlsi.Colorado.edu/~fabio*.