TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY
Julkaisu 842 • Publication 842

Stanislav Stanković

XML-Based Framework for Representation of Decision Diagrams

Tampere 2009

Tampereen teknillinen yliopisto. Julkaisu 842
Tampere University of Technology. Publication 842

Stanislav Stanković

XML-Based Framework for Representation of Decision Diagrams

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 3rd of November 2009, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2009

ISBN 978-952-15-2248-2 (printed)
ISBN 978-952-15-2258-1 (PDF)
ISSN 1459-2045

Preface

Graph-like structures are often used as a means of organizing data with complex internal structure. In recent decades, decision diagrams, a special class of directed acyclic graphs, have found numerous applications in a wide range of fields, from circuit design and testing to information theory and image processing. A variety of types of decision diagrams have proven to be an efficient method of representation of discrete functions, both in terms of memory size and processing time requirements. Over time a variety of decision diagram types have been introduced to represent different classes of discrete functions and to address different application needs.

At the moment, a plethora of software packages employing decision diagrams in some way is present either in industrial environment, or in academic circles. However, most of these software packages make use of proprietary formats for the internal representation and storage of decision diagrams. This makes data exchange between different software packages difficult. As far as we are aware, no attempt has been made to establish a standard format for the representation of such structures.

Our primary aim is to attempt to amend this situation by proposing a uniform framework for the representation of various classes of decision diagrams. Such a platform needs to satisfy several important criteria. Above all it needs to achieve a high degree of generality, i.e. it should to be able to describe as many as possible of the decision diagram classes introduced so far. It should also be easily extensible in order to accommodate possible new types of decision diagrams that may be introduced in the future. This abstract platform needs to be flexible enough to be transformed into various application specific formats. Finally, one cannot reasonably expect a wide adoption of a new standard which would require radical changes in the present software systems. Thus, this framework needs to be based on a technology that can be easily implemented on most operating systems, and programming environments.

In order to satisfy these criteria, we have chosen to build our framework using XML, an already established data description language created especially for the task of representing complex data structures. In our work we demonstrate that properties of XML make it well suited for the problem of representing decision diagrams. Furthermore, XML is in a wider sense a family of closely related languages dedicated to a particular data manipulation task.

In order to demonstrate the applicability of the proposed framework we exploit the features of XML to convert the abstract representation of decision diagrams into formats suitable for a range of distinct applications, such as hardware register transfer level (RTL) models in VHDL, EDIF based netlists, SVG vector images containing graphical representation of diagrams or branching programs in C++.

The author hopes that the discussion and examples presented in this thesis prove the validity of the chosen approach.

Acknowledgements

The material presented in this thesis represents the result of research conducted by the author between January 2005 and January 2009, at the Department of Signal processing of Tampere University of Technology (TUT), Finland.

This work was accomplished under the supervision of Professor Jaakko Astola. I wish to express my sincere gratitude for his scientific expertise as well as his support and guidance throughout this years. Without his help none of this work would have been possible.

I am also very grateful to Professor Jarmo Takala of Tampere University of Technology, the co-author of my publications for their contribution, advices and encouragement.

I wish to thank to Professor Michael Miller from University of Victoria and Professor Raimund Ubar of Tallinn Technical University, the reviewers of this thesis, for their valuable comments and recommendations.

I owe thanks to Professor Claudio Moraga for his valuable advices and support.

My thanks to Ms Pirkko Ruotsalainen, Ms Virve Larmila and Ms Ulla Siltaloppi for their assistance in practical arrangements.

I am tankful to my friends and colleagues from TUT, especially to Annamaria, Atanas, Vladislav, Michail, Kostadin, Aram, Robert, and may others for making working hours at university as pleasant as my free time. I also wish to thank my constant online companions Joannis, Valentin and Uroš for providing the much needed breaks from the daily routine.

My deepest gratitude and love goes to my family. Above all, to my father Radomir, for his love, support and help not only personal but professional, as well. Then, also, to my mother Milena from whom I acquired both my interest in computer science and first lessons in programming. Also, to my sister Smiljana.

Finally, my warmest thanks to my dear Jugoslava for her unconditional love, support and patience, and for giving me the happiest moments of my life.

Tampere, October 2009
Stanislav Stanković

Contents

List of Figures

List of Tables

List of Abbreviations

∗BMD	Multiplicative Binary Momentum Diagram
2D	Two Dimensional
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
BDT	Binary Decision Tree
BMD	Binary Momentum Diagram
CLB	Configurable Logic Block
CPU	Central Processing Unit
DD	Decision Diagram
DOM	Document Object Model
DSP	Digital Signal Processor
DT	Decision Tree
DTL	Decision Type List
EDIF	Electronic Design Interchange Format
ESOP	Exclusive-OR Sum-Of-Products
EVBDD	Edge Valued Binary Decision Diagram
FDD	Functional Decision Diagram
FEVBDD	Factorized Edge Valued Binary Decision Diagram
FPRM	Fixed Polarity Reed-Muller
FPGA	Field-Programmable Gate Array
GRM	Generalized Reed-Muller

HDD	Hybrid Decision Diagram
HTML	Hypertext Markup Language
KDD	Kronecker Decision Diagram
LUT	Look Up Table
MCNC	Microelectronics Center of North Carolina
MTBDD	Multi Terminal Binary Decision Diagram
nD	Negative Davio
OBDD	Ordered Binary Decision Diagram
pD	Positive Davio
PKDD	Pseudo Kronecker Decision Diagram
POS	Product of Sums
PPRM	Positive Polarity Reed-Muller
QDD	Quaternary Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
RTL	Register Transfer Level
S	Shannon Decomposition
SAX	Simple API for XML
SBDD	Shared Binary Decision Diagram
SGML	Standard Generalized Markup Language
SOP	Sum of Products
SVG	Scalable Vector Graphics
TDD	Ternary Decision Diagrams
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
VLSI	Very-large-scale Integration
WDD	Walsh Decision Diagram
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
ZBDD	Zero-Suppressed Binary Decision Diagram

1 Introduction

The basic idea that motivated the research presented in this thesis is to establish a standardized method for the representation of various classes of decision diagrams. Our main aim is to provide a means for efficient data exchange between various existing and potential, future software systems that deal with decision diagrams in some way.

Although decision diagrams have already been employed as a valuable tool in several important applications for a significant period of time, to our best knowledge no attempts at standardization have been made so far.

Such a platform has to satisfy several important criteria. Above all it needs to achieve generality, in the sense that it needs to be able to record all the significant structural features of as many distinct classes of decision diagrams already present in practice and the literature as possible. The platform needs to be easily extensible, in order to accommodate possible new types of decision diagrams that might be introduced in the future. In order to do so, the format specification must be focused on the common structural features shared by most classes of decision diagrams.

Furthermore, the proposed solution needs to be flexible enough to be applicable in various application scenarios. An abstract decision diagram representation should be easily convertible to particular application specific formats.

Finally, the whole solution needs to be easy to implement on various operating systems and programming environments. It would be unrealistic to expect a wide acceptance of a standard which would require significant changes in existing and established software packages.

In order to fulfill these requirements we have selected the XML data modeling language as a basis for our solution. The Extensible Markup Language

(XML) is a versatile data description language designed specially for the task of representing data with a complex and variable internal structure. The XML does not proscribe a strict data structure to which a certain class of documents needs to conform. Rather, it specifies a set of general rules that these documents need to adhere to. In a wider sense, XML is also a family of languages, XML proper and various derivatives, which offers a set of tools for efficient data manipulation, such as conversion of data between different formats. Recursive properties inherent in XML match the recursive properties exhibited by decision diagrams very well. XML parsers, software modules dedicated to processing XML documents, exist for all major software platforms as libraries for all common programming languages.

The work presented in this thesis is interdisciplinary. It is a software engineering task, related to the field of data structures, with applications in such areas as logic design and information theory and touching on topics such as group theory, graph theory and spectral techniques.

In order to make this thesis as simple to understand as possible, we decided to organize it in the following way.

The first part of this document is introductory in nature. We begin by examining the mathematical background of decision diagrams as a canonical representation of discrete functions. We start with the formal definition of discrete functions, discuss their classification and present several methods for their representation gradually introducing the more complex methods, culminating with decision diagrams. Next we turn our attention to spectral transforms which provide a uniform interpretation of various types of decision diagrams. Formal definitions of decision diagrams are given next. Further, decision diagrams are examined from the graph theory point of view, with special attention given to their topological properties, as these properties in large part determine the organization of the proposed XML based framework. The topological properties of decision diagrams, such as the size, i.e. number of nodes, maximal path length, etc., are of great importance to the applications of decision diagrams, especially the minimization of logic circuits.

In the second part of this thesis, we introduce the basic notions of the XML, its syntax and data description principles. Here we also present, the other members of the XML family which are of interest for our work. The XML Schema language is used to specify the file format of particular types of XML documents, and also to validate individual XML documents. Internal data addressing in XML based solutions is performed via the XPath language. Most importantly, a mechanism for the conversion of XML documents to other XML or non-XML based formats is provided by XSLT. We pay special attention to the XSLT language as it is employed by all of the application examples for the presented framework.

We focus next on the core of the proposed solution itself, the structure of XML documents for the representation of decision diagrams. We first discuss the data structures commonly employed for representation of acyclic directed graphs from the theoretical point of view. The details of XML Schema specification for decision diagrams are presented next. Since the primary task of the proposed framework is representation of various classes of decision diagrams, we give several examples of XML documents for a range of decision diagram types, starting with reduced ordered binary decision diagrams (ROBDDs) and functional decision diagrams (FDDs), through multi-output binary decision diagrams (MTBDDs) and shared binary decision diagrams (SBDDs) to multivalued ternary (TDDs) and quaternary (QDDs) decision diagrams.

In order to prove the validity of the proposed framework, we also present several examples of its potential application. However, before we are able to do so, we first need to examine the possible applications of decision diagrams in general, especially in the field of logic design. This discussion is presented in Chapter 4.

The majority of these examples deal with the application of decision diagrams in logic design, especially the implementation of switching functions using FPGA devices. The application of high-level programing languages in hardware design is known as High Level Synthesis. A quick overview of the related software solutions is given. We also examine the standard way in which switching functions expressed in decision diagram form are implemented in hardware. Therefore, a brief introduction to the concept of netlists is given in Section 5.2. Several high-level hardware description languages are currently in wide use. In our examples, we make use of two of them, VHDL as a general purpose hardware modeling language and EDIF for the specific task of netlist specification. An introduction to the basic concepts of both of them is given in Section 5.9 and Section 5.6 respectively. Their mutual similarities and differences are discussed in Section 5.5.

The first example we present deals with the implementation of switching functions using various families of FPGA devices. We have developed a set of two XSLT stylesheets capable of converting an XML document containing a ROBDD of a given function into a RTL hardware model expressed in VHDL syntax. In the next example, we compare the efficiency of MTBDD implementations of multi-output switching functions versus the more common SBDD implementation. In recent years, six input LUT based FPGA devices have been introduced as an optimal solution to the problem of granularity of logic blocks. QDDs have been proposed as a tool for logic design for such devices. We introduce an algorithm for generating a QDD function representation from its ROBDD. The XSLT implementation of this algorithm is presented in Section 5.13.

Information theory is one of the possible fields for application of decision diagrams. In the example, given in Chapter 6, we introduce a ROBDD based method for the calculation of the entropy estimate of a given binary string.

We further demonstrate the flexibility of the proposed framework in Chapter 7. A set of XSLT stylesheets is employed to automatically generate graphic representations of decision diagrams. The final product of this process is a vector image in SVG format.

Finally, in the last example we discuss the behavioral and structural aspects of system modeling with regards to decision diagrams in Chapter 8.

Concluding remarks are given in Chapter 9.

This thesis consists, in the greatest part, of the material originaly presented in the following publications by the author:

P 1 *S. Stanković, J. Astola, "XML framework for various types of decision diagrams for discrete functions",* IEICE Trans. Inf. and Syst.*, Vol. E90-D, No. 11, 2007, 1731-1740.*

P 2 *S. Stanković., J. Takala, J. Astola, "Method for Automatic Generation of RTL in VHDL Using Decision Diagrams", Proc. The 2006 Int. TICSP Workshop on Spectral Methods and Multirate Signal processing, SMMSP, Florence, Italy, 2006.*

P 3 *S. Stanković, J. Astola, "QDD Based Method of Automatic Circuit Design for Xilinx Virtex-5 FPGA Devices",* Journal of Multiple-Valued Logic and Soft Computing*, accepted for publicaiuton.*

P 4 *S. Stanković, J. Astola, "Calculating Entropy Estimate Using Binary Decision Diagrams", Proc. XI International Symposium on Problems of Redundancy in Information and Control Systems, 02 - 06 July, 2007, Saint Petersburg, Russia, 32-36.*

P 5 *S. Stanković, J. Astola, "XSLT Based Method for Automatic Generation of a Graphical Representation of a Decision Diagram Represented using XML", 7th International Workshop on Boolean Problems, Freiberg, Germany, 21-22 Sept. 2006.*

P 6 *S. Stanković, J. Astola, "Method for automatic generation of branching programs using decision diagrams", Proc. The 2007 Int. TICSP Workshop on Spectral Methods and Multirate Signal processing, SMMSP 2007, Moscow, Russia, September 3-4, 2007, paper cr1023.*

The general concept of XML-based framework for representation of Decision Diagrams was introduced in P 1. The applications of this framework in logic design have been discussed in P 2 and later in P 3. Other publications by the author discuss the application of the framework in other fields. Decision Diagram based method for entropy estimation was presented in P 4. An XML-based method for authomatic visualization of Decision Diagrams was proposed in P 5 and the method of authomatic generation of branching programs was presented in P 6.

The whole thesis and the publications cited above represent original work, of which the author was the main contributor. In particular, the XML-based methodology for representation of Decision Diagrams, the presented framework, conversion algorithms and related software implementations were proposed and developed by the author. Furthermore, all experimental results represent original work by the author.

2 Mathematical Background of Decision Diagrams

In this chapter we present the mathematical background of decision diagrams. These underlying concepts to a great extent determine the conditions which the XML-based framework for the representation of decision diagrams needs to satisfy. We begin our discussion by restating the definition of discrete functions. Special interest is devoted to switching functions, a special class of discrete functions which forms the foundation of digital circuit technology based on two stable state circuits. We continue by examining several forms of the representation of discrete functions, from the most simple tabular forms to graphics representation, ending with a formal definition of decision trees and decision diagrams. Further, we list the topological properties of decision diagrams that are of interest from an application point of view. We end this chapter with an overview of several important classes of decision diagrams, especially those which will be used in examples of applications of the proposed framework.

2.1 DISCRETE FUNCTIONS

In this section, we discuss some basic properties of discrete functions, and give an overview of certain classes of discrete functions that are of special interest in this work. We devote special attention to switching functions, since these functions are most important in practice. In the reminder of this thesis, we make use of switching functions to introduce many important concepts and give generalizations to other classes of discrete functions as appropriate.

Table 2.1 Classes of discrete functions.

Integer	$f : \times_n^{i=1}\{0, 1, ..., m_i - 1\} \rightarrow \{0, 1, ..., \tau_i - 1\}$
Multivalued	$f : \{0, 1, ..., \tau_i - 1\}^n \rightarrow \{0, 1, ..., \tau_i - 1\}$
Switching, Boolean	$f : B^n \rightarrow B$
Pseudo-logic	$f : \times_n^{i=1}\{0, 1, ..., m_i - 1\} \rightarrow \{0, 1\}$
Pseudo-Boolean	$f : \{0, 1\}^n \rightarrow R$, R - field of real numbers
Galois	$f : \{GF(p)\}^n \rightarrow \{GF(p)\}$

A general definition of discrete functions can be stated in the following form:

Definition 2.1 *Let A and B be sets. A relation is a subset r of $A \times B$. A relation f is a function if, and only if, for every $a \in A$ there exists exactly one $b \in B$ such that $(a, b) \in f$. A relation f is called an incompletely specified function if for each $a \in A$ there is at most one $b \in B$ such that $(a, b) \in f$, and there is at least one a for which there is no $(a, b) \in f$. The sets A and B are usually called the domain and the range of f, and written $f : A \rightarrow B$.*

Depending on the choice of domain and range there are different classes of discrete functions. We list the most important classes in Table 2.1.

The number of discrete functions is exponential in the cardinality of the domain. Consider discrete functions $f : X \rightarrow Y$, where each function is specified by a vector of its values with length $|X|$, where $|X|$ denotes the cardinality of the set X. Y^X is the set of all functions from X to Y. The total number of functions is $|Y|^{|X|}$.

The set of all subsets of a set A is called the power set of A and denoted by $\mathcal{P}(A)$. Often, the power set of A is denoted by 2^A due to the following considerations. The set of all functions from a set A to the set $2 = \{0, 1\}$ is 2^A. Each function in 2^A can be viewed as the characteristic function of a subset in A and can be identified with that subset. Therefore, there is a bijection between 2^A and $\mathcal{P}(A)$, and the power set of A can be denoted by 2^A.

From an application point of view, another very important concept is the concept of multi-output discrete functions.

Definition 2.2 *A multi-output discrete function is a function defined as:*

$$f : \times_{i=1}^n D_i \rightarrow \times_{i=1}^m R_i \tag{2.1}$$

where D_i, R_i are finite, non-empty sets.

The multi-output function f is clearly equivalent to a system $(f_1, ..., f_m)$ of single output functions $f_i : D_i \rightarrow R_i$, $i = 1, ..., n$.

Multi-output discrete functions can be used to describe the behavior of a system with multiple inputs and outputs. Especially, multi-output switching functions are used to describe the behavior of logic circuits.

2.1.1 Switching Functions

As shown in Table 2.1, switching functions represent a special class of discrete functions. We review some of their properties.

Definition 2.3 *A Boolean algebra* $\langle B; \vee, \wedge; ^{-}; 0, 1\rangle$ *is defined as a set* B*, two elements of* B*,* 0 *and* 1*, two binary operations* $\vee$*,* $\wedge$*, and a unary operation* $^{-}$*. These operations are known as logic OR, AND and NOT, respectively, and satisfy the following postulates:*

1. *Idempotencity* $x \vee x = x$, $x \wedge x = x$,
2. *Commutativity* $x \vee y = y \wedge x$, $x \vee y = y \wedge x$,
3. *Associativity* $x \vee (y \vee z) = (x \vee y) \vee z$, $x \wedge (y \wedge z) = (x \wedge y) \wedge z$,
4. *Absorption* $x \vee (x \wedge y) = x$, $x \wedge (x \vee y) = x$,
5. *Distributivity* $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$,
6. *Complement* $x \vee \bar{x} = 1$, $x \wedge \bar{x} = 0$,

for $x, y, z \in B$.

Boolean functions are defined in terms of expressions over a Boolean algebra.

In the case of a two-valued Boolean algebra $B = \{0, 1\}$ *and the operations can be expressed as:*

The logic OR operation

$\vee$	0	1
0	0	1
1	1	1

the logic AND operation

$\wedge$	0	1
0	0	0
1	0	1

and the unary logic NOT operation

$^{-}$	
0	1
1	0

.

Logic AND, OR and NOT operations are extended to B^n component wise.

Example 2.1 *For all* $x = (a_1, ..., a_n)$, $y = (b_1, ..., b_n)$, *where* $a_i, b_i \in B$, $i = 1, ..., n$, $x \vee y = (a_1 \vee b_1, ..., a_n \vee b_n)$.

Definition 2.4 *Let* $\langle B; \vee; \wedge; -; 0; 1\rangle$ *be a Boolean algebra. A variable that takes values in the set* B *is a Boolean variable. The expression that is obtained from the Boolean variables and constants by combining with the operations* $\vee$, $\wedge$, *and* $-$ *and parentheses is a Boolean expression. A Boolean mapping is a Boolean function if and only if every element in* B^n *has exactly one element in* B *associated with it. If a mapping* $f : B^n \rightarrow B$ *is represented by a Boolean expression, then* f *is a Boolean function. However, not all the mappings* $f : B^n \rightarrow B$ *are Boolean functions.*

Switching functions possess certain properties which make them suitable for the theoretical foundation of circuit design.

1. There exist exactly four distinct functions of a single variable x: $f_0(x) = 0$, $f_1(x) = 1$, $f_2(x) = x$ and $f_3(x) = \bar{x}$.
2. There are 16 two variable switching functions. They are used to describe the elementary logic gates (AND, OR, EXOR, etc.)
3. The number of n-variable switching functions with one output is equal to 2^{2^n}.
4. The number of m-output, n-variable switching functions is equal to 2^{m2^n}.
5. A logic circuit with n inputs and one output can be described by an n-variable switching function $f = (x_1, x_2, ..., x_n)$.
6. A logic circuit with n inputs and m outputs can be described in terms of m-tuple $f = (f_1, f_2, ..., f_m)$.

Switching functions, and discrete functions in general, can be represented in a variety of forms. The central subject of this work, decision diagrams, are one of the several forms of representation encountered in practice. In order to give a better view of the position of decision diagrams in relation to other forms of representation, we discus some of these forms in the following section. These forms will also be used for initial specification of functions in certain examples.

2.2 REPRESENTATION OF DISCRETE FUNCTIONS

Depending on the intended application, a variety of methods have been proposed for the representation of discrete functions. These methods can be divided into three large groups:

Table 2.2 An example of a binary switching function.

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

1. Decision tables and tabular forms in general,
2. Algebraic expressions,
3. Graphic methods, graphs, hypercubes, etc.

We present an overview of these methods with a special focus on switching functions.

2.2.1 Tabular Forms

Discrete functions can always be represented in tabular form, due to the fact that their domain is finite.

Example 2.2 *Table 2.2 presents a three-variable switching function f by listing function values for all possible inputs.*

The left side of the table represents the elements of the domain. The right part displays the corresponding function values.

In the case of binary functions this form of representation is known as the truth-table. The vector of function values is known as the truth-vector. The same concept is readily extended to multi-output functions.

Example 2.3 *Consider a three-variable two-output binary function $f = (f_1, f_2)$. We present the tabular representation of this function in Table 2.3.*

The tabular display is also convenient for other classes of discrete functions, as illustrated in Example 2.4 and Example 2.5 below.

Example 2.4 *Table 2.4 shows an example of a ternary three-valued discrete function in tabular form, with ternary variables on the left and the function values on the right.*

In the tabular form all the values of the variables are shown explicitly. Any interdependence of the function values is ignored. However, this manner of displaying causes a serious problem. The size of the table is exponential in

Table 2.3 An example of a binary multi-output switching function.

x_1	x_2	x_3	f_1	f_2
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	1

Table 2.4 An example of a ternary function.

x_1	x_2	f
0	0	0
0	1	1
0	2	0
1	0	2
1	1	2
1	2	1
2	0	2
2	1	2
2	2	1

the number of variables. Tabular representations quickly become unwieldy for even a relatively small number of discrete variables. Other methods of representation of discrete functions aim to achieve better economy by exploiting internal dependencies between function values.

2.2.2 Algebraic Expressions

An algebraic expression is a formal description of a discrete function in terms of algebraic relations between variables.

Definition 2.5 *If a discrete k-valued variable x_i takes its values in a set $Q_i = \{0, 1, \ldots, k-1\}$, then x_i^S, $S \subseteq Q_i$, is a literal of x_i, where*

$$x_i^S = \begin{cases} 1, & \text{if } x_i \in S, \\ 0, & \text{if } x_i \notin S. \end{cases}$$

When S contains only one element, $x_i^{\{j\}}$ is denoted by x_i^j.

Definition 2.6 *Let x_i take values in $Q_i, i = 1, ..., n$ and $x_{i_1}^{S_{i_1}}, ..., x_{i_j}^{S_{i_j}}$, $1 \leq i_1 \leq i_2 \leq ... \leq i_j \leq n$ be literals. The logic AND of literals $x_{i_1}^{S_{i_1}} \wedge ... \wedge x_{i_j}^{S_{i_j}} \left(= x_{i_1}^{S_{i_1}} ... x_{i_j}^{S_{i_j}}\right)$ is called a product term.*

A product term of the form $x_1^{S_1} \ldots x_n^{S_n}$ is called a minterm. We emphasize that all the variables appear in a minterm and there can be at most one literal for a variable.

Definition 2.7 *Let $\rho \subseteq 2^{Q_1} \times 2^{Q_2} \times \ldots \times 2^{Q_n} = \times_{i=1}^{n} 2^{Q_i}$. Then,*

$$\bigvee_{(S_1,\ldots,S_n)\in\rho} x_1^{S_1} \ldots x_n^{S_n}, \tag{2.2}$$

is a Sum-of-Products expression (SOP), where $\bigvee_{(S_1,S_2,\ldots,S_n)\in\rho}$ denotes the logic OR (inclusive OR) of product terms. If OR is replaced by the exclusive OR (EXOR) then the resulting expression

$$\bigoplus_{(S_1,S_2,\ldots,S_n)\in\rho} x_1^{S_1} x_2^{S_2} \cdots x_n^{S_n}$$

is an Exclusive-Sum-Of-Froduct expression (ESOP).

Example 2.5 *[141] Table 2.5 is an example of a multiple-valued input two-valued output function $f : Q_1 \times Q_2 \times Q_3 \to Q_1$, where $Q_1 = \{0,1\}$, $Q_2 = \{0,1,2\}$, and $Q_3 = \{0,1,2,3\}$. Thus, the variables $x_1 \in Q_1$, $x_2 \in Q_2$, $x_3 \in Q_3$ take two, three, and four values, respectively.*

The minterm expression for f is

$$\begin{aligned} f \;=\; & x_1^{\{0\}} x_2^{\{0\}} x_3^{\{0\}} \vee x_1^{\{0\}} x_2^{\{1\}} x_3^{\{1\}} \vee x_1^{\{0\}} x_2^{\{1\}} x_3^{\{3\}} \\ & \vee x_1^{\{0\}} x_2^{\{2\}} x_3^{\{2\}} \vee x_1^{\{1\}} x_2^{\{0\}} x_3^{\{0\}} \vee x_1^{\{1\}} x_2^{\{0\}} x_3^{\{3\}} \\ & \vee x_1^{\{1\}} x_2^{\{1\}} x_3^{\{1\}} \vee x_1^{\{1\}} x_2^{\{1\}} x_3^{\{3\}} \vee x_1^{\{1\}} x_2^{\{2\}} x_3^{\{2\}}. \end{aligned}$$

The SOP derived by extracting common terms in products, is

$$f = x_1^{\{0,1\}} x_2^{\{0\}} x_3^{\{0\}} \vee x_1^{\{0,1\}} x_2^{\{1\}} x_3^{\{1,3\}} \vee x_1^{\{0,1\}} x_2^{\{2\}} x_3^{\{2\}} \vee x_1^{\{1\}} x_2^{\{0,1\}} x_3^{\{3\}}.$$

After removing redundant literals, it is

$$f = x_2^{\{0\}} x_3^{\{0\}} \vee x_2^{\{1\}} x_3^{\{1,3\}} \vee x_2^{\{2\}} x_3^{\{2\}} \vee x_1^{\{1\}} x_2^{\{0,2\}} x_3^{\{3\}}.$$

The concept of SOP and ESOP expressions can be extended and generalized to derive other algebraic expressions. The product terms in these two expressions can be viewed as particular examples of basis function in the spaces of n-variable discrete functions with the domain and the range specified over some finite sets. It is assumed that the domain and the range are enriched with the necessary operations to express the algebraic structure of a finite group for the domain and a field for the range that can be a finite field or the field of complex numbers C.

Values for variables and function values can be interpreted as either logic (binary or multiple-valued) values or in general case as elements of the field

Table 2.5 The truth-table of the function f in Example 2.5.

$x_1x_2x_3$	f	$x_1x_2x_3$	f	$x_1x_2x_3$	f
000	1	020	0	110	0
001	0	021	0	111	1
002	0	022	1	112	0
003	0	023	0	113	1
010	0	100	1	120	0
011	1	101	0	121	0
012	0	102	0	122	1
013	1	103	1	123	0

assumed for the range of the functions considered. The minterms can be replaced with various other complete sets of linearly independent functions. Recall that here the term complete means that the number of basis functions is equal to the cardinality of the domain for the functions. Further, instead of OR and EXOR, other operations over the field assumed for the range can be used as the addition to define series-like expressions for discrete functions $f : D \rightarrow R$, $D = \times_{i=1}^{n} D_i$ as

$$f(x_1, x_2, \dots, x_n) = \sum_{i=1}^{|D|} c_i \phi_i(x_1, x_2, \dots, x_n), \tag{2.3}$$

where the coefficients c_i and the basis functions ϕ_i take values in the range R, and the addition is defined as the addition in the field imposed on R that is viewed as the support set of a field.

Expressions where the coefficients c_i and the basis functions ϕ_i take values in a finite field are called *bit-level expressions*, where bits are understood as either binary or multiple-valued. If c_i and $\phi_i(x_1, x_2, \dots, x_n)$ take values in the field of complex numbers C, the expressions are called *word-level expressions.* The expressions with integer-valued coefficients and basis functions are viewed as particular cases of expressions over C. These expression are most often used in dealing with switching (binary-valued) and multiple-valued logic functions due to simple encoding of n-tuples of logic values by the corresponding integers.

The properties of particular kinds of algebraic expressions are discussed in detail in what follows.

In particular, SOP expressions are defined by using minterms as basis functions ϕ_i, $i = 0, 1, \dots 2^n - 1$, where n is the number of variables. SOPs are widely used to describe behavior of switching functions and, therefore, will be discussed in more details in what follows. Another reason is that various other expressions can be derived from SOP expressions by converting minterms into different sets of basis functions.

We can compare the basic notion of the Sum-Of-Product expression to the tabular representation examined in the previous section. The assignment of variable values is, in this setting, expressed in terms of minterms multiplied by the corresponding function values. Minterms that are multiplied by 0 have no effect on the function output and are discarded. The remaining minterms, those that are multiplied by 1, make up the final function representation.

Example 2.6 *The binary function presented in tabular form in Example 2.2 can be represented by the following SOP expression:* $f = x_2x_3 \vee \bar{x}_2\bar{x}_3 \vee x_1x_2\bar{x}_3$.

Sum-of-Products can be seen as series-like expressions, where minterms play the role of basis functions, and again, the corresponding function values the role of coefficients.

In matrix notation minterms can be generated through the Kronecker product of the basic matrices $X(1) = [\bar{x}_i, x_i]$, for $i = 1, ..., n$. This Kronecker product produces an identity matrix. Therefore, the set of minterms (columns of the identity matrix) is know as the trivial basis.

Definition 2.8 *The Shannon (S) Expression [149] of a switching function* f *with respect to the variable* x_i *is:*

$$f = \bar{x}_i f_0 \vee x_i f_1 \tag{2.4}$$

where

$f_0 = f(x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_n)$ *and* $f_1 = f(x_1, ..., x_{i-1}, 1, x_{i+1}, ..., x_n)$

are the co-factors of f *with respect to* x_i.

Remark 2.1 *Since product terms in 2.4 has no comon minters,* $\vee$ *can be repalced by* $\oplus$. *In this case the Shannon expression has the form:*

$$f = \bar{x}_i f_0 \oplus x_i f_1. \tag{2.5}$$

In matrix notation, the Shannon expansion can be expressed in terms of basic matrices $\mathbf{X}(1)$ and $\mathbf{B}(1)$,

$$f = \begin{bmatrix} \bar{x}_i & x_i \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} \tag{2.6}$$

$$f = \mathbf{X}(1)\mathbf{B}(1)\mathbf{F}, \tag{2.7}$$

where $\mathbf{X}(1) = \begin{bmatrix} \bar{x}_i & x_i \end{bmatrix}$, $\mathbf{B}(1) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and, $\mathbf{F}(1) = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}$.

The Shannon expansion rule can be applied recursively to a switching function $f(x_1, ..., x_n)$, with respect to the variable x_i, $i = 1, ..., n$, which leads to the complete SOP for the considered function.

Example 2.7 *Recursive application of the Shannon expansion to a two-variable switching function* $f(x_1, x_2)$*:*

$$\begin{aligned} f(x_1,x_2) &= \bar{x}_1 f(0,x_2) \oplus x_1 f(1,x_2) \\ &= \bar{x}_1(\bar{x}_2 f(0,0) \oplus x_2 f(0,1)) \\ &\quad \oplus x_1(\bar{x}_2 f(1,0) \oplus x_2 f(1,1)) \\ &= \bar{x}_1\bar{x}_2 f(0,0) \oplus \bar{x}_1 x_2 f(0,1) \\ &\quad \oplus x_1\bar{x}_2 f(1,0) \oplus x_1 x_2 f(1,1) \end{aligned}$$

In the general case for an n-variable switching function, this can be expressed in the following form:

$$f = \mathbf{X}(n)\mathbf{B}(n)\mathbf{F} = \left(\bigotimes_{i=1}^{n} \mathbf{X}(1)\right)\left(\bigotimes_{i=1}^{n} \mathbf{B}(1)\right)\mathbf{F} \tag{2.8}$$

which is the matrix notation for the complete Sum-of-Products.

In the complete disjunctive form there are no common terms, thus is it possible to formally replace EXOR by OR in the Shannon expansion rule and it will remain true. This is, however, not the case for the reduced Sum-of-Products derived from the complete Sum-of-Products by exploiting properties of the Boolean algebra.

The binary Shannon expansion can be generalized to the p-valued case. The following example illustrates the generalization of the Shannon decomposition for four-valued discrete functions.

Example 2.8 *The Shannon expansion of an arbitrary r-variable four-valued function $f(x_1, x_2, ..., x_r)$, with respect to x_1 is: $f(x_1, x_2, ..., x_r) = x_1^0 f(0, x_2, ..., x_r) \vee x_1^1 f(1, x_2, ..., x_r) \vee x_1^2 f(2, x_2, ...x_r) \vee x_1^3 f(3, x_2, ..., x_r)$, or in short form $f = \vee_{(a_1,a_2,...,a_n)} f(a_1, a_2, ..., a_n) x_1^{a_1} x_2^{a_2} ... x_n^{a_n}$, where where x_i^k are literals of four-valued variables as defined in Definition 2.5.*

A different choice of matrix $\mathbf{X}(1)$ and the corresponding $\mathbf{B}(1)$ in (2.8) produces a different expansion. We present some examples of such expansions.

Remark 2.2 *Recalling one of the basic properties of Boolean logic, $\bar{x}_i = 1 \oplus x_i$, the Shannon expansion can be rewritten as:*

$$\begin{aligned} f &= \bar{x}_i f_0 \oplus x_i f_1 = (1 \oplus x_i) f_0 \oplus x_i f_1 \\ &= 1 \cdot f_0 \oplus x_i f_0 \oplus x_i f_1 = 1 \cdot f_0 \oplus x_i (f_0 \oplus f_1) \end{aligned} \tag{2.9}$$

This expansion is called the positive Davio (pD) expansion. Every switching function f can be written, using this expansion, in the following form:

$$f = c_0 \oplus c_1 x_i \tag{2.10}$$

where $c_0 = f_0$ and $c_1 = f_0 \oplus f_1$.

Remark 2.3 *The expansion that is obtained by iterative application of the positive Davio expansion is called the Positive Polarity Reed-Muller expansion (PPRM).*

Example 2.9 *The positive Polarity Reed-Muller expansion of a two variable switching function $f(x_1, x_2)$:*

$$\begin{aligned}
f &= 1 \cdot f(0, x_2) \oplus x_1(f(0, x_2) \oplus f(1, x_2)) \\
&= 1 \cdot (1 \cdot f(0,0) \oplus x_2(f(0,0) \oplus f(0,1))) \\
&\quad \oplus x_1(1 \cdot f(0,0) \oplus x_2(f(0,0) \oplus f(0,1)) \\
&\quad \oplus 1 \cdot f(1,0) \oplus x_2(f(1,0) \oplus f(1,1))) \\
&= 1 \cdot f(0,0) \oplus x_2(f(0,0) \oplus f(0,1)) \oplus x_1 \cdot (f(0,0) \oplus f(1,0)) \\
&\quad \oplus x_2(f(0,0) \oplus f(0,1) \oplus f(1,0) \oplus f(1,1)) \\
&= c_0 \oplus c_1 x_2 \oplus c_2 x_1 \oplus c_3 x_1 x_2
\end{aligned}$$

where,

$$\begin{aligned}
c_0 &= f_0 \\
c_1 &= f(0,0) \oplus f(0,1) \\
c_2 &= f(1,0) \oplus f(1,1) \\
c_3 &= f(0,0) \oplus f(0,1) \oplus f(1,0) \oplus f(1,1)
\end{aligned}$$

These are the Reed-Muller coefficients.

The matrix notation of the Reed-Muller expansion has the following form:

$$f = \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}, \tag{2.11}$$

$$f = \mathbf{X}_{rm}(1)\mathbf{R}(1)\mathbf{F}. \tag{2.12}$$

The extension to functions of an arbitrary number of variables is done by the Kronecker product

$$f = \mathbf{X}(n)\mathbf{B}(n)\mathbf{F} = \left(\bigotimes_{i=1}^{n} \mathbf{X}_{rm(1)} \right) \left(\bigotimes_{i=1}^{n} \mathbf{R}(1) \right) \mathbf{F}. \tag{2.13}$$

Remark 2.4 *Similarly to the case of the positive Davio expansion, the negative Davio (nD) expansion is derived from the Shannon expansion by the relation $x_i = 1 \oplus \bar{x}_i$.*

In matrix notation:

$$f = \begin{bmatrix} 1 & \bar{x}_i \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}. \tag{2.14}$$

The positive and negative Davio rules can be combined in an expansion of a single switching function.

Remark 2.5 *Fixed Polarity Reed-Muller (FPRM) expansions are a generalization of the Positive Polarity Reed-Muller (PPRM) expansions, in which we can choose between either the positive or negative Davio expansion for each variable x_i.*

A Reed-Muller expansion is usually specified by a binary polarity vector $H = (h_0, ..., h_n)$, where $h_i \in 0, 1$ specifies the polarity for variable x_i. By convention $h_i = 0$ implies a negative literal $\bar{x}_i$, and $h_i = 1$ a positive literal x_i. Expansions for different polarities differ in the number of non-zero coefficients.

Example 2.10 *Consider a binary polarity vector $H = (1, 0, 1)$. A Fixed Polarity Reed-Muller matrix defined by this vector has the form:*

$$\begin{aligned} R(1) \otimes \bar{R}(1) \otimes R(1) &= \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}. \end{aligned}$$

Remark 2.6 *A Fixed Polarity Reed-Muller expansion with the minimal number of non-zero coefficients is known as a minimal expansion.*

For an n-variable function there are 2^n possible choices of polarities for variables and, therefore, 2^n different Fixed-polarity Reed-Muller expressions. Finding the minimal expansion is an NP-complete problem. There can be more than one minimal Fixed Polarity Reed-Muller expansion for a given function, [9].

We can further extend the concept of a Fixed Polarity Reed-Muller expansion by including the possibility to also choose the Shannon expansion rule for an individual variable x_i. Since we now have three possible choices, Shannon, positive and negative Davio, the total number of expansions for an n-valued function is 3^n.

Coefficients in these expansions can be calculated using the Kronecker transform matrix defined as:

$$\mathbf{K}(n) = \bigotimes_{i=1}^{n} \mathbf{K}_i(1) \tag{2.15}$$

where $\mathbf{K}_i(1)$ can be any of the matrices for S, pD, and nD expansion rules.

The choice of the decomposition rule for the variables of a function is specified using a Decision Type List (DTL).

In a Fixed Polarity Reed-Muller expression, the polarity of a variable is fixed throughout each individual minterm. Each variable can appear as either a positive or negative literal, but never both in a single expression.

Remark 2.7 *A Generalized Reed-Muller expression (GRM) is a Reed-Muller-like expression where the polarity of each individual variable can be different in different product terms.*

Example 2.11 *Consider a three-variable switching function $f(x_1, x_2, x_3)$. The generalized Reed-Muller expression for this function will have the following form*

$$f = c_0 \oplus c_1\widetilde{x}_1 \oplus c_2\widetilde{x}_2 \oplus c_3\widetilde{x}_1\widetilde{x}_2 \oplus c_4\widetilde{x}_3 \oplus c_5\widetilde{x}_1\widetilde{x}_3 \oplus c_6\widetilde{x}_3\widetilde{x}_3 \oplus c_7\widetilde{x}_1\widetilde{x}_2\widetilde{x}_3,$$

where $c_i \in \{0, 1\}$ and $\widetilde{x}_i \in \{x_i, \bar{x}_i\}$.

There are $2^{n2^{n-1}}$ possible generalized Reed-Muller expressions for an n variable function, since each input can be 0, 1, $\bar{x}_i$ or x_i.

Generalized Reed-Muller expressions retain one important constraint. No two product terms may have an identical set of variables. These product terms are called primary products. However, this constraint is not present in EXOR Sum-of-Products expressions (ESOPs), an even more general class of algebraic expressions.

