Math. Struct. in Comp. Science (2008), *vol.* 18, *pp.* 411–426. © 2008 Cambridge University Press
doi:10.1017/S096012950800666X Printed in the United Kingdom

The Decomposition Tree for analyses of Boolean functions

MAIK FRIEDEL[†], SWETLANA NIKOLAJEWA[‡] and
THOMAS WILHELM[§]

[†]*Biocomputing Group, Fritz Lipmann Institute (FLI), Beutenbergstr. 11, D-07745 Jena, Germany*
Email: maikfr@fli-leibniz.de
[‡]*Department of Bioinformatics, Friedrich Schiller University Jena, Ernst-Abbe-Platz 2, D-07743 Jena, Germany*
[§]*Theoretical Systems Biology, Institute of Food Research, Norwich Research Park, Norwich NR4 7UH, United Kingdom*
Email: Thomas.Wilhelm@bbsrc.ac.uk

Received 17 October 2006; revised 30 October 2007

We present a new data structure, called a Decomposition Tree (DT), for analysing Boolean functions, and demonstrate a variety of applications. In each node of the DT, appropriate bit-string decomposition fragments are combined by a logical operator. The DT has 2^k nodes in the worst case, which implies exponential complexity for problems where the whole tree has to be considered. However, it is important to note that many problems are simpler. We show that these can be handled in an efficient way using the DT. Nevertheless, many problems are of exponential complexity and cannot be made any simpler: for example, the calculation of prime implicants. Using our general DT structure, we present a new worst case algorithm to compute all prime implicants. This algorithm has a lower time complexity than the well-known Quine–McCluskey algorithm and is the fastest corresponding worst case algorithm so far.

1. Introduction

Boolean functions (BFs) have widespread applications in nearly all fields of science and engineering. The Reduced Ordered Binary Decision Diagram (ROBDD), which is obtained from the Binary Decision Diagram (BDD), is probably the most powerful data structure known to date for the manipulation of large logic functions (Bryant 1992). It provides a compact representation of Boolean expressions, and there are efficient algorithms for performing all kinds of logical operations on ROBDDs (Andersen 1997). However, the ROBDD construction itself is costly because it is based on the isomorphism of nodes and subgraphs.

In this paper we present a new data structure for BFs, called a Decomposition Tree (DT). As with BDDs and ROBDDs, it is based on the general idea of a Shannon

[†] Supported by Grant 0312704E of the Bundesministerium für Bildung und Forschung.

expansion. However, BDDs and ROBDDs always show a variable ordering (which is, in principle, arbitrary), and the size of a BDD and ROBDD for a given BF has a strong dependence on the chosen order. The problem of finding the best variable ordering is NP-hard (Bollig and Wegener 1996). In contrast to BDDs and ROBDDs, our DT is completely symmetric in the sense that it represents all variables equivalently. The DT provides a unified approach to tackling many different BF problems.

For a given function with k variables, we take as input the truth table, which is represented by a bit-string of length 2^k. In all our calculations, appropriate bit-string decomposition fragments are combined using a logical operator (in each node of the DT). The use of different operators leads to different applications of the general DT structure, such as a general classification of a given BF. For many problems, we only need functions of a particular subclass. In molecular biology, for instance, gene regulatory networks are simulated with canalysing (Kauffman 2000) and hierarchically canalysing Boolean functions (Szallasi and Liang 1998). Such functions have also been used to study such diverse problems as decision structures in social systems (Klüver and Schmidt 1999), the convergence behaviour of non-linear filters (Shmulevich *et al.* 2004) and artificial life (Kleer *et al.* 1993). Monotonic functions play a special role in game theory, computational learning, harmonic analysis and signal processing. Non-linear functions are essential for cryptographic transformations (Hirose and Ikeda 1994; Preneel *et al.* 1991). Functions with unate properties are used in the design of conventional cryptosystems (Hirose and Ikeda 1994), and functions with special symmetry characteristics are important for circuit restructuring (Jeong *et al.* 1993).

Each Boolean function can be represented by its disjunctive normal form (DNF). A lot of BF research has been devoted to minimal DNFs (Clote and Kranakis 2002; Strzemecki 1992; Wang *et al.* 2001; Wegener 2000; Wegener 1987). The generation of prime implicants (PIs) of a given function is an important first step in calculating its minimal DNF, and early interest in PIs (Quine 1952) was mainly inspired by this problem. Meanwhile, a variety of other applications has been found. PIs are used for alternative representations of Boolean expressions in various problems of artificial intelligence (Reiter and Kleer 1987), in the context of safety engineering to analyse fault trees (Dutuit and Rauzy 1997), to implement Assumption-Based Truth Maintenance, to characterise diagnoses, to compile formulas for Transcranial Magnetic Stimulation and to implement circumscription (Forbus and de Kleer 1992; Kleer *et al.* 1993). PIs play a role in expert system development to find all irredundant rules from a given rule system and in Electronic Design Automation (Crama and Hammer 2006). We show that one can simply generate all PIs of a given BF using our Decomposition Tree and the AND-operator for the manipulation of the appropriate bit-strings.

In the first part of the paper (Section 2), we introduce the Decomposition Tree and in the second part (Section 3), we discuss different applications. Section 3.1 demonstrates efficient ways to classify BFs. In Section 3.2, we present an efficient recursive algorithm to compute prime implicants. It is shown that our PI algorithm has a lower time complexity than the well-known algorithm of Quine and McCluskey (Coudert 1994; McCluskey 1956; Quine 1952) which also uses the truth table input format. Finally, we show how the DT can be used to construct the ROBDD of a given BF.

2. Decomposition Trees

Let $f : \{0,1\}^k \rightarrow \{0,1\}$ be a Boolean function on k variables. The Decomposition Tree is based on the Decomposition Set.

