KTH Information and
Communication Technology

Advances in Functional Decomposition:

Theory and Applications

ANDRÉS MARTINELLI

Doctoral Dissertation

Department of Electronic, Computer, and Software Systems
School of Information and Communication Technology
Royal Institute of Technology (KTH)

Stockholm, Sweden 2006

TRITA-ICT/ECS AVH 06:06
ISSN 1653-6363
ISRN KTH/ICT/ECS AVH-06/06--SE

KTH-ICT-ECS
SE-164 40 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av doktorsexamen torsdagen den 12 oktober 2006 kl 9.00 i Sal E, KTH-Forum, plan 5, Isafjordsgatan 39, Kista, Stockholm.

Abstract

Functional decomposition aims at finding efficient representations for Boolean functions. It is used in many applications, including multi-level logic synthesis, formal verification, and testing.

This dissertation presents novel heuristic algorithms for functional decomposition. These algorithms take advantage of suitable representations of the Boolean functions in order to be efficient.

The first two algorithms compute *simple-disjoint* and *disjoint-support* decompositions. They are based on representing the target function by a Reduced Ordered Binary Decision Diagram (BDD). Unlike other BDD-based algorithms, the presented ones can deal with larger target functions and produce more decompositions without requiring expensive manipulations of the representation, particularly BDD reordering.

The third algorithm also finds disjoint-support decompositions, but it is based on a technique which integrates circuit graph analysis and BDD-based decomposition. The combination of the two approaches results in an algorithm which is more robust than a purely BDD-based one, and that improves both the quality of the results and the running time.

The fourth algorithm uses circuit graph analysis to obtain *non-disjoint* decompositions. We show that the problem of computing non-disjoint decompositions can be reduced to the problem of computing multiple-vertex dominators. We also prove that multiple-vertex dominators can be found in polynomial time. This result is important because there is no known polynomial time algorithm for computing all non-disjoint decompositions of a Boolean function.

The fifth algorithm provides an efficient means to decompose a function at the circuit graph level, by using information derived from a BDD representation. This is done without the expensive circuit re-synthesis normally associated with BDD-based decomposition approaches.

Finally we present two publications that resulted from the many detours we have taken along the winding path of our research.

Contents

List of Figures

List of Tables

Acknowledgments

Thanks to all my colleagues at the Department of Electronic, Computer and Software systems at KTH, for so many interesting discussions and refreshing cups of tea. Thanks to Lena Beronius, for all her patience and her help in making the paperwork look human. Thanks to my first supervisor, Mads Dam, for his generosity. Thanks to Babak Sadighi, from the Swedish Institute of Computer Science, for always believing in me.

Thanks to my dearest old friends Pablo Giambiagi, Lars-Åke Fredlund and Elaine Vieira. I would not have survived this journey without them.

My deepest, and warmest thanks to my four mothers: Patricia Mac Elroy, Marina Villegas, Rosita Wachenchauzer, and Elena Dubrova. Patricia is my mum, and I am the person I am today because of her. Marina is my scientific mother; she showed me early in life that pursuing a scientific career was certainly a wonderful prospect. Rosita is my computer science mother, who introduced me to the delicious intricacies of theoretical computing. Elena is my supervisor, and I reached this point because of her patience, support, encouragement and good will. This thesis is dedicated to them.

Chapter 1

Introduction

This dissertation is a collection of papers I have published during my work as a PhD student at KTH. All, except the last one, are concerned with the manipulation of Boolean functions typically used to model problems at the logic synthesis step of the integrated circuit design flow. The last one is a peep into the future, as it proposes an idea that will surely put to the test our current conceptions and assumptions about computing devices.

From a "historical" perspective, many of the ideas presented in this dissertation were born "on the move", while I was traveling with other members of my research group. The idea for *Paper C* came to our minds while traveling by boat to Grinda island in Stockholm's archipelago. Summer is always a good time in Sweden to take the whole group to a more inspiring environment for a group meeting. We were so absorbed into the discussion that we missed the boat stop at Grinda, and had to get off on the next stop, Göllna island; which turned out to be even better for a day's trip. This paper, and the algorithm presented therein, are known within our group by the nickname *Grinda*.

Another idea which was born "on the fly" was the one that resulted in *Paper G*. We were returning from the DATE 2005 conference in Munich. Inspired by the presentations on emergent technologies, we realized that in Kauffman networks we had a starting point for creating a computational device based on the gene regulatory networks of living cells. Until that moment we had only been looking into the subject from a biologist's perspective, and trying to help with the simulation of Kauffman networks of large size.

The idea that has traveled with us the longest is the one presented in

Paper F. About four years ago, we found a mistake in a proof of a statement related to best orderings for a Binary Decision Diagram. Since then we struggled to find an alternative proof. This topic, although related, was not the main line of our work, so we mostly discussed it during conference trips, over a beer, and on bowling or billiard sessions which we often did together. Maxim Teslenko looked devastated the morning he showed up with a counter-example overthrowing this hypothesis that was believed to be true in the CAD community for over fifteen years.

This is not to say that all these ideas came to us easily, without hard work. They would not have come unless we had done a lot of reading and processing of piles of existing literature, nor would they have come if there had not been an excellent communication and mutual understanding among us as members of a research group. The ideas were born in a kind of environment that encouraged, and I would dare say was fundamental, for productive research work.

Now that this dissertation brings a certain kind of closure to my life, a sensation of a circle completed, I only hope to be able to keep sharing the kind of experiences that brought me to this point. And may good research and generous colleagues be a constant in my future life.

1.1 Context and Motivation

This dissertation revolves around the concept of a discrete function, particularly what is known as a Boolean function. It focuses on the problem of breaking apart such a function as a composition of hopefully simpler functions.

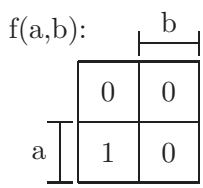
Functional Decomposition

What do we mean by functional decomposition?

In a general sense, functional decomposition refers to the various ways in which a function can be defined in terms of building blocks. This is different from the well known tabular definitions, like the *truth table* or the *Karnaugh map* depicted below.

a	b	f(a,b)
0	0	0
0	1	0
1	0	1
1	1	0

(a) Truth table



(b) Karnaugh map

These are definitions of a function "by extension". As such, they suffer from the most basic problem when dealing with discrete functions: they are very large. These representations explicitly assign a particular output value to each of the possible combinations of input values. When we consider that a given Boolean function of n variables accepts 2^n different combinations of input values, we start realizing the problem.

It is known, however, that a certain set of basic functions can be used in a "compositional" way to build any other possible (and more complex) function [1].

[1] This is known since Boole's ground breaking "Laws of thought" [27], published in 1858.

Let us see a common set of basic functions or *operators* that can represent any complex Boolean function.

1. The *identity* function.

a	$\mathrm{f}(a) = a$
0	0
1	1

2. The *negation*, or "not" function (noted as a bar).

a	$\mathrm{f}(a) = \bar{a}$
0	1
1	0

3. The *conjunction*, or "and" function (noted as a dot).

a	b	$\mathrm{f}(a, b) = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

In terms of these simple elements, the function described before in our truth table example can be represented as

$$f(a, b) = a \cdot \bar{b}.$$

This is a mathematical "composition" of some basic operators, something that can be more clearly seen if we change the shorthand algebraic notation to a more verbose functional style,

$$f(a, b) = \mathrm{and}(a, \mathrm{not}(b)).$$

We have actually performed a "decomposition" of a function into simpler components: "and", "not", and single variables a and b.

We can use other sets of "operators". For example,

1. The *identity* function.

a	$\mathrm{f}(a) = a$
0	0
1	1

2. The *negation*, or "not" function.

a	$\mathrm{f}(a) = \bar{a}$
0	1
1	0

3. The *disjunction*, or "or" function (noted as $+$).

a	b	$\mathrm{f}(a,b) = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

In this case, our example function will be represented as the following "composition" of the single variables a and b, with the "or" and "not" operators:

$$f(a,b) = \mathrm{not}(\mathrm{or}(\mathrm{not}(a), b)).$$

Or, in shorthand algebraic notation:

$$f(a,b) = \overline{\bar{a} + b}.$$

Note that these are two different representations of the *same* Boolean function.

Any given function can be decomposed in many ways[2], depending on how we choose the basic building blocks. Even for the same building blocks we may have different ways to express a function. For example, for the first set of operators:

$$f(a,b) = a \cdot \bar{b} = a \cdot \overline{\overline{\overline{b}}} = \overline{\overline{\overline{a \cdot a}} \cdot \bar{b}} = \cdots .$$

Within the specific context of this dissertation we will call "decomposition" or "functional decomposition" to that kind of decomposition which expresses a function with respect to certain building blocks, but we will not

[2] Actually in an infinite number of ways.

make any particular assumptions on the complexity or variety of our building blocks. For example, a four variable function f may be decomposed as

$$f(w, x, y, z) = h(w, g(x, y, z))$$

or

$$f(w, x, y, z) = h(g_1(w, x), g_2(y, z))$$

or

$$f(w, x, y, z) = h(g_1(w, x, y), g_2(x, y, z))$$

for certain functions h, g, g_1 and g_2 of arbitrary complexity.

We will categorize our decompositions into different classes depending on the depth of the nesting in the resulting formula ("two level" or "multi-level"), the sharing of variables among the different support sets ("disjoint" or "non-disjoint"), the means by which they were obtained ("Algebraic" or "Boolean"), or others. The first two decompositions in our example above are what we call "disjoint" decompositions, while the third one is what we call a "non-disjoint" decomposition. Each of the specific classes of decomposition we target in our work will be introduced later on, when we review the contributions of this dissertation.

Whichever the application domain is, the cost of using algorithms that in some way manipulate or depend on discrete functions seriously depends on the "complexity" of those functions. Decomposition techniques are recognized to reduce such complexity, even though the exact meaning of a "complex" function varies along the different domains of application. It is not the aim of this dissertation to discuss the suitability of decomposition in this respect, but rather to address the practical issues involved in producing such decompositions for logic synthesis applications.

Finding different decompositions for a given function is known to be a hard problem. Hard in the sense that we will always encounter a particular function whose analysis will exceed our time or space constraints. Finding all useful decompositions is, in most cases, unfeasible for large functions, so different approaches will each produce only a subset of decompositions in a reasonable time or within a reasonable space. It is this difficulty that calls for a battery of new and improved heuristics and algorithms to tackle the problem of decomposition in the most efficient way.

Logic Synthesis

Logic synthesis is a step in the computer-aided design (CAD) flow of integrated circuits. It plays a significant role in determining the overall circuit quality. In this section we establish a context for this problem, and briefly review previous synthesis efforts.

Very Large Scale Integration (VLSI) technology has been the key enabler for implementing modern digital systems. Today's microprocessors, memories, and application-specific integrated circuits (ASICs) are the beneficiaries of a steady doubling, over the last thirty years, of transistor counts every 18 months (known as Moore's law). This unprecedented increase in integration levels has led to dramatic reductions in production costs and significant increases in performance and functionality. The design of such highly complex systems was also critically dependent on the use of CAD tools in all steps of the design process: synthesis, optimization, analysis, and verification. This dissertation addresses one of the synthesis steps in this automatic design flow, namely the creation of a low-level structural description of a design from a more abstract form. The major synthesis steps in this design flow are depicted in Figure 1.1.

The starting point of design synthesis is typically a textual description of the desired functional behavior, written in an appropriate hardware description language (HDL). At this level, the design is specified in terms of abstract data manipulation operations which are organized into larger blocks using control constructs. High-level synthesis transforms such a description into an appropriate structural representation at the register-transfer level (RTL). Typical RTL components include data storage elements (registers, memories, etc.), functional modules (adders, shifters, etc.), and data steering logic (buses, multiplexers, etc.). The next major synthesis step creates multi-level logic gate realizations for each of the combinational (i.e. memory-less) parts of the RTL description.

Such multi-level logic synthesis is the primary application area of this dissertation. The primitive building blocks used in such synthesis are typically 3- to 4-input single-output cells from a precharacterized technology library. The final synthesis step generates a complete layout of the design by placing and routing its gate-level implementation, and by synthesizing a suitable power/ground distribution grid and a clock tree. Each of the above synthesis steps (high-level, logic, and physical) involves a multiple-objective optimization that seeks an appropriate trade-off among the design's area,

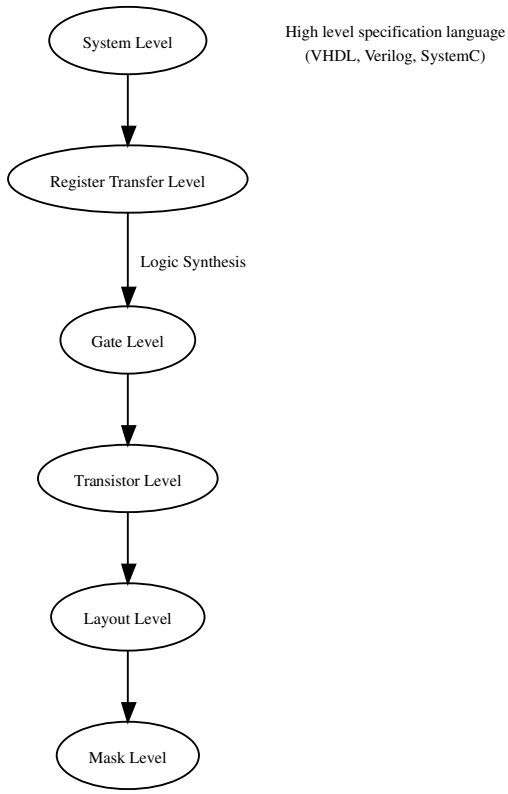


Figure 1.1: Major synthesis steps in the design of digital integrated circuits.

delay, testability, and more recently, power consumption. Area minimization leads to increased chip yields, and hence lower costs, as smaller circuits can be manufactured more reliably, and are easier to fit on a chip; smaller circuits also often have decreased delay. Delay minimization creates faster circuits which are essential in high-performance computing applications. Improving the testability properties of a circuit can lead to higher reliability and reduced testing costs. Finally, minimizing power consumption has become crucial with the proliferation of hand-held and portable computing devices, and is becoming a major issue in high-performance designs as well. These design objectives interact in complex ways. Synthesizing a circuit that optimizes across a set of these objectives is a difficult task due to the

tremendously large space of potential solutions. Finding a solution in this space that meets the specified objectives may, therefore, be computationally expensive, if not impossible. In the face of such complexity, most synthesis approaches resort to a serialization of the design creation process by approximating, or entirely ignoring, some of the contributing components of the various optimization objectives. For example, in physical synthesis, layout generation is serialized into the steps of placement, global routing, and detailed routing. Placement is done by making certain assumptions about the routing requirements and the resulting placement solution becomes a constraint for the subsequent routing optimization. In most cases, this is an acceptable strategy that yields good layouts. In some cases, however, the placement constraints preclude the successful routing of the design or lead to routing solutions that do not meet the delay objectives. In such cases, it is necessary to iterate the placement/routing steps until an acceptable solution meeting all objectives is found. This same serialization paradigm is currently the predominant way for dealing with the complexity of multi-level logic synthesis. Specifically, the synthesis process is split into two phases: a technology-independent global restructuring of the RTL logical specifications followed by a technology mapping of the resulting structure to a specified cell library. The technology-independent optimizations work on logic representations that do not directly model, and hence are unconstrained by, the particular primitive building blocks in this library. The technology mapping phase, on the other hand, is constrained by the structure produced in the technology-independent phase and can only achieve local optimizations as it makes choices to produce the gate-level implementation. Iteration between these two phases may, therefore, be necessary to satisfy all optimization objectives, especially delay. There are two fundamental concepts influencing research in multi-level synthesis, as well as synthesis in general: derivation of flexibility in the implementation of a design, and exploiting this flexibility when optimizing the implementation. One source of flexibility is the incomplete specification of a design, or the parts within it. Thus, the implementation changes remain consistent with the specification. The other source of flexibility is invariant transformations which leave the behavior of the actual implementation unchanged. Most research has been done regarding the second source of flexibility as it perceived to be a more difficult problem and to have a more significant impact on the design quality.

Chapter 2

Background

This chapter presents the general mathematical background needed for this dissertation. Background material that is specific to a particular chapter is introduced in the corresponding chapter.

2.1 Basic Notation

We let $\mathbb{M} = \{0, 1, \ldots, m-1\}$ be an arbitrary finite set of values, and a set of Boolean values is denoted by $\mathbb{B} = \{0, 1\}$. We use early lower-case letters a, b, c, a_1, a_2, etc. to denote elements over a finite set, and lower-case letters f, g, h, g_1, g_2, etc to denote functions. We use $x_1, x_2, \ldots, x_n$ to denote variables that functions may depend on. We use capital letters A, B, C, etc for vectors or sets, and usually denote the elements of the set by indexed lower-case letters. For example, the elements of a set A are denoted as a_1, a_2, etc.

2.2 Sets, Relations, and Functions

There are many excellent books providing comprehensive coverage of set theory. Among those are two classic works by Fraenkel [71] and Halmos [79]; they are suggested for further reading, as this section provides only the minimum notation and definitions needed to motivate further concepts.

Sets

A *set* is a collection of objects called elements, or members. If a is a *member* of set A then we write $a \in A$; similarly *subset* membership is denoted with $A \subseteq B$, whenever for every element in $x \in A$ we have also $x \in B$. If A is a *proper* subset (or *strict* subset) of B, i.e. $A \subseteq B$ and $A \neq B$, we denote it by $A \subset B$. The number of elements in set A will be denoted by $|A|$.

A *partition* P of a given set S is a set $P = \{S_0, \ldots, S_{n-1}\}$ such that $\bigcup_{i=0}^{n-1} S_i = S$ and $\forall i, j, i \neq j, S_i \cap S_j = \emptyset$.

Relations

Let A and B be sets. A *binary relation* R between A and B is a subset of the Cartesian product $A \times B$. We use the notation aRb to denote that $(a, b) \in R$.

Binary relations represent relationships between the elements of two sets. A more general type of relation is the n-ary relation, which expresses relationships among elements of more than two sets. However, this dissertation uses only binary relations, and therefore we do not introduce n-ary relations. In the following, we use the term *relation* to mean *binary relation*.

Relations from a set A to itself are of special interest. A *relation on* the set A is a relation from A to A, i.e. a subset of $A \times A$.

Let R be a relation on A and let P be a property of binary relations (such as reflexivity, symmetry, or transitivity). The *closure* of R with respect to P is the smallest relation containing R that has property P.

A relation on a set A is called an *equivalence relation* if it is reflexive, symmetric, and transitive. Let R be an equivalence relation on A. The set of all elements b of A such that bRa for an element $a \in A$ is called the *equivalence class* of a. The equivalence classes of R form a partition of A.

Functions

A *function* $f : A \to B$ from A to B is a relation, which has the property that every element $a \in A$ is the first element of exactly one ordered pair (a, b) of the relation. So, a function $f : A \to B$ assigns to each element $a \in A$ a unique element $b = f(a)$ in B, called the *image* of a. A is called the *domain* of f and B is called the *co-domain* of f. The *range* of f is the set of all images of elements of A.

A function $f : A \to B$ can be specified by using a rule $a \mapsto f(a)$, assigning to each element $a \in A$, its image $f(a)$ in B.

The *composition* of two functions $f : A \to B$ and $g : C \to D$, where $D \subseteq A$ is denoted by $g \circ f$, where $(g \circ f)(x) = f(g(x))$.

A function $f : A \to B$ is called *injective* when different elements of A always have different images or, in other words, if and only if $a \neq b$ implies that $f(a) \neq f(b)$.

A function $f : A \to B$ is called *surjective* when the range is the whole co-domain B or, in other words, if and only if for every element $b \in B$ there is an element $a \in A$ with $f(a) = b$.

A function $f : A \to B$ is called *bijective* when there is a one to one correspondence between elements of A and B or, more specifically, if and only if it is both injective and surjective.

Two functions $f : A \to B_1$ and $g : A \to B_2$ are *isomorphic* if, and only if, there exists a bijection $\phi : B_2 \to B_1$ such that $f(X) = \phi(g(X))$.

A surjective function $g : A \to B_2$, $B_2 \subseteq B_1$, is said to be a *projection* of $f : A \to B_1$ if, and only if, for all $x, y \in A$, $g(x) \neq g(y) \Rightarrow f(x) \neq f(y)$. Alternatively, g is a projection of f if, and only if, there exists a surjective function $\sigma : B_1 \to B_2$, such that $g = f \circ \sigma$.

Functions can be used to model set membership. For a subset B of set A such a function is defined as a mapping $\chi : A \to \{0, 1\}$ such that $\chi(a) = 1$ if $a \in B$, and $\chi(a) = 0$ otherwise. We refer to this type of function as the *characteristic function* of the corresponding set.

In a similar manner, functions can be used to model partitions of a set. For a partition $P = \{S_0, \ldots, S_{n-1}\}$ of a set A (see Section 2.2), such a function is defined as a mapping $\chi : A \to \{0, \ldots, n-1\}$ such that $\chi(a) = i$ if, and only if, $a \in S_i$. We also refer to this type of function as the *characteristic function* of the corresponding set, without risk of confusion.

Observe that for a partition $P = \{S_0, \ldots, S_{n-1}\}$, every characteristic function χ induces an *equivalence relation* $\equiv_\chi$ defined as $s \equiv_\chi s'$ if, and only if, $\chi(s) = \chi(s')$. The sets $S_0, \ldots, S_{n-1}$ represent all the *equivalence classes* of $\equiv_\chi$.

2.3 Decision Diagrams

This section gives an introduction to Binary and Multi-Valued Decision Diagrams.

Binary Decision Diagrams

Binary Decision Diagrams (*BDD*s) are rooted, directed acyclic graphs. They were originally proposed by Lee [101] and Akers [2], but were later popularized by Bryant [36], who refined the data structure and presented a number of algorithms for their efficient manipulation. A BDD is associated with a finite set of Boolean variables and represents a Boolean function over these variables. We denote the BDD that represents a function f as $\mathcal{F}$.

The vertices of a BDD are usually referred to as *nodes*. A node v is either *non-terminal*, in which case it is labeled with a Boolean variable $\mathbf{var}(v) \in \{x_1, \ldots, x_n\}$, or *terminal*, in which case it is labeled with either $\mathbf{0}$ or $\mathbf{1}$. Each non-terminal node v has exactly two children, $\mathbf{then}(v)$ and $\mathbf{else}(v)$. A terminal node has no children. The value of the Boolean function f, represented by BDD $\mathcal{F}$, for a given valuation of its Boolean variables can be determined by tracing a path from its root node to one of the two terminal nodes. At each node v, the choice between $\mathbf{then}(v)$ and $\mathbf{else}(v)$ is determined by the value of $\mathbf{var}(v)$: if $\mathbf{var}(v) = 1$, $\mathbf{then}(v)$ is taken (denoted graphically as a solid edge in the graph), if $\mathbf{var}(v) = 0$, $\mathbf{else}(v)$ is taken (denoted graphically as a dashed edge in the graph). Every BDD node v corresponds to some Boolean function f_v . The terminal nodes correspond to the trivial constant functions $f_0 = \mathbf{0}$, $f_1 = \mathbf{1}$. For a function f, variable x_i, and Boolean value b, the *cofactor* $f|_{x_i=b}$ is found by substituting the value b for variable x_i:

$$f|_{x_i=b} = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

An important property of BDDs is that the children of a non-terminal node v correspond to cofactors of function f_v . That is, for every non-terminal node v, $f_{\mathbf{then}(v)} = f_v|_{\mathbf{var}(v)=1}$, and $f_{\mathbf{else}(v)} = f_v|_{\mathbf{var}(v)=0}$. We will also refer to the cofactor of a BDD node v, with the understanding that we mean the BDD node representing the cofactor of the function represented by node v.

A BDD is said to be *ordered* (*OBDD*) if there is a total ordering of the variables such that every path through the BDD visits nodes according to the ordering. Let $\mathbf{index}(x) \in \{1, \ldots, n+1\}$, where $x \in \{x_1, \ldots, x_n\}$ represent such a total ordering. Then for every child v' of a non-terminal node v, either v' is a terminal node or

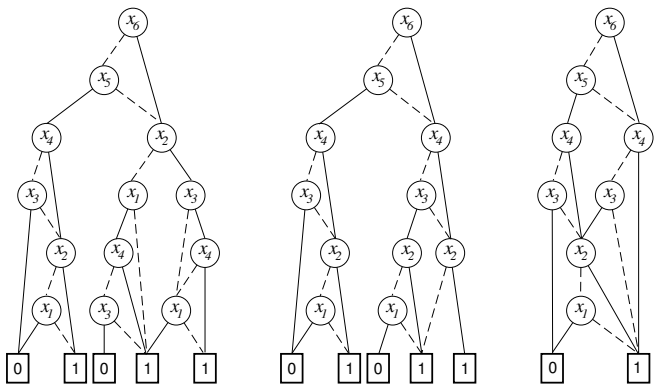
$$\mathbf{index}(\mathbf{var}(v)) < \mathbf{index}(\mathbf{var}(v')).$$

Notice that when we specify a variable ordering $\langle x_0, x_1, \ldots, x_{n-1} \rangle$, we implicitly define $\mathbf{index}(v) = i$, if and only if $\mathbf{var}(v) = x_i$.

When referring to the OBDD representing the function f as $\mathcal{F}$, the variable associated with the top node v of $\mathcal{F}$ is represented also as $\mathbf{topVar}(\mathcal{F})$ (i.e. $\mathbf{topVar}(\mathcal{F}) = \mathbf{var}(v)$).

A *reduced* OBDD (*ROBDD*) is one which contains no redundant nodes, i.e. a non-terminal node labeled with the same variable and with identical children as some other non-terminal node, or a terminal node labeled with the same value as some other terminal node, or non-terminal nodes having two identical children.

Any OBDD can be reduced to an ROBDD by repeatedly eliminating, in a bottom-up fashion, any instances of duplicate and redundant nodes. If two nodes are duplicates, one of them is removed and all of its incoming pointers are redirected to its duplicate. If a node is redundant, it is removed and all incoming pointers are redirected to its unique child.



(a) BDD (b) OBDD (c) ROBDD

Figure 2.1: Example BDDs for the same Boolean function.

Figure 2.1 shows three equivalent data structures, a BDD, an OBDD, and an ROBDD, each representing the same Boolean function, f. Tracing paths from the root node to the terminal nodes of the data structures, we can see, for example, that $f(0,0,1,0,0,1) = 1$ and $f(0,1,0,1,1,1) = 0$. The most commonly used of these three variants is the ROBDD and this will also be the case in this thesis. For simplicity, and by convention, from this point on we will refer to ROBDDs simply as BDDs.

It is important to note that the reduction rules for BDDs described in the previous paragraphs have no effect on the function being represented. They

do, however, typically result in a significant decrease in the number of BDD nodes. More importantly still, as shown by Bryant [36], for a fixed ordering of the Boolean variables, BDDs are a *canonical* representation. This means that there is a one-to-one correspondence between BDDs and the Boolean functions they represent.

The canonical nature of BDDs has important implications for efficiency. For example, it makes checking whether or not two BDDs represent the same function very easy. This is an important operation in many situations, such as the implementation of iterative fixed-point computations. In practice, these reductions are taken one step further. Many BDD packages (e.g. CUDD [144]) will actually store all BDDs in a single, multi-rooted graph structure, known as the *unique-table*, where no two nodes are duplicated. This means that comparing two BDDs for equality is as simple as checking whether they are stored in the same place in memory.

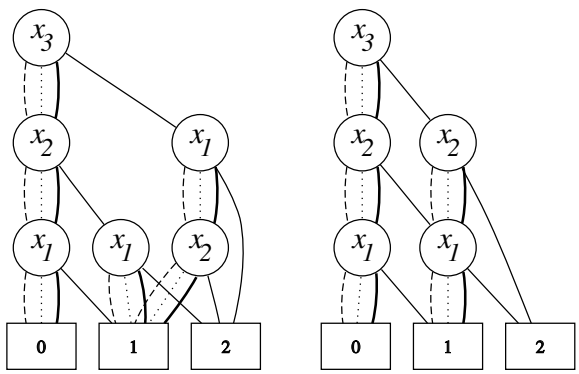
It is also important to note that the choice of an ordering for the Boolean variables of a BDD can have a tremendous effect on the size of the data structure, i.e. its number of nodes. Finding the optimal variable ordering, however, is known to be computationally expensive [25]. For this reason, the efficiency of BDDs in practice is largely reliant on the development of application-dependent heuristics to select an appropriate ordering, e.g. [73]. There also exist techniques such as dynamic variable reordering [131], which can be used to change the ordering for an existing BDD in an attempt to reduce its size.

One of the main appeals of BDDs is the efficient algorithms for their manipulation which have been developed, e.g. [36, 37, 30]. A common BDD operation is the ITE ("If Then Else") operator, which takes three BDDs, $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$, and returns the BDD representing the function $f_{\mathcal{F}} f_{\mathcal{G}} + \bar{f_{\mathcal{F}}} f_{\mathcal{H}}$. The ITE operator can be implemented recursively, based on the property $\mathrm{ITE}(\mathcal{F}, \mathcal{G}, \mathcal{H})|_{x_k=b} = \mathrm{ITE}(\mathcal{F}|_{x_k=b}, \mathcal{G}|_{x_k=b}, \mathcal{H}|_{x_k=b})$.

Multi-Valued Decision Diagrams

Multi-Valued Decision Diagrams (*MDD*s) are also rooted, directed, acyclic graphs [89]. An MDD is associated with a set of k variables, $x_1, \ldots, x_k$, and an MDD $\mathcal{M}$ represents a function $f_{\mathcal{M}} : \mathbb{M}_{x_1} \times \ldots \times \mathbb{M}_{x_k} \rightarrow \mathbb{M}$, where $\mathbb{M}_{x_i}$ is the finite set of values that variable x_i can assume, and $\mathbb{M}$ is the finite set of possible function values. It is usually assumed that $\mathbb{M}_{x_k} = \{0, \ldots, m_k - 1\}$ and $\mathbb{M} = \{0, \ldots, m - 1\}$ for simplicity. Note that BDDs are the special

case of MDDs where $\mathbb{M} = \mathbb{B}$ and $\mathbb{M}_{x_i} = \mathbb{B}$ for all i. MDDs are similar to the "shared tree" data structure described in [163]. Like BDDs, MDDs consist of *terminal* nodes and *non-terminal* nodes. The terminal nodes are labeled with an integer from the set $\mathbb{M}$. A non-terminal node m is labeled with a variable $\mathbf{var}(m) \in \{x_1, \ldots, x_k\}$. Since variable x_i can assume values from the set $\mathbb{M}_{x_i}$, a non-terminal node m labeled with variable x_i has $|\mathbb{M}_{x_i}|$ children, each corresponding to a cofactor $f_m|_{x_i=c}$, with $c \in \mathbb{M}_{x_i}$. We refer to the child c of node m as $\mathbf{child}_c(m)$, where $f_{\mathbf{child}_c}(m) = f_m|_{\mathbf{var}(m)=c}$. Every MDD node corresponds to some integer function. The BDD notion of ordering can also be applied to MDDs, to produce ordered MDDs (*OMDD*s). A non-terminal MDD node m is redundant if all of its children are identical, i.e., if $\mathbf{child}_i(m) = \mathbf{child}_j(m)$ for all $i, j \in \mathbb{M}_{\mathbf{var}(m)}$. Two non-terminal MDD nodes m_1 and m_2 are duplicates if $\mathbf{var}(m_1) = \mathbf{var}(m_2)$ and $\mathbf{child}_i(m_1) = \mathbf{child}_i(m_2)$ for all $i \in \mathbb{M}_{\mathbf{var}(m)}$. Based on the above definitions, we can extend the notion of reduced BDDs to apply also to MDDs. It can be shown [89] that reduced OMDDs (*ROMDD*s) are a canonical representation for a fixed variable ordering. Finally, like BDDs, the number of ROMDD nodes required to represent a function may be sensitive to the chosen variable ordering. Example MDDs are shown in Figure 2.2, all representing the same



(a) MDD (b) ROMDD

Figure 2.2: Example MDDs for the same function.

function over three variables, x_1, x_2, x_3 with $m_1 = m_2 = m_3 = 4$ and $m = 3$. The value of the function is zero if none of the variables has value 1, one if exactly one of the variables has value 1, and two if two or more of the

variables have value 1. Figure 2.2(a) shows an MDD that is not ordered nor reduced, and Figure 2.2(b) shows the ROMDD for the function, for the given variable ordering. Unless otherwise stated, the remainder of the dissertation will assume that all MDDs are ROMDDs.

Chapter 3

Previous work

3.1 Functional Decomposition

Research in the subject of Boolean function decomposition is almost as old as digital circuit engineering. The first major investigation on decomposition was carried out by Ashenhurst [7] in 1959. The basis for the different types of decompositions studied in his work is the *simple disjoint decomposition*, of type

$$f(X) = h(g(Y), Z) \tag{3.1}$$

for Boolean functions $f : \mathbb{B}^{|X|} \to \mathbb{B}$, $g : \mathbb{B}^{|Y|} \to \mathbb{B}$, $h : \mathbb{B}^{|Z|+1} \to \mathbb{B}$. Such a decomposition exists *trivially* for Y given by any singleton set x_i or the whole set $X = \{x_1, x_2, \ldots, x_n\}$.

When f, g and h are Boolean functions then, the original function f specifying an n-input, 1-output 2-valued circuit is replaced by the specifications of two 2-valued circuits, one having $|Y|$ inputs and one output, and the other having $|Z| + 1$ inputs and one output (see Figure 3.1).

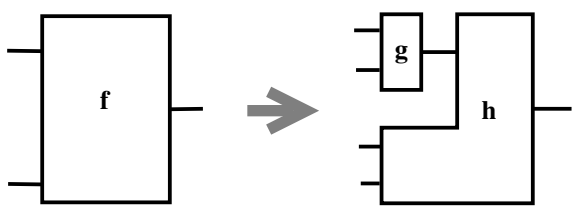


Figure 3.1: Simple disjoint decomposition.

If Ω_n is an upper bound on the cost of realizing a Boolean function of n variables, then the total cost of realizing these two new circuits is bounded above by $\Omega_{|Y|}+\Omega_{(1+|Z|)}$. Because the cost bound Ω_n usually increases nearly exponentially with n [138], the discovery of any nontrivial decomposition of the form (3.1) greatly reduces the cost of realizing f.

The notion of a *bound set* is fundamental in decomposition theory.

Definition 3.1.1*:* Any set of variables Y such that f has a decomposition of type (3.1) is called a *bound set* for f.

Once a decomposition of type (3.1) has been selected, either g, h, or both may be similarly decomposed, giving one of the following *complex disjoint* decomposition types [90]:

$$\begin{array}{ll} \textit{multiple}: & f(X,Y,Z) = h(g(X),k(Y),Z), \\ \textit{iterative}: & f(X,Y,Z) = h(g(k(X),Y),Z), \end{array} \tag{3.2}$$

or more generally *tree-like* decompositions as in

$$f(X,Y,X,W) = h(g(k(X),Y),l(Z),W).$$

Ashenhurst's fundamental contribution is a theorem that states that any Boolean function has a unique *disjoint tree-like decomposition* such that all possible simple disjoint decompositions of f can be derived from it. He proved that any n-variable Boolean function that is *non-degenerate*, i.e. which actually depends on all n variables to determine its output, has a *composition tree*, which is a decomposition reflecting all bound sets, and thus a "most decomposed" one. Hence, the realization of the given function in correspondence with its composition tree (with suitable assumption about the cost of logic elements) should have a cost that is close to minimal. In the sixties it was even conjectured that such an implementation must be a minimal one. However, Paul [126] found a counterexample showing a circuit, derived by a technique other than decomposition, that has smaller cost than the one implementing the composition tree. Such examples seem to be very rare.