Definition 2.9 *An EXOR Sum-of-Products expression is a sum of arbitrary product terms of the form:*

$$f = \bigoplus_{I} \widetilde{x}_1\widetilde{x}_2...\widetilde{x}_n, \tag{2.16}$$

where I is a set of all the possible products, and $\widetilde{x}_i \in \{1, x_i, \bar{x}_i\}$.

Example 2.12 *For a two-variable switching function f there exists total of 9 EXOR Sum-of-Products expressions:*

$$\bar{x}_1\bar{x}_2, \quad \bar{x}_1x_2, \quad \bar{x}_1 \cdot 1, \quad x_1\bar{x}_2, x_1 \cdot 1, \quad 1 \cdot \bar{x}_2, \quad 1 \cdot x_2, 1. \tag{2.17}$$

We are not limited strictly to Boolean operations and Boolean values when dealing with discrete function expansions. For instance, the so-called arithmetic expressions are defined as expressions where the coefficients and the

Table 2.6 Boolean and Arithmetic operations.

Boolean	Arithmetic
$x_1 \wedge x_2$	$x_1 x_2$
$x_1 \vee x_2$	$x_1 + x_2 - x_1 x_2$
$x_1 \oplus x_2$	$x_1 + x_2 - 2x_1 x_2$

basis functions take values over the field of rational numbers. These expressions are effectively used in the representation of multi-output functions, since it permits their representations by a single expression with integer-valued coefficients polynomial, while bit-level expressions require a separate expression for each output. Further, arithmetic expressions are useful in compact representation of arithmetic circuits as adders, multipliers, and various control circuits.

Remark 2.8 *Arithmetic expansions can be derived if operations in $GF(2)$ are replaced with operations in the field of rational numbers Q. Table 2.6 gives a list of Boolean and the corresponding arithmetic operations.*

The values 0 and 1 are treated as rational numbers, $0, 1 \in Q$, where Q is the field of rational numbers. These expansions belong to the class of word-level expansions.

For an n-variable function f the arithmetic expansion is defined as:

$$f = \left(\bigotimes_{i=1}^{n} \mathbf{X}_a(1) \right) \left(\bigotimes_{i=1}^{n} \mathbf{A}^{-1}(1) \right) \mathbf{F} \tag{2.18}$$

where, $A^{-1}(1) = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$.

Walsh expansions are another example of word-level expressions.

Walsh expansions for discrete functions are defined in terms of discrete version of Walsh functions, which were introduced in 1923 by J. L. Walsh, [175].

Definition 2.10 *Walsh functions of order n, denoted as $wal(w, x)$, $x, k \in 0, 1, ..., 2^n$, are defined as columns of the Walsh matrix:*

$$\boldsymbol{W}(n) = \bigotimes_{i=1}^{n} \boldsymbol{W}(1) \tag{2.19}$$

where $W(1) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

The expansion of a function f using the Walsh matrix can be expressed in the following form:

$$f = \frac{1}{2}\left[\begin{array}{cc} 1 & 1-2\bar{x}_i \end{array}\right]\left[\begin{array}{cc} 1 & 1 \\ 1 & -1 \end{array}\right]\left[\begin{array}{c} f_0 \\ f_1 \end{array}\right]. \tag{2.20}$$

Increasing the freedom of choice of the form of product terms increases the expressive power of the arithmetic expressions. The number of possible expressions for each function grows, increasing the probability of finding an optimal expression. However, this is also reflected in the computational complexity of the problem. A heuristic approach is necessary as exact algorithms for finding all possible expressions of a given function of a certain class quickly become inefficient for even a relatively small number of variables.

2.2.3 Spectral Techniques

The notion of arithmetic expressions is closely associated with the concept of spectral techniques. The idea behind spectral techniques is to isomorphically transform a problem from one space into another space where some of the properties of the problem would be exposed better, thus making a problem easier to solve. The same idea can be directly applied to discrete functions. Discrete functions transformed into some other space might, hopefully, be easier to handle. Namely, this new space could provide us with the opportunities for a more compact representation of functions.

The Fourier transform first introduced by J. B. Fourier in the 18th century [65] represents the foundation of spectral techniques. A Discrete Fourier Transform represents a special case of the Fourier transform.

In general terms, we can define discrete spectral transforms in the following manner.

Definition 2.11 *Let $P(G)$ be the vector space of functions $f : G \to P$, where g is a finite group of order $g = |G|$ and P is a field. Let $\Phi = \{\phi_i\}, i = 0, ..., g-1$ be a set of g linearly independent functions in $P(G)$. Each function $f \in P(G)$ can be expressed as a linear combination of the functions ϕ_i.*

$$f = \sum_{i \in G} s_i \phi_i(x), \quad s_i \in P. \tag{2.21}$$

If G is the direct product of n groups G_j, it can represent the domain for an n-variable function $f(x_1, ..., x_n)$, where $x_j \in G_j$. In the case of switching functions $G_j = C_2$, where C_2 is the cyclic group of order 2, i.e., $C_2 = (\{0,1\}, \oplus)$, and $\oplus$ is the addition modulo 2, logic XOR.

We can write the expression (2.21) in matrix form

$$\mathbf{F} = \mathbf{R}\mathbf{S}_f, \tag{2.22}$$

where $\mathbf{F}$ is a function f in vector form, $\mathbf{R}$ is the matrix representation of a set of linearly independent functions, and $\mathbf{S}_f$ is the vector of the corresponding spectral coefficients.

From (2.22) it follows that

$$\mathbf{S}_f = \mathbf{R}^{-1}\mathbf{F}. \tag{2.23}$$

If $G = \times_{i=1}^{n} G_i$ the matrix $\mathbf{R}$ can be represented via the corresponding matrices R_i on G_i. In this case we write

$$\mathbf{R(n)} = \bigotimes_{i=1}^{n} \mathbf{R}_i(1). \tag{2.24}$$

We can observe the similarity between (2.21) and (2.3) from the previous section, and also (2.22) and the matrix notation of the Shannon expansion (2.6). Indeed every arithmetic expression presented in the previous section corresponds to one spectral transform.

Therefore, we can define the Reed-Muller transform in terms of

$$\mathbf{R}(n) = \bigotimes_{i=1}^{n} \mathbf{R}_i(1), \quad \mathbf{R}_i(1) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{2.25}$$

Since $\mathbf{R}(n)$ is a self inverse matrix, $\mathbf{R}(n)^{-1} = \mathbf{R}(n)$ over $GF(2)$, the Reed-Muller spectral coefficients are calculated in the following way

$$\mathbf{S}_f = \mathbf{R}(n)\mathbf{F}. \tag{2.26}$$

This transform is known as the Positive Polarity Reed-Muller transform, since spectral coefficients are the coefficients in the PPRM expression. By choosing different $\mathbf{R}(1)$ matrices in the Kronecker product, we can create the Fixed Polarity Reed-Muller transform, which is equivalent to the Fixed Polarity Reed Muller expressions.

In the same way as the case of arithmetic expressions, we can make a transition from $GF(2)$ to C, in effect treating the 0 and 1 as integer values and replacing binary operations with arithmetic ones.

Therefore,

$$\mathbf{A} = \bigotimes_{i=1}^{n} \mathbf{A}_i(1), \quad \mathbf{A}_i(1) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{2.27}$$

However, over C the matrix

$$\mathbf{A}(1) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \tag{2.28}$$

is not self inverse.

The Arithmetic transform is, therefore, defined in terms of

$$\mathbf{S}_a = \mathbf{A}^{-1}\mathbf{F} \tag{2.29}$$

where,

$$\mathbf{A}^{-1} = \bigotimes_{i=1}^{n} \mathbf{A}_i^{-1}(1), \mathbf{A}_i^{-1}(1) = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}. \tag{2.30}$$

This represents the most well-known form of the arithmetic transform. Formally, other spectral transforms can be constructed in the same manner, by choosing a different linearly independent basis in C. The arithmetic transform is frequently used to represent stochastic and probabilistic properties of discrete systems [2], [98], [155].

Discrete Walsh functions form a basis in $C(G)$. Using this basis we can define the Walsh transform. Its matrix representation is

$$\mathbf{W} = \bigotimes_{i=1}^{n} \mathbf{W}_i(1), \mathbf{W}_i(1) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{2.31}$$

The Walsh transform is equivalent to the Fourier transforms on the dyadic groups C_2^n [63]. It expresses the properties corresponding to the properties of the classical Fourier transform on R. Therefore, it is often called the Walsh-Fourier transform.

We have presented only a brief overview of a narrow selection of spectral techniques topics. These techniques represent a large area of research with numerous applications in various fields from signal processing, through system theory to logic design. For further information on spectral techniques please refer, for example, to [90], [95], [155], [164].

2.2.4 Decision Diagrams

Discrete functions can be represented using various types of graphical methods, see for instance [141]. In this section we focus primarily on decision diagrams and their general properties.

Decision diagrams as a means of representing discrete functions attracted considerable attention after the publication of the paper [17] by Bryant.

In order to better illustrate the concept of decision diagrams, we need to first examine some related concepts.

Definition 2.12 *A graph G is a pair (V, E) where V is a set of nodes and E is a set of two-element subsets of V, so called edges.*

Definition 2.13 *A graph $G(V, E)$ is* directed *if, and only if, each element $e \in E$ is an ordered pair $e = (a, b)$ such that $a \in V$ and $b \in V$.*

Remark 2.9 *Let E be a set of edges of a directed graph $G(V, E)$. The first element of an pair $e = (a, b)$, $e \in E$ is called the* parent node *and element b is known as the* child *or the* descendant node.

Definition 2.14 *Let $G(V, E)$ be a graph. Nodes $x, y \in V$ are* adjacent *if and only if $(x, y) \in E$, otherwise they are* non-adjacent. *A graph $H(V_1, E_1)$ is a* subgraph *of G, if and only if $V_1 \subset V$ and $E_1 \subset E$, where the sign $\subset$ indicates a proper subset. A* walk *in a graph is a sequence*

$$(v_0, e_0, v_1, e_2, v_2, ..., e_n, v_n)$$

where $e_i = \{v_{i-1}, v_i\}$, for $i = 1, ..., n$. The length of the walk is the number of edges n in the sequence.

Remark 2.10 *A* path *is a walk in the graph whose nodes are distinct, except possibly for the first and the last. A* circuit *is a path where the first and last node are identical.*

Remark 2.11 *A* tree *is a graph that contains no cycles. If there is a special element $v_0 \in V$ called the* root, *then the tree is a* rooted tree.

We have introduced the concept of the decomposition of a function $f(x_1, ..., x_n)$ using different expansion rules in the previous section. As shown in Example 2.8, the recursive application of a decomposition rule to each variable x_i results in a series-like expression. In the case of the Shannon decomposition rule for binary functions can be displayed using a tree-like graphic form as seen in Figure 2.1.

This graphic representation is known as the Decision Tree (DT).

In this example, the Shannon decomposition is used and such decision tree is often referred as the Shannon decision tree.

A generalization of the concept of decision trees can be obtained by using other decomposition rules, such as those considered in Subsection 2.2.2, the examples of which are the positive Davio and the negative Davio rules. Examples of these generalizations will be considered in Section 2.4.

A formal description of decision trees for the representation of discrete functions can be given by viewing trees as a particular case of directed acyclic graphs.

Definition 2.15 *Let $f(x_1, ..., x_n)$ be an n-variable discrete function, where each $x_i = 0, ..., m_i$, $i = 1, ..., n$ is an m_i-valued variable, $m_i \in N$.*

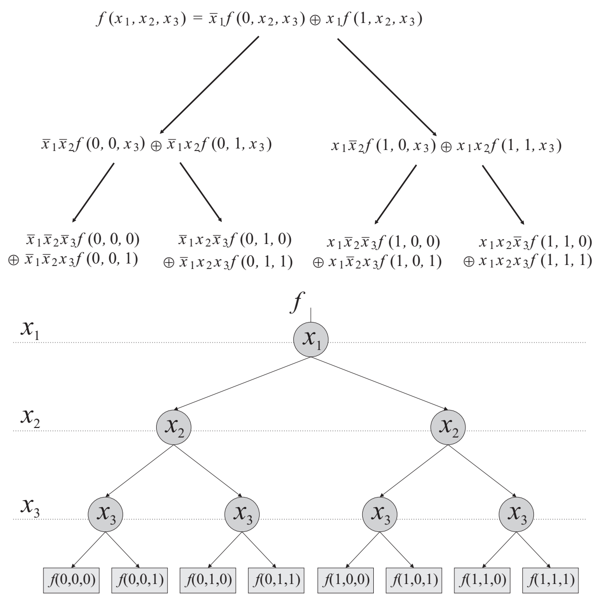

$f(x_1,x_2,x_3) = \bar{x}_1 f(0,x_2,x_3) \oplus x_1 f(1,x_2,x_3)$
$\bar{x}_1\bar{x}_2 f(0,0,x_3) \oplus \bar{x}_1 x_2 f(0,1,x_3)$
$x_1\bar{x}_2 f(1,0,x_3) \oplus x_1 x_2 f(1,1,x_3)$
$\bar{x}_1\bar{x}_2\bar{x}_3 f(0,0,0) \oplus \bar{x}_1\bar{x}_2 x_3 f(0,0,1)$
$\bar{x}_1 x_2\bar{x}_3 f(0,1,0) \oplus \bar{x}_1 x_2 x_3 f(0,1,1)$
$x_1\bar{x}_2\bar{x}_3 f(1,0,0) \oplus x_1\bar{x}_2 x_3 f(1,0,1)$
$x_1 x_2\bar{x}_3 f(1,1,0) \oplus x_1 x_2 x_3 f(1,1,1)$
f
x_1
x_2
x_3
f(0,0,0)
f(0,0,1)
f(0,1,0)
f(0,1,1)
f(1,0,0)
f(1,0,1)
f(1,1,0)
f(1,1,1)

Fig. 2.1 A tree-like representation of the Shannon expansion for $n = 3$.

A decision tree is a tree-like graphic representation of the function $f(x_1, ..., x_n)$, obtained by the recursive application of one or more expansion rules from a specified set of expansion rules, for each variable x_i, $i = 1, ..., n$.

A decision tree consists of a set of non-terminal nodes, V, a set of terminal nodes T and a set of connecting edges E.

Each non-terminal node in V represents a decomposition of the function $f(x_1, ..., x_n)$, for a variable x_i, $i = 0, ..., n$. In the case of m-valued discrete variables, a non-terminal node associated with the decomposition of the function for x_i has exactly m outgoing edges pointing to sub-graphs representing co-factors of the function for particular values of the variable x_i.

Terminal nodes in T represent elements of the range of the function taken by the function $f(x_1, ..., x_n)$ for particular values of discrete variables.

Remark 2.12 *The non-terminal node v_0 corresponds to the first step in the recursive decomposition of the function $f(x_1, ..., x_n)$. This element is the root of the decision tree. The root node corresponds to the first variable and represents the function f.*

Definition 2.16 *Let $f(x_1, ..., x_n)$ be an n-variable discrete function, and $D(V, T, E)$ a decision tree associated with it. A path in D is a sequence of elements $p_k = (u_0, e_1, u_1, e_2, ..., u_{i-1}, e_i, u_i, ..., u_{n-1}, e_n, u_n)$. Such that $u_0 = v_0$, $u_i \in V$, $e_i = \{u_{i-1}, u_i\} \in E$ and $u_n \in T$.*

Each path in a decision tree starts at the root node of the tree $u_0 = v_0$, and ends in one of the terminal nodes ($u_n \in T$). A path in a decision tree assumes a particular assignment of values to the variables associated with the nodes traversed by the path.

The choice of expansion rule determines the labels associated with non-terminal nodes and edges.

Different types of decision trees correspond to different classes of discrete functions and expansion rules. Each node of a decision tree has exactly one incoming edge. The number of outgoing edges is determined by the decomposition rule associated with the node.

The function is determined from its decision tree by the application of the rules inverse to the expansion rules used to generate the tree. The appropriate inverse rules are applied first to the values in the terminal nodes, and then, following the structure of the tree ,to cofactors in non-terminal nodes.

In Figure 2.2, we can observe that several paths, starting from the root of the diagram, may lead to identical sub-trees. By removing these redundancies we obtain a simpler representation of the original function. If no further reduction is possible, this new, much more compact representation, is known as the decision diagram. The exact set of rules by which the reduction of a decision tree is performed depends on the class of the decision tree, as will be discussed in Section 2.4, where different types of decision diagrams are described.

In order to discuss reduction of decision trees into decision diagrams, the following definition is needed.

Definition 2.17 *Let $G(V_G, E_G)$, and $H(V_H, E_H)$ be two directed graphs consisting of sets of nodes V_G and V_H respectively and sets of edges E_G and E_H respectively. G and H are isomorphic if, and only if, there exists a bijection $q : V_G \to V_H$ such that $(a, b) \in E_G$ if, and only if, $(q(a), q(b)) \in E_H$.*

As in the case of decision trees, the simplest and most widely used decision diagrams are derived by the reduction of binary decision trees, i.e., trees defined in terms of the Shannon decomposition rule. These decision diagrams can formally be defined as follows, [9].

Definition 2.18 *An (ordered) binary decision diagram is a rooted directed acyclic graph with a set of non-terminal nodes V and a set of terminal nodes T. A non-terminal node has a label $index(v) \in \{1, 2, \ldots, n\}$ and two children $low(v)$ and $high(v) \in V$. A terminal node has a label $value(v) \in \{0, 1\}$. Further, for any non-terminal node v, if $low(v)$ ($high(v)$) is a non-terminal, then $index(low(v)) > index(v)$, ($index(high(v)) > index(v)$).*

The function f represented by the decision diagram is defined as follows in a recursive manner.

Let $(x_1, ..., x_n) \in \{0, 1\}^n$ and $v \in V$. The value of $f(x_1, ..., x_n)$ at v, $f_v(x_1, ..., x_n)$ is

1. $f_v(x_1, ..., x_n) = value(v)$ if v is a terminal node, *2.* $f_v(x_1, ..., x_n) = \overline{x}_{index(v)} f_{low(v)}(x_1, ..., x_n) + x_{index(v)} f_{high(v)}(x_1, ..., x_n)$ *and* $f(x_1, ..., x_n) = f_{root}(x_1, ..., x_n)$.

Due to the recursion 2, these diagrams can be derived by the recursive application of the binary Shannon decomposition rule with respect to all the variables in a given function f.

Example 2.13 *Figure 2.2 shows the binary decision tree and the binary decision diagram for a three-variable switching function* $f(x_1, x_2, x_3) = x_2x_3 \oplus \bar{x}_2\bar{x}_3 \oplus x_1x_2\bar{x}_3$. *As stated in 2.6, in the case of the Shannon decomposition, the coefficients correspond to the function values. Therefore, in the binary case, edges will have two possible labels, 0 and 1, or alternatively the literals* $\bar{x}_i$ *and* x_i *respectively. Values of terminal nodes are function values and can be either* 0 *or* 1.

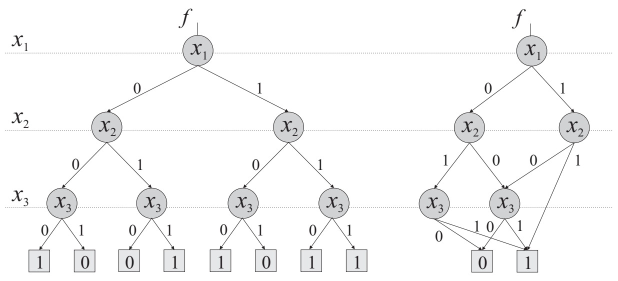


Fig. 2.2 The Binary Decision Tree and the Binary Decision Diagram for the function $f(x_1, x_2, x_3) = x_2x_3 \oplus \bar{x}_2\bar{x}_3 \oplus x_1x_2\bar{x}_3$.

In a decision tree, the edges are all of length 1 and, therefore, all the paths have the same length that is equal to the number of levels in the decision tree, equivalently, the number of variables in the represented function. Since a decision diagram is obtained from a decision tree by deleting nodes where no decision is made, and by sharing isomorphic subtrees, in a decision diagram there can be edges connecting nodes at level i with nodes at level $i + k$. The length of such edges is obviously k. In this case, the length of a path from the root node to a terminal node is equal to the sum of the lengths of all the edges the path consists of.

For a representation of functions with multiple-valued variables and also functions taking values in different fields such as finite fields of certain orders or the complex field, more general types of decision diagrams has been defined, see for instance [9]. A possible generalization of the definition of binary decision diagrams can be given by referring to the domain of the function to be represented as a finite group and the range as a field.

Consider functions on a finite group G which is a direct product of n finite groups $G_1, \ldots, G_n$, where the order of G_i, $|G_i| = g_i$, $i = 1, \ldots, n$, into a field P. Because

$$G = G_1 \times G_2 \times \cdots \times G_n,$$

the function f on G can be viewed as a function of n variables $f(x) = f(x_1, \ldots x_n)$, where $x_i \in G_i$. For each $i = 1, \ldots, n$, define (δ function) $\delta : G_i \to P$ by

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0, \text{ the zero of } G_i, \\ 0 & \text{otherwise.} \end{cases}$$

To simplify the notation, we denote each function just by δ and the domain specifies which function is in question. We can define the more general concept of a decision diagram in an analogous way.

Definition 2.19 *Let G, P, f, and δ be as above. An ordered decision diagram over G is a rooted directed graph with two types of nodes. A non-terminal node has a label $index(v)$, and $g_{index(v)}$ children, $child(j, v)$, $j = 1, \ldots, g_{index(v)}$. Each edge $(v, child(j, v))$ has a distinct label $element(j, v) \in G_{index(v)}$. A terminal node has a label $value(v) \in P$. Further, for any non-terminal node v, if $child(i, v)$ is non-terminal, then $index(child(i, v)) > index(v)$.*

Again, the function f is represented by the decision diagram in a recursive manner as follows.

Let $(x_1, \ldots, x_n) \in G$ and $v \in V$. The value of f at v, $f_v(x_1, \ldots, x_n)$ is

1. If v is a terminal node, $f_v(x_1, \ldots, x_n) = value(v)$
2. If v is a non-terminal node,

$$f_v(x_1, \ldots, x_n) = \sum_{j=1}^{g_{index(v)}} \delta(x_{index(v)} - element(j, v)) f_{child(j,v)}(x_1, \ldots, x_n),$$

and $f(x_1, \ldots, x_n) = f_{root}(x_1, \ldots, x_n)$.

The recursion 2. is an exact generalization of the binary case, but because the variables take values in the domain groups G_i, the concept (label) $element(j, v)$ provides the correspondence between the elements of g_i and outgoing edges of a node having the index i. This correspondence allows us to consider such diagrams as being derived by the recursive application of the generalized Shannon decomposition rule, an example of which is discussed in Example 2.8.

Notice that in a decision diagram over a group $G = G_1 \times G_2 \times \cdots \times G_n$, the nodes on the level i (root being the level 1) correspond to the factor G_i in the sense that the elements of g_i correspond to the outgoing edges (decisions) of the nodes on the level i.

In a decision tree the decomposition rule applied to variables in f need not necessarily be the same for all the variables. Moreover, there are decision trees where a different decomposition rule may be selected for each node in the decision tree. Examples of such trees will be discussed in Section 2.4. Such decomposition rules are usually selected that, for the given function, the number of isomorphic subtrees is the largest, leading after reduction to the most compact decision diagram.

Different types of decision trees correspond to different classes of discrete functions and expansion rules [9], examples of which will be discussed in Section 2.4.

Similarly as for algebraic expressions for discrete functions, depending on the decomposition rules and the range of functions to be represented, bit-level and word-level decision diagrams can be distinguished.

In bit-level decision diagrams, terminal nodes represent logic values 0 and 1 or elements of finite fields. In word-level decision diagrams, values of terminal nodes can be integers, real numbers, complex numbers or vectors and matrices with these numbers as entries.

Remark 2.13 *If the order of discrete variables is identical in each path from the root node to a constant node, this diagram is an ordered decision diagram, otherwise, it is a Free Decision Diagram [9].*

An ordered decision diagram is a canonic representation of a discrete function, [17]. Notice that Free decision diagrams also provide canonic representations under certain conditions. A canonic representation is a unique representation of the function, thus, once the order of variables is specified the decision diagram becomes a unique representation of a given function. Function values can be read by following each path from the root of the diagram, observing the operations determined by the decomposition rule as specified by traversed node and edge labels.

2.3 TOPOLOGICAL PROPERTIES OF DECISION DIAGRAM

As for any directed acyclic graph, a decision diagram is distinguished by its topology. Certain properties of individual diagrams are of great importance at the application level, especially for the task of hardware implementation of switching functions. We list the most important of these properties here.

Definition 2.20 *Let $D(V, T, E)$ be a decision diagram, where V is a set of non-terminal nodes, T is a set of terminal nodes and E is aset of edges. The size of the decision diagram is the number of elements in V.*

Of special interest, at the implementation level, is the number of non-terminal nodes of the diagram, since it is equivalent to the number of functional hardware units. Edges in a decision diagram represent interconnections in the corresponding network and terminal nodes are inputs in the networks derived from decision trees.

Definition 2.21 *The number of paths in the decision diagram is the total number of different sequences p_k.*

Definition 2.22 *The path length $L(p_k)$ is the number of non-terminal nodes $v_i \in V$ traversed on an individual path p_k.*

Definition 2.23 *The maximal path length is the length of the longest path in the diagram, $max(L(p_k))$.*

Definition 2.24 *The average path length is calculated by averaging over different paths in the diagram*

$$avg = \sum_k \frac{L(p_k)}{k}.$$

On the hardware level, the lengths of paths in the diagram are related to the propagation delay of the circuit. The reduction of the path length can be of special importance when dealing with certain hardware platforms. This is also an important feature for the application of decision diagrams in the verification of logic circuits. A further discussion on this and related topics will be presented in Section 5.13.

Definition 2.25 *The set of non-terminal nodes associated with the variable x_i in an ordered decision diagram constitute the i-th level of the diagram.*

Definition 2.26 *In ordered decision diagrams the width of the diagram is the largest number of nodes at an individual level of the diagram.*

Example 2.14 *The decision diagram in Fig. 2.2 has the following:*

Size	*5*
Width	*2*
Max. path length	*3*
Avg. path length	*2.5*

2.4 CLASSIFICATION OF DECISION DIAGRAMS

Decision diagrams can be classified according to the underlying function and the decomposition rule. In this section we define several classes of decision diagrams, which will be addressed in the remainder of this thesis.

The most common class of decision diagrams is the class of binary decision diagrams associated with Shannon decomposition of switching functions.

Remark 2.14 *A Ordered Binary Decision Diagram (OBDD) of a switching function $f(x_1, ..., x_n)$, $i = 1, ..., n$, is a decision diagram with a set of non-terminal nodes V, the set of terminal nodes $T = \{0, 1\}$, and a set of edges E,* *corresponding to the recursive application of Shannon decomposition to each binary variable x_i.*

Let $a_1, ...a_{i-1} \in \{0,1\}$ and $v \in V$ corrspond to the function $f_v(a_1, ..., a_{i-1}, x_i, x_{i+1} ..., x_n)$ of the variables $x_i, ..., x_n$. Then the descendants of v corrspond to the function $f_v(a_1, ..., a_{i-1}, x_i, ..., x_n) = \bar{x}_i f_0(a_1, ..., a_{i-1}, 0, x_{i+1} ..., x_n) \vee x_i f_1(a_1, ..., a_{i-1}, 1, x_{i+1}, ..., x_n)$.

A reduced ordered binary decision diagram is obtained by applaying the following two reduction rules to a binary decision tree:

1. If both outgoing edges of a given node n point to the same descendant node, the node n is removed. This rule is derived from the basic property of Boolean algebra, $\bar{x} \oplus x = 1$.

2. If two sub-graphs represent the same function, delete one, and reconnect its incoming edges to the remaining sub-graph. This rule can be formalized as $x \vee x = x$.

We show an illustration of these two rules in Fig. 2.3.

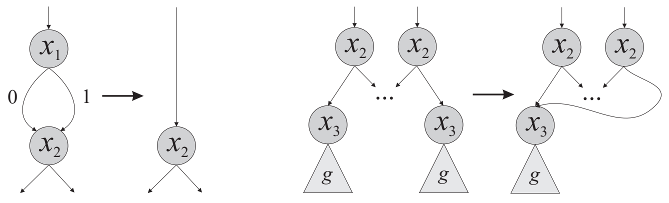


Fig. 2.3 Binary decision diagram reduction rules.

Definition 2.27 *An OBDD is a Reduced Ordered Binary Decision Diagram (ROBDD) if it satisfies the following two conditions:*

1. *It does not contain a non-terminal node v with descendants v_1 and v_2, $v, v_1, v_2 \in V$ such that $v_1 = v_2$.*

2. *It does not contain two non-terminal nodes $v_1 \neq v_2$, $v_1, v_2 \in V$, such that the sub-graphs rooted at v_1 and v_2 are isomorphic.*

Each assignment of values $\in \{0,1\}$ to all variables of $f(x_1, ..., x_n)$ corresponds to a unique from root to a terminal node in ROBDD. The value of the terminal node is the value of f with this assignment. A key factor in usefulness of ROBDDs is the fact that while each assignment corresponds to a unique path, a single path may correspond to a large number of assignments.

Example 2.15 *Consider $f(x_1, x_2, x_3) = x_2x_3 \oplus \bar{x}_2\bar{x}_3 \oplus x_1x_2\bar{x}_3$ a three variable switching function. Fig. 2.4 represents its ROBDD. The path $p = \{(0,1), (1,5), (5,3)\}$ corresponds to the function value $f(000) = 0$.*

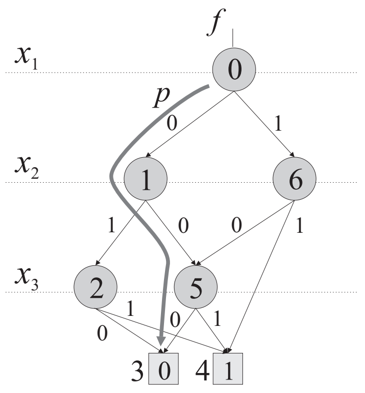


Fig. 2.4 The Binary Decision Diagram of the function $f(x_1, x_2, x_3) = x_2x_3 \oplus \bar{x}_2\bar{x}_3 \oplus x_1x_2\bar{x}_3$.

In a similar manner we can define decision diagrams for multi-valued discrete functions.

Quaternary Decision Diagrams are a graphic representation of a generalized Shannon decomposition of four-valued discrete functions, as demonstrated in Example 2.8.

Remark 2.15 *An Ordered Quaternary Decision Diagram, (OQDD), of a function* $f : \{0,1,2,3\}^n \rightarrow T$ *arises in the same way from* 4*-valued Shannon decomposition* $f_v(x_1, ..., x_i, ..., x_n) = x_i^0 f_0(x_1, ..., x_i = 0, ..., x_n) \vee x_i^1 f_1(x_1, ..., x_i = 1, ..., x_n) \vee x_i^2 f_2(x_1, ..., x_i = 2, ..., x_n) \vee x_i^3 f_3(x_1, ..., x_i = 3, ..., x_n)$.

Example 2.16 *Fig. 2.5 shows a Quaternary Decision Diagram for the four-valued, three-variable function* $f(x_1, x_2, x_3) = x_1^0x_3^1 \vee 2x_1^0x_3^2 \vee 3x_1^0x_3^2 \vee 3x_1^0x_3^3 \vee x_1^1 \vee x_1^2x_2^1 \vee 2x_1^2x_2^{2,3}x_3^3 \vee 3x_1^2x_2^{2,3}x_3^3 \vee x_1^3x_2^0 \vee x_1^3x_2^1x_3^1 \vee 2x_1^3x_2^1x_3^2 \vee 3x_1^3x_2^1x_3^3 \vee 2x_1^3x_2^2 \vee 3x_1^3x_2^3$*, in Figure 2.5.*

The concept of decision diagrams is readily extended to multi-output functions.

Shared Decision Diagrams are graphic representations of multi-output discrete functions, where each sub-function is represented with a decision diagram with a separate root node. These diagrams are usually partially overlapping. Identical sub-diagrams of individual diagrams are represented only once.

Definition 2.28 *Let* $f(f_1, ..., f_m)$ *be an* m*-output discrete function, and let* $D_1(V_1, T_1, E_1), ..., D_m(V_m, T_m, E_m)$ *be reduced ordered decision diagrams associated with individual outputs* $f_1, ..., f_m$*. A Shared Decision Diagram*

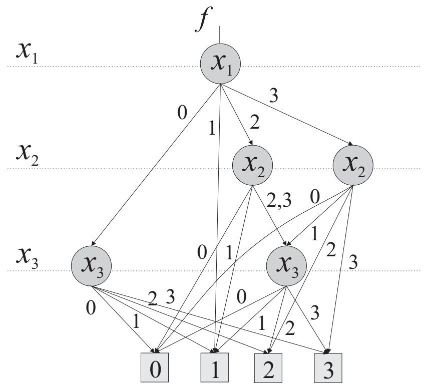


Fig. 2.5 A Quaternary Decision Diagram for a three-variable discrete function.

(SDD), $D(V,T,E)$ is defined as $D = D_1 \cup ... \cup D_m$, where $V = V_1 \cup ... \cup V_m$, $T = T_1 \cup ... \cup T_m$, and $E = E_1 \cup ... \cup E_m$, and there are no two non-terminal nodes $v_1 \neq v_2$, $v_1, v_2 \in V$, such that sub-graphs rooted at v_1 and v_2 are isomorphic.

Example 2.17 *Figure 2.6 shows a Shared binary decision diagram for a two-output three-variable function $f(f_1, f_2)$.*

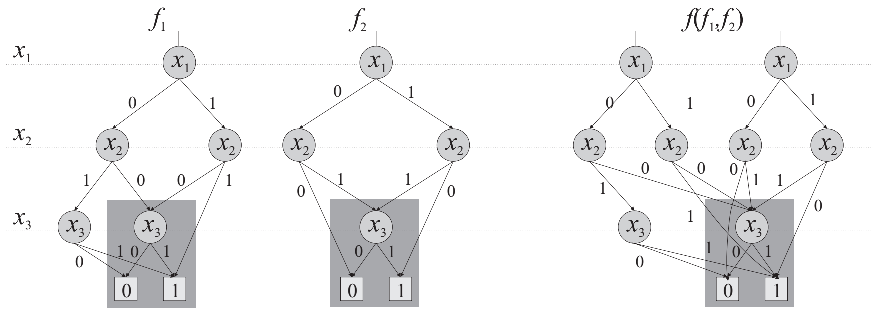


Fig. 2.6 A Shared Decision Diagram for a multi-output three-variable function $f(f_1, f_2)$.

Shared binary decision diagrams were introduced in [121] and their properties were further studied in [46], [119].

A decision diagram with a set of terminal nodes T with more then two elements is called Multi-Terminal Binary Decision Diagram.These diagrams can be used to represent multi-output switching functions, [30], [118].

Example 2.18 *Consider a multi-output binary function $f = (f_0, f_1, f_3)$ defined by the following table,*

x_1	x_2	x_3	f_0	f_1	f_3	f_z
0	0	0	0	0	0	0
0	0	1	0	1	0	2
0	1	0	1	1	1	7
0	1	1	0	0	1	1
1	0	0	1	0	1	5
1	0	1	0	1	0	2
1	1	0	1	1	1	7
1	1	1	0	0	1	1

We can represent this multi-output function in integer form, by summing each individual output f_i, multiplied by 2^i. Thus, we obtain $f_z = \sum_i 2^i f_i$.

Thus, multi-terminal biary decision diagrams give a natural way of representing multi-output switching function. Each node corresponds to Shannon decompsition $f = \bar{x}_i f_0 + x_i f_1$, where variable x_i is interpreted to take integer values 0 and 1, and f integer values $\{0, 1, ..., 2^n - 1\}$. Note that the Boolean $\vee$ changes to integer addition.

Example 2.19 *We present the Multi-terminal binary decision diagram for the function from Example 2.18 in Figure 2.7.*

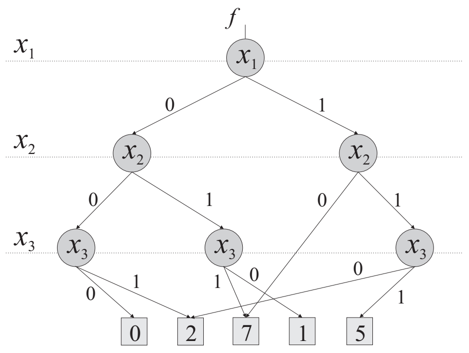


Fig. 2.7 A Multi-terminal Binary Decision Diagram.

More information about the representation of multi-output switching functions using multi-terminal binary decision diagrams can be found in [30].

The main objective of decision diagram minimization is to produce the most compact representation of a given function in terms of one or more of the topological features mentioned in the previous section. Usually it is the size of the diagram, the number of paths, or the maximal path length. We have defined decision diagrams so far with respect to the Shannon expansion rule.

However, the choice of different expansion rules may lead to more compact function representation.

Functional Decision Diagrams are a graphic representation of the Positive-Polarity Reed-Muller expression defined by (2.10).

They are, thus, a representation of the recursive application of pD-expansion to a switching function f. The values of the terminal nodes give the Reed-Muller coefficients for f. The edges of Functional decision diagrams are labeled with 1 and x_i, since 1 and x_i are assigned to f_0 and $f_0 \oplus f_1$ to which the outgoing edges point.

Remark 2.16 *A Functional Decision Diagram, (FDD), of a switching function $f(x_1, ..., x_n)$ is obtained by recursive application of the positive Davio decomposition to the variables $x_1, ..., x_n$. Thus, a non-terminal node v associated with the variable x_i, $x_1, ..., x_{i-1}$ fixed, corresponds to the function tion $f_v(x_1, ..., x_i, ..., x_n) = f_0(x_1, ..., 0, ..., x_n) \oplus x_i(f_0(x_1, ..., 0, ..., x_n) \oplus f_1(x_1, ..., 1, ..., x_n))$.*

Because the Reed-Muller transform is selfinverse we obtain the values of its Reed-Muller spectrum by traversing all the paths in the FDD of a given function f. The values of the truth vector of f are derived by traversing all the paths in the FDD and applying positive Davio rule at each node.

Example 2.20 *Figure 2.8 is the functional decision diagram for the Positive Polarity Reed-Muller expression of the function of Example 2.2. The Reed-Muller coefficients of this function are specified by the following vector $\mathbf{c} = [11100011]^T$ determined by (2.25).*

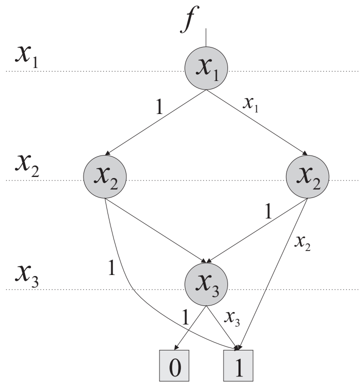


Fig. 2.8 A Functional Decision Diagram for a Positive Polarity Reed-Muller expansion of the three-variable switching function in Example 2.20.

The reduction rules used in the construction of the diagram in Fig. 2.20 are identical to the BDD reduction rules. However, due to the specific properties of pD expansion, some additional reduction rules have been defined in [145] for FDTs. A positive Davio node can be eliminated if one of its outgoing edges points to the zero-terminal node. This rule is derived from relations $x \cdot 0 = 0$ and $\bar{x} \cdot 0 = 0$. This reduction rule is known as the Zero-suppression rule and the decision diagrams obtained using this rule are known as the Zero-suppressed decision diagrams (ZBDDs), [120].

The main idea behind the Functional decision diagrams, that a suitable expansion rule can reduce the complexity of the function representation, is further extended in Kronecker Decision Diagrams. In the case of the Kronecker decision diagrams, any of the three expansion rules, S, pD, nD, can be chosen at each step of the recursion, where S denotes the Shannon expansion and pD and nD, the positive and the negative Davio expansions respectively.

Thus, in each level, non-terminal nodes correspond to either Shannon, positive or negative Davio expansion rule. The outgoing edges of nodes at each level are labeled accordingly, with $\bar{x}_i$, x_i for Shannon, 1, x_i for positive Davio, and 1 and $\bar{x}_i$ for negative Davio expansion. The type of decomposition associated with each particular level is specified in the Decision Type List.

Definition 2.29 *A Decision Type List, (DTL) for $f(x_i, ...x_n)$, $i = 1, ..., n$, is a set of labels $D = \{d_i, ..., d_n\}$, where $d_i = \{S, pD, nD\}$, where S indicates the Shannon, pD the positive and nD the negative Davio expansion.*

Example 2.21 *Fig. 2.9 is a Kronecker Decision Diagram for the function in Example 2.2. The decomposition of the function is defined by the decision type list $\{pD, S, nD\}$. The coefficients are specified by the following vector $\mathbf{c} = [01110001]^T$. This vector is also formed by the multiplication of the matrix $\boldsymbol{K}(3)$ with the truth of vector of f, where*

$$\boldsymbol{K}(3) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

since the inverse over $GF(2)$ of the matrix $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ corresponding to the negative Davio expansion is $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$.