Definition 2.1 (D_i-decomposition). The D_i-decomposition of function f in input x_i is a segmentation of f into two functions f_0^i and f_1^i, which are defined by the positive and negative values of the input x_i:

$$D_i : \begin{array}{l} f_0^i = f(x_1, \ldots, x_{i-1}, 0, x_{i+1} \ldots, x_k) \\ f_1^i = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_k)\,. \end{array} \tag{1}$$

The bit-string representations of f_0^i,f_1^i with length 2^{k-1} are called the decomposition fragments of the D_i-decomposition.

The previous definition can be generalised to decompositions in more inputs.

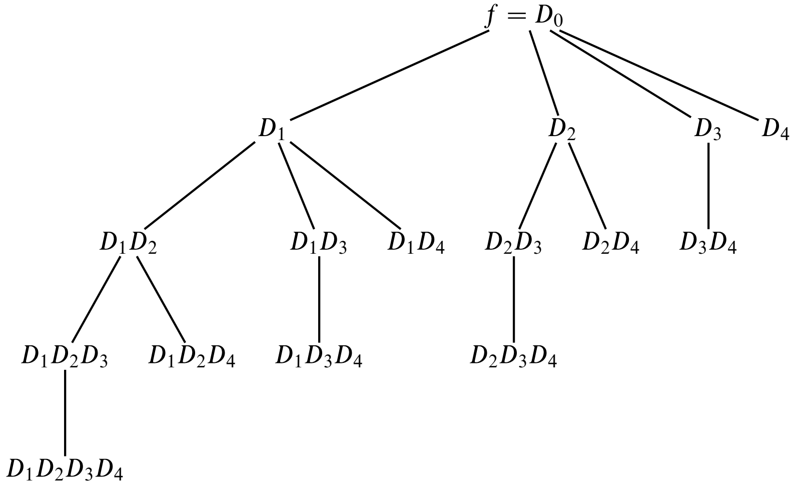
Definition 2.2 (D_iD_j-decomposition). Given the D_i- and D_j-decompositions, $i < j \in \{1,\ldots k\}$, the D_iD_j-decomposition is a combination of the D_i and D_j decompositions:

$$D_iD_j : \begin{array}{l} f_{00}^{ij} = f(x_1, ., \overset{i}{0}, ., \overset{j}{0}, ., x_k) \\ f_{01}^{ij} = f(x_1, ., \overset{i}{0}, ., \overset{j}{1}, ., x_k) \\ f_{10}^{ij} = f(x_1, ., \overset{i}{1}, ., \overset{j}{0}, ., x_k) \\ f_{11}^{ij} = f(x_1, ., \overset{i}{1}, ., \overset{j}{1}, ., x_k)\,. \end{array} \tag{2}$$

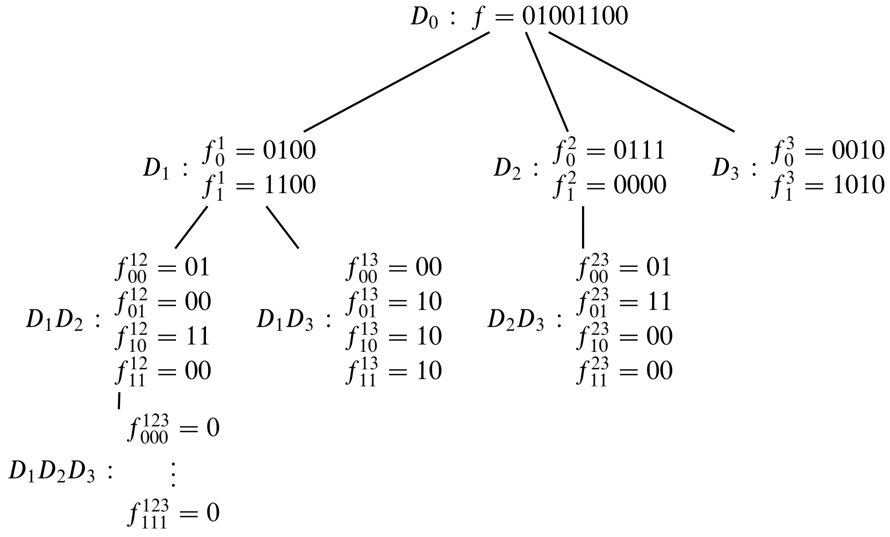
The decomposition can be extended to an arbitrary input combination of size $l \leqslant k$: $D_{i_1}D_{i_2}\ldots D_{i_l}$, $i_1 < i_2 < \ldots < i_l \in \{1,\ldots,k\}$. This has 2^l bit-string fragments of length 2^{k-l}. The Decomposition Set D contains 2^k possible decompositions: $\{D_0$, D_1, D_2, $\ldots$, D_k, D_1D_2, $\ldots$, $D_1D_2D_3\ldots D_k\}$, where $D_0 \equiv f$.

Definition 2.3 (Decomposition Tree). The Decomposition Tree (DT) is a rooted tree of all 2^k possible decompositions of a given function f on k inputs. A DT has k levels, where the root is defined to be in level 0. The root node is defined as decomposition D_0, which is equal to the function f. At level 1, all $D_i \in D$ are child nodes of D_0. Level l contains $\binom{k}{l}$ nodes $D_{i_1}D_{i_2}\ldots D_{i_l} \in D$. An edge between a parent node $P \in D$ at level l and a child node $C \in D$ at level $l+1$ exists if $P = \prod_{j=1}^{l} D_{i_j}$ and $C = P \cdot D_{i_{l+1}}$, where $i_1 < i_2 < \ldots < i_l < i_{l+1}$ and $i_j \in \{1 \ldots k\}$.

Example 2.1 (General Decomposition Tree for $k = 4$).