Curtis [48] and Roth and Karp [130] extended Ashenhurst theory to decompositions of type

$$f(X) = h(g(Y),Z) \tag{3.3}$$

with g,h being multiple-valued functions of type $g : \mathbb{B}^{|Y|} \rightarrow \mathbb{M}$ and $h : \mathbb{M}\times\mathbb{B}^{|Z|} \rightarrow \mathbb{B}$. The function g can be alternatively encoded by $k = \lceil \log_2 m \rceil$

Boolean functions $g_1, g_2, \ldots, g_k$, giving a decomposition of the form

$$f(X) = h(g_1(Y), \ldots, g_k(Y), Z) \tag{3.4}$$

often referred to as a *disjoint-support* decomposition (see Figure 3.2). In this thesis we call any of these decomposition types a *disjoint-support* decompositions, and may use the multi-valued form or the binary-encoded form as needed.

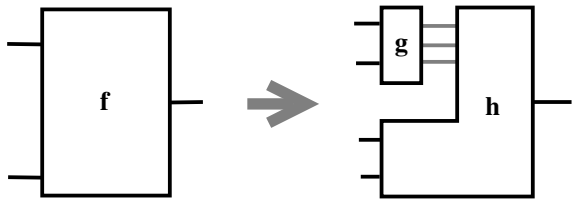


Figure 3.2: Disjoint-support decomposition.

Disjoint-support decompositions define a more general notion of bound set, the k-bound set.

Definition 3.1.2*:* The set Y is said to be a *k-bound set*, with $k > 1$, if k is the minimum value for which there exists a decomposition

$$f(X) = h(g(Y), Z) \tag{3.5}$$

where g are h are surjective functions of type

$$g : \mathbb{B}^{|Y|} \rightarrow \mathbb{M}$$

and

$$h : \mathbb{M} \times \mathbb{B}^{|Z|} \rightarrow \mathbb{B},$$

with $\mathbb{M} = \{0, \ldots, k-1\}$.

Ashenhurst's main theorem does not extend directly to multiple-valued functions (a counterexample can be found in [60]), which means that there is no unique disjoint tree-like disjoint-support decomposition for this type of functions in general. However, Von Stengel [154] has defined a class of multiple-valued functions for which an analogous of Ashenhurst's main theorem holds.

A *Non-disjoint support decomposition* of a Boolean function f is a representation of type

$$f(X,Y,Z) = h(g_1(X,Y),\ldots,g_k(X,Y),Y,Z) \tag{3.6}$$

where X,Y,Z are sets of variables partitioning the support set of f, and h and g_i are Boolean functions of type $g_i : \mathbb{B}^{|X\cup Y|} \to \mathbb{B}$, $i \in \{1,\ldots,k\}$, and $h : \mathbb{B}^{|Y\cup Z|+k} \to \mathbb{B}$.

3.2 Functional Decomposition Algorithms

The classical method for recognizing a bound set is based on representing the function by a *decomposition chart* [7, 48]. The decomposition chart for $f(Y,Z)$ is a two-dimensional table where the columns represent the variables from the set Y and the rows the variables from the set Z. Then Y is a bound set if and only if the chart has *column multiplicity* at most 2, i.e. there are at most 2 distinct columns in the chart.

Figure 3.3 shows such a chart for a Boolean function, for the partitioning of variables $\{\{x_1,x_2\},\{x_3\}\}$, where the set $\{x_1,x_2\}$ is indeed a bound set.

		$x_1\ x_2$			
		00	01	10	11
x_3	0	0	1	1	0
	1	1	0	0	1

Figure 3.3: Decomposition chart for an example Boolean function.

In the case of disjoint-support decompositions, the k-bound sets can be determined by a decomposition chart by relaxing the requirement of having exactly 2 different columns, to allow a number of columns up to k [90].

In the case of non-disjoint support decomposition, a Boolean function with n variables has a simple non-disjoint decomposition of type

$$f(X,Y,Z) = h(g(X,Y),Y,Z)$$

if each of its $2^{|Y|}$ decomposition charts representing sub-functions $f_Y(X,Z)$ has at most two distinct columns. The $2^{|Y|}$ charts are obtained by fixing the variables of Y to all combination of their values from $\mathbb{B}^n$.

Shortly after their introduction, decomposition charts were abandoned in favor of *cube* representation [90], and computing column multiplicity on charts was replaced by computing *compatible classes* for a set of cubes. Two assignments $\hat{x}_1$, $\hat{x}_2 \in \mathbb{B}^{|X|}$ are said to be *compatible* with respect to the reference function $f(X,Y)$ if, for all $\hat{y} \in \mathbb{B}^{|Y|}$ such that $f(\hat{x}_1,\hat{y})$ and $f(\hat{x}_2,\hat{y})$ are defined, $f(\hat{x}_1,\hat{y}) = f(\hat{x}_2,\hat{y})$ [90]. The set X is a k-bound set if and only if $\mathbb{B}^{|X|}$ can be partitioned into $k' \leq k$ mutually compatible classes [90]. If $f(X)$ is completely specified, i.e. total, then compatibility is an *equivalence relation* and k is the number of *equivalence classes*. It is easy to see a one-to-one correspondence between a column in a decomposition chart and a compatible class.

Due to the exponential size of decomposition charts and cube representations, early decomposition algorithms were rarely applied to functions modeling large practical circuits. Instead, *algebraic* methods were used [33]. A milestone work in this subject is due to Brayton and McMullen [33], whom in 1982 introduced the notion of *kernels*, and proposed a method for fast algebraic decomposition based on this notion. The same technique, with minor modifications, is still used today in many systems for multi-level optimization [29, 112, 136].

Binary Decision Diagrams made it possible to develop new algorithms for decomposition, feasible for much larger functions than previously possible. In a BDD, the column multiplicity can be easily computed by moving the variables Y to the upper part of the graph and counting the number of children below the boundary line, usually called *cut* line. The decomposition $f(X) = h(g(Y),Z)$ exists if and only if there are only two children below the cut line [132].

This approach has been adopted by a number of BDD-based decomposition algorithms [132, 99, 41, 135]. Stanion and Sechen [146] used the cut technique to find *quasi-algebraic* decompositions of the form $f(X) = g(Y) \diamond h(Z)$, where "$\diamond$" is any binary Boolean operation and $|Y \cap Z| = k$ for some $k \geq 0$. This type decomposition is often referred to as *bi-decomposition* [159, 119].

Decomposition algorithms following a BDD-cut strategy proved to be orders of magnitude faster than those based on decomposition charts and cube representations. However, they require a reordering of the BDD to move the target set of variables to the top of the graph or to check bi-decompositions for partitions which are not consistent with the variable order. As an alternative, a number of methods use the fact that BDDs themselves are a

decomposed representation of the function and exploit their structure, rather than cut, to find disjoint decompositions. Karplus [91] extended the classical concept of *dominator* on graphs [103] to 0,1-dominators on BDDs. A node v is a 0-dominator if every path from the root to the terminal node labeled **0** contains v. A node v is a 1-dominator if every path from the root to the terminal node labeled **1** contains v. If v is a 1-dominator, then the function represented by the BDD possesses a conjunctive (AND) decomposition. If v is a 0-dominator, then the function can be decomposed disjunctively (OR). This idea was extended by Yang et al [161] to XOR-type decompositions and to more general type of dominators. Minato and De Micheli [118] presented an algorithm which computes disjoint decompositions by generating an irreducible sum-of-product form for the function from its BDD and applying factorization. The algorithm of Bertacco and Damiani [15] makes a single traversal of the BDD to identify the decomposition of the co-factors and then combine them to obtain the decomposition for the entire function. The algorithm is impressively fast; however, as Sasao has observed in [133], it fails to compute some of the disjoint decompositions. This problem was corrected by Matsunaga [113], who added the missing cases in [15] allowing to treat the OR/XOR functions correctly. The algorithm [113] appears to be the fastest of existing exact algorithms for finding all disjoint decompositions.

In recent years, *dominators* reappeared also as the foundation of different decomposition techniques, working on function representations that rely on less constrained circuit graph structures than BDDs. Dominators have been applied to combinational equivalence checking [56], under the name of *proper cuts*, and to testing [137, 17] and design for low power [43], under the names of *headlines* or *supergates*.

3.3 Logic Synthesis

The quest for the automatic synthesis of logic circuits has a long history. In this section we highlight prominent milestones from the last five decades of research and development in this area. We divide the presentation into three parts: early theoretical work in the fifties and sixties, widespread adoption in the seventies and eighties, and modern research efforts.

Early Work

Two-Level Synthesis Synthesis algorithms were first sought for the two-level logic minimization problem. Quine [127] proposed the first solution to this problem in the 1950s; his method was subsequently improved by McCluskey [114], and has since become known as the Quine-McCluskey two-level minimization procedure. The essence of this procedure is a systematic exploration of the search space of two-level circuits seeking a realization with minimal area. The enumerative nature of such an approach makes it exponentially complex in both space and time, and limits its applicability to relatively small functions with, typically, a dozen or fewer inputs. The advantage of two-level forms is that they can be directly implemented in VLSI using programmable logic structures, such as PLAs and PALs [69], whose areas and delays can be estimated with high accuracy. However, general use of two-level synthesis is hampered by the computational infeasibility of optimally synthesizing large functions in two levels, and by the practical technological limits on the maximum fan-in and fan-out of logic gates. In addition, it can be easily shown that certain multi-level realizations are both smaller and faster than the corresponding optimal two-level forms. Despite these shortcomings, exact and approximate two-level synthesis is sometimes used as a step in multi-level synthesis algorithms.

Multi-Level Synthesis Research in multi-level synthesis emerged soon after the initial solutions to the two-level minimization problem were stated. Similar in spirit to those of the two-level problem, the original multi-level approaches were based on a systematic exploration of the solution search space. The dominant view at that time was that two-level circuits were a special case of multi-level circuits, and that the algorithmic solution to the former should generalize to solve the latter. The fundamental notion in multi-level synthesis is that of functional decomposition, studied in this dissertation. As mentioned earlier in Section 3.1, Ashenhurst [7] was the first to derive a condition for checking whether a Boolean function has a non-trivial simple disjoint decomposition. His observation laid the foundation for classical decomposition theory, which was shortly generalized by Curtis [48], and Roth and Karp [130], to handle other, more complex, decomposition forms. These works represent the first accounts of complete multi-level synthesis algorithms. The general approach was a search procedure that examined all possible decompositions lexicographically, pruning the search by some

simple lower bounds on circuit cost, and terminating when a minimum-cost realization was found. Several other enumeration techniques for multi-level synthesis were explored in the 1960s. Hellerman [81] proposed an algorithm that enumerated all directed acyclic graphs, and tested whether each generated graph implements the desired function. The advances in two-level minimization motivated Lawler [100] to generalize the notion of two-level prime implicants to the multilevel case. His approach showed how these multi-level implicants can be used to obtain "absolutely minimal" factored forms. Gimpel [75] proposed an optimal algorithm for synthesis of three-level networks in terms of NAND gates. Gimpel's approach is similar in spirit to the work of Lawler: it generalized the two-level enumeration approach to three levels. Davidson presented a branch-and-bound algorithm for NAND network synthesis [50]. The algorithm constructs a network realization by a sequence of local decisions starting from the primary outputs, and incrementally introduces new gates. Most of this early work on multi-level synthesis, while theoretically significant, failed to achieve the elusive goal of generating optimal circuits. The complexity of exhaustively enumerating the solution space limited the applicability of these approaches to very small circuits, and rendered them impractical for general-purpose synthesis.

Practical Synthesis

The growing complexity of VLSI in the late seventies necessitated new scalable synthesis techniques that sought approximate, rather than optimal, multi-level circuit solutions. Most synthesis tools in use today are based on the premise that the search for optimal solutions is intractable, and are designed, instead, to find acceptable sub-optimal realizations. These tools typically operate on a multi-level representation of the functions being synthesized, continually transforming it until a satisfactory solution is found, and can be roughly classified into two broad categories based on the granularity of transformations used. Local transformation approaches modify the current "solution" incrementally by making appropriate changes in its immediate neighborhood. In contrast, global transformation approaches seek good multi-level topologies by making large-scale changes to the implementation structure while disregarding technological considerations; a second "mapping" phase insures compliance of the resulting multi-level structure with technology constraints. The algorithms presented in this dissertation fall in this category.

Local Transformation approaches Local optimization methods perform rule-based transformations, which are a set of ad hoc rules that are applied iteratively to patterns found in the network of logic gates. In the local optimization method each rule introduces a transformation by replacing a small sub-graph of several gates in the network with another sub-graph which is functionally equivalent but has a simpler realization according to some cost function. Initially the network consists of AND, OR, INV gates, decoders, multiplexers, adders, etc. After the simplification step these primitives are translated into an interconnection of INV to NAND gates through a sequences of transformations. Technology specific transformations are then applied as a final step in the process. Such transformations have limited optimization capability since they are local in nature, and do not have global view on the design. Examples of systems based on this approach are LSS [49] and LORES/EX [85].

Global Transformations approaches The computational limitations of the classical theory for functional decomposition motivated the development of algorithms which are effective in partitioning complex logic functions. These ideas are based on the notion of algebraic factorization applied to *sum-of-products* (SOP) expressions; the technique is described in [33] and [34]. Algebraic decomposition techniques have experienced the most success to date in the field of multilevel synthesis. They are capable of handling large combinational blocks, and produce very good results for control logic. However, representing logic of higher level abstraction with SOP forms makes it difficult to explore the structural flexibility of the original description. It can lead to the loss of a compact description of the original equations, and algebraic decomposition is too restrictive to rediscover their structure. Examples of systems which rely on the algebraic techniques are MIS [32], SOCRATES [9], and more recently SIS [136]. In more recent years much attention has been also given to AND-XOR decompositions [151, 42, 57, 63]. The advent of Binary Decision Diagrams and their variants rekindled interest in classical decomposition techniques. In recent years researchers have successfully applied Roth and Karp decomposition in FPGA synthesis [41, 98, 124, 134, 158]. These approaches decompose a function recursively until each of the generated sub-functions meets a given fan-in constraint, typically 5. However, since fan-in count is the only notion of node complexity in these approaches, they do not extend easily to a

library-specific synthesis. A number of approaches have also been developed which explore the structure of the decision diagram representation of a given function [15, 160, 162, 57].The close relation between BDDs and multiplexer circuits has also lead to several approaches to synthesis of pass transistor logic (PTL) [16, 38, 42, 107]; they are primarily based on a mapping of decomposed BDDs to PTL.

Chapter 4

Contributions in this Dissertation

This chapter reviews the subject matter of the seven publications that make the core of this dissertation. It complements the material given in the publications with additional examples, and includes all proofs omitted in the papers. It also presents an unpublished result which extends the technique described in *Paper B*.

Section 4.1 introduces the first two algorithms, which produce *simple-disjoint* and *disjoint-support* decompositions. They are based on representing the target function as a Binary Decision Diagram. Unlike other algorithms using similar techniques, the ones presented in this thesis can deal with large target functions and produce more decompositions, without requiring expensive manipulations of the representation, particularly BDD reordering.

Different ways of representing a function often lead to very different decomposition alternatives. Two of these alternatives are explored in this dissertation, based on analyzing the circuit graph representation of the target function.

The algorithm presented in Section 4.2 produces disjoint-support decompositions, like the ones obtained by the first two algorithms, but it is based on a technique which integrates circuit graph analysis and BDD-based decomposition. The combination of the two approaches results in a technique which is more robust than the ones based purely on BDD, and that improves both the performance and the quality of the results obtained.

Our fourth algorithm, which efficiently computes *non-disjoint support* decompositions is introduced in Section 4.3.

Section 4.4 presents our fifth algorithm, which provides an efficient means to decompose a function at the circuit graph level, by using information derived from a BDD representation, without requiring expensive circuit re-synthesis.

We end this review of contributions by presenting two publications that resulted from the many detours we have taken along the winding path of our research.

Section 4.5 presents a result of a more theoretical nature. It answers a long standing question regarding the relation between the bound sets of a Boolean function and the "best" variable orders for its BDD representation.

Lastly, a leap into the future closes this list of contributions. In section 4.6 we introduce a novel model of computation, which opens a whole new line of research in the area of molecular circuit implementation, and will surely challenge our knowledge of functional decomposition.

4.1 BDD Based Disjoint-Support Boolean Decomposition

Since the development of BDDs, research on decomposition algorithms got a new life. BDDs allow for larger and more complicated functions to be decomposed. However, regardless of how fast an algorithm is, we are always dealing with a problem that grows exponentially with respect to the number of variables of a function. In *Paper A*, on page 73, we explore an interesting extension to traditional cut methods on BDDs, allowing us to check if any interval of consecutive variables on a BDD is a bound set, without requiring expensive reordering of the BDD variables. This algorithm works specifically for simple disjoint decompositions. Later on, and inspired by this idea, we extend this result to disjoint-support decompositions in *Paper B*, on page 87.

Cutting In order to avoid expensive chart or compatible classes computations, Lai, Pan and Pedram [99] devised a BDD method for checking if a certain set of variables $Y \subset X$ form a bound set for a function $f(X)$. It is based on the property that there exist functions $f_i : \mathbb{B}^{|Z|} \rightarrow \mathbb{B}$, with $Y \cup Z = X$ and $Y \cap Z = \emptyset$ such that

$$f(X) = \sum_{i=0}^{2^{|Y|}-1} \alpha_i(Y) f_i(Z) \tag{4.1}$$

where $Y = \{x_1, x_2, \ldots, x_{|Y|}\}$, $\alpha_i(Y) = x_1^{i_1} x_2^{i_2} \ldots x_{|Y|}^{i_{|Y|}}$, where i_j is the j-th bit of the binary expansion of i, and $x_i^0 = \bar{x}_i$, $x_i^1 = x_i$. The number of different functions in the set $\{f_0, \ldots, f_{2^{|Y|}-1}\}$ is clearly equivalent to the number of compatible classes, or to the number of different columns in a decomposition chart for f (see Section 3.2).

Let $\mathcal{F}$ be the BDD representing f with respect to the variable ordering $\langle x_1, x_2, \ldots, x_n \rangle$. For a given *cut level* c, $1 \leq c < n$, the "upper" part of the BDD $\mathcal{F}$ is the set of nodes $\mathbf{v}$ such that $index(\mathbf{v}) \leq c$. Respectively, the "lower" part of $\mathcal{F}$ is the set of nodes $\mathbf{v}$ such that $index(\mathbf{v}) > c$. We denote by $cut_{\mathcal{F}}(c)$ the boundary line between these to parts. Whenever the BDD $\mathcal{F}$ is clear from the context we write $cut(c)$ instead of $cut_{\mathcal{F}}(c)$.

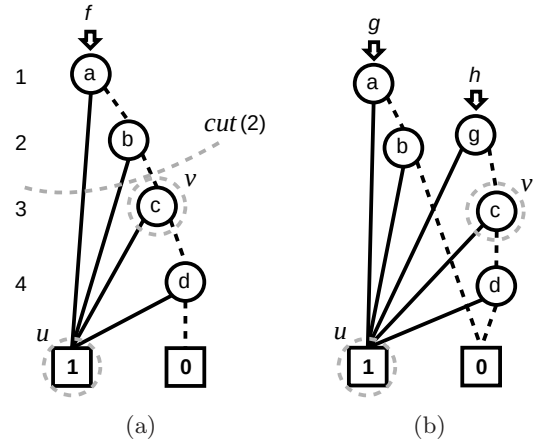


Figure 4.1: Cutting a BDD.

If the set of consecutive[1] variables Y is at the top of the BDD $\mathcal{F}$, the nodes adjacent to and below $cut(|Y|)$ represent the functions f_i for all i. Since BDDs are canonical, the number of these nodes is exactly the number of different functions in the set $\{f_0, \ldots, f_{2^{|Y|}-1}\}$ corresponding to equation (4.1). Thus, the set Y is a bound set for f, if and only if there are at most two nodes adjacent to and below $cut(|Y|)$.

Figure 4.1 gives an intuitive idea of this method. The gray line in Figure 4.1(a) shows $cut(2)$, and the nodes adjacent to and below the cut are encircled in gray. In this example $\{a, b\}$ is a bound set for

$$f(a, b, c, d) = a + b + c + d.$$

With respect to equation (4.1), the cut nodes u and v represent

$$f_1 = f_2 = f_3 = \mathbf{1},$$

and

$$f_0 = c + d.$$

Also note that each function α_i is represented by a path from the root to a node below the cut, e.g. $\alpha_0 = a^0\, b^0 = \bar{a}\,\bar{b}$ is represented by the dotted path from the root to node v.

[1]Consecutive with respect to the BDD variable order. For example, for the ordering $\langle x_1, x_2, \ldots, x_n \rangle$, the set $\{x_2, x_3, x_4\}$ is a set of consecutive variables, but the set $\{x_2, x_4\}$ is not.

The sub-functions g and h of decomposition $f(a,b,c,d) = h(g(a,b),c,d)$ are easily obtained from the BDD as shown in Figure 4.1(b):

$$g(a,b) = a + b,$$
$$h(g,c,d) = g + c + d.$$

Slicing Although cutting a BDD renewed the hopes of practical application of Boolean decomposition, it has one essential drawback: the set of variables to be checked has to be at the top of the BDD. For example, a BDD with variable ordering $\langle x_1, x_2, \ldots, x_n \rangle$ only allows us to check the sets $\{x_1, x_2\}$, $\{x_1, x_2, x_3\}$, $\{x_1, x_2, x_3, x_4\}$ and so on. If this is not the case, the BDD must be reordered. Not only is reordering computationally expensive, but it can also lead to an ordering of the variables that causes the BDD to blow up in size, and thus calculating the cut becomes unfeasible. Bryant shows in [36] a classical example. The BDD for the function $f = x_1x_2 + \cdots + x_{2n-1}x_{2n}$ has $2n+2$ nodes for the variable order $\langle x_1, x_2, \ldots, x_{2n-1}, x_{2n} \rangle$, whereas the size increases to 2^{n+1} nodes for the order $\langle x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n} \rangle$.

In *Paper A* we attacked the reordering problem by devising a method that is similar to the "cutting" method in the previous section, but which allows to check if *any* interval of consecutive variables of a BDD forms a bound set for the function f. Since it is not limited to ranges of variables starting at the top of the BDD, in contrast to the previous method, it allows to check $O(n^2)$ bound set candidates instead of $O(n)$ without requiring reordering. We call this method *slicing*, since two cuts are required to delimit the interval of variables to check (a *slice* of the BDD).

Recall from the previous section that if we partition the support set of f into two disjoint sets Y and Z, we can represent f as shown in equation (4.1). Consider an abstract picture of a BDD $\mathcal{F}$ of an n-variable function $f(X)$ shown in Figure 4.2. Two cut lines on levels a and b of the BDD are denoted by $cut(a)$ and $cut(b)$, $a, b \in \{0, \ldots, n\}$, $a < b$. Let Y be the set of variables which lies between the cut lines, Z_1 be the set of variables above $cut(a)$ and Z_2 be the set of variables below $cut(b)$. We have $X = Y \cup Z$, $Y \cap Z = \emptyset$, and $Z = Z_1 \cup Z_2$, $Z_1 \cap Z_2 = \emptyset$.

Let $cut\_set(a)$ denote a set of nodes $v \in \mathcal{F}$ with indexes $a < index(v) \leq b$ which are children of the nodes of $\mathcal{F}$ above the $cut(a)$. Let $\mathcal{F}_v$ stand for the BDD rooted at $v \in cut\_set(a)$. Then, $cut\_set(b_v)$ is the set of nodes $u \in \mathcal{F}_v$

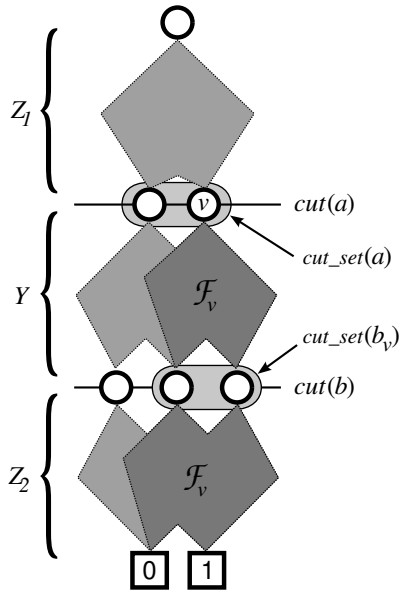


Figure 4.2: Abstract view of a BDD slice.

with indexes $b < index(u) \leq n + 1$ which are children of the nodes of $\mathcal{F}_v$ above the $cut(b)$.

Let $\alpha_v(Z_1)$ be a function representing the sum of all paths of $\mathcal{F}$ leading to a node $v \in cut\_set(a)$. Then f can be co-factored with respect to α_v as

$$f(X) = \sum_{\forall v \in cut\_set(a)} \alpha_v(Z_1) \cdot f|_{\alpha_v}(Y, Z_2). \tag{4.2}$$

If $|cut\_set(b_v)| = 2$, then Y is a bound set for $f|_{\alpha_v}$, and $f|_{\alpha_v}$ can be decomposed as

$$f|_{\alpha_v}(Y, Z_2) = h_v(g_v(Y), Z_2), \tag{4.3}$$

for some $h_v : \mathbb{B}^{|Z_2|+1} \rightarrow \mathbb{B}$ and $g_v : \mathbb{B}^{|Y|} \rightarrow \mathbb{B}$. The function g_v is represented by the BDD rooted at v whose terminal nodes are obtained by replacing the two nodes of $cut\_set(b_v)$.

Using this notation, we can formulate the following theorem.

Theorem 1. *A set of variables Y is a* bound set *for $f(X)$ if, and only if:*

1. *for all $v \in cut\_set(a)$, Y is a bound set for the co-factor $f|_{\alpha_v}(Y, Z_2)$ in* (4.2)*, and*

2. *for all pairs* $v, u \in cut\_set(a)$*, sub-functions* $g_v(Y)$ *and* $g_u(Y)$ *in* (4.3) *are either equivalent, or complement of each other.*

Proof. See the proof of Theorem 8 of *Paper A*, on page 73 of this thesis. The formulation of Theorem 1 differs from the one of Theorem 8, but their essence is the same. □

Figure 4.3 illustrates this theorem. In this example $\{b, c\}$ is a bound set for $F(a, b, c, d) = a \oplus (b + c) \oplus d$. The gray lines in Figure 4.3(a) show the "slice" delimited by $cut(1)$ and $cut(3)$. The sub-functions g and h of the decomposition are easily obtained from the BDD, as shown in Figure 4.3(b).

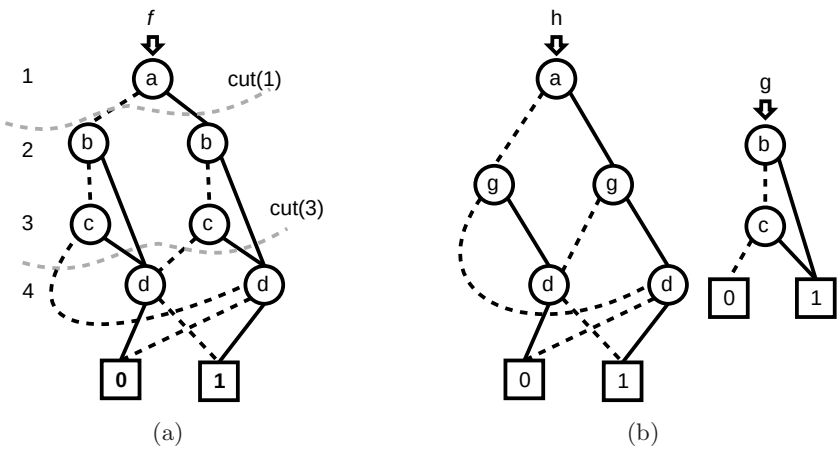


Figure 4.3: Slicing a BDD.

Since two functions are equivalent, or complement of each other if, and only if, their BDD representations are graph isomorphic up to the terminal nodes, this method can be implemented very efficiently. See *Paper A* on page 73 for details on the algorithm and experimental results.

Disjoint-Support Slicing In *Paper B*, we have generalized the result of *Paper A* to disjoint-support decompositions.

Consider again Figure 4.2. If, for some node $v \in cut\_set(a)$, we have $|cut\_set(b_v)| = k$, then Y is a k-bound set for $f|_{\alpha_v}$ in (4.2) and $f|_{\alpha_v}$ can be

decomposed as

$$f|_{\alpha_v}(Y, Z_2) = h_v(g_v(Y), Z_2), \tag{4.4}$$

for some $h_v : \mathbb{B}^{|Z_2|} \times \mathbb{M} \to \mathbb{B}$ and $g_v : \mathbb{B}^{|Y|} \to \mathbb{M}$, where $\mathbb{M} = \{0, 1, \ldots, k-1\}$. The function g_v is represented by the MDD rooted at v whose k terminal nodes are obtained by replacing the nodes of $cut\_set(b_v)$.

We can extend Theorem 1 to the k-bound set case as follows:

Theorem 2. *A set of variables Y is a* k-bound set *for $f(X)$ if:*

1. *for all $v \in cut\_set(a)$, Y is a k-bound set for the co-factor $f|_{\alpha_v}(Y, Z_2)$ in (4.2), and*
2. *for all pairs $v, u \in cut\_set(a)$, sub-functions $g_v(Y)$ and $g_u(Y)$ in (4.4) are isomorphic.*

We present a proof of the disjoint-support slicing technique which is not included in *Paper B*.

Proof. Since Y is a k-bound set for all $f|_{\alpha_v}$, each of $f|_{\alpha_v}$ can be decomposed as in (4.4). Furthermore, since all sub-functions $g_v(Y)$ are isomorphic, we can also decompose $f|_{\alpha_v}$ as

$$f|_{\alpha_v}(Y, Z_2) = h_v(g(Y), Z_2). \tag{4.5}$$

where $g(Y) = \phi_v(g_v(Y))$ for some bijection $\phi_v : \mathbb{M} \to \mathbb{M}$.

From (4.2) and (4.5) we can conclude that f can be represented as

$$f(X) = h(g(Y), Z),$$

with $h = \sum_{\forall v \in cut\_set(a)} \alpha_v \cdot h_v$. Thus, Y is a k-bound set for $f(X)$. □

As with the slicing method shown in the previous section, the conditions stated in Theorem 2 can be checked very efficiently on an MDD representation of the function. Figure 4.4 illustrates this method. In this example $\{b, c, d\}$ is a 3-bound set for

$$F(a, b, c, d, e) = a(b(\bar{d} + e) + \bar{b}ce) + \bar{a}bde + \bar{b}\bar{c}(a + e).$$

The gray lines in Figure 4.4(a) show the "slice" delimited by $cut(1)$ and $cut(4)$. The sub-functions g and h can be easily obtained from the MDD, as shown in Figure 4.4(b). The pseudo-code for a SLICE algorithm is shown

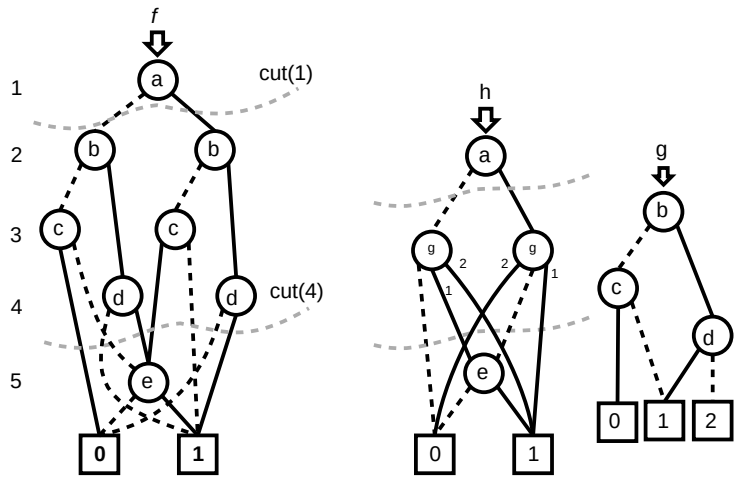


Figure 4.4: Disjoint-Support Slicing.

on *Paper B*, Figure B.3. .

Note, however, that in contrast with Theorem 1, the conditions formulated in Theorem 2 are sufficient, but not necessary, for a set of variables to be a k-bound set. The isomorphism condition in this generalization is too strong, and good decomposition candidates may be lost depending on the particular variable ordering [2]. There is a solution to this problem, and we present it in the next section.

Disjoint-Support Slicing Revisited Although the slicing method described in the previous section is useful in practice, it may overlook certain decompositions that are desirable. For example, the function

$$f(a,b,c,d,e) = a(b+c+d+e) + \bar{a}(bcde)$$

has a disjoint-support decomposition

$$f(a,b,c,d,e) = h(a,g(b,c,d),e)$$

that will be found by the slicing algorithm if the variable order is $\langle b,c,d,e,a\rangle$, but will not be found if the variable order is $\langle a,b,c,d,e\rangle$. Figure 4.5 illus-

[2]Note, however, that the slicing method detects classical bound sets in any position. The problem arises when looking for k-bound sets, with $k > 2$.

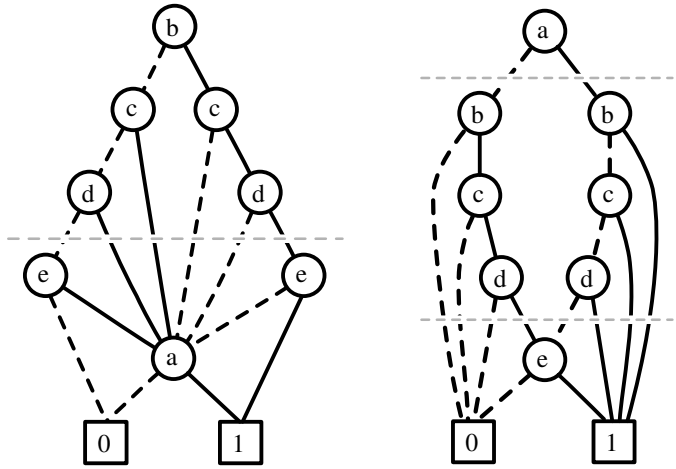


Figure 4.5: BDDs for function $a(b + c + d + e) + \bar{a}bcde$ for two different variable orderings.

trates this example. It shows the BDDs representing function f for two different variable orderings. Figure 4.5(a) shows the BDD for the ordering $\langle b, c, d, e, a\rangle$. The gray dashed line shows that the slicing method (in this case reduced to a simple cut) detects the 3-bound set $\{b, c, d\}$. When the set $\{b, c, d\}$ is in the middle, as shown in Figure 4.5(b), the slice method does not recognize it as a 3-bound set. This is due to the requirement of Theorem 2 that the two sub-graphs induced by the slice have to be pairwise isomorphic. In the example, the sub-graphs determined by the slice in Figure 4.5(b), between the gray dashed lines, are not isomorphic.