Note that the order in which the expansion rules occur is identical along each path of the diagram. Pseudo Kronecker Decision Diagrams add one more degree of freedom by eliminating this constraint.

Pseudo Kronecker Decision Diagrams (PKDDs) are the class of decision diagrams where a different expansion rule from the set S, pD, nD can be freely chosen for each individual node. The outgoing edges of each node are labeled according to the selected decomposition rule.

Example 2.22 *A pseudo Kronecker binary decision diagram for the binary function in Example 2.2 is shown in Figure 2.10.*

A detailed classification of decision diagrams can be found in [9] and [182].

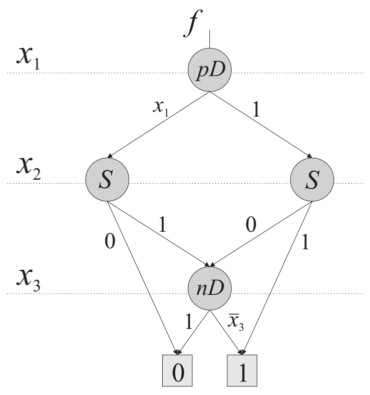


Fig. 2.9 A Kronecker Decision Diagram for a three-variable switching function, derived using the BDD reduction rules.

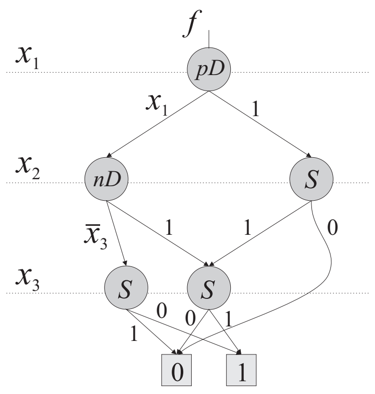


Fig. 2.10 A Pseudo Kronecker Decision Diagram for a three-variable switching function, derived using the BDD reduction rules.

The system proposed in this work is capable of representing all types of decision diagrams introduced in this chapter. We present the appropriate examples in the following chapter. Since the proposed framework is focused on the most general properties of a decision diagram, a directed acyclic graph structure, the system can be extended to other, possibly new, classes of decision diagrams.

3 XML Framework for Decision Diagrams

Many software packages use decision diagrams in some form. A large number of programs has been developed to enable different kinds of experiments with various decision diagrams, as reported in numerous publications on this subject, see, for example, [47], [119], [154] and references therein. The CUDD library is used in many applications as a standard in this area [150], [151]. An extension of these methods to the programming of decision diagrams for multiple-valued functions was done in [118]. Some other packages, for example, PUMA [47] and BEMITA [119] support work with particular classes of decision diagrams.

However, most of these software solutions use a proprietary, application specific format for storing decision diagrams. In general, they offer only limited possibilities of data exchange with other similar software.

Different applications of existing diagrams, as well as newly defined decision diagrams often require modifications of existing programming packages to appreciate their peculiar features [85], [123]. To our best knowledge there is no commonly accepted standard for these specific tasks.

With this motivation we propose a robust and flexible standard based on the XML for storage and exchange of various types of decision diagrams.

The inherent properties of XML permit us to develop a framework flexible enough to encapsulate the existing types of decision diagrams [17], [48], [85]. It also provides us with the possibility of extending this standard in the future to accommodate new forms of decision diagrams.

The proposed XML framework consists of several separate components. We list and briefly describe the purpose of each of them. The structure and functionality of these components is discussed in detail in later sections.

The main components of the XML framework are:

1. An XML Schema, a description of the structure of XML documents containing a decision diagram.
2. An XML document, a file representing the decision diagram created based on the XML Schema.
3. An XML parser, a software component capable of reading and writing decision diagrams according to the specifications of the proposed XML Schema.

We begin our discussion by first presenting some basic notions regarding XML data description language in Section 3.1. The general structure of XML documents is presented in Section 3.2. In Section 3.3 we give a brief introduction to XPath, one of the langages from the wider XML family, which is necessary for the understanding of the following sections.

We describe in detail the first and most important component of the proposed XML framework, the XML Schema in Section 3.7.

The second component is in fact a simple XML document that conforms to specifications of the given XML Schema. We present and discuss in detail examples of XML documents for various kinds of decision diagrams in Section 3.8.

The third component represents the implementation of the described standard. The idea behind the whole XML framework for decision diagrams is that a developer of a software that uses and generates decision diagrams would make a module capable of importing and exporting its internal data structures in XML form according to the specification provided by the framework. This component is built on the basis of an existing XML parser but its structure depends on the software platform and application of which it is a part.

XML parsers exist for all major platforms and programming environments [60]. We present an overview of general XML processing software in Section 3.4.

3.1 EXTENSIBLE MARKUP LANGUAGE

The developments of modern information technologies impose demands for handling large amounts of ever more complex information in an efficient manner. The Extensible Markup Language (XML) represents an attempt at addressing these needs.

Historically, the rise of XML as a data description standard has been influenced by the success of HTML. The basic idea behind the introduction of XML was to provide the means of separation of the contents of web documents from the formating and display instructions, the whose inconsistent treatment has been a serious issue of HTML. However, XML has since greatly outgrown its initial intended purpose and is now applied in numerous other fields, whenever the transfer of a large amount of data is required. Some of its inherent properties make it well suited for the task of representation of decision diagrams. We will discuss these properties of XML in the following text.

The W3 Consortium [60] is an international body responsible for the development and maintenance of XML specification and related standards.

XML is a general purpose data description language. It was designed especially for the task of representing data with complex internal structure. XML is based on an earlier Standard Generalized Markup Language (SGML) from which it inherits its basic syntax, a feature which it shares with HTML. More information on SGML can be found in [73] and [80].

XML does not try to specify the exact structure of documents containing data. It is not a single file format trying to encompass all possible varieties of data structures. Rather, XML specifies a set of rules that each document containing a particular type of data needs to conform to. This structure can then be easily recognized and interpreted by software applications.

Pecisely this property makes XML suitable for the purpose of this project. We can specify a set of rules that capture only the common features of the structures of various types of decision diagrams and thus, describe the diagrams in a uniform manner.

Another important feature of XML is human readability. XML represents data in textual form that is, in theory, understandable to humans.

Extensibility is yet another important property of XML. It is classified as an extensible markup language, since users can introduce their own syntax elements. A large variety of special purpose languages has been derived from XML. The list of these languages include RSS, XSLT, MathML, SVG, etc. Some of these languages add functionalities to XML-based systems in general, while others are designed with a specific application in mind. We discuss some of these languages in the following sections.

3.2 XML DOCUMENTS

As mentioned earlier, XML represents data in the form of structured text.

Definition 3.1 *An XML Element is a basic unit of data in XML. It is a certain amount of data presented in textual form and surrounded by textual markup.*

Definition 3.2 *A markup syntax used to indicate the beginning and the end of a portion of data in XML is known as an XML tag. Tags have the following form,* `<tag_name>` *for the tag indicating the beginning of the data sequence, and* `</tag_name>` *for the end tag.*

The list of XML tags is application specific and defined by the user. Each XML document is a hierarchy of different elements as seen in Example 3.1.

Example 3.1 *The hierarchy of XML elements within a document.*

```
<book>
 <title>Hocus Pocus</title>
 <author>K. Vonnegut</author>
</book>
```

These elements can be organized in a recursive manner. As shown in Example 3.2, each element can contain one or more elements of the same type. Decision diagrams also demonstrate strong recursiveness, and we exploit this property of XML documents to a great extent in our proposed system.

Example 3.2 *Nested XML elements.*

```
<node>
 <node>
  <node>A</node>
  <node>B</node>
 </node>
 <node>
 </node>
</node>
```

Additional information can be attached to each XML element using a system of attributes.

Definition 3.3 *An XML attribute is a name-value pair of the following form `name='value'`, attached to the start tag of an XML element.*

For example, we could attach an attribute indicating the genre of the book from Example 3.1:

Example 3.3 *XML element with an attribute:*

```
<book genre='fiction'>
 <title>Hocus Pocus</title>
 <author>K. Vonnegut</author>
</book>
```

XML tags and attributes defined in this way represent the core of XML syntax. Each XML document must conform to the following set of XML syntax rules:

1. Every start-tag must have a matching end-tag.
2. Elements may nest, but may not overlap.
3. There must be exactly one root element.
4. Attribute values must be quoted.
5. An element may not have two attributes with the same name.
6. Comments and processing instructions may not appear inside tags.
7. No unescaped `<`, `>` or `&` signs may occur in the character data of an element or attribute.

Definition 3.4 *If an XML document follows all of the XML syntax rules this document is said to be well-formed.*

These requirements must be ensured by the an XML parser which produces a particular XML document.

3.3 XPATH

Manipulation of XML documents can, and usually does, include a set of specific operations performed over a certain subset of elements of the document. This ability to access specific XML elements is essential. Individual XML elements can be accessed using a special syntax specified by the XML Path Language (XPath).

XPath is an extension of XML syntax designed specifically for this task and introduced by The W3 Consortium [185].

The mechanism of addressing specific XML elements by XPath relies on three concepts:

1. Axis specifiers,
2. Node tests,
3. Predicate.

XPath syntax can be expressed in either expanded or abbreviated form.

Remark 3.1 *The Axis Specifier indicates the navigation direction within the tree representation of the XML document.*

The axes available in XPath are:

Extended	Abbreviated
`child`	default
`attribute`	`@`
`descendant`	`//`
`descendant-or-self`	n/a
`parent`	`..`
`ancestor`	n/a
`ancestor-or-self`	n/a
`following`	n/a
`preceding`	n/a
`following-sibling`	n/a
`preceding-sibling`	n/a
`self`	`.`
`namespace`	n/a

Example 3.4 *Consider the XML code from Example 3.2.*

```
<book genre='fiction'>
 <title>Hocus Pocus</title>
 <author>K. Vonnegut</author>
</book>
```

XPath expression `//title` would select the `<title>` element of the element `<book>`

Remark 3.2 *XPath Node tests are logic expressions that need to be satisfied in order for a particular XML element to be selected.*

Example 3.5 *Consider the following XML code.*

```
<library>
 <book genre='fiction'>
  <title>Hocus Pocus</title>
  <author>K. Vonnegut</author>
 </book>
 <book genre='economy'>
  <title>Freakonomics</title>
  <author>S. Levitt</author>
 </book>
 <book genre='fiction'>
  <title>Londonistani</title>
  <author>G. Malkani</author>
 </book>
</library>
```

The XPath expression `//book/@genre='fiction'` would select all the `genre` attributes of the descendant `<book>` elements with the value `'fiction'`. In this way individual attributes can be selected and modified directly.

Definition 3.5 *XPath predicates are logic expressions used to select particular XML elements according to the value of some of their subelements. Predicates are expressed using the following syntax* [log. expression].

Example 3.6 *Consider again the XML code from the previous example.*

```
<library>
 <book genre='fiction'>
  <title>Hocus Pocus</title>
  <author>K. Vonnegut</author>
 </book>
 <book genre='economy'>
  <title>Freakonomics</title>
  <author>S. Levitt</author>
 </book>
 <book genre='fiction'>
  <title>Londonistani</title>
  <author>G. Malkani</author>
 </book>
</library>
```

The XPath expression `//book[@genre='fiction']` would select all the descendant `<book>` elements with the attribute `genre` set to the value `'fiction'`, Londonistani by G. Malkani and Hocus Pocus by K. Vonnegut.

To accommodate this the XPath possesses a full set of functions and operators including:

A union operator	|
Boolean operators	`and`, `or`, `not()`
Arithmetic operators	`+`, `-`, `*`, `div`, `mod`
Comparison operators	`=`, `!=`, `<`, `>`, `<=`, `>=`

The XPath syntax is an essential part of the transformation mechanism employed by the proposed framework.

Further information about XPath can be found, for example, in [69].

3.4 SOFTWARE FOR PROCESSING XML DOCUMENTS

Function libraries dedicated to processing XML documents exist for all current operating systems, and programming environments.

Remark 3.3 *A software component designed for reading and writing an XML document is called an XML parser.*

Although the functionality of an individual parser implementation may vary from system to system, there are two general parser architectures specified by the W3 Consortium:

1. Simple API for XML (SAX), where API stands for Application Programming Interface.
2. Document Object Model (DOM).

Document Object Model is a standard introduced and developed by the W3 Consortium. Its specification is given in the form of an official W3 Consortium Recommendation [43], in the manner similar to the XML family of languages.

SAX started off as a Java based XML API. There exists no formal definition of SAX. The Java implementation of a SAX parser is considered normative and implementations for other languages and programming environments try to replicate its structure and functionality. A detailed specification of the current Java implementation of SAX can be found in [115]. Additional information about SAX can be found in [15].

The difference between these two architectures is fundamental. While DOM is designed for the ease of manipulation of XML elements, SAX is streamlined toward speed and minimal memory requirements. In general, DOM architecture requires that the whole XML tree is present in system memory during manipulation. This organization permits relatively easy addition and removal of individual elements. It is, thus, efficient in terms of processing time. SAX architecture processes XML documents sequentially. Only a small portion of the document needs to be present in the system at any given time. Memory requirements are usually many times smaller than in the case of DOM. However, the tradeoff is in the difficult manipulation of elements. SAX architecture is therefore, usually chosen for applications where additional processing of XML documents is not required, for read and write only applications, or in cases of severely limited system resources.

We have used a DOM based XML parser provided as a part of the Microsoft .NET framework [58] to generate the examples presented in this document.

There exsists a wide range of books discussing programming methods regarding SAX and DOM, for example, [81] and [113].

3.5 DATA STRUCTURES FOR DECISION DIAGRAMS

Different software packages use different methods for the representing decision diagrams in computer memory. As stated earlier, most of these packages were focused on the particular application of specific classes of decision diagrams and, therefore, focused on particular aspects of these classes. Nonetheless, some general principles do apply. We follow the same basic principles in the design of data structures which form the basis of the proposed framework.

The data structures used for representing decision diagrams are derived from the graph theory interpretation of decision diagrams. A decision diagram is treated as any other acyclic directed graph.

In order to achieve generality of representation, we focus on the common structural features shared by different types of decision diagrams. A data structure model suitable for representing decision diagrams must provide the facilities for the representation of basic diagram elements, i.e., nodes and their interconnections. Node elements must be represented in a uniform way. A place for storage of additional data associated with each node needs to be provided. It is desirable to provide the means for explicit representation of diagram edges, as the explicit representation of edges is necessary for efficient manipulation of decision diagrams.

Furthermore, we must not impose a strict and rigid structure. Instead, we seek a flexible data structure which can be freely extended as needed, and adopted for possible new applications. As the exact structure of each individual decision diagram is determined by the underlying discrete function, the number of parents of each node is not known in advance. Thus, we must not impose an upper limit for the number of incoming edges.

As we are aiming to represent a wide variety of decision diagram classes, we cannot limit the number of children of each node. The total number of nodes in the diagram, number of levels, or the number of nodes per level, must not be limited either.

To accommodate these demands, we propose the following data structure architecture, based on the so-called linked lists of elements.

A linked lists is a well established concept in computer science. It is a dynamic data structure, that can be altered at run time. This type of data structures do not require static allocation of a block of memory of a fixed size, and is, thus, well suited for applications where an upper limit on size cannot be set in advance. Furthermore, they provide the means for efficient data manipulation.

A linked list consists of a set of elements. Each element in the linked list contains one or more slots for storing individual pieces of information, such as numerical or textual values, and a pointer to the memory address of the following element in the list. The last element in the list, the tail of the list, contains a null pointer, thus marking the end of the structure.

New elements can be added to the list, either by appending the tail of the list, by setting the previous null pointer to the address of the new element, or they can be inserted at the beginning of the list. In the second case the pointer of the new element is set to the address of the previous head of the list.

Removal of elements is performed by rearranging the pointers in the list. The pointer of the preceding element is set to the address of the following element. The memory space occupied by the deleted element is marked as unused by a separate junk collector routine. The number of elements in the linked list is limited only by the size of the available memory.

Each node is represented as one entity storing the identifier of the node, optionally the level in the diagram to which this node belongs and other type specific data i.e., the decomposition rule.

Out of these additional properties, only the node identifier is strictly necessary. It serves as an anchor for the edge elements of the diagrams, and unambiguously declares each separate node to manipulation algorithms. It must, therefore, have a unique value throughout the particular diagram.

The information about the level of the decision diagram can be omitted in the case of free decision diagrams, as the notion of diagram levels has no meaning in their context. However, in the case of this class of decision diagrams, we must always explicitly state the logic variable for which the decomposition is performed at each node. The same data slot will be used for this purpose.

Terminal and non-terminal nodes are stored in the same way. The type of node, thus, must be specified. Terminal nodes include the information about the value of the logic constant which they represent. In the general case of multi-terminal decision diagrams, this value is an integer numeric value. Indeed this value can, if necessary, even be symbolic.

The root node is also clearly indicated by the value of its type property.

Information about the descendants of each node is stored in the form of a linked list. Each element in the linked list consists of two fields:

1. A pointer to the child node of the node in question, which represents a real link existing in the decision diagram.

2. A pointer to another field in the linked list which serves the function of internal navigation through the list.

Optionally, information about parents of each node can be stored explicitly. Strictly speaking this is not necessary, as all the needed information to reconstruct the full decision diagram can be derived from the list of nodes and their children. However, the inclusion of this data can prove beneficial, as it greatly simplifies some of the processing algorithms, thus justifying the overhead. This information can always be safely omitted in cases where the memory size of the data structures is a concern. For the sake of generality we continue our discussion assuming that this information is included explicitly.

Thus, two linked lists of the described structure are created and attached to each node in a decision diagram. The lists represent its parents and descendants.

We present an example of a node element in Fig. 3.1.

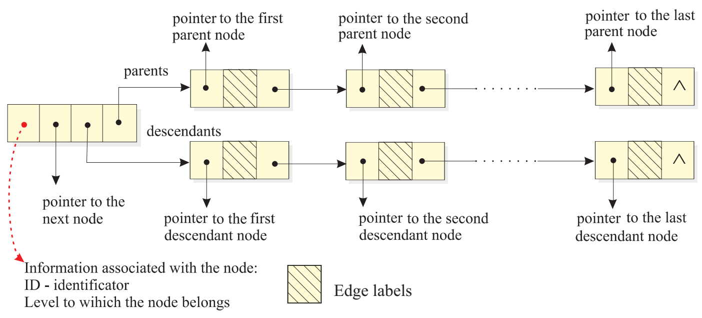

pointer to the first parent node
pointer to the second parent node
pointer to the last parent node
parents
descendants
pointer to the next node
pointer to the first descendant node
pointer to the second descendant node
pointer to the last descendant node
Information associated with the node:
ID - identificator
Level to wihich the node belongs
Edge labels

Fig. 3.1 Diagram of Node data structure.

The terminal nodes do not contain a list of children. The root node of the diagram does not contain the list of parent nodes.

Each edge element in either of these lists can have additional blocks of information associated with it.

The nodes themselves are also stored in the form of linked lists. It is important to point out that the links between the elements of these lists do not have a semantic meaning in the context of the decision diagram itself. Rather, they represent the means of accessing elements in the data structure in the computer memory. Furthermore, the order in which nodes are arranged in this list is not important. It is convenient to arrange the nodes according to one of the recursive tree traversal orders if possible, for example, in the leftmost first traversal order.

The head of the main data structure, the list of nodes, is usually the root node of the diagram. However, this again is a matter of convenience, as the root node is explicitly marked. This is especially important for the representation of shared decision diagrams with multiple root nodes. The head of the list will be set at one of the root nodes, i.e., the first. The remaining root nodes will be stored further down in the list and explicitly marked. They can be processed accordingly by external algorithms.

As an example, we present the data structures for a binary decision diagram of the function $f = \bar{x}_1 + x_1x_2 + x_1\bar{x}_2\bar{x}_3$ in Fig. 3.2. Note that the red lines in the diagram represent the actual decision diagram edges, while the black ones represent internal memory pointers.

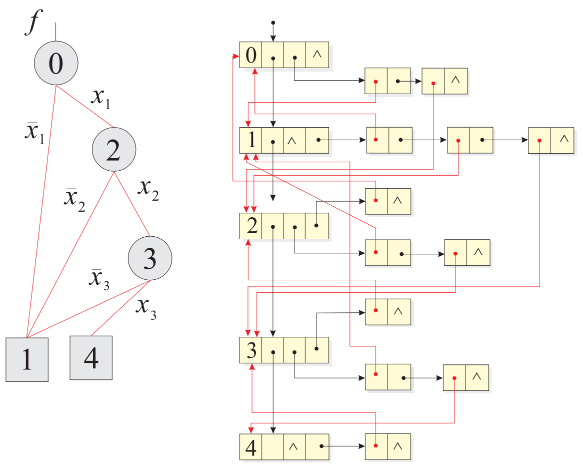


Fig. 3.2 BDD and the corresponding data structure.

This architecture is identical to the standard way that graphs are described in the theory of data structures. This permits us to use the standard, well adapted, and efficient algorithms for the addition and removal of nodes, accessing elements of the graph, etc. It is also consistent with the usual way decision diagrams have been represented in various previous applications [123], [150], [151].

3.6 XML SCHEMA

As stated earlier, XML models data by specifying a set of rules that describe the internal structure of a specific kind of data. Each individual application specific XML document needs to conform to these rules in order to be correctly recognized and processed by the intended software applications.

XML Schema language is a syntax used for the specification of such rules. It is an extension of the basic XML syntax introduced by the W3 Consortium in 2001. Specification of the XML Schema language can be found in [174] and [186].

XML Schema is one of the several languages that can be used for this task. Historically, this language has been introduced to correct the shortcomings of a previous XML specification standard, the Document Type Definition (DTD) language. The most important feature of the XML Schema language from the point of view of this work, is that it permits recursion in the specification of XML elements, essential for modeling decision diagrams. Furthermore, the application of XML Schema language ensures the compatibility of the XML documents with the data types native to object oriented, high-level programing languages, such as C++. This is very important from the point of view of integration of the proposed framework with the existing software packages.

In this section, we present only the details necessary for understanding the implementation of decision diagram data structures. A detailed description of XML Schema language and related programming methods can be found, for example, in [12].

Definition 3.6 *An XML document that conforms to a specific set of rules specified by an XML Schema is said to be a* valid *XML document. An XML document described by an XML Schema is called an instance document.*

The XML Schema document is associated with an instance document through the `xsi:schemaLocation` attribute attached to the header of the document. The XML parser uses the url associated with that attribute to locate the schema and validate the document.

The data model employed by the XML Schema to specify the structure of XML documents consists of:

1. The vocabulary - the XML element and associated attribute names,
2. The content model - relationships between these elements and their internal structure,

3. The data types - defining the content of each individual element and attribute.

There are 19 basic data types supported by the XML Schema languages:

1. string,
2. boolean,
3. decimal,
4. float,
5. double,
6. duration,
7. dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth,
8. hexBinary,
9. base64Binary,
10. anyURI,
11. QName,
12. NOTATION,

These datatypes serve as the basic building blocks of XML elements.

An XML element is specified by an `<xs:element name='fullName' type='xs:string'>` schema element. The attribute `name` specifies the XML tag used for the particular XML element. XML elements can have either a simple content specified by the attribute `type`, as seen in Example 3.7, or a complex content specified separately.

Example 3.7 *The schema element stated as:*

```
<xs:element name='title' type='xs:string'>
```

specifies an XML tag of the following structure:

```
<title>Hocus Pocus</title>
```

The XML elements with complex content can include nested elements. Complex element data types are defined using an `<xs:complexType>` schema element. A set of sophisticated tools for the specification of contents of XML elements is provided by the schema language. The designer can specify the minimal and maximal numbers of occurrences of individual nested elements as well as the order in which they must occur in a particular parent element.

The application of these methods ensures error resilience of the final XML documents.

Example 3.8 *The following schema specification:*

```
<xs:element name='book'>
 <xs:complexType>
  <xs:sequence>
   <xs:element name='title' type='xs:string'/>
   <xs:element name='author' type='xs:string'/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

defines an XML element with two nested elements which must appear in exact order,

```
<book>
 <title>Hocus Pocus</title>
 <author>K. Vonnegut</author>
</book>
```

The minimal number of occurrences of a subelement can be specified using the `minOccurs` attribute, as in Example 3.9.

Example 3.9 *The XML Schema from Example 3.8 can be extended to specify that the title and the author name must occur at least once in the* `<book>` *element.*

```
<xs:element name='book'>
 <xs:complexType>
  <xs:sequence>
   <xs:element name='title' type='xs:string' minOccurs='1'/>
   <xs:element name='author' type='xs:string' minOccurs='1'/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

The XML attributes can have only simple contents, and are specified by an
`<xs:attribute name='language' type='xs:string'/>` schema element, as shown in Example 3.10.

Example 3.10 *To associate the* `genre` *attribute with the* `<book>` *element we extend the XML Schema from Example 3.9:*

```
<xs:element name='book'>
 <xs:complexType>
  <xs:sequence>
   <xs:element name='title' type='xs:string' minOccurs='1'/>
   <xs:element name='author' type='xs:string' minOccurs='1'/>
  </xs:sequence>
 </xs:complexType>
 <xs:attribute name='genre' type='xs:string'/>
</xs:element>
```

The XML element in the instance document has the form:

```
<book genre='fiction'>
 <title>Hocus Pocus</title>
 <author>K. Vonnegut</author>
</book>
```

We employ these basic methods to model the decision diagram data structures, as presented in Section 3.5. We discuss the rules specified this way in detail in the next section.

3.7 DECISION DIAGRAMS AS XML DOCUMENTS

This section deals with the specific XML implementation of the decision diagram data structures described in the previous section. This implementation was originally presented in [159]. We use the mechanisms of the XML Schema system to specify the custom XML data types that correspond to the elements of our architecture.

The core element of the described architecture is a singular node of the decision diagram. In our XML Schema specification we define a basic complex element type `NodeType` in the following manner.

```
<xsd:complexType name='NodeType'>
 <xsd:sequence>
  <xsd:element name='next' type='dd:NodeType' minOccurs='0'
  maxOccurs='1' nillable='true'/>
  <xsd:element name='parents' type='dd:PointType' minOccurs='0'
  maxOccurs='1' nillable='true'/>
  <xsd:element name='children' type='dd:PointType' minOccurs='0'
  maxOccurs='1' nillable='true'/>
 </xsd:sequence>
 <xsd:attribute name='terminal' type='xsd:integer'/>
 <xsd:attribute name='id' type='xsd:integer'/>
 <xsd:attribute name='level' type='xsd:integer'/>
 <xsd:attribute name='constant' type='xsd:integer'/>
 <xsd:attribute name='rule' type='xsd:string'/>
</xsd:complexType>
```

The linked list of nodes is implemented through the use of a recursive declaration in this element type where each `NodeType` element has one nested element of the same type in its structure, i.e., the following element in the list of nodes. Each node element also contains two nested objects of the type `PointType` which represent links to the roots of the linked lists of the parents and the linked lists of the descendants.

The additional information about the node, i.e., the ID and the level to which the node belongs are stored in appropriate attribute elements.

`PointType` elements are defined in much the same way. We use the same mechanism of nested elements to represent links in the linked lists as each `PointType` element of a linked list has one element of the same type as a nested object.

```
<xsd:complexType name='PointType'>
 <xsd:choice>
  <xsd:element name='next_child' type='dd:PointType'
  minOccurs='0' maxOccurs='1' nillable='true'/>
  <xsd:element name='next_parent' type='dd:PointType'
  minOccurs='0' maxOccurs='1' nillable='true'/>
 </xsd:choice>
 <xsd:attribute name='point' type='xsd:integer'/>
 <xsd:attribute name='variables' type='xsd:string'/>
</xsd:complexType>
```

This way of representing the element links and the hierarchy is native to XML and it is understood and supported by all XML parsers, thus allowing us to use this mechanism to automatically generate the corresponding data structures based on an XML document.

However, these links represent only the links that are a part of the structure we designed to represent a decision diagram and are not inherent to the decision diagram itself. The possible complexity of node connections in the decision diagram forces us to use another method. Each `PointType` element contains a field carrying the ID of the actual node to which the actual decision diagram link points to.

These connections cannot be automatically stored or generated in XML and need to be reestablished after an XML document has been parsed. The main reason for this is the possibility of establishing a cycle in the graph which would pose an impassable obstacle for any XML parser. Although closed cycles are not permitted in acyclic directed graphs, there is no simple way to impose this constraint in terms of the XML syntax.

The control of this must be implemented separately while creating the decision diagram. The functions for reestablishing the real node connections in the graph must also be implemented separately based on the XML parser.

The `TreeType` element represents a wrapper element whose task is to encapsulate the decision diagram structure itself and store the additional information regarding the nature of the decision diagram, i.e., the type of the decision diagram, number of variables, nodes, levels, edges, decomposition rules applied, etc. This element stores the additional information either in the form of XML attributes or as additional elements.

```
<xsd:complexType name='TreeType'>
 <xsd:all>
  <xsd:element name='root' type='dd:NodeType'
  minOccurs='1' maxOccurs='1'/>
 </xsd:all>
 <xsd:attribute name='type' type='xsd:string'/>
 <xsd:attribute name='num_levels' type='xsd:integer'/>
</xsd:complexType>
```

Furthermore, not all of these features need to be specified in advance. Additional storage attributes or entire elements can be declared and included in each separate XML file. This feature can be used to adapt the basic system to work with special classes of decision diagrams. These additional attributes and elements can store any kind of data, scalar numerical values, vectors and matrices or string expressions.

For example, since the edges of a diagram are stored explicitly in the form of dedicated XML elements, additional attributes can be associated with each edge. This feature permits us to represent Binary decision diagrams with negated edges, and various other types with attributed edges. In this way we can indicate complemented and weighted edges as encountered in Edge Valued Decision Diagrams (EVBDD) [100], or Multiplicative Binary Moment Diagrams (∗BMD) [19], [20], and various related generalizations [50]. The software system will determine the way these edges are interpreted according to the value of the attribute 'type' specified in the top level element of the XML hierarchy.

An example of an edge element with an aditional 'weight' attribute, is shown in Fig. 3.3:

```
<next_child point='2' weight='4' variables='9'/>
```

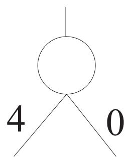


Fig. 3.3 An example of a weighted edge.

This ensures flexibility of the framework by making it open for further extensions and generalizations.

Extensions to the decision diagram XML Schema can also be included. In this way, the XML framework can be specifically tailored for every application.

The XML Schema document specifying the described XML Decision Diagram structure is available publicly for the purpose of validation of individual decision diagram documents. It can be accessed at:

http://www.cs.tut.fi/ stankovs/XML/xmldd.xsd

Furthermore, we provide the complete source code of this document in Appendix A.

3.8 EXAMPLES OF XML REPRESENTATIONS OF VARIOUS CLASSES OF DECISION DIAGRAMS

In this section, we examine examples of various types of decision diagrams represented using the XML-based framework presented in earlier sections. The general principle of the application of the XML framework is explained first on a simple example of a Binary Decision Diagram (BDD). We then extend this discussion to the choice of decomposition rules used to generate a decision tree by examining the use of the XML framework on the Kronecker (KDD) and Pseudo-Kronecker Decision Diagrams (Pseudo-KDDs). We then turn our focus to non-binary types of decision diagrams, especially to Ternary Decision Diagrams (TDDs), and finally to Heterogeneous Decision Diagrams (HDDs) which represent the most general type of decision diagrams in terms of the data structures [123].

Example 3.11 *Consider the function $f(x_1, x_2, x_3) = x_1\bar{x}_2 \vee x_2x_3$ with three logic variables. The corresponding decision diagram is shown in Fig. 3.4. Since the variables are binary-valued, the non-terminal nodes have two outgoing edges. The function also takes two values and, therefore, there are two constant nodes.*

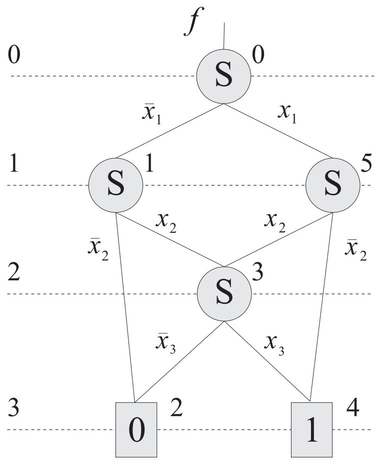


Fig. 3.4 Binary decision diagram in Example 1.

We can observe some of the important structural features of this diagram.

The number of variables in the function determines the number of levels of the resulting BDD. The level attribute associated with each node object in an XML document stores this information as it is very important for many tasks regarding BDDs. The same decomposition rule is applied at every node of the diagram, so there is no need to store this information explicitly.

The total number of nodes in the BDD cannot be determined a priory only by examining the function. By definition, the number of outgoing edges for each node is always two for every non-terminal node. However, the number of incoming edges cannot be predicted based on the form of the function only. The flexibility of the framework we propose permits us to easily describe this kind of structure. The linked list associated with each non-terminal node representing its descendants has exactly two elements. The number of elements of the linked list that stores the information about the incoming edges of the node is not limited. One such list will be associated with each node either terminal or non-terminal.

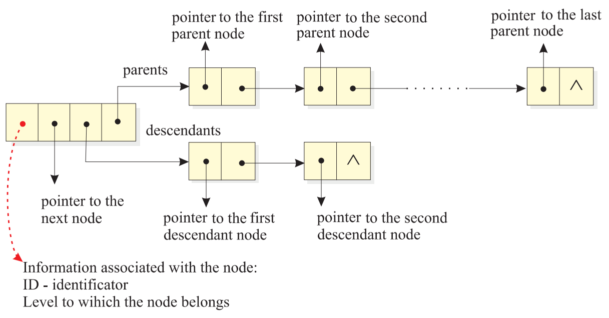

pointer to the first parent node
pointer to the second parent node
pointer to the last parent node
parents
descendants
pointer to the next node
pointer to the first descendant node
pointer to the second descendant node
Information associated with the node:
ID - identificator
Level to wihich the node belongs

Fig. 3.5 Data structure for a non-terminal node.

Fig. 3.5 represents the structure of a non-terminal node and two linked lists associated with that node. The structure of a terminal node with a linked list that stores the information about its parents is shown in Fig. 3.6.

Terminal nodes of the diagram have one additional attribute field named 'constant' which holds the value of the logic constant associated with them.

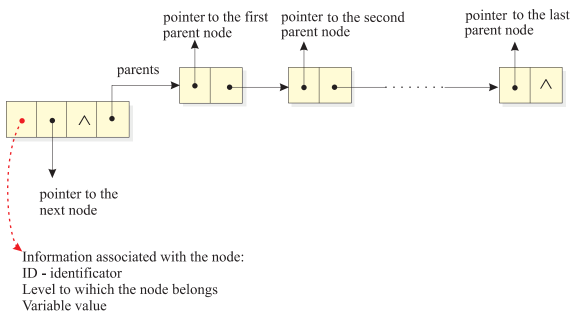

pointer to the first parent node
pointer to the second parent node
pointer to the last parent node
parents
pointer to the next node
Information associated with the node:
ID - identificator
Level to wihich the node belongs
Variable value

Fig. 3.6 Data structure for the terminal node.

The root node of the decision diagram does not have parents and, therefore, will have only one linked list associated with it as can be seen in Fig. 3.7.

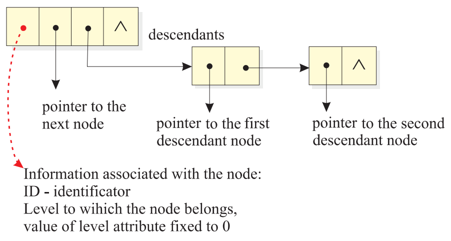

descendants
pointer to the next node
pointer to the first descendant node
pointer to the second descendant node
Information associated with the node:
ID - identificator
Level to wihich the node belongs,
value of level attribute fixed to 0

Fig. 3.7 Data structure for the root node.

To better illustrate this example, we present the complete XML code representing this particular binary decision diagram:

```
<tree Type='BDD' num_levels='4'>
 <root id='0' level='0'>
  <next id='1' level='1'>
   <next id='2' level='3' constant='0'>
    <next id='3' level='2'>
     <next id='4' level='3' constant='1'>
      <next id='5' level='1'>
       <parents point='0'/>
       <children point='3'>
        <next_child point='4'/>
       </children>
      </next>
      <parents point='3'>
       <next_parent point='5'/>
      </parents>
     </next>
    <parents point='1'>
     <next_parent point='5'/>
    </parents>
    <children point='2'>
     <next_child point='4'/>
     </children>
     </next>
    <parents point='1'>
     <next_parent point='3'/>
    </parents>
   </next>
   <parents point='0'/>
   <children point='2'>
    <next_child point='3'/>
   </children>
```

```
    </next>
  <children point='1'>
   <next_child point='5'/>
  </children>
 </root>
</tree>
```

Note the absence of the `<parents>` *sub-element in the root node (ID=0) and the* `<children>` *sub-elements in terminal nodes (ID=3, ID=4).*

In Section 2.4 we have discussed the representation of multi-output discrete functions using shared decision diagrams.

We can imagine a single shared decision diagram as a set of mutually overlapping single output decision diagrams sharing the same inputs and parts of their structure. The main difference would be the existence of several root nodes, each corresponding to one output. From the point of view of data structures, the only difference between shared and single output decision diagrams is that shared decision diagrams have more than one root node, that is more than one node at the level 0.

The internal structure of XML documents is determined by the organization of linked lists used to represent a BDD. One single node represents the head of the main linked list. It does not necessarily follow that this particular node is the only root of the underlying decision diagram. The number of nodes on each level of the diagram is not limited and the same applies to the level 0. Edges, the connections between nodes, are stored explicitly, and the number of descendants and parents of the node is not limited. This number can be 0 for any node in the main linked list. Therefore the representation of Shared decision diagrams fits naturally into the proposed framework. The root node of the linked list, the topmost node in the XML hierarchy, is simply the first node from which processing of the diagram begins.

Example 3.12 *Consider the following functions, $f_1 = x_0\overline{x}_1$, $f_2 = x_0 \oplus x_1$, $f_3 = x_0 \vee \overline{x}_1$. These functions can be represented by a Shared binary decision diagram shown in Fig. 3.8. The data structures that represent this diagram are presented in Fig. 3.9. Notice that the nodes* 1, 5, 7 *are root nodes of the decision diagrams corresponding to functions f_1, f_2, f_3. The starting point of the data structure is the node* 1. *The other two root nodes appear deeper in the linked list of the nodes of the decision diagrams.*

In the previous examples, we have examined a decision diagram which was obtained by the recursive application of the Shannon decomposition rule. However, in Section 2.4 we already stated that this class of diagrams can be seen as a particular case of a wider class of Functional Decision Diagrams.

We now turn our attention to two additional classes of Functional Decision Diagrams.

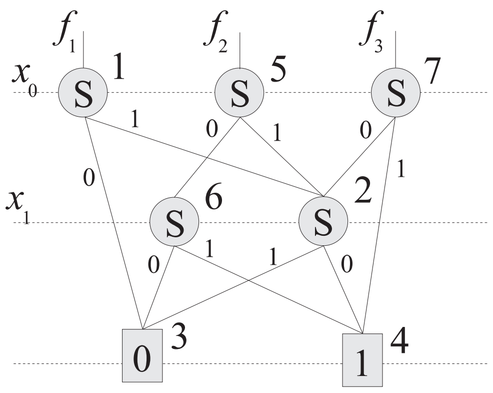


Fig. 3.8 An example of a shared binary decision diagram.

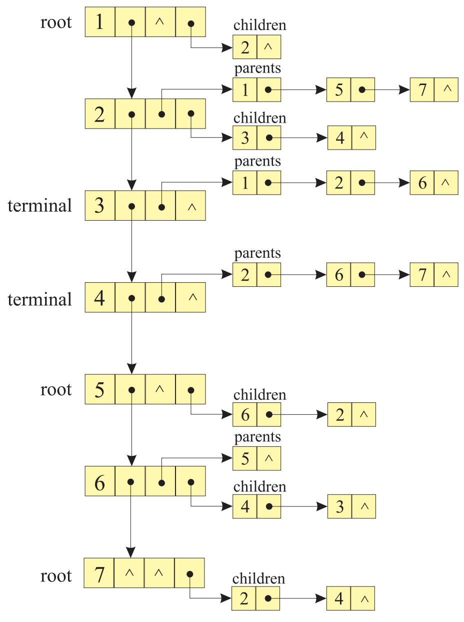


Fig. 3.9 Data structures representing a shared decision diagram.

1. Kronecker decision diagrams (KDDs) where one of the three possible decomposition rules is chosen freely for each variable (`level`) of the decision diagram.
2. Pseudo-Kronecker decision diagrams (Pseudo-KDDs) where the decomposition rule is chosen freely for each node of the decision diagram irrespective of other nodes at the same level of the diagram.