Example 2.2 (A specific Decomposition Tree ($k = 3$)). We decompose the function $x_1\bar{x}_2 \vee \bar{x}_2x_3$ with the bit-string 01001100 (truth table). The decomposition tree is:


$D_0 : f = 01001100$
$D_1 : f_0^1 = 0100$, $f_1^1 = 1100$
$D_2 : f_0^2 = 0111$, $f_1^2 = 0000$
$D_3 : f_0^3 = 0010$, $f_1^3 = 1010$
$D_1D_2 : f_{00}^{12} = 01$, $f_{01}^{12} = 00$, $f_{10}^{12} = 11$, $f_{11}^{12} = 00$
$D_1D_3 : f_{00}^{13} = 00$, $f_{01}^{13} = 10$, $f_{10}^{13} = 10$, $f_{11}^{13} = 10$
$D_2D_3 : f_{00}^{23} = 01$, $f_{01}^{23} = 11$, $f_{10}^{23} = 00$, $f_{11}^{23} = 00$
$D_1D_2D_3 : f_{000}^{123} = 0$, $\vdots$, $f_{111}^{123} = 0$

To detect a special pattern in a given Boolean function (for example, membership in the subclass of monotonic functions or all prime implicants) we combine the necessary decomposition fragments with a Boolean operation. For most applications the order of the 2^i bit-strings in nodes of level i is arbitrary, but sometimes it is important (*cf.* monotonic BFs, Section 3.1.1). The operator can be any logical operation, for instance AND, OR, XOR. The result of applying this operator to the decomposition fragments is a Boolean function, which we call *operator-combination ($\odot$-combination).*

Definition 2.4 ($\odot$-combination). Without loss of generality, given a decomposition $D_1D_2\dots D_l$ of function f. The $\odot$-combination is a Boolean function $g : \{0,1\}^{k-l} \rightarrow \{0,1\}$ defined by applying the $\odot$-operator $2^l - 1$ times to the decomposition fragments:

$$g(x_{l+1},..,x_k) = \underbrace{f^{12\dots l}_{00\dots 0} \odot f^{12\dots l}_{00\dots 1} \odot \dots \odot f^{12\dots l}_{11\dots 1}}_{2^l} . \tag{3}$$

This can also be written as

$$\begin{array}{rl} & f(0,0,\dots,0,x_{l+1},\dots,x_k) \\ \odot & f(0,0,\dots,1,x_{l+1},\dots,x_k) \\ & \dots \\ \odot & f(1,1,\dots,1,x_{l+1},\dots,x_k) \\ \hline = & g(x_{l+1},\dots,x_k) \end{array} \tag{4}$$

Using the DT and appropriate $\odot$-combinations, different properties of BFs can be detected. According to the concepts detailed below (Sections 3.1.3 and 3.1.5), the BF of Example 2.2 has

(i) two positive unates in x_1 and x_3 because fragments $f^1_0 \leqslant f^1_1$ and $f^3_0 \leqslant f^3_1$; and is

(ii) canalysing, because fragment $f^2_1 \equiv 0$, thus the function is canalysing (forcing) from 1 to 0 in input x_2, which means that it has the representation $f = \bar{x}_2 \wedge h(x_1,x_3)$, with $h(x_1,x_3) = x_1 \vee x_3$ (Nikolajewa *et al.* 2007).

Fortunately, one usually does not need to consider the whole tree (2^k nodes) to search for a special pattern. For instance, for the prime implicant calculation, the tree can be cut if the $\wedge$-combination gives the constant function $g = 0$ ($g = 1$) or a function containing only one true point, because all $\wedge$-combinations of the decompositions in the corresponding subtree will lead to constant functions (see Section 3.2). Furthermore, each node of the DT can be calculated independently of the other nodes. Therefore, for many problems, only tiny parts of the whole DT have to be considered. For instance, the class $x_1^{a_1}x_3^{a_3}$ where $x_i^{a_i} = \begin{cases} x_i, \text{ if } a_i=1 \\ \bar{x}_i, \text{ if } a_i=0 \end{cases}$ of implicants of a given BF with $k = 4$ can be detected by only calculating the node D_2D_4 (see Section 3.2). Another example are quadratic Boolean functions where only D_iD_j nodes have to be considered.

3. Applications

3.1. *Classification of Boolean functions*

There are five characteristic classes of BFs (Clote and Kranakis 2002): 0-preserving, 1-preserving, self-dual, monotonic and linear functions. The first three of these can be detected simply from the truth table. However, it is more difficult to decide whether a given BF is a monotonic or linear function. In the following we show how the classification problem can be solved using the DT for these two, as well as for other classes of BFs.

3.1.1. *Monotonic Boolean functions*

Definition 3.1. Let $a = (a_1, \ldots, a_k)$ and $b = (b_1, \ldots, b_k)$ be different k-element binary vectors. We say that a precedes b, denoted $a \prec b$, if $a_i \leqslant b_i$ for all $1 \leqslant i \leqslant k$. A Boolean function $f(x_1, \ldots, x_k)$ is said to be monotonic if for any two vectors a and b such that $a \prec b$, the relation $f(a) \leqslant f(b)$ holds.

Detection: If the first and last decomposition fragment of each node in the DT combined using the $\leqslant$-operator always gives the 1-constant function, then f is said to be monotonic.

Application: Monotonic functions are used in game theory, computational learning theory, harmonic analysis and signal processing (Bshouty and Tamon 1996; Makino and Ibaraki 1997; Shmulevich *et al.* 2004; Wendt *et al.* 1986). This is one of the characteristic classes (Clote and Kranakis 2002).

3.1.2. *Linear Boolean functions*

Definition 3.2. The Boolean function on k inputs is said to be linear if it can be represented as $f(x_1, x_2, \ldots, x_k) = a_0 \oplus a_1 x_1 \oplus \ldots \oplus a_k x_k$, where $a_i \in \{0, 1\}$ ($\oplus$ denotes XOR).

Detection: f is a linear function if the $\oplus$-operator applied to all the k pairs of decomposition fragments of the k nodes at the first level of the DT give constant functions. If the $\oplus$-combination of D_i is the 0-constant function, then $f(x_1, ., 0, ., x_k) = f(x_1, ., 1, ., x_k)$ (tautology in x_i), and thus coefficient $a_i = 0$. If the $\oplus$-combination of D_i is the 1-constant function, then $f(x_1, ., 0, ., x_k) = \bar{f}(x_1, ., 1, ., x_k)$ and f can be written as

$$f(x_1, ., x_k) = x_i \oplus g(x_1, .x_{i-1}, x_{i+1}, x_k),$$

since

$$\bar{x}_i f(x_1, ., 1, ., x_k) \wedge x_i \bar{f}(x_1, ., 1, ., x_k) = x_i \oplus f(x_1, ., 1, ., x_k).$$

Application: Linear functions are used in cryptography to establish non-linearity criteria for cryptographic transformations (Hirose and Ikeda 1994; Preneel *et al.* 1991). This is one of the characteristic classes (Clote and Kranakis 2002).