Lets formulate necessary and sufficient conditions for the existence of a k-bound set. Let $f : \mathbb{B}^{|Y \cup Z|} \to \mathbb{B}$, $Y \cap Z = \emptyset$, and $\mathbb{M} = \{0, \ldots, k-1\}$ for some $k > 1$.

Theorem 3. *A set of variables Y is a k-bound set for $f(Y, Z)$ if, and only if, there is a function $g : \mathbb{B}^{|Y|} \to \mathbb{M}$ such that*

1. *for all $\hat{z} \in \mathbb{B}^{|Z|}$, $f(Y, \hat{z})$ is a projection of g, and*
2. *for all $\hat{y}_1$, $\hat{y}_2 \in Y$ we have*

$$g(\hat{y}_1) = g(\hat{y}_2) \Leftrightarrow f(\hat{y}_1, z) = f(\hat{y}_2, z).$$

Proof.

⇒) By the definition of k-bound set, there are functions $g : \mathbb{B}^{|Y|} \to \mathbb{M}$ and $h : \mathbb{M} \times \mathbb{B}^{|Z|} \to \mathbb{B}$ such that $f(Y,Z) = h(g(Y),Z)$. Then, for every $\hat{z} \in \mathbb{B}^{|Z|}$, $f(Y,\hat{z}) = h(g(Y),\hat{z})$. The function $h(g,\hat{z})$ is a surjective mapping from $\mathbb{M}$ into $\mathbb{B}$. Therefore, by the definition of projection, $f(Y,\hat{z})$ is a projection of g. The second condition follows from the surjectivity of g and the minimality of the set $\mathbb{M}$.

⇐) We can write

$$f(Y,Z) = \sum_{\hat{z} \in \mathbb{B}^{|Z|}} \alpha_{\hat{z}}(Z) f(Y,\hat{z}) \tag{4.6}$$

where

$$\alpha_{\hat{z}}(Z) = \begin{cases} 1 & \text{if } Z = \hat{z} \\ 0 & \text{otherwise} \end{cases}$$

By hypothesis, for all $\hat{z} \in \mathbb{B}^{|Z|}$, $f(Y,\hat{z})$ is a projection of g, i.e. there are functions $\sigma_{\hat{z}} : \mathbb{M} \to \mathbb{B}$ such that $\sigma_{\hat{z}}(g(Y)) = f(Y,\hat{z})$. So, replacing in (4.6),

$$f(Y,Z) = \sum_{\hat{z} \in \mathbb{B}^{|Z|}} \alpha_{\hat{z}}(Z) \sigma_{\hat{z}}(g(Y)).$$

If we make

$$h(g,Z) = \sum_{\hat{z} \in \mathbb{B}^{|Z|}} \alpha_{\hat{z}}(Z) \sigma_{\hat{z}}(g)$$

then,

$$f(Y,Z) = h(g(Y),Z).$$

The second condition guarantees that the size of $\mathbb{M}$ is minimal. Therefore, by definition of k-bound set, Y is a k-bound set for f.

□

A straightforward corollary to Theorem 3 states our result in a manner similar to that of Theorems 1 and 2:

Corollary 1. *A set of variables Y is a* k-bound set *for $f(X)$ if, and only if:*

1. *for all $v \in cut\_set(a)$, Y is a k_v-bound set for the co-factor $f|_{\alpha_v}(Y,Z_2)$ in* (4.2) *with $k_v \leq k$, and*

2. *there exists a function* $g : \mathbb{B}^{|Y|} \rightarrow \mathbb{M}$ *with the smallest set* $\mathbb{M}$*, such that each sub-function* $g_v(Y)$ *induced by the decomposition* (4.4) *is a projection of* g*.*

This corollary shows that there is a straightforward algorithm to compute a disjoint-support decomposition form a BDD representation of a function, if we can compute the MDD for the sub-function $g(Y)$ from the MDDs of its projections $g_v(Y)$. Such computation is possible by means of the following technique.

In order to simplify the exposition of the algorithm, we will assume that all our MDDs will have Boolean variables, but an arbitrary number of constant nodes. These MDDs are usually referred to as *multi-terminal* BDDs. The algorithm KERNEL allows us to compute a function $g : \mathbb{B}^{|X|} \rightarrow \mathbb{M}$ given $g_1 : \mathbb{B}^{|X|} \rightarrow \mathbb{M}_1$ and $g_2 : \mathbb{B}^{|X|} \rightarrow \mathbb{M}_2$, based on their respective representations as MDDs. Figure 4.6 shows a recursive implementation of the KERNEL algorithm in pseudo-code, using an MDD data structure as implemented in the Colorado University Decision Diagram package (CUDD [144])[3].

The procedures CONST?, MK-CONST, ELSE, THEN, ITE, and TOPVAR are provided with the CUDD package, and implement the following functions:

- CONST?($\mathcal{G}$) checks whether the BDD $\mathcal{G}$ is a constant or not;
- MK-CONST(k) returns a BDD representing a constant function of value k.
- ELSE($\mathcal{G}$) returns the *else* child of $\mathcal{G}$ (see Section 2.3);
- THEN($\mathcal{G}$) returns the *then* child of $\mathcal{G}$ (see Section 2.3);
- ITE(v,$\mathcal{G}$,$\mathcal{H}$) returns a BDD representing the function $vf_{\mathcal{G}} + \bar{v}f_{\mathcal{H}}$;
- TOPVAR($\mathcal{G}$) returns the top variable of BDD $\mathcal{G}$.

Figures 4.7 and 4.8 give an illustration of this procedure, based on the example at the beginning of this section (Figure 4.5(b)). Figure 4.7 shows an example of the computation of sub-function g and mappings σ_1 and σ_2, such that $g \circ \sigma_1 = g_1$ and $g \circ \sigma_2 = g_2$, for the functions given in tabular form. Figure 4.8 shows an example of the application of algorithm KERNEL to obtain the MDD for function g from the MDDs of g_1 and g_2.

[3]Strictly speaking, the CUDD package calls these multi-terminal BDDs *Algebraic Decision Diagrams* or ADDs.

```
algorithm KERNEL(G, G')
    if CONST?(G) and CONST?(G')
        return MK-CONST(GET-NUMBER(G,G'))
    if CONST?(G)
        top_var = TOPVAR(G')
        c0 = KERNEL(G, ELSE(G'))
        c1 = KERNEL(G, THEN(G'))
    else if const?(G')
        top_var = TOPVAR(G)
        c0 = KERNEL(G', ELSE(G))
        c1 = KERNEL(G', THEN(G))
    else
        if TOPVAR(G) < TOPVAR(G')
            top_var = TOPVAR(G)
            c0 = KERNEL(G', ELSE(G))
            c1 = KERNEL(G', THEN(G))
        else if TOPVAR(G) > TOPVAR(G')
            top_var = TOPVAR(G')
            c0 = KERNEL(G, ELSE(G'))
            c1 = KERNEL(G, THEN(G'))
        else
            top_var = TOPVAR(G)
            c0 = KERNEL(ELSE(G'), ELSE(G))
            c1 = KERNEL(THEN(G'), THEN(G))
    return ITE(top_var, c1, c0)
end
```

Figure 4.6: Pseudo code of the `Kernel` algorithm.

	g_1	g_2	g
000	0	0	0
001	0	1	1
010	0	1	1
011	0	1	1
100	0	1	1
101	0	1	1
110	0	1	1
111	1	1	2

three different pairs:

$00 \mapsto 0$

$01 \mapsto 1$

$11 \mapsto 2$

$\nearrow$

$\searrow$

σ_1		
0	$\mapsto$	0
1	$\mapsto$	0
2	$\mapsto$	1

σ_2		
0	$\mapsto$	0
1	$\mapsto$	1
2	$\mapsto$	1

Figure 4.7: Calculating the sub-function g and mappings σ_1 and σ_2.

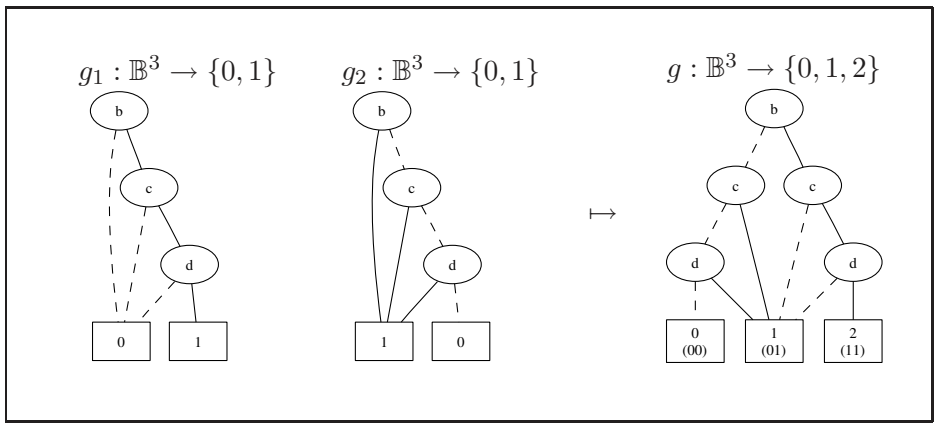


Figure 4.8: Calculating the MDD for function g from the MDDs of g_1 and g_2.

4.2 Hybrid Disjoint-Support Decomposition

In previous sections we have shown that it is possible to devise BDD-based heuristics which quickly find many disjoint-support decompositions, and which can handle large functions. However, one problem with such techniques is that the decompositions that can be obtained do not necessarily simplify the function. For example, a circuit implemented as the two cofactors of a Shannon decomposition joined by a multiplexer is usually not optimal.[4]

Another problem is that, in contrast to simple disjoint decompositions, that are "too few", disjoint-support decompositions are "too many". So, an algorithm which first generates all decompositions and then decides which of them simplify the function is not feasible for large functions.

In *Paper C*, on page 99 we present our approach to overcome these problems. First, a set of *proper cut* points is identified in a circuit representation of the function by applying a structural decomposition method. Then, the circuit is partitioned along these cut points into a set of smaller sub-circuits, which are treated independently. This procedure allows us to reduce the search space for disjoint-support decompositions at the next stage, in which we apply a BDD-based technique similar to the ones presented earlier. Finally, the overall decomposition is determined by combining the intermediate results.

Preliminaries

Let $C = (V, E)$ denote a directed acyclic graph representing a single-output circuit, where V represents a set of gates and primary inputs. A particular vertex $root \in V$ is marked as the circuit output. The set of edges $E \subseteq V \times V$ describes the nets connecting the gates. We will call this type of graph a *circuit graph*.

The *cone of influence* of a vertex v, is a subset of V containing all the vertices from which v is reachable.

A vertex v *dominates* another vertex w in V if every path from w to $root$ contains v [103]. We call v a *dominator* of w. Vertex v is the *immediate dominator* of w, denoted by $v = idom(w)$, if v dominates w and every other

[4] Shannon decomposition is a special case of decomposition, where $f(x, Y) = \bar{x} \cdot g_0(Y) + x \cdot g_1(Y)$, where $g_0(Y) = f(0, Y)$, $g_1(Y) = f(1, Y)$ and $x \notin Y$.

dominator of w dominates v. Every vertex $v \in V$ except *root* has a unique immediate dominator [108]. The edges $\{(idom(w), w) \mid w \in V - \{root\}\}$ form a directed tree D rooted at *root*, which is called the *dominator tree* of C. A *reduced* dominator tree [95] D_R contains all vertices $v \subseteq D$ such that:

1. v is a primary input or
2. $\exists u \in D_R$ such that $v = idom(u)$.

A vertex is called a *proper cut* if it dominates all primary input vertices in its cone of influence.

Circuit-Based Decomposition

The concept of *proper cuts* was first introduced in combinational equivalence checking [56]. It was later applied to testing [137, 17] and design for low power [43] where it is known under the alternative names of *headlines* or *supergates*. The definition of proper cut states that every path from any primary input in the cone of influence of a proper cut v to the root contains v. This guarantees that all re-converging paths in the circuit are completely enclosed within the cone and, therefore, that those primary inputs belong to a bound set (see Figure 4.9). The primary input vertices and the root vertex are *trivial* proper cuts, i.e. they always exist.

We have chosen to use at the first stage of our algorithm a circuit-based technique, rather than a BDD-based one, because manipulating circuits is much faster. Therefore, for functions with no proper cuts, the presented technique does not bring a significant overhead. The running time of our algorithm is normally similar, or even faster, than the running time of a BDD-based algorithm.

The algorithm presented in *Paper C* for finding proper cuts is based on the concept of a reduced dominator tree constructed by using an extension of the Lengauer-Tarjan algorithm [103] for finding dominators in a graph.

It is straightforward to prove that a proper cut is always a vertex of the reduced dominator tree D_R.

Lemma 1. *A vertex $v \in V$ is a proper cut only if $v \in D_R$.*

Figure 4.9 gives an example. The circuit shown represents $((d \oplus e)(ab + \bar{a}c)) + (\overline{ab + \bar{a}c})f$. Notice the two proper cut points, and their respective cones of influence in gray. They correspond to bound sets $\{d, e\}$ and $\{a, b, c\}$ respectively.

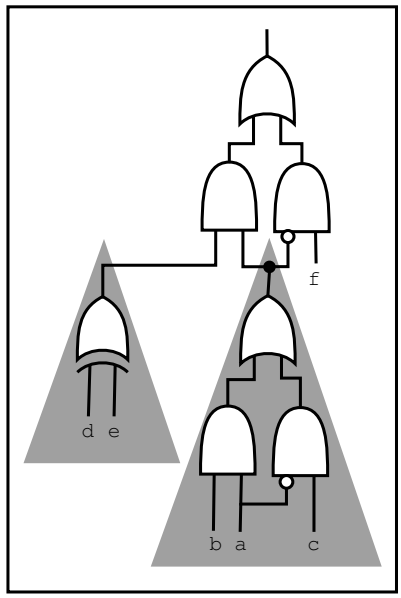


Figure 4.9: Proper cut points.

The pseudo-code of the algorithm ProperCut which uses a reduced dominator tree D_R to identify the set of proper cuts P [95] is shown in Figure 4.10.

ProperCut processes the circuit from the inputs toward the output in topological order. The array $T[v]$ contains vertices $u \in D_R$ with open re-convergences. At the primary inputs, $T[v]$ is initialized to an empty set. Then, at each following vertex v, $T[v]$ is updated to the union of $T[v_i]$ for all vertices v_i in its fan-in. If v is in the reduced dominator tree, then the set $Doms(v)$ of vertices having v as an immediate dominator is removed from $T[v]$ and, after performing the proper cut checking, v is added to $T[v]$. This substitution of $Doms(v)$ vertices by their dominator allows us to keep the size of $T[v]$ small and, what is more important, let's us keep the support-set of $T[v]$ dependent on vertices having v as an immediate dominator only, rather than vertices on previous topological levels.

```
algorithm ProperCut(V, E, root);
  D_R, Doms = Dominator(V, E, root)
  for each v ∈ V in topological order do
      if v ∈ Inputs then
          T[v] = ∅;
      else
          T[v] = ⋃_{v_i ∈ FI(v)} T[v_i];
      if v ∈ D_R then
          T[v] = T[v] − Doms(v);
          if T[v] = ∅ then
              P = P ∪ {v};
          T[v] = T[v] ∪ {v};
  return P
end
```

Figure 4.10: Pseudo-code of the algorithm ProperCut.

BDD-based Decomposition

After the set of proper cuts is identified, the circuit is partitioned along these cut points into a set of smaller sub-circuits which are processed independently using the BDD-based decomposition technique presented in section 4.1. The algorithm successively goes through all possible linear intervals of variables of a BDD and, for each interval, checks whether it is a bound set or not. In this way many decompositions are found very quickly, without expensive variable reordering.

The integration of circuit-based and BDD-based techniques results in an algorithm which is more robust than the pure BDD-based method regarding both quality of the result and running time. Our experiments on benchmark circuits suggest that the resulting algorithm has a significant potential for a large number of circuits. For details on these results, see *Paper C*, included on page 99 in this thesis.

4.3 Circuit Based Non-Disjoint Decomposition

We have shown in *Paper C* that we can extract a lot of information about decompositions from the structure of a circuit graph. Following a similar route, *Paper D*, on page 113, presents a result that relates a different type of decomposition, the *non-disjoint support* decomposition, to certain structural properties of a circuit graph.

Multiple-Vertex Dominators

Recall the definition of a *circuit graph* and *single vertex dominator* from Section 4.2. Many graphs do not contain any single-vertex dominators except for the primary inputs and *root*. It is more common that a vertex is dominated by a set of vertices.

A set of vertices $\{v_1, \ldots, v_k\}$ is a *multiple-vertex dominator* of size k [5] (also called *generalized* dominator [77]) for a vertex u, if (1) every path from u to *root* contains some v_i, and (2) for every v_i, there exist at least one path from u to *root* which contains v_i and does not contain any other v_j, $i, j \in \{1, \ldots, k\}$, $i \neq j$. A set of vertices $\{v_1, \ldots, v_k\}$ is a *common multiple-vertex dominator* for a set of vertices $U \subseteq V - \{v_1, \ldots, v_k\}$, if, for every $u \in U$, there exist $W \subseteq \{v_1, \ldots, v_k\}$ such that W is a multiple-vertex dominator for u.

Theorem 4. *Suppose a circuit graph $C = (V, E)$ represents a Boolean function $f(X, Y, Z)$, where X, Y, Z are sets of variables partitioning the support set of f. Let $V_X, V_Y, V_Z \subset V$ be sets of primary input vertices corresponding to the variables of the sets X, Y, Z. Let $v_{g_1}, \ldots, v_{g_k} \in V$ be a set of vertices such that:*

1. *$\{v_{g_1}, \ldots, v_{g_k}\}$ is a common multiple-vertex dominator for V_X,*
2. *$(V_X \cup V_Y) \subset \bigcup_{i=1}^{k} I(v_{g_i})$, where $I(v_{g_i})$ is the cone of influence of v_{g_i}.*

Then, there exist a decomposition of f of type

$$f(X, Y, Z) = h(g_1(X, Y), \ldots, g_k(X, Y), Y, Z) \tag{4.7}$$

where Boolean functions g_i are the functions rooted by the vertices v_{g_i}, $\forall i \in \{1, \ldots, k\}$, of C.

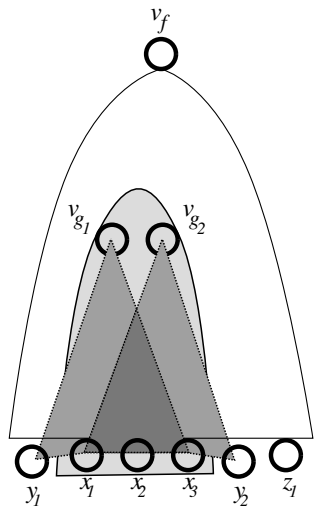


Figure 4.11: Nodes $\{v_{g_1}, v_{g_2}\}$ are a common multiple vertex dominator for the set of inputs $\{x_1, x_2, x_3\}$

Figures 4.11 and 4.12 illustrate this theorem. Figure 4.11 shows an abstracted circuit graph representing a Boolean function $f(X, Y, Z)$, where $X = \{x_1, x_2, x_3\}$, $Y = \{y_1, y_2\}$, $Z = \{z_1\}$. The output (*root*) node is marked v_f. The nodes $\{v_{g_1}, v_{g_2}\}$ are a *common multiple vertex dominator* for the set of inputs X; the domination relation is depicted by the light gray bell shaped area. The cones of influence of each of these nodes (reaching $X \cup \{y_1\}$ and $X \cup \{y_2\}$ respectively) are shown in dark gray. The theorem shows that, under these conditions, function f can be decomposed as $h(g_1(X, y_1), g_2(X, y2), Y, Z)$, where $h : \mathbb{B}^{|Y \cup Z|+2} \rightarrow \mathbb{B}$, $g_1 : \mathbb{B}^{|X \cup \{y_1\}|} \rightarrow \mathbb{B}$ and $g_2 : \mathbb{B}^{|X \cup \{y_2\}|} \rightarrow \mathbb{B}$. The resulting decomposition is illustrated in Figure 4.12.

We present a proof of Theorem 4 which is not included in *Paper D*.

Proof. Let $M = \{v_{g_1}, \ldots, v_{g_k}\}$ be the common multiple-vertex dominator. Let G be the set of all nodes from which M is reachable (the gray areas in Figure 4.11), $G = \bigcup_{i=1}^{k} I(v_{g_i})$, and let H denote all the nodes in the circuit graph C that do not belong to G, i.e. $H = V - G$ (the white area on Figure 4.11). Transform the graph C following the next three steps:

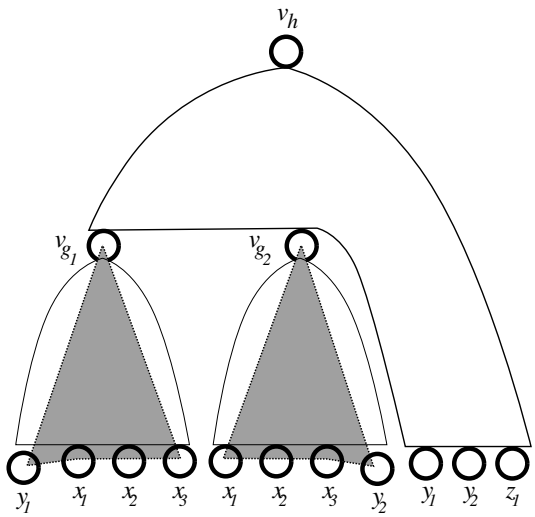


Figure 4.12: Non-disjoint support decomposition of the function represented in Figure 4.11

1. Pick an edge of the graph, $(v, u) \in E$, such that $u \in H$, $v \in G$, and $v \notin M$.

2. Make an isomorphic copy of the sub-graph induced by $I(v)$, and call w the root of this copy.

3. Remove edge (v, u) from the graph, and add an edge (w, u)

After these steps, the transformed graph still represents f, since isomorphic sub-graphs always represent the same Boolean function. Moreover, the sub-graph created in step 2 does not contain any primary input vertices in V_X. If this was not the case, it would contradict the fact that M is a common multiple-vertex dominator for V_X, since it would imply there is a path from a node in V_X to the root which does not contain any vertex in M.

Lets repeat the steps 1–3 above until no more edges can be picked. In the end, we will have a circuit graph C' that still represents function f, but in which every path from a node v in G to the root contains a node in M. Lets call v_h the root of C'. The graph is now similar to the one shown in Figure 4.12. We can "split" the graph C' by creating k new primary input vertices g_1,g_2,...,g_k, and replacing every edge (v_{g_i}, u), $1 \leq i \leq k$, with an

edge (g_i, u). The resulting graph has $k+1$ root nodes $\{v_h, v_{g_1}, \ldots, v_{g_k}\}$, representing functions $\{h, g_1, \ldots, g_k\}$ in equation (4.7). □

Theorem 4 allows us to reduce the problem of computing non-disjoint decompositions to the problem of computing multiple-vertex dominators. In the next section, we show that the problem of computing all multiple-vertex dominators of a fixed size can be solved in polynomial time.

Computing Multiple-Vertex Dominators

It is possible to compute all single-vertex dominators for a directed graph in time less than quadratic in the number of vertices. For example, the well-known Lengauer-Tarjan algorithm [103] has the worst-case complexity $O(n \cdot \log n)$. However, algorithms for computing all multiple-vertex dominators for a directed graph have exponential worst case complexity [77]. A subset of immediate multiple-vertex dominators can be computed in $O(n^2)$ time [5], but immediate dominators are not particularly interesting from the decomposition point of view. Good decompositions require multiple-vertex dominators of a small size k which are common for large sets V_X. The following theorem shows that it is possible to compute multiple-vertex dominators of a fixed size in polynomial time.

Theorem 5. *If there exists an $O(\tau(n))$ algorithm for computing all single-vertex dominators, then there exists an $O(n^{k-1}\tau(n))$ algorithm for computing all multiple-vertex dominators of size k.*

Proof. See *Paper D*, included in this thesis on page 113. □

If the Lengauer-Tarjan algorithm [103] is used for computing single-vertex dominators, then the set of multiple-vertex dominators can be obtained in $O(n^k \log n)$ time. Clearly, the simple algorithm constructed in the proof will not be feasible for large circuit graphs if $k > 2$. However, for small k, even this straightforward approach gives good results. Many practical applications of decomposition require only small values of k (e.g. multi-level logic synthesis [136, 161], or FPGA technology mapping [132, 41]).

Our experiments support the claim that the problem of computing non-disjoint decompositions of Boolean functions can be solved efficiently using multiple-vertex dominators. They also show that the technique can decompose functions for which BDDs cannot be build, such as the 16-bit multiplier

*C*6288 from the IWLS'02 benchmark set. The details of the experiments can also be found in *Paper D*, included in this thesis on page 113.

4.4 Efficient Circuit Re-Synthesis

Paper E, on page 123, presents a technique to transform the original circuit implementing $f(X, Y)$ into a circuit implementing the decomposed representation $h(g(X), Y)$. Previous algorithms [161, 99, 111] computed circuits for the decomposed representation from BDDs of g and h, by applying various BDD-to-circuit transformation techniques. The algorithm presented in *Paper E* uses BDDs only for an *analysis* of the decomposition. The actual *synthesis* of the circuits for g and h is done by restricting the original circuit with respect to a given assignment of input variables. This guarantees that the sizes of the circuits of g and h are strictly smaller than the size of the original circuit.

In the sequel, let X be a bound set for f and let $\mathcal{G}_g$ and $\mathcal{G}_h$ be BDDs representing the functions g and h in the decomposition $f(X, Y) = h(g(X), Y)$. These BDDs are computed by the slicing method introduced in Section 4.1.

Constructing the circuit for h

Suppose $\hat{x}$ is an assignment of variables of X leading to the 0-terminal node in $\mathcal{G}_g$. Then $g(\hat{x}) = 0$, and thus $f(\hat{x}, Y) = h(g(\hat{x}), Y) = h(0, Y)$. Therefore, a circuit implementing the co-factor $h_0 = h(0, Y)$ can be obtained from the circuit implementing f by applying the assignment $\hat{x}$ to the inputs X and propagating the constants through the circuit using the usual reduction rules:

- If an OR (AND) gate has one of its inputs assigned to 1 (0), it is replaced by constant 1 (0);
- If an OR (AND) gate has one of its inputs assigned to 0 (1), this input is removed.

Similarly, the circuits implementing co-factors $h_i(Y)$, $i \in \{1, 2, \ldots, k-1\}$, can be obtained by propagating an assignment of variables of X leading to i-terminal node of $\mathcal{G}_g$. Recall that g is a function of type $g : \{0, 1\}^{|X|} \rightarrow \{0, 1, ..., k-1\}$, so $\mathcal{G}_g$ is a multi-terminal BDD with k terminal nodes.

In general, different assignments $\hat{x}$ result in different circuits for $h_i(Y)$. To maximize the sharing of common logic of the i circuits implementing co-factors $h_i(Y)$, $i \in \{0, 1, \ldots, k-1\}$, i assignments $\hat{x}_i$ are chosen so that they differ in the fewest number of bit positions.

The decomposition $h(g(X), Y)$ is obtained by combining the co-factors in a Shannon expansion as follows:

$$h(g(X), Y) = \sum_{i=0}^{k-1} g_1^{i_1}(X) g_2^{i_2}(X) \dots g_r^{i_r}(X) h_i(Y) \tag{4.8}$$

where $(i_1, i_2, \dots, i_r)$ is the binary expansion of i, $r = \lceil \log_2 k \rceil$, and the term $g_j^{i_j}$ is defined by

$$g_j^{i_j} = \begin{cases} g_j & \text{if } i_j = 1 \\ \overline{g}_j & \text{otherwise} \end{cases}$$

for $j \in \{1, 2, \dots, r\}$.

The circuit implementing expression (4.8) is constructed by feeding the co-factors $h_i(Y)$ into a multiplexer with k control inputs $g_1(X)$, $g_2(X), \dots, g_r(X)$.

As an example, consider the BDD shown in Figure 4.13. This BDD represents the function

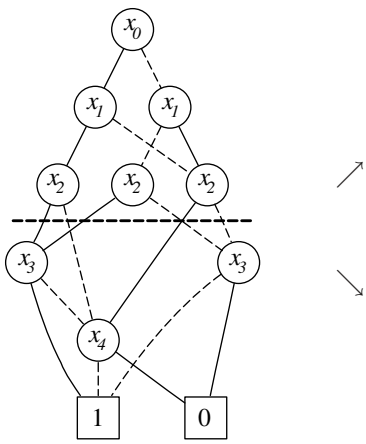
$$f = (x_0' + x_1')(x_2' x_3') + x_2(x_3(x_0' \oplus x_1) + x_4') + x_0 x_1 x_4'.$$

The cut line shows that $\{x_0, x_1, x_2\}$ is a bound set for f. Let's see how to structurally decompose function f with the information provided by its BDD representation. We have to find assignments of $\{x_0, x_1, x_2\}$ such that the path represented by each of the assignments reaches each of the sub-functions $\{h_0, h_1, h_2\}$. In this case, when $x_0 = 1$, $x_1 = 1$, and $x_2 = 1$ we reach sub-function h_1. This means that by making such assignment and propagating the constants, we will obtain function h_1:

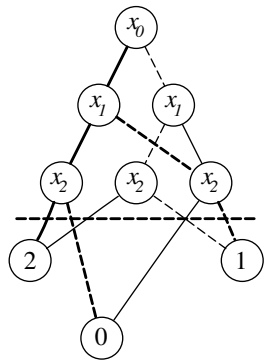
$$\begin{aligned} h_1 &= f[x_0 \leftarrow 1, x_1 \leftarrow 1, x_2 \leftarrow 1] \\ &= (1' + 1')(1' x_3') + 1(x_3(1' \oplus 1) + x_4') + 1\,1\,x_4' \\ &= x_3 + x_4'. \end{aligned}$$

Similarly, when $x_0 = 1$, $x_1 = 1$, and $x_2 = 0$ we reach sub-function h_0:

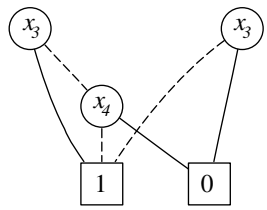
$$\begin{aligned} h_0 &= f[x_0 \leftarrow 1, x_1 \leftarrow 1, x_2 \leftarrow 0] \\ &= (1' + 1')(0' x_3') + 0(x_3(1' \oplus 1) + x_4') + 1\,1\,x_4' \\ &= x_4'. \end{aligned}$$



(a) BDD representation for f



(b) Function g



(c) Function h

Figure 4.13: Binary decision diagrams representing the function $f = (x'_0 + x'_1)(x'_2x'_3) + x_2(x_3(x'_0 \oplus x_1) + x'_4) + x_0x_1x'_4$ and an example decomposition. The bound set is $\{x_1, x_2, x_3\}$, and the free set $\{x_3, x_4\}$.

When $x_0 = 1$, $x_1 = 0$, and $x_2 = 0$ we reach sub-function h_2:

$$\begin{aligned} h_2 &= f[x_0 \leftarrow 1, x_1 \leftarrow 0, x_2 \leftarrow 0] \\ &= (1' + 0')(0' x_3') + 0(x_3(1' \oplus 0) + x_4') + 1\,0\,x_4' \\ &= x_3'. \end{aligned}$$

Finally, we construct h. Thus,

$$\begin{aligned} h(g_0, g_1, x_3, x_4) &= g_0' g_1' h_0 + g_0' g_1 h_1 + g_0 g_1' h_2 + g_0 g_1 h_2 \\ &= g_0'(g_1' x_4' + g_1(x_3 + x_4')) + g_0 x_3' \\ &= g_0'(g_1 x_3 + x_4') + g_0 x_3'. \end{aligned}$$

In the next section, we will consider the problem of constructing the circuits for the functions $g_1(X), g_2(X), \ldots, g_r(X)$ encoding the k-valued function $g(X)$.

Constructing the circuit for g

Suppose that $\hat{y}$ is an assignment of variables of Y such that $h_i(\hat{y}) \neq h_j(\hat{y})$ for some $i, j \in \{0, 1, \ldots, k-1\}$, $i \neq j$. Then $f(X, \hat{y}) = h(g(X), \hat{y})$ where the co-factor $h(g(X), \hat{y})$ is neither constant 0, nor constant 1, i.e. it depends on $g(X)$.

Since h is a function of type $\{0, 1, ..., k-1\} \times \{0,1\}^{|Y|} \rightarrow \{0,1\}$, the co-factor $h(g(X), \hat{y})$ is a function of type $\{0, 1, ..., k-1\} \rightarrow \{0,1\}$. Note that, for $k = 2$, $h(g(X), \hat{y})$ is either an identity, or a complement. Since, for a given bound set X, the function $g(X)$ is unique up to complementation [7], both $g(X)$ and $\overline{g}(X)$ can be used for the decomposition. Thus, at this step, the problem of constructing the circuit for $g(X)$ is solved for $k = 2$. For larger values of k, the following strategy is used.

The k-valued function $g(X)$ can be expressed as

$$g(X) = \sum_{i=0}^{k-1} i \cdot g^i(X)$$

where $g^i : \{0, 1, \ldots, k-1\}^{|X|} \rightarrow \{0,1\}$ are multiple-valued *literals* defined as:

$$g^i(X) = \begin{cases} 1 & \text{if } g(X) = i \\ 0 & \text{otherwise} \end{cases}$$

For a given encoding of the k possible values of $g(X)$, each of the functions $g_1(X)$, $g_2(X), \ldots, g_r(X)$, $r = \lceil \log_2 k \rceil$, encoding $g(X)$, can be represented as a sum of some literals $g^i(X)$'s. For example, if $k = 4$ and the encoding is $0 = (00), 1 = (01), 2 = (10), 3 = (11)$, then $g_1(X) = g^2(X) + g^3(X)$ and $g_1(X) = g^1(X) + g^3(X)$.

Consider a decomposition chart of $h(g(X), Y)$ with columns representing k values of $g(X)$ and the rows representing all combinations of the variables of Y. Any non-constant row of $h(g(X), Y)$ represents a sum of some literals $g^i(X)$, $i \in \{0, 1, \ldots, k-1\}$.

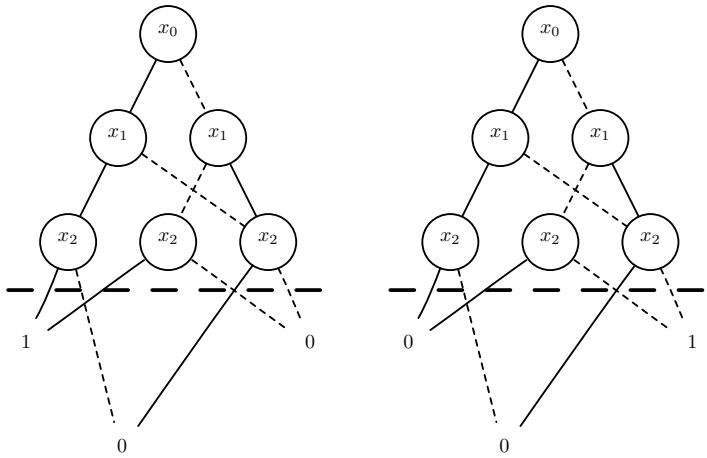
In the best case, there exist rows in the decomposition chart corresponding directly to the encoded functions $g_1(X)$, $g_2(X)$, ..., $g_r(X)$. If $h(g(X), \hat{y})$ $= g_j(X)$ for some assignment $\hat{y}$ of the variables of Y, then the circuit implementing $g_j(X)$ can be obtained from the circuit implementing f by applying the assignment $\hat{y}$ to the inputs Y and propagating the constants.