Example 3.13 *In Fig. 3.10 and Fig. 3.11, we give examples of one KDD and one Pseudo-KDD respectively.*

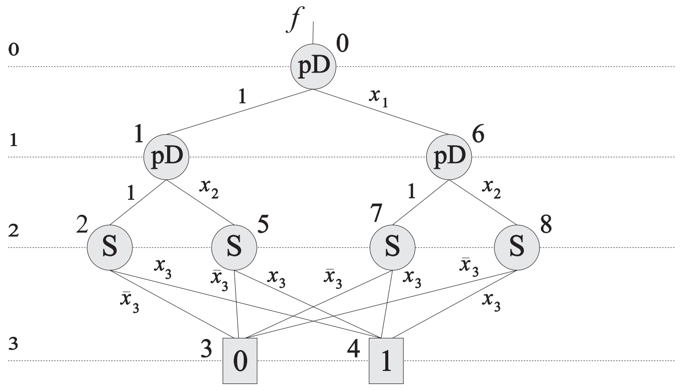


Fig. 3.10 A Kronecker decision diagram.

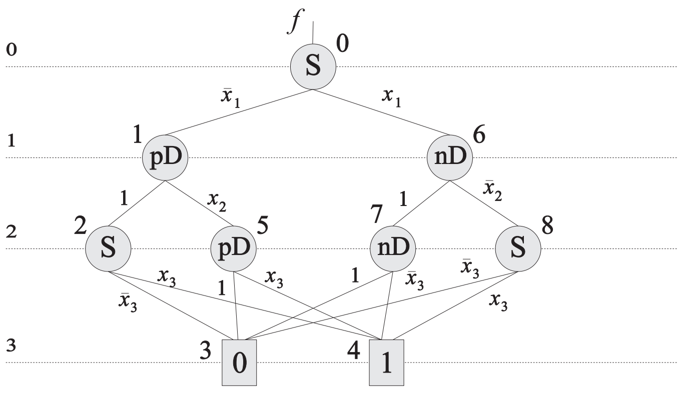


Fig. 3.11 A Pseudo-Kronecker decision diagram.

Notice that the basic structure of these diagrams is similar to that of the BDD we have presented. This fact is reflected in the identical data struc-

tures that are used to describe these diagrams in the XML form. Our point of interest here is that the information about the applied decomposition rule cannot be transferred implicitly but needs to be specified explicitly in the XML file. We achieve this by associating the additional 'rule' attribute with each node element in our XML structure. This attribute can take any of the following string expressions as valid values: 'Shannon', 'pDavio', 'nDavio'. Furthermore, additional expressions can be defined by the user when needed, provided that there is implemented functionality to appropriately treat the elements with these values in the software system that integrates a version of our framework.

Each element of the <parents> and <children> linked lists contains the attribute 'variables' which is used to explicitly state the name of the variable associated with each outgoing edge of a particular node. This feature is available for any type of DDs. However, it is explicitly used in the case of KDDs and Pseudo-KDDs since the nature of the variables of outgoing edges depends on the decomposition rule applied at any given node. This attribute can take any valid string expression as its value, which can be useful when working with various types of multi-valued logic DDs where one edge can be associated with multiple values of a particular logic variable.

We present a partial XML code with one typical non-terminal node from the previous two examples to illustrate the described properties:

```
<tree Type='Kronecker BDD' num\_levels='4'>
 <root id='0' level='0' rule='pDavio'>
  ...
   <next id='8' level='2' rule='Shannon'>
    <parents point='6' variables='x2'/>
    <children point='3' variables='x3\_comp'>
     <next\_child point='4' variables='x3'/>
    </children>
   </next>
  ...
 </root>
</tree>
```

All the above examples share the same basic structure of binary decision diagrams. We used the KDD and the Pseudo-KDD just to illustrate the way additional information about the nature of the decision diagram can be attached to the basic structure. In the next example we demonstrate the ability of our framework to describe decision diagrams with non-binary structure, i.e., decision diagrams for multiple-valued functions.

We examine Multiple-valued Decision Diagrams (MDDs) for three-valued logic functions. As proposed in [142], these diagrams will be called the Ternary Decision Diagrams (TDDs), and should not be mixed with the ternary decision diagrams for functions of binary-valued variables, which can also be represented by the proposed XML structures.

Example 3.14 *Fig. 3.12 shows a ternary decision diagram. Since it represents a function with ternary-valued variables, the nodes in this diagram have three outgoing edges. The function may take three different values and, therefore, there are three constant nodes.*

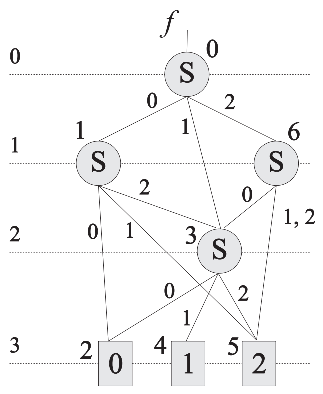


Fig. 3.12 Example of a ternary decision diagram.

The main constraint imposed on the structure of the TDD is the number of outgoing edges of which there can be at most three for each non-terminal node of the diagram. The main difference with the BDD in the first Example is that here the linked list which stores the information about the descendants of this node will have three elements. The number of incoming edges to each node is unlimited as was the case with all previous types of decision diagrams, thus, we can use the identical data structure for this task.

Fig. 3.13 shows the structure of a non-terminal node in the TDD with associated data structures.

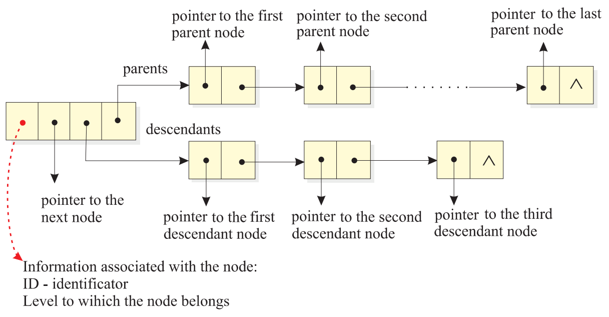

pointer to the first parent node
pointer to the second parent node
pointer to the last parent node
parents
descendants
pointer to the next node
pointer to the first descendant node
pointer to the second descendant node
pointer to the third descendant node
Information associated with the node:
ID - identificator
Level to wihich the node belongs

Fig. 3.13 A Non-terminal node for a ternary decision diagram.

We present a partial XML code representing node number 1 in the previous example to illustrate the complex structure of the children list associated with this particular node:

```
<tree type='bdd' num_levels='4'>
 <root id='0' level='0'>
  ...
   <next id='1' level='1'>
    ...
    <parents point='0' variables='0'/>
    <children point='2' variables='2'>
     <next_child point='4' variables='5'>
      <next_child point='5' variables='3'/>
     </next_child >
    </children>
   </next>
  ...
 </root>
</tree>
```

We can note that here the set of valid values that the attribute `‘constant’`*, associated with each terminal node, can take consists of three numerical values (0, 1, 2), which is consistent with the definition of TDD.*

Another interesting point with this example is that the edge connecting the sixth node (`ID=6`*) with the non-terminal node number 5 (*`ID=5`*) is associated with two values of the logic constant x_1. The* `‘variables’` *attribute of the corresponding element in the XML document is associated with a vector containing all these values (1, 2) and it is given in the form of a string expression. This feature permits us to avoid unnecessary declaration of two separate edge elements for two different values of a logic variable.*

In this case we have used the TDD generated by the application of the generalized Shannon decomposition rule at all the nodes of the decision diagram. However, this is just one class of bit-level ternary decision diagrams. All matters related to the freedom of choice of the decomposition rule discussed in the examples with KDDs and Pseudo-KDDs apply here. These issues are addressed in an identical way through the use of the `‘rule’` attribute attached to each node in the diagram.

Finally, we present the most general example, Heterogeneous decision diagrams [85], where there is no constraint on the number of outgoing edges per each node.

Example 3.15 *Fig. 3.14, is a heterogeneous decision diagram. In this diagram the node for the variable x_1 has 8 outgoing edges, four of them pointing to the constant node 0, three to the constant node 1, and an edge points to the node for the variable x_2. This node has two outgoing edges.*

The proposed XML framework is flexible enough to encapsulate even this type of decision diagrams. The information about outgoing edges of each node

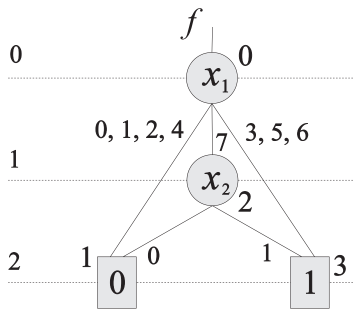


Fig. 3.14 A Heterogeneous decision diagram.

will be stored in the same kind of linked list with an unlimited number of elements as is the case of incoming edges whose number is not limited.

The structure of a non-terminal node in a heterogeneous decision diagram with the attached linked lists is shown in Fig. 3.15.

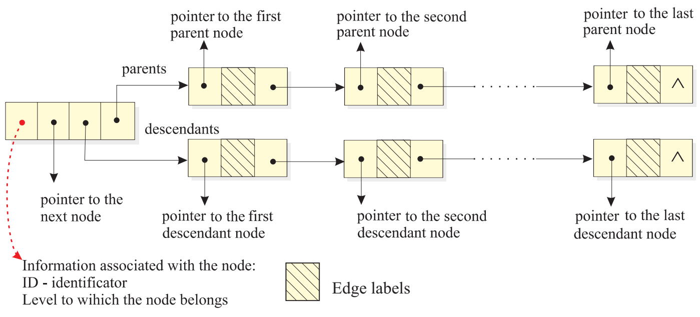

pointer to the first parent node
pointer to the second parent node
pointer to the last parent node
parents
descendants
pointer to the next node
pointer to the first descendant node
pointer to the second descendant node
pointer to the last descendant node
Information associated with the node:
ID - identificator
Level to wihich the node belongs
Edge labels

Fig. 3.15 A non-terminal node in a heterogeneous decision diagram.

The structure of the terminal nodes in heterogeneous decision diagrams is almost identical to the structure in the previous examples with the only difference that now the ‘`constant`’ *attribute can take any valid string expression as its value. The user can also exploit the* ‘`rule`’ *attribute attached to each of the nodes to specify the decomposition rule as needed, in the same way as in the previous cases.*

Again we present the XML code of one of the nodes in the heterogeneous decision diagram from the previous example:

```
<tree type='hmdd'>
 <root id='0' level='0'>
  <next id='1' level='2' constant='0'>
   ...
    <parents point='0' variables='0 1 2 4'>
     <next_child point='2' variables='7'/>
    </parents>
   ...
 </root>
</tree>
```

3.9 MEMORY REQUIREMENTS

Although the exact memory requirements for an XML-based representation of decision diagrams depends on the class of the represented decision diagram and the amount and type of additional information that needs to be explicitly included, some general remarks can be made. Primary concerns driving the development of XML and all the systems derived from it are flexibility and human readability.

XML documents are textual documents. Compactness of representation is not a strong point of any XML-based file format, because this representation is not intended for such purposes. They might posses some memory overhead in comparison to any application-specific binary format for the representation of decision diagrams. There are two ways to somewhat reduce this problem. The first approach would be through the use of some general lossless compression algorithm on final XML documents. The compressibility of XML documents is discussed in, [55]. The other, perhaps more desirable method would be the use of Binary XML, a newly proposed XML based standard. However, at the moment, this standard is still at the development stage. We expect to exploit its promissing features as soon as it matures. For further reference, please, see [184].

3.10 TIME CONSIDERATIONS

Another important issue regarding the manipulation of decision diagrams is the required processing time. In the case of the framework proposed in this thesis, this question is related to a more general discussion of the required processing time for large XML documents. As mentioned earlier, the third component of the proposed framework, the XML parser, is charged with the task of processing actual XML documents. There are a number of factors influencing its time efficiency. Those are the choice of software platform, i.e. the combination of the operating system and the programming language.

The second important factor is the architecture of the XML parser itself. XML as a standard was designed with the representation of very large amounts of data in mind. Two distinct XML processing architectures have been proposed by the W3 Consortium, DOM and SAX. DOM represents a more robust platform permitting an arbitrary modification of the structure of the XML document. It does, however, require that the whole document is at the same time present in the memory of the system. On the other hand, SAX provides a mechanism for reading and processing XML documents in a sequential manner. It requires only a part of the document to be present in the system buffer at a time. Hence, the SAX model is likely to be faster and to require less memory. Furthermore, a very large decision diagram could be continuously streamed from the source application to the destination application and processed in real time. The processing time is determined by the choice of the processing architecture. A potential user is presented with a choice of two architectures and can select the one which better suits his particular application.

3.11 XSLT

From its conception, XML was intended to be a meta language, a basis upon which other specific syntaxes can be built. Different data formats are defined following the basic rules of the XML syntax. Data exchange is one of the priorities and the key aspects behind the introduction of XML. The ability to easily convert data from one XML-based syntax to another is an important feature of XML.

The aim of the XML-based framework is to provide an uniform way of representing various types of decision diagrams. In order to attain this universality, the XML based framework focuses on common elements of the structure shared by all types of decision diagrams. This form of representation is therefore abstract by nature. It does not exploit particular properties of individual classes of decision diagrams. In order to make the practical use of decision diagrams represented in such a manner, these abstract XML documents need to be converted into some application-specific format.

We discuss the relationship between the XML-based framework and application software in detail in Chapter 5.

Extensible Stylesheet Language Transformations (XSLT) is a member of the XML family of languages especially dedicated for the task of converting one XML-based format into other data description formats. Primarily, the target formats of the conversion process are intended to be XML-based, however XSLT is capable of converting XML data into any hierarchical text form.

XSLT was introduced in 1999 in the form of the W3 Consortium recommendation [180]. Although its primary aim is conversion between different forms of XML, XSLT is capable of converting XML documents even to non-XML file formats such as, for example, VHDL, a hardware description language widely used in logic design.

In this section we give a short introduction to basic XSLT concepts and the programming tool set.

Figure 3.16 demonstrates the position of the XSLT mechanism in the process of converting data between two XML-based formats. The first step in this process is to establish the relationship between elements of the original XML hierarchy and equivalent elements of destination hierarchy. Notice that this relationship need not one to one. One element from the original hierarchy can be represented by several elements in the destination hierarchy and vice versa.

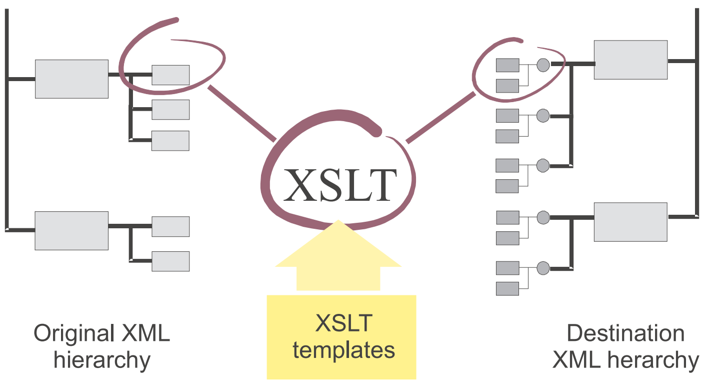


Fig. 3.16 The position of the XSLT mechanism in the process of conversion of data between two XML-based formats.

XSLT provides a mechanism for efficient modeling of these relationships. It is a declarative language. It does not state a program consisting of a deliberate sequence of instructions. Rather, XSLT defines a set of template rules which define the relationships between the entities of two XML based formats. A collection of XSLT templates that completely specifies the relationship between two XML syntaxes represents an *XSLT stylesheet*. The order in which template rules are applied is not important and can be either sequential or recursive depending on the nature of the source XML document and the way it is parsed.

The XSLT processor analyzes the source XML document. It identifies the token elements and transforms them according to the template rules specified in the XSLT stylesheet. A new XML document is generated as an output.

Example 3.16 *Consider for example the following XML document.*

```
<book>
 <title>
  Londonistani
 </title>
 <author>
  G. Malkani
 </author>
</book>
<book>
 <title>
  Hocus Pocus
 </title>
 <author>
  K. Vonnegut
 </author>
</book>
<book>
 <title>
  The Curse of Lono
 </title>
 <author>
  H. S. Thompson
 </author>
</book>
```

This document consists of a series of `<book>` elements. Each `<book>` in turn consist of `<title>` and `<author>` elements. These elements form a hierarchy as shown in Figure 3.17.

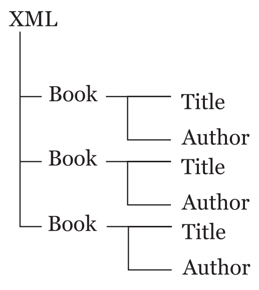


Fig. 3.17 Hierarchy of XML elements.

We can convert this sample XML document into a new document using the following set of XSLT templates.

The conversion process will proceed as follows. The XSLT processor starts to analyze the document from the top of the hierarchy. For each `<book>` element the XSLT processor applies, for example, the following template:

```
<item><xsl:value-of select='title'/> by <xsl:value-of select='author'/></item>
```

producing an `<item>` element in the output document. Values of the `<author>` and `<title>` elements are copied.
Finally, the output XML document will have the following structure:

```
<item>Londonistani by G. Malkani</item>
<item>Hocus Pocus by  K. Vonnegut</item>
<item>The Curse of Lono by H. S. Thompson</item>
```

Although XSLT as a language does not adhere to the principle of the so-called structural programming, it does offer a set of instructions for the full control of the execution flow of a style sheet. Depending on the value of an attribute of an element, the execution flow and the final output of the XSLT transformation process can be altered. These instructions include `<xsl:if>` and `<xsl:choose>` instructions.

Example 3.17 *The following XML document represents a list of products.*

```
<product price='160'>
 <type>
  Portable Game System
 </type>
 <name>
  Nintendo DS
 </name>
</product>
<product price='270'>
 <type>
  Game System
 </type>
 <name>
  Nintendo Wii
 </name>
</product>
<product price='216'>
 <type>
  Portable Game System
 </type>
 <name>
  Play Station Portable
 </name>
</product>
```

We can generate a list of possible shopping items taking the price as the criterion, using the following XSLT template:

```
<xsl:template name='cheap_stuff' match='products'>
<xsl:if test='@price &lt; 250'>
 <item>
  <type>
   <xsl:value-of select='type'/>
  </type>
  <name>
   <xsl:value-of select='name'/>
  </name>
 </item>
</xsl:if>
</xsl:template>
```

This template checks the value of the `price` *attribute of every* `<product>` *element. If this value is smaller then 250, the template will copy the element to the list of shopping items. The new output XML document will have the following form:*

```
<item price='160'>
 <type>
  Portable Game System
 </type>
 <name>
  Nintendo DS
 </name>
</item>
<item price='216'>
 <type>
  Portable Game System
 </type>
 <name>
  Play Station Portable
 </name>
</item>
```

Furthermore, the value of any attribute can be evaluated using complex logic expressions. For logic evaluation, the XSLT relies on the XQuery language, another member of the XML family. All of the standard logic operators are supported by XSLT and XQuery.

`eq`	$=$
`ne`	$!=$
`lt`	$<$
`gt`	$>$
`le`	$<=$
`ge`	$>=$

The XSLT language also includes a set of basic arithmetic operators for addition, multiplication, subtraction, and division of numerical data.

Example 3.18 *Consider a list of geometric objects given in XML form.*

```
<rectangle>
 <width>
  3
 </width>
 <height>
  4
 </height>
</rectangle>
<circle>
 <radius>
  7
 </radius>
</circle>
</rectangle>
```

By the application of the following XSLT template we can calculate the surface areas of these objects.

```
<xsl:template name='surf' match='rectangle|circle'>
 <xsl:choose>
  <xsl:when test='rectangle'>
   <object type='rectangle'>
    <area><xsl:value-of select='width * height'/></area>
   </object>
  </xsl:when>
  <xsl:when test='circle'>
    <object type='rectangle'>
     <area><xsl:value-of select='radius * radius * 3.14'/></area>
    </object>
  </xsl:when>
 </xsl:choose>
</xsl:template>
```

For details on the development of XSLT stylesheets we refer to [86] and [106].

In our work we make extensive use of XSLT mechanism. We present several examples of the application of the XML-based framework in Chapter 5. Each of these examples relies on a different set of XSLT style sheets to convert the decision diagrams represented in the abstract form to a specific output format. These formats range from hardware descriptions using VHLD and EDIF syntax to SVG-based graphic representations of decision diagrams, demonstrating the versatility and flexibility of the XML approach.

4
Applications of Decision Diagrams

In this chapter, we briefly review these applications of decision diagrams that have been the motivation for the development of the XML framework. In the latter part of the chapter we discus the role of the proposed framework in such applications.

Tree-like structures, such as acyclic directed graphs, are widely applied in computer science. The inherent flexibility of such data structures offers, in some cases, significant advantages over other ways of organizing data. To properly understand these advantages we examine the relationships between graphs and other common data structures.

A certain hierarchy of data structures can be established, regarding their complexity and expressive power.

The basic building block of this hierarchy is a singular piece of data which possesses a certain semantic value, for example, an isolated number or a single word. Such individual pieces of data rarely have any significance if not placed in the proper context i.e., other pieces of data. A one dimensional vector is a more complex data structure. The elements of the vector can be completely independent. However, if the value of a certain element depends on one or more neighboring elements, such a structure is known as the Markov chain [74], [84]. By adding more dimensions, we derive the next in this hierarchy of data structures, a matrix notation. Again, elements can be either statistically independent or dependent on the neighboring elements. In this case we are dealing with Markov fields [70]. However, in many cases, the value of a certain piece of data is not determined by its immediate neighborhood, but rather by groups of distant elements. One such example would be the dependence of the semantic value of words in a complex sentence in human speech. In such cases, the previously described matrix forms do not suffice. Such complex interdependences of data are best expressed in terms of graph-like structures.

However, the flexibility of graph-like structures carries an inherent problem. Many tasks, such as data retrieval or insertion, that can be achieved with algorithms of linear complexity in matrix structures, become practically intractable in most general graph-like forms. For example, finding an optimal path through a graph is a NP complete problem. To overcome some of these problems, we impose certain restrictions on the structure of graphs. This leads us to acyclic directed graphs.

Tree-like structures are an optimal, in terms of memory and processing time, data structures for many problems.

In this work, we devote most of our attention to the application of decision diagrams in logic design and signal processing.

Decision diagramsform a widely used tool in logic design. Their main advantage is the flexible way of representing logic circuits that permits efficient optimization of the final design. This optimization can be performed with respect to several factors:

1. The number of gates,
2. Chip area,
3. Energy consumption,
4. Delay,
5. Clock period, etc.

Most of these criteria can be expressed in terms of the he properties of decision diagram properties. For example, the number of gates is directly related to the number of nodes in the corresponding decision diagram, and the delay of the circuit is related to the path length.

4.1 LOGIC CIRCUIT MINIMIZATION

A great part of our work is related to the application of decision diagrams to the minimization of circuit designs.

The idea of applying biary decision diagrams for logic circuit minimization can be traced back to work by Lee in 1959 [102]. Concept was futher explored in papers by Ubar [170] and [13] in 1976, and Akers [4], [5].

Reduced Ordered Binary Decision Diagrams as a method of representation of switching circuits were formalized by Bryant in [17]. Various other classes of decision diagrams have been introduced to represent different classes of switching functions more efficiently.

The idea behind the application of decision diagrams as a minimization tool stems from the fact that a reduced ordered decision diagram is a canonic representation of a given discrete function. Therefore, a completely reduced decision diagram is a minimal representation of a function.

The topological properties of decision diagrams map relatively well into the features of logic circuits. ROBDDs have been a logical choice for the representation of switching function due to the properties of the underlying algebraic structure. The choice of the algebraic structure has a direct influence on the number and complexity of the non-terminal nodes in a decision diagram, as well as the complexity of the interconnections. The number and the complexity of the nodes is reflected on the number of gates and the surface area of the final design. The complexity and the lengths of the paths in a diagram determines the complexity of the interconnections and the delay of the final design.

The choice of a proper underlying structure is, therefore, crucial for the efficient implementation of a circuit design. This choice is determined by the type of the implemented discrete function and the properties of the technology onto which the final design is going to be mapped. For example, Quaternary Decision Diagrams have been proposed in [144] as the optimal method for representing switching functions implemented using 6-input LUT FPGA architectures. We discuss this particular problem in detail in the Section 5.13.

Furthermore, the choice of the decomposition rules determines the number of non-terminal nodes in the decision diagram. The application of Functional Decision Diagrams is justified by this assumption. Use of Kronecker and Pseudo Kronecker Decision Diagrams can significantly reduce the complexity of the circuit design in some cases, see [48], and [10]. We must keep in mind, however, that each new degree of freedom that leads to more flexibility in the design and, therefore, to the greater expressive power of a particular class of decision diagrams, leads also to more complex processing. It has been shown that the selection of optimal decomposition rule for each level in the case of Kronecker or each node in the case of Pseudo Kronecker Decision Diagrams is a NP-complete problem.

Various heuristic algorithms have been proposed as a solution to this problem, [40], [79].

4.2 SENSITIVITY ANALYSIS AND TEST PATTERN GENERATION

Another important step in hardware design is the verification of the gate level designs. A variety of errors may occur in digital circuits due to different reasons, such as short circuits, broken lines, wrong voltage levels. These errors can be divided into two groups.

Soft errors are results of transient effects like electric noise from power supply or other electronic devices present in the immediate environment. These errors may or may not reoccur on a regular basis. Therefore, this kind of errors are hard to test and detect. However, a soft error implies non-permanent fault in the operation of the circuit.

Hard errors are errors which imply a permanent damage to the structure and behavior of the device, and will reoccur every time for the same input pattern. They may be caused by various mechanical effects, such as vibration, corrosion, metal migration, etc.

Even though various models of errors exist, the most commonly used model is a single *stuck-at 0/1 fault*. In short, this model assumes that a single input of the logic circuit is stuck at a constant value. Depending on the nature of the switching function implemented by a particular circuit, some errors covered by this model cannot be detected. For example, for a stuck at 0 fault we cannot detect at which of the inputs of an AND circuit it appears. The same holds for the stuck at 0 fault at the output of the AND circuit. These faults are called equivalent since they produce the same effect although they appear at different positions within the circuit.

The so-called multiple faults, can be seen as a set of individual single errors occurring simultaneously at different inputs of the device. The detection of such errors is all the more difficult as the number of different possible errors grows exponentially.

A test for an error requires an assignment of a set of values at the inputs of the device, that produces an output different from the desired output. In order to detect an error when access to the system is restricted to its inputs and outputs, it is also necessary to provide the conditions that the error propagates from the place of appearance to the output of the system. Since the enumeration of all possible input patterns is impractical for most of the devices, test pattern generation methods have an important place in the field of circuit verification [79]. These methods are based on two distinct approaches:

1. the algebraic, which deals with the Boolean expression representation of the switching function,

2. the topological, which is related to the topology of the particular gate level implementation.

One of the frequently used approaches for test pattern generation is based on the Boolean difference. This approach was first suggested in [147] by Sellers et al. Marinos et al. developed this method in [109]. This work has been further extended in multiple papers by various authors, in particular, by Larrabee [101]. For further details about the Boolean difference, please, refer to [9].

The theoretical work on the problem of testing digital circuits gave rise to the notion of easily testable devices.

To be easily testable, a device needs to satisfy certain criteria:

1. the set of test sequences should be as short as possible,
2. the gate network cannot be statically redundant, that is, it should not contain repeated parts of circuitry,
3. test sequences should be easily generated,
4. tests should also detect the location of the error, if possible.

It has been shown by Reddy [135] that such circuits can be generated using Positive Polarity Reed-Muller expansions, therefore, Functional Decision Diagrams can be used as a tool for the development of easily testable devices.

However, PPRM-based devices have certain problems. For example, certain functions can have a large number of products, and therefore, require a large number of AND gates. Furthermore, multiple errors cannot be detected. As a solution for this problems Generalized Reed Muller expressions have been proposed in [140].

More information about digital circuit testing can be found in [1], [66], [67], [79].

In Section 5.12 we present several examples of XML representation of FDDs.

These examples illustrate the possibilities of applications of the proposed XML framework and a concrete way of how it can be exploited in this context.

4.3 PROBABILISTIC ANALYSIS OF DIGITAL CIRCUITS

Given a switching function that models conditions under which some event will occur, the probability of this event is equal to the probability that the output of the function will take the value of 1.

This probability depends on the probability of individual logic variables. For example, assume that the probability $x_i = 1$ is $p(x_i) = 0.5$:

1. the probability of a two-input AND circuit is equal to 1/4,
2. the probability of a three-input OR circuit is 7/8,
3. the probability of a two-input XOR circuit, 2/4 etc.

Assume that all input variables are independent and have values 0 and 1 with probability 1/2. Then the probability of the event that $f = 1$ is also known as the *function density*. It can be easily shown that the density of a switching function can be computed using the Shannon expansion.

Therefore, the density can be calculated by traversing the ROBDD of the given function. As with many other methods involving the traversal of a decision diagram, there are two possible approaches, top-down and bottom-up.

In [165], Thornton and Nair have presented a top-down approach to compute the density of a function represented by a ROBDD. Although very intuitive, this approach has certain shortcomings:

1. The Binary Decision Diagram must be constructed beforehand, and a separate traversal routine is used to calculate the output probability.
2. The partial probability assigned to a non-terminal node in the diagram, does not represent the output probability of the subfunction rooted in that node.
3. As a consequence of the property 2, the algorithm is not applicable to the Shared Binary Decision Diagrams. A separate traversal of the diagram is necessary for each function.

Most of these problems can be overcome by applying the bottom-up approach [116]. This method is based on the following observations. Assume again that the probability of each input $p(0) = p(1) = 0.5$, and that they are independent. Since each non-terminal node implements the Shannon expansion, the probability assigned to that node is $1/2$ of the sum of the probabilities assigned to its two children nodes. Therefore, the probability assigned to each node will remain constant for every function represented in a Shared BDD. These probabilities can be calculated and assigned at the moment when a node is created. Construction of a ROBDD and the calculation of probabilities can be done simultaneously.

Either of the above methods can be adapted to take into account complemented edges in a diagram. A complemented edge points to the complement of the particular subfunction and clearly has output probability $1 - p(f(x_i))$. Both of these approaches can easily be modified for cases where $p(0) \neq p(1)$.

The density was initially used by Parker and McCluskey in [128] to evaluate the effectiveness of the random testing of combinational circuits. In recent years, the calculation of output probabilities has found a new application as a method for estimating the power dissipation in circuits. We discuss this application of decision diagrams in greater detail in Section 6.

Another direction of decision diagram driven probabilistic circuit verification is focused on the probabilities of sets of states of transition systems. The probabilistic model checker PRISM, developed by Kwiatkowska et al. [99], makes use of this approach.

4.4 CROSS-CORRELATION OF FUNCTIONS

Cross-correlation is an important concept in signal processing. In essence, it is a measure of similarity of two signals. The cross-correlation has a special role in spectral techniques as it can be used as a tool for calculating spectral coefficients.

Consider the cross-correlation of the truth vectors of two functions f and f_c, [95]. By properly choosing the f_c we are able to compute the spectral coefficients of various functions [90], [164]. A generalized formula for spectral coefficients has the following form [165]:

$$S_f(f_c) = 2^n - 2N_d = 2N_s - 2^n, \tag{4.1}$$

where n is the number of logic variables, N_s stands for the number of input patterns for which functions f and f_c have the same value, and N_d is the number of patterns where their values are different.

The calculation of cross-correlation can be performed using methods described in the previous section. Namely, N_s and N_d can be expressed in terms of the output probabilities of the functions f and f_c. Let $P(f \cdot f_c)$ be the probability that the values of both functions are 1, and $P(\bar{f} \cdot \bar{f}_c)$ be the probability that the outputs are 0. N_s can be computed as follows:

$$N_s = 2^n(P(f \cdot f_c) + P(\bar{f} \cdot \bar{f}_c)), \tag{4.2}$$

and N_d, avoiding the $P(f \cdot f_c)$ as follows:

$$N_d = 2^n(P(f) + P(f_c) - 2P(\bar{f} \cdot \bar{f}_c)). \tag{4.3}$$

The output probabilities of the functions $P(f)$ and $P(f_c)$ can be computed using the ROBDD based method described in the previous section. Furthermore, probability $P(f \cdot f_c)$ can be computed by manipulating the edges of the ROBDDs for f and f_c, enabling us to use the already established algorithm for probability computation to calculate spectral coefficients.

From 4.1 and 4.3 it follows:

$$S_f = 2^n(1 - 2P(f) - 2P(f_c) + 4P(\bar{f} \cdot \bar{f}_c)). \tag{4.4}$$

This method has been employed by Thornton, Miller and Drechsler, in [164] for calculation of Rademacher-Walsh, Haar, and Reed-Muller spectral coefficients.

4.5 POWER CONSUMPTION ANALYSIS USING DECISION DIAGRAMS

The power consumption in digital CMOS circuits is closely associated with their *switching activity*. The main source of power dissipation with this technology comes from charging and discharging the node capacitances. Let V_{DD}^2

be the supply voltage, and f_{clk} clock frequency of a circuit. The following formula was proposed in [182] as a good approximation of the power consumption of the circuit:

$$P = 1/2 V_{DD}^2 f_{clk} \sum_i C_i E_i, \tag{4.5}$$

where C_i is the sum of all the input capacitances of the transistors that are driven by the signal i and E_i is the transition probability at signal i. This probability is the switching activity, that is, the probability that either $0 \longrightarrow 1$ or $1 \longrightarrow 0$ transition will occur, $E_i = p(0 \longrightarrow 1) + p(1 \longrightarrow 0)$. It can be calculated using Binary Decision Diagrams.

Switching activity at the output of a gate depends on the logic function of the gate. Transition probabilities of a gate can be calculated from output probabilities.

Parker and McCluskey [129] have introduced a method to generate polynomial representations of the output probabilities of basic logic gates:

$$p(x_1 \wedge x_2) = p(x_1)p(x_1), \tag{4.6}$$

$$p(x_1 \vee x_2) = p(x_1) + p(x_1) - p(x_1)p(x_1), \tag{4.7}$$

$$p(\bar{x}) = 1 - p(x). \tag{4.8}$$

Transition probabilities on a signal E_i can be computed in two ways. The first method, proposed by Cirit in [29] is based on a general delay model and makes use of global Binary Decision Diagrams.

As already stated, $E_i = p(0 \longrightarrow 1) + p(1 \longrightarrow 0)$. The probability that the transition $0 \longrightarrow 1$ occures at node i can be approximated as $p(0 \longrightarrow 1) = p(i = 0)p(i = 1)$, where $p(i = 1)$, $p(i = 0)$ are the probabilities that the signal at node i, will take the value 0 and 1 respectively. Likewise $p(1 \longrightarrow 0) = p(i = 1)p(i = 0)$. These probabilities can be calculated using BDDs as described in the previous section.

Binary Decision Diagrams were first proposed for this application by Chakravarti et al in [27]. This method did not take into account the gate delay. It also ignored the power dissipation due to hazards and glitches. This problem was addressed by application of a general delay model by Ghosh et al. in [71]. The method uses symbolic simulation to generate a set of switching functions that represent the condition for switching activity at distinct time points for each gate. However, the symbolic simulations are also the main disadvantage of this method, because symbolic formula become impractically large even for medium sized circuits.

A local OBDD construction has been used to ameliorate this problem by Marculescu in [108], and further extended to handle highly correlated input streams in [107].

Buch et al. [21] computed the power dissipation of each node without considering the temporal signal correlation. This method has been used in conjunction with BDDs by Drechsler, Lindgren et al. [51], [52] and Popel [132].

The second approach to calculating transition probabilities is based on polynomial propagation by means of Arithmetic Decision Diagrams, or alternatively, Binary Momentum Diagrams. Ferreira et al. proposed a ZBDD-based method for power consumption [61]. This method has been extended to include BMDs by Ferreira and Trullemanns [62].

4.6 INFORMATION MEASURES ON BDD AND SWITCHING ACTIVITY

The switching probabilities of nodes in a circuit are closely associated with the concept of entropy. It can be demonstrated that the upper bound of a switching probability of a node is half of its entropy. We discuss the calculation of the information measures using decision diagrams in greater detail in later sections. For now, it will suffice to say that the entropy of a function $f(x)$ can be calculated using the following formula:

$$H(f) = -\sum_{a=0}^{1} p(f = a) \log p(f = a) \tag{4.9}$$

The entropy $H(f)$ can be computed in a recursive manner using the conditional entropy $H(f|x_i)$ for each variable x_i where

$$H(f|x_i) = p(x_i = 0)H(f|x_i = 0) + p(x_i = 1)H(f|x_i = 1) \tag{4.10}$$

This calculation can be efficiently performed using binary decision diagrams.

We propose an additional ROBDD-based method for calculation of the entropy in Section 6.

4.7 FORMAL VERIFICATION

Another important step in the process of logic design is the verification of the circuit implementation. A number of methods have been developed to determine if the actual gate-level design matches the behavior of the desired logic function. The simplest method is to determine if the circuit produces the desired output for each combination of input values. However, this method quickly becomes impractical as the number of possible inputs grows.

Formal circuit verification is based on the notion that a mathematical proof of correctness of the design implicitly covers all of the possible input patterns. An explicit enumeration of the input patterns is thus avoided. This task can be accomplished by means of equivalence checking. Two circuits are equivalent if and only if the canonical representations of their output functions are the same.

A Reduced Order Binary Decision Diagram is a canonical representation of a switching function. Thus, two logic circuits are equivalent if their ROBDDs are equivalent. In practice this can be achieved by first constructing the ROBDDs for the functions, and then checking if they are isomorphic

This checking can be computationally complex. It can be avoided in the following manner. Let f and g be two switching functions. These two functions can be considered equivalent if and only if the ROBDD for $f\ g = \bar{f} \oplus g = f \oplus \bar{g}$ consists only of a terminal node 1.

As mentioned earlier, ROBDDs are used to represent switching functions in order to avoid complexity explosion during the evaluation of circuits. However, the size of ROBDDs can also grow exponentially for many classes of switching functions.

Other classes of canonical decision diagrams can be used to address this problem. This includes Reduced Order Functional Decision Diagrams and various classes of Word Level Decision Diagrams. For example, Edge Valued Decision Diagrams have been introduced in [100] and [173] especially for the task of verifying arithmetic circuits. Closely related, the Binary Momentum Diagrams (BMDs), proposed in [20], are used for verification of multipliers. This work was further extended by the introduction of Kronecker Multiplicative Binary Momentum Diagrams (K*BMDs), in [47], [53]. Of special interest is the class of Boolean Expression Diagrams, since they permit the representation of any combinational circuit in linear space. These diagrams can be constructed directly from the circuit graph.

The XML framework works with all these diagrams, or can be extended to accommodate them. Thus, it can be efficiently exploited for the formal verification of circuits.

4.8 VERIFICATION OF SEQUENTIAL CIRCUITS

The verification methods that we have discussed so far have been focused on combinational circuits, which can be efficiently represented using graph-like structures. The output of a sequential circuits depends not only on the current input of the circuit, but also on the values of its memory elements, i.e., outputs, from the previous time intervals, and so they cannot be well represented with the same structures as combinational circuits. The operation of a sequential circuits is described by another graph-like structure, the *Finite State Automaton*. The equivalence of Finite State Automatons can be evaluated by explicit representation of state sets. However, this is not applicable to larger systems.

Approaches that try to address this problem are based on an implicit set representation. Each sequential circuit contains a combinational component and a register of length n. The contents of the register represents an input sequence of a length k. Therefore, the operation of the register component of the circuit can be described by a n-input, k-output function. Thus, ROBDD based methods can be applied.

The verification of sequential circuits using ROBDDs was first explored by Bryant [18]. Touati et al. [166] and Burch et al. [23] developed symbolic BDD based techniques for storing and representing state sets of sequential circuits. Related work, with a somewhat different approach has been done by Coudert et al. [35].

BDDs have been used to determine synchronizing sequences for sequential circuits by Pixley et al. [131].

Binary decision diagrams, as a tool for the verification of sequential circuits, suffer from potential size explosion for large state sets. This problem has been addressed by Hu [89] by the application of implicitly conjoined BDDs. Furthermore, Hu and Dill developed a technique of to eliminate of state variables in [88]. These methods have been implemented in EVER and Murφ software packages by Hu [87] and Dill [42] respectively. A similar package called SMV was developed by McMillan [114].

Hybrid Decision Diagrams have been exploited by Clarke et al. [31] to support word-level model checking.

Ravi and Somenzi in [134] suggested to reduce the size of intermediate BDDs by replacing them with smaller BDDs of dense subsets.

VIS is a software developed by Brayton [14] which integrates model checking with other verification techniques, and VERITY developed by Kuehlmann et al. [97].

The XML-based framework can be exploited in these applications, whenever decision diagrams are the underlying data structures.