3.1.3. *Positive (negative) unate*

Definition 3.3. A BF has a positive (negative) unate in x_i if

$$\begin{aligned} f(x_1, \ldots, x_{i-1}, 1, x_{i+1} \ldots, x_k) &\geqslant f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_k) \\ (f(x_1, \ldots, x_{i-1}, 0, x_{i+1} \ldots, x_k) &\geqslant f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_k)). \end{aligned}$$

A BF is said to be unate if it is unate (positive or negative) in all variables.

Detection: If the corresponding $\geqslant$-combination of D_i is 1-constant, then x_i is positive (negative) unate. If this holds for all i, then f is unate. The function of Example 2.2 is unate.

Application: If f is positive unate in x_i, then $f(x) = x_i f(x_1,.,1,.x_k) \vee f(x_1,.,0,.x_k)$. This fact is used by the Unate Recursive Paradigm and is important in the design of conventional cryptosystems (Brayton *et al.* 1984; Brayton 1992; McGeer *et al.* 1993; Rudell 1985).

3.1.4. *Symmetry*

Definition 3.4. A BF is said to be partially symmetric if the function is not changed when a pair of variables is exchanged. A BF is totally symmetric if it does not change when any possible pair of variables is exchanged.

Detection: A function is totally symmetric if all decompositions at the first level are the same. For example, the totally symmetric function 01101000 ($x_1\bar{x}_2\bar{x}_3 \wedge \bar{x}_1\bar{x}_2x_3 \wedge \bar{x}_1x_2\bar{x}_3$) yields the same decomposition for all variables:

$$D_1 : \begin{matrix} 0110 \\ 1000 \end{matrix}, \qquad D_2 : \begin{matrix} 0110 \\ 1000 \end{matrix}, \qquad D_3 : \begin{matrix} 0110 \\ 1000 \end{matrix}.$$

If some but not all of the first level decompositions are the same, it is a partially symmetric function. Other types of symmetry can be detected by a comparison of decompositions corresponding to nodes at higher levels.

Application: Symmetric functions can be synthesised using fewer logic elements, so they play an important role in logic synthesis and functional verification. They are used for efficient circuit restructuring (Boyar *et al.* 2000; Chung and Liu 1998; Jeong *et al.* 1993). The detection of such symmetries is a popular field of research (Benini and de Micheli 1997; Mohnke and Malik 1993).

3.1.5. *Canalysing Boolean functions*

Definition 3.5. A Boolean function f on k variables is said to be a canalysing function if $\exists a, b \in \{0, 1\}$ and $\exists i \in \{1, \ldots, k\}$ such that $\forall x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k$

$$f(x_1, x_2, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_k) = b.$$

Detection: If there exists i such that the fragment f_a^i of a node in the first level of the DT is a b-constant function, then f is canalysing. For example, the function of Example 2.2 is canalysing. Moreover, the structure of the DT indicates how to extend the concept of canalysing single inputs to canalysing combinations of inputs. For example, in Example 2.2 the combination $\bar{x}_2x_3$ canalyses from 1 to 1.

Application: Canalysing functions have simplified logic expressions of the form

$$f(x) = x_i^{a_i} \odot g(x_1, ., x_{i-1}, x_{i+1}, ., x_k),$$

where $g \in \{0,1\}^{k-1}$, $\odot \in \{\vee, \wedge\}$. There are many applications in genetic network modeling (Kauffman 2000; Nikolajewa *et al.* 2007), artificial life (Kleer *et al.* 1993), non-linear digital filter design (Shmulevich *et al.* 2004) and sociology (Klüver and Schmidt 1999).

3.1.6. *Quadratic Boolean functions*

Definition 3.6. A function f is quadratic if the degree of the highest order term in the algebraic normal form is $\leqslant 2$ (Preneel *et al.* 1991).

Detection: Linear terms in the algebraic normal form can be detected if the corresponding $\oplus$-combination in D_i is a constant function (*cf.* the detection of linear functions). Similarly, quadratic terms in the algebraic normal form are obtained if the $\oplus$-combination in D_iD_j is a constant function: if the $\oplus$-combination in D_iD_j is 1-constant, then

$$f(x_1,\ldots,x_k) = x_i^{a_i} x_j^{a_j} \oplus g(x_1,.,x_{i-1},x_{i+1},.,x_{j-1},x_{j+1},.,x_k),$$

where g does not depend on x_i and x_j.

Application: Quadratic Boolean functions have applications in cryptography (Preneel *et al.* 1991).

3.2. *Computation of prime implicants*

An important step in the logic minimisation of a given Boolean function is the detection of the corresponding prime implicants. All implicants and prime implicants can be detected with the help of the $\wedge$-operation applied to each node in the DT.

Definition 3.7. An *implicant* of a BF f is a product term p in the sum of products (minterms) of f that implies f. p implies f means that f takes the value 1 whenever p equals 1. This means that p is fully covered by f: $p \leqslant f$. An implicant p of f is called *prime*, if it is not fully covered by any other implicant of f, that is, $p \nleqslant q$, for any other implicant q of f.

All implicants of a given function can be derived from true points of the $\wedge$-combination, which we call $\wedge$-patterns.