In the worst case, the literals $g^i(X)$, $i \in \{0, 1, \ldots, k-1\}$, need to be computed by ANDing selected rows of $h(g(X), Y)$. Afterwards, the functions $g_1(X)$, $g_2(X)$, ..., $g_r(X)$ are obtained as a combination of $g^i(X)$.

Let us return to the example started on page 53. In order to obtain function h, according to the explanation above, we produce a table that shows the values taken by the sub-functions h_0, h_1, and h_2 for different values of the variables in the free set $\{x_3, x_4\}$.

x_3	x_4	**0**	**1**	**2**	
0	**0**	1	1	1	
0	**1**	0	0	1	$\leftarrow g_0$
1	**1**	0	1	0	$\leftarrow g_1$
1	**0**	1	1	0	

The row marked g_0 allows us to discriminate the paths that go to h_0 or to $\{h_1, h_2\}$. Similarly, the row marked g_1 allows us to discriminate the paths that go to h_2 or to $\{h_0, h_1\}$. This is enough to discriminate the paths going to each of the three h_i functions: when $g_0 = 1$ we select h_0, when $g_0 = 0$ and $g_1 = 0$ we select h_1, when $g_0 = 0$ and $g_1 = 1$ we select h_2 (see the paths marked with thick lines in fig. 4.13(b)). In order to obtain the function g_0, we make $x_3 = 0$, $x_4 = 1$ and propagate the constants (Fig. 4.14(a))



(a) g_0 (b) g_1

Figure 4.14: Binary encoding of function g.

$$\begin{aligned} g_0 &= f[x_3 \leftarrow 0, x_4 \leftarrow 1] \\ &= (x_0' + x_1')(x_2' 0') + x_2(0(x_0' \oplus x_1) + 1') + x_0 x_1 1' \\ &= (x_0' + x_1')x_2'. \end{aligned}$$

In a similar way, we obtain g_1 by making $x_3 = 1$, $x_4 = 1$ and by propagating the constants (Fig. 4.14(b))

$$\begin{aligned} g_1 &= f[x_3 \leftarrow 1, x_4 \leftarrow 1] \\ &= (x_0' + x_1')(x_2' 1') + x_2(1(x_0' \oplus x_1) + 1') + x_0 x_1 1' \\ &= (x_0' \oplus x_1)x_2. \end{aligned}$$

To summarize, the re-synthesis technique described in this section works by structurally partitioning the original circuit representation according to the information provided by the partitioned BDD blocks. After all the blocks have been recovered, the BDDs are not needed and can be discarded. The resulting circuit is proportional to the original circuit representation, and not to the intermediate BDD representation. This is an advantage because BDDs can grow exponentially in some cases, and therefore decomposition algorithms which synthesize the circuit directly from BDDs may cause an exponential increase in the circuit's size. To cope with this space explosion,

each block of the partitioned circuit has to be re-synthesized before further processing. The extra re-synthesis, on the other hand, may impose a prohibitive time/space penalty on the design flow. The presented approach is free from these problems.

4.5 On the Relation of Bound Sets and Best Orderings

The result we present in *Paper F*, on page 131, is of theoretical interest, and answers a long standing question regarding the relation between the bound sets of a Boolean function and the best variable orderings for its corresponding BDD representation.

BDDs have proved to be an efficient data structures for representation and manipulation of Boolean functions for logic synthesis, testing and verification. Although a function may require, in the worst case, a BBD of size exponential in the number of variables, many practical functions have a representation which is linear in the number of variables [30].

As we mentioned in Section 2.3, a major concern with BDDs is that the size of the graph varies for different variable orderings and, for some functions, it is highly sensitive to the ordering. For example, BDDs representing adders have exponential number of nodes in the worst case and linear number of nodes in the best case. Hence, care must be taken to select a suitable ordering for the variables, minimizing the size of the graph.

The problem of computing a best variable ordering is known to be co-NP-complete [72], and therefore heuristic algorithms are used in practice. Many ordering heuristics analyze the structure of the logic circuit, implementing the function under consideration, and use its underlying topology to determine a best ordering [30, 110]. However, if there is no circuits to refer to, finding a good order is more difficult. It happens, for example, when computing the set of reachable states of a finite state machine from an initial state. Many intermediate BDDs are generated and, if no suitable ordering is found, their size may grow too large and exceed the peak memory limit.

If there is no circuit to refer to, then some properties of the function must be used to guide the ordering of the variables. Several different strategies have been investigated in this respect. In [88], it has been observed that *symmetric* variables tend to be adjacent in the best orderings. A number of heuristics for finding best orderings utilizing this property have been developed, including [122] and [125]. However, a counterexample has been shown in [125] of a function for which no order with the symmetric variables adjacent is best.

In [117], it has been shown that minimizing *width* of a ROBDD often

leads to a reduction in the number of nodes. The width on a given level of a ROBDD is defined as a number of distinct nodes in the lower block that are adjacent to the boundary between the two blocks. Minimal-width strategy has been used in the heuristic for finding a best variable ordering from [117]. However, in [64] a counterexample has been shown, giving a function for which no minimal-width ordering is best.

In [87], it was suggested to keep adjacent the variables from the bound sets of the function which are explicitly given by its composition tree. Recall from Section 3.1 that the composition tree of a Boolean function is a structure reflecting all its non-overlapping bound sets.

We call an ordering preserving all bound sets from a composition tree *bound-set preserving*.

Let $\langle X \rangle$ denote a set of variable orderings induced by all possible permutations over the set X. Then, the main result of [87] is given by the following theorem:

Theorem 6 ([87]). *If $f : \mathbb{B}^n \to \mathbb{B}$ has a decomposition of type*

$$f(X) = g(h_1(Y_1), h_2(Y_2), \ldots, h_k(Y_k));$$

where $\{Y_i\}$, $1 \leq i \leq k$, is a partition of $X = \{x_1, \ldots, x_n\}$, the functions h_i are of type $h_i : \mathbb{B}^{|Y_i|} \to \mathbb{B}$, and function g is of type $g : \mathbb{B}^k \to \mathbb{B}$, then there exists a variable ordering belonging to the set $\langle\langle Y_1 \rangle, \langle Y_2 \rangle, \ldots, \langle Y_k \rangle\rangle$ which is best.

Our *Paper F* presents a counter-example to Theorem 6. The counter-example is constructed by showing a function f which has a decomposition of type

$$f = g(h_1(Y_1), h_2(Y_2), h_3(Y_3), h_4(Y_4), x_m);$$

where $\{Y_1, Y_2, Y_3, Y_4, x_m\}$ is a partition of X; and g is a function

$$g = h_3(h_4(h_2' + x_m') + h_1' x_m) + h_3'(h_4 x_m + h_1(h_2 \oplus x_m)),$$

where $h_i(Y_i) = \bigvee_{j \in Y_i} x_j$, $i \in \{1, 2, 4\}$, $h_3(Y_3) = (h_{31}(Y_{31}) \oplus x_k)'$, $h_{31}(Y_{31}) = \bigvee_{j \in Y_{31}} x_j$, $Y_{31} = Y_3 - \{x_k\}$, and "$\oplus$" stands for XOR. We also require that the BDDs for h_1, h_2, and h_4 are much larger than the BDD for h_3,

$$|\mathcal{G}_{h_1}| = |\mathcal{G}_{h_2}| = |\mathcal{G}_{h_4}| >> |\mathcal{G}_{h_3}|.$$

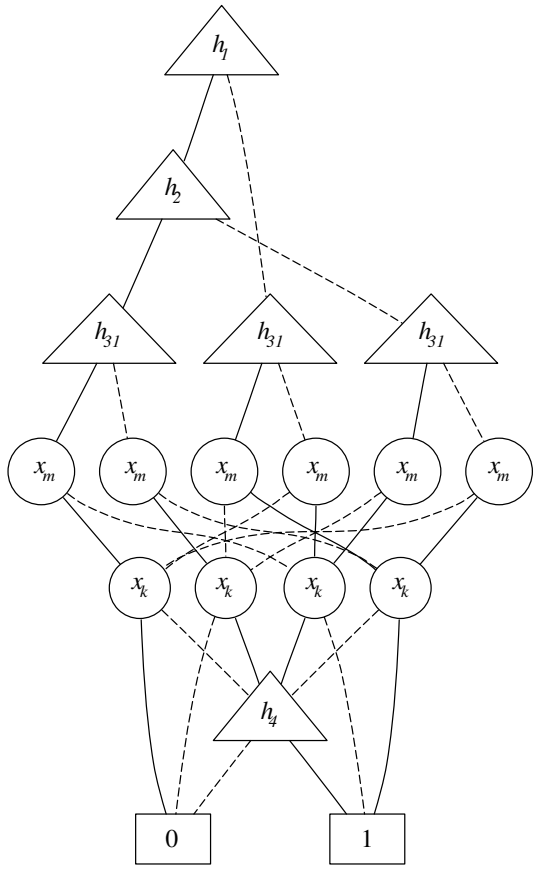


Figure 4.15: The structure of $\mathcal{G}_f$ for any of the best variable orderings.

Under these conditions, the set of all bound-set-preserving orderings of the BDD $\mathcal{G}_f$ is given by

$$\langle\langle Y_1\rangle, \langle Y_2\rangle, \langle\langle Y_{31}\rangle, x_k\rangle, \langle Y_4\rangle, x_m\rangle.$$

We show, however, that the set of best variable orderings for this function is

$$(\langle Y_1\rangle, \langle Y_2\rangle, \langle Y_{31}\rangle, x_m, x_k, \langle Y_4\rangle).$$

None of these orderings is bound-set-preserving, thus contradicting the claim in [87]. The structure of $\mathcal{G}_f$ for any of these best variable orderings

is illustrated in Figure 4.15. In this figure, the nodes shaped as a triangle represent sub-graphs that are omitted for clarity, and the circular nodes represent variables in the usual manner.

4.6 From Nature to Electronics: Kauffman Networks

Paper G, on page 139, is a look into the future. It presents an algorithm that improves the state-of-the-art in the analysis of *Random Boolean Networks* (*RBN*s). RBNs are used in a number of applications in biology and physics, including cell differentiation, immune response, evolution, gene regulatory networks and neural networks.

The paper also presents an idea on how this genetic system could be used as a new and general model of computation. The compositional properties of this model, that we have just started to uncover, will certainly challenge our knowledge of functional decomposition.

Motivation

The exponential improvement in speed and integration of silicon transistor technology is expected to slow down as devices approach nanometer dimensions [116]. The search for functional nanometer-scale structures led to the exploration of alternative computation schemes. A number of devices based on gating the flow of electrons have been proposed, including quantum dots [76], organic molecules [152], carbon nanotubes [156], nanowires [84], and the motion of single atoms or molecules [68]. Other computation schemes, operating on different principles, include electrons confined in quantum dot cellular automatons [104, 6], magnetic dot cellular automatons [46], and solutions of interacting DNA molecules [105, 31]. Computation can also be performed by purely mechanical means [58, 80], as in the calculating engine of Babbage [148].

We consider a possibility of a computation scheme based on random Boolean networks (RBNs). An RBN is a synchronous Boolean automaton with n vertices. Each vertex has k incoming edges, selected at random, and an associated Boolean function. Functions are selected so that they evaluate to the values 0 and 1 with given probabilities p and $1 - p$, respectively.

Our interest in RBNs is due to their attractive fault-tolerant features. The parameters of an RBN can be tuned so that the network exhibits *self-organized critical behavior* ensuring both stability and evolutionary improvements. On one hand, different kind of faults, e.g. a change in the state of a particular vertex, or connection, typically cause no variations in network's

dynamics. On the other hand, if a sufficient number of mutations is allowed, a network can adopt to the changing environment by re-configuring its structure.

Background on RBN

RBNs were introduced by Kauffman in 1969 in the context of gene expression and fitness landscapes [92]. Later, they were applied to the problems of cell differentiation [83], immune response [94], evolution [28], and neural networks [8, 4]. They have attracted the interest of physicists due to their analogy with the disordered systems studied in statistical mechanics, such as the mean field spin glass [54, 52, 53].

The parameters k and p determine the dynamics of an RBN. If a vertex controls many other vertices, and the number of controlled vertices grows in time, the RBN is said to be in a *chaotic phase* [109]. Typically such a behavior occurs for large values of $k \sim n$. The next states of the RBN are random with respect to the previous ones. The dynamics of the network is very sensitive to changes in the state of a particular vertex, associated Boolean function, or network connections.

If a vertex controls only a small number of other vertices and their number remains constant in time, the RBN is said to be in a *frozen phase* [70]. Usually, independently on the initial state, after a few steps, the network reaches a stable state. This behavior usually occurs for small values of k, such as $k = 0$ or 1.

There is a critical line between the frozen and the chaotic phases, when the number of vertices controlled by a vertex grows in time, but only up to a certain limit [10]. Statistical features of RBNs on the critical line are shown to match the characteristics of real cells and organisms [92, 93]. The minimal disturbances create typically only slight variations in the network's dynamics. Only some rare perturbations evoke radical changes.

For a given probability p, there is a critical number of inputs k_c below which the network is in the frozen phase and above which the network is in the chaotic phase [54]:

$$k_c = \frac{1}{2p(1-p)}. \tag{4.9}$$

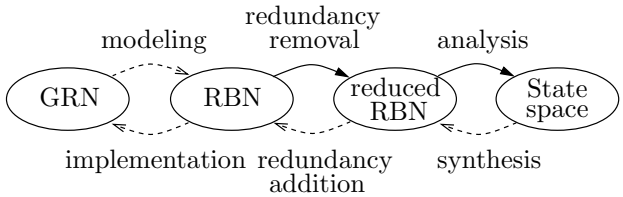


Figure 4.16: Solid and dotted arrows show solved and open problems, respectively.

Contribution and Future Work

Paper G presents an efficient algorithmic framework for the simulation of large RBNs. The presented algorithm for computing attractors uses BDDs to represent the state space of the network implicitly. This allows us to obtain exact results for much larger networks than previously possible. Previous algorithms could only handle networks with less than 32 non-redundant vertices [11, 157, 19, 143]. For larger networks, the median instead of the exact values on the number of attractors was computed by simulation [143].

The ideas we describe are preliminary, and more research is needed to justify them. Figure 4.16 summarizes what remains to be done. Solid arrows show the problems which have been solved (partially or completely). These are the problem of removing redundant vertices from an RBN and the problems related to the analysis of the state space of an RBN, e.g. computing attractors.

Dotted arrows show the problems which have not been solved yet. *Synthesis* is the problem of constructing a reduced RBN which realizes the functionality specified by a given state transition graph. *Redundancy addition* is the problem of adding redundancy to a reduced RBN so that resulting RBN exhibits *critical line* behavior. *Modeling* and *implementation* are the problems of deriving an RBN model of a given *Gene Regulatory Network* (*GRN*), and designing a GRN corresponding to the behavior of a given RBN, respectively. The level of understanding of the organizing principles of gene regulation and signal transduction networks in cells needs to be advanced before the modeling and implementation problems can be addressed. Then, a functional nano-scale device operating on the principles of gene interactions may become a reality.

4.7 Conclusion and Open Problems

Each of the seven papers included in this dissertation indicates open problems and paths to follow that are still relevant after their publication.

In general, and regarding the decomposition algorithms presented herein, there are two major lines for future work. One of them involves integrating these algorithms into existing logic synthesis tools in order to explore the best way to use these techniques in an industrial setting. It is important to ascertain the influence of early stages in the design flow over the performance and applicability of our techniques, and also to find the ways to maximize the optimization that following stages may achieve as a result of our manipulation of the circuit. The other line is, of course, improving the algorithms themselves. There is still plenty of room for optimization we have not yet implemented and for improvements on theoretical grounds we have not yet discovered. In particular, ongoing work includes developing a more efficient algorithm for computing multiple-vertex dominators.

Regarding the Random Boolean Networks presented in our last paper, they offer a wealth of new lines of work. Compositionality is not as straightforward to define in this context as it is in the case of CMOS based electronics. Due to the particular way inputs and outputs are represented within this model (inputs are bits, outputs are "fixed points" or sub-graphs) there is no unique way in which one can define compositionality. It is far from clear which of these alternative views will yield the best practical results. In turn, each of these possible definitions of compositionality will lead to decomposition techniques quite diverse from the ones that have proved so successful in CMOS technology. There is indeed an exciting research future in this area.

Chapter 5

Complete List of Publications

The following is the complete list of publications produced during my PhD studies. A star ⋆ indicates the publications included in this thesis. I also give the details of my specific contributions to each of the publications marked.

2005

⋆ **Kauffman Networks: Analysis and Applications**, E. Dubrova, M. Teslenko, and A. Martinelli. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD 2005)*, November 6–10, 2005, San Jose, CA, USA, pp. 479–484.

I came out with the idea of applying Kauffman networks to logic synthesis. I also contributed my expertise on BDDs to the implementation of the algorithm for computing attractors.

An Efficient Structural Technique for Boolean Decomposition, A. Martinelli and E. Dubrova. In *Proceedings of SPIE – VLSI Circuits and Systems II, Vol. 5837*, June 2005, Sevilla, Spain, pp. 913–918.

Achieving Fault Tolerance by Cost Bound Decomposition, A. Martinelli and E. Dubrova. In *Proceedings of the Swedish System-on-Chip Conference (SSoCC'05)*, April 18–19, 2005, Tammsvik, Sweden.

⋆ **Bound Set Selection and Circuit Re-Synthesis for Area/delay Driven Decomposition**, A. Martinelli and E. Dubrova. In *Proceed-*

ings of the Design and Test in Europe Conference 2005 (DATE'05), interactive presentation, March 7–11, 2005, Munich, Germany, pp. 430–431.

The idea as well as the implementation of the algorithm in this paper are mine.

⋆ **Bound-set Preserving ROBDD Variable Orderings May Not Be Optimum**, M. Teslenko, A. Martinelli, and E. Dubrova. In *IEEE Transactions on Computers*, Vol. 54, number 2, pp. 236–238, February 2005.

The problem addressed by the paper was investigated by the three authors over a long period of time (about four years). The counterexample itself is due to the first author, M. Teslenko.

2004

⋆ **On Relation Between Non-Disjoint Decomposition and Multiple-Vertex Dominators**, E. Dubrova, M. Teslenko, and A. Martinelli. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2004)*, May 23–26, 2004, Vancouver, Canada, pp. 493–496.

The idea of this paper was originated by the first author, E. Dubrova. Together with the second author, M. Teslenko, I contributed to the implementation of the algorithm and conducted experiments to evaluate it.

⋆ **Disjoint-Support Boolean Decomposition Combining Functional and Structural Methods**, A. Martinelli, R. Krenz, and E. Dubrova. In *Proceedings of the IEEE Asia and South Pacific Design Automation Conference 2004 (ASP-DAC 2004)*, January 27–30, 2004, Yokohama, Japan, pp. 183–189.

I proposed the idea of using the combined approach, and produced the tool which integrated the functional and structural algorithms. I also implemented the functional algorithm, while the structural one was implemented by R. Krenz.

2003

Roth-Karp Decomposition Combining Functional and Structural Techniques, R. Krenz, A. Martinelli, and E. Dubrova. In *Proceedings of International Workshop on Logic Synthesis 2003 (IWLS'03)*, pp. 18–23, Laguna Beach, CA, May 2003.

⋆ **A BDD-Based Fast Heuristic Algorithm for Disjoint Decomposition**, T. Bengtsson, A. Martinelli, and E. Dubrova. In *Proceedings of the IEEE Asia and South Pacific Design Automation Conference 2003, (ASP-DAC 2003)*, Kitakyushu, Japan, January 2003, pp. 191–196.

My contributions to this paper are implementing the presented heuristic, together with Tomas Bengtsson, and providing my own implementation of the exact decomposition algorithm which is used in the experimental results section to evaluate the heuristic.

2002

⋆ **Roth-Karp Decomposition of Large Boolean Functions with Application to Logic Design**, A. Martinelli, T. Bengtsson, E. Dubrova, and A. J. Sullivan. In *Proceedings of NORCHIP 2002 (NORCHIP'02)*, Copenhagen, Denmark, November 2002, pp. 183–189.

This paper is based on my generalization of the approach from the paper in IWLS'02 cited below. I also implemented most of the code.

A Fast Heuristic Algorithm for Disjunctive Decomposition of Boolean Functions, T. Bengtsson, A. Martinelli, and E. Dubrova. In *Proceedings of International Workshop on Logic Synthesis 2002, (IWLS'02)*, pp. 51–57, New Orleans, Louisiana, USA, June 2002.

Papers

Paper A

A BDD-Based Fast Heuristic Algorithm for Disjoint Decomposition

Tomas Bengtsson, Andrés Martinelli, Elena Dubrova. Published in the "Proceedings of the IEEE Asia and South Pacific Design Automation Conference 2003" (ASP-DAC 2003), January, 2003, Kitakyushu, Japan, pp. 191–196.

A BDD-Based Fast Heuristic Algorithm for Disjoint Decomposition

Tomas Bengtsson* Andrés Martinelli† Elena Dubrova†

Abstract

This paper presents a heuristic algorithm for disjoint decomposition of a Boolean function based on its ROBDD representation. Two distinct features make the algorithm feasible for large functions. First, for an n-variable function, it checks only $O(n^2)$ candidates for decomposition out of $O(2^n)$ possible ones. A special strategy for selecting candidates makes it likely that all other decompositions are encoded in the selected ones. Second, the decompositions for the approved candidates are computed using a novel IntervalCut algorithm. This algorithm does not require reordering of ROBDD. The combination of both techniques allows us to decompose the functions of size beyond that possible with the exact algorithms. The experimental results on 582 benchmark functions show that the presented heuristic finds 95% of all decompositions on average. For 526 of those functions, it finds 100% of the decompositions.

A.1 Introduction

The *disjoint decomposition* of a Boolean function is a representation of type $f(X) = h(g(Y), Z)$ with Y and Z being sets of variables partitioning the set X. Disjoint decomposition has many applications in computer science and discrete mathematics, including logic synthesis (decomposition of Boolean functions), reliability theory (decomposition of coherent systems [22]), game

*Jönköping University, Embedded systems/ING, Jönköping, Sweden
†Royal Institute of Technology, IMIT/KTH, Stockholm, Sweden

theory (decomposition of simple n-persons games [139]) and combinatorial optimization problems over graphs and networks (see [121] for an overview).

This wide range of applications makes it important to have efficient algorithms for finding all, or at least some, decompositions for a given structure. Fast decomposition algorithms are known for binary relations and graphs [23, 47, 78]. For Boolean functions, however, the existing methods either involve the solution of an NP-complete problem (as in [20]) or have exponential running time [55, 141, 142, 150]. More recent ROBDD-based decomposition algorithms, including [15, 118, 113], show much better average-time performance.

This paper presents a heuristic algorithm targeting to find all disjoint decompositions of an n-variable Boolean function represented by a ROBDD. The heuristic is based on two properties: (1) all decompositions of a Boolean function (which can be $O(2^n)$) can be uniquely described by a certain subset of decompositions A (which is only $O(n)$); (2) there exist a best variable ordering for a ROBDD in which the variables Y from any decomposition $f(X) = h(g(Y), Z)$ belonging to A are adjacent.

If we had such a best ordering, we could examine all its linear intervals to find which Y results in a decomposition $f(X) = h(g(Y), Z)$. However, computing best orderings is infeasible for large functions. The algorithm presented in this paper is *heuristic* because it starts from a "good" ordering which is not necessarily keeping the variables Y adjacent. The experimental results show that if *sifting* ordering algorithm [131] is used to get a "good" initial order, then our heuristic finds 95% of all decompositions on average. The presented heuristic algorithm is also able to decompose functions which are too large for the exact algorithms.

A.2 Previous work

The first major investigation on the subject was carried out by Ashenhurst [7]. He studied simple disjoint decomposition $f(X) = h(g(Y), Z)$ for Boolean functions $f, g, h : B^n \to B$, where $B = \{0, 1\}$. Ashenhurst's fundamental contribution is a theorem which states that any Boolean function has a unique *disjoint tree-like decomposition* such that all possible simple disjoint decompositions of f are exhibited.

Curtis [48] and Roth and Karp [130] extended Ashenhurst theory to the decomposition of type $f(X) = h(g(Y), Z)$ with g, H being multiple-

valued functions of type $g\ B^{|Y|} \rightarrow M$ and $h\ M \times B^{|Z|} \rightarrow B$, where $M = \{0, 1, ..., m-1\}$. The function g can be encoded by $k = \lceil \log_2 m \rceil$ Boolean functions $g_1, g_2, \ldots, g_k$, giving a decomposition of the form

$$f(X) = h(g_1(Y), \ldots, g_k(Y), Z),$$

often referred to as *Roth-Karp* decomposition. Unfortunately Ashenhurst's main theorem does not extend directly to multiple-valued functions (for a counterexample see chapter 4 of [60]). A consequence of this is that there is no unique disjoint tree-like Roth-Karp decomposition. Von Stengel [154] has defined a class of multiple-valued functions for which Ashenhurst's main theorem holds.

Early algorithms for decomposition used *decomposition charts* [7], [48]. The decomposition chart for $f(Y, Z)$ is a two-dimensional table where the columns represent all combinations of the variables from the set Y and the rows represent all combinations of the variables from the set Z. The set Y is a bound set if and only if the chart has *column multiplicity* at most two, i.e. there are at most two distinct columns in the chart [7].

In a short time, decomposition charts were abandoned in favor of *cube* representation [90]. The task of computing column multiplicity on charts was replaced by the task of computing *compatible classes* for a set of cubes. Two assignments $x_1, x_2 \in B^{|Y|}$ are said to be *compatible with respect to the reference function* $f(Y, Z)$ if, for all $y \in B^{|Z|}$ such that $f(x_1, y)$ and $f(x_2, y)$ are defined, $f(x_1, y) = f(x_2, y)$ [90]. The set Y is a bound set if and only if $B^{|Y|}$ can be partitioned into $k \leq 2$ mutually compatible classes [90]. If $f(X)$ is completely specified, then compatibility is an equivalence relation and k is the number of equivalence classes. It is easy to see the one-to-one mapping between a column in a decomposition chart and a compatible class.

Due to the exponential size of decomposition charts and cube representations, early decomposition algorithms were rarely applied to large practical circuits. Instead, *algebraic* methods were used [33]. ROBDDs [36] made possible developing new algorithms for decomposition, feasible for much larger functions than previously possible.

In a ROBDD, the column multiplicity can be easily computed by moving the variables Y to the upper part of the graph and checking the number of children below the boundary line, usually called *cut* line. The decomposition $f(X) = h(g(Y), Z)$ exists if and only if there are only two children below the cut line [132].

This approach has been adopted by a number of BDD-based decomposition algorithms [132, 99, 41, 135]. Stanion and Sechen [146] used cut to find *quasi-algebraic* decomposition of the form $f(X) = g(Y) \odot h(Z)$, where "$\odot$" is any binary Boolean operation and $|Y \cup Z| = k$ for some $k \geq 0$. This type decomposition is often referred to as *bi-decomposition* [159, 119].

BDD-based decomposition algorithms following cut-strategy proved to be orders of magnitude faster than those based on decomposition charts and cube representations. However, they require reordering of variables of BDD to move the variables on the top or to check bi-decompositions for partitionings which are not consistent with the variable order. As an alternative, a number of methods use the fact that BDDs themselves are a decomposed representation of the function and exploit the structure of BDDs, rather than cut, to find disjoint decompositions. Karplus [91] extended the classical concept of *dominator* on graphs [103] to 0,1-dominators on BDDs. A node v is a 1-dominator (0-dominator) if every path from the root to one (zero) terminal node contains v. If v is a 1-dominator, then the function represented by the BDD possesses a conjunctive (AND) decomposition. If v is a 0-dominator, then the function can be decomposed disjunctively (OR). This idea was extended by Yang et al [161] to XOR-type decompositions and to more general type of dominators. Minato and De Micheli [118] presented an algorithm which computes disjoint decompositions by generating irreducible sum-of-product for the function from its BDD and applying factorization. The algorithm of Bertacco and Damiani [15] makes a single traversal of the BDD to identify the decomposition of the co-factors and then combine them to obtain the decomposition for the entire function. The algorithm is impressively fast; however, as Sasao has observed in [133], it fails to compute some of the disjoint decompositions. This problem was corrected by Matsunaga [113], who added the missing cases in [15] allowing to treat the OR/XOR functions correctly. The algorithm [113] appears to be the fastest of existing exact algorithms for finding all disjoint decompositions.

A.3 New heuristic algorithm

The new heuristic algorithm is based on the following two properties.

Proposition 1. *All disjoint decompositions of an n-variable Boolean function can be uniquely described by a certain subset of disjoint decompositions A. The size of A is* $O(n)$.

Proposition 2. *There exist a best variable ordering for a ROBDD for f in which the variables Y from any decomposition $f(X) = h(g(Y), Z)$ belonging to A are adjacent.*

Property 1 follows from the results of [154]. We describe these results briefly in Section A.3. Property 2 follows from the main theorem of [59].

The presented algorithm examines all linear intervals of variables from a given ordering of a ROBDD and, for each interval Y, checks whether it is a bound set. The procedure **IntervalCut** described in Section A.3, is used to perform the checking as well as to compute the functions g and h in the resulting decomposition $f(X) = h(g(Y), Z)$.

Properties of the disjoint decomposition

This section describes the properties of the disjoint decomposition from [154], implying Property 1. The formulation of the definitions and theorems is adjusted to the notation of this paper.

Definition A.3.1*:* A bound set Y of $f(X)$, $Y \subset X$, is strong if any other bound set of $f(X)$ is either a subset of Y, a super-set of Y, or disjoint to Y.

The partial order induced by set theoretical inclusion between pairs of strong bound sets of f defines a tree.

Definition A.3.2*:* The decomposition tree $T(f)$ of $f(X)$ is a tree whose nodes represent all strong bound sets of $f(X)$, related by inclusion. Any node has two labels:

(a) a type, which is either "prime" or "full",
(b) an associated function.

The following Theorem shows how decompositions of a function can be derived from its decomposition tree and characterizes the functions associated with the nodes. It also states that the decomposition tree is unique for a given function (up to isotopy/isomorphy). Remind that two Boolean functions are *isotopic* if they are identical up to complementation of variables or function values. Two binary operations $\circ$ and $\bullet$ are *isomorphic* if there is a bijection $\phi : B \to B$ such that $\phi(a \circ b) = \phi(a) \bullet \phi(b)$.

Theorem 7. *Let $T(f)$ be the decomposition tree of a Boolean function $f(X)$ with support set X. Let $Y_1, \ldots, Y_k$ be the children of the root X. Then $f(X)$ has a decomposition of type*

$$f(X) = h(g_1(Y_1), g_2(Y_2), \ldots, g_k(Y_k))$$

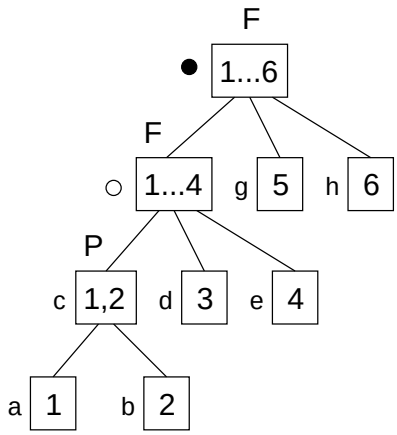


Figure A.1: Example of a decomposition tree.

for functions $g_i : B^{|Y_i|} \to B$ $(1 \le i \le k)$ *and* $h : B^k \to B$ *where*

(a) h *is non-decomposable if* X *is labeled "prime",*

(b) h *is an associative and commutative Boolean operation if* X *is labeled "full",*

(c) h *is unique up to isotopy in (a) and up to isomorphy in (b).*

An example of a decomposition tree is shown in Figure A.1. Abbreviations "P" and "F" stand for labels "prime", and "full", respectively. Letters a, b, c, d, e, g, h denote the functions associated with the nodes, whereas $\bullet$ and $\circ$ denote operations. In accordance with the tree, the complete disjoint decomposition of the function is

$$f(x_1, \ldots, x_6) = (c(a(x_1), b(x_2)) \circ d(x_3) \circ e(x_4)) \bullet g(x_5) \bullet h(x_6)$$

with $\bullet$ and $\circ$ being associative and commutative Boolean operations. a, b, c, d, e, g, h are non-decomposable Boolean functions. In this case all those functions except c are unary Boolean functions (identity or complement).

Theorem 7 shows that the decompositions associated with strong bound sets uniquely represent all disjoint decompositions of a function. These are the decompositions A of Property 1. It was proved in [120] that the number of strong bound sets of an n-variable Boolean function is $O(n)$, while the number of all bound sets is $O(2^n)$.

IntervalCut procedure for finding bound sets

Let V be a set of nodes of a ROBDD G of an n-variable function $f(X)$. Every non-terminal node $v \in V$ has an associated variable index, $index(v) \in \{1, \ldots, n\}$. The index of the root node is 1. In order to have a unified notation in the proof of the main result, we assume that the terminal nodes also have an index, which is $n + 1$.

Suppose that all nodes with $index \leq i$ are in the upper part of the graph and all nodes with $index > i$ are in the lower part of the graph, for some $i \in \{1, \ldots, n\}$. The boundary line between the upper and lower parts of the graph is called $cut(i)$. If the number of nodes with $index > i$ which are children of the nodes above the $cut(i)$ is two, then the set of variables $Y = \{x_1, \ldots, x_i\}$ is a bound set [99].

One possibility to check whether a set of variables Y is a bound set is to move the variables Y to the top of the ROBDD and then check the number of children below $cut(|Y|)$, as in [99, 41]. However, re-ordering is computationally expensive. Instead, we have developed a procedure, called **Interval Cut** which checks whether a given linear interval of variables of a ROBDD is a bound set without reordering. To describe the procedure, we first introduce some definitions.

Suppose the variables Y lie between two cuts, $cut(a)$ and $cut(b)$, such that $a < b$, $a, b \in \{0, \ldots, n\}$. Let $cut\_set(a)$ denote a set of nodes $v \in G$ with indexes $a < index(v) \leq b$ which are children of the nodes above the $cut(a)$ of G. Let G_v stand for a ROBDD rooted at some $v \in cut\_set(a)$. Then, $cut\_set(b_v)$ is the set of nodes $u \in G_v$ with indexes $b < index(u) \leq n + 1$ which are children of the nodes of G_v above the $cut(b)$. If $|cut\_set(b_v)| = 2$, then g_v is a Boolean function represented by the sub-graph rooted at v whose terminal nodes are obtained by replacing the two nodes of $cut\_set(b_v)$. The resulting g_v is unique up to complementation.

Using this notation, we can describe the pseudo code of the algorithm **IntervalCut**(G, a, b) as shown in Figure A.2. Next, we prove that it computes the decompositions correctly.

Theorem 8. *Algorithm* ***IntervalCut***(G, a, b) *computes a decomposition* $f(X) = h(g(Y), Z)$ *in* $O(|cut\_set(a)| \cdot max(|g_v|))$ *time,* $v \in cut\_set(a)$.