4.9 APPLICATION OF DECISION DIAGRAMS IN DIGITAL IMAGE PROCESSING

Digital image processing is another area in which decision diagrams have been applied. Originally Binary Decision Diagrams were proposed as a method for encoding of bitmap images by Starkey and Bryant in [153]. In this approach, a ROBDD is constructed using pixel coordinates as function variables. For an image block of the size $2^n \times 2^n$ an alternated sequence of pixel coordinates defines the variable ordering as $x_{n-1}, y_{n-1}, ..., x_0, y_0$. Each individual path from the root node to the terminal nodes represents a group of pixels.

A different approach was proposed by Lursnisap et al. in [105]. A binary image block can be seen as a Karnaugh map. Classical Boolean minimization is applied to obtain a SOP representation of the function. The ROBDD can then be constructed directly.

These methods were originally developed for bitmap images with binary valued pixels. However, the same methodology can easily be extended to grayscale images, either through the use of MTBDDs as suggested by Starkey and Bryant or through the use of SBDDs by separating the image in a set of multiple bitplanes. The second approach has been employed by Lurnisap et al in [105].

Another approach to the BDD image encoding was introduced by Wu and Chung [179]. In this method an image segment is recursively subdivided into two halves in the x and y direction alternatively. In this way, a SOP representation of image is obtained and a ROBDD is constructed.

Furthermore, in [103] and [104] Leelapatra et al. demonstrate how a standard geometric image transforms such as image translation and orthogonal rotation can be implemented directly on ROBDD image representations.

Decision diagram based image encoding has been employed specially for image processing tasks which involve a sliding window. In these algorithms the resulting value of each individual pixel is determined by the value of the pixels in its neighborhood. The values of neighboring pixels define a discrete function which determines the pixel value. This function can be efficiently implemented using decision diagrams. An example of this approach can be found in a research report by Robert and Malandain [136].

Therefore, due to the use of decision diagrams for the representation of images and for performing various operations over them, the proposed framework can be employed in particular tasks of signal processing.

Another important application of decision diagrams in image processing is in various lossless compression schemes. One such example can be seen in the paper by Mateu-Villarroya and Prades-Nebot [110].

4.10 POSITION OF THE XML-BASED FRAMEWORK IN THE APPLICATION CONTEXT

In the introductory part of this Thesis we have stated that the intended purposed of the proposed XML-based framework is the efficient representation of various types of decision diagrams in a uniform way. This stated purpose comes into the proper light only if we take into account the possible applications of decision diagrams.

Namely, as the extent of the applications of decision diagrams has grown wider, a great variety of software packages aimed at dealing with these structures were developed. These software solutions differ significantly in their scope and architecture. On the one hand we have software packages aimed at specific industry related tasks, such as the optimization or testing of the specific kinds of logic circuits, focused on a few classes of decision diagrams. Usually, these software systems treat decision diagrams as a valuable tool for achieving their primary task and employing them as a part of a longer process.

Most of such packages use decision diagrams only as an internal data structure, generating them at exploitation time. Import, export, and storage facilities of such software packages are generally dedicated to the final product of the process, i.e., an optimized netlist representing a circuit.

In the other end of the spectrum are the packages focused on the decision diagrams themselves, for example, various heuristic algorithms aimed at optimization of some property of a decision diagrams, e.g., minimization of the size or the path length, finding the optimal variable ordering or the optimal Kronecker diagram, etc. These software packages are sometimes capable of dealing with several classes of decision diagrams, as the same optimization strategies can be employed across different decision diagram classes.

The proposed framework is intended to take the position of an intermediary between various software packages aimed at working with decision diagrams. Fig. 4.1 illustrates the position of the XML framework in a wider application focused scope.

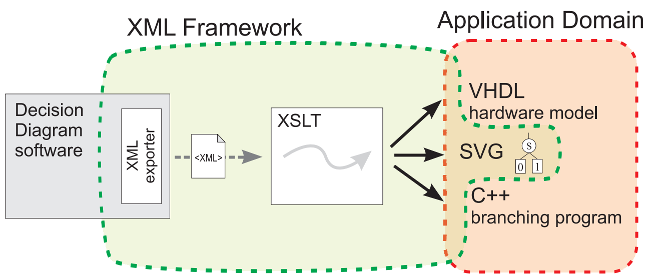


Fig. 4.1 Position of the XML-based framework in a larger context.

The proposed framework offers a neutral, abstract format of representation of decision diagrams, and facilitates the seamless data exchange between different software packages.

The existing software packages can be easily extended with modules capable of storing decision diagrams in the form of XML documents. Software modules for the import and export of data in an XML format are known as *XML parsers*. XML is an established standard for data interchange and such XML parsers exist for all the relevant computer platforms and software development environments. The application of XML Schema and the validity of XML documents which it ensures, guarantees that any such parser can be used to handle XML documents described by our framework.

Importing data from XML decision diagram documents into existing packages can be facilitated in two ways. One solution is to extend an existing package with the same kind of XML parsing module. However, another possibility exists purely within the XML family.

An XML-based framework for decision diagrams is capable of converting its own XML documents into any other textual, i.e., ASCII-based, file format. This conversion process relies on the XSLT data transform language introduced in Section 3.11.

In what follows we will present several examples of the applicatin of the proposed framework in different fields.

The first set of examples presented in Chapter 5 is focused on the application of decision diagrams in logic design. We present a method of FPGA implementation of switching functions represented using several types of functional decision diagrams, primarily reduced ordered binary decision diagrams, Kronecker and Pseudo-Kronecker decision diagrams. In these examples, the final hardware design is expressed in terms of the VHDL hardware description syntax.

Furthermore, we compare the efficiency of the implementation of multi-output switching functions using Multi-terminal binary decision diagrams versus the more common Shared binary decision diagrams.

To conclude this series of examples, we present an example of implementation of switching functions on 6 input LUT based FPGA devices using Quaternary decision diagrams. The EDIF language for netlist specification is used in this example.

In order to properly estimate the possibility of exploiting the proposed XML framework in these areas, we first provide a brief review of the related work.

5

Applications of the XML Framework in Logic Design

In this chapter we present applications of the XML framework in logic design. To make this presentation self-contained and understandable, we first briefly introduce the necessary definitions and notations.

Since the initial interest in binary decision diagrams in the logic design community was sparked by Bryant [17], a variety of classes of decision diagrams have been introduced to address specific circuit design and verification problems.

For example, Ternary Decision Diagrams (TDDs) have been proposed in [93] to represent incompletely specified switching functions. In the work of Sasao and others a total of seven distinct classes of TDDs have been introduced, depending on the value associated with the third outgoing edge of each non-terminal node. In EXOR-TDDs and AND-TDDs introduced in [142] and [143] the third outgoing edge points to $f_0 \otimes f_1$ and $f_0 \vee f_1$ values respectively. SOP-TDDs represent products in SOP functional representation. Likewise, ESOP-TDDs deal with ESOP representation. Kleene-TDDs [92] are useful for the verification of functions in the presence of unknown inputs.

Zero suppressed BDDs can be used, in the same manner, to represent sets of cubes [120].

Quaternary decision diagrams have been proposed by Sasao in [144] as an optimal method for the implementing switching functions on 6-input LUT FPGA devices.

MTBDDs mentioned in Section 2.4 have been originally introduced by Clarke et al. [33] to represent Walsh coefficients employed to reduce the number of possible component matches that need to be checked during the technology mapping stage of circuit design.

As mentioned in Section 2.4, MTBDDs represent the most basic type of word-level decision diagrams. The wide class of world-level decision diagrams received special attention as a tool for the verification of arithmetic circuits. Various edge valued diagrams have been proposed as the extension of the basic concept.

Specifically, EVBDDs have been employed by Lai and Sastry in [100] as a tool for multi-level hierarchical circuit verification. Factored EVBDDs have been introduced as a further extension of this concept [173].

Multiplicative binary momentum diagrams (*BMD) are used by Bryiant in [20] for verification of carry save adders. They have been used as a tool for verifiying of other arithmetic circuits, such as multipliers. However, this class of decision diagrams is not well suited for bit-level verification tasks. In order to devise a unified tool for both bit and word-level verification, Clarke and Fujita have proposed a class of diagrams known as Hybrid (HDDs) or Kronecker Binary Momentum Diagrams (KBMDs). They have used these diagrams in [31] for testing the STR divider as implemented in Pentium processors. This idea was further explored by Drechsler and Becker in [47] with an introduction of Kronecker *BMDs.

We begin our discussion from the most basic application of decision diagrams in logic design, the implementation of switching functions using ROBDDs, and gradually move to other classes of decision diagrams. The results related to the implementation of switching functions using ROBDDs presented in later sections of this chapter were originally published in [161].

In the following sections we examine basic concepts of hardware design and function implementation, using decision diagrams that are needed for understanding the presentation related to the applications of the XML based framework.

In order to better estimate the usability of the proposed framework, we begin our discussion by giving a quick overview of other software solutions intended for similar purpos.

5.1 RELATED WORK

As the complexity of hardware grows, it quickly becomes impossible to manually handle even individual designs. To address this problem, automated systems for logic design and circuit verification and testing have been developed. The idea behind these systems is to let the designer work on a higher level of abstraction and let the system perform the mapping at the most basic technological level. This can be achieved by applying some high level formal language. This approach is known as High Level Synthesis. A large body of theoretical work exists on this matter, for example, [163], as well as a significant number of commercially available products.

One way of addressing this problem is by using a special dedicated syntax. VHDL and Verilog are most well known examples of special purpose hardware description languages. The advantage of this approach is that the language and syntax are specially tuned for particularities of hardware modeling. However, many hardware devices represent an implementation of algorithms previously developed in software. It is sometimes beneficial to let the developers use the syntax that they are already familiar with.

A number of systems based on syntactic rules of languages such as C/C++ or Matlab exist. These system ether produce VHDL/Verilog models as an output or perform direct mapping onto technology. In what follows we present a list of some of these systems.

We begin our overview with one of the best known of such languages, System C. System C can best be viewed as a system behavior modeling language.

The main difference between hardware design and software implementation of an algorithm is that the software instructions are most often executed in a sequential manner, while processes that happen in hardware exhibit a certain degree of parallelism. In order to use a syntax of a standard programing language, one needs to provide the means for handling concurrent processes.

System C is a library of C routines and macros designed with this exact purpose in mind. It is dedicated especially to transaction level and behavioral modeling. Processes described using System C framework can communicate in a simulated real-time environment. Furthermore, System C is a simulation kernel. A System C code can be compiled together with the simulation kernel library into an executable code that behaves like the described system.

Since it is based on C/C++, System C provides some interesting features not present in standard hardware description languages, such as on object oriented approach to modeling and template classes. However, this potentially greater expressive power is contrasted by noticeable syntactic overhead. Code written in System C tends to be longer than the equivalent VHDL code. Furthermore, at the current level of development, the performance of the simulation kernel is still behind the commercial simulation tools.

System C is currently being developed as an open source effort by a group known as the Open System C Initiative, [124]. System C started originally as a project of Synopsys Inc., an EDA company. The project gained a wider acceptance when the cooperation with several large electronic companies, such as ARM and CoWare, was established.

Companies that offer System C based solutions include:

1. Synopsys Inc.

2. Cadence

3. CoWare

4. Mentor Graphics

For example, Catapult C is a System C based solution by Mentor Graphics intended for both ASIC and FPGA design of wireless, video and image processors, [25].

Estrel-C is a similar product by Cadence and CoWare offers SystemC Modeling Library as a part of its product palette, [59].

Cynlib developed by Cynapps, now Forte EDS was at one time the main competitor to System C, [37]. This is a C++ class library for high-level hardware design. It is an extension of C++ that implements many of the Verilog semantic features.

Handel C by Celoxica is another C based product, [26]. It steams from work done at the Oxford University Computing Laboratory in the early nineties [125]. It is a non-standard extension of C language that permits hardware instantiation and parallelism. As all C/C++ based hardware description languages, it is behavior oriented. It includes most of the common C language features. However, floating point arithmetic is not a standard part of Handel C, but it can be implemented using external libraries. Parallel behavior is described using some of the CSP (communicating sequential processes) keywords, while general file structure is borrowed from Occam.

FpgaC, is another subset of C language. It has its origin in Transmogrifier C developed at the University of Toronto by D. Karchmer under the supervision of J. Rose, P. Chow, D. Lewis, and D. Galloway, [68]. It is one of the first efforts of this kind which predates even Handel-C. FpgaC has a slightly different intended scope. It was designed to be an efficient High Level Language for reconfigurable computing, rather than a hardware description language for efficient custom circuits.

Transmogrifier C has been available as a BSD licensed Open Source platform since 1996. Originally it had more nonstandard HDL specific extensions. Gradually, they were phased out and replaced by more standard C features, under the influence of StreamC.

StreamsC was developed by Los Alamos National Lab., by M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, [72]. It is now available as Impulse C, and contains a compiler and a set of libraries intended for the development of FPGA-based applications.

Impulse C includes simulation tools as well as C-to-RTL scheduling and optimization technology. It supports the CSP programming model, while remaining compatible with ANSI C. It is data-flow oriented.

Intended applications include image processing, DSP algorithms and other high-performance computing applications.

Impulse C permits development of mixed hardware/software algorithms. The core concepts of Impulse C are processes and signals. Processes are independently synchronized, concurrently executed segments of code. Hardware and software components communicate through buffered data streams that are implemented directly in the hardware. Impulse C thus makes possible the development of parallel applications on a high level of abstraction, without the need for cycle-by-cycle synchronization.

Other high-level programming languages have been used as a basis for similar hardware modeling solutions, as well. The MATCH compiler, represents one such effort. It permits conversion of Matlab code into a VHDL syntax.

MATCH is a Matlab compilation environment for Distributed heterogeneous Computing Systems developed by D. Banjee, A. Chanderay, S. Hauck and N. Sheray, [11], [83] and founded by DARPA from 1998 to 2001. It is now available commercially from the Accel Chip company.

Pure FPGA-based systems are usually unsuitable for a complete algorithm implementation. Often, large sections of code are executed only rarely. Mapping these into FPGA would produce unnecessary overhead. The solution can be found in a combination of embedded systems general purpose CPUs and dedicated FPGA hardware. MATCH is designed to facilitate efficient mapping of given algorithms to such a heterogeneous architecture.

Most of these solutions focus primarily on the behavioral approach to hardware modeling. In our work, we take a somewhat different approach focusing on structural descriptions of decision diagrams in the XML data description language. By taking this approach, we are able to avoid most of the problems regarding the conversion of a behavioral model to a structural model of an RTL, since there is a well defined mapping of decision diagram structural entities onto hardware elements. We are able to take this approach because most of the software tools dealing with decision diagrams, which are to serve as a front end to the proposed system, focus on the problems of structural optimization of decision diagrams.

5.2 NETLISTS

A Netlist represents a description of connectivity of an electronic device. In the most general terms, a netlist consists of a set of basic functional elements and a set of their mutual connections. Thus, a netlist is a graph-like structure with a strong resemblance to decision diagrams.

A netlist represents a structural hardware description. No additional synthesis and optimization is required by the software system that performs the mapping of a netlist to a specific technology.

These basic functional elements can be any kind of physical devices from resistors and capacitors to integrated circuits. A one-to-one correspondence between functional elements of a netlist and nodes of the decision diagram can be established. Netlists are usually a technology dependent description of hardware design. The type of functional blocks is, therefore, determined by the underlying technology. In the case of FPGA technology, these functional elements are logic blocks of the FPGA device, usually LUTs.

It is possible to describe netlists in a variety of methods. However, most of these methods follow some common principles. A netlist usually contains one explicit definition for each type of functional element used. An instance of an element defined in this way is used whenever such an element occurs in the design.

Each functional element has a set of points of connection, know as 'pins' or 'ports'. These ports can have input or output direction, or be bidirectional. In netlist terminology, physical and logical connections between these elements are called 'nets'. The connections can be expressed in the form of ordered pairs, consisting of the identifier of an output port of one device and the input port of the other.

A set of properties further describing its functionality is associated with each instance of an element. A logic function implemented by a particular LUT is an example of one such property into the case of FPGA technology.

For more complex designs, netlists can be organized in hierarchical structures, where functional elements of one level in the hierarchy are the whole netlists themselves.

A number of description languages are used in practice to describe netlists. SPICE and EDIF are one of the best known examples. In our work we have made the use of the EDIF description language. Therefore, we present an overview of this language in Section 5.6.

5.3 EDIF NETLIST DESCRIPTION LANGUAGE

Electronic Design Interchange Format, (EDIF), is a description language and a file format for the representation of electronic netlists and schematics. It was designed as a neutral format to facilitate data interchange between software platforms produced by various industrial companies.

There are three versions of the EDIF language still in use, each with its advantages and shortcomings, EDIF 2 0 0, EDIF 3 0 0 and EDIF 4 0 0. Standards of the EDIF language and format have been established and maintained by the EDIF Steering Committee and EIA, Electronic Industry Association. EDIF specification is considered complete and no additional development of the standard is expected in the future.

5.4 EDIF HISTORICAL DEVELOPMENT

The EDIF language originated as an attempt to solve the problems of data exchange between various Electronic Design Automation (EDA) companies in the beginning of the eighties. As the complexity of circuit designs grew and the market for EDA products evolved a need for efficient means of data exchange emerged. Most companies involved with this segment of the market owned their proprietary software solutions and large electronic design databases. The problem of data translation from one proprietary format to another increased exponentially with each additional data format.

Under pressure from customers, a number of these companies reluctantly agreed to start work on a neutral data exchange format. Preliminary work on this standard was formalized in November 1983 by an EDIF Steering Committee. This committee included representatives of Daisy Systems, Mentor

Graphics, Motorola, National Semiconductor, Tektronix, Texas Instruments, and the University of California, Berkeley.

EDIF 1 0 0, introduced in 1985, was the first version of the data format presented to the public. This rudimentary version was later suppressed by a new EDIF 2 0 0 specification published in 1988.

EDIF 2 0 0 is a result of a series of compromises made to permit compatibility with earlier legacy software systems. Due to this fact, EDIF 2 0 0 specification has a number of fundamental ambiguities, which make the process of writing EDIF interpreters especially difficult. Some of these problems were addressed by the EDIF 3 0 0 specification which was introduced in 1993. However, EDIF 3 0 0 is not compatible with previous versions of the standard. Therefore, the development of a completely new set of software tools was required. This fact contributed greatly to the continued use of the previous EDIF 2 0 0 version.

EDIF 3 0 0 was further extended to EDIF 4 0 0 in 1996.

In our work we have made use of EDIF 2 0 0, mainly due to practical reasons. It is the only ASCII-based netlist file format which is used by Xilinx ISE system, the software package that we have used in our experiments. The ASCII output is one of the requirements of the XSLT-based conversion system.

However, only minor changes in the proposed system would be necessary to make it possible to produce output compatible with other versions of EDIF.

More information about the EDIF language in general and historical trends in its development can be found, for example, in [28], [54], and [94].

5.5 EDIF NETLISTS IN COMPARISON TO VHDL MODELS

Since in our work we make use of both EDIF and VHDL, two different hardware description languages, it is good to compare the properties of these two languages. Such a comparison will justify the choice of one over the other for certain applications or groups of experiments. The differences between these two description languages are profound and stem from the original goals behind their introduction.

From the beginning VHDL was envisioned as a language much larger in scope. It was intended as a language capable of describing hardware designs independently from any particular platform, with a focus on hardware functionality [130]. It is a much more versatile platform which permits both structural and behavioral hardware models.

EDIF netlists are usually targeted at specific families of devices, relying on extensive device libraries provided by hardware manufacturers. Netlists are strictly structural hardware descriptions.

However, the process of implementing a VHDL hardware design involves several complex steps, including the interpreting of the VHDL code, logic synthesis, optimization of design, usually by some heuristic method, and finally mapping onto a particular technology.

The process of implementing EDIF netlist is somewhat simpler, mainly because the EDIF netlists can be directly mapped to a lattice of specified logic devices. The synthesis and optimization steps are not required from the interpreting software system, since it is assumed that this part of the process was already done by the software system which generated a netlist.

In our own work, the process of generation and optimization of a decision diagram performs this role. Additional heuristic optimization by a software system can be even counter-productive and may introduce the ambiguity in the final experimental results.

On the other hand, VHDL remains in much wider use in practice, and the output in this format is crucial for the usability of the proposed system.

A description of an interface between VHDL and EDIF languages is presented in [148].

5.6 BASICS OF EDIF SYNTAX

The syntax of EDIF language follows the syntactical conventions of Lisp. Each instruction both the reserved word and parameters (operands) are enclosed by a pair of parenthesis. Instructions can contain other, recursively nested instructions. The set of basic tokens of the EDIF language contains keywords defining instructions, i.e. `library`, `cell`, `instance`, strings, integer numbers, symbolic constants and identifiers.

The first part of an EDIF document represents a series of definitions of components that the device consists of. The instances of these components are used to describe the structure of the device. These definitions are declared using a `cell` instruction. The contents of the cell can be described in several ways. In EDIF language syntax this is achieved using one of multiple defined `viewType`s. These view types include, `BEHAVIOR`, description of the behavior of a cell, `DOCUMENT`, documentation associated with the cell, and many others. In our work we particularly focus on the `NETLIST` view type. This view type requires a declaration of input and output ports of the device using `interface`, `port` and `direction` instructions. Additional component specific properties can be associated with the component definition using the `property` instruction.

Example 5.1 *The following EDIF code represents a declaration of a six-input LUT component in a Xilinx Virtex5 FPGA family of devices.*

```
(cell LUT6
      (cellType GENERIC)
        (view view_1
          (viewType NETLIST)
          (interface
            (port I0
              (direction INPUT)
            )
            (port I1
              (direction INPUT)
            )
            (port I2
              (direction INPUT)
            )
            (port I3
              (direction INPUT)
            )
            (port I4
              (direction INPUT)
            )
            (port I5
              (direction INPUT)
            )
            (port O
              (direction OUTPUT)
            )
            (property TYPE (string "LUT6") (owner "Xilinx"))
            (property XSTLIB (boolean (true)) (owner "Xilinx"))
            (property INIT
            (string "FF00F0F0CCCCAAAA") (owner "Xilinx"))
          )
     )
)
```

The second segment of a typical EDIF description represents a series of instances of declared components. Components are instantiated using `instance`, `viewRef` and `cellRef` instructions. Each instance must have a unique identifier associated with it.

Example 5.2 *The declaration of an instance `XLXI_612` of the LUT6 component from Example 5.1 has the following form:*

```
(instance XLXI_612
(viewRef view_1 (cellRef LUT6 (libraryRef UNISIMS)))
(property XSTLIB (boolean (true)) (owner "Xilinx"))
(property INIT (string "FF00F0F0CCCCAAAA") (owner "Xilinx"))
)
```

In this example, the additional property `INIT` associated with this instance specifies the switching function implemented by this LUT. In this case LUT6 behaves as a 4X1 multiplexer controlled by a two-bit variable.

The structure of the device is completed by specifying a set of connections between the instances of components. These connections are defined using a `net` instruction and a list of references to connected ports.

Example 5.3 *A connection between two instances of LUT6 in a device is defined using the following piece of code:*

```
(net XLXN_751_0
(joined
(portRef I2 (instanceRef XLXI_751))
(portRef O (instanceRef XLXI_3))
)
)
```

In this example the output port of `XLXI_3` is connected to the second input port of `XLXI_751` LUT.

This is just a short overview of some of the basic features of the EDIF language. The EDIF standard is much more extensive and permits programming techniques not covered in this section. However, the EDIF documents created by the proposed system, which are used in the experiments in following sections, have the same basic form as the one demonstrated here. More information on EDIF syntax can be found in [152]. We discuss the relationship between these hardware descriptions and various types of decision diagrams in the sections to follow.

5.7 VHDL

With the increasing complexity of logic circuits, a need for higher levels of abstraction during the electronic design process soon becomes evident. While it is feasible to manually design at gate level the circuit consisting of tens or hundreds of components, the design of a circuit with millions or even billions of logic gates far exceeds human capabilities.

High Level hardware description languages represent a solution for this problem. The VHDL, along with Verilog and SystemC, is one of most commonly used languages designed for this task.

The current version of VHDL 1076-1987 is an IEEE standard.

In the following sections we examine some of the basic aspects of the VHDL and discuss how VHDL can be used to implement the discrete functions represented in decision diagram form.

5.8 VHDL HISTORICAL NOTES

The origins of the VHDL can be traced to the Very High Speed Integrated Circuit program by the Department of Defense of the United States started in the late 1970s. Indeed the acronym VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. The goal of the VHSIC program was to produce the next generation of integrated circuits for application in military projects. One of the basic requirements of this project was to design circuits of very high complexity at the same time shortening the design time, practically pushing the technology limits. It became evident during the early stages that these requirements exceeded the capabilities of the design tools available at that time.

As a solution for this problem a new hardware description language was proposed in 1981. From its conception VHDL had to fulfill a double role. On the one hand it had to be flexible and robust enough to be able to describe the circuits of increasing complexity, not only circuits on the standards of the moment, but also anticipated future circuits. On the other, it needs to do so in a standardized manner to permit easy exchange of data between various designer teams.

Basic VHDL syntax and concepts were derived from Ada, a classic example of an object oriented programming language, and also a product of a United States Department of Defense project.

The use of VHDL spread to the larger public. In 1986 it was proposed as an IEEE 1076 standard. A year later this standard was adopted.

However, this first official edition of the language standard did not address all of the important issues. Above all, it did not specify the methods for handling multi-valued logic. This problem was corrected when the IEEE 1164 standard was introduced.

The syntax of language has been made more consistent with the second issue of IEEE 1076, in 1993. In 2000 and 2002 minor additions have been made to the standard, by introducing of the protected types, a concept taken from C++ language.

Several standards have been developed on the basis of VHDL IEEE 1076 to extend its functionality to other fields. For example, the IEEE standard 1076.1 provided analog and mixed-signal circuit design extensions. IEEE 1076.2 added better handling of real and complex data types. IEEE standard 1076.3 introduced signed and unsigned types to facilitate arithmetical operations on vectors.

5.9 VHDL BASIC CONCEPTS

As stated earlier, VHDL borrows many of the concepts from the Ada language. However, significant differences were made in order to adapt the language to hardware description requirements.

VHDL is a structured, strongly typed description language with many concepts adopted from object oriented programming languages. VHDL was designed to provide platform independent hardware descriptions. It supports several methods of hardware description, in effect supporting both structural and behavioral hardware models.

In order to respond to the complexity of modern logic circuits, VHDL contains levels of representation that can be used to represent all the levels of description from the bidirectional switch level to the system level.

In this section we will go through some of the basic concepts of VHDL, necessary for understanding of sections that follow.

`Entity` is the basic building block of VHDL hardware description. All designs are expressed in terms of entities. Entities can be organized into hierarchical structures, a feature borrowed from object oriented programming. The uppermost level of the design is the top-level entity.

Example 5.4 *The following VHDL code is a declaration of an entity for a device which implements the Shannon decomposition of a function for variable x:*

```
ENTITY my_shannon IS
  PORT (f0, f1, x : IN std_logic; z : OUT std_logic);
END ENTITY my_shannon;
```

This device has three input and one output port.

Each entity has a port list defining input and output ports of the device associated with it.

`Architecture` in VHDL terms, is a formal description of the structure or behavior of an entity. Each entity must have at least one architecture associated with it in order to be valid. Multiple, usually technology-specific architectures can be associated with one entity. For example, one architecture might be behavioral in nature, while the other might be a structural description of the entity.

Example 5.5 *An architecture associated with a VHDL entity is declared using the following set of instructions:*

```
ARCHITECTURE my_shannon_arch OF my_shannon IS
  SIGNAL ... ;
BEGIN

  ...

END my_shannon_arch;
```

Entities may contain instances of other entities in their own architectures. The hierarchy of entities is achieved in this way.

A `Configuration` statement is used to bind a component instance to an entity and its architecture.

One method of specifying a hardware design in VHDL is to specify a set of component instances and their mutual interconnections in the architecture of an entity. This is known as a structural method of hardware design.

Example 5.6 *Consider, for example, an RS flip-flop circuit. This circuit can be implemented in VHDL using the following code.*

```
ARCHITECTURE netlist OF rsff IS
 COMPONENT nand2
  PORT (a, b : IN BIT;
        c, : OUT BIT);
 END COMPONENT
BEGIN
 U1: nand2
  PORT MAP (set, qd, q);
 U2: nand2
  PORT MAP (reset, q, qb);
END netlist
```

Architecture presented in this example is structural in nature. Entities `U1` *and* `U2` *are instances of the NAND component specified in the first section of the architecture declaration. Architecture is a net list consisting of these two component instances and their interconnections, given using port associations.*

Note that port associations of components in the previous design were stated in an implicit way. These associations can be specified in much more strict explicit form using *named associations*. A named association is an ordered pair of the input and the output port of two connected components separated by a '`=>`' sign.

Example 5.7 *Connections between devices* `U1` *and* `U2` *from the previous example can be stated using named associations.*

```
U1: nand2 port map (a => set, b => qb, c => q);
```

Another way of specifying a hardware design is by using a behavioral architecture. Behavioral architectures use concurrent signal assignment statements. The way these statements are treated highlights an important difference between hardware description languages, such as VHDL, and structured programming languages. In a typical structured programming language, these statements would be executed in a sequence. Hardware description languages are designed to model physical process in electronic devices. Therefore, these statements will be executed in parallel, i.e. concurrently. A specific ordering of these statements inside an architecture is irrelevant in VHDL. The order of the execution is determined by events that occur on the signals involved.

Example 5.8 *We have used behavior hardware description to implement the architecture of the binary Shannon decomposition node as follows.*

```
ARCHITECTURE my_shannon_arch OF my_shannon IS
  SIGNAL a, b : std_logic;
BEGIN
    a <= f0 AND NOT(x);
    b <= f1 AND x;
    z <= a XOR b;
END my_shannon_arch;
```

VHDL explicitly supports the sequential execution of statements. This is achieved using the PROCESS instruction.

Example 5.9 *RS flip-flop from Example 5.6 can be implemented in a sequential manner using the following VHDL code:*

```
ARCHITECTURE sequential OF rsff IS
BEGIN
 PROCESS (set, reset)
BEGIN
 IF set = '1' AND reset = '0' THEN
  q <= '0' AFTER 2 ns;
  qb <= '1' AFTER 4 ns;
 ELSEIF set = '0' AND reset = '1' THEN
  q <= '1' AFTER 2 ns;
  qb <= '0' AFTER 4 ns;
 ELSEIF set = '0' AND reset = '0' THEN
  q <= '1' AFTER 4 ns;
  qb <= '1' AFTER 4 ns;
 END IF;
END PROCESS
END sequential;
```

As seen in the previous example, a VHDL process statement consists of several segments. The first part is called the sensitivity list, and contains a list of signals that initiate the execution of the process.

The second part of the statement is used for declarations of potential local variables, and is known as the *declarative region*. This part of the process statement is located between the end of the sensitivity list and the BEGIN instruction.

The main body of the process is contained in the process statement part. The process statement starts with BEGIN and ends with END PROCESS

In our own work, we have combined structural and behavioral models of hardware. However, we did not make use of sequential processes.

VHDL object types include:

1. Signals - representations of physical connections between hardware components.

2. Variables - local storage of temporary data, valid only inside a single process.

3. Constants - specifically named values.

Signals are the means of communication of dynamic data between entities.

Example 5.10 *The following statement is a declaration of a system clock signal in VHDL:*

```
SIGNAL clk : std_logic := '1';
```

Signals can be global to entities or architecture local signals.

Variables are intended to be local storage of data specific to an individual process.

Example 5.11 *In terms of VHDL, syntax variables are declared using statements similar to the following code:*

```
VARIABLE a : INTEGER;
```

The difference between signals and variables is fundamental. Signals are intended to model physical processes in electronic circuits. Therefore, they must emulate the delays of the circuits. Signals are scheduled. No signal assignment is instantaneous. In contrast, variables are abstract data representations. Variable assignments happen immediately. However, they cannot be used to model real operation of hardware. One consequence of this is that variables require less memory, while signals need more information for scheduling and signal attributes.

Constants are introduced in order to achieve better code readability.

VHDL supports scalar and composite data types.

Scalar types represent singular data, in exactly the same way as in other formal languages. Scalar data types supported by VHDL encompass four general classes:

1. Integer types,

2. Real types,

3. Enumerated types,

4. Physical types.

We do not go into details of Integer, Real and Enumerated data types, as they are similar to data types in other programming languages. Physical data types are of special interest because they represent the main difference in comparison to other programing languages. This difference is again dictated by the demands of the hardware design modeling. Physical data types are used to represent physical quantities. The only predefined physical type present in the VHDL standard is the TIME type.

Example 5.12 *In this example the variable* `current` *is declared as a* `TIME` *type variable with a fixed range of values.*

```
TIME current IS RANGE 0 to 100000000
```

Complex data types include two classes:

1. Arrays,
2. Records.

Arrays represent groups of individual data of the same type.

Example 5.13 *A 32-bit bus of a device, can be declared using an array of 1 bit elements.*

```
TYPE data_bus IS ARRAY (0 TO 31) OF BIT;
```

The VHDL standard supports multi-dimensional and unconstrained array types. However, signals specified using such arrays may not always be simulated by software systems or implemented in hardware.

Finally, VHDL supports records, as the most complex data type. Records are groups of data of different types handled as a single object.

Example 5.14 *The following fragment of a code represents and example of a record consisting of one integer and one real variable:*

```
RECORD
 a : INTEGER;
 b : REAL;
END RECORD;
```

5.10 IMPLEMENTATION OF SWITCHING FUNCTIONS USING FPGA DEVICES

Field programmable gate arrays (FPGAs) are a useful tool for rapid prototyping as well as for small volume production of logic circuits. In general terms, these devices belong to a wider family of Universal Circuit Array devices. In recent years with the improvement in their performance, FPGA devices have made significant inroads into traditional ASIC areas.

A great part of our work is devoted to experiments with multiple FPGA families, so we devote special attention to these devices in what follows.

These FPGA devices consist of logic blocks capable of realizing a set of basic functions, programmable interconnections and switches between blocks. The logic blocks vary in nature from simple pairs of transistors, more complex basic logic gates (AND, OR, NAND, NOR) and multiplexers to Look-up Tables and more complex multi input AND-OR structures.

The complexity of logic blocks determines the complexity of a FPGA device. In general FPGA devices form a whole spectrum with fine-grain devices on one end and the so-called coarse-grain devices on the other end. The complexity of logic blocks is determined according to one of several criteria:

1. the number of NAND circuits,

2. number of transistors,

3. number of inputs and outputs,

4. normalized area of logic blocks vs. the total area of the FPGA device.

The implementation of a function using an FPGA device involves a multi level decomposition of the function.

The granularity of the FPGA device determines its performance and influences the design method to a large extent. Larger logic blocks are capable of implementation of larger subfunctions and, therefore, require a smaller number of decomposition levels, and consequently fewer routing resources and smaller propagation delay. However, larger blocks are generally slower and less efficiently manipulated.

The question of optimal granularity of FPGA devices has been discussed in literature, for example, in [3], [96] and [137].

Routing in FPGA devices is established by connecting segments of hard-wired lines with programmable switches. The number of these segments determines the density of elements on an FPGA device [168]. A large number of available logic blocks implies a large number of segments. However, the increase in the number of segments increases the number of necessary programmable switches. In modern VLSI technology, the interconnections and switches represent the part of the circuit which introduces the biggest amount of delay.

The topic of optimal number and length of interconnections of switches has also been discussed in literature [138]. However, this problem is closely linked with the particular architecture employed by the particular FPGA device.

The manner in which the logical blocks are programmed also varies in different families of FPGA devices. In general, FPGA devices can be hard or soft programmable.

Hard programmable FPGA devices are programmed by setting of programmable switches between segments of interconnections. These switches are open circuits that can be converted into short circuits by applying a current pulse, thus establishing contact between logic blocks. Antifuse-based FPGA devices represent this group.

An antifuse can exist in three separate states, ON, OFF, and a transition ON-OFF state. Programming of antifuse is performed with a mixed sequence of digital control and high voltage analog waveform. The programming of antifuse involves the change in its electro-chemical properties and is irreversible.

Soft programmable FPGA devices are usually LUT-based. The programming methods for LUT-based FPGAs are equivalent to the programming of other ROM-based devices.

The most common type of devices found in commercially available product families are 4-input LUT-based devices, with theoretically more optimal, 6-input LUT devices recently gaining prominence.

FPGA is a relatively new technology. In recent years, methods for FPGA synthesis have been proposed at a rapid pace. Most of these methods have been especially tailored for particular FPGA architectures. Therefore, it is hard to present all of them in a systematic manner.

In broad terms, it can be said that all of these methods consist of two main design phases:

1. Technology-independent optimization, during which the particular properties of logic elements are disregarded. This phase focuses on general minimization of the given function, following methods that apply to logic design in general.

2. Technology mapping, which is a realization of the desired function by modification of parameters of programmable blocks of an actual FPGA device.

In an FPGA device, the term *module function* indicates a single-output switching function implemented by an individual module. Depending on the type of device, the module function can be fixed as a choice of functions is offered to the designer. *Cluster function* is a function that describes the functionality of a whole section of the network, i.e., a group of interconnected modules.

Implementation of a switching function using an FPGA device represents a process of personalizing the given device by adapting some of its parameters. The main goal is to obtain a network of programmable modules which is equivalent to the given function, with a minimal number of nodes and critical path delay.

In the case of antifuse-based FPGAs, individual modules implement the single identical fixed function. New larger functions are implemented by manipulating interconnections between modules. On the other hand, LUT-based programmable modules with n inputs are capable of implementing any of switching function of n variables, by storing its truth vector in the memory matrix. They can implement certain functions of more than n variables under some restrictions. The implementation of a function is achieved both by manipulating the interconnections of modules and the contents of LUTs.

Decision diagrams have served as a basis for many implementation algorithms for FPGA platforms.

Some design methods related to antifuse FPGA devices can be found in [75], [76], [126], and [127]. Methods for LUT-based devices are presented in [16], [64], [111], [122], [167], [178], and [183].

5.11 DECISION DIAGRAMS AND THE XSLT CONVERSION MECHANISM

In the following sections, we present examples of application of the proposed XML-based framework to several different tasks in logic design. The methods described in this section were originally proposed in [161].

In all of these examples we encounter abstract XML documents describing decision diagrams which are converted into some application specific format during the process. This conversion is always achieved by means of the XSLT transform mechanism.

The XML-based framework is intended to provide a universal method of representing a nuber of decision diagram classes. Application specific formats are designed with other intended purposes. They are most often much more specific in scope.

The transition from a universal abstract representation to a more focused application specific format is the task with which the XSLT transform mechanism is charged. This change in focus is manifested in two ways, the discarding of the redundant information that might be present in an explicit form in XML documents, and extracting information present in an implicit form from the diagram.

The discarding of explicit lists of parent nodes in some tasks is an example of the first aspect, calculating relative geometric positions of diagram nodes in 2D planes for the visualization purpose is illustration of the other.

Both these kind of problems are handled by the XSLT transform mechanism.

In general, we design a separate set of XSLT style sheets for each specific application. However, all of the employed XSLT style sheets share some basic features.

Each XSLT style sheet consists of a set of templates. A separate XSLT template is designed for processing each of the decision diagram structural features, such as nodes, edges, etc. An application-specific XSLT style sheet focuses only on elements of the structure of interest for the particular application, ignoring the others. In this way the same XML documents can be safely used for different applications. A system will not reject a document containing the additional data not anticipated by the system.

Furthermore, on several places XSLT templates are employed to perform some needed calculations. Once again, the calculation of relative geometric coordinates of structural elements for visualization purposes is a good example.

Due to the nature of the required conversions, on several places the transform process is divided into multiple steps. In this case an intermediary, usually somewhat simplified, XML document is employed by the system. Each step of such a process is performed by a separate dedicated XSLT style sheet. Where this is the case, this organization was chosen for the sake of the algorithm efficiency.

5.12 APPLICATION TO FUNCTIONAL BINARY DECISION DIAGRAMS

In the following sections we present examples of the application of the proposed XML based framework for hardware implementation of discrete functions.

We focus first on binary functions, represented by Reduced Ordered Binary Decision Diagrams, Kronecker, and Pseudo-Kronecker Decision Diagrams, and implementation using FPGA devices. The devices considered in the first set of examples belong to Xilinx Spartan 3 and Altera Stratix III FPGA families.

Final hardware designs are represented using VHDL syntax.

Possible expansion rules in functional binary decision diagrams include the following:

1. Shannon, $f = \bar{x}_1 f_0 \oplus x_1 f_1$,
2. Positive Davio, $f = f_0 \oplus x_1 f_2$,
3. Negative Davio, $f = f_1 \oplus \bar{x}_1 f_2$.

Fig. 5.1 represents logic circuits realizing each of these three expansions. Each node in the decision diagram thus corresponds to a hardware unit realizing the given expansion. Hardware realizing a switching function using a decision diagram is a network of interconnected units performing the given expansion.

Since the basic functionality is repeated for every node in the decision diagram, we can reuse the same basic logic circuits in the hardware implementation. Corresponding hardware architecture consists of a library of basic components realizing three different expansions and the network of interconnected instances of these components.