Example 3.1 ($\wedge$-combinations for $f(x_1,x_2,x_3)$=11011100).

$$D_1D_2 : \begin{array}{rcl} f^{12}_{00} & = & 11 \\ \wedge\, f^{12}_{01} & = & 01 \\ \wedge\, f^{12}_{10} & = & 11 \\ \wedge\, f^{12}_{11} & = & 00 \\ \hline g(x_3) & = & 00 \end{array} \qquad D_1D_3 : \begin{array}{rcl} & & 10 \\ & \wedge & 11 \\ & \wedge & 10 \\ & \wedge & 10 \\ \hline g(x_2) & = & 10 \end{array} \tag{5}$$

D_1D_2 has no $\wedge$-pattern since its $\wedge$-combination $g(x_3)$ is 0-constant. D_2D_3 and $D_1D_2D_3$ have no $\wedge$-pattern either. From the truth table representation of $g(x_2)$, it follows that D_1D_3 has the $\wedge$-pattern at the point $x_2 = 0$. We will show that this true point of the $\wedge$-combination leads to the implicant $p(x_2) = x_2^{a_2} = \bar{x}_2$.

Lemma 3.1 ($\wedge$-patterns imply implicants). If the $\wedge$-combination of a function f's $D_{i_1}D_{i_2}\dots D_{i_l}$-decomposition has the $\wedge$-pattern at point $(a_{l+1},\dots,a_k)$, then f contains the implicant

$$p = x_{i_{l+1}}^{a_{l+1}} \dots x_{i_k}^{a_k}$$

where

$$x_i^{a_i} = \begin{cases} x_i, & \text{if } a_i = 1 \\ \bar{x}_i, & \text{if } a_i = 0 \end{cases} .$$

Proof. Without loss of generality, given the D_1-decomposition of function f with an $\wedge$-pattern at the point $(a_2,\dots,a_k)$. We show that the product term $p(x_2,\dots,x_k) = \bigwedge_{i=2}^{k} x_i^{a_i}$ is an implicant of f. This product term p is zero for all input combinations $x_2 \dots x_k$ except for $p(x_2 = a_2,\dots,x_k = a_k) = 1$. Thus we only have to prove $p \leqslant f$ at point $(a_2,\dots,a_k)$. From the $\wedge$-combination,

$$f(1,a_2,\dots,a_k) = f(0,a_2,\dots,a_k) = 1$$

or

$$\begin{aligned} p(a_2,\dots,a_k) &\leqslant f(x_1,a_2,\dots,a_k) \\ &= x_1 f(1,a_2,\dots,a_k) \vee \bar{x}_1 f(0,a_2,\dots,a_k) \\ &= x_1 \vee \bar{x}_1 \\ &= 1, \end{aligned}$$

so it follows that $p \leqslant f$. That proves, by Definition 3.7, that p is an implicant of function f. For any decomposition in $1 \leqslant l \leqslant k$ inputs, using the Shannon Expansion 2^l times, it can be shown that p is an implicant of f. □

Corollary 3.1 (False points of the $\wedge$-combination). Assume a given decomposition $D_{i_1}D_{i_2}\dots D_{i_l}$ and a function g on $k-l$ inputs as the corresponding $\wedge$-combination. If $g(a_{l+1},\dots,a_k) = 0$, then the product term

$$p = x_{i_{l+1}}^{a_{l+1}} \dots x_{i_k}^{a_k}$$

is not an implicant of f.

Corollary 3.2 (Termination condition). If there is no $\wedge$-pattern in a given decomposition $D_{i_1}D_{i_2}\dots D_{i_l}$, then no product term of a subset of

$$\{x_{i_{l+1}}^{a_{l+1}}, x_{i_{l+2}}^{a_{l+2}},\dots,x_{i_k}^{a_k}\}$$

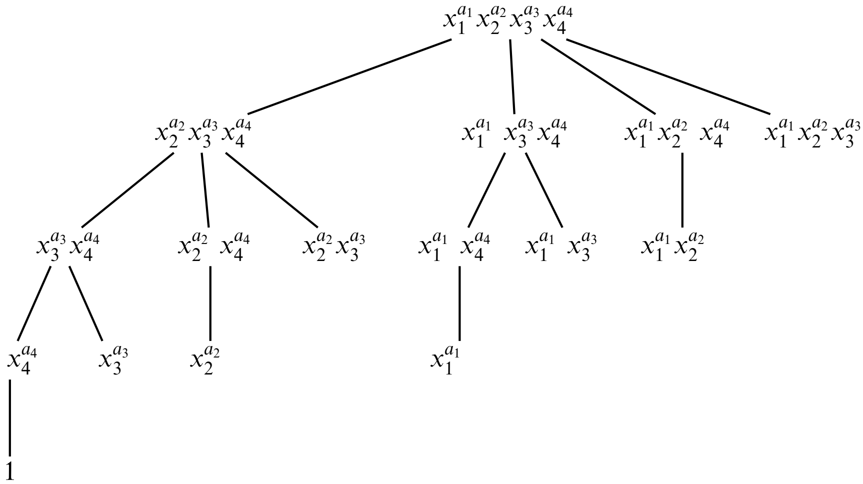
is an implicant of f.

Each decomposition $D_{i_1}D_{i_2}\dots D_{i_l} \in D$, $i_1 < \dots < i_l \in \{1\dots k\}$ corresponds to a class of implicants

$$x_{i_{l+1}}^{a_{l+1}} x_{i_{l+2}}^{a_{l+2}} \dots x_{i_k}^{a_k},$$

where $i_{l+1} < \ldots < i_k \in \{1 \ldots k\} \setminus \{i_1, \ldots, i_l\}$. It follows that the Decomposition Tree implies an Implicant Tree: the next example shows this for $k = 4$.

Example 3.2 (General Implicant Tree for $k = 4$).



The following checking condition is required for the prime implicant.

Lemma 3.2 (Prime implicant checking condition). If $\exists(a_1, \ldots, a_k) \in \{0,1\}^k$ such that $f(a_1, a_2, \ldots, a_k) = 1$ and the following k equations are fulfilled

$$\begin{aligned} f(\bar{a}_1, a_2, \ldots, a_k) &= 0 \\ f(a_1, \bar{a}_2, \ldots, a_k) &= 0 \\ &\cdots \\ f(a_1, a_2, \ldots, \bar{a}_k) &= 0\,, \end{aligned} \tag{6}$$

then a product term $p = x_1^{a_1} x_2^{a_2} \ldots x_k^{a_k}$ is a prime implicant of f.