Proof: Let Y be the variables between $cut(a)$ and $cut(b)$, Z_1 be the variables above $cut(a)$ and Z_2 be the variables below $cut(b)$. We have $Z_1 \cup Z_2 = Z$ and $Y \cup Z = X$.

IntervalCut(G, a, b)
input: ROBDD G of $f(X)$, two cuts $cut(a)$ and $cut(b)$, $a < b$, $a, b \in \{0, \ldots, n\}$.
output: "not a bound set" if the set of variables Y between $cut(a)$ and $cut(b)$ is not a bound set of $f(X)$; functions g and h if Y is a bound set resulting in $f(X) = h(g(Y), Z)$.

for *all* $v \in cut\_set(a)$
  if $(|cut\_set(b_v)| > 2)$
    return(*"not a bound set"*);
for *all* $v_1, v_2, \ldots, v_k \in cut\_set(a)$
  if $(g_{v_i} \neq g_{v_{i+1}})$ /* *up to complementation* */
    return(*"not a bound set"*);
h = *substitute each sub-graph* g_v, $\forall v \in cut\_set(a)$, *by a node;*
$g = g_v$;
return(g, h);

Figure A.2: Pseudo code of the **IntervalCut** procedure.

Let $k_v(Z_1)$ be a function which is a sum of all the paths leading to a node $v \in cut\_set(a)$. Then f can be co-factored with respect to k_v as

$$f(X) = \sum_{\forall v \in cut\_set(a)} k_v(Z_1) \cdot f|_{k_v}(Y, Z_2) \tag{A.1}$$

If $|cut\_set(b_v)| = 2$, then Y is a bound set for $f|_{k_v}$ so it can be decomposed as

$$f|_{k_v}(Y, Z_2) = h_v(g_v(Y), Z_2) \tag{A.2}$$

for some h_v, g_v. Furthermore, if for all $v \in cut\_set(a)$ the functions g_v are equal up to complementation, then we can denote g_v by g and write (A.2) as

$$f|_{k_v}(Y, Z_2) = h_v(g(Y), Z_2) \tag{A.3}$$

From (A.1) and (A.3) we can conclude that f can be represented as

$$f(X) = h(g(Y), Z)$$

with $h = \sum_{\forall v \in cut\_set(a)} k_v \cdot h_v$.

Let $max(|g_v|)$ be the size of the largest sub-graph representing g_v, for some $v \in cut\_set(a)$. Since substitution of a ROBDD by a node is a constant-time operation, the complexity of the pseudo code in Figure A.2 is $O(|cut\_set(a)| \cdot max(|g_v|))$.

□

A.4 Experimental results

To make a thorough evaluation of the presented heuristic, we have implemented an exact decomposition algorithm[1] from [67] and applied both, exact and heuristic versions, to *iwls*93 benchmark set. For all single outputs, for which the exact algorithm did not time out[2], 582 in total, we have computed the total number of strong bound sets found by each algorithm. In the first set of experiments, we used *sifting* ordering algorithm [131] to get a good initial order for ROBDDs. The heuristic algorithm has succeeded to find 95% of all the decompositions on average. For 526 of those 582 single-output functions, it found 100% of the decompositions. In the second set of experiments, we switched the sifting off, and build ROBDDs using the breadth first traversal order from the benchmark's circuit description. For 191 functions out of 582 the result got worse (by 57% on average). Nevertheless, the heuristic still found all the decompositions for 365 functions.

We have also applied the presented heuristic to the benchmarks reported in [118], [15] and [113]. The results are summarized in Table A.1. Column 4 shows how many non-trivial strong bound sets are found for each benchmark by our algorithm. Every output is handled as a separate function. The number given in Column 4 is the total sum of bound sets for all the outputs. Columns 5-8 show runtime comparison. Our experiments were run on Sun Ultra 60 operating with two 360 MHz CPU and with 1024 MB RAM main storage. The algorithm [118] uses a SUN Ultra 30, [15] uses a PC equipped with 150 MHz Pentium and 96 MB RAM main storage and [113] uses a PC with Pentium-II 233Mhz processor.

[1] We have chosen [67] because this algorithm actually builds decomposition trees. It computes only $O(n)$ strong bound sets which are the nodes of $T(f)$.

[2] Time limit 30 min per circuit.

Table A.1: Experimental results; "−" indicates that information for the benchmark is not provided; ">" indicates that information is only provided for one of the outputs.

				CPU time (sec)			
name	in	out	bound sets	presented heuristic	exact alg [118]	exact alg [15]	exact alg [113]
alu2	10	6	3	0.0002	-	0.28	-
alu4	14	8	2	0.0009	-	0.37	0.15
apex1	45	45	83	0.008	59.0	1.01	-
apex2	38	3	16	0.001	5.9	1.14	-
apex3	54	50	23	0.008	44.3	-	-
apex4	9	19	4	0.002	-	0.33	-
apex5	114	88	196	0.032	-	2.34	-
apex6	135	99	258	0.008	13.1	2.62	0.41
apex7	49	37	96	0.006	1.7	1.03	0.37
b9	41	21	49	0.001	-	-	0.02
C432	36	7	10	0.002	415.4	1.23	0.28
C499	41	32	68	5.2	-	83.47	8.80
C880	60	26	45	0.046	-	2.71	0.92
C1355	41	32	0	5.2	-	91.25	8.87
C1908	33	25	15	0.23	-	7.58	1.42
C3540	50	22	18	2.8	-	21.1	3.48
cmb	16	4	4	0.002	-	0.36	-
CM42	4	10	10	0.0006	-	0.15	-
CM85	11	3	15	0.0003	-	0.27	-
CM150	21	1	1	<0.0001	-	0.51	-
comp	32	3	47	0.002	-	0.71	-
count	35	16	47	0.007	-	0.73	0.01

continued on next page...

Table A.1 – continued from previous page

				CPU time (sec)			
name	in	out	bound sets	presented heuristic	exact alg[118]	exact alg [15]	exact alg [113]
dalu	75	16	42	0.015	>0.8	-	-
des	256	245	688	0.041	-	-	0.36
e64	65	65	63	0.51	-	1.31	-
f51m	8	8	6	0.0004	-	0.26	-
frg2	143	139	532	0.032	19.2	2.86	0.15
k2	45	45	85	0.008	-	1.04	-
lal	26	19	57	0.002	-	0.55	-
misex2	25	18	29	0.003	-	0.57	-
mux	21	1	1	0.0001	-	0.48	-
pair	173	137	725	0.040	-	4.02	7.36
PARITY	16	1	1	0.001	-	0.38	-
rot	135	107	296	0.039	-	22.62	-
seq	41	35	135	0.009	67.8	1.10	-
s298	17	20	15	0.0004	-	0.40	-
s420	35	18	18	0.007	-	0.75	-
s444	24	27	65	0.001	-	0.54	-
s526	24	27	45	0.002	-	0.52	-
s641	54	42	138	0.003	-	1.12	-
s832	23	24	37	0.003	-	0.54	-
s953	45	52	40	0.003	-	20.97	-
s1196	32	32	33	0.002	-	0.71	-
s1238	32	32	33	0.002	-	0.75	-
s1423	91	79	38	0.066	-	12.48	-
s1488	14	25	38	0.002	-	0.36	-
s1494	14	25	38	0.002	-	0.34	-
term1	34	10	65	0.002	-	0.75	-
too_large	38	3	17	0.001	>1.0	-	0.09
ttt2	24	21	44	0.002	-	0.55	-

continued on next page...

Table A.1 – concluded from previous page

				CPU time (sec)			
name	in	out	bound sets	presented heuristic	exact alg[118]	exact alg [15]	exact alg [113]
vda	39	17	30	0.003	>0.5	0.4	-
x3	135	99	278	0.008	-	2.69	-
x4	94	71	180	0.008	-	1.90	-

A.5 Conclusion

This paper presents a heuristic algorithm for finding disjoint decompositions of Boolean functions. Benchmark experiments demonstrate the effectiveness of the described technique.

Future work includes extension of the presented algorithm to Roth-Karp decomposition. We are also investigating a possibility of combining **IntervalCut** with decomposition algorithms exploiting the structure of BDDs, like [113].

Acknowledgment

This work was supported in part by IBM Partnership Award.

Paper B

Roth-Karp Decomposition of Large Boolean Functions with Application to Logic Design

Andrés Martinelli, Tomas Bengtsson, Elena Dubrova and Andrew J. Sullivan. *Published in the "Proceedings of the 20st IEEE Norchip Conference 2002" (NORCHIP 2002), Copenhagen, Denmark, November 2002, pp. 183–189.*

Roth-Karp Decomposition of Large Boolean Functions with Application to Logic Design

Andrés Martinelli* Tomas Bengtsson† Elena Dubrova*
Andrew J. Sullivan‡

Abstract

This paper presents an algorithm for Roth-Karp decomposition of Boolean functions. Roth-Karp decomposition is an extension of classical simple disjoint decomposition $f(X) = h(g(Y), Z)$ allowing the number of outputs in the extracted logic block $g(Y)$ to be greater than one. Roth-Karp decomposition has many applications in CAD, including logic synthesis, testing and verification. Many efficient algorithms for finding all simple disjoint decompositions have been presented. However, no feasible exact algorithm is known for Roth-Karp decomposition. As a practical alternative, we propose a heuristic algorithm that quickly finds many, but not all, Roth-Karp decompositions using a BDD representation of the function. The algorithm does not require time-costly variable reordering of the BDD. An extensive set of experiments on benchmark functions demonstrates the effectiveness of our approach.

B.1 Introduction

Most approaches to the logic synthesis of digital systems consist of two phases: a technology-independent phase that manipulates and optimizes functions; and a technology-mapping phase that maps functions onto a set of

*{andres, elena}@imit.kth.se, Royal Institute of Technology, IMIT/KTH, Stockholm, Sweden

†beto@ing.hj.se, Jönköping University, Embedded systems/ING, Jönköping, Sweden

‡sullia@us.ibm.com, IBM EDA group Fishkill, N.Y., USA

gates in a specific target technology. The technology-independent phase for two-level synthesis, resulting in two-level devices such as programmable logic arrays, is based on minimization techniques [26]. For multi-level synthesis *decomposition* is the essential step in the technology-independent phase, leading to devices with multi-level structure such as field-programmable gate arrays [35].

Generally, the problem of decomposition of functions can be formulated as follows. Given a function f, express it as a composite function of some set of new functions. Sometimes, a composite expression can be found in which the new functions are significantly simpler than f. Then the design of a logic circuit realizing f may be accomplished by designing circuits realizing the simpler functions of the composite representation, thus reducing the the overall cost of implementing f.

However, the problem of selecting the "best" decomposition minimizing the overall cost of realization of a given function appears to be far too difficult to be solved exhaustively. Therefore, all efforts to apply decomposition theory to the design of Boolean and multi-valued logic circuits restrict the decomposition to be obtained to a particular type. In this paper we consider disjoint decompositions only. The basis for the different types of disjoint decomposition is the *simple disjoint* decomposition where a function$f(X)$, $X = \{x_1, x_2, \ldots, x_n\}$, is expressed as a composite function of two functions g and h, namely

$$f(X) = h(g(Y), Z) \tag{B.1}$$

where Y and Z are sets of variables forming a partition of the set of variables X. If f, g and h are Boolean functions, then in equation (B.1) the original function f specifying an n-input, 1-output Boolean circuit is replaced by the specification of two Boolean circuits, one having $|Y|$ inputs and one output, and the other having $1 + |Z|$ inputs and one output. Every set of variables X such that f has a decomposition like (B.1) is called a *bound set* for f. Such a decomposition exist trivially for X given by any singleton set x_i or the all-set X.

If C_n is an upper bound on the cost of realizing an Boolean function of n variables, then the total cost of realizing these two circuits is bounded above by $C_{|Y|}+C_{(1+|Z|)}$. Because the cost bound C_n usually increases nearly exponentially with n [138], the discovery of any nontrivial decomposition of the form (B.1) greatly reduces the cost of realizing f.

Unfortunately, the fraction of all Boolean functions of n variables possessing nontrivial disjoint decompositions of type (B.1) approaches zero as n approaches infinity [138, p. 90]. Therefore, simple disjoint decomposition has been extended to a more general type of decomposition, known as *Roth-Karp* decomposition [90]. This decomposition has the form

$$f(X) = h(g(Y), Z)$$

with $f : \{0,1\}^n \to \{0,1\}$, $g : \{0,1\}^{|Y|} \to \{0,1,\ldots,m-1\}$ and $h : \{0,1,\ldots,m-1\} \times \{0,1\}^{|Z|} \to \{0,1\}$. In such a decomposition the m-valued function h of Boolean variables can be coded by $k = \lceil \log_2 m \rceil$ Boolean functions $g_1, g_2, \ldots, g_k$, giving a decomposition of the form

$$f(X) = h(g_1(Y), g_2(Y), \ldots, g_k(Y), Z) \tag{B.2}$$

with all functions being Boolean. Methods for choosing good encodings are presented in [102, 82]. The decomposition (B.2) includes as a subclass the simple disjoint decompositions ($m = 1$) mentioned above. As long as f is a function of more than three variables, such a decomposition can always be found with $g_1(Y), g_2(Y), \ldots, g_k(Y)$ and h each having fewer arguments than f, for there always exists a decomposition of the form

$$f(X) = f(Y, x_n) = h(g_1(Y), g_2(Y), x_n)$$

with $Y = \{x_1, \ldots, x_{n-1}\}$. Thus, the decomposition (B.2) allows the simplification of *any* Boolean function.

The rest of the paper is organized as follows. Section B.2 reviews previous work in the area of decomposition. Section B.3 presents the new algorithm for computing Roth-Karp decomposition. Section B.4 shows the experimental results. Section B.5 concludes the paper.

B.2 Previous work

The first major investigation on the subject was carried out by Ashenhurst [7]. He studied simple disjoint decomposition $f(X) = h(g(Y), Z)$ for Boolean functions $f, g, h : B^n \to B$, where $B = \{0,1\}$. Ashenhurst's fundamental contribution is a theorem which states that any Boolean function has a unique *disjoint tree-like decomposition* such that all possible simple disjoint decompositions of f are exhibited.

Curtis [48] and Roth and Karp [130] extended Ashenhurst theory to the decomposition of type $f(X) = h(g(Y), Z)$ with g, h being multiple-valued functions of type $g : B^{|Y|} \rightarrow M$ and $h : M \times B^{|Z|} \rightarrow B$, where $M = \{0, 1, \ldots, m-1\}$. The function g can be encoded by $k = \lceil \log_2 m \rceil$ Boolean functions $g_1, g_2, \ldots, g_k$, giving a decomposition of the form $f(X) = h(g_1(Y), \ldots, g_k(Y), Z)$, often referred to as *Roth-Karp* decomposition. Unfortunately Ashenhurst's main theorem does not extend directly to multiple-valued functions (for a counterexample see chapter 4 of [60]). A consequence of this is that there is no unique disjoint tree-like Roth-Karp decomposition. Von Stengel [154] has defined a class of multiple-valued functions for which Ashenhurst's main theorem holds.

Early algorithms for decomposition used *decomposition charts* [7], [48]. The decomposition chart for $f(Y, Z)$ is a two-dimensional table where the columns represent all combinations of the variables from the set Y and the rows represent all combinations of the variables from the set Z. The set Y is a bound set if and only if the chart has *column multiplicity* at most two, i.e. there are at most two distinct columns in the chart [7].

In a short time, decomposition charts were abandoned in favor of *cube* representation [90]. The task of computing column multiplicity on charts was replaced by the task of computing *compatible classes* for a set of cubes. Two assignments $x_1, x_2 \in B^{|Y|}$ are said to be *compatible with respect to the reference function* $f(Y, Z)$ if, for all $y \in B^{|Z|}$ such that $f(x_1, y)$ and $f(x_2, y)$ are defined, $f(x_1, y) = f(x_2, y)$ [90]. The set Y is a bound set if and only if $B^{|Y|}$ can be partitioned into $k \leq 2$ mutually compatible classes [90]. If $f(X)$ is completely specified, then compatibility is an equivalence relation and k is the number of equivalence classes. It is easy to see the one-to-one mapping between a column in a decomposition chart and a compatible class.

Due to the exponential size of decomposition charts and cube representations, early decomposition algorithms were rarely applied to large practical circuits. Instead, *algebraic* methods were used [33]. ROBDDs [36] made it possible to develop new algorithms for decomposition, feasible for much larger functions than previously possible.

In a ROBDD, the column multiplicity can be easily computed by moving the variables Y to the upper part of the graph and checking the number of children below the boundary line, usually called *cut* line. The decomposition $f(X) = h(g(Y), Z)$ exists if and only if there are only two children below the cut line [132].

This approach has been adopted by a number of BDD-based decompo-

sition algorithms [132, 99, 41, 135]. Stanion and Sechen [146] used cut to find *quasi-algebraic* decomposition of the form $f(X) = g(Y) \odot h(Z)$, where "$\odot$" is any binary Boolean operation and $|Y \cup Z| = k$ for some $k \geq 0$. This type decomposition is often referred to as *bi-decomposition* [159, 119].

BDD-based decomposition algorithms following cut-strategy proved to be orders of magnitude faster than those based on decomposition charts and cube representations. However, they require reordering of variables of BDD to move the variables on the top or to check bi-decompositions for partitions which are not consistent with the variable order. As an alternative, a number of methods use the fact that BDDs themselves are a decomposed representation of the function and exploit the structure of BDDs, rather than cut, to find disjoint decompositions. Karplus [91] extended the classical concept of *dominator* on graphs [103] to 0,1-dominators on BDDs. A node v is a 1-dominator (0-dominator) if every path from the root to one (zero) terminal node contains v. If v is a 1-dominator, then the function represented by the BDD possesses a conjunctive (AND) decomposition. If v is a 0-dominator, then the function can be decomposed disjunctively (OR). This idea was extended by Yang et al [161] to XOR-type decompositions and to more general type of dominators. Minato and De Micheli [118] presented an algorithm which computes disjoint decompositions by generating irreducible sum-of-product for the function from its BDD and applying factorization. The algorithm of Bertacco and Damiani [15] makes a single traversal of the BDD to identify the decomposition of the co-factors and then combine them to obtain the decomposition for the entire function. The algorithm is impressively fast; however, as Sasao has observed in [133], it fails to compute some of the disjoint decompositions. This problem was corrected by Matsunaga [113], who added the missing cases in [15] allowing to treat the OR/XOR functions correctly. The algorithm [113] appears to be the fastest of existing exact algorithms for finding all disjoint decompositions.

B.3 Generalized cut algorithm

Notation

The bound sets used in Roth-Karp decomposition are a more general case of the notion of classic (Boolean) bound sets defined in Section B.1.

Definition B.3.1*:* The set $\{Y\}$ is said to be a k-bound set if there exists a decomposition

$$f(X) = h(g(Y), Z) \tag{B.3}$$

for some functions g, h of type $g : \{0,1\}^{|Y|} \rightarrow \{0,1,\ldots,m-1\}$ and $h : \{0,1,\ldots,m-1\} \times \{0,1\}^{Z} \rightarrow \{0,1\}$, such that $2 \leq m < 2^k$.

These bound sets can be determined by decomposition chart or cut methods, by relaxing the requirement of having exactly 2 different columns (or 2 different cut nodes), to allow a number of columns (or cut nodes) up to 2^k [90].

Basic idea of the method

Let V be a set of nodes of a ROBDD G of an n-variable function $f(X)$. Every non-terminal node $v \in V$ has an associated variable index, $index(v) \in \{1,\ldots,n\}$. The index of the root node is 1, and we let the terminal nodes have also an index, which is $n+1$.

Suppose that all nodes with $index \leq i$ are in the upper part of the graph and all nodes with $index > i$ are in the lower part of the graph, for some $i \in \{1,\ldots,n\}$. The boundary line between the upper and lower parts of the graph is called $cut(i)$.

If the number of nodes with $index > i$ which are children of the nodes above the $cut(i)$ is at most 2^k, for a given k, then, by Definition B.3.1, the set of variables $Y = \{x_1,\ldots,x_i\}$ is a k-bound set.

One possibility to check whether a set of variables Y is a bound set is to move the variables Y to the top of the ROBDD and check the number of children below $cut(|Y|)$, as in [99, 41]. However, re-ordering is computationally expensive. Instead, we have developed a procedure, called **Generalized-IntervalCut** which checks whether a given linear interval of variables of a ROBDD is a k-bound set without reordering of variables. **GeneralizedIntervalCut** is an extension of **IntervalCut** algorithm introduced in [13] for simple disjoint decomposition. To describe the procedure, we first present some definitions.

Suppose the variables Y lie between two cuts, $cut(a)$ and $cut(b)$, such that $a < b$, $a, b \in \{0,\ldots,n\}$. Let $cut\_set(a)$ denote a set of nodes $v \in G$ with indexes $a < index(v) \leq b$ which are children of the nodes above the $cut(a)$ of G. Let G_v stand for a ROBDD rooted at some $v \in cut\_set(a)$. Then,

GeneralizedIntervalCut(G, k, a, b)
input: ROBDD G of $f(X)$, $k \geq 1$, two cuts $cut(a)$ and $cut(b)$, $a < b$, $a, b \in \{0, \ldots, n\}$.
output: "not a bound set" if the set of variables Y between $cut(a)$ and $cut(b)$ is not a k-bound set of $f(X)$; functions g and h if Y is a k-bound set resulting in $f(X) = h(g(Y), Z)$.

for all $v \in cut\_set(a)$
  if ($|cut\_set(b_v)| < 2$ **or** $|cut\_set(b_v)| > 2^k$)
    return("not a k-bound set");
for all $v_1, v_2, \ldots, v_k \in cut\_set(a)$
  if ($g_{v_i} \neq g_{v_{i+1}}$) /* up to isomorphism */
    return("not a bound set");
h = substitute each subgraph g_v, $\forall v \in cut\_set(a)$, by a variable node;
$g = g_v$;
return(g, h);

Figure B.1: Pseudo code of the **GeneralizedIntervalCut** procedure.

$cut\_set(b_v)$ is the set of nodes $u \in G_v$ with indexes $b < index(u) \leq n + 1$ which are children of the nodes of G_v above the $cut(b)$.

If $|cut\_set(b_v)| = m$, $2 \leq m \leq 2^k$, then g_v is a m-valued function represented by the subgraph rooted at v whose terminal nodes are obtained by replacing the m nodes of $cut\_set(b_v)$ by constants $\{0, 1, \ldots, m-1\}$. The resulting g_v is unique up to isomorphism[1]. Using this notation, we can describe the pseudo code of the algorithm **GeneralizedIntervalCut**(G, k, a, b) as shown in Figure B.3.

B.4 Experimental results

We implemented the presented heuristic and applied it to a large set of benchmarks. Only some representative results are shown in Table B.1, for space limitation reasons. The first three columns show information about the benchmarks: their name, the number of primary inputs and the number primary outputs. Columns 4 to 9 show the number N of k-bound sets found for different values of k, and the time spend by our algorithm to find

[1]Two functions are *isomorphic* if, and only if, their ROBDD representations are graph isomorphic up to the constant nodes; i.e. if, and only if, there exists a bijection $\phi : \{0, \ldots, m-1\} \rightarrow \{0, \ldots, m-1\}$ such that $f(x) = \phi(g(x))$

Table B.1: Experimental results; time is reported in seconds and includes ROBDD building and minimization times. The case when $k = 1$ represents classical (Boolean) bound sets, as defined in Section B.1.

benchmarks			bound sets					
			$k = 1$		$k = 2$		$k = 3$	
name	in	out	N	t (sec)	N	t (sec)	N	t (sec)
9symml	9	1	0	0,03	5	0,03	12	0,02
C1355	41	32	0	213,5	32	214,08	62	213,13
C1908	33	25	668	32,4	758	32,45	857	32,54
C3540	50	22	2993	322,24	3039	322,78	2989	324,01
C432	36	7	342	3,65	448	3,65	466	3,67
C499	41	32	0	213,01	32	212,91	62	212,73
C880	60	26	14332	73,09	14246	74,32	14305	74,86
alu2	10	6	55	0,08	65	0,08	72	0,08
alu4	14	8	145	0,37	177	0,4	195	0,37
apex1	45	45	5633	14,55	6028	14,71	6135	14,77
apex2	39	3	6	3,8	31	3,84	68	3,85
apex4	9	19	4	0,28	27	0,27	44	0,28
apex5	117	88	25286	52,88	45349	53,48	62147	53,45
apex6	135	99	147837	64,04	242510	64,76	244018	64,79
apex7	49	37	8764	3,34	11744	3,37	11730	3,35
b9	41	21	2616	1,11	4336	1,13	4858	1,12
cm150a	21	1	1	2,1	3	2,08	9	2,13
cm42a	4	10	7	0,01	7	0,02	7	0,01
cm85a	11	3	27	0,05	56	0,05	62	0,05
cmb	16	4	285	0,06	285	0,06	285	0,05
comp	32	3	147	29,79	267	30,08	266	29,86
count	35	16	734	0,62	2642	0,66	2667	0,65
des	256	245	253580	1000,71	261548	1001,94	257766	1002,46
e64	65	65	47447	12,42	46195	12,23	44883	12,14
f51m	8	8	54	0,03	83	0,03	102	0,05
frg2	143	139	91905	103,15	119457	103,97	120067	103,78
lal	26	19	1514	0,4	2433	0,38	2433	0,39
misex2	25	18	1880	0,38	2259	0,38	2272	0,39
mux	21	1	1	2,01	3	2	9	2,01
pair	173	137	160887	486,98	222231	495,93	276550	503,61
parity	16	1	104	0,04	104	0,04	104	0,04
rot	135	107	200868	493,82	246026	501,45	254197	508,59
seq	41	35	1045	14,44	1735	14,6	2870	14,69
term1	34	10	677	0,58	942	0,62	1368	0,62
too_large	38	3	3	2,62	19	2,62	41	2,62
ttt2	24	21	933	0,4	1565	0,4	1599	0,41
vda	17	39	427	0,84	502	0,86	635	0,89
x3	135	99	151793	64,1	242510	64,76	244018	64,8
x4	94	71	35949	20,16	44939	20,28	45201	20,24

them. The timings include ROBDD building and minimization[2] times, and are expressed in seconds. All the experiments were run on a Sun Ultra 60 operating with two 360 MHz CPU and with 1024 MB RAM main storage.

B.5 Conclusions

We have presented a practical heuristic algorithm that quickly finds many, although not all, Roth-Karp decompositions. The algorithm works on a ROBDD representation of the function to be decomposed, without the usual reordering overhead of other *cut*-based methods. This paper reflects results from ongoing work, and the preliminary implementation performance shown in Section B.4 can be further improved.

Future work includes an extension of the algorithm to non-disjoint decomposition where $Y \cap Z \neq \emptyset$. We are also investigating a possibility of combining **GeneralizedIntervalCut** with decomposition algorithms exploiting the structure of BDDs, like [113].

Acknowledgment

This work was supported in part by IBM Partnership Award.

[2]In order to make the implementation more efficient, we use a fast "sifting" [144] algorithm to make the ROBDD size smaller.

Paper C

Disjoint-Support Boolean Decomposition Combining Functional and Structural Methods

Andrés Martinelli, René Krenz and Elena Dubrova. Published in the "Proceedings of the IEEE Asia and South Pacific Design Automation Conference 2004" (ASP-DAC 2004), January, 2004, Yokohama, Japan, pp. 597–599.

Disjoint-Support Boolean Decomposition Combining Functional and Structural Methods

Andrés Martinelli* René Krenz* Elena Dubrova*

Abstract

This paper presents an algorithm for disjoint-support decomposition of Boolean functions which combines functional and structural approaches. First, a set of proper cut points is identified in the circuit by using dominator relations (structural method). Then, the circuit is partitioned along these cut points and a BDD-based decomposition is applied to the resulting smaller functions (functional method). Previous work on Boolean decomposition used only single methods and did not integrate a combined strategy. The experimental results show that the presented technique is more robust than a pure BDD-based approach and produces better-quality decompositions.

C.1 Introduction

Boolean decomposition is a technique used in many applications, including multi-level logic synthesis [41, 132], testing [137, 17], formal verification [56], combinatorial optimization problems over graphs and networks [121].

In general terms, the problem of decomposition of functions can be formulated as follows: Given a function f, express it as a composite function of some set of new functions. Often, a composite expression can be found in which the new functions are significantly simpler than f.

*Royal Institute of Technology, IMIT/KTH, Stockholm, Sweden

The basic type of decomposition is the *simple disjoint* decomposition where a function $f(X)$ is expressed as a composite function of two functions g and h, namely

$$f(X) = h(g(Y), Z) \tag{C.1}$$

where Y and Z are sets of variables forming a partition of the set of variables $X = \{x_1, x_2, \ldots, x_n\}$ of f. Every set of variables X for which a decomposition like (C.1) exists is called a *bound set* for f.

The fraction of all Boolean functions of n variables possessing simple disjoint decompositions of type (C.1) approaches zero as n approaches infinity [138, p. 90]. Therefore, a more general type of decomposition, known as *disjoint-support* (or *Roth-Karp* [90]) decomposition is usually considered. Disjoint-support decomposition has the form

$$f(X) = h(g(Y), Z)$$

with $f : \{0,1\}^n \to \{0,1\}$, $g : \{0,1\}^{|Y|} \to \{0,1,...,m-1\}$ and $h : \{0,1,...,m-1\} \times \{0,1\}^{|Z|} \to \{0,1\}$. The m-valued function h can be encoded by $k = \lceil \log_2 m \rceil$ Boolean functions $g_1, g_2, \ldots, g_k$, giving a representation of the form

$$f(X) = h(g_1(Y), g_2(Y), \ldots, g_k(Y), Z) \tag{C.2}$$

with all functions being Boolean. Methods for choosing good encodings where presented in [102, 82]. Disjoint-support decomposition includes as a special case non-disjoint decomposition. For example, if V is the set of overlapping variables of h and g_1, then the non-disjoint decomposition $f(X) = h(g_1(Y, V), V, Z)$ can be treated as a disjoint-support decomposition $f(X) = h(g_1(Y, V), g_2(V), Z)$, with g_2 being the identity function.

It is possible to extend algorithms for simple disjoint decomposition to the disjoint-support case. For example, [111] presents such an extension of the algorithm proposed in [13]. It is a BDD-based heuristic algorithm which quickly finds many disjoint-support decompositions and can handle large functions. One problem with this approach is that the decompositions found in this way do not necessarily simplify the function. For example, a circuit implemented as the two cofactors of a Shannon decomposition joined by a multiplexor is usually not optimal. Shannon decomposition is a special case of the decomposition (C.2), with $Z = x_1$, $g_1(Y) = f(0, x_2, \ldots, x_n)$, $g_2(Y) = f(1, x_2, \ldots, x_n)$, and $h = x_1' g_1(Y) + x_1 g_2(Y)$.

Another problem is that, in contrast to the case of simple disjoint decompositions, that are "too few", disjoint-support decompositions are "too

many". So, an algorithm which first generates all disjoint-support decompositions and then checks which of them simplify the function is not feasible.

Our approach to overcome these problems is the following. First, a set of *proper cut* points is identified in a circuit representation of the function by applying a structural decomposition method. The circuit is partitioned along these cut points into a set of smaller sub-circuits which are treated independently. This allows us to reduce the search space for disjoint-support decompositions at the next stage, since all bound sets which overlap proper cut partitions are pruned. Finally, the overall decomposition is determined by combining the intermediate results.

We have chosen to use at the first stage of our algorithm a circuit-based technique rather than a BDD-based one, because manipulating circuits is much faster. Therefore, for functions with no proper cuts, the presented technique does not bring a significant overhead. The running time of our algorithm is normally similar, or even faster, than the running time of a BDD-based algorithm.

The rest of the paper is organized as follows. Section C.2 reviews previous work. Section C.3 summarizes the notation. Section C.4 presents the first phase of our algorithm (circuit-based method). Section C.5 presents the second phase of our algorithm (BDD-based method). Section C.6 shows the experimental results. Section C.7 concludes the paper and discusses some open problems.

C.2 Previous work

The concept of *proper cuts* was first introduced in combinational equivalence checking [56]. Later it was applied to testing [137, 17] and design for low power [43] where it is known under the alternative names of *headline* or *supergate*. A vertex v is a proper cut if every path from any primary input in the cone of influence of v to the root contains v. The presented algorithm for finding proper cuts is based on the concept of *reduced dominator tree* constructed by using an extension of the Lengauer-Tarjan algorithm [103] for finding dominators in a graph. A proper cut is required to dominate all the primary inputs in its cone of influence. This guarantees that all re-converging paths are completely enclosed within the cone and, therefore, that those primary inputs belong to a bound set.

Disjoint-support decomposition was introduced by Roth and Karp [130]. They defined the notion of compatible classes describing the conditions for the existence of bound sets. Two assignments $x_1, x_2 \in B^{|Y|}$ are said to be *compatible with respect to the reference function* $f(Y, Z)$ if, for all $y \in B^{|Z|}$ such that $f(x_1, y)$ and $f(x_2, y)$ are defined, $f(x_1, y) = f(x_2, y)$ [90]. The set Y is a bound set if and only if $B^{|Y|}$ can be partitioned into $k \leq 2$ mutually compatible classes [90]. If $f(X)$ is completely specified, then compatibility is an equivalence relation and k is the number of equivalence classes.

A number of BDD-based decomposition algorithms have been developed. Karplus [91] presented a technique for AND- and OR-type decomposition based on dominators in BDDs. It was extended by Yang et al [161] to XOR-type decompositions. Stanion and Sechen [146] target *quasi-algebraic* decomposition of the form $f(X) = g(Y) \odot h(Z)$, where "$\odot$" is any binary Boolean operation and $|Y \cup Z| = k$ for some $k \geq 0$. This type of decomposition is often referred to as *bi-decomposition* [132, 159, 119, 45]. Bengtsson [13] developed a fast heuristic for simple disjoint decomposition which iteratively examines all linear intervals of variables of a ROBDD, and for every interval checks whether it is a bound set. This algorithm has been extended to disjoint-support decompositions in [111]. Minato and De Micheli [118] presented an algorithm which computes simple disjoint decompositions by generating irreducible sum-of-product for the function from its BDD and applying factorization. The algorithm of Bertacco and Damiani [15] makes a single traversal of the BDD to identify the simple disjoint decomposition of the co-factors and then combine them to obtain the decomposition for the entire function. The algorithm is impressively fast; however, as Sasao has observed in [133], it fails to compute some of the disjoint decompositions. This problem was corrected by Matsunaga [113], who added the missing cases in [15] allowing to treat the OR/XOR functions correctly. The algorithm [113] appears to be the fastest of existing exact algorithms for finding all simple disjoint decompositions.

C.3 Preliminaries

In this section we summarize the basic notation and definitions used in the sequel.

Let $C = (V, E, root)$ denote a single-output circuit, where V represents a set of gates and primary inputs. A particular vertex $root \in V$ is marked

as the circuit output. The set of edges $E \subseteq V \times V$ represents the nets connecting the gates. Each edge $(u, v) \in E$ is associated with an inverter attribute $i(u, v) \in \{0, 1\}$ where $i = 1$ or $i = 0$ indicates whether the edge function is to be complemented or not, respectively.