Example 5.15 *Hardware realization of a logic function, $f(x_1, x_2, x_3) = x_1\bar{x_2} \vee x_2 x_3$, using the corresponding BDD can be seen in Fig 5.2.*

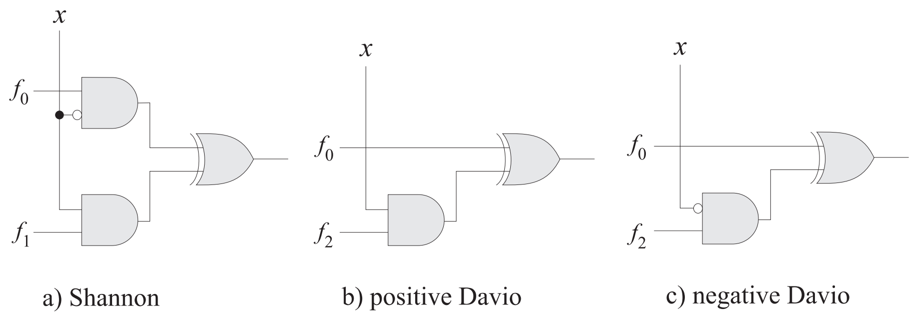


Fig. 5.1 Logic circuits corresponding to three types of expansions, $f_2 = f_0 \oplus f_1$.

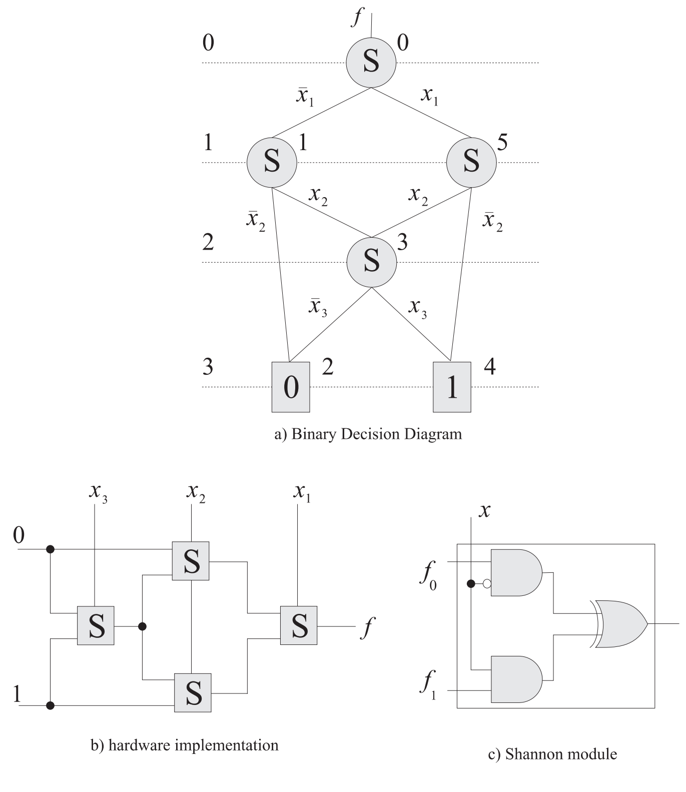


Fig. 5.2 BDD and a logic network corresponding to the function $f(x_1, x_2, x_3) = x_1\bar{x_2} \vee x_2x_3$.

All VHDL hardware designs produced by the proposed system share a common form, due to the fact that all binary decision diagrams have the same well-defined basic structure.

The design consists of one top-level entity and its associated structural architecture. Since we deal with binary decision diagrams, the number of incoming ports is determined by the number of logic variables of the implemented switching function. Two additional input ports are provided for the logical constants. Since we have limited ourselves to dealing only with single output decision diagrams, the top-level entity has only one output port.

Separate entities representing hardware implementations of each of the three standard decomposition rules are provided. References to these entities will be created appropriately for each node of a decision diagram in the architecture section of the design. The first part of the architecture contains declarations of all the external entities used in the architecture description. In our case, the internal signals of the hardware architecture correspond to the edges of the decision diagram. Since we are dealing with binary decision diagrams, the std_logic type is used for ports of entities and internal signals of the design.

Therefore, the basic outline of a VHDL design looks as follows:

```
ENTITY entity_name IS
  PORT ( list_of_input_ports :
  IN std_logic;  output_port :
  OUT std_logic);
END ENTITY entity_name;
ARCHITECTURE arch_name OF entity_name IS

...declarations of components as needed...

 COMPONENT my_shannon
  PORT (f0, f1, x : IN std_logic; z :
  OUT std_logic);
 END COMPONENT;
 COMPONENT my_pdavio
  PORT (f0, f1, x : IN std_logic; z :
  OUT std_logic);
 END COMPONENT;
 COMPONENT my_ndavio
  PORT (f0, f1, x : IN std_logic; z :
  OUT std_logic);
 END COMPONENT;

SIGNAL internal_signals : std_logic;
BEGIN

  ...actual structure of the decision diagram...

END arch_nam;
```

The key step in developing an XSLT style sheet for any particular transformation is to identify the correspondence between entities of the source and the target XML document formats. The process of producing a VHDL hardware description from XML documents consists of two distinct steps:

1. Preprocessing,
2. High-level synthesis.

This two-stage organization of conversion process is convenient for several reasons. In Section 3.2 we have stated that the structure of XML documents is designed to follow the recursive properties of decision diagrams. This is a logical choice as recursion is an inherent property of both XML and decision diagrams. However, the concept of recursion does not exist in hardware design. It is incompatible with VHDL. Therefore, a preprocessing step is necessary to remove the recursive aspects of the original XML document and produce a more suitable version of the XML representation of the decision diagram. A separate XSLT stylesheet is used for this purpose.

The system takes the original XML document as input, applies the preprocessing script, and produces an intermediary XML document which is used as a starting point for the second stage of the conversion process.

The original XML wrapper element `<dd:tree>` is copied to the intermediary document and its attributes are retained. Nodes of the diagram are represented with `<node>` elements in an intermediary document. This new `<node>` does not have any additional nested elements of the same type. In this way a complex recursive structure of the original document is converted into a simple net list. The attributes of the non-terminal nodes are preserved.

For example, a set of non-terminal nodes is represented in the original XML document with the following fragment of code:

```
<dd:next terminal="0" id="928"
constant="-1" level="2" rule="Shannon">
 <dd:next terminal="0" id="472"
 constant="-1" level="3" rule="Shannon">
  <dd:next terminal="0" id="203"
  constant="-1" level="4" rule="Shannon">
     .
     .
     .
  </dd:next>
 </dd:next>
</dd:next>
```

is converted to the list of the form:

```
<node terminal="0" id="928"
constant="-1" level="2" rule="Shannon">
 ...
```

```
</node>
<node terminal="0" id="472"
constant="-1" level="3" rule="Shannon">
 ...
</node>
<node terminal="0" id="203"
constant="-1" level="4" rule="Shannon">
 ...
</node>
```

There is no logical operation performed in terminal nodes of decision diagrams. Terminal nodes simply indicate values of the logical constants. Therefore, terminal and non-terminal nodes will assume different roles in the RTL model. This difference is clearly indicated in the intermediary XML document by the storing of the terminal nodes in a separate list.

The complex structure of nested `<dd:children>`/`<dd:next_child>` elements, representing the linked list of descendants of the node, is replaced with a simple list of `<child>` elements, each containing a pointer to a descendant node as an attribute. Again, we illustrate this by an example.

A list representing a pair of nodes from the original document:

```
<dd:children point="1596" variables="0">
 <dd:next_child point="387" variables="1" />
</dd:children>
```

is represented in the intermediate document as:

```
<child point="1596" variables="0"/>
<child point="387" variables="1" />
```

Parent lists are discarded totally, as they are not needed for this particular application.

In this way all recursive aspects of the original XML document are discarded.

The second step of the process represents the final conversion of an intermediate XML form to VHDL by using a separate XML style sheet.

The XSLT style sheet first identifies `<tree>`, the root element of our XML hierarchy. This is a wrapper element and it corresponds to the general template of a VHDL document. A skeleton of a VHDL document is created at this stage including the header with the declaration of a new entity. The name `my_DD` is given to this entity by default.

The XSLT style sheet generates a port list of `my_BDD` entity automatically. Ports with names con* (where '*' denotes a numeral) represent inputs for logical constants. Since we are dealing exclusively with binary logic in this example, there is only two of them, con1 and con0. Logical variables of the discrete functions are mapped to ports with names of the form x*. Their number is determined by the number of levels in the decision diagram. One output port is assigned, with the default name ’o’. All declared ports are of the type `‘std_logic’`.

Next, the XSTL style sheet generates a declaration of architecture and associates it with the declared entity. Name `my_dd_arch` is assigned by default to this architecture. Depending on the value of the `‘type’` attribute in the tree element declarations for the appropriate components will be included in the VHDL document. Either a single `my_shannon`, `my_pdavio` or `my_ndavio` component or the combination of components will be included in the declaration.

Edges of the decision diagram correspond to internal signals of the VHDL architecture. These signals are declared in the header of the architecture declaration. The number of the internal signals is equivalent to the number of non-terminal nodes excluding the root node. The names assigned to these signals are of the form o*, where ’*’ denotes a numeral.

What follows is the body of the hardware architecture with the list of component instances representing the nodes. The component type is selected according to the value of the `‘rule’` attribute of the node element. The port map is generated for each component instance by analyzing the list of its children. Each component has two inputs and one output of the type `‘std_logic’`.

Appropriate register entities are added to the design to facilitate the connections of the described unit with other modules. The register entities can be a part of a pre-generated module library or generated when needed by using XSLT. The XSLT can be used to automatically produce a test bench for the RTL model generated in this way.

An example of the complete code of VHDL implementation of the function 9sym from the MCNC benchmark set [181], generated using this XML document is available in Apendix B.

5.12.1 Efficiency of Implementation of FDDs

To test the validity of the proposed concept, we have used the system to automatically produce RTL descriptions in VHDL of hardware implementations of a number of binary functions. We have focused on functions from the MCNC (Microelectronics Center of North Carolina) set of logic design benchmark functions [181]. We produced a number of different binary decision diagrams for each function. The binary decision diagrams were converted to valid XML documents. For this task, we have used a software created by Suzana Stojiljković at the Faculty of Electronics at the University of Niš, Niš, Serbia, [162], in conjunction with an export module capable of producing the XML documents. We have used these XML documents as an input to the proposed system, to produce hardware descriptions in VHDL. We have made the final RTL synthesis of the produced VHDL code for various FPGA technologies, using the Mentor Graphic Leonardo Spectrum synthesis tool.

Table 5.1 Selected benchmark functions implemented using a Shannon binary decision diagram and Xilinx Spartan 3 FPGA technology.

function	num. of inputs	max. freq.	slack	acc. inst.
misex1_5	8	163.6	0.1	24
alu4_0	14	163.6	0.1	38
cordic_0	23	83.1	0.5	93
apex2_0	39	40.2	0.2	3006
apex1_0	45	122.9	0.2	56

Table 5.2 Selected benchmark functions implemented using a binary decision diagram with a positive Davio decomposition and Xilinx Spartan 3 FPGA technology.

function	num. of inputs	max. freq.	slack	acc. inst.
misex1_5	8	163.6	0.1	27
alu4_0	14	143.0	0.2	66
cordic_0	23	163.6	0.1	71
apex2_0	39	130.4	0.14	54
apex1_0	45	146.6	0.3	39

In this section, we present the results obtained for five representative functions selected from the benchmark library. For each function, we have generated four different binary decision diagrams:

1. a binary decision diagram with the Shannon decomposition at each level,
2. a binary decision diagram with the positive Davio decomposition at each level,
3. a Kronecker binary decision diagram with the positive Davio applied at the topmost level and the Shannon derivative on all other levels,
4. a Kronecker binary decision diagram with the positive Davio applied at the topmost and lowest level and the Shannon decomposition in all other levels.

We have tested the examples for two FPGA families from two different companies, the Stratix III family from Altera, and the Spartan 3 from Xilinx. The first set of tables represents the results of implementation of selected functions by using the Spartan 3 product family.

The second set of tables corresponds to the results obtained for the Stratix III FPGA technology from Altera.

To better illustrate these results, we present in Fig. 5.3 a binary decision diagram for the function misex1_5 from the benchmark set generated by recursive application of the Shannon expansion and a RTL schematic of its hardware implementation in VHDL. Please notice that logic variables x_4 and x_7 do not have any influence on the function of the circuit. This is due to the

Table 5.3 Selected benchmark functions implemented using a Kronecker binary decision diagram and Xilinx Spartan 3 FPGA technology.

function	num. of inputs	max. freq.	slack	acc. inst.
misex1_5	8	163.6	0.1	31
alu4_0	14	129.5	0.3	77
cordic_0	23	83.1	0.5	93
apex2_0	39	40.2	0.2	3006
apex1_0	45	122.9	0.2	56

Table 5.4 Selected benchmark functions implemented using a Kronecker binary decision diagram with different choice of decomposition rules and Xilinx Spartan 3 FPGA technology.

function	num. of inputs	max. freq.	slack	acc. inst.
misex1_5	8	163.6	0.1	26
alu4_0	14	129.5	0.3	77
cordic_0	23	83.0	0.4	93
apex2_0	39	40.2	0.2	3006
apex1_0	45	120.7	0.1	53

Table 5.5 Selected benchmark functions implemented using a Shannon binary decision diagram and Altera Stratix III FPGA technology.

function	num. of inputs	max. freq.	slack	acc. inst.
misex1_0	8	392.2	0.0	17
alu4_0	14	175.0	0.2	50
cordic_0	23	116.3	0.1	65
apex2_0	39	46.2	0.6	1625
apex1_0	45	43.8	0.4	39

Table 5.6 Selected benchmark functions implemented using a positive Davio binary decision diagram and Altera Stratix III FPGA technology.

function	num. of inputs	max. freq.	slack	acc. inst.
misex1_0	8	392.2	0.0	17
alu4_0	14	151.4	0.1	48
cordic_0	23	149.7	0.2	60
apex2_0	39	166.6	0.1	43
apex1_0	45	210.8	0.0	33

Table 5.7 Selected benchmark functions implemented using a Kronecker binary decision diagram and Altera Stratix III FPGA technology.

function	num. of inputs	max. freq.	slack	acc. inst.
misex1_0	8	392.2	0.0	17
alu4_0	14	167.0	0.1	53
cordic_0	23	114.6	0.0	65
apex2_0	39	43.5	0.3	39
apex1_0	45	173.5	0.0	44

Table 5.8 Selected benchmark functions implemented using a Kronecker binary decision diagram with a different choice of decomposition rules and Altera Stratix III FPGA technology.

function	num. of inputs	max. freq.	slack	acc. inst.
misex1_0	8	392.2	0.0	17
alu4_0	14	167.0	0.1	53
cordic_0	23	106.0	0.1	65
apex2_0	39	43.5	0.3	39
apex1_0	45	215.1	0.0	38

fact that misex1_5 is the fifth output of the multi-output musex1 function. In Fig. 5.4 we show a binary decision diagram created using the positive Davio decomposition for the same switching function and the corresponding RTL schematic. The evident simpler structure of this circuit clearly indicates the advantage of the choice of a different decomposition rule.

5.13 COMPARISON OF EFFICIENCY OF QUATERNARY AND BINARY DECISION DIAGRAMS

In the following example we turn our attention to the implementation switching functions using Quaternary Decision Diagrams and LUT-based FPGA devices. The following discussion was originally published in [158].

One of important properties of modern LUT-based FPGA devices is that the delay time for interconnections will often be larger than that for the LUTs. The complexity of interconnections is inversely proportional to the size of logic functions implemented by LUTs. In commercially available LUT-based FPGA devices, interconnections are much slower than in other masked type gate-arrays. The size and the complexity of an individual programmable block versus the complexity of interconnections in a particular FPGA device family represents a trade-off that must be balanced. It can be shown [144], that a six-input LUT represents an optimal solution for this problem.

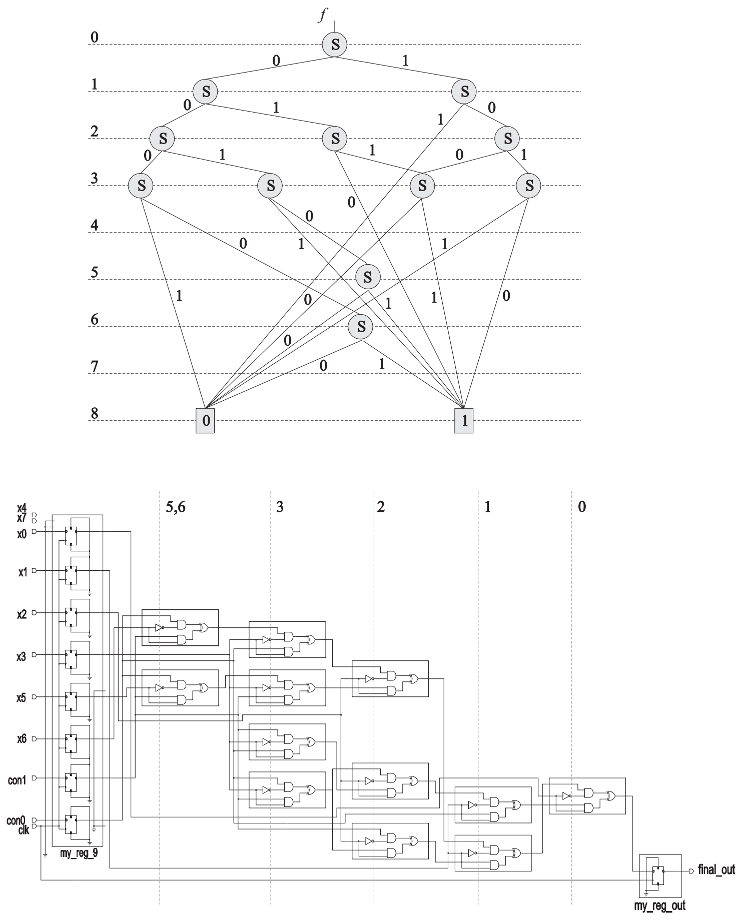


Fig. 5.3 Decision diagram created using a Shannon decomposition rule and RTL schematic for the fifth output of the misex1 benchmark function.

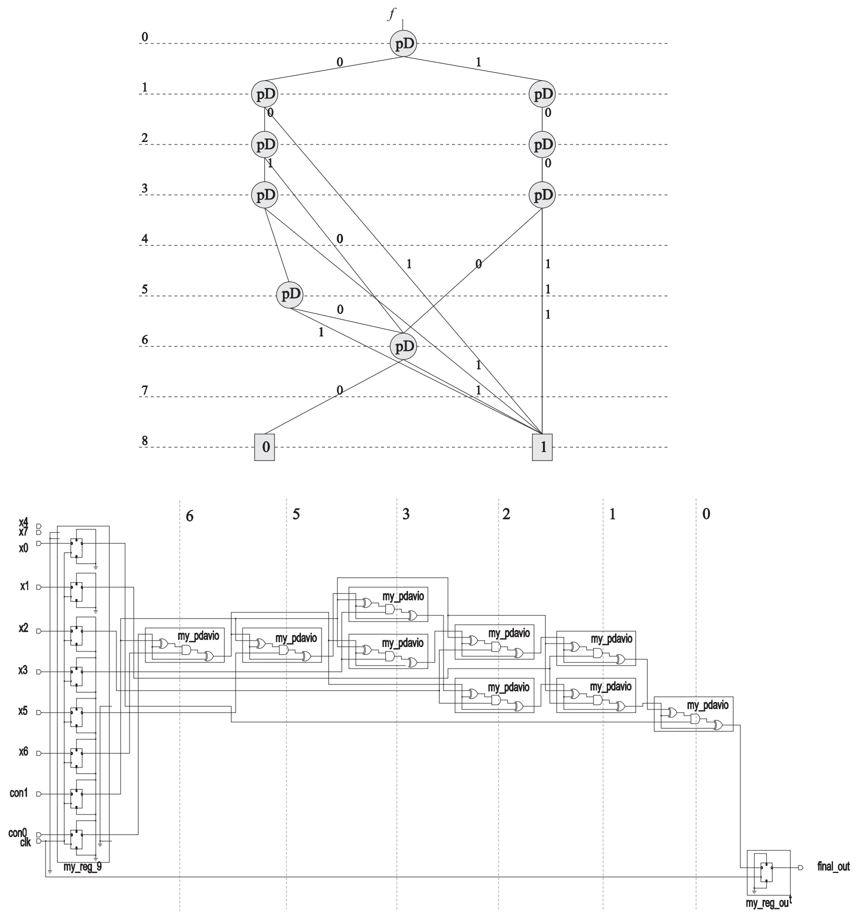


Fig. 5.4 Decision diagram created using a positive Davio decomposition rule and RTL schematic for the fifth output of the misex1 benchmark function.

Virtex-5, introduced by Xilinx, Inc., is a family of FPGA devices based on six-input LUT devices. A detailed technical specification of this family of devices can be found in [172].

A six-input LUT can natively realize an arbitrary $k = 6$ switching function. A 4-to-1 multiplexer controlled by two-bit control inputs is one such function. Let f_0, f_1, f_2, f_3 be inputs, and S_0, S_1 be the control signals of a multiplexer. The combination of values of S_0 and S_1 determines the output of the signal. The function is of the form:

$$\bar{S_0}\bar{S_1}f_0 + \bar{S_0}S_1f_1 + S_0\bar{S_1}f_2 + S_0S_1f_3. \tag{5.1}$$

When expressed using hexadecimal values with four bits per symbol, the truth vector of this function of length 64 is FF00F0F0CCCCAAAA. Therefore, in order to implement the functionality of a 4-to-1 multiplexer, using a six-input LUT we need to store this vector in the LUT.

The same 4-to-1 multiplexer circuit represents a hardware implementation of the Shannon decomposition for a four-valued case, the basic element of the Quaternary Decision Diagram. Quaternary decision diagrams (QDDs) have been proposed in [144], by Sasao and Butler, as a natural choice to efficiently represent logical circuits for implementation with six-input LUT FPGA technology. However, to our knowledge, no commercially available logic design software tools exploit these properties of QDDs.

Therefore, we implemented several functions represented in terms of QDDs using the Virtex-5 family by Xilinx, and did an analysis of the complexity of these implementations.

The final output of the system is a netlist in EDIF format, suitable for use with standard place and route tools.

A quaternary decision diagram of a function can be generated from its truth vector by a direct application of the Shannon decomposition and related reduction rules. In many cases, however, it is more convenient to use a BDD of a function as a source, and to produce a reduced QDD based on it. There is a wide variety of software tools for the minimization of BDDs, and the XML framework which we use as a basis of the proposed system is already well adapted to handling BDDs. By taking this approach, no separate mechanism for optimization of QDDs is necessary.

Quaternary decision diagrams are a special case of more general multi-valued decision diagrams. Software methods for dealing with multi-valued decision diagrams have been discussed in several publications, for example, in [117] and [118].

Assume that a function to be realized has $n = 2r$ variables. Let $X = (x_1, x_2, ..., x_n)$ be the set of input variables of this function. Let $Q = \{0, 1, 2, 3\}$ and $B = \{0, 1\}$. A mapping $f{:}Q^r \rightarrow B$ is a four-valued input two-valued output function. An arbitrary switching function of n variables $f(X)$ can be converted into a four-valued two output function as follows: $f_Q(X_1, X_2, ..., X_r) = f_B(X)$, where $X_i = (x_{2i-1}, x_{2i})$ takes the value 0, 1, 2, or 3 if and only if $(x_{2i-1}, x_{2i}) \in \{(0, 0), (0, 1), (1, 0), (1, 1)\}$, respectively.

Table 5.9 An example of the encoding of a binary and into an equivalent quaternary function.

x_1	x_2	x_3	x_4	f	X_1	X_2	
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	2	0
0	0	1	1	1	0	3	1
0	1	0	0	1	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	1	1	2	1
0	1	1	1	0	1	3	0
1	0	0	0	0	2	0	0
1	0	0	1	1	2	1	1
1	0	1	0	0	2	2	0
1	0	1	1	1	2	3	1
1	1	0	0	1	3	0	1
1	1	0	1	1	3	1	1
1	1	1	0	0	3	2	0
1	1	1	1	0	3	3	0

Example 5.16 *In Table 5.9 we present a four-variable binary function and its equivalent two quaternary input one binary output function.*

It is evident from Example 5.16, that four-valued logical variables X of the function f_Q are obtained by pairing binary variables x of the original function, f_B, $X_i = (x_{2i-1}, x_{2i})$. It follows that each node in the QDDs corresponds to a grouping of 3 nodes in a BDD, a node at the $2i-1$ level and its two immediate descendants from the $2i$-th level. Fig. 5.5 shows the correspondence between nodes of a binary decision tree and a quaternary decision tree.

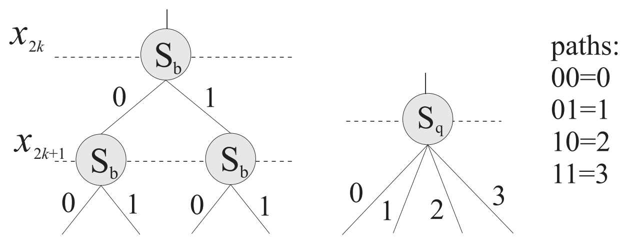


Fig. 5.5 Relationships among nodes of binary and quaternary decision diagrams.

Notice that pairing variables in different ways may reduce the size of quaternary decision diagrams [144].

Diagram conversion proceeds as follows. Since the total number of levels in a quaternary decision diagram is $n/2$, half of the number of levels in equivalent BDDs, the main processing will be done on nodes on even levels, assuming that the root node of the diagram belongs to the level 0.

Each binary node on an even level will be replaced by a four output node. The edges of the new node point to the grandchildren of the original node. These edges will be labeled by values read from paths that connect the original node to its grandchildren. If one of the outgoing edges of the original node points directly to a terminal node, it will be replaced by two respective branches pointing to the same terminal node. Notice, that since we have assumed that the original switching function has $n = 2r$ variables, the terminal nodes will belong to the odd level, $n + 1 = 2r + 1$. Odd level non-terminal nodes are discarded.

However, this simple transformation is not sufficient in the case of ROBDDs. An even non-terminal node might point to a child node which is not on the level immediately below. An odd level non-terminal node, that has a direct parent two or more levels higher in the diagram must not be removed. Instead, it needs to be moved to the even level immediately above. Fig. 5.6 illustrates this situation. Finally we are free to discard other odd level non-terminal nodes.

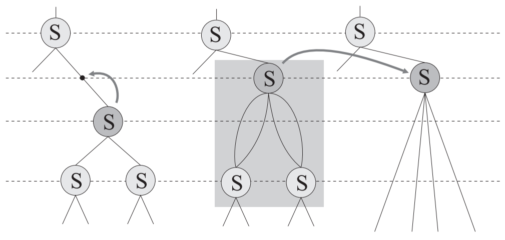


Fig. 5.6 Replacement of an odd level binary node with a quaternary node.

We can formally state the algorithm for the conversion of a BDD into a QDD as follows:

1. Start from the root node at level 0.
2. If a node is at an even level $l = 2j$, replace the node with a four-output node with pointers to the original grandchildren.
3. Else check levels of all parents of the current node.
4. If the current node has a parent two or more nodes higher in the hierarchy, move it to the first even level immediately above.
5. Discard all other odd level nodes.

5.13.1 XSLT Implementation of the Conversion Algorithm

This particular transformation algorithm can be efficiently implemented using a set of XSLT templates grouped into three separate style sheets which are applied in a consecutive order to the source XML document representing a BDD. The first two style sheets correspond to the preprocessing stages of the conversion process. These two stages represent the actual implementation of the algorithm described in Section 5.13.

In general, XSLT establishes the correspondence between objects of the source and destination XML hierarchies. In this particular case, groups of nodes of BDDs will be replaced with four output nodes of a quaternary decision diagram.

The system scans the source XML document, looking for elements of the type `<dd:node>`. Whenever an element of this type is encountered, a template `node_temp` is evoked. This template examines the `level` attribute of the node element. If a node belongs to an even level, it is replaced with a four-valued node in the new intermediary XML document.

A non-recursive list of child nodes will be created for each four-valued node. This list is created by scanning the list of

`<dd:children>`, `<dd:next_child>`

subelements of the `<dd:node>` element of the source document. For each of these elements an appropriate XSLT template is evoked.

If an edge of the original node points to a non–terminal node on the level immediately below, pointers to the two grand-child nodes will be placed in the list in the intermediate XML document, as shown in Fig. 5.7. We also provide examples of XML code for binary and corresponding four-valued node for comparison.

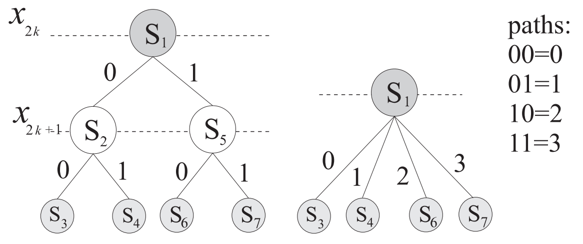


Fig. 5.7 Replacement of an odd level binary node with children on a level immediately below.

```
<node id="2877" level="0" terminal="0" rule="NA">
 <child id="1141" variables="0" />
 <child id="1815" variables="0" />
 <child id="2314" variables="1" />
 <child id="2875" variables="1" />
</node>
```

If an edge points to a terminal node or a non-terminal node two or more levels down in the hierarchy, two pointers to the same child node will be added to the list. This situation can be better understood by observing Fig. 5.8 and the following code sample.

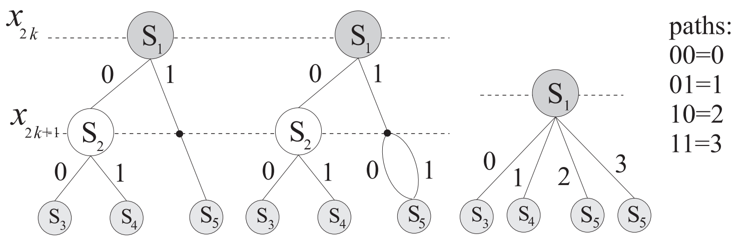


Fig. 5.8 Replacement of an odd level binary node with children deeper in the hierarchy.

```
<node id="12" level="8" terminal="0" rule="NA">
 <child id="0" variables="0" />
 <child id="0" variables="0" />
 <child id="1" variables="1" />
 <child id="0" variables="1" />
</node>
```

Non-terminal nodes belonging to odd levels will undergo a somewhat different treatment. As stated before, these nodes should be removed unless they have a parent which is located two or more levels higher in the decision diagram. The information about parents of the node will be extracted from the original XML document and transferred to the intermediary XML file. The system associates a non-recursive list of parent nodes with each of these elements. Information about corresponding levels is associated with each `<parent>` element. The following code example represents a node element in the intermediate XML document with the associated parent list. A list of children of these nodes will be created by repeating twice the pointers to the children of this node, as indicated previously in Fig. 5.6. The following code is an example of the node with a parents list.

```
<node id="199" level="5" terminal="0" rule="NA">
 <parent id="374" papa_level="4" />
 <parent id="2173" papa_level="4" />
 <child id="96" variables="0" />
 <child id="2" variables="1" />
 <child id="96" variables="0" />
 <child id="2" variables="1" />
</node>
```

The second XSLT style sheet discards all non-terminal nodes which do not have parents two or more levels higher in the hierarchy. It will scan the intermediary XML product of the previous style sheet, looking for the nodes with parent lists. Nodes that do satisfy this criterion will be transferred to the next even level immediately above.

As evident from the previous discussion, there exists a direct 1:1 correspondence between quaternary decision diagrams and LUT-based FPGA devices. Each node of a quaternary decision diagrams will be represented by a six-input LUT. Two inputs are used for each four-valued control variable X_i.

Example 5.17 *Fig 5.9 is a FPGA implementation of the function in Table 5.9.*

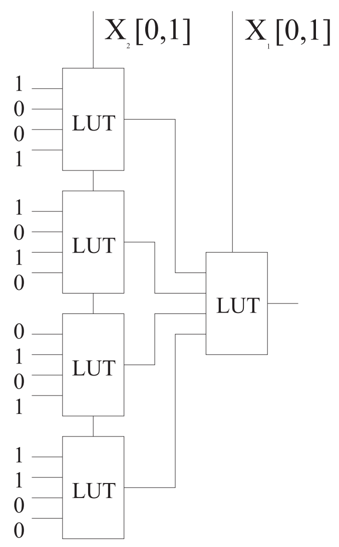


Fig. 5.9 Hardware implementation of a given function using a quaternary decision tree.

In particular, four-valued Shannon decomposition nodes are functionally equivalent to 4-to-1 multiplexers. The netlist implementing a logic circuit described via quaternary decision diagrams will consist of a series of six-input LUT devices corresponding to the nodes of the diagram, connected with nets which are equivalent to the edges of the decision diagram. These netlists can be expressed in EDIF syntax.

Once again we employ a set of XSLT templates to produce the final EDIF documents.

The conversion process begins by establishing the correspondence between elements of XML documents and entities in the EDIF netlist. A series of XSLT templates is applied to convert XML elements to the EDIF syntax.

The XSLT style sheet first identifies the `<tree>` element in the input XML document, and based on the information associated with this element, create the header in the output EDIF document. This template also declares the hardware components that are used in the remainder of the net list, namely, `LUT6`, standard `VCC`, `GND`, and IO buffers.

The QDD device has two input pins for each variable X_i and two additional input pins for logical constants 0 and 1, as well as input pins for clock, power, and ground. There is only one output pin.

Example 5.18 *The following EDIF code is a declaration of a device implementing a three-variable logic function $f(X_0, X_1, X_2)$, where each four-valued logic variable X is represented by two bits.*

```
(cell QDD
      (cellType GENERIC)
        (view view_1
          (viewType NETLIST)
          (interface
  (port XLXN_0
   (direction INPUT)
  )
  (port XLXN_1
   (direction INPUT)
  )
  (port XLXN_2
   (direction INPUT)
  )
  (port XLXN_3
   (direction INPUT)
  )
  (port XLXN_4
   (direction INPUT)
  )
  (port XLXN_5
   (direction INPUT)
  )
  (port XLXN_6
   (direction INPUT)
  )
  (port XLXN_7
   (direction INPUT)
  )
  (port XLXN_8
   (direction INPUT)
  )
  (designator "xc5vlx30-3-ff324")
  (property TYPE (string "QDD") (owner "Xilinx"))
  (property NLW_UNIQUE_ID (integer 0) (owner "Xilinx"))
  (property NLW_MACRO_TAG (integer 0) (owner "Xilinx"))
  (property NLW_MACRO_ALIAS
  (string "QDD_QDD") (owner "Xilinx"))
)
```

The input ports `XLXN_0` *and* `XLXN_1` *represent the logical constants 0 and 1. The pairs of remaining six input ports represent 2-bit logic variables* $\{X_0, X_1, X_2\}$.

The string "xc5vlx30-3-ff324" is a designator code of the particular device in the Virtex-5 FPGA family.

XML `<node>` elements representing quaternary nodes will be replaced by instances of a `LUT6` component implementing the 4-to-1 multiplexer. The first two input pins `I0` and `I1` of each component instance are used for the 2-bit control variable X_i. The remaining four inputs are used for $f(X_i = 0)$, $f(X_i = 1)$, $f(X_i = 2)$, and $f(X_i = 3)$.

Example 5.19 *The following EDIF code represents the declaration of a* `LUT6` *component implementing a 4-to-1 multiplexer:*

```
(cell LUT6
 (cellType GENERIC)
  (view view_1
   (viewType NETLIST)
   (interface
    (port I0
     (direction INPUT)
    )
    (port I1
     (direction INPUT)
    )
    (port I2
     (direction INPUT)
    )
    (port I3
     (direction INPUT)
    )
    (port I4
     (direction INPUT)
    )
    (port I5
     (direction INPUT)
    )
    (port O
     (direction OUTPUT)
    )
   (property TYPE (string "LUT6") (owner "Xilinx"))
   (property XSTLIB (boolean (true)) (owner "Xilinx"))
   (property INIT
   (string "FF00F0F0CCCCAAAA") (owner "Xilinx"))
  )
 )
)
```

An instance of this component is evoked for every non-terminal node using the code:

```
(instance XLXI_<id of the node>
 (viewRef view_1 (cellRef LUT6 (libraryRef UNISIMS)))
   (property XSTLIB (boolean (true)) (owner "Xilinx"))
   (property INIT
  (string "FF00F0F0CCCCAAAA") (owner "Xilinx"))
)
```

where, `<id of the node>` *is the actual value of the* `id` *attribute of the* `<node>` *element.*

The vector FF00F0F0CCCCAAAA represents the content of the LUT, in this case the truth vector of a 4-to-1 multiplexer.

The XSLT style sheet also creates the instances of IO buffers and `VCC` and `GND` components declared in the header.

Next, a series of `Nets` describing the interconnections of hardware components is created. The system evokes the `x_nets` template, which connects the input pins `I0` and `I1` with the pins representing logic variables $X_0, ..., X_i$.

Example 5.20 *For example, the two pins of the logic variable X_2 are connected to each node at level 2 of the quaternary decision diagram.*

```
(net XLXN_x2
 (joined
  (portRef XLXI_4)
  (portRef I0 (instanceRef XLXI_<node 1 id>))
  (portRef I0 (instanceRef XLXI_<node 2 id>))
  ...
 )
)
(net XLXN_x3
 (joined
  (portRef XLXI_5)
  (portRef I1 (instanceRef XLXI_<node 1 id>))
  (portRef I1 (instanceRef XLXI_<node 2 id>))
  ...
 )
)
```

where `XLXI_4` and `XLXI_5` are the pins representing variable X_2, and `<node 1 id>`, `<node2 id>`, etc., are identifiers of the nodes at level 2.

The `const_nets` template creates Nets which connects logical constant input pins to the input pins of n level non-terminal nodes. For constant 0, for each `<child>` element containing the `const = 0` attribute, another connection `portRef I0 (instanceRef XLXI_node id))` is added to the Net declaration.

Example 5.21 *The following code is a net that connects the pin `XLXI_1` representing the logical constant 1 to the inputs of nodes `id = 2`, `id = 3` and `id = 12`:*

```
(net XLXN_c1
 (joined
  (portRef XLXI_1)
  (portRef I2 (instanceRef XLXI_2))
  (portRef I4 (instanceRef XLXI_2))
  (portRef I4 (instanceRef XLXI_12))
  (portRef I2 (instanceRef XLXI_3))
  )
)
```

Finally, the nets connecting non-terminal nodes are created, using the template `nets_template`.

This template creates a net for each `child` element. The XML attribute `pid`, attached to the `child` element, specifies the identifier of the destination `LUT6` component and `cid` specifies the origin of the connection. The input port of the destination `LUT6` component is specified by the `variables` attribute of a `child` element.

Example 5.22 *The following code is an example of the net connecting the output of the node `id = 2736` to the second input port `I2` of the node `id = 2875`:*

```
(net XLXN_2875_2
 (joined
  (portRef I2 (instanceRef XLXI_2875))
  (portRef O (instanceRef XLXI_2736))))
```

The final EDIF netlist is compatible with standard place-and-route tools, such as the ones provided by Xilinx, Ltd. as a part of Xilinx ISE software package.

Table 5.10 Comparison of the size and complexity of binary and quaternary decision diagrams for the selected benchmark functions.

	Depth		Nodes		Size (KB)	
Name	BDD	QDD	BDD	QDD	BDD	QDD
EX1010_0	10	5	467	85	124.0	22.4
EX1010_5	10	5	485	84	131.0	22.6
EX1010_9	10	5	509	91	142.0	24.5
MISEX1_0	8	4	35	6	3.9	1.89
MISEX1_3	8	4	35	7	3.9	2.14
MISEX1_6	8	4	20	3	2.8	1.09
MISEX3_0	14	7	950	169	435.0	45.1
MISEX3_4	14	7	524	93	160.0	25.1
MISEX3_8	14	7	365	73	86.6	19.7
MISEX3_12	14	7	455	79	124.0	21.3
RD84_0	8	4	71	12	8.81	3.41
RD84_1	8	4	25	4	6.07	3.04
RD84_2	8	4	44	7	5.07	2.05
RD84_3	8	4	71	12	8.81	3.37
SAO2_0	10	5	134	25	19.8	6.79
SAO2_1	10	5	128	24	18.9	6.41
SAO2_2	10	5	143	24	21.7	6.62
SAO2_3	10	5	137	25	20.6	6.91
SQRT8_0	8	4	7	1	0.888	0.5
SQRT8_1	8	4	14	2	1.6	0.8
SQRT8_2	8	4	35	6	4.0	1.8
SQRT8_3	8	4	71	10	8.87	2.93

5.13.2 A complexity Comparison of QDD and BDD Based Implementation

In this section we present a comparison of the complexities and the efficiencies of the FPGA implementations of selected benchmark functions using quaternary decision diagrams and other methods of logic design. We have used functions from the MCNC set of benchmarks, [181], for both sets of experiments.