3.2.1. *Prime implicants computation algorithm* Generally, at level l of the DT, one has to carry out the operator-combination of 2^l fragments of length 2^{k-l} for each node (*cf.* Example 2.2). By considering $\wedge$-combinations, we can reduce this to a comparison of two fragments of length 2^{k-l} by using the bit-string of the $\wedge$-combination of the parent node of the DT (level $l-1$, length 2^{k-l+1}): after the appropriate decomposition of this string, one obtains the two required bit-strings of length 2^{k-l} (*cf.* the example in Section 3.2.4). This leads to a significant reduction in the time complexity, which is based on the following lemma.

Lemma 3.3. Without loss of generality, given a Boolean function $f \in \{0,1\}^k$ and its $\wedge$-combination $g \in \{0,1\}^{k-1}$. If p is an implicant/prime implicant of function g, then p is also an implicant/prime implicant of f.

Proof. We have

$$p = f(0, x_2, ., x_k) \wedge f(1, x_2, ., x_k)$$

$$p \leqslant g \leqslant f$$

$$\Rightarrow p \leqslant f\,.$$

□

Lemma 3.3 is used in each node of the DT, so we only need the ∧-combination of 2 bit-strings in each node. The following short recursive algorithm computes the prime implicants of a given Boolean function f.

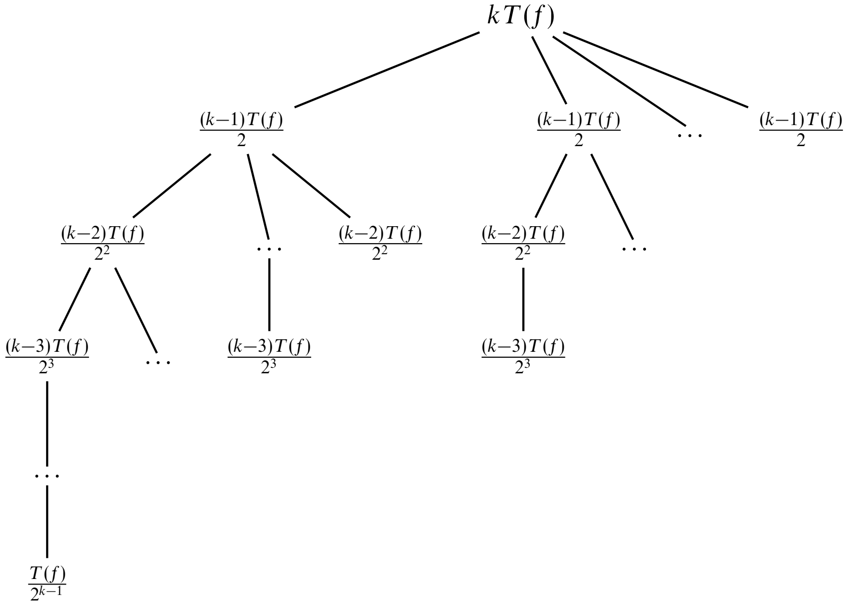
Algorithm A1. PRIME IMPLICANT COMPUTATION

```
  INPUT : BF $f(x_1,\ldots,x_k)$ as a bit-string $S$ of length $2^k$
  OUTPUT : the set of all $PI$s for $f$
{
  PROCEDURE GetPrimeImplicants($D_c$)
  {
    $ChildD_c := \emptyset$;
    for all $s \in D_c$ do
    {
      if $s \neq \{0...0\}$ then
      {
        $ChildD_c := ChildD_c \cup$ $\wedge$-combinations(Decompositions($s$));
        $PI := PI \cup$ TranslateToPI($s$, $ChildD_c$);
      }
    }
    $D_c := ChildD_c$;
    if $D_c \neq \emptyset$ then GetPrimeImplicants($D_c$);
  }

  $PI := \emptyset$;
  $D_c :=$ $\wedge$-combinations(Decompositions($S$));
  GetPrimeImplicants($D_c$);
  return $PI$;
}
```

The function *Decompositions(s)* builds and returns a set of all possible decompositions of a given function s according to the considered level of the DT. Set D_c contains all ∧-combinations. The function *TranslateToPI(s, ChildD$_c$)* proves each ∧-pattern of s (the true point of the ∧-combination) according to Lemma 3.1 with the help of $ChildD_c$. $ChildD_c$ contains all child ∧-combinations for the considered level. If the ∧-pattern corresponds to a prime implicant, it is translated into the corresponding logical expression (Lemma 3.1). Our algorithms are implemented in C++ and are available from the authors upon request.

3.2.2. *Time complexity* In this section we make a conservative estimate of the required run-time based on the number of true points $T(f) = \sum_{i=0}^{2^k-1} f_i$ of a given function f. Because of the $\wedge$-combination, the number of true points that have to be analysed in each node can be at most half of the number of true points of its parent node. For each of the true points (at most $T(f)/2^i$ for a node in level i) one has to do $k-i$ prime implicant checking operations (with the true points of the corresponding $\wedge$-combinations of the child level). The maximal number of operations per node can be illustrated as follows:



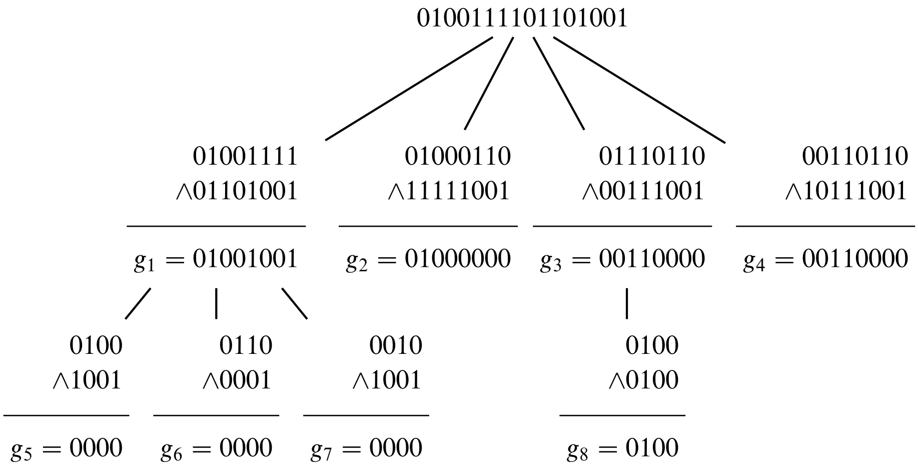
The time complexity is given by

$$\begin{aligned}
\sum_{i=0}^{k-1} (k-i)\frac{T(f)}{2^i}\binom{k}{k-i} &= T(f)\sum_{i=0}^{k-1}\binom{k}{i}\frac{k-i}{2^i} \\
&\leqslant T(f)\sum_{i=0}^{k-1}\binom{k}{i}\frac{k}{2^i} \\
&\leqslant T(f)k\sum_{i=0}^{\infty}\binom{k}{i}\left(\frac{1}{2}\right)^i \\
&= T(f)k(1+\frac{1}{2})^k \\
&\leqslant 2^k k(1+\frac{1}{2})^k \\
&= O(3^k k)\,.
\end{aligned}$$

The penultimate equality sign holds from Newton's binomial series. Therefore, the time complexity for the whole prime implicant algorithm is $O(3^k k)$. This is less than the time complexity needed by the the well-known algorithm of Quine and McCluskey for the truth table input, which has a runtime of $O(N^{\log 3} \log^2 N) = O(3^k k^2)$ $(N = 2^k)$ (Wegener 1989; 2000). Of course, heuristics, such as ESPRESSO (Rudell 1985) are faster, but our approach is the fastest worst case algorithm for prime implicant calculation that we are aware of.

3.2.3. *Space complexity* It is not necessary to store all nodes while traversing the DT. As our prime implicant algorithm only has to store two successive levels of the tree, the algorithm needs $\max_i\{2^{k-i}\binom{k}{i}\}$ bits for D_c, which can be improved to $O(2^{k \log 3}/k^{1/2})$: *cf.* Wegener (2000, page 102).

3.2.4. *Example*



Given the function 0100111101101001 $(k = 4)$, the DT with the corresponding $\wedge$-combinations is shown above. At the first level one obtains the $\wedge$-combination set $D_c = \{g_1, g_2, g_3, g_4\}$. During calculation of the first level we can mark all $\wedge$-patterns of function f that are covered by $\wedge$-patterns of the functions at level one. This results in just one $\wedge$-pattern, at position 10 of f, that is not covered by any pattern of level 1. Therefore, we obtain the prime implicant $x_1\bar{x}_2x_3\bar{x}_4$. The $\wedge$-pattern of g_1 contains all possible implicants of the type $x_2^{a_2}x_3^{a_3}x_4^{a_4}$. Since we have three $\wedge$-patterns in this function (positions 1, 4 and 7), we get the three different implicants $x_2x_3x_4$, $x_2\bar{x}_3\bar{x}_4$ and $\bar{x}_2\bar{x}_3x_4$. Combination g_2 leads to the implicant $\bar{x}_1\bar{x}_3x_4$. From g_3 we get the implicants $\bar{x}_1x_2\bar{x}_4$ and $\bar{x}_1x_2x_4$, and from g_4 we get $\bar{x}_1x_2\bar{x}_3$ and $\bar{x}_1x_2x_3$. In the next step we delete level 0 and calculate level 2 from the DT. By decomposing function g_1, we get the $\wedge$-combinations g_5, g_6 and g_7 (0-constant). We do not need to decompose function g_2, because it only contains one $\wedge$-pattern, which cannot lead to any new $\wedge$-pattern. The $\wedge$-combinations g_1 and g_2 produce no $\wedge$-pattern sets and thus do not give any additional implicants. The decomposition of function g_3, which represents implicants of class $x_1^{a_1}x_2^{a_2}$, contains one $\wedge$-pattern, which is the implicant $\bar{x}_1x_2$. Again we can mark all $\wedge$-patterns of the parent level that are covered by $\wedge$-patterns of the child level. It follows that all implicants found in g_1 and g_2 are prime implicants. The implicants of the functions g_3 and g_4 are fully covered by the implicant $\bar{x}_1x_2$ and are therefore not prime.

Summarising, we have that function f yields the prime implicants $x_1\bar{x}_2x_3\bar{x}_4$, $x_2x_3x_4$, $x_2\bar{x}_3\bar{x}_4$, $\bar{x}_2\bar{x}_3x_4$, $\bar{x}_1\bar{x}_3x_4$, $\bar{x}_1x_2$.

3.3. *ROBDD construction*

The DT can be used to construct the corresponding ROBDD of a given BF. At each decomposition node (starting at the highest level) one has to compare the corresponding decomposition fragments for equivalence. If an equivalence is found, the corresponding edge to the terminal node in the ROBDD is constructed, and the DT can be bound accordingly.

3.4. *Summary*

The following table summarises all the applications of the Decomposition Tree we have discussed.

Type of pattern	Detection by the DT
Implicant	$\wedge$-combination
Prime implicant	$\wedge$-combination (Algorithm A1)
Monotonic function	$\leqslant$-combinations
Linear function	$\oplus$-combinations of the D_i
Positive (negative) unate	$\leqslant$ ($\geqslant$)-combination of the D_i
Quadratic function	$\oplus$-combination of the D_i, D_iD_j
Symmetry	All decompositions of the first level are the same
Canalysing function	fragment $f_a^i \equiv b$
ROBDD	equivalence of decomposition fragments.

4. Conclusion

We have presented a new data structure called the Decomposition Tree, which provides a unified approach to analysing Boolean functions in a variety of ways. By decomposing bit-strings and combining them using different operators, one can: classify the function; construct the corresponding ROBDD; and efficiently compute its prime implicants. The required decompositions and operator-combinations can be done in a highly parallel manner as each node can be computed independently from all other nodes. The DT may also be used for logic minimisation: following a fast classification of the function, one can apply the appropriate minimisation algorithm. Because the DT represents the most general decomposition of a given Boolean function, we also expect a range of other possible applications. We have focused here on applications using the same logic operator for all bit-string combinations, but, in general, one could also apply different ones.

Acknowledgment

We would like to thank an anonymous referee for a number of valuable comments.

References

Andersen, H. R. (1997) An introduction to binary decision diagrams, Course Notes available at `http://www.itu.dk/~hra/notes-index.html`.

Benini, L. and de Micheli, G. (1997) A Survey of Boolean Matching Techniques for Library Binding. *ACM Transactions on Design Automation of Electronic Systems* **2** (9) 193–226.

Bollig, B. and Wegener, I. (1996) Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comp.* **45** 993–1002.

Boyar, J., Peralta, R. and Pochuev, D. (2000) On the muliplicative complexity of Boolean functions over the basis (∧,⊕,1). *Theoretical Computer Science* **235** 43–57.

Bryant, R. E. (1992) Symbolic Boolean manipulation with ordered binary decision diagrams, School of Computer Science, Carnegie Mellon, Pittsburgh.

Brayton, R., Hachtel, G., McMullen, C. and Sangiovanni-Vincentelli, A. (1984) *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer.

Brayton, R. (1992) Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys* **24** (3) 293–318.

Bshouty, N. and Tamon, C. (1996) On the Fourier spectrum of monotone functions. *J. ACM* **43** (4) 747–770.

Chung, K. S. and Liu, C. L. (1998) Local transformation techniques for multi-level logic circuits utilizing circuit symmetries for power reduction. *Proceedings of International Symposium on Low Power Electronics and Design* 215–220.

Clote, P. and Kranakis, E. (2002) *Boolean Functions and Computation Models*, Springer Verlag.

Coudert, O. (1994) Two-level logic minimization: an overview. *Integration* **17** (2) 97–140.

Crama, Y. and Hammer, P. L. (2006) Boolean Functions – Theory Algorithms and Applications. In preparation. Available at `http://www.rogp.hec.ulg.ac.be/Crama/`.

Dutuit, Y. and Rauzy, A. (1997) Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within Aralia. *Reliab. Eng. Syst. Safety* **58** 127–144.

Forbus, K. and de Kleer, J. (1992) *Building problem solvers*, MIT Press.

Hirose, S. and Ikeda, K. (1994) Nonlinearity criteria of Boolean functions. KUIS Technical Report KUIS-94-0002, Kyoto University, Japan.

Jeong, S. W., Kim, T. S. and Somenzi, F. (1993) An efficient method for optimal BDD ordering computation. In: *Proc. International Conference on VLSI and CAD*, Taejon, Korea.

Kauffman, S. (2000) *Investigation*, Oxford University Press.

De Kleer, J., Mackworth, A. and Reiter, R. (1993) Characterizing diagnoses and systems. *Artificial Intelligence* **59** 63–67.

Klüver, J. and Schmidt, J. (1999) Topology Metric and Dynamics of Social Systems. *Journal of Artificial Societies and Social Simulation* **2**.

Makino, K. and Ibaraki, T. (1997) The maximum latency and identification of positive Boolean functions. *SIAM J. Comput.* **26** (5) 1363–1383.

McCluskey, E. J. (1956) Minimization of Boolean functions. *Bell System Technical Journal* **35** 1417–1444.

McGeer, P. C., Sanghavi, J. V., Brayton, R. K. and Sangiovanni-Vincentelli, A. L. (1993) Espresso-Signature: A New Exact Minimizer for Logic Functions. *Design Automation Conference* 618–624.

Mohnke, J. and Malik, S. (1993) Permutation and phase independent Boolean comparison. *Integration* **16** 102–129.

Nikolajewa, S., Friedel, M. and Wilhelm, T. (2007) Boolean Networks with biologically relevant rules show ordered behavior. *BioSystems* **90** (1) 40–47.

Quine, J. O. W. (1952) The Problem of Simplifying Truth Functions. *American Math. Monthly* **59** 521–531.

Preneel, B., Govaerts, R. and Vandewalle, J. (1991) Cryptographic properties of quadratic Boolean functions. In : *Abstracts of the 1st International Conference on Finite Fields and Applications.*

Reiter, R. and De Kleer, J. (1987) Foundations of Assumption-based Truth Maintenance Systems: Preliminary Report. *AAAI-87*, Seattle, Washington 183–189.

Rudell, R. (1985) Espresso software program, Computer Science Dept., University of California, Berkeley.

Szallasi, Z. and Liang, S. (1998) Modeling the Normal and Neoplastic Cell Cycle With Realistic Boolean Genetic Networks: Their Application for Understanding Carcinogenesis and Assessing Therapeutic Strategies. In: *Pacific Symp. on Biocomputing* **3** 66–76.

Shmulevich, I., Lähdesmäki, H. and Egiazarian, K. (2004) Spectral Methods for Testing Membership in Certain Post Classes and the Class of Forcing Functions. *IEEE Signal Processing Letters* **11** (2) 289–292.

Strzemecki, T. (1992) Polynomial-time algorithms for generation of prime implicants. *Journal of Complexity* **8** 37–63.

Wang, Y., McCrosky, C. and Song, X. (2001) Single-faced Boolean Functions and their Minimization. *Computer Journal* **44** (4) 280–291.

Wegener, I. (2000) *Branching Programs and Binary Decision Diagrams – Theory and Application*, SIAM Monographs on Discrete Mathematics and Applications.

Wegener, I. (1987) *The Complexity of Boolean Functions*, Wiley.

Wegener, I. (1989) *Effiziente Algorithmen für grundlegende Funktionen*, B. G. Teubner.

Wendt, P., Coyle, E. and Gallagher, N. (1986) Stack Filters. *IEEE Trans. Acoustics Speech Signal Process* **34** (4) 898–911.