A vertex v *dominates* another vertex $w \neq v$ in C if every path from w to *root* contains v. Vertex v is the *immediate dominator* of w, denoted $v = idom(w)$, if v dominates w and every other dominator of w dominates v. Every vertex v in C except *root* has a unique immediate dominator [108].

The edges $\{(idom(w), w) \mid w \in V - \{root\}\}$ form a directed tree D rooted at *root*, which is called the *dominator tree* of C. The dominator children $Doms(v) \subset V$ of vertex v are the set of vertices having v as immediate dominator, i.e., $Doms(v) = \{u \mid idom(u) = v\}$.

A *reduced* dominator tree [95] D_R contains all vertices $v \subseteq D$ such that:

1. v is a primary input or
2. $\exists u \in D_R$ such that $v = idom(u)$.

The bound sets used in disjoint-support decomposition are a more general case of the notion of classical bound sets, as described in Section C.1.

Definition C.3.1*:* The set Y is said to be a k-bound set if there exists a decomposition

$$f(X) = h(g(Y), Z) \tag{C.3}$$

for some functions g, h of type $g : \{0, 1\}^{|Y|} \rightarrow \{0, 1, \ldots, m-1\}$ and $h : \{0, 1, \ldots, m-1\} \times \{0, 1\}^{|Z|} \rightarrow \{0, 1\}$, such that $2 \leq m < 2^k$.

These bound sets can be determined by decomposition chart or cut methods, by relaxing the requirement of having exactly 2 different columns (or 2 different cut nodes), to allow a number of columns (or cut nodes) up to 2^k [90].

Let V be a set of nodes of a ROBDD G of an n-variable function $f(X)$. Every non-terminal node $v \in V$ has an associated variable index, $index(v) \in \{1, \ldots, n\}$. The index of the root node is 1, and we let the terminal nodes have also an index, which is $n + 1$.

Suppose that all nodes with $index \leq i$ are in the upper part of the graph and all nodes with $index > i$ are in the lower part of the graph, for some $i \in \{1, \ldots, n\}$. The boundary line between the upper and lower parts of the graph is called $cut(i)$.

If the number of nodes with $index > i$ which are children of the nodes above the $cut(i)$ is at most 2^k, for a given k, then, by Definition C.3.1, the set of variables $Y = \{x_1, \ldots, x_i\}$ is a k-bound set.

C.4 Circuit-based proper cut decomposition

Let C_v denote the cone of influence of v, i.e. a sub-graph of C including all the vertices from which v is reachable by a directed path.

Definition C.4.1*:* A vertex is a proper cut if it dominates all primary input vertices in its cone of influence.

This guarantees that all re-converging paths are completely enclosed within the cone and, therefore, that those primary inputs form a bound set. For all non-primary input vertices w, there exists at least one primary input vertex u from which w is reachable by a directed path. Therefore, $v = dom(u)$ implies $v = dom(w)$ for all $w \in C_v$. The primary input vertices and the root vertex are *trivial* proper cuts, i.e. they always exists.

It is easy to prove prove that a proper cut is always a vertex of the reduced dominator tree.

Lemma 2. *A vertex $v \in V$ is a proper cut only if $v \in D_R$.*

The pseudo-code of the algorithm ProperCut which uses a reduced dominator tree to identify the set of proper cuts P is shown in Figure C.1 [95]. We use Lengauer-Tarjan algorithm [103] is used for finding dominators. It is efficient for large circuits.

ProperCut processes the circuit from the inputs toward the output in topological order. The array $T[v]$ contains vertices $u \in D_R$ with open re-convergences. At the primary inputs, $T[v]$ is initialized to an empty set. Then, at each following vertex v, $T[v]$ is updated to the union of $T[v_i]$ for all vertices v_i in its fan-in. If v is in the reduced dominator tree, then the set $Doms(v)$ of vertices having v as an immediate dominator is removed from $T(v)$ and, after performing the proper cut checking, v is added to $T[v]$. This substitution of $Doms(v)$ vertices by their dominator allows us to keep the size of $T[v]$ small and, what is more important, lets us keep the support-set of $T[v]$ dependent on vertices having v as an immediate dominator only, rather than vertices on previous topological levels.

```
algorithm ProperCut(V, E, root);
  D_R, Doms = Dominator(V, E, root)
  for each v ∈ V in topological order do
      if v ∈ Inputs then
          T[v] = ∅;
      else
          T[v] = ⋃_{v_i ∈ FI(v)} T[v_i];
      if v ∈ D_R then
          T[v] = T[v] − Doms(v);
          if T[v] = ∅ then
              P = P ∪ {v};
          T[v] = T[v] ∪ {v};
  return P
end
```

Figure C.1: Pseudo-code of the algorithm ProperCut.

C.5 BDD-based decomposition

After the set of proper cut points is identified, the circuit is partitioned along these cut points into a set of smaller sub-circuits which are processed independently using the BDD-based decomposition technique similar to [111].

The algorithm successively goes through all possible linear intervals of variables of a BDD and, for each interval, checks whether it is a bound set or not. In this way many decomposition are found quickly, without expensive variable re-ordering.

Suppose the variables Y lie between two cuts, $cut(a)$ and $cut(b)$, such that $a < b$, $a, b \in \{0, \ldots, n\}$. Let $cut\_set(a)$ denote a set of nodes $v \in G$ with indexes $a < index(v) \leq b$ which are children of the nodes above the $cut(a)$ of G. Let G_v stand for a ROBDD rooted at some $v \in cut\_set(a)$. Then, $cut\_set(b_v)$ is the set of nodes $u \in G_v$ with indexes $b < index(u) \leq n + 1$ which are children of the nodes of G_v above the $cut(b)$.

If $|cut\_set(b_v)| = m$, $2 \leq m \leq 2^k$, then g_v is a m-valued function represented by the sub-graph rooted at v whose terminal nodes are obtained by replacing the m nodes of $cut\_set(b_v)$ by constants $\{0, 1, \ldots, m-1\}$. The

resulting g_v is unique up to isomorphism[1]. Using this notation, the pseudo code of the algorithm GeneralizedIntervalCut(G, k, a, b) is described in Figure C.2.

algorithm GeneralizedIntervalCut(G, k, a, b)
  for each $v \in cut\_set(a)$
    if $(|cut\_set(b_v)| < 2$ **or** $|cut\_set(b_v)| > 2^k)$
      return ("not a k-bound set");
  for each $v_1, v_2, \ldots, v_k \in cut\_set(a)$
    if $(g_{v_i} \neq g_{v_{i+1}})$ /* up to isomorphism */
      return ("not a bound set");
  h = substitute each sub-graph g_v, $\forall v \in cut\_set(a)$,
    by a variable node;
  $g = g_v$;
  return (g, h);
end

Figure C.2: Pseudo-code of the GeneralizedIntervalCut algorithm.

C.6 Experimental results

All experiments were performed on a PC with a 2GHz Pentium4 CPU and 1024MByte main memory, running Linux Mandrake 8.2. We used a set of 188 combinational circuits from IWLS'02 benchmark set which comprises a total of 17633 outputs.

Figure C.3 shows a comparison of the running times of the pure BDD-based approach against the combined one. Each cross in the figure represents a single output function. Those above the line mark an improvement in the running time. As one can see, in the majority of cases, the combined approach is faster. Crosses below the line, representing cases where the running time of the combined tool is slower, are primarily circuits with no simple disjoint decomposition (i.e. no proper cuts), where the time spent

[1] Two functions are *isomorphic* if, and only if, their ROBDD representations are graph isomorphic up to the constant nodes; i.e. if, and only if, there exists a bijection $\phi : \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$ such that $f(x) = \phi(g(x))$.

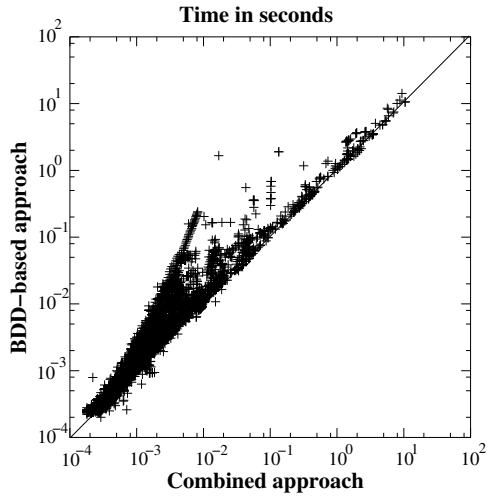


Figure C.3: Runtime comparison for the combined versus BDD-based approaches.

on circuit exploration simply adds up as an overhead on the BDD-based algorithm.

Some representative results, aiming to show the number of disjoint-support decompositions computed by the combined approach, are given in Table C.1. The first three columns show information about the benchmarks: their name, the number of primary inputs and the number of primary outputs. Column 4 shows the number of proper cuts found in the first phase of the algorithm. Columns 5 to 7 show the number of k-bound sets found in the second phase, for different values of k, as the total sum of the results for individual outputs.

Notice that although columns 4 and 5 both show simple disjoint decompositions, the results they report do not overlap and should be considered separately. They respectively represent those decompositions found during the structural and the BDD-based phases of the algorithm, respectively. Since the heuristics used in each phase may not find all bound sets, and since they are dependent on the structure of the circuit and BDD ordering, the combination of the two can result in one finding bound sets which cannot be found by the other.

Also notice the cases like `cm42a`, `decod` or `parity`: in these, only zeroes are reported for the second phase of the algorithm. This is because after the partitioning along the cut points found in the first phase, the resulting functions only contain trivial disjoint-support decompositions, so the BDD-based algorithm is not invoked at all. This is one the reasons for the running time improvement.

C.7 Conclusion

We present a decomposition technique which integrates circuit-based and BDD-based decompositions. The combination of the two approaches results in an algorithm which is more robust than the pure BDD-based method, regarding both, quality of the result and running time.

Our experiments on benchmark circuits suggest that the developed algorithm has a significant potential for a large number of circuits. However, there are also limitations. The main one is that our method depends on dominator relations of the circuit. If the circuit under consideration has no internal dominators, the presented technique reduces to a BDD-based decomposition. We have found that the majority of practical circuit graphs contain a substantial number of internal dominator vertices (between 5 and 0.5 per input) which warrants an efficient performance of our algorithm. For circuits with no internal dominators, in the future we plan to use complementary methods for structuring the decomposition process, such as generalized dominators [77] and min-cut [44].

benchmarks			bound sets			
			classical		Roth-Karp	
name	in	out	proper cuts	k=1	k=2	k=3
9symml	9	1	0	0	10	17
alu2	10	6	1	3	55	89
alu4	14	8	0	2	141	263
apex2	39	3	0	9	57	119
apex6	135	99	229	9056	32520	36084
apex7	49	37	104	2814	8406	9435
b9	41	21	37	335	1320	1465
C1355	41	32	0	0	11624	21708
C1908	33	25	0	2758	5337	8279
C3540	50	22	18	79	676	1378
C432	36	7	16	0	97	259
C499	41	32	0	0	12404	22428
C880	60	26	57	204	1574	3150
cm150a	21	1	1	0	5	11
cm42a	4	10	20	0	0	0
cm85a	11	3	9	20	90	100
cmb	16	4	20	325	325	325
comp	32	3	7	108	520	696
cordic	25	2	0	18	71	95
count	35	16	136	136	544	544
decod	5	16	48	0	0	0
des	256	245	640	30527	202664	327911
e64	65	65	2016	0	0	0
f51m	8	8	0	26	85	123
frg2	143	139	1	14	29	31
lal	26	19	598	32927	142542	169797
misex2	25	18	50	237	1522	1550
mux	21	1	1	0	5	11
pair	173	137	889	28416	113431	173717
parity	16	1	14	0	0	0
rot	135	107	177	9159	45070	65873
seq	41	35	34	3951	15762	28193
term1	34	10	26	340	822	870
too_large	38	3	0	7	43	109
ttt2	24	21	10	843	2583	2665
x3	135	99	129	11980	31663	36259
x4	94	71	66	11066	30591	34785

Table C.1: Experimental results. Notice that 'proper cuts' and disjoint-support case 'k=1' represent different simple disjoint decompositions, found in the first and the second phase respectively, and should be counted separately.

Paper D

On the Relation Between Non-Disjoint Decomposition and Multiple-Vertex Dominators

Elena Dubrova, Maxim Teslenko and Andrés Martinelli. Published in the "Proceedings of the IEEE International Symposium on Circuits and Systems 2004" (ISCAS 2004), May 23-26, 2004, Vancouver, Canada, pp. 493–496.

On Relation Between Non-Disjoint Decomposition and Multiple-Vertex Dominators

Elena Dubrova[*,†] Maxim Teslenko[*] Andrés Martinelli[*]

Abstract

This paper addresses the problem of non-disjoint decomposition of Boolean functions. Decomposition has multiple applications in logic synthesis, testing and formal verification. First, we show that the problem of computing non-disjoint decompositions of Boolean functions can be reduced to the problem of finding multiple-vertex dominators of circuit graphs. Then, we prove that there exists an algorithm for computing all multiple-vertex dominators of a fixed size in polynomial time. Our result is important because no polynomial-time algorithm for non-disjoint decomposition of Boolean functions is known. A set of experiments on benchmark circuits illustrates our approach.

D.1 Introduction

Non-disjoint decomposition of a Boolean function f is a representation of type

$$f(X, Y, Z) = h(g_1(X, Y), \ldots, g_k(X, Y), Y, Z) \tag{D.1}$$

where X, Y, Z are sets of variables partitioning the support set of f, and h and g_i are Boolean functions, $i \in \{1, \ldots, k\}$. Applications of decomposition include multi-level logic optimization [136, 161], FPGA technology mapping [132, 41, 134], testing [137], and formal verification [97].

[*]{elena, maxim, andres}@imit.kth.se, Royal Institute of Technology, IMIT/KTH, Stockholm, Sweden

[†]This work was supported in part by the Research Grant No 6426 from the Swedish Research Council Vetenskpsrådet.

The problem of computing non-disjoint decomposition is hard. No algorithm for computing all possible non-disjoint decompositions of a Boolean function in polynomial-time is known. Binary Decision Diagram (BDD) based decomposition algorithms show a good average-time performance [161, 99, 111]. However, these approaches are limited by the excessive memory consumption of decision diagrams.

This paper has two main contributions. First, we show that the problem of computing non-disjoint decompositions of Boolean functions is related to the problem of finding multiple-vertex dominators of circuit graphs. A circuit graph is a common format for representing Boolean functions. Most practical functions have small circuit representations. Second, we prove that there exists a $O(n^k \log n)$ algorithm for computing all multiple-vertex dominators of a fixed size k, where n is the number of vertices of the circuit graph.

The presented approach allows us to compute all non-disjoint decompositions which are reflected in the circuit structure. However, these may not be all possible decompositions of the function. For example, if a function is represented by a circuit implementing $f = a(b + c)$, then the disjoint decomposition $h = a \cdot g$, $g = b + c$ will be identified. However, if the function is represented by the circuit realizing $f = ab + ac$, then no disjoint decomposition will be found.

The paper is organized as follows. Section D.2 describes previous work. Section D.3 shows a relation between non-disjoint decomposition and multiple-vertex dominators. The existence of a polynomial algorithm for computing multiple-vertex dominators is proved in Section D.4. Section D.5 summarizes the experimental results. Conclusion and future work are given in Section D.6.

D.2 Previous work

Non-disjoint decomposition of Boolean functions was pioneered by Curtis [48] in 1962. Curtis has shown that a Boolean function possesses a simple non-disjoint decomposition of type $f(X, Y, Z) = h(g(X, Y), Y, Z)$ if each of its $2^{|Y|}$ decomposition charts representing sub-functions $f_Y(X, Z)$ has at most two distinct columns. The decomposition chart for $f_Y(X, Z)$ is a two-dimensional table where the columns describe all combinations of the variables from the set X and the rows list all combinations of the variables

from the set Z. $2^{|Y|}$ charts are obtained by fixing the variables of Y to all combination of their values from $\{0,1\}^n$.

Roth and Karp [130] have extended simple non-disjoint decomposition to a more general type given by equation (D.1). They used *cube* representation and reduced the problem of computing column multiplicity to the problem of computing compatible classes for a set of cubes.

Due to the exponential size of decomposition charts and cube representations, early decomposition algorithms were not applicable to large functions. Instead, *algebraic* decomposition methods were used in practice. A milestone work is [33], where the notion of *kernels* is introduced and a method for fast algebraic decomposition based on kernels is developed. This technique, with minor modifications, is used in many systems for multi-level optimization [29, 112, 136].

BDDs made possible developing algorithms for Boolean decomposition, feasible for much larger functions than previously possible. In a BDD, the column multiplicity can be computed by moving the variables X to the upper part of the graph and checking the number of children below the boundary line, called *cut* line. This approach has been adopted by a number of BDD-based decomposition algorithms [99, 132, 111]. Stanion and Sechen [146] used cut to find *quasi-algebraic* decomposition of the form $f(X,Y,Z) = g(X,Y) \odot h(Y,Z)$, where "$\odot$" is an arbitrary Boolean binary operation. This type decomposition is often referred to as *bi-decomposition* [159, 119].

Another group of decomposition methods exploit the structure of BDDs, rather than cut. Karplus [91] extended the classical concept of dominator on graphs to $0,1-$ dominators on BDDs. A node v is a 1-dominator (0-dominator) if every path from the root to one (zero) terminal node contains v. This idea was extended by Yang et al [161] to XOR-type decompositions. Minato and De Micheli [118] presented an algorithm which computes all disjoint decompositions by generating irreducible sum-of-product for the function from its BDD and applying factorization. Algorithms [15] and [113] makes a single traversal of the BDD to identify the disjoint decomposition of the co-factors and then combine them to obtain all disjoint decomposition for the entire function.

D.3 Relation between non-disjoint decomposition and multiple-vertex dominators

In this section, we present a fundamental theorem showing the relation between non-disjoint decomposition of Boolean functions and multiple-vertex dominators of circuit graphs. We start with the definitions used in the sequel.

Let $C = (V, E)$ denote a single-output directed acyclic circuit graph, where V represents a set of gates and primary inputs. A particular vertex $root \in V$ is marked as the circuit output. The set of edges $E \subseteq V \times V$ describes the nets connecting the gates.

The *cone of influence* of a vertex v, $I(v)$, is a subset of V containing all the vertices from which v is reachable.

A vertex v *dominates* another vertex w in V if every path from w to $root$ contains v [103]. We denote by $D(v)$ the set of vertices dominated by v. Vertex v is the *immediate dominator* of w, denoted by $v = idom(w)$, if v dominates w and every other dominator of w dominates v. Every vertex $v \in V$ except $root$ has a unique immediate dominator [108]. The edges $\{(idom(w), w) \mid w \in V - \{root\}\}$ form a directed tree rooted at $root$, which is called the *dominator tree* of C.

Many graphs do not contain any single-vertex dominators except primary inputs and $root$. It is more common that a vertex is dominated by a set of vertices.

A set of vertices $\{v_1, \ldots, v_k\}$ is a *multiple-vertex* dominator of size k [5] (also called *generalized* dominator [77]) for a vertex u, if (1) every path from u to $root$ contains some v_i, and (2) for every v_i, there exist at least one path from u to $root$ which contains v_i and does not contain any other v_j, $i, j \in \{1, \ldots, k\}$, $i \neq j$.

A set of vertices $\{v_1, \ldots, v_k\}$ is a *common multiple-vertex dominator* for a set of vertices $U \subseteq V - \{v_1, \ldots, v_k\}$, if, for every $u \in U$, there exist $W \subseteq \{v_1, \ldots, v_k\}$ such that W is a multiple-vertex dominator for u.

Let X, Y, Z be sets of variables partitioning the support set of a Boolean function f.

Theorem 9. *Suppose a Boolean function $f(X, Y, Z)$ is represented by a circuit graph $C = (V, E)$. Let $V_X, V_Y, V_Z \subset V$ be sets of primary input vertices corresponding to the variables of the sets X, Y, Z. Let $v_{g_1}, \ldots, v_{g_k} \in V$ be a set of vertices such that:*

1. $\{v_{g_1}, \ldots, v_{g_k}\}$ *is a common multiple-vertex dominator for* V_X,
2. $(V_X \cup V_Y) \subset \bigcup_{i=1}^{k} I(v_{g_i})$.

Then, there exist a decomposition of f *of type*

$$f(X, Y, Z) = h(g_1(X, Y), \ldots, g_k(X, Y), Y, Z)$$

where Boolean functions g_i *are the functions rooted by the vertices* v_{g_i}, $\forall i \in \{1, \ldots, k\}$, *of* C.

Theorem 9 allows us to reduce the problem of computing non-disjoint decompositions to the problem of computing multiple-vertex dominators. This result is important because no polynomial-time algorithm for computing all non-disjoint decompositions of a Boolean function is known. In the next section, we show that the problem of computing all multiple-vertex dominators of a fixed size can be solved in polynomial-time.

D.4 Computing all multiple-vertex dominators of a fixed size in polynomial time

It is possible to compute all single-vertex dominators for a directed graph in time less than quadratic in the number of vertices. For example, a well-known Lengauer-Tarjan algorithm [103] has the worst-case complexity $O(n \cdot \log n)$. However, algorithms for computing all multiple-vertex dominators for a directed graph have exponential worst case complexity [77]. A subset of immediate multiple-vertex dominators can be computed in $O(n^2)$ time [5], but immediate dominators are not particularly interesting from the decomposition point of view. Good decompositions require multiple-vertex dominators of a small size k which are common for large sets V_X. In this section, we show that it is possible to compute multiple-vertex dominators of a fixed size in polynomial time.

Let $C = (V, E)$ be a circuit graph with $|V| = n$ vertices.

Theorem 10. *If there exists an* $O(\tau(n))$ *algorithm for computing all single-vertex dominators, then there exists an* $O(n^{k-1}\tau(n))$ *algorithm for computing all multiple-vertex dominators of size* k.

Proof. Assume there exists an $O(\tau(n))$ algorithm for computing all single-vertex dominators. Let $T(C)$ denote the dominator tree of a circuit-graph

$C = (V, E, root)$, and let $M(C)$ denote the set of all possible multiple-vertex dominators of size k. To compute $M(C)$, we do the following:

1. Compute $T(C)$.
2. For each $\{v_1, \ldots, v_{k-1}\} \in V^{k-1}$ do Steps 3 to 6
3. Mark as "non-existing" all edges in C such that $E' = E - \{(u, w) | u \in \bigcup_{i=1}^{k-1} D(v_i) \vee w \in \bigcup_{i=1}^{k-1} D(v_i)\}$.
4. Compute $T(C')$ for the resulting modified graph C'.
5. Compute $M(C)$ by checking the following condition $\forall u_j \in T(C')$: If u_j is a single-vertex dominator for some $w \in \bigcup_{i=1}^{k-1} I(v_i) - \bigcup_{i=1}^{k-1} D(v_i)$ in C', then $\{u_j, v_1, .., v_{k-1}\}$ is a multiple-vertex dominator of size k for w in C, if u_j does not dominate any of v_i in C, $i \in \{1, \ldots, k-1\}$.
6. Undo Step 3.

Steps 1 and 4 are $O(\tau(n))$. Steps 3, 5 and 6 can be done in $O(n)$ time. The correctness of Step 5 follows directly from the definition of multiple-vertex dominator. The overall complexity is

$$O(\tau(n)) + n^{k-1}(max(O(\tau(n)), O(n))) = O(n^{k-1}\tau(n)).$$

□

If Lengauer-Tarjan algorithm [103] is used for computing single-vertex dominators, then $M(C)$ can be obtained in $O(n^k \log n)$ time. Clearly, the simple algorithm constructed in the proof will not be feasible for large circuit graphs if $k > 2$. However, as we show in the next sections, for small k, ever this straightforward approach gives good results. Many practical applications of decomposition (multi-level logic synthesis [136, 161], FPGA technology mapping [132, 41], etc.) require only small values of k.

D.5 Experimental results

This section illustrates the performance of the algorithm constructed in the proof of the Theorem 10 for IWLS'02 benchmark set.

Column 5 of Table D.1 shows the number, N_{2dec}, of computed decompositions $f(X, Y, Z) = h(g_1(X, Y), g_2(X, Y), Y, Z)$ with $k = 2$. Note, that N_{2dec} does not include decompositions with $k = 1$. Only interesting cases, where a 2-vertex dominator dominates at least 3 inputs ($|X| > 2$) are counted. Every output is treated as a separate function. The numbers shown in Column

Table D.1: Benchmark results.

name	*in*	*out*	*gates*	N_{2dec}	t (sec)
apex5	114	88	3781	2609	2.34
apex6	135	99	801	639	0.11
b9	41	21	166	50	0.02
comp	32	3	158	103	0.07
count	35	16	163	15	0.01
C1355	41	32	546	1032	3.98
C1908	33	25	448	350	1.63
C2670	233	140	951	261	1.26
C3540	50	22	1089	345	8.42
C432	36	7	247	169	0.23
C499	41	32	442	1000	2.53
C5315	178	123	1952	4205	5.57
C6288	32	32	2370	153	70.23
C7552	207	108	2282	12816	7.73
C880	60	26	338	253	1.15
des	256	245	4733	1134	5.24
frg2	143	139	2011	1389	0.85
i2	201	1	434	32	0.26
i3	132	6	259	0	0.02
i4	192	6	439	0	0.04
i5	133	66	447	3	0.04
i6	138	67	831	30	0.08
i7	199	67	1104	36	0.13
i8	133	81	3444	791	2.12
i9	88	63	981	63	0.74
i10	257	224	2935	6543	17.06
pair	173	137	1907	5272	0.94
rot	135	107	1199	1125	1.19
s1196	32	32	510	224	0.41
s1238	32	32	565	238	0.55
s1423	91	79	554	2974	0.98
s9234	247	250	2206	2323	2.49
term1	34	10	746	45	0.17
too_large	38	3	8746	90	644.02
x1	51	35	1317	187	1.03
x3	135	99	1464	268	0.22
x4	94	71	794	315	0.13

4 are the total sum of decompositions for all outputs of the circuit. Unfortunately, it is not possible to compare our results to the results of other algorithms for non-disjoint decompositions, because none of them reports the number of all decompositions for a given function.

Column 6 shows runtime, in seconds, measured using the Unix command

time (user time). The experiments were performed on a PC with a 1.4 GHz Pentium4 CPU and 1 GByte main memory. One can see that, for circuits with less than 1000 gates, the runtime is of order of 1 sec. The largest circuit, *too_large*, with 8746 gates, takes 10 min. *gates* in Column 4 are the 2-input AND gates, because our implementation uses an AND/INVERTER graph [97] for representing circuits. The presented algorithm can decompose functions for which BDDs cannot be build, such as 16-bit multiplier $C6288$.

D.6 Conclusion

This paper shows that the problem of computing non-disjoint decompositions of Boolean functions can be reduced to the problem of finding multiple-vertex dominators in circuits. We also prove that, for a given circuit, all multiple-vertex dominators of a fixed size can be found in polynomial time. This implies that certain non-disjoint decompositions (the ones reflected in the circuit structure) can be computed in polynomial time.

Our ongoing work includes developing a more efficient algorithm for computing multiple-vertex dominators.

Paper E

Bound Set Selection and Circuit Re-Synthesis for Area/Delay Driven Decomposition

Andrés Martinelli and Elena Dubrova. Published in the "Proceedings of the Design, Automation & Test in Europe Conference 2005" (DATE 2005), March 7–11, 2005, Munich, Germany, pp. 430–431.

Bound Set Selection and Circuit Re-Synthesis for Area/Delay Driven Decomposition

Andrés Martinelli* Elena Dubrova*

Abstract

This paper addresses two problems related to disjoint-support decomposition of Boolean functions. First, we present a heuristic for finding a subset of variables, X, which results in the disjoint-support decomposition $f(X,Y) = h(g(X),Y)$ with a good area/delay trade-off. Second, we present a technique for re-synthesis of the original circuit implementing $f(X,Y)$ into a circuit implementing the decomposed representation $h(g(X),Y)$. Preliminary experimental results indicate that the proposed approach has a significant potential.

E.1 Introduction

Disjoint-support decomposition of a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ is a representation of the form $f(X,Y) = h(g(X),Y)$ where $X \cap Y = \emptyset$, $g : \{0,1\}^{|X|} \rightarrow \{0,1,...,k-1\}$ and $h : \{0,1,...,k-1\} \times \{0,1\}^{|Y|} \rightarrow \{0,1\}$. The k-valued function g can be encoded as

$$f(X,Y) = h(g_1(X), g_2(X), \ldots, g_{\lceil \log_2 k \rceil}(X), Y)$$

giving a decomposition with all functions being Boolean. Every set of variables X for which such a decomposition exists is called a *bound set* for f. This paper addresses two problems related to disjoint-support decomposition. First, we present a heuristic for finding a bound set which results in

*{andres,elena}@imit.kth.se, Royal Institute of Technology, IMIT/KTH, 164 46 Kista, Sweden

a disjoint-support achieving a good area/delay trade-off. Choosing a suitable bound set is important because disjoint-support decomposition does not necessarily simplify the function.

Second, we present a technique for transforming the original circuit implementing $f(X, Y)$ into a circuit implementing the decomposed representation $h(g(X), Y)$. Previous algorithms computed circuits for the decomposed representation from Binary Decision Diagrams (BDDs) of g and h, by applying various BDD-to-circuit transformation techniques. The algorithm presented in this paper uses BDDs only for *analysis* of the decomposition. The actual *synthesis* of the circuits for g and h is done by restricting the original circuit with respect to a given assignment of input variables. This guarantees that the sizes of the circuits of g and h are strictly smaller than the size of the original circuit.

E.2 Bound Set Selection

To find a suitable bound set X for f, we examine all linear intervals of variables of the BDD representing f. To check whether a given linear interval is a bound set, we use IntervalCut algorithm [111]. IntervalCut is very fast, because it does not require expensive BDD re-ordering.

If a bound set X with the column multiplicity $k < |X|$ is found, it is stored together with the following three parameters characterizing the associated decomposition $f(X, Y) = h(g(X), Y)$:

1. the number of outputs having X as a bound set: $s(X)$;
2. the number of outputs of g: $c(X) = \lceil \log_2 k \rceil$;
3. the difference in sizes of the bound set X and the free set Y: $d(X) = ||X| - |Y||$, $d(X) \in \{0, 1, \ldots, n-1\}$.

Let $\mathbf{X}$ be the set of bound sets computed by IntervalCut. The best candidate is selected from $\mathbf{X}$ as follows. First, a subset $\mathbf{X_s}$ of $\mathbf{X}$ containing all bound sets with the maximum $s(X)$ is chosen. Maximizing of $s(X)$ increases the sharing of common logic among different outputs of the circuit. Next, a subset $\mathbf{X_c}$ of $\mathbf{X_s}$ containing all bound sets with the minimum $c(X)$ is selected. Minimizing of $c(X)$ promotes the selection of bound sets with the smallest column multiplicity (more precisely, smallest $\log_2 k$). Finally,

a subset $\mathbf{X_d}$ of $\mathbf{X_c}$ containing largest bound sets with the minimum $d(X)$ is obtained. Minimizing of $d(X)$ allows balancing the partitioning of logic between the functions g and h.

Any element of $\mathbf{X_d}$ is considered to be a "best" bound set for f, i.e. the one which produces a decomposition with the best area/delay trade-off. The original circuit implementing f is transformed into the circuit implementing $h(g(X), Y)$ by applying the algorithm described in the next section.

E.3 Transformation Algorithm

Let X be a bound set for f and let G_g and G_h be BDDs representing the functions g and h in the decomposition $f(X, Y) = h(g(X), Y)$. These BDDs are computed by IntervalCut.

Constructing the circuit for h

Suppose A is an assignment of variables of X leading to the 0-terminal node in G_g. Then $g(A) = 0$, and thus $f(A, Y) = h(g(A), Y) = h(0, Y)$. Therefore, a circuit implementing the co-factor $h(0, Y)$ can be obtained from the circuit implementing f by applying the assignment A to the inputs X and propagating the constants through the circuit using the usual reduction rules. Similarly, circuits implementing co-factors $h(i, Y)$, $i \in \{1, 2, \ldots, k-1\}$, can be obtained by propagating an assignment of variables of X leading to the i-terminal node of G_g. Recall, that g is a function of type $g : \{0,1\}^{|X|} \rightarrow \{0, 1, ..., k-1\}$, so G_g is a multi-terminal BDD with k terminal nodes.

To maximize the sharing of common logic of the i circuits implementing co-factors $h(i, Y)$, $i \in \{0, 1, \ldots, k-1\}$, i assignments A are chosen so that they differ in the fewest number of bit positions.

The function $h(g(X), Y)$ is obtained by combining the co-factors in a Shannon expansion as follows:

$$h(g(X), Y) = \sum_{i=0}^{k-1} g_1^{i_1}(X) g_2^{i_2}(X) \ldots g_r^{i_r}(X) h(i, Y) \qquad \text{(E.1)}$$

where $(i_1, i_2, \ldots, i_r)$ is the binary expansion of i, $r = \lceil \log_2 k \rceil$, and the term

$g_j^{i_j}$ is defined by

$$g_j^{i_j} = \begin{cases} g_j & \text{if } i_j = 1 \\ \overline{g}_j & \text{otherwise} \end{cases}$$

for $j \in \{1, 2, \ldots, r\}$.

Constructing the circuit for g

Suppose that B is an assignment of variables of Y such that $h(i, B) \neq h(j, B)$ for some $i, j \in \{0, 1, \ldots, k-1\}$, $i \neq j$. Then $f(X, B) = h(g(X), B)$ where the co-factor $h(g(X), B)$ is neither constant 0, nor constant 1, i.e. it depends of $g(X)$.

Since h is a function of type $\{0, 1, ..., k-1\} \times \{0, 1\}^{|Y|} \rightarrow \{0, 1\}$, the co-factor $h(g(X), B)$ is a function of type $\{0, 1, ..., k-1\} \rightarrow \{0, 1\}$. Note that, for $k = 2$, $h(g(X), B)$ is either an identity, or a complement. Thus, at this step, the problem of constructing the circuit for $g(X)$ is solved for $k = 2$. For larger values of k, the following strategy is used.

The k-valued function $g(X)$ can be expressed as

$$g(X) = \sum_{i=0}^{k-1} i \cdot g^i(X)$$

where $g^i : \{0, 1, \ldots, k-1\}^{|X|} \rightarrow \{0, 1\}$ are multiple-valued *literals* defined as:

$$g^i(X) = \begin{cases} 1 & \text{if } g(X) = i \\ 0 & \text{otherwise} \end{cases}$$

For a given encoding of k values of $g(K)$, each of the functions $g_1(X)$, $g_2(X), \ldots, g_r(X)$, $r = \lceil \log_2 k \rceil$, encoding $g(X)$, can be represented as a sum of some literals $g^i(X)$'s.

Consider a decomposition chart of $h(g(X), Y)$ with columns representing k values of $g(X)$ and the rows represent all combinations of the variables of Y. Any non-constant row of $h(g(X), Y)$ represents a sum of some literals $g^i(X)$, $i \in \{0, 1, \ldots, k-1\}$.

In the best case, there exist rows in the decomposition chart corresponding directly to the encoded functions $g_1(X), g_2(X), \ldots, g_r(X)$. If $h(g(X), A) = g_j(X)$ for some assignment A of the variables of Y, then the circuit implementing $g_j(X)$ can be obtained from the circuit implementing f by applying the assignment A to the inputs Y and propagating the constants.

In the worst case, the literals $g^i(X)$, $i \in \{0, 1, \ldots, k-1\}$, need to be computed by ANDing selected rows of $h(g(X), Y)$. Afterward, the functions $g_1(X), g_2(X), \ldots, g_r(X)$ are obtained as a combination of $g^i(X)$.

E.4 Conclusion and Future Work

This paper has two contributions: (1) a heuristic for finding a bound set X which results in the disjoint-support decomposition with a good area/delay trade-off; (2) an algorithm which transforms the original circuit into the decomposed circuit.

Our preliminary experimental results on IWLS'02 benchmarks set show that the proposed technique usually results in a smoother trade-off between area and delay compared to the one of SIS. More experiments are needed to make a thorough evaluation.

Paper F

Bound-Set Preserving ROBDD Variable Orderings May Not Be Optimum

Maxim Teslenko, Andrés Martinelli and Elena Dubrova. Published in the "IEEE Transactions on Computers", Vol. 54, no. 2, February 2005, pp. 236–238.

Bound-Set Preserving ROBDD Variable Orderings May Not Be Optimum

Maxim Teslenko* Andrés Martinelli* Elena Dubrova*

Abstract

This paper reports a result concerning the relation between the best variable orderings of a ROBDD G_f and the decomposition structure of the Boolean function f represented by G_f. It was stated in [87] that, if f has a decomposition of type $f(X) = g(h_1(Y_1), h_2(Y_2), \ldots, h_k(Y_k))$, where $\{Y_i\}$, $i \in \{1, 2, \ldots, k\}$, is a partition of X, then one of the orderings which keeps the variables within the sets $\{Y_i\}$ adjacent is a best ordering for G_f. Using a counterexample, we show that this statement is incorrect. and explore under which conditions this claim does not hold.

F.1 Introduction

This paper gives a counterexample to the following theorem from [87, p. 58, Theorem 3.8]. Let $f(X)$ be a Boolean function of type $f : B^n \rightarrow B$ on $B = \{0, 1\}$, of the variables $X = \{x_1, x_2, \ldots, x_n\}$. Let $\langle X \rangle$ denote a set of variable orderings induced by all possible permutations over the set X.

Theorem 11. *If $f(X)$ has a decomposition of type*

$$f(X) = g(h_1(Y_1), h_2(Y_2), \ldots, h_k(Y_k))$$

where $\{Y_i\}$, $1 \leq i \leq k$, is a partition of X, and h_i, g are functions of type $h_i : B^{|Y_i|} \rightarrow B$, $g : B^k \rightarrow B$, then there exists a variable ordering belonging to the set $\langle \langle Y_1 \rangle, \langle Y_2 \rangle, \ldots, \langle Y_k \rangle \rangle$ which is best.

*The authors are with the Department of Microelectronics and Information Technology, Royal Institute of Technology (KTH), Stockholm, Sweden. E-mail: {maximt, andres, elena}@imit.kth.se.

F.2 Counterexample

A set of variables $Y \subseteq X$ is a *bound set* for $f(X)$ if f can be decomposed as $f(X) = g(h(Y), Z)$, where $Z = X - Y$, and h and g are functions of type $h : B^{|Y|} \rightarrow B$, $g : B \times B^{|Z|} \rightarrow B$.

We say that two sets X and Y *overlap* if $X - Y \neq \emptyset$, $X \cap Y \neq \emptyset$ and $Y - X \neq \emptyset$.

Definition 1. *A* bound-set-preserving *ordering is an ordering which keeps the variables from all non-overlapping bound sets of the function adjacent.*

It was proved in [7] that, for any Boolean function $f(X)$ depending on all its variables, the set of all non-overlapping bound sets related by inclusion form a tree which is unique for $f(X)$ (up to complementation). Therefore, the set of bound-set-preserving orderings is uniquely defined for a given function.

Theorem 12. *There exists a function for which no bound-set-preserving ordering is best.*

Proof. By construction. Suppose a Boolean function $f(X)$ has a decomposition of type

$$f(X) = g(h_1(Y_1), h_2(Y_2), h_3(Y_3), h_4(Y_4), x_m),$$

where $\{Y_i\}$, $1 \leq i \leq 4$, and x_m is a partition of X, g is a function

$$g = h_3(h_4(h_2' + x_m') + h_1' x_m) + h_3'(h_4 x_m + h_1(h_2 \oplus x_m)),$$

where $h_i(Y_i) = \bigvee_{j \in Y_i} x_j$, $i \in \{1, 2, 4\}$, $h_3(Y_3) = (h_{31}(Y_{31}) \oplus x_k)'$ where $Y_{31} = Y_3 - \{x_k\}$, "$\oplus$" is an XOR and $h_{31}(Y_{31}) = \bigvee_{j \in Y_{31}} x_j$.

From the structure of f, we can see that the set of all bound-set-preserving orderings of G_f is given by $\langle\langle Y_1\rangle, \langle Y_2\rangle, \langle\langle Y_{31}\rangle, x_k\rangle, \langle Y_4\rangle, x_m\rangle$.

Since, h_1, h_2 and h_4 are totally symmetric functions, the structure and the size of their ROBDDs do not depend on the variable ordering. In addition, h_1, h_2 and h_4 are OR operations, and thus their ROBDDs do not contain any pairs of sub-graphs representing functions which are complements of each other. According to , this implies that the OBDD resulting after the substitution of nodes h_1, h_2 and h_4 in G_g by their corresponding ROBDDs is reduced. So, each node labeled by h_i, $i \in \{1, 2, 4\}$, contributes exactly $|G_{h_i}|$ nodes to G_f (terminal nodes are not included in the count).

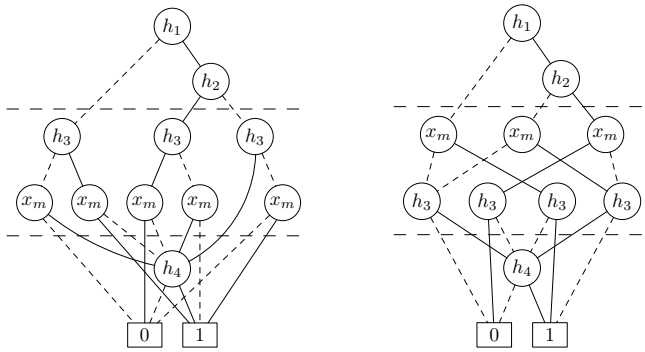


Figure F.1: Two cases of ROBDDs for g with the smallest number of nodes labeled by h_1, h_2, h_4.

On the other hand, since h_3 is decomposable by an XNOR operation, its ROBDD contains pairs of sub-graphs representing functions which are complements of each other. Therefore, the OBDD resulting after the substitution of nodes h_3 may be non-reduced [62]. The amount of reduction cannot be estimated without analyzing the structure of G_g and G_{h_3} for each particular order.

To make the size of G_f less dependent on the size of G_{h_3}, we impose the condition that the ROBDDs for h_1, h_2 and h_4 is much larger than G_{h_3}, i.e. $|G_{h_1}| = |G_{h_2}| = |G_{h_4}| >> |G_{h_3}|$.

Then, the only potential candidates for best orderings of G_f are the orderings of G_g which have the smallest number of nodes labeled by h_1, h_2 and h_4.

By exhaustive search through all possible orderings of G_g, we can determine that ROBDDs for orderings (h_1, h_2, h_3, x_m, h_4) and (h_1, h_2, x_m, h_3, h_4), shown in Figure F.1, are the only two ROBDDs that have one node for each of h_1, h_2 and h_4. ROBDDs for all other orderings have more than one node per at least one of h_1, h_2 or h_4. The overall size of G_f is given by $G_f = |G_{h_1}| + |G_{h_2}| + |G_{h_4}| + N_{[h_3, x_m]}$, where $N_{[h_3, x_m]}$ is the number of nodes within the interval shown in Figure F.1 by dotted lines.

Next, we show that the number of nodes in G_f can be reduced by making the ordering not bound-set-preserving.

Suppose the nodes h_3 in G_g are substituted by ROBDDs for $h_3 = (h_{31} \oplus x_k)'$. There are six possible choices to order the variables h_{31}, x_k and x_m

within the interval shown in Figure F.1 by dotted lines. For each choice, we compute $N_{[h_3,x_m]}$. Note, that each node labeled by h_{31} contributes exactly $|G_{h_{31}}|$ nodes to G_f, since h_{31} is an OR operation and the reasoning from above applies.

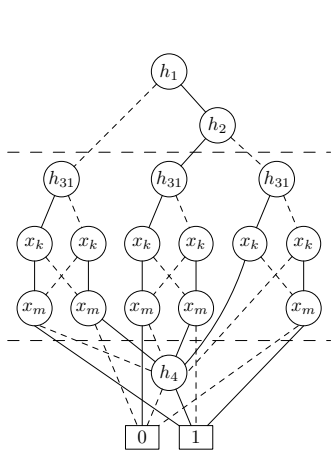
1. For the ordering (h_{31}, x_k, x_m), $N_{[h_3,x_m]} = 3|G_{h_{31}}| + 6 + 5$ (Fig. F.2(a)).
2. For (h_{31}, x_m, x_k), $N_{[h_3,x_m]} = 3|G_{h_{31}}| + 6 + 4$ (Fig. F.2(b)).
3. For (x_m, h_{31}, x_k), $N_{[h_3,x_m]} = 3 + 4|G_{h_{31}}| + 4$ (Fig. F.2(c)).
4. Since $h_3 = (h_{31} \oplus x_k)'$ and XNOR is symmetric, the graph for the ordering (x_m, x_k, h_{31}) is the same as the graph for the ordering (x_m, h_{31}, x_k) (Fig. F.2(c)) with the variables x_k and h_{31} permuted. $N_{[h_3,x_m]} = 3 + 4 + 4|G_{h_{31}}|$.
5. For (x_k, x_m, h_{31}), we have $N_{[h_3,x_m]} = 3 + 6 + 4|G_{h_{31}}|$. The structure of the graph is the same as in Figure F.2(b) with the variables x_k and h_{31} permuted.
6. For (x_k, h_{31}, x_m), $N_{[h_3,x_m]} = 3 + 6|G_{h_{31}}| + 5$. The structure of the graph is the same as in Figure F.2(a) with the variables x_k and h_{31} permuted.

For $|G_{h_{31}}| \geq 4$ the ordering $(h_1, h_2, h_{31}, x_m, x_k, h_4)$ (Fig. F.2(b)) gives us the smallest number of nodes. This ordering does not preserve the bound set Y_3. Therefore, the best ordering for G_f is not bound-set-preserving. This proves the theorem. □

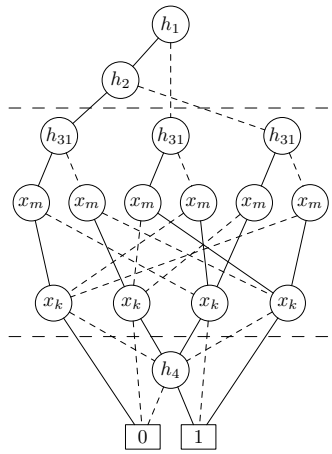
Theorem 12 also holds for ROBDDs with complemented edges (for the same function as in the proof).

F.3 Conclusion

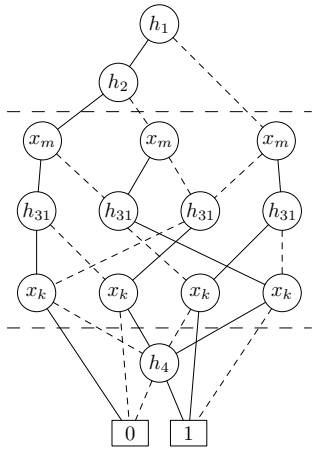
In this paper, we show that bound-set-preserving orderings may not be best for ROBDDs. Such cases, however, are rare. Their existence does not diminish the practical value of using bound sets as a guide for grouping ROBDD variables, but should be noted as a possibility.



(a) $(h_1, h_2, h_{31}, x_k, x_m, h_4)$



(b) $(h_1, h_2, h_{31}, x_m, x_k, h_4)$



(c) $(h_1, h_2, x_m,\ h_{31}, x_k, h_4)$

Figure F.2: ROBDD for different orderings.

Paper G

Kauffman Networks: Analysis and Applications

Elena Dubrova, Maxim Teslenko and Andrés Martinelli. Published in the "Proceedings of the ACM/IEEE International Conference on Computer-Aided Design 2005" (ICCAD 2005), November 6–10, 2005, San Jose, California, USA, pp. 479–484

Kauffman Networks: Analysis and Applications

Elena Dubrova* Maxim Teslenko* Andres Martinelli*

Abstract

A Kauffman network is an abstract model of gene regulatory networks. Each gene is represented by a vertex. An edge from one vertex to another implies that the former gene regulates the latter. Statistical features of Kauffman networks match the characteristics of living cells. The number of cycles in the network's state space, called attractors*, corresponds to the number of different cell types. The attractor's length corresponds to the cell cycle time. The sensitivity of attractors to different kinds of disturbances, modeled by changing a network connection, the state of a vertex, or the associated function, reflects the stability of the cell to damage, mutations and virus attacks. In order to evaluate attractors, their number and lengths have to be computed. This problem is the major open problem related to Kauffman networks. Available algorithms can only handle networks with less than a hundred vertices. The number of genes in a cell is often larger. In this paper, we present a set of efficient algorithms for computing attractors in large Kauffman networks. , enabling the modeling of real living cells. The resulting software package will make possible analyzing Kauffman networks with more than 10.000 vertices, thus enabling the modeling of real living cells. The resulting software package is hoped to be of assistance in understanding the principles of gene interactions and discovering a computing scheme operating on these principles.*

G.1 Introduction

The *gene regulatory network* is one of the most important signaling networks in living cells. It is composed of the interactions of proteins with the

*Royal Institute of Technology, IMIT/KTH, 164 46 Kista, Sweden

genome [3]. The major discovery related to gene regulatory networks was made in 1961 by French biologists François Jacob and Jacques Monod [86]. They found that a small fraction of the thousands of genes in the DNA molecule acts as tiny "switches". By exposing a cell to a certain hormone, these switches can be turned "on" or "off". The activated genes send chemical signals to other genes which, in turn, get either activated or repressed. The signals propagate along the DNA molecule until the cell settles down into a stable pattern.

Jacob and Monod's discovery showed that DNA is not just a blueprint for the cell, but rather an automaton which allows for the creation of different types of cells. It answered the long open question of how one fertilized egg cell could differentiate itself into brain cells, lung cells, muscle cells, and other types of cells that form a newborn baby. Each kind of cells corresponds to a different pattern of activated genes in the automaton.

In 1969 Stuart Kauffman proposed using Boolean networks for modeling gene regulatory networks [92]. Each gene is represented by a vertex in a directed graph. An edge from one vertex to another implies a causal link between the two genes. The "on" state of a vertex corresponds to the gene being expressed. Time is viewed as proceeding in discrete steps. At each step, the new state of a vertex v is a Boolean function of the previous states of the vertices which are predecessors of v.

We discovered that many problems related to Kauffman networks are similar to the problems in logic synthesis and verification of electronic circuits. For example, the problem of finding relevant elements in Kauffman networks [12] is similar to the problem of removing redundancy in sequential logic circuits [14]. The problem of identifying state cycles in Kauffman networks [145] is related to the problem of image computation in model checking [115].

After examining the state-of-the-art in Kauffman networks, we found that existing methods for their analysis are quite immature compared to the approaches used in logic synthesis and verification. There are efficient techniques for removing redundancy from a circuit with millions of gates [14] and for verifying finite state machines with 10^{20} states [40]. The programs available for computing state cycles in Kauffman networks can only deal with networks with less than 32 relevant vertices [11, 157, 19, 143]. The number of genes in a cell is often larger. For example, the tiny worm Caenorhabditis elegans has 19.099 genes. A small flower in the mustard family, Arabidopis, has 25.498 genes [140].

To bridge this gap, we developed algorithms for redundancy removal and partitioning for Kauffman networks that have linear-time complexity and are feasible for networks with millions of vertices [66, 61, 65]. These algorithms are first steps towards solving the more central problem of computing state cycles in large Kauffman networks, which is addressed in this paper.

G.2 Kauffman Networks

In this section, we give a brief introduction to Kauffman networks. For a more detailed description, the reader is referred to [4].

Definition of Kauffman Networks

Kauffman networks are a class of *random nk-Boolean networks* [8]. A random nk-Boolean network is a synchronous Boolean automaton with n vertices. Each vertex has exactly k incoming edges, assigned at random, and an associated Boolean function. Functions are selected so that they evaluate to the values 0 and 1 with given probabilities p and $1-p$, respectively. Time is viewed as proceeding in discrete steps. At each step, the new state of a vertex v is a Boolean function of the previous states of the predecessors of v.

A Kauffman network is a random nk-Boolean network with $k = 2$ and $p = 0.5$, i.e. each vertex has two predecessors and Boolean functions are assigned to vertices independently and uniformly at random from the set of 16 possible 2-variable Boolean functions [129]. The state σ_{v_i} of a vertex v_i at time $t+1$ is determined by the states of its predecessors v_l and v_r, $i, l, r \in \{1, 2, \ldots, n\}$, as:

$$\sigma_{v_i}(t+1) = f_{v_i}(\sigma_{v_l}(t), \sigma_{v_r}(t))$$

where $f_{v_i} : \{0,1\}^2 \to \{0,1\}$ is the Boolean function associated to v_i. The vector $(\sigma_{v_1}(t), \sigma_{v_2}(t), \ldots, \sigma_{v_n}(t))$ represents the state of the network at time t. An example of a Kauffman network with ten vertices is shown in Figure G.1. We use "$\cdot$", "+" and "$'$" to denote the Boolean operations AND, OR and NOT, respectively.

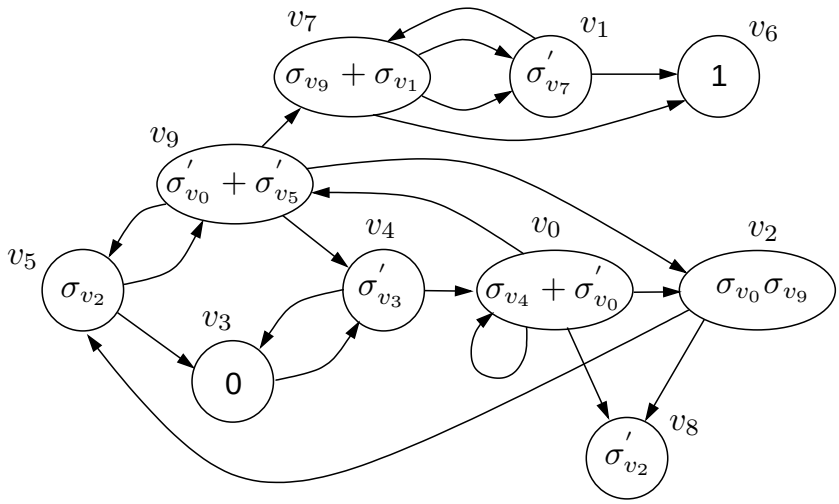


Figure G.1: Example of a Kauffman network. The state of a vertex v_i at time $t+1$ is given by $\sigma_{v_i}(t+1) = f_{v_i}(\sigma_{v_l}(t), \sigma_{v_r}(t))$, where v_l and v_r are the predecessors of v_i, and f_{v_i} is the Boolean function associated to v_i.

Frozen and chaotic phases

The parameters k and p determine the dynamics of the network. For a given probability p, there is a critical number of inputs, k_c, below which the network is in the frozen phase and above which the network is in the chaotic phase [54]:

$$k_c = \frac{1}{2p(1-p)}. \tag{G.1}$$

If a network is in the *frozen phase*, then, independently of the initial state, a stable state is reached after a few steps [70]. Small changes in network's connections, states of vertices, or associated Boolean functions, typically create no variations in the network's dynamics.

In the *chaotic phase*, the length of state cycles is of order of 2^n. The dynamics of the network is very sensitive to changes in network's connections, states of vertices, or associated Boolean functions [109].

On the critical line between the frozen and the chaotic phases, the network exhibits *self-organized critical behavior*, ensuring both stability and evolutionary improvements [10]. Statistical features of random nk-Boolean

networks on the critical line are shown to match the characteristics of real cells and organisms [92, 93, 4]. For $p = 0.5$, the critical number of inputs is $k_c = 2$, so Kauffman networks are on the critical line.

Apart from gene regulatory networks, Kauffman networks have been applied to the problems of cell differentiation [83], immune response [94], and evolution [28]. They have also attracted the interest of physicists due to their analogy with disordered systems studied in statistical mechanics, such as the mean field spin glass [52].

Attractors

Since the number of possible states of a Kauffman network is finite (up to 2^n), any sequence of consecutive states of a network eventually converges to either a single state, or a cycle of states, called *attractor*. The number and length of attractors represent two important parameters of the cell modeled by a Kauffman network. The number of attractors corresponds to the number of different *cell types*. For example, humans have 20.000-25.000 genes (the exact number is not known yet) and about 250 cell types [106]. The attractor's length corresponds to the *cell cycle time*. Cell cycle time refers to the amount of time required for a cell to grow and divide into two daughter cells. The length of the total cell cycle varies for different types of cells.

The human body has a sophisticated system for maintaining normal cell repair and growth. The body interacts with cells through a feedback system that signals a cell to enter different phases of the cycle [51]. If a person is sick, e.g suffers from cancer, then this feedback system does not function normally and cancer cells enter the cell cycle independently of the body's signals. The number and length of attractors of a Kauffman network serve as indicators of the health of the cell modeled by the network [145]. The sensitivity of attractors to different kinds of disturbances, modeled by changing the state of a vertex, the associated Boolean function, or a network connection, reflects the stability of the cell to damage, mutations and virus attacks.

In order to evaluate attractors, their number and length have to be computed. This problem is the major problem in the analysis of Kauffman networks, for which no efficient solution is found so far. Available algorithms for exact computation of attractors can only handle networks with less than 32 non-redundant vertices [11, 157, 19, 143]. For larger networks, the median instead of the exact values on the number of attractors is computed using the

```
algorithm RemoveRedundant(V, E)
    /* I. Edge Combining */
    for each v ∈ V do
        if two incoming edges of v come from the same vertex then
            Simplify f_v;
    /* II. Constant Propagation */
    R_1 = ∅;
    for each v ∈ V do
        if f_v is a constant then
            Append v at the end of R_1;
    for each v ∈ R_1 do
        for each u ∈ S_v − R_1 do
            Simplify f_u by substituting constant f_v;
            if f_u is a constant then
                Append u at the end of R_1;
    Remove all v ∈ R_1 and all edges connected to v;
    /* III. Copy Propagation */
    for each v ∈ V do
        if f_v is a 1-variable function then
            Remove the edge (u, v), where u is the
            predecessor of v on which v does not depend;
    /* IV. Dead Code Elimination */
    R_2 = ∅;
    for each v ∈ V do
        if S_v = ∅ then
            Append v at the end of R_2;
    for each v ∈ R_2 do
        for each u ∈ P_v − R_2 do
            if all ancestors of u are in R_2 then
                Append u at the end of R_2;
    Remove all v ∈ R_2 and all edges connected to v;
end
```

Figure G.2: The algorithm for finding redundant vertices in Kauffman networks.

following technique [143]. Repeatedly, an initial state is chosen at random and the attractor reachable from this state is computed. If 1000 consecutive attempts yield no new attractor, the algorithm terminates. The resulting number is used as a lower bound on the number of attractors in the network.

G.3 Redundancy Removal

Redundancy is an essential feature of biological systems, ensuring their correct behavior in presence of internal or external disturbances. An overwhelming percentage (about 95%) of DNA of humans is redundant to the

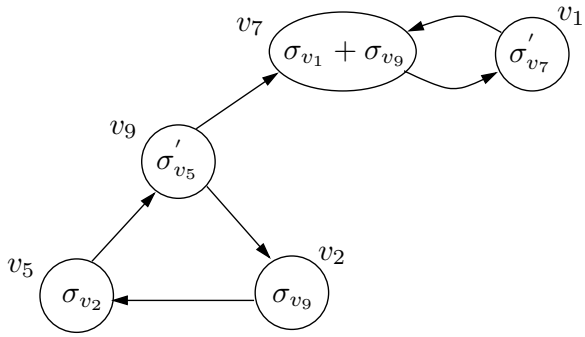


Figure G.3: Reduced network G_R for the Kauffman network in Figure G.1.

metabolic and developmental processes. Such "junk" DNA is believed to act as a protective buffer against genetic damage and harmful mutations, reducing the probability that any single, random offense to the nucleotide sequence will affect the organism [147].

In the context of Kauffman networks, redundancy is defined as follows. Let $G = (V, E)$ be a Kauffman network, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges connecting the vertices.

Definition G.3.1*:* A vertex $v \in V$ of a Kauffman network G is redundant if the network obtained from G by removing v has the same number and length of attractors as G.

If a vertex in not redundant, it is called *relevant* [11].

In [11], an algorithm for computing the set of all redundant vertices was presented. This algorithm has a high complexity, and therefore is only applicable to small Kauffman networks with up to a hundred vertices. In [61], we presented an algorithm RemoveRedundant (Figure G.2), which quickly finds structural redundancy and some simple cases of functional redundancy. The phases II and IV of Remove Redundant are similar to the *decimation procedure* of [19], although a detailed comparison is hard to do because no pseudo-code is shown in [19]. The ordering of the phases of the algorithm is very important. For example, if the phase IV is performed before the phase II, then usually less redundant vertices are found.

Let $P_v = \{u \in V \mid (u, v) \in E\}$ be a set of *predecessors* of $v \in V$ and $S_v = \{u \in V \mid (v, u) \in E\}$ be a set of *successors* of v.

RemoveRedundant first checks whether there are vertices v with two incoming edges coming from the same vertex. If yes, the associated functions f_v are simplified.

Then, RemoveRedundant classifies as redundant all vertices v whose associated function f_v is constant 0 or constant 1. Such vertices are collected in a list R_1. Then, for every vertex $v \in R_1$, successors of v are visited and the functions associated to the successors are simplified. The simplification is done by substituting the constant value of f_v in the function of the successor u. If as a result of the simplification the function f_u reduces to a constant, then u is appended to R_1.

Second, RemoveRedundant finds all vertices whose associated function f_v is a single-variable function. The edge between v and the predecessor of v which v does not depend on is removed.

Next, RemoveRedundant classifies as redundant all vertices which have no successors. Such vertices are collected in a list R_2. For every vertex $v \in R_2$, both predecessors of v are visited. If all successors of some predecessor $u \in P_v$ are redundant, u is appended at the end of R_2.

The worst-case time complexity of RemoveRedundant is $O(|V|+|E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in G.

As we mentioned before, RemoveRedundant might not identify all cases of functional redundancy. For example, a vertex may have a constant output value due to the correlation of its input variables. For example, if a vertex v with an associated OR (AND) function has predecessors v_l and v_r with functions $f_{v_l} = \sigma_{v_j}$ and $f_{v_r} = \sigma'_{v_j}$, then the value of f_v is always 1 (0). Such cases of redundancy are not detected by RemoveRedundant.

Let G_R be the reduced network obtained from G by removing redundant vertices. The reduced network for the example in Figure G.1 is shown in Figure G.3. Its state transition graph is given in Figure G.4. Each vertex of the state transition graph represents a 5-tuple $(\sigma(v_1)\sigma(v_2)\sigma(v_5)\sigma(v_7)\sigma(v_9))$ of values of states on the relevant vertices v_1, v_2, v_5, v_7, v_9. There are two attractors: {01111, 01110, 00100, 10000, 10011, 01011}, of length six, and $\{00101, 11010, 00111, 01010\}$, of length four. By Definition G.3.1, by removing redundant vertices we do not change the total number and length of attractors in a Kauffman network. Therefore, G_R has the same number and length of attractors as G.

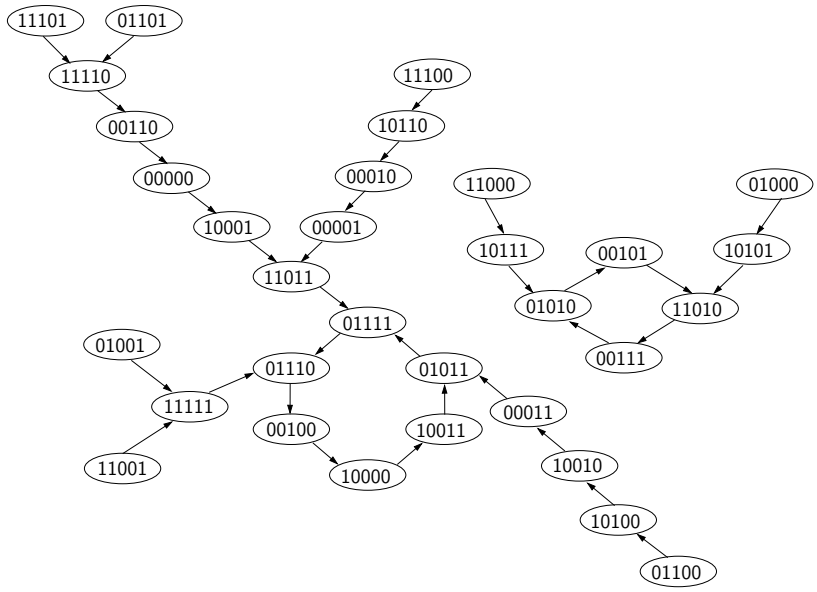


Figure G.4: State transition graph of the Kauffman network in Figure G.3. Each state is a 5-tuple $(\sigma(v_1)\sigma(v_2)\sigma(v_5)\sigma(v_7)\sigma(v_9))$.

G.4 Partitioning

The vertices of G_R induce a number of connected components.

Definition G.4.1*:* Two relevant vertices are in the same component if and only if there is an undirected path between them.

A path is called *undirected* if it ignores the direction of edges.

Connected components can be computed in $O(|V| + |E|)$ time, where $|V|$ is the number of vertices and $|E|$ is the number of edges of G_R, using the following algorithm [149]. To find a connected component number i, the function ComponentSearch(v) is called for a vertex v which has not been assigned to a component yet. ComponentSearch does nothing if v has been assigned to a component already. Otherwise, ComponentSearch assigns v to the component i and calls itself recursively for all predecessors and successors of v. The process repeats with the counter i incremented until all vertices are assigned.

In [65], we have shown that attractors of a Kauffman network can be

computed compositionally from the attractors of the connected components of G_R. Let $\{G_1, G_2, \ldots, G_p\}$ be the set of components of G_R, N_i be the number of attractors of G_i, L_{ij} be the length of the jth attractor G_i and $I = I_1 \times I_2 \times \ldots \times I_p$ be the Cartesian product of sets $I_i = \{i_1, i_2, \ldots, i_{N_i}\}$, $i = \{1, 2, \ldots, p\}$, $j = \{1, 2, \ldots, N_i\}$. Then, the total number of attractors in G_R is given by

$$N = \sum_{\forall(i_1,\ldots,i_p)\in I} \prod_{j=2}^{p} (((L_{1i_1} \bullet L_{2i_2}) \bullet L_{3i_3}) \ldots \bullet L_{j-1i_{j-1}}) \circ L_{ji_j}$$

where "$\bullet$" is the least common multiple operation and "$\circ$" is the greatest common divisor operation. The maximum length of attractors is given by

$$L_{max} = \max_{\forall(i_1,\ldots,i_p)\in I} ((L_{1i_1} \bullet L_{2i_2}) \bullet L_{3i_3}) \ldots \bullet L_{pi_p}$$

where "$\bullet$" is the least common multiple operation.

G.5 Computation of Attractors

To be able to compute attractors in a large Kauffman network, it is important to use an efficient representation for its set of states, and for the transition relation on this set. In our current implementation, we use *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [36].

A *transition relation* defines the next state values of the vertices in terms of the current state values. We derive the transition relation in the standard way [40], by assigning every vertex v_i of the network a state variable x_{v_i} and making two copies of the set of state variables: $s = (x_{v_1}, x_{v_2}, \ldots, x_{v_r})$, denoting the variables of the current state, and $s^+ = (x^+_{v_1}, x^+_{v_2}, \ldots, x^+_{v_r})$, denoting the variables of the next state. Using this notation, the characteristic formula for the transition relation of a Kauffman network is given by:

$$T(s, s^+) = \bigwedge_{i=1}^{r} (x^+_{v_i} \leftrightarrow f_i(x_{v_{i_1}}, x_{v_{i_2}})),$$

where r is the number of relevant vertices, f_i is the Boolean function associated with the vertex v_i and v_{i_1} and v_{i_2} are the predecessors of v_i.

As an example, consider the reduced Kauffman network in Figure G.3 and its state transition graph in Figure G.4. We have $s = (x_{v_1}, x_{v_2}, x_{v_5}, x_{v_7}, x_{v_9})$

and $s^+ = (x^+_{v_1}, x^+_{v_2}, x^+_{v_5}, x^+_{v_7}, x^+_{v_9})$. The transition relation is given by:

$$\begin{aligned} T(s, s^+) &= (x^+_{v_1} \leftrightarrow x'_{v_7}) \wedge (x^+_{v_2} \leftrightarrow x_{v_9}) \wedge (x^+_{v_5} \leftrightarrow x_{v_2}) \\ &\quad \wedge (x^+_{v_7} \leftrightarrow (x_{v_1} + x_{v_9})) \wedge (x^+_{v_9} \leftrightarrow x'_{v_5}). \end{aligned}$$

Let $T^i(s, s^+)$ denote the transition relation describing the set of next states s^+ that can be reached from any current state s in i steps. For $i = 2$, $T^2(s, s^+)$ is computed as follows:

$$T^2(s, s^+) = \exists s^{++}.(T(s, s^{++}) \wedge T(s^{++}, s^+)).$$

By applying squaring iteratively, we can obtain $T^{2^i}(s, s^+)$ in i steps for any i [39].

One one hand, for a Kauffman network with r relevant vertices, it cannot take more than 2^r steps to reach an attractor from any state. One the other hand, "overshooting" is not a problem because, once entered, an attractor is never left. Therefore, for any initial state s, the next state s^+ obtained by the transition defined by $T^{2^r}(s, s^+)$ is a state of an attractor.

Let $F_i(s)$ denote the set of states reachable from a given set of initial states in i steps. Using the transition relation $T^{2^r}(s, s^+)$, we can compute the set of states $F_{2^r}(s)$ that can be reached from *any* state in 2^r steps as:

$$F_{2^r}(s^+) = \exists s.T^{2^r}(s, s^+).$$

$F_{2^r}(s^+)$ represents the set of states of *all* attractors. It remains to distinguish between different attractors. This can be done by picking up an arbitrary state s of $F_{2^r}(s^+)$ and following its next states until s is not reached again. This process is repeated starting from a state of $F_{2^r}(s^+)$ which was not visited previously until $F_{2^r}(s^+)$ is covered.

Our simulation results show that the length and the number of attractors in a Kauffman network with n vertices are of order of $\sqrt{n}$, which makes the proposed approach efficient.

G.6 Simulation Results

This section shows simulation results for Kauffman networks of sizes from 10 to 10^7 vertices (Table G.1). Column 2 gives the average number of relevant vertices computed using RemoveRedundant. Column 3 shows

total number of vertices	average number of relevant vertices	average size of the largest component	average number of components	average number of attractors
10	5	5	1.1	2.67
10^2	25	25	1.4	11.7
10^3	93	92	1.8	23.9*
10^4	270	266	2.4	-
10^5	690	682	3.1	-
10^6	1614	1596	3.7	-
10^7	3502	3463	4.3	-

Table G.1: Simulation results. Average values for 1000 networks. "$*$" indicates that the average is computed only for successfully terminated cases.

the average size of the largest connected component of the sub-graph G_R induced by the relevant vertices and column 4 gives the average number of components. Column 5 shows the average number of attractors.

The simulation results show that we need to find a better way of partitioning. Currently, the size of the largest component of the sub-graph induced by the relevant vertices (column 3) is $\Theta(r)$, where r is the number of relevant vertices in the sub-graph, i.e. we observe so called "giant" component phenomena [123]. A technique resulting in a more balanced partitioning is needed.

Another problem is that, on random graphs, ROBDDs blow up more frequently than on sequential circuits. Currently, we cannot compute the exact number of attractors in most networks with 10^3 vertices and larger. The number of attractors shown in column 5 for networks with 10^3 vertices is the average value computed for successfully terminated cases only. We did have occasional blow ups for networks with 100 vertices as well. The number of attractors shown in column 5 for networks with 100 vertices is the average value computed for 1000 successfully terminated cases. In our future work, we plan to investigate possibilities for implementing the algorithm presented in Section G.5 using Boolean circuits [18, 1, 96, 24], rather than ROBDDs, and combined approaches [128, 155]. We will also try reducing the state space by detecting equivalent state variables [153] and by partitioning the transition relation [74].

G.7 Applications

In this section we present some ideas on how Kauffman networks can be used for implementing Boolean functions and for achieving fault-tolerance. The ideas we describe are preliminary, more research is needed to justify them.

Implementing logic functions by Kauffman networks

An interesting direction of research is investigating how Kauffman networks can be used for implementing logic functions. One possibility is to use the states of relevant vertices of a network to represent variables of the function, and to use the attractors to represent the function's values.

To be more specific, suppose that we have a Kauffman network G with r relevant vertices $v_1, \ldots, v_r$ and m attractors $A_1, A_2, \ldots, A_m$. The basins of attractions of A_i's partition the Boolean space B^r into m connected components. We assign a value i, $i \in \{0, 1, \ldots, m-1\}$ to the attractor A_i and assume that the set of minterms represented by the states in the basin of attraction of A_i is mapped to k. Then, G implements the function $f : \{0,1\}^r \rightarrow \{0, 1, \ldots, m-1\}$ of variables $x_1, \ldots, x_r$, where the value of the variable x_i corresponds to the state of relevant vertex v_i. The mapping is unique up to permutation of m output values of f. If $m = 2$, then G implements a Boolean function.

As an example, consider the Kauffman network G shown in Figure G.5. The vertices v_4 and v_5 are relevant vertices, determining the dynamic of G according to the reduced network in Figure G.6(a). The state transition graph of the reduced network is shown in Figure G.6(b). There are two attractors, A_1 and A_2. We assign the logic 0 to A_1 and the logic 1 to A_2. The initial states $00, 01$ and 10 terminate in the attractor A_1 (logic 0) and the initial state 11 terminates in the attractor A_2 (logic 1). So, G implements the 2-input Boolean AND.

Stability

Extensive experimental results confirm that Kauffman networks are tolerant to faults, i.e. typically the number and length of attractors are not affected by small changes [93, 4]. The following types of fault models are used to model the effects of diseases, mutations, or injuries on a cell:

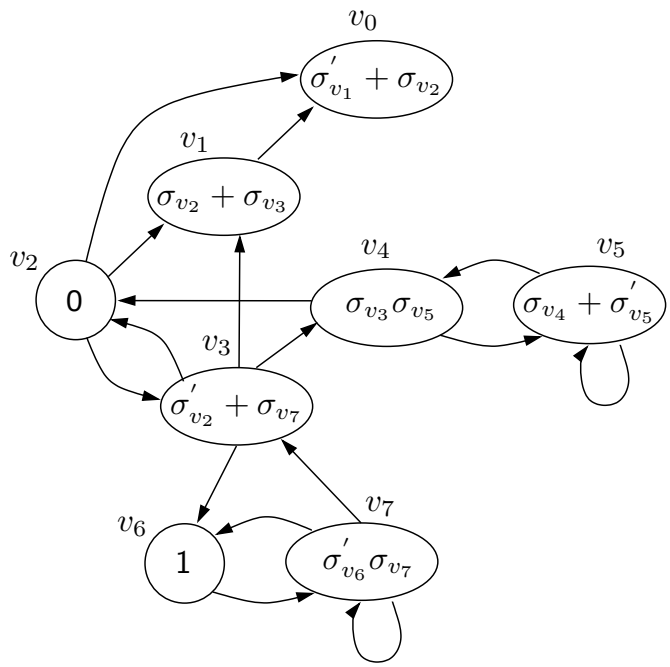


Figure G.5: Example of a network implementing the 2-input AND.

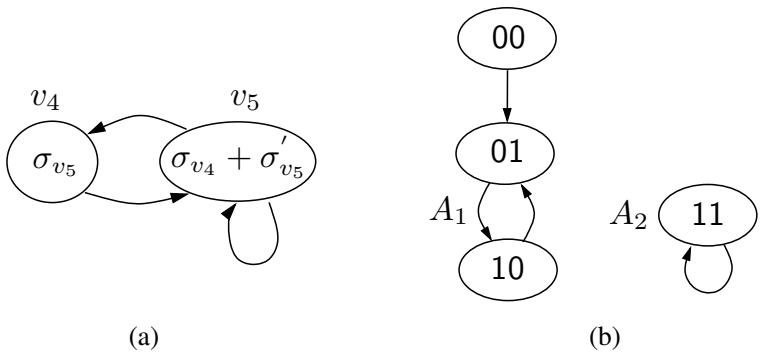


Figure G.6: (a) Reduced network for the Kauffman network in Figure G.5. (b) Its state transition graph. Each state is a pair $(\sigma(v_4)\sigma(v_5))$. There are two attractors: $A_1 = \{01, 10\}$ and $A_2 = \{11\}$.

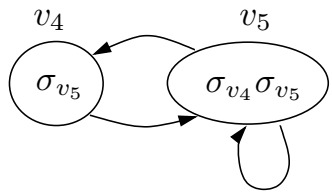


Figure G.7: An alternative reduced network for the 2-input AND.

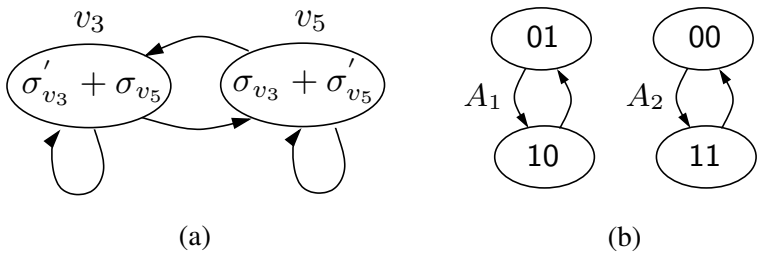


Figure G.8: (a) Reduced network for the Kauffman network in Figure G.5, after three mutation described in Section G.7 has been applied. (b) Its state transition graph. Each state is a pair $(\sigma(v_3)\sigma(v_5))$. There are two attractors: $A_1 = \{01, 10\}$ and $A_2 = \{00, 11\}$.

- a predecessor of a vertex v is changed, i.e. the edge (u, v) is replaced by an edge (w, v), $v, u, w \in V$;
- the state of a vertex is changed to the complemented value;
- Boolean function of a vertex is changed to a different Boolean function.

On one hand, the stability of Kauffman networks is due to the large percentage of redundancy in the network. $\Theta(n - \sqrt{n})$ of n vertices are typically redundant. On the other hand, the stability is due to the non-uniqueness of the network representation. The same dynamic behavior can be achieved by many different Kauffman networks. For instance, the 2-input AND gate could be implemented in many other ways than the one shown in Figure G.5. For example, the reduced network in Figure G.7 has the same state transition graph as the one in Figure G.6.

Evolvability

An essential feature of living organisms is their capability to adapt to a changing environment. Kauffman networks have been shown to be successful in evolving to a predefined target function.

As an example, suppose that the following three mutations are applied to the network in Figure G.5:

1. edge (v_4, v_5) is replaced by (v_3, v_5);
2. edge (v_2, v_3) is replaced by (v_3, v_3);
3. edge (v_7, v_3) is replaced by (v_5, v_3).

After removing redundant vertices from the resulting modified network, we obtain the reduced network shown in Figure G.8. Its state space has two attractors, A_1 and A_2. If we assign the logic 0 to A_1 and the logic 1 to A_2, then the initial states 00 and 11 terminate in 1, while 01 and 10 terminate in 0. So, the modified network implements the 2-input Boolean XNOR.

The example given above is intended to demonstrate that an evolution from one functionality to another is possible.

G.8 Conclusion and Future Work

This paper presents a set of algorithms for the analysis of Kauffman networks. Redundancy removal and partitioning algorithms have been presented previously in [66, 61, 65]. The algorithm for computing attractors is a new contribution, as well as the proposed applications.

We would like to stress that the major challenge is the *size* of the networks we are targeting. Small Kauffman networks are of theoretical interest only. They cannot adequately model gene interactions of living cells. We aim at developing a practical software package, applicable to real world size problems.

A software package that can model gene interactions is of primary importance to biology and medicine. Such a package will provide a framework for obtaining simulation results that can be independently evaluated by *in vivo* experiments. It can be used for various purposes, including:

1. to study the effects of diseases, mutations, or injuries on a cell;

2. to infer gene interactions that produce abnormal cells, e.g. cancer;

3. to understand the process of aging of a cell over time.

In the future, we will also investigate possibilities for enhancing Kauffman networks as a model. Kauffman networks have a number of drawbacks. First, input connectivity of gene regulatory networks is much higher than $k = 2$. For example, it is more than 20 in β-globine gene of humans and more than 60 for the platelet-derived growth factor β receptor [4]. We will consider networks with a higher input connectivity k and a smaller probability p, satisfying the equation (G.1).

Second, using Boolean functions for describing the rules of regulatory interactions between the genes seems too simplistic. It is known that the level of gene expression depends on the presence of activating or repressing proteins. However, the *absence* of a protein can also influence the gene expression [4]. Using multiple-valued functions instead of Boolean ones for representing the rules of regulations could be a better option.

Third, the number of attractors in Kauffman networks is a function of the number of vertices. However, organisms with a similar number of genes may have different numbers of cell types. For example, humans have 20.000-25.000 genes and more than 250 cell types [106]. The flower Arabidopis has a similar number of genes, 25.498, but only about 40 cell types [21]. We will investigate which other factors influence the number of attractors.

As a longer-term goal, we will attempt to develop a computing scheme based on the principles of gene interactions. A living cell is, essentially, a molecular computer that configures itself as part of the execution of its code. By understanding how genes interact with each other, we might find a way to build a novel type of computer chips. As silicon transistor technology approaches nano-meter dimensions and its speed and integration slow down, the need for new ways of computing becomes more and more evident.

Bibliography

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2000. ISBN 3-540-67282-6.

[2] S. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[3] B. Alberts, D. Bray, J. Lewis, M. Ra, K. Roberts, and J. D. Watson. *Molecular Biology of the Cell*. Garland Publishing, New York, 1994.

[4] M. Aldana, S. Coopersmith, and L. P. Kadanoff. Boolean dynamics with random couplings. http://arXiv.org/abs/adap-org/9305001.

[5] S. Alstrup, J. Clausen, and K. Jorgensen. An $O(|v| * |e|)$ algorithm for finding immediate multiple-vertex dominators. *Information Processing Letters*, 59(1):9–11, 1996.

[6] I. Amlani, A. O. Orlov, G. Toth, G. H. Bernstein, C. S. Lent, and G. L. Snider. Digital logic gate using quantum-dot cellular automata. *Science*, 284:289–291, 1999.

[7] R. Ashenhurst. The decomposition of switching functions. In *Proceedings International Symp. Theory of Switching*, volume 29, pages 74–116, 1959.

[8] H. Atlan, F. Fogelman-Soulie, J. Salomon, and G. Weisbuch. Random Boolean networks. *Cybernetics and System*, 12:103–121, 2001.

[9] K. Bartlett, W. Cohen, A. de Geus, and G. Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Trans-*

actions on Computer-Aided Design of Integrated Circuits and Systems, 5(4):582–596, October 1986.

[10] U. Bastola and G. Parisi. The critical line of Kauffman networks. *J. Theor. Biol.*, 187:117, 1997.

[11] U. Bastola and G. Parisi. The modular structure of Kauffman networks. *Phys. D*, 115:219, 1998.

[12] U. Bastola and G. Parisi. Relevant elements, magnetization and dynamic properties in Kauffman networks: a numerical study. *Physica D*, 115:203, 1998.

[13] T. Bengtsson, A. Martinelli, and E. Dubrova. A BDD-based fast heuristic algorithm for disjoint decomposition. In *Proceedings of Asia and South Pacific Design Automation Conference, ASP-DAC03*, pages 191–196, Kitakyushu, Japan, January 2003.

[14] M. Berkelaar and K. M. van Eijk. Efficient and effective redundancy removal for million-gate circuits. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, page 1088. IEEE Computer Society Press, 2002.

[15] V. Bertacco and M. Damiani. The disjunctive decomposition of logic functions. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 78–82, 1997.

[16] V. Bertacco, S. Minato, P. Verplaetse, L. Benini, Micheli, and G. De. Decision diagrams and pass transistor logic synthesis. Technical Report CSL-TR-97-748, Stanford, CA, USA, 1997.

[17] B. B. Bhattacharya and S. C. Seth. On the reconvergent structure of combinational circuits with applications to compact testing. In *Proceeding of International Symposium on Fault-Tolerant Computing*, pages 264–269, 1987.

[18] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, Amsterdam, The Netherlands, March 1999.

[19] S. Bilke and F. Sjunnesson. Stability of the Kauffman model. *Physical Review E*, 65:016129, 2001.

[20] L. J. Billera. On the composition and decomposition of clutters. *Journal of Comb. Theory*, 11:234–241, 1971.

[21] K. D. Birnbaum, D. E. Shasha, J. Y. Wang, J. W. Jung, G. M. Lambert, D. W. Galbraith, and P. N. Benfey. A global view of cellular identity in the arabidopsis root. In *Proceedings of the International Conference on Arabidopsis Research*, Berlin, Germany, July 2004.

[22] Z. W. Birnbaum and J. D. Esary. Modules of coherent binary systems. *SIAM Journal of Applied Math.*, 13:444–451, 1965.

[23] Z. W. Birnbaum and R. H. Möhring. A fast algorithm for the decomposition of graphs and posets. *Math. Oper. Res*, pages 170–177, 1984.

[24] P. Bjesse. DAG-aware circuit compression for formal verification. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 42–49, November 2004.

[25] B. Bollig and I. Wegner. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1006, 1996.

[26] M. Bolton. *Digital Systems Design with Programmable Logic*. Addison-Wesley Pub. Co., 1990.

[27] G. Boole. *The laws of thought*. Prometeus Books, New York, 2003. ISBN 1-59102-089-1. Originally published: An investigation of the laws of thought. 1854.

[28] S. Bornholdt and T. Rohlf. Topological evolution of dynamical networks: Global criticality from local dynamics. *Physical Review Letters*, 84:6114–6117, 2000.

[29] D. Bostick and G. D. Hachtel. The Boulder optimal logic design system. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 62–65, November 1987.

[30] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proc. 27th Design Automation Conference*, pages 37–111, 1990.

[31] R. S. Braich, N. Chelyapov, C. Johnson, P. W. K. Rothemund, L. Adleman, R. P. Cowburn, and M. E. Welland. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296:499–502, 2002.

[32] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangle covering problem. *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 66–69, November 1987.

[33] R. K. Brayton and C. McMullen. The decomposition and factorization of Boolean expression. In *Proceedings of the IEEE International Symposium of Circuits and Systems*, pages 49–54. IEEE, 1982.

[34] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6): 1062–1081, November 1987.

[35] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[36] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986. ISSN 0018-9340.

[37] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[38] P. Buch, A. Narayan, A. R. Newton, and A. Sangiovanni-Vincentelli. Logic synthesis for large pass transistor circuits. In *p-ICCAD*, pages 663–670, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8200-0.

[39] J.R. Burch, E.M. Clarke, D. E. Long, K.L. McMillan, and D.L. Dill. Symbolic Model Checking for sequential circuit verification. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–442, April 1994.

[40] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[41] S.-C. Chang, M. Marek-Sadowska, and T. Hwang. Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15:1226–1235, 1996.

[42] S. Chattopadhyay, S. Roy, and P. P. Chaudhuri. KGPMIN: an efficient multilevel multioutput AND-OR-XOR minimizer. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 16(3):257–265, March 1997.

[43] D. Cheng. Power estimation of digital CMOS circuits and the application to logic synthesis for low power, December 1995. Ph.D. Thesis, University of California at Santa Barbara.

[44] J. Cong, H. P. Li, S. K. Lim, Toshiyuki Shibuya, and Dongmin Xu. Large scale circuit partitioning with loose/stable net removal and signal flow based clustering. In *International Conference on Computer-Aided Design*, pages 441–446, 1997.

[45] J. Cortadella. Bi-decomposition and tree-height reduction for timing optimization. In *Proceedings of the ACM/IEEE International Workshop on Logic Synthesis*, New Orleans, July 2002. ACM/IEEE.

[46] R. P. Cowburn and M. E. Welland. Room temperature magnetic quantum cellular automata. *Science*, 287:1466–1468, 2000.

[47] W. H. Cunningham. Decomposition of directed graphs. *SIAM Journal of Algebraic and Discrete Methods*, 3:214–221, 1982.

[48] H. A. Curtis. *A New Approach to the Design of Switching Circuits*. D. van Nostrand company, Princeton, New Jersey, 1962.

[49] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan. LSS: Logic synthesis through local transformations. *IBM Journal of Research and Development*, 25(4):272–280, July 1981.

[50] E. S. Davidson. An algorithm for NAND decomposition under network constraints. *IEEE Transactions on Computers*, C-18(12):1098–1109, December 1969.

[51] R. Dawkins. *The Selfish Gene*. Oxford University Press, Oxford, 1989.

[52] B. Derrida and H. Flyvbjerg. Multivalley structure in Kauffman's model: Analogy with spin glass. *J. Phys. A: Math. Gen.*, 19:L1103, 1986.

[53] B. Derrida and H. Flyvbjerg. Distribution of local magnetizations in random networks of automata. *J. Phys. A: Math. Gen.*, 20:L1107, 1987.

[54] B. Derrida and Y. Pomeau. Random networks of automata: a simple annealed approximation. *Biophys. Lett.*, 1:45, 1986.

[55] J-P. Deschamps. Binary simple decomposition of discrete functions. *Digital Processes*, 1:123–130, 1975.

[56] W. E. Donath and H. Ofek. Automatic identification of equivalence points for Boolean logic verification. *IBM Technical Disclosure Bulletin*, 18(8):2700–2703, 1976.

[57] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. In *Proceedings of the 31st annual conference on Design automation*, pages 415–419, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-653-0.

[58] K. E. Drexler. *Nanosystems*. Wiley, New York, 1992.

[59] E. Dubrova. Composition trees in finding best variable orderings for ROBDDs. In *Proceedings of Design, Automation & Test in Europe Conference*, page 1084, 2002.

[60] E. Dubrova. *Logic Synthesis and Verification*, chapter 4. Kluwer Academic Publishers, 2002.

[61] E. Dubrova. Modeling of gene regulatory systems by random Boolean networks. In *Bioengineered and Bioinspired Systems*, Sevilla, Spain, 9-11 May 2005.

[62] E. Dubrova and L. Macchiarulo. A comment on graph-based algorithm for Boolean manipulation. *IEEE Transactions on Computers*, 49(10): 1290–1292, October 2000.

[63] E. Dubrova, D. Miller, and J. Muzio. AOXMIN: A three-level heuristic AND-OR-XOR minimizer for Boolean functions. In *Proceedings of the 3rd International Workshop on the Applications of the Reed-Muller Expansion in Circuit Design*, page 209, 1997.

[64] E. Dubrova and D. M. Miller. On dependable criteria for dynamic reordering algorithms. In *Proc. 7th Int. Workshop on Post-Binary ULSI Systems*, pages 46–48, 1998.

[65] E. Dubrova and M. Teslenko. Compositional properties of Random Boolean Networks. *Physical Review E*, 71, May 2005.

[66] E. Dubrova, M. Teslenko, and H. Tenhunen. Computing attractors in dynamic networks. In *Proceedings of International Symposium on Applied Computing (IADIS'2005)*, pages 535–543, Algarve, Portugal, February 2005.

[67] E. V. Dubrova, C. Muzio, and B. von Stengel. Finding composition trees for multiple-valued functions. In *Proceedings of 27th International Symposium on Multiple-Valued Logic*, pages 19–26. IEEE, 1997.

[68] D. M. Eigler, C. P. Lutz, and W. E. Rudge. An atomic switch realized with the scanning tunnelling microscope. *Nature*, 352:600–602, 1991.

[69] G. Fleisher and L. Maissel. An introduction to array logic. *IBM Journal of Research and Development*, 19:98–109, March 1975.

[70] H. Flyvbjerg and N. J. Kjaer. Exact solution of Kauffman model with connectivity one. *J. Phys. A: Math. Gen.*, 21:1695, 1988.

[71] A. A. Fraenkel. *Abstract Set Theory.* North-Holland Publishing, Amsterdam, 1976.

[72] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Proc. 24th ACM/IEEE Design Automation Conf.*, pages 348–355, 1987.

[73] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54, Amsterdam, February 1991.

[74] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Computer Aided Verification (CAV'94)*, pages 299–310, Stanford, July 1994. Springer-Verlag.

[75] J. F. Gimpel. The minimization of TANT networks. *IEEE Transactions on Electronic Computers*, EC-16(1):18–38, February 1967.

[76] D. Goldhaber-Gordon, M. S. Montemerlo, J. C. Love, G. J. Opiteck, and J. C. Ellenbogen. Overview of nanoelectronic devices. *Proc. IEEE*, 85:521–540, 1997.

[77] R. Gupta. Generalized dominators and post-dominators. In *Proceedings of 19th Annual ACM Symposium on Principles of Programming Languages*, pages 246–257, 1992.

[78] M. Habib and M. C. Maurer. On the x-join decomposition for undirected graphs. *Journal of Appl. Discr. Math.*, 3:198–205, 1979.

[79] P. Halmos. *Naive set theory.* Springer-Verlag, New York, 1974. ISBN 0-387-90092-6.

[80] A. J. Heinrich, C. P. Lutz, J. A. Gupta, and D. M. Eigler. Molecule cascades. *Science*, 298:1381–1387, 2002.

[81] L. Hellermann. A catalog of three-variable OR-invert and AND-invert logical circuits. *IEEE Transactions on Electronic Computers*, EC-12: 198–223, June 1963.

[82] J.-D Huang, J.-Y Jou, and W.-Z. Shen. Encoding in Roth-Karp decomposition with application to two-output LUT architecture. In *Computers and Digital Techniques, IEE Proceedings, Vol.146, Iss.3*, pages 131–138. IEE, 1999.

[83] S. Huang and D. E. Ingber. Shape-dependent control of cell growth, differentiation, and apoptosis: Switching between attractors in cell regulatory networks. *Experimental Cell Research*, 261:91–103, 2000.

[84] Y. Huang, X. Duan, Y. Cui, L. Lauhon, K. Kim, and C. M. Lieber. Logic gates and computation from assembled nanowire building blocks. *Science*, 294:1313–1317, 2001.

[85] J. Ishikawa, H. Sato, M. Hiramine, K. Ishida, S. Oguri, Y. Kazuma, and S. Murai. A rule-based reorganization system LORES/EX. *Proc. International Conference on Computer Design*, pages 262–266, October 1988.

[86] F. Jacob and J. Monod. Genetic regulatory mechanisms in the synthesis of proteins. *Journal of Molecular Biology*, 3:318–356, 1961.

[87] S.-W. Jeong. *Binary Decision Diagrams and their Applications to Implicit Enumeration Techniques in Logic Synthesis*. PhD thesis, University of Colorado, 1992.

[88] S.-W. Jeong, B. Plessier, G. D. Hatchel, and F. Somenzi. Variable ordering and selection for SSM traversal. In *Proceedings of the IEEE Int. Conf. on Computer Aided Design*, pages 476–479, 1991.

[89] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[90] R. M. Karp. Functional decomposition and switching circuit design. *Journal of Soc. Indust. Appl. Math.*, 11(2):291–335, June 1963.

[91] K. Karplus. Using If-Then-Else DAGs for multi-level logic minimization. Technical Report UCSC-CRL-88-29, University of California Santa Cruz, 1988.

[92] S. A. Kauffman. Metabolic stability and epigenesis in randomly constructed nets. *Journal of Theoretical Biology*, 22:437–467, 1969.

[93] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection of Evolution*. Oxford University Press, Oxford, 1993.

[94] S. A. Kauffman and E. D. Weinberger. The NK model of rugged fitness landscapes and its application to maturation of the immune response. *Journal of Theoretical Biology*, 141:211–245, 1989.

[95] R. Krenz and E. Dubrova. On-the-fly proper cut recognition based on circuit graph analysis. In *Proceedings of NORCHIP'02*, Copenhagen, Denmark, November 2002. poster.

[96] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the 38th ACM/IEEE Design Automation Conference*, pages 232–237, Las Vegas, Nevada, June 2001.

[97] A. Kuehlmann, M.K. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the 38th ACM/IEEEDesign Automation Conference*, pages 232–237, Las Vegas, NV, June 2001. IEEE Computer Society Press.

[98] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula. BDD based decomposition of logic functions with application to FPGA synthesis. In *p-DAC*, pages 642–647, 1993.

[99] Yung-Te Lai, K.-R.R. Pan, and M. Pedram. BDD-based function decomposition: algorithms and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15: 977–990, 1996.

[100] E. L. Lawler. An approach to multilevel Boolean minimization. *Journal of the ACM*, 11(3):283–295, July 1964.

[101] C. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38(4):985–999, 1959.

[102] C. Legl, B. Wurth, and K. Eckl. Computing support-minimal subfunctions during functional decomposition. *Transactions on Very Large Scale Integration (VLSI) systems*, 6(3):354–363, September 1998.

[103] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *Transactions of Programming Languages and Systems*, 1(1):121–141, July 1979.

[104] C. S. Lent and P. D. Tougaw. A device architecture for computing with quantum dots. *Proc. IEEE*, 85:541–557, 1997.

[105] R. J. Lipton. Dna solution of hard computational problem. *Science*, 268:542–545, 1995.

[106] A. Y. Liu and L. D. True. Characterization of prostate cell types by CD cell surface molecules. *The American Journal of Pathology*, 160: 37–43, 2002.

[107] T.-H. Liu, M. K. Ganai, A. Aziz, and J. L. Burns. Performance driven synthesis for pass-transistor logic. In *VLSID '99: Proceedings of the 12th International Conference on VLSI Design*, pages 372–377, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0013-7.

[108] E. S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[109] B. Luque and R. V. Sole. Stable core and chaos control in Random Boolean Networks. *Journal of Physics A: Mathematical and General*, 31:1533–1537, 1998.

[110] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. International Conference on Computer-Aided Design*, pages 6–9, 1988.

[111] A. Martinelli, T. Bengtsson, E. Dubrova, and A. J. Sullivan. Roth-Karp decomposition of large Boolean functions with application to logic design. In *Proceedings of NORCHIP'02*, Copenhagen, Denmark, November 2002.

[112] H. Mathony and U. G. Baitinger. CARLOS: An automated multilevel logic design system for CMOS semi-custom integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(3):346–355, March 1988.

[113] Y. Matsunaga. An exact and efficient algorithm for disjunctive decomposition. In *Proceedings of SASIMI'98*, pages 44–50, 1998.

[114] E. J. McCluskey. Minimization of Boolean functions. *Bell System Technical Journal*, 35:1417–1444, 1959.

[115] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[116] J. D. Meindl, Q. Chen, and J. A. Davis. Limits on silicon nanoelectronics for terascale integration. *Science*, 293:2044–2049, 2001.

[117] S. Minato. Minimum-width method of variable ordering for binary decision diagrams. *IEICE Trans. Fundamentals*, E-75-A(3):392–399, 1992.

[118] S. Minato and G. De Micheli. Finding all simple disjunctive decompositions using irredundant sum-of-products forms. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 111–117, 1998.

[119] A. Mishchenko, B. Steinbach, and M. Perkowski. An algorithm for bi-decomposition of logic functions. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 103–108. IEEE, 2001.

[120] R. H. Möhring. Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and Boolean functions. *Annals of Operations Research*, 4:195–225, June 1985.

[121] R. H. Möhring and F. J. Radermacher. Substitution decomposition of discrete structures and connections to combinatorial optimization. *Ann. Discrete Math*, 19:257–264, 1984.

[122] D. Möller, P. Molitor, and R. Drechsler. Symmetry based variable ordering for ROBDDs. In *IFIP Workshop on Logic and Architecture Synthesis*, 1994.

[123] M. Molloy and B. Reed. The size of the giant component of a random graph with a given degree sequence. *Combin. Probab. Comput.*, 7: 295–305, 1998.

[124] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimum functional decomposition using encoding. In *p-DAC*, pages 408–414, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-653-0.

[125] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of IEE/ACM Workshop on Logic Synthesis*, pages 1–10, 1995.

[126] W. Paul. Realizing Boolean functions on disjoint sets of variables. *Theoretical Computer Science*, 2:383–396, 1976.

[127] W. Van Orman Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59(8):521–531, October 1952.

[128] S. M. Reddy, W. Kunz, and D. K. Pradhan. Novel verification framework combining structural and OBDD methods in a synthesis environment. In *Proceedings of the 32th ACM/IEEE Design Automation Conference*, pages 414–419, San Francisco, June 1995.

[129] V. G. Redko. Kauffman's nk Boolean networks, 1998. http://pespmc1.vub.ac.be/BOOLNETW. html.

[130] J. P. Roth and R. M. Karp. Minimization over Boolean graphs. *IBM Journal*, 6:227–238, April 1962.

[131] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceeding of IEEE/ACM International Conference on Computer-Aided Design*, volume 29, pages 42–47, 1993.

[132] T. Sasao. *FPGA design by generalized functional decomposition*, pages 233–258. Kluwer Academic Publishers, 1993.

[133] T. Sasao and M. Matsuura. DECOMPOS: An integrated system for functional decomposition. In *Proceedings of the ACM/IEEE International Workshop on Logic Synthesis*, 1998.

[134] H. Sawada, T. Suyama, and A. Nagoya. Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 355–358. IEEE, 1995.

[135] H. Sawada, S. Yamashita, and A. Nagoya. Restructuring logic representations with easily detectable simple disjunctive decompositions. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 755–759. IEEE, 1998.

[136] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of California Berkley, May 1992.

[137] S. C. Seth, L. Pan, and V. D. Agrawal. PREDICT-probabilistic estimation of digital circuit testability. In *Proceeding of International Symposium on Fault-Tolerant Computing*, pages 220–225, June 1985.

[138] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical J.*, 28:59–98, January 1949.

[139] L. S. Shapley. Solutions of compound simple games. In *Advances in Game Theory*, number 52 in Ann. of Math. Study, pages 267–280. Princeton University Press, 1964.

[140] J. Shelley. Here we go again, 29 December 2004. http://www.gdnctr.com/dec_29_00.htm.

[141] V. Y. Shen and A. C. McKellar. An algorithm for the disjunctive decomposition of switching functions. *IEEE Trans. Computers*, C-19: 239–245, 1970.

[142] V. Y. Shen, A. C. McKellar, and P. Weiner. A fast algorithm for the disjunctive decomposition of switching functions. *IEEE Trans. Computers*, C-20:239–246, 1970.

[143] J. E. S. Socolar and S. A. Kauffman. Scaling in ordered and critical random Boolean networks. http://arXiv.org/abs/cond-mat/0212306.

[144] F. Somenzi. *CU Decision Diagram Package, Release 2.3.0.* University of Colorado at Boulder, 1998.

[145] Z. Somogyvari and S. Payrits. Length of state cycles of random boolean networks: an analytic study. *Journal of Physics A: Mathematical and General*, 33:6699–6706, 2000.

[146] T. Stanion and C. Sechen. Quasi-algebraic decompositions of switching functions. In *Proceedings of Sixteenth Conference on Advanced Research in VLSI*, pages 358–367. IEEE, 1995.

[147] J. Suurkula. Over 95 percent of DNA has largely unknown function, 2004. http://www.psrast.org/junkdna.htm.

[148] D. D. Swade. Redeeming charles babbage's mechanical computer. *Scientific American*, 268(2):62–68, 1993.

[149] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[150] A. Thayse. A fast algorithm for the proper decomposition of Boolean functions. *Philips Res. Rep.*, 27:140–147, 1972.

[151] C.-C. Tsai and M. Marek-Sadowska. Multilevel logic synthesis for arithmetic functions. In *p-DAC*, pages 68–73. IEEE, 1996.

[152] G. Y. Tseng and J. C. Ellenbogen. Nanotechnology: Enhanced: Toward nanocomputers. *Science*, 294:1293–1294, 2001.

[153] C. A. J. van Eijk and J. A. G. Jess. Detection of equivalent state variables in finite state machine verification. In *1995 ACM/IEEE International Workshop on Logic Synthesis*, pages 3–35 – 3–44, Tahoe City, CA, May 1995.

[154] B. von Stengel. Eine dekompositionstheorie für mehrstellige funktionen. In *Mathematical Systems in Economics*, volume 123. Anton Hain, Frankfurt, 1991.

[155] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification (CAV'00)*, pages 125–138, Chicago, IL, July 2000. Springer-Verlag.

[156] S. J. Wind, J. Appenzeller, R. Martel, and V. Derycke. Vertical scaling of single-wall carbon nanotube cmos field effect transistors using top gate electrodes. *Appl. Phys. Lett.*, 80:3817, 2002.

[157] A. Wuensche. The DDlab manual, 2000. http://www.cogs.susx.ac.uk/users/andywu/man_ contents.html.

[158] B. Wurth, K. Eckl, and K. Antreich. Functional multiple-output decomposition: Theory and an implicit algorithm. In *p-DAC*, pages 54–59, 1995.

[159] S. Yamashita, H. Sawada, and A. Nagoya. New methods to find optimal non-disjoint bi-decompositions. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 59–68. IEEE, 1998.

[160] C. Yang, M. Ciesielski, and V. Singhal. BDS: a BDD-based logic optimization system. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 92–97. IEEE, 2000.

[161] C. Yang, V. Singhal, and M. Ciesielski. BDD decomposition for efficient logic synthesis. In *Proceedings of International Conference on Computer Design*, pages 626–631, 1999.

[162] Y. Ye and K. Roy. A graph-based synthesis algorithm for AND/XOR networks. In *Proceedings of the 34th annual conference on Design automation*, pages 107–112, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-920-3.

[163] D. Zampunièris. *The Sharing Tree Data Structure*. PhD thesis, Department of Computer Science, University of Namur, Belgium, 1997.

Index