All of the experiments were conducted on an Intel Pentium4 3.20Ghz with 1.99GB of RAM reference machine.

In the first set of experiments we compare QDD to BDD-based implementations of switching functions. In these experiments, the outputs of multi-output functions are treated as separate binary functions, i.e. EX1010_5 indicates the fifth output of the benchmark function EX1010.

For each function a reduced ordered BDD was created by the proposed algorithm in XSLT. We have used the software proposed in [162] for the optimization of BDDs. Next, a quaternary decision diagram was created from the original BDD, using the proposed system.

In the first two columns of the table we compare the sizes of binary and quaternary decision diagrams of each benchmark function. In the next two columns we compare the numbers of levels. As evident from the theoretical comparison of binary and quaternary decision diagrams, the number of non-terminal nodes as well as the depth of the QDD are always significantly smaller than for the equivalent BDD. This fact is reflected in the size and complexity of final designs. Last two columns of this table represent the size of the produced XML files. The XML representation of QDDs is always (in some cases considerably) smaller due to the reduced number of non-terminal nodes and interconnections.

In the second set of experiments, we compared the proposed FPGA design method to a standard industrial approach. We compared the quality of the synthesis obtained by creating quaternary decision diagrams and synthesis by Xilinx ISE software system. Multi-output switching functions from the MCNC set of benchmarks [181] were originally represented as lists of cubes. In the first phase of the experiments, the benchmark functions represented in this way are converted into VHDL representations. Circuit synthesis, technology mapping, placing and routing have been performed by the Xilinx ISE version 9.2i software system.

The time required by the Xilinx ISE 9.2i is proportional to the number of cubes in the function representation. This proved to be the limiting factor determining the choice of presented benchmark functions. For example, this Xilinx ISE 9.2i was capable to implement the 65 input function e64, specified with a set of 65 cubes. It was unable to implement Mul8, a 16-input multiplier specified with 28466 cubes. The cube representation of the Mul8 function was obtained using Espresso software. After more then two hours of processing the Xilinx ISE ran out of available memory. However, the system proposed in this thesis had no such limitations and successfully implemented both functions.

In both cases we have implemented the final design using the XC5VLX30 device from the Virtex-5 FPGA family. This particular device consists of 4800 slices with 30720 logic cells and 19200 CLB flip-flops. A single Virtex-5 CLB comprises two slices, with four 6-input LUTs and four flip-flops each, with total of eight LUTs and eight flip-flops per CLB.

Benchmark functions with an odd number of inputs were extended to an even number of levels by adding another input set to the constant zero, in order to permit the conversion to the quaternary decision diagram form. This is indicated in the tables with experimental results. Certain of the benchmark functions in the MCNC set have some of the cube outputs set as don't-care symbols. For each such function two new functions are generated, one where the don't-care symbol "-" is replaced with constant 0 and another where it is replaced with the constant 1. These functions are labeled with "_0" and "_1" suffixes added to their corresponding names. The obtained results are presented in Table 5.11.

Table 5.11 Standard synthesis method results.

Name	In	Out	Slices	LUTs	Av. Delay (s)	Max. Delay	Proc. Time (s)
5xp1_0	7+1	10	8	15	1.14	1.333	68.00
9sym	9+1	1	6	11	1.032	1.387	62.00
bw_0	5+1	28	9	11	1.098	1.436	74.00
con1	7+1	2	2	5	0.938	1.142	63.00
e64	65+1	65	101	134	2.686	3.320	73.00
inc_0	7+1	9	9	19	1.253	1.474	68.00
inc_1	7+1	9	9	20	1.232	1.562	63.00
rd53_0	5+1	3	3	3	0.937	1.159	60.00
rd73_0	7+1	3	7	10	1.01	1.184	57.00
vg2_0	25+1	8	6	9	1.049	1.286	65.00
vg2_1	25+1	8	2	4	0.951	1.426	60.00
xor5	7+1	1	1	1	0.845	1.010	59.00
misex1	8	7	7	14	1.108	1.459	15.11
rd84	8	4	8	12	1.234	1.429	13.23
sao2_0	10	4	9	28	1.093	1.437	29.09
sao2_1	10	4	5	4	0.994	1.453	4.99
sqrt8	8	4	5	7	1.012	1.439	8.01
duke2_1	22	1	4	8	0.988	1.578	87.00

The columns in this table represent the following values:

1. the number of inputs of the function,

2. the number of outputs of the function,

3. the number of slices on the FPGA device,

4. the number of LUTs used in final implementation,

5. average delay on 10 most critical paths,

6. total time needed for synthesis, mapping and routing.

In the second phase of the experiments, a quaternary decision diagram is created for each benchmark function. Again the Xilinx ISE software is used for technology mapping. In contrast to the previous phase of the experiments, Xilinx ISE software did not perform any synthesis or optimization. Synthesis and optimization of the logic circuit was performed during the process of generating the quaternary decision diagram.

The results of this phase of the experiments are presented in Table 5.12.

By examining the results from both phases of experiments we arrive at the following conclusions. In general, a standard industrial approach to synthesis, produces more compact implementations of benchmark functions. However, in some case, for example for E64, con1, inc_0, rd53_0 and 5xp1_0 functions the quaternary decision diagram methods produces better results. Processing times required by both methods are comparable.

It must be noted that Xilinx ISE is a highly optimized software package, which has been gradually improved through several versions over time. On the other hand, the method based on a quaternary decision diagram is not meant to be a separate, standalone solution for all logic design needs. Rather

Table 5.12 Results for synthesis using Quaternary decision diagrams.

Name	In	Out	Slices	LUTs	Av. Delay (s)	Max. Delay	Proc. Time (s)
5xp1_0	7+1	10	16	36	1.283	1.584	60.22
9sym	9+1	1	6	12	1.054	1.284	57.74
bw_0	5+1	28	20	41	1.375	1.994	60.07
con1	7+1	2	2	3	0.866	1.135	57.66
e64	65+1	65	70	114	2.313	4.093	117.45
inc_0	7+1	9	9	16	1.066	1.343	59.85
inc_1	7+1	9	10	17	1.176	1.396	58.99
rd53_0	5+1	3	5	8	1.023	1.238	57.66
rd73_0	7+1	3	8	17	1.132	1.277	57.93
vg2_0	25+1	8	175	405	2.63	3.449	105.56
vg2_1	25+1	8	9	19	1.234	1.376	69.48
xor5	7+1	1	1	2	0.817	1.163	59.86
misex1	8	7	12	22	1.196	1.434	41.00
rd84	8	4	8	27	1.11	1.320	39.00
sao2_0	10	4	29	64	1.505	1.321	97.00
sao2_1	10	4	2	4	0.961	1.203	10.00
sqrt8	8	4	8	18	1.091	1.402	30.00
duke2_1	22	1	17	41	1.086	1.377	59.00

it is intended to be a part of some large tool set similar to, for example, the Xilinx ISE. In that sense, the system proposed in this thesis represents a proof of concept, based on which more optimized design tools could be developed.

5.14 COMPARISON OF EFFICIENCY OF SHARED BINARY DECISION DIAGRAMS AND MULTI-TERMINAL BINARY DECISION DIAGRAMS

Multi-output switching functions are used to efficiently describe functionality of a large variety of circuit systems. Therefore, they are one of the most common types of discrete functions encountered in practice. In Section 2.4 we have already introduced two types of decision diagrams of special interest when dealing with those functions, Shared Binary Decision Diagrams (SBDDs) and Multi-Terminal Binary Decision Diagrams (MTBDDs). In general, the advantages of one model over the other depend on the properties of a particular function. However, there are examples in which the application of multi-terminal binary decision diagrams leads to more compact representation. This fact was the motivation for the introduction of multi-terminal decision diagrams, [30].

In this section we present a system capable of generating hardware models in VHDL of Boolean functions represented using MTBDDs. Using the proposed method we have produced a set of hardware designs of selected standard benchmark functions. We compare the complexity of these hardware designs with implementations of the same functions produced using shared decision diagrams.

We begin our discussion with a quick reminder of methods for representing multi-output functions using decision diagrams. Although both the MTBDDs and SBDDs, like all reduced ordered decision diagrams, are canonic representations of a given switching function, in some cases there is a significant difference in size and complexity between diagrams obtained in this way.

Example 5.23 *Consider a multi-output binary function $f = (f_0, f_1, f_3)$,*

x_1	x_2	f_0	f_1	f_3	f_z
0	0	0	0	1	0
0	1	0	1	0	2
1	0	1	1	1	7
1	1	0	0	1	1

We can represent this multi-output function in integer form, by summing each individual output f_i, multiplied by 2^i. Thus, we obtain $f_z = \sum_i 2^i f_i$. By doing so we have changed the underlying algebraic structure, and switched from a finite field of order 2 to the field of rational numbers. The Shannon expansion rule in the field of rational numbers has the following form: $f = \bar{x}_i f_0 + x_i f_1$. Note that, in this field, the Boolean operation XOR is replaced with arithmetic addition.

Fig. 5.10 shows a Multi-terminal binary decision diagram obtained by recursively applying the Shannon expansion to integer function f_z.

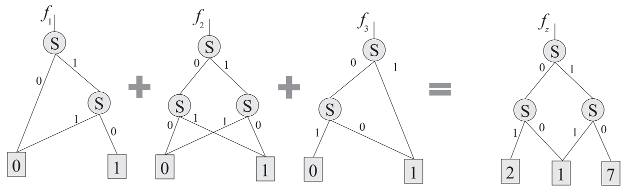


Fig. 5.10 A multi-output binary function represented by MTBDD.

This multi-terminal decision diagram consists of three non-terminal and three terminal nodes, producing the total size equal to six. An equivalent Shared binary decision diagram has five non-terminal and two terminal nodes, giving seven nodes of total size, as evident from Fig. 5.11.

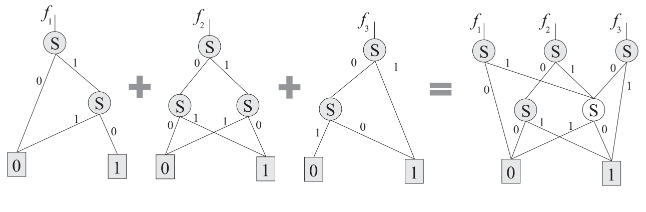


Fig. 5.11 A multi-output binary function represented by shared BDD.

Table 5.13 Size comparison of MTBDDs and Shared BDDs of some benchmark functions.

Function	In	Out	Cubes	MTBDD			SBDD	
				NTN	TN	size	NTN	size
5xp1	7	10	75	127	128	257	88	90
Add2	4	3	11	13	6	29	17	15
Apex4	9	19	438	442	319	761	1024	1026
Ex1010	10	10	1024	894	176	1070	1079	1081
Misex1	8	7	32	17	11	28	47	49
Rd53	5	3	32	15	6	21	23	25
Rd73	7	3	141	28	8	36	43	45
Rd84	8	4	256	36	8	44	59	61
Sao2	10	4	58	95	10	105	162	164

In Table 5.13 we present the comparison of sizes of multi-terminal binary decision diagrams and Shared binary decision diagrams for some functions taken from MCNC [181]. The columns in this table represent the following, In - number of input variables, Out - number of outputs, Cubes - number of cubes, NTN - number of non-terminal nodes in the corresponding decision diagram, TN - number of terminal nodes in the corresponding decision diagram, Size - total number of nodes (sum of the numbers of terminal and nonterminal nodes).

These results clearly demonstrate the benefit of using MTBDDs in some cases.

We focus on the integer equivalent of the Shannon expansion rule. Since the basic functionality of each node is the same, we can reuse one standard component. The final implementation will, thus, be a hierarchical structure of interconnected instances of this component. Interconnections will correspond to the edges in the binary decision diagram. The structure of the diagram, nodes, and edges that connect them, is directly mapped into the hardware level. This organization is intuitive and corresponds well to the native XML logic and the VHDL syntax.

5.14.1 Hardware Implementation of Multi-output Switching Functions Using MTBDD

The hardware structure presented in this section is similar to the one already described in Section 5.12 for binary decision diagrams. The main difference here lies in the structure of the node component. Since we now deal with integer-valued functions, some modifications of the original design are needed. These changes are reflected in the higher complexity of individual components.

In Fig. 5.12, we present a circuit implementing the Shannon expansion rule in integer form. Each component will still have two inputs and one output. The width of these inputs and outputs is determined by the particular platform-specific integer representation. This value can be specified as a parameter in the software system we propose.

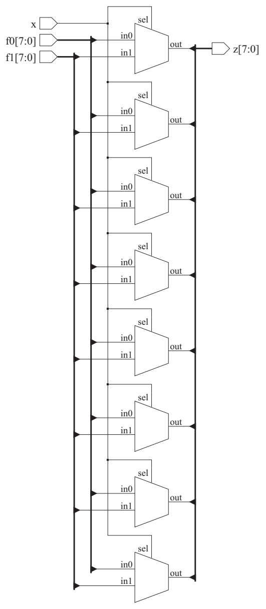


Fig. 5.12 Circuit implementation of the integer Shannon expansion rule.

The greater complexity of individual components is counterbalanced by a smaller total number of components needed to realize a multi-output switching function using MTBDDs.

As in the previous examples, we employ a set of XSLT templates to produce MTBDD hardware models based of XML decision diagram documents.

The first style sheet is responsible for the preprocessing of the original XML documents, removing redundant elements of the XML structure. This part of the conversion process is similar to the one described in previous sections. Therefore, we will not reiterate these details. In further discussions we focus only on the differences in the proposed methodology.

The second XSLT styles sheet produces the final output, VHDL hardware model, of the structure described in the previous sections.

Establishing a correspondence between the elements of the source XML hierarchy and VHDL hardware description is the crucial moment of the XSLT conversion process. The top level `<dd:tree>` element corresponds to the basic skeleton of the VHDL description, the VHDL hardware entity, and the associated architecture.

The port list of the VHDL entity is generated automatically based on the number of inputs and outputs of the original multi-output discrete function.

Input ports that correspond to logical variables of the functions are of the type `std_logic`. The outputs and inputs that correspond to integer constants are implemented as a `SIGNED` type, as specified in the `ieee.numeric_std` library. The width of these ports can be specified by the user. In our experiments we have used 32bit representations of integer numbers.

Each non-terminal `<dd:node>` represents one instance of the component presented in Fig. 5.12. As stated earlier, terminal nodes are equivalent to integer inputs of the hardware entity. Edges of the diagram are represented by the internal signals of the VHDL architecture.

Example 5.24 *We give an example of the VHDL code for the MISEX1 benchmark function:*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.numeric_std.all;

ENTITY DD_proba IS  PORT ( con20, ...  , con79 :  IN SIGNED (31 downto
0); clk, x0, ...  , x7 :  IN std_logic; final_out :  OUT SIGNED (31
downto 0));
END ENTITY DD_proba;

ARCHITECTURE DD_arch OF DD_proba IS
COMPONENT my_reg_arith_9
 PORT ( con_reg_in_20, ...  , con_reg_in_79 :  IN SIGNED (31 downto 0);
clk, reg_in_0, ...  , reg_in_7:  IN std_logic; con_reg_out_20, ...  ,
con_reg_out_79 :  OUT SIGNED (31 downto 0); reg_out_0, ...  , reg_out_7:
OUT std_logic);
END COMPONENT;
COMPONENT my_reg_out_arith  PORT (clk :  IN std_logic; my_in_out :  IN
SIGNED (31 downto 0); my_out_out :  OUT SIGNED (31 downto 0));
END COMPONENT;
COMPONENT my_arith
 PORT (f0, f1:  IN SIGNED (31 downto 0); x :  IN std_logic; z :  OUT
SIGNED (31 downto 0));
END COMPONENT;

SIGNAL x_tmp_0, ...  , x_tmp_7:  std_logic;
 SIGNAL con_tmp_20, ..., o14015 :  SIGNED (31 downto 0);
BEGIN
 in_reg :  my_reg_arith_9
  PORT MAP (con20, ...  , x_tmp_7);
 a14048 :  my_arith
  PORT MAP (o10777, o14037, x_tmp_0, o14048);
```

Table 5.14 The selected benchmark functions implemented using MTBDDs and Altera Cyclon II FPGA technology.

function	nets	inst.	max. freq. (Mhz)
5XP1	2304	1226	154.7
MISEX1	4627	2396	128.0
RD53	1452	860	195.5
RD73	1782	1184	139.3
RD84	2177	1480	124.7
SQRT8	6303	4166	119.6

```
   ...
  out_reg :  my_reg_out_arith
    PORT MAP(clk, o14048, final_out);

END DD_arch;
```

The selected benchmark MISEX1 is an 8 input function. This can be observed in the number of input ports declared in the main entity of the VHDL design. Signals in this device are treated as 32bit integer values.

5.14.2 Comparison with SBDDs

To provide some idea about the size and performance of actual hardware implementations of Multi-terminal decision diagrams, we present the results of the series of experiments. We have used multi-output switching functions from the MCNC benchmark set [181]. For each selected function a multi-terminal decision diagram was created. These diagrams were converted into valid XML documents. For this task, we used the software package presented in [162], in conjunction with an export module capable of producing the XML documents. We have used these XML documents as an input to the proposed system to produce a hardware descriptions in VHDL.

We made the final RTL synthesis of the produced VHDL code for Altera Cyclon II and Xilinx Spartan FPGA technology using the Mentor Graphic LeonardoSpectrum synthesis tool. We show the results of Multi-terminal decision diagrams based implementation for Altera Cyclon II device family in the Table 5.14.

Table 5.14 presents the number of inputs and outputs of each function, the maximal working frequency obtained by simulation, slack at the critical path and the number of accumulated instances, for a given FPGA architecture.

We compare these results to SBDD-based implementations of switching functions from the same dataset. Table 5.15 shows results for implementation on Altera Cyclon II platform.

The results of experiments regarding the MTBDD-based implementations using Xilinx Spartan FPGA technology are shown in Table 5.16.

Table 5.15 The selected benchmark functions implemented using SBDDs and Altera Cyclon II FPGA technology.

function	nets	inst.	max. freq. (Mhz)
5XP1	129	119	165.8
MISEX1	80	69	254.7
RD53	52	44	235.9
RD73	76	66	167.5
RD84	98	87	144.8
SQRT8	85	74	179.4

Table 5.16 The selected benchmark functions implemented using MTBDDs and Xilinx Virtex II FPGA technology.

function	nets	inst.	max. freq. (Mhz)
5XP1	2348	1270	108.2
MISEX1	4697	2464	81.2
RD53	1633	1041	120.3
RD73	2003	1405	93.3
RD84	2428	1731	85.6
SQRT8	7679	5542	80.0

Table 5.17 The selected benchmark functions implemented using SBDDs and Xilinx Spartan II FPGA technology.

function	nets	inst.	max. freq. (Mhz)
5XP1	166	159	136.5
MISEX1	97	87	157.1
RD53	56	48	169.8
RD73	89	79	105.8
RD84	117	107	100.2
SQRT8	92	83	115.4

In Table 5.17 we present the results for SBDD-based implementation using Xilinx Spartan FPGA technology.

From the data presented in Table 5.13 it is expected that MTBDD-based implementation would have an advantage over SBDD-based approach, at least for certain functions, due to significantly smaller number of non-terminal nodes. However, the results of these experiments reveal that this advantage is annulled by the greater complexity of individual nodes and their interconnections.

6

Entropy Estimation Using Decision Diagrams

The notion of entropy as introduced by Shannon in [6], [36], [149], represents one of the fundamental concepts in modern information theory and signal processing. In essence, entropy describes the amount of information carried by a signal. The applicability of this concept greatly transcends the field of information theory in a narrow sense. Although the mathematical definition of information entropy is well known, the task of calculating the entropy of a particular signal is by no means trivial.

If the signal is expressed in the form of a vector of characters taken from a specific alphabet, a good measure of the information entropy should satisfy the following criteria:

1. The Measure should be continuous.

2. If probabilities of occurrence for all the characters of the alphabet are equal, the entropy should be 1.

3. If the probability of a certain character is one, the entropy should be 0.

4. The amount of entropy should be equal, independently of how the process is divided.

According to the formula, $-\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$, introduced by Shannon in [149], an entropy is calculated based on the probability distribution of the characters in the vector. A reliable estimate of this probability distribution often represents the most difficult part of the entropy calculation. Over the years, a variety of methods has been proposed for the estimation of either the probability distribution or the information entropy directly.

The formula introduced by Astola and Ryabko in [8] provides a good estimate of the entropy of a given signal. The calculation required for this method can be time consuming. Furthermore, the complexity of this calculation depends strongly on the length of the given sequence. In this thesis we will present a method for improving the speed and reducing the complexity of this calculation by exploiting the properties of binary decision diagrams.

If characters of the alphabet are represented as discrete numerical values, the signal output of the source can be seen as a discrete function.

We introduce a method for the estimation of the entropy of binary streams based on binary decision diagrams. By exploiting the useful properties of binary decision diagrams we may be able, in some cases, to significantly reduce the time and complexity of calculation of entropy estimates. This method was originally presented in [156].

6.1 CALCULATION OF ENTROPY ESTIMATES USING BDDS

In [8], Astola and Ryabko introduced the following method for calculating the estimate of the entropy of a given vector.

Consider an alphabet A and let $A^* = \bigcup_{i=1}^{\infty}$ be the set of all finite words over A. Let $f = f_1 f_2 \dots f_t$ be a given vector of length t, and $v = v_1 \dots v_k$ be a possible subvector of f, $f, v \in A^*$. Denote the rate of a subvector v occurring in the vector x as $\nu_f(v)$. For example, if $f = 000100$ and $v = 00$, then $\nu_f(00) = 3$, since, we calculate as follows $(00)0100 \rightarrow 0(00)100 \rightarrow 00(01)00$ $\rightarrow 000(10)0 \rightarrow 0001(00)$, and the sequence $v = 00$ appears three times.

For any $0 \leq k \leq t$ the empirical Shannon entropy of the order k is defined as follows:

$$h_k^* = -\sum_{v \in A^k} \frac{\bar{\nu}_f(v)}{(t-k)} \sum_{a \in A} \frac{\nu_f(va)}{\bar{\nu}_f(v)} \log \frac{\nu_f(va)}{\bar{\nu}_f(v)}. \tag{6.1}$$

In order to calculate the entropy estimate by means of (6.1) we need to determine the number of occurrences of all the possible subvectors v of the length k in the given vector. This is the most computationally intensive part of the process. It can be implemented using a straightforward method, by moving a window of length k over a given vector and increasing the appropriate counter for each encountered subvector. In a way this is a brute force method. It does not take into account the individual properties of a particular vector. Assume, for example, that the second half of a vector is equal to the first half of the vector. If we take into account this information, we could reduce the amount of needed calculation by half. The application of decision diagrams permits us to do this.

In this thesis we focus only on the calculation of the entropy estimate for binary vectors. For the sake of simplicity, we demonstrate the proposed method first for the simplest case of $k = 2$. We also assume that given vectors are of the length $t = 2^n$, so that binary decision diagrams can be constructed.

We begin with the following observation. Let $f_{s1} = f_1 f_2 ... f_{\frac{t}{2}}$ and $f_{s2} = f_{\frac{t}{2}} f_{\frac{t}{2}+1} ... f_t$ be the first and second half of the given vector. It naturally follows that

$$\nu_f(v) = \nu_{f_{s1}}(v) + \nu_{f_{s1}}(v) + c, \tag{6.2}$$

which is the number of occurrences of a certain subvector of length $k = 2$ in a given binary vector, is equal to the number of occurrences of this subvector in the first half of the vector, plus the number of occurrences of this subvector in the second half of the given vector. Minor corrections of plus one should be made if the given subvector occurs exactly at the split of two halves of the given vector so that its first character remains in the first half of the vector while the last ends up in the second half

$$c = 1 \Leftrightarrow v_1 = f_{\frac{t}{2}}, v_2 = f_{\frac{t}{2}+1}, \tag{6.3}$$

$$c = 0, \text{otherwise.} \tag{6.4}$$

We can recursively apply this observation to each half of the vector, splitting them further into shorter and shorter segments, until we reach the segments of the length 2. At this point, it is trivial to calculate of occurrence of a subvector. If the given subvector v is identical to a particular segment f_{sn}, then $\nu_f(v)$ needs to be incremented by one. We must keep in mind that the modifications of the calculated number every time the considered subvector occurs at the split between the examined segments of the original vector, potentially at every step of the recursion.

This recursive procedure resembles the structure of a binary decision tree. If we mark the levels of a binary decision diagram from 0 for the topmost level containing the root node to $n+1$ for the level of terminal nodes, then the nodes of the second lowest level n in the diagram correspond to subvectors of length 2 in the underlying binary vector. From the properties of decision diagrams, it follows that the exact of number of times a certain subvector v occurs in a given vector f is equivalent to the number of paths that point to the node which is the root of the subdiagram representing the subvector v. It can be shown that this number is equal to the weighted sum of incoming edges to the node. The weight associated with a particular edge equals $w_i = 2^l$, where l is the difference of the level between the parent and the node in question.

Let $F = [0110101010101111]$ be a given binary vector of the order $n = 4$. In Fig. 6.1 we present this vector by a binary decision diagram and the results of the calculation of the number of occurrences of individual subvectors.

To calculate the number of occurrences of all the possible subvectors of length $k = 2$ in a binary vector, we need to iterate through all the nodes at the n-th level of a decision diagram and calculate the number of the corresponding paths as a described weighted sum. Before we can perform any calculations,

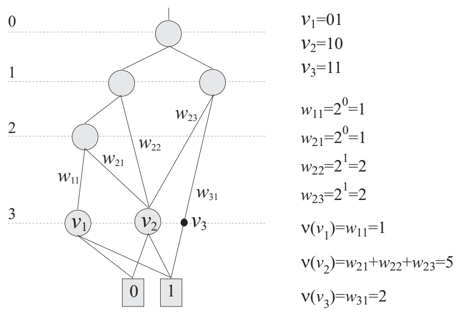


Fig. 6.1 Example of a calculation of entropy estimate via binary decision diagrams.

Table 6.1 LUT of indices for $k = 2$.

	00	01	10	11
00	00	00	01	01
01	10	10	11	11
10	00	00	01	01
11	10	10	11	11

we need to extend the diagram with virtual nodes at each place where an edge intersects with the n-th level. These virtual nodes will correspond to subvectors representing pairs of identical binary values $v = 00, v = 11$, which possibly exist in the original vector f.

However, this represents only a partial estimate of entropy, since it does not take into account the subvectors that occur on the splits between the segments of the vector. In total there are $2^n - 1$ of these subvectors, for a vector of the length 2^n.

We can show that a binary decision diagram contains all the information necessary to completely calculate the occurrence rate of subvectors of the order $k = 2$. For a given binary vector, a subvector of the length $k = 2$ starting at the position i is determined by the last character of the preceding vector at the $i - 1$ position, and the first character of the following vector at the $i + 1$ position. Consider an example ...1100..., and let $v_{i-1} = 11$ and $v_{i+1} = 00$. It clearly follows that $v_i = 10$. The complete set of these relations can be expressed in tabular form, see Table 6.1. This lookup table is identical for all binary decision diagrams and needs to be generated only once at the beginning of the process.

Entries of this table can be replaced by the integers using encoding $00 = 0$, $01 = 1$, $10 = 2$, and $11 = 3$. Thus, Table 1 can be concisely expressed by the function $f = \overline{x}_2x_3 + 2x_2\overline{x}_3 + 3x_2x_3$, where x_2, and x_3 are Boolean variables 0 and 1 interpreted as integers 0 and 1. By choosing between this analytical expression and the tabular representation we are permitted to trade between temporal complexity and memory requirements.

If the subvectors v_{i-1} and v_{i+1} are represented by adjacent nodes at the $(n-1)$ level in the decision diagram, subvector v_i will correspond to their common parent node. In Fig. 6.2 we demonstrate the correspondence between subvectors of the previous example and the nodes of the decision tree. This decision tree can be reduced to the decision diagram in Fig. 6.1 as shown in Fig. 6.3.

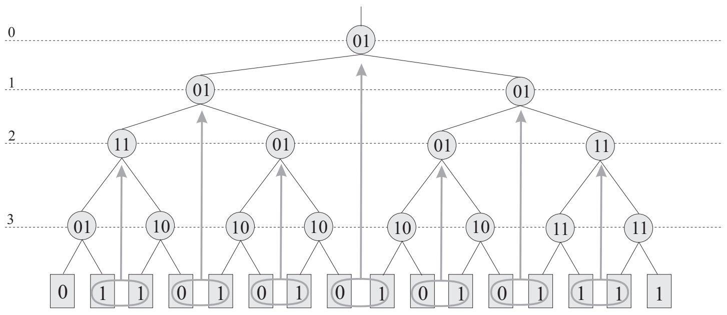


Fig. 6.2 Subvectors of order $k = 2$ over nodes of a decision tree.

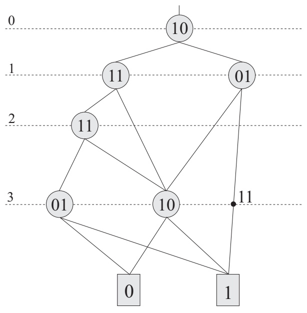


Fig. 6.3 Subvectors of the order $k = 2$ over nodes of a reduced ordered decision diagram.

Fig. 6.4 shows the order in which subvectors for the $k = 2$ case are read during the traversal through the decision diagram. For the sake of clarity we present this over a decision tree.

It is evident that in order to completely determine the occurrence rates of all of the subvectors of a given vector we need to visit all the nodes in the diagram. The complete algorithm, therefore, represents an inorder traversal

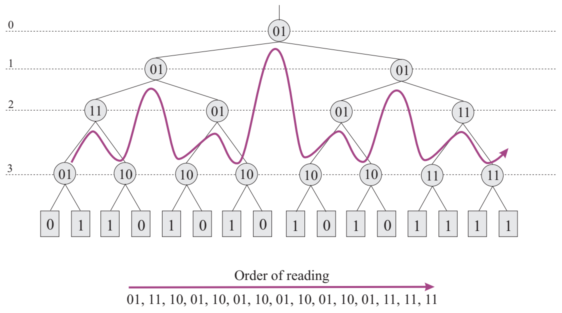


Fig. 6.4 The order of reading of the subvectors for $k = 2$, presented on a decision tree.

of the decision diagram, with the following additional steps performed at each node:

If the level of the current node is $< n + 1$,

1. Determine to which subvector the node corresponds,

2. Increase the value of appropriate counter by the weight of the corresponding incoming edge,

3. Store the index of the counter.

If the level of the current node is $> n + 1$,

1. Based on indices stored in children nodes, determine the preceding and following subvector,

2. Increase the appropriate counter.

After the complete traversal of the tree we obtain the final results, $\nu(v_1) = \nu(01) = 5 + 1 = 6$, $\nu(v_2) = \nu(10) = 0 + 5 = 5$, $\nu(v_1) = \nu(11) = 2 + 2 = 4$.

Finally, we can compare these values with values obtained by applying of the original method. For the original vector $F = [0110101010101111]$ after we apply the sliding window we obtain 01, 11, 10, 01, 10, 01, 10, 01, 10, 01, 10, 01, 11, 11, 11, that is, $\nu(v_1) = \nu(01) = 6$, $\nu(v_2) = \nu(10) = 5$, $\nu(v_1) = \nu(11) = 4$. It is evident that both methods have produced identical results. The rates of occurrence of subvectors can then directly be used in (6) to calculate the final entropy estimate.

The complexity of this algorithm is proportional to the number of nodes in the decision diagram, versus the length of the given vector which was the complexity of the standard approach. The same argument about efficiency of decision diagrams in general can be applied to this algorithm. For further details, please refer to [17], [32], [119], [182].

Table 6.2 Number of steps needed to calculate the occurrence rate of subvectors using BDDs, and the standard approach.

function	BDD	Standard Approach
BW 0	10	31
BW 23	16	31
5XP 10	12	127
MISEX1 0	66	255
RD84 0	4	255
9SYM 0	6	511
APEX4 0	258	511
APEX4 10	6	511
APEX4 17	34	511
CLIP 0	10	511
EX1010 0	10	1023
MISEX3 0	8194	16383
MISEX3 3	514	16383
MISEX3 4	4	16383
MISEX3 7	130	16383

The described method can easily be generalized for an arbitrary length of subvectors of the form $k = 2^j$.

Similar as in the previous example, for the case $k = 4$, $j = 2$, consider a subvector $v_i = 1101$, and its neighboring subvector $v_{i+4} = 1001$. After a simple observation we determine that there are additional subvectors spanning a split between v_i and v_{i+4}, namely $v_{i+1} = 1011$, $v_{i+1} = 0110$, $v_{i+1} = 1100$. This fact will be reflected in the structure and size of the lookup table of indices needed for the process.

For the general case $k = 2^j$, LUT will have $2^j x 2^j$ cells where we need to store $k-1$ index values, thus adding to the memory complexity of the overall algorithm. As this table needs to be stored only once for all the calculations of entropy estimates for the same k, in real application for a reasonable length of subvectors this overhead will not be significant.

6.2 EXPERIMENTAL RESULTS

To demonstrate the validity of the proposed method, we have conducted a set of experiments. We have counted the number of steps needed to calculate the rate of occurrence of subvectors in a given vector using both the standard and the BDD based approach. In these experiments the (MCNC) set of benchmark functions was used [181]. Table 6.2 presents the selected results. The first column in the table represents the number of steps needed for the BDD based method, and the second represents the results for the standard approach.

The number of steps in the standard approach is determined by the length of the original vector, while in the BDD-based approach it is determined by the number of nodes. The example of function misex3, and its particular outputs 0, 3, 4, and 7, illustrates that the approach using BDD, depends on the number of nodes, unlike the standard approach, where the complexity remains the same for all the outputs. The same conclusions can be made for the function APEX and other entries in this table.

We did not take into account the number of steps needed to construct the decision diagram, since the basic assumption was that the functions were already represented in decision diagram form.

7 Visualization of Decision Diagrams Using SVG

In this chapter we demonstrate how an abstract data structure can be converted into a human understandable form. We demonstrate the flexibility of the proposed system by showing how it can be used to automaticaly generate a graphic representation for a given decision diagram. The final result of this process is a vector image in SVG format. In order to achieve this, we make use of XSLT scripts similar to ones used in previous sections. We have chosen the SVG formate because it represents a pure XML derivate. The implementation of XSLT conversion scripts is, therefore, relatively simple and straightforward. The method presented in this chapter was originally proposed in [160].

7.1 SVG GRAPHICS DESCRIPTION LANGUAGE

SVG language is an XML derivate for description of 2D vector graphics both static and animated. It is, thus, both descriptive and scripted in nature. It is an open standard introduced and sponsored by the W3 Consortium [146]. SVG confirms to the specification of the general XML standard producing well-formed and valid XML documents, as specified in [60]. SVG documents can easily be processed with standard XML parsing tools or easily transformed by means of XSLT. In this work, we focus on the descriptive aspect of SVG since we are dealing only with static graphics.

SVG is an emerging standard and aimed primarily at displaying vector graphics on the Web. Its wider acceptance is still expected in the future. It is, however, at the moment supported on all of the relevant computer platforms and by all web browsers either natively (Firefox 1.5 and newer) or in the form of a plug-in (Microsoft Internet Explorer). Most of the standard software packages for processing 2D vector graphics available on the market, including Adobe Illustrator and Corel DRAW, support data exchange through SVG as well. Illustrations produced using SVG can easily be displayed on the web or embedded in to TeX documents with minimal additional processing.

Furthermore, since SVG complies totaly with the XML specification, data stored in this format can be combined with data stored in other XML derived formats in a single document. A user could produce a single file containing an XML decision diagram and its graphical representation in SVG format.

Further information about SVG graphics format can be found in [57] and [176]. Principles of development of software systems that make use of SVG are explained in [24].

7.2 VIZUALISATION OF DECISION DIAGRAMS

Although the actual visualization of the topology of decision diagrams may vary, there exists one commonly accepted way of representation accepted by most researchers working in this field. In Fig. 7.1, we show an example of a BDD [17], for the function $f(x_1, x_2, x_3) = x_1\bar{x}_2 \vee x_2x_3$.

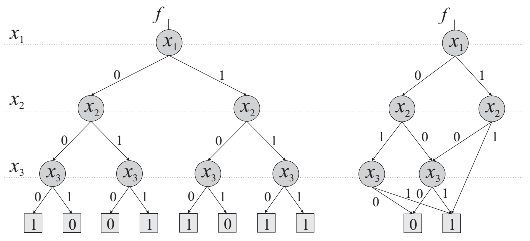


Fig. 7.1 Binary Decision Diagram.

By convention, non-terminal nodes are represented by circles either empty or carrying the textual label that signifies the applied decomposition rule. Terminal nodes are represented by squares or rectangles containing a logical constant. Edges are naturally represented by lines connecting the nodes. As an option, levels of decision diagrams can be also represented. These general principles of vizualisation can easily be expressed in SVG terms.

All valid SVG documents share the same basic structure. The proposed system will need to repeat this structure in every SVG document it produces. As is the case with all XML based data description languages, an SVG document is a structured text with a hierarchy of elements. The topmost element in this hierarchy is the `<svg:svg>` element, which represents a container of actual graphical elements in the document. This element can have various attributes describing the properties of the document associated with it. For the purpose of this demonstration, we specify only the width and height of the document page. Actual implementation can have more precise control over the document, i.e., display and printing properties.

The list of graphic elements supported by SVG includes the usual primitive elements found in other vector graphic software packages. We make use of a very reduced set of these elements using only `<svg:rect>` elements representing rectangles for terminal nodes, `<svg:ellipse>` to represent non-terminal nodes and `<svg:line>` elements for the edges in the diagram. Graphical elements in one document can be grouped using `<svg:g>` container elements. Groups can contain other groups, thus creating a convenient hierarchy of elements. Labels on the graphical representation are implemented using appropriate `<svg:text>` elements. This SVG element permits us to specify the size and font of its inscription. By convention, we use TimesRoman font, widely available on all platforms.

These elements represent basic building blocks out of which the proposed system builds final documents. We present the example of the code of an SVG document containing some of these basic elements:

```
...header of the document...
<svg:svg width='21cm' height='15cm' version='1.1' ...>

...graphic elements...
<svg:g id='node0' text-align='center'>

...Ellipse representing a non-terminal node...
<svg:ellipse cx='10.5cm' cy='1.75cm' rx='0.5cm' ry='0.5cm'
stroke='rgb(0,0,0)' fill='rgb(240,240,240)' stroke-width='1' />

...Line element representing an edge...
<svg:line x1='10.5cm' y1='2.25cm' x2='5.25cm' y2='4.75cm'
style='fill:none;stroke:rgb(128,128,128);stroke-width:1' />

...Label as a text element...
<svg:text x='7.975cm' y='3.4cm' font-family='Times Roman'
font-size='12' fill='black'>0</svg:text>
</svg:g>

</svg:svg>
```

We begin the transformation process by applying a preprocessing XSLT script to produce a more suitable version of the XML document. The preprocessing script will remove all the recursive aspects of the XML document. It will also calculate the relative coordinates of graphic elements that will form the graph representation.

As the first step, the preprocessing diagram creates a wrapper element named `<diagram>` and assigns some arbitrary values for the width and height of future SVG document.

This script produces a list of decision diagram nodes, `<node>` elements without any nested elements. The position of a graphic representation for each node is recalculated based on the number of nodes at each level and on the total number of levels in the diagram. The script assumes some abstract dimensions for the final plot and arranges nodes equidistantly from the central vertical axis of the diagram recalculating relative horizontal distance for each level. These coordinates are stored in the attributes of node elements on the list.

Each node contains a simple list of its descendants, `<edge>` elements. The attributes indicating the ID, level and expansion rules are preserved. The information about the parents of the node that might be present in the original XML document is discarded as it is not needed for graphical representation. Edges connecting the nodes will be drawn based on the information from the lists of children of each node. The distinction between terminal and non-terminal nodes is marked by the value of the `'type'` attribute associated with each node element.

The preprocessing script produces a list of `<line>` elements that will form the grid of level markers in the final graphical representation. Their vertical positions are calculated according to the number of levels in the diagram.

Final conversion from the intermediary XML document to the SVG document is done by a separate XSLT style sheet. This style sheet performs simple template matching, replacing the elements of the XML document with the appropriate geometric representation using the precompiled coordinates stored in the intermediary XML document.

This XSLT style sheet consists of a series of templates describing the correspondence between elements of the intermediary XML document and the SVG graphic elements.

At the beginning of the conversion process, XSLT will evoke the first template that will identify the `<diagram>` element and based on the information on the width and the height produce `<svg:svg width='21cm' height='15cm'>` element of the SVG document, thus creating a container for the rest of the graphic elements that will be produced.

The grid of the diagram is generated by a separate template in the XSLT style sheet based on the vertical coordinates stored in the `<line>` elements of the intermediary XML document. A dashed line is added to the SVG document for each level with non-terminal nodes of the original decision diagram. An appropriate label, marking the level, is attached to every line in the form of an `<svg:text>` element.

The XSLT script proceeds then to process the nodes of the decision diagram starting from the root node. A separate template in the XSLT style sheet will be activated for every `<node>` element in the intermediary XML code. Depending on the value of the `'type'` attribute, this template will create either a `<svg:ellipse>` representing a circle for a non-terminal node or a `<svg:rectangle>` representing a rectangle for a terminal node. It also produces the text element `<svg:text>` with an appropriate label for the given node. These graphic elements will be placed on the coordinates calculated by the preprocessing XSLT script.

For each `<edge>` element, a member of the list of children of the given node, another template is activated. This template draws a line representing the edge or the diagram by producing a `<svg:line>` element in the SVG document. Start coordinates of the line are read from the attributes of the node element. Based on the value of the `'point'` attribute of the `<edge>` element, the template identifies the target node and reads the end coordinates of the line from its attributes.

Graphical elements representing the node, its label and the outgoing edges, will be grouped together using a `<svg:g>` element for easier manual manipulation with other 2D vector graphics software.

We present the full code of the generated SVG document for a binary decision diagram from the previous example in Appendix C.

This set of XSLT style sheets produces the basic form of decision diagrams that can be modified in order to represent various classes of diagrams including, binary, multiple valued, Kronceker or pseudo Kronecker diagrams. These scripts can easily be customized both in terms of graphic properties or by the functionality for different applications.

7.3 EXAMPLES

This set of XSLT style sheets produces the basic forms of decision diagrams suitable for representation of various classes of diagrams. These scripts can easily be customized both in terms of graphic properties or functionality for different applications.

In Fig. 7.2 we give a graphic representation of a binary decision diagram for the $9sym$ function taken from the MCNC set of benchmarks for logic design [181], produced using the XML framework.

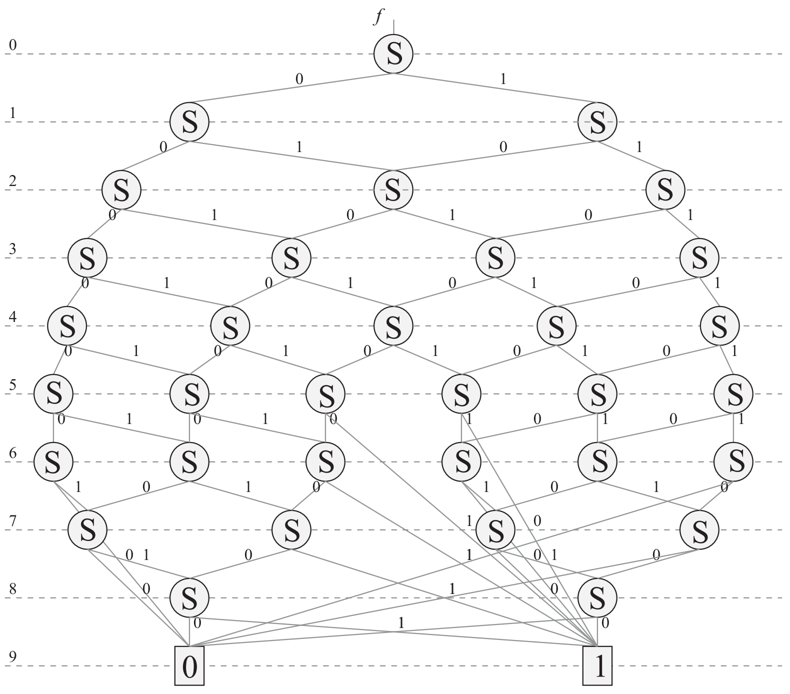


Fig. 7.2 A graphic representation of a binary decision diagram of a 9sym benchmark function.

8 Automatic Code Generation Using the XML Framework

In most of the examples presented in the previous sections we have focused on structural hardware models. However, in section 5.9, while discussing the basic properties of a VHDL data description language we have briefly mentioned the possibility of other approaches to hardware design. Namely, the essential difference between structural versus behavioral models is one of the fundamental topics in current hardware design. The same discussion applies not only to hardware design, but also to system modeling in a wider sense.

In this chapter, we examine the position of decision diagrams, as a form of representation of discrete functions, in relation to behavioral and structural system modeling approaches. We discuss this problem by examining the relations of Shannon binary decision diagrams and branching programs.

We introduce an extension to the basic XML based framework capable of converting a decision diagram represented in XML form into a branching program in some high-level programming language such as C/C++. For this task, we use a mechanism, based on XSLT, similar in structure to the ones from previous sections. The results presented in this chapter were originally published in [157].

8.1 BRANCHING PROGRAMS AND DECISION DIAGRAMS

We begin our discussion by focusing again on the so-called reduced order binary decision diagrams. There is clear and intuitive correspondence between the Shannon decomposition, as defined by (2.5), and a well known *If-Then-Else* programming structure. The first term of the resulting decomposition, the one obtained for value $x_i = 1$, corresponds to the Then branch of the If-Then-Else structure. The *Else* branch is equivalent to the second term of the decomposition, $x_i = 0$. By applying this decomposition rule recursively we obtain a decomposition tree.

A decomposition diagram obtained by the complete reduction of this tree corresponds to the optimal branching program that implements the given logic function $f(x_1, x_2, ..., x_n)$.

Therefore, we can say that a binary decomposition diagram is a graph-like data structure which describes the behavior of a certain system, not by data stored in its constituent components, since all of the nodes represent an identical operation, but by its topology. By extension, the same statement holds for any other type of decision diagram.

The fact that we are able to make a smooth transition between a structural and behavioral model, highlights an important duality inherent to decision diagrams with significant implications on hardware design.

The correspondence between decision diagrams and branching programs has been discussed in great detail in [7] and [91].

Decision diagrams as a starting point for code generation have been used in several applications. One interesting example of such an application in the field of signal processing can be found in [136]. In this example, the value of each individual image pixel is determined by the value of its immediate neighborhood. The values of surrounding pixels define the function for the calculation of its value. ROBDDs are used to efficiently in terms of time produce a C++ code that performs this calculation in each individual case.

8.2 THE XSLT CONVERSION MECHANISM

The process of generatiing a branching programs in C/C++ syntax, based on the given decision diagrams, has several stages. Our system performs this operation by applying a series of XSLT templates to the elements of the original XML document. The final output is a new document containing the code in C/C++. In order to achieve the optimal organization of the operation, these XSLT templates are grouped into three separate style sheets. The first two of them perform preprocessing, modifying the structure of the original document. During this process a series of intermediary XML documents is produced. Only the last XSLT style sheet performs the actual mapping of XML elements onto C/C++ syntax.

The recursive organization of XML documents corresponds very well with the structure of a code dictated by properties of common high-level programming languages. XML documents are, as stated earlier, hierarchically structured. Each `<dd:node>` element contains in itself, other nested elements of the same type, and nodes that belong are situated in its descending subdiagrams.

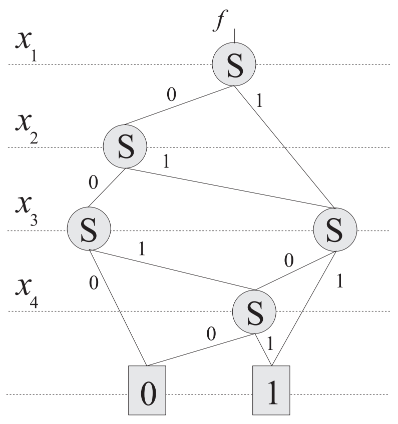
Every such element corresponds with one basic If-Then-Else structure. In the final output, the `<dd:node>` element is replaced with the following C/C++ code segment:

```
if(x1) then
{
  ...
}
else
{
  ...
}
```

This replacement process is repeated recursively for the descendants of the given node. The resulting C/C++ code will be nested in the appropriate branches of the If-Then-Else structure.

8.3 EXAMPLE

In Fig 8.1, we present the example of the entire branching code corresponding to the Boolean function $f(x_1, x_2, ..., x_n)$. We can note that the structure of the code follows closely the structure of the binary decision diagram generated for the same function.



```
bool example1(x1, x2, x3, x4){
 if (x1==1){
  if(x3==1){return true;}
  else{
   if(x4==1){return true;}
   else{return false;}
  }
 }
 else{
  if(x2==1){
   if(x3==1){return true;}
   else{
    if(x4==1){return true;}
    else{return false;}
   }
  }
  else{
   if(x3==1){
    if(x4==1){return true;}
    else{return true}
   }
   else{return false;}
 }}}
```

Fig. 8.1 BDD and a branching program corresponding to the function $f(x_1, x_2, x_3) = x_1\bar{x}_2 \vee x_2x_3$.

8.4 GENERALIZATIONS

The generalization that can be made to the method we presented in this Thesis is threefold.

So far we have limited ourselves to examining just the binary decision diagrams. The binary case is the most simple example of a larger theory and, thus, easiest to follow. However, the proposed system is capable of generating a C/C++ code even for non-binary, e.g., ternary or quaternary decision diagrams. There are no significant methodological differences in the way an XSLT mechanism works in a non-binary case. The main difference is that the `<dd:node>` XML element corresponds to the *Switch-Case* instead of the *If-Then-Else* structure.

The second generalization direction is aimed at classes of decision diagrams derived using decision rules other than the Shannon decomposition. In this case the standard If-Then-Else branching structures must be replaced with procedures performing operations equivalent to each particular decomposition rule.

Both of these modifications can easily be combined as in, for example, the case of Pseudo-Kronecker decision diagrams. The resulting program would be a semi-recursive hierarchy of Boolean or arithmetic operations.

Finally, since all of the needed programming structures, If-Then-Else, Switch-Case, recursiveness, etc., are present in the great majority of programming languages, and since the underlying algorithm remains unchanged, the proposed system can be adapted to produce its output in the syntax of any formal language that satisfies these basic criteria.

9 Conclusions

Graph-like structures are an efficient method of organizing data, especially data with complex interdependences. Decision diagrams, formally acyclic directed graphs, have risen to prominence as a tool for the representation of various classes of discrete functions. Different types of decision diagrams have been introduced over time for the representation of different specific function classes. They have found numerous applications in different fields, such as logic design and signal processing. Consequently a variety of software packages which employ decision diagrams in some way have been developed. However, most of these packages use proprietary file formats for data storage. To our knowledge there has been no comprehensive attempt to create a standardized format of the representation of decision diagrams.

Our primary aim has been to establish a uniform way to represent various classes of decision diagrams to facilitate easy data exchange between various existing software platforms. The proposed solution had to be general enough to be able to describe as many classes of decision diagrams already in use as possible. It also had to be flexible enough to be easily adapted for different applications. Ease of implementation was also a goal.

As a basis for our framework we have chosen XML, a general purpose data description language designed especially for the task of representing data with complex internal structures. The XML family, XML in narrow sense and its derivatives, offers a set of tools for easy and efficient, in terms of computational complexity, data manipulation.

Using an XML Schema mechanism we have developed a format specification for XML documents representing decision diagrams. This format specification focuses on common structural features which all classes of decision diagrams share. It also provides facilities for the storage of additional data. It is, therefore, capable of representing a wide variety of decision diagram classes.

The proposed schema is published online and XML documents that contain decision diagrams can be validated against it to ensure that they conform to the proposed specifications.

Using this XML schema, we have produced and presented the examples of XML documents for several common classes of decision diagrams, such as ROBDDs, FDDs, KDDs, PKDDs, TDDs, QDDs, MTBDDs, SBDDs, etc.

Parsers, software libraries for XML processing, are present on all platforms and development environments. They represent a very efficient software architectures, in terms of memory usage and processing time. Certain overhead in size is present in conjunction with the textual nature of an XML representation but is, in our opinion, counterbalanced by other good properties of the proposed system.

A platform built on the XML basis is highly extensible. New data can be easily included in the documents. If confronted with an extended document which they are not designed to process, software modules will process the data that is recognizable to them and ignore the additional data. The whole document will not be rejected. Communication between disparate software solutions can be established in this way. A scenario in which several software agents interact and each makes use of only a part of the data present in the same document is, thus, possible.

In the second part of the thesis, we have turned our attention to possible applications of the proposed XML-based framework. We have focused first on the application of decision diagrams in logic design.

The XML documents represent an abstract form of representation which needs to be converted into some application specific format. This conversion is achieved using XSLT data transformation language. We have developed a special set of XSLT templates for each example presented in this Thesis.

The first example we presented focuses on implementing switching functions on FPGA devices. In this example, the functions are represented using ROBDDs with Shannon decomposition, Kronecker and Pseudo Kronecker DDs. We have demonstrated how XML documents containing these decision diagrams can be converted into a hardware description expressed using a VHDL hardware modeling language.

We explored the efficiency of implementing multi output switching functions using MTBDDs vs. the more common approach based on SBDDs. Using the similar XML based mechanism we have demonstrated that for certain examples MTBDDs offer more compact functional representations. The higher complexity of nodes and interconnections is counterweighted by the smaller size of the diagram.

Implementation of switching functions using FPGA devices was also examined. The question of optimal granularity of logic blocks in FPGA devices was explored earlier in the literature. Six-input LUT-based FPGA devices have been shown to be an optimal solution in the trade-off between the size of the logic blocks and the complexity of the interconnections. QDDs have been proposed earlier as a tool for logic design regarding these devices. We have demonstrated how the proposed XML-based framework can be used to automatically generate LUT-6 implementation of switching functions using QDDs. The proposed method generates a QDD representation of the function from an already existing ROBDD. It then generates a hardware description in the form of a netlist using EDIF syntax, ready for direct technology mapping.

In the following part of the thesis we demonstrated how decision diagrams and the proposed framework can be used for applications in information theory. We presented a BDD-based method for calculating the entropy estimate of a given binary vector. The method is divide and conquer in nature and its complexity is proportional to the number of nodes in the corresponding ROBDD.

The flexibility of the proposed XML framework is demonstrated again with the example of the automatic generation of graphic representation of decision diagrams. We provided a set of XSLT style sheets that convert XML decision diagram documents into vector images in SVG file format. The images produced in this way can easily be edited by any established vector graphics software package. They can also be imported into various textual documents.

Finally, we discussed the behavioral and structural modeling dichotomy inherent in decision diagrams. A decision diagram is essentially a data structure. However, a decision diagram is a canonic representation of a function, and a function can be seen as the model of the behavior of a certain system. This duality associated with decision diagrams is best seen in the one to one relationship between decision diagrams and the flow control programming structures. We introduced a set of XSLT scripts which convert a given ROBDD into a branching program in C++ syntax. Those XSLT scripts can easily be adapted for other high-level programming languages, and several other types of decision diagrams.

We believe that these examples demonstrate that the proposed XML framework represents a useful addition to the tool set available for people working with decision diagrams. We hope that others will decide to use it and continue to build upon it and extend it further.

As for our future work, we intend to continue to seek out new application possibilities for decision diagrams and the XML-based framework alike.

References

1. M. M. Abramovici, A. Breuer, A. Friedman, *Digital Systems Testing and Testable Design*, Wiley-IEEE, Press, New York, USA, 1994.

2. S. Agaian, J. Astola, K. Egiazerian, *Binary Polynomial Transforms and Nonlinear Digital Filters*, Marcel Dekker, New York, United States, 1995.

3. E. Ahmed, J. Rose, "The effect of LUT and cluster size on deep submicron FPGA performance and density", *IEEE Trans. Very Large Signal Integration (VLSI) Systems*, Vol. 12, No. 3, 2004, 288-298.

4. S. B. Akers, "Binary decision diagrams" IEEE Trans. Comput. vol. C-27, 509-516, June 1978.

5. S. B. Akers, "Functional testing with binary decision diagrams", in Proc. 8th Ann. IEEE Conf. Fault-Tolerant Comput., 75-82, 1978.

6. R. B. Ash, *Information Theory*, John Wiley & Sons, 1967.

7. P. Ashar and S. Malik, "Fast functional simulation using branching programs", *IEEE Int. Conf. on Computer-Aided Design, ICCAD 1995*, San Jose, Califormia, USA, November 5-9, 1995, 408-412.

8. J. Astola, B. Ryabko, "Universal codes as a basis for time series testing", *Statistical Methodology*, Elsevier, Vo. 3, No. 4, 2006, 375-397.

9. J. Astola, R. S. Stanković, *Fundamentals of Switching Theory and Logic Design, A Hands on Approach*, Springer, 2006.

10. H. M. Babu, T. Sasao, “Representations of multiple-output switching functions using multiple-valued pseudo-Kronecker decision diagrams”, *Proc. 30th Int. Symp. on Multiple-Valued Logic*, Portland, Oregon, USA, May 23-25, 2000, 147-152.

11. P. Banerjee, “An overview of a compiler for mapping MATLAB programs onto FPGA”, *Proc. Asia and South Pacific Design Automation Conf., ASP-DAC 2003*, Kitakyushu, Japan, January 21-24, 2003, 477-482.

12. C. Binstock, D. Peterson, M. Smith, M. Wooding, C. Dix, C. Galtenberg, *XML Schema Complete Reference*, Addison Wesley, 2002.

13. R. T. Boute, “The Binary Decision Machine as a programmable controller”. EUROMICRO Newsletter, Vol. 1(2):1622, January 1976.

14. R. K. Brayton, G. D. Hatchel, A. Sangiovanni-Vichentelli, F. Somenzi, A. Aziz, S. T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, T. Villa, “VIS”, *Proc. 1st Int. Conf. on Formal Methods in Computer-Aided Design*, in *Lecture Notes in Computer Science*, Vol. 1166, Springer, 1996, 248-256.

15. D. Brownell, *SAX2*, O’Reilly, 2002.

16. S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic, *Field Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

17. R. E. Bryant, “Graph-based algorithms for Boolean functions manipulation”, *IEEE Trans. Computers*, Vol. C-35, No. 8, 1986, 667-691.

18. R. E. Bryant, “Symbolic Boolean manipulation with ordered Binary Decision Diagrams”, *ACM Computing Surveys*, Vol. 24, 1992, 293-318.

19. R. E. Bryant, Y. A. Chen, “An efficient graph representation for arithmetic circuit verification”, *IEEE Trans. Computer-Aided Design of Intgerated Circuits and Systems*, Vol. 20, No. 12, 2001, 1443-1454.

20. R. E. Bryant, Y. A. Chen, *Verifcation of Arithmetic Functions with Binary Moment Diagrams*, Tech Rept. CMU-CS-94-160, May 31, 1994.

21. P. Buch, A. Narayan, A. R. Newton, A. L. Sangiovanni-Vincentelli, “Logic Synthesis for Large Pass-Transistor Circuits”, *Proc. Int. Conf. on Computer Aided Design, ICCAD 1993*, San Jose, California, USA, November 9-13, 1993, 663-670.

22. *BuildGates User guide Release 2.0*, Ambit Design Systems, Santa Clara, CA, USA, December 1997.

23. J. Burch, E. Clarke, D. Long, K. McMillan, D. Dill, "Symbolic model checking for sequential circuit verification", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 4, 1994, 401-424.

24. K. Cagle, *SVG Programming: The Graphical Web*, Apress, 8 July 2002.

25. *Catapult Synthesis*, Mentor Graphics Corp, http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis.

26. Celoxica Corp, *Handle C Design Language*, http://www.celoxica.com.

27. S. Chakravarti, "On the complexity of using BDDs for synthesis and analysis of Boolean circuits", *Proc. 27th Annual Conference on Communication, Control and Computing*, Allerton, Illinois, September 1989, 730-739.

28. D. L. Chalmers, "History of EDIF and experiences of CAD 031", *Proc. IEE Colloquium on Electronic Interchange Format - EDIF*, London, UK, November 16, 1988, 1/1 - 1/4.

29. M. Cirit, "Estimating dynamic power consumption of cMOS Circuits", *Proc. 5th Int. Conf. on Computer-Aided Design, ICCAD 1987*, Santa Clara, CA, USA, November 1987, 534-537.

30. E. M. Clarke, M. Fujita, X. Zhao, "Multi-terminal decision diagrams and hybrid decision diagrams", in T. Sasao, M. fujita, (Eds.), *Representation of Discrete Functions*, Kluwer Academic Publishers, 1995, 91-108.

31. E. M. Clarke, M. Khaira, X. Zhao, "Word-level model checking: Avoiding Pentium FDIV error", *Proc. 33rd Design Automation Conference, DAC-1996*, Las Vegas, Nevada, USA, June 3-7, 1996, 645-648.

32. E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, "Spectral transforms for extremely large Boolean functions", in: Kebschull, U., Schubert, E., Rosenstiel, W., (Eds.=, *Proc. IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 16-17.9.1993, Hamburg, Germany, 86-90.

33. E. M. Clarke, K.L. McMillan. X. Zhao, M. Fujita, and J. Yang "Spectral Transforms for Large Boolean Functions with Application to Technology Mapping", *Proc. 30th Design Automation Conference, DAC-1993*, Dallas, Texas, USA, June 14-18, 1993, 54-60.

34. O. Coudet and J. Madre, "Implicit and incremental computation of primes and essential primes of Boolean functions", *Proc. 29th Design Automation Conference, DAC-1992*, Anaheim, California, USA, June 8-12, 1992, 36-39.

35. O. Coudert, Ch. Berthet, J. Ch. Madre, "Verification of sequential machines using Boolean functional vectors", *Proc. IMEC-IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, November 1990, 111-128.

36. T. M. Cover, J. A. Thomas, *Elements of Information Theory*, John Wiley & Sons, 1991.

37. *CynApps Suite, Cynthesis Application for Higher Level Design*, http://www.cynapps.com.

38. M. Dagenais, V. Agarwal and N. Rumin, "McBOOLE: A new procdure of exact logic minimization", *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, Vol. CAD-5, No. 1, 1986, 229-233.

39. M. Davio, J. P. Deschamps, A. Thayse, *Digital Systems with Algorithm Implementation*, Wiley & Sons, 1983.

40. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw Hill, 1994.

41. *DesigWare Components, Quick Reference Guide, Version 1998.02*, Synopsys, Mountain View, CA, USA, February 1998.

42. D. L. Dill, "The Murφ verification System", *Proc. 8th Int. Conf. on Computer-Aided Verification*,in *Lecture Notes in Computer Science*, Vol. 1102, Springer, 1996, 390-393.

43. *Document Object Model (DOM) Level 1 Specification, W3C Recommendation*,
http://www.w3.org/TR/REC-DOM-Level-1/, October 1, 1998.

44. R. Drechsler, *Advanced Formal Verification*, Springer, 2004.

45. R. Drechsler, *Formal Verification of Circuits*, Springer, 2000.

46. R. Drechsler, B. Becker, *Binary Decision Diagrams - Theory and Implementations*, Kluwer Academic Publishers, 1998.

47. R. Drechsler, B. Becker, "OKFDDs-algorithms, applications and extensions", in [145], 163-190.

48. R. Drechsler, B. Becker, "Ordered Kronecker functional decision diagrams - a data structure for representation and manipulation of Boolean functions", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-17, No. 10, 1998, 965-973.

49. R. Drechsler and B. Becker, "Overview of decision diagrams", *IEE Proc. Comput. Digit. Tech.*, Vol. 144, No. 3, May 1997, 187-193.

50. R. Drechler, B. Becker, S. Ruppertz, “The K∗BMD: A verification data structure”, *IEEE Design & Test of Computers*, April - June, 1996, 51-59.

51. R. Drechsler, M. Kerttu, P. Lindgren, M. Thornton, “Low power optimization techniques for BDD mapped circuits”, *Proc. Asia and South Pacific Design Automation Conference, ASP-DAC 2001*, Yokohama, Japan, January 30-February 2, 2001, 615-621.

52. R. Drechsler, M. Kerttu, P. Lindgren, M. Thornton, “Low power optimization techniques for BDD mapped circuits using temporal correlation”, *Proc. Int. Workshop on System-on-Chip for Real Time Applications*, Banff, Alberta, Canada, July 6-7, 2002, 400-409.

53. R. Drechsler, R. S. Stanković, T. Sasao, “Spectral transforms and word-level decision diagrams”, *Proc. Workshop on Synthesis and System Integration of Mixed Technologiues, SASIMI-97*, December 1-2, 1997, Osaka, Japan, 39-44.

54. *Edif Electronic Design Interchange Format Version 2 0 0*, Electronic Industries Assn, June 1989.

55. *Efficient XML Interchange Working Group*, W3 Consortium Recommendation,
http://www.w3.org/XML/EXI/

56. J. R. Egan, C. L. Liu, “Bipartite Folding and partitioning of a PLA”, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-3, No. 3, 1982, 191-199.

57. J. Eisenberg, *SVG Essentials (O’Reilly XML)*, O’Reilly Media Inc., 5 February 2002.

58. D. Esposito, *Applied XML Programming for Microsoft .NET*, Microsoft Press, October 9, 2002.

59. *Esterel-C Language (ECL)*, Cadence website,
http://www.cadence.com.

60. *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation,
http://www.w3.org/TR/2004/REC-xml-20040204/, 4 February 2004.

61. R. Ferreira, A. M. Trullemans, “BDD variants for probability polynomials”, *Proc. Int. Conf. MALOPD ’99*, Moscow, Russia, September 13-14, 1999, 12-19.

62. R. Ferreira, A. M. Trullemans, J. Costa, J. Monteiro, “Probabilistic bottom-up RTL power estimation”, *Proc. First Int. Symp. on Quality of Electronic Design, ISQED-2000*, San Jose, California, USA, March 20-22, 2000, 439-443.

63. N. J. Fine, "On the Walsh functions", *Trans. Amer. Math. Soc.*, No. 3, 1949, 372-414.

64. R. J. Francis, J. Rose, Z. Vranesic, "Technology mapping of Look-up Table based FPGAs for performance", *Proc. Int. Conf. Computer Aided Design*, Santa Clara, CA, USA, March 11-14, 1991, 568-571.

65. J. B. Fourier, "Theorie analytique de la chaleur", *Euvres 1*, see also the Fourier's paper from 1811 published 1824 in *Mem. de l'Acad. des Sci.*, Paris, (2), 4, 1819/20, 135-55, published 1824.

66. H. Fujiwara, *Design and Test for Digital Systems*, Kogakutsho, Tokyo, Japan, 2004.

67. H. Fujiwara, *Logic Logic Testing and Design for Testability*, MIT Press, Boston, USA, 1985.

68. D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs", *Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM 95)*, Napa, California, USA, April 19-21, 1995, 136-144.

69. J. R. Gardner, Z. L. Rendon, *XSLT and XPATH: A Guide to XML Transformations*, Prentice Hall PTR, 2001.

70. S. Geman, D. Geman, "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images", *IEEE Trans. Pattern Anal. Machine Intelligence*, Vol. 6, No. 6, 1984, 721-741.

71. A. Ghosh, S. Devedas, K. Keutzer, J. White, "Estimation of average switching activity in combinational and sequential circuits", *Proc. 29th Design Automation Conference, DAC-1992*, Anaheim, Callifornia, USA, June 8-12, 1992, 253-259.

72. M. Gokhale, J. Stone, J. Arnold, M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language", *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, Napa, California, April 17-19, 2000, 49-56.

73. C. F. Goldfarb, Y. Rubinsky, *The SGML Handbook*, Oxford University Press, 1991.

74. G. Grimmett, D. Stirzaker, *Probability and Random Processes*, 2nd ed., Oxford University Press, 1992.

75. W. Gunther, R. Drechsler, "Action: Combining logic synthesis and technology mapping for MUX based FPGAs", *Proc. 26th Euromicro Conf.*, Maastricht, Netherlands, September 5-7, 2000, Vol. 1, 130-137.

76. W. Gunther, R. Drechsler, "Performance driven optimization for MUX based FPGAs", *Proc. 14th Int. Conf. on VLSI Design*, Bangalore, India, January 3-7, 2001, 311-316.

77. G. D. Hachtel, A. R. Newton, A. L. Sangiovanni-Vincentelli, "Some results in optimal PLA folding", *Proc. Int. Conf. on Circuits and Computers, ICCC-80*, Rye, New York, USA, October 1-3, 1980, 1023-1028.

78. G. D. Hachtel, A. R. Newton, A. L. Sangiovanni-Vincentelli, "An algorithm for optimal PLA folding", *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, Vol. CAD-1, No. 2, 1982, 63-77.

79. G. D. Hachtel, F. Somenzi, *Logic Synthesis and Verification*, Kluwer Academic Publishers, 2000.

80. B. V. Hagen, *SGML for Dummies*, John Wiley & Sons, 1997.

81. E. R. Harold, *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*, Addison-Wesley, 2002.

82. E. R. Harold, W. S. Means, *XML in a Nutshell*, 2nd Edition, O'Reilly, 2002.

83. M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "A system for synthesizing optimized FPGA hardware from Matlab", *Proc. Int. Conf. on Computer Aided Design*, San Jose, CA, USA, November 4-8, 2001, 314-319.

84. F. Harary, *Graph Theory*, Reading, MA: Addison-Wesley, p. 6, 1994.

85. H. M. Hasan Babu, T. Sasao, "Representations of multiple-output switching functions using multiple-valued pseudo-Kronecker decision diagrams", *Proc. 30th Int. Symp. on Multiple-Valued Logic*, Portland, Oregon, USA, May 23-25, 2000, 147-152.

86. S. Holzner, Inside XSLT, New Riders Publishing, 10 July 2001.

87. A. J. Hu, D. L. Dill, A. J. Drexler, C. H. Yang, "High-level Specification and Verification with BDDs", *Proc. 4th Int. Conf. on Computer-Aided Verification*, in *Lecture Notes in Computer Science*, Vol. 663, Springer, 1992, 82-95.

88. A. J. Hu, D. L. Dill, "Reducing BDD size by exploiting functional dependencies", *Proc. 30st Design Automation Conference, DAC-1993*, Dallas, USA, Juner 14-18, 1993, 266-271.

89. A. J. Hu, G. York, D. L. Dill, "New techniques for efficient verification with implicitly conjoined BDDs", *Proc. 31st Design Automation Conference, DAC-1994*, San Diego, California, USA, June 6-10, 1994, 276-281.

90. S. L. Hurst, D. M. Miller, J. C. Muzio, *Spectral Techniques in Digital Logic*, Academic Press, 1985.

91. Y. Iguchi, T. Sasao, M. Matsura, "Evaluation of multiple-output logic functions using decision diagrams", *Proc. Asia and South Pacific Design Automation Conference, ASP-DAC 2003*, January 21-24, 2003, 312-315.

92. Y. Iguchi, T. Sasao, and M. Matsura, "On properties of Kleene TDDs", *Proc. Asia and South Pacific Design Automation Conference, ASP-DAC 1997*, Yokohama, Japan, January 28-31, 1997, 473-476.

93. G. Jennings, "Symbolic incompletely specified functions for correct evaluation in the presence of indeterminate input variables", *Proc. Twenty-Eight Annual Hawaii Int. Conf. on System Sciences, HICSS-28*, Vol. I, Architecture, January 3-6, 1995, 23-31.

94. H. J. Kahn, R. F. Goldman, "The eElectronic Design Interchange Format EDIF: present and future", *Proc. 29th Design Automation Conference, DAC-1992*, June 8-12, 1992, 666-671.

95. M. G. Karpovsky, *Finite Orthogonal Series in the Design of Digital Devices*, Wiley & Sons, 1976.

96. J. L. Kouloheris, A. Gamal, "FPGA performance versus cell granularity", *Proc. IEEE Custom Integration Circuit Conf.*, San Diego, CA, USA, May 12-15, 1991, 6.2/1-6.2/4.

97. A. Kuhlmann, A. Srinivasan, D. P. Lapotin, "Verty - A formal verification program for custom CMOS circuits", *IBM Journal for Research and Development*, Vol. 39, No. 1/2, January/March 1995, 149-165.

98. G. A. Kukharev, V.P. Shmerko, and E.N. Zaitseva, *Multiple-Valued Data Processing Algorithms and Systolic Processors*, Minsk: Science and Engineering, 1990.

99. M. Kwiatkowska, G. Norman, D. Parker, "PRISM: Probabilistic Symbolic Model Checker", *Proc. TOOLS 2002, Lecture Notes in Computer Science*, No. 2324, 2002, 200-204.

100. Y. T. Lai, S. Sastry, "Edge-valued Binary Decision Diagrams for Multilevel Hierarchical Verification", *Proc. Design Automation Conference, DAC-1992*, Anaheim, California, USA, June 8-12, 1992, 668-613.

101. T. Larrabee, "Test pattern generation using Boolean satisfiability", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-11, No. 1, 1992, 4-15.

102. C. Y. Lee, "Representation of switching circuits by binary-decision programs" Bell. Syst. Tech. J., vol. 38, 985-999, July 1959.

103. W. Leelapatra, K. Kanchanasut, C. Lurnisap, “Displacement BDD and geometric transformations of binary decision diagram encoded images”, *Pattern Recognition Letters*, No. 29, Elsevier, 2008, 438-456.

104. W. Leelapatra, K. Kanchanasut, C. Lurnisap, “Geometric transformations of BDD encoded image”, *Int. J. Appl. Math.*, Vol. 1, No. 36, 2008, 438-456.

105. C. Lurnisap, K. Kanchanasut, T. Siriboon, “Basic binary decision diagram operations for image processing”, *Proc. Third Asian Computing Science Conf.*, *Lecture Notes in Computer Science*, Springer, 1997, 368-370.

106. S. Mangano, *XSLT Cookbook*, O’Reilly Media Inc., December 2002.

107. R. Marculescu, D. Marculescu, M. Pedram, “Efficient power estimation for highly correlated input streams”, *Proc. 32th Design Automation Conference, DAC-1995*, San Francisco, California, USA, June 12-16, 1995, 628-634.

108. R. Marculescu, D. Marculescu, M. Pedram, “Information theoretic masures for power analysis”, *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, Vol. CAD-15, No. 6, 1996, 599-610.

109. P. Marinos, “Derivation of minimal complete sets of test-input sequences using Boolean differences”, *IEEE Trans. Computers*, Vol. 20, No. 1, 1981, 25-32.

110. P. Mateu-Villarroya, J. Prades-Nebot, “Lossless image compression using ordered binary-decision diagrams”, *IEEE Electronics Letters*, Vol. 37, No. 3, 2001, 162-163.

111. C. Maxfield, *The Design Warrior’s Guide to FPGAs*, Elsevier, Amsterdam, Netherlands, 2004.

112. E. McCluskey, “Minimization of Boolean functions”, *The Bell Systems Technical Journal*, Vol. 35, November 1956, 1417-1444.

113. B. McLaughlin., J. Edelson, *Java and XML*, O’Reilly Media, December 8, 2006.

114. K. L. McMillan, *Symbolic Model Checking: an Approach to the State Explosion Problem*, PhD Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, United States, 1992.

115. D. Megginson, *Java SAX, Features and Properties*, http://www.saxproject.org/get-set.html

116. D. M. Miller, "An improved method for computing a generalized spectral coefficient," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17, No. 3, pp. 233-238, March, 1998.

117. D. Miller, R. Drechsler, "Implementing a multiple-valued decision diagram package", *Proc. 28th Int. Symp. on Multiple-Valued Logic*, Fukuoka, Japan, 27-29 May, 1998, 52-57.

118. D. Miller, R. Drechsler, "On the construction of multiple-valued decision diagrams", *Proc. 32nd Int. Symp. on Multiple-Valued Logic*, Boston, Massachusetts, USA, May 15-18, 2002, 245-253.

119. S. Minato, *Binary Decision Diagrams and Applictions for VLSI Synthesis*, Kluwer Academic Publishers, 1996.

120. S. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems", *Proc. Design Automation Conference, DAC-1995*, San Francisco, California, USA, June 12-16, 1993, 272-277.

121. S. Minato, N. Ishiura, S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation", *Proc. 27th Design Automation Conference, DAC-1990*, Orlando, Florida, USA, June 24-28, 1990, 52-57.

122. S. Muroga, *Logic Design and Switching Theory*, Jon Wiley & Sons, 1979, Reprinted edition Krieger Publishing Company, Malaber, FL, USA, 1990.

123. S. Nagayama, T. Sasao, "Compact representations of logic functions using heterogeneous MDDs", *Proc. 33rd Int. Symp. on Multiple-Valued Logic*, Tokyo, Japan, May 15-18, 2003, 247-252.

124. *Overview of the Open SystemC Initiative*, SystemC website, http://www.systemc.org.

125. I. Page, "Hardware-software co-synthesis research at Oxford", *Proc. IEE Vacation School on Hardware/Software Co-design*, Manchester, UK, July 13-16, 1997.

126. A. Pal, "An algorithm for optimal logic design using multiplexers", *IEEE Trans. Computers*, Vol. C-35, No. 8, 1986, 755-757.

127. A. Pal, R. K. Gorai, V. V. Raju, "Synthesis of multiplexer networks using ratio parameters and mapping onto FPGAs", *Proc. 8th Int. Conf. on VLSI Design*, New Delhi, India, January 4-7, 1995, 63-68.

128. K. P. Parker, E. J. McCluskey, "Analysis of logic circuits with faults using input probabilities", *IEEE Trans. Computers*, Vol. 24, No. 5, 1975, 573-578.

129. K. P. Parker, E. J. McCluskey, “Probabilistic Treatment of General Combinational Networks”, *IEEE Trans. Computers*, Vol. 24, No. 6, 1996, 588-598.

130. D. L. Perry, *VHDL: Programming by Examples*, McGraw-Hill Professional, 2002.

131. C. Pixley, S. W. Jeong, G. D. Hatchel, “Exact calculation of synchronizing sequences on based on Binary decision diagrams”, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-13, No. 8, 1994, 1024-1034.

132. D. V. Popel, “Synthesis of low-power digital circuits derived from Binary decision diagrms, *Proc. IEEE European Conference on Circuit Theory and Design, ECCTD'01*, Espoo, Finland, August 28-31, Vol. 3, 2001, 317-320.

133. W. Quine, “The problem of simplifying truth functions”, *Amer. Math. Monthly*, Vol. 59, 1952, 521-531.

134. K. Ravi, F. Somenzi, “High-density reachability ananlysis”, *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, San Jose, California, USA, November 5-9, 1995, 154-158.

135. S. M. Reddy, *Easily Testable Realization for Logic Functions*, Technical Report No. 54, Univ. of Iowa, USA, May 1972.

136. L. Robert, G. Malandain, “Fast binary image processing using Binary decision diagrams”, *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'97)*, San Juan, Puerto Rico, June 17-19, 1997, 97-102.

137. J. Rose, R. J. Francis, D. Lewis, P. Chow, “Architecture of Field Programmable Gate Arrays, the effect of logic block functionality on area efficiency”, *IEEE Journal on Solid-State Circuits*, Vol. 25, No. 5, 1990, 1217-1225.

138. J. Rose, A. Gamal, A. Sangiovanni-Vincentelli, “Architecture of Field Programmable Gate Arrays”, *Proc. IEEE*, Vol. 81, No. 7, 1993, 1013-1029.

139. R. Rudell and A. Sangiovanni-Vincentelli, “Multiple-valued minimization for PLA optimization”, *IEEE Trans. Computer Aided Design of Integrated Circutis and Systems*, Vol. CAD-6, No. 5, 1987, 727-750.

140. T. Sasao, “Easily testable realizations for Generalized Reed-Muller expressions”, *IEEE Trans. Computers*, Vol. 46, No. 6, 1997, 709-716.

141. T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

142. T. Sasao, "Ternary decision diagrams and their applications", *Proc. Int. Symp. on Multiple-Valued Logic, ISMVL-97*, Antigonish, Nova Scotia, Canada, May 28-30, 1997, 241-250.

143. T. Sasao. "Ternary decision diagrams and their applications" in: [145], 269-292.

144. T. Sasao, J. T. Butler, "A design method for Look-up Table type FPGA by Pseudo-Kronecker expansion", *Proc. 24th Int. Symp. on Multiple-Valued Logic*, May 25-27, 1994, 97-106.

145. T. Sasao, M. Fujita, (ed.), *Representations of Discrete Functions*, Kluwer Academic Publishers, 1996.

146. *Scalable Vector Graphics (SVG) 1.1 Specification*, W3C Recommendation,
http://www.w3.org/TR/SVG11, 14 January 2003.

147. F. F. Sellers, M. Y. Hsiao, L. W. Bearnson, "Analyzing errors with the Boolean difference", *IEEE Trans. Computers*, Vol. 17, No. 7, 1968, 676-683.

148. M. Shahdad, "An interface between VHDL and EDIF", *Digest of Papers of Thirty-Third IEEE Computer Society Int. Conf. Compcon Spring '88*, February 29-March 3, 1988, 316 - 317.

149. C. E. Shannon, "A mathematical theory of communications", *Bell Sys. Tech. J.*, Vol. 27, 1848.

150. F. Somenzi, "Efficient manipulation of decision diagrams", *Int. Journal on Software Tools for Technology Transfer*, 3, 2001, 171-181.

151. F. Somenzi, "CUDD Decision Diagram Package",
http://bessie.colorado.edu˜/fabio/CUDD

152. M. A. Spink, "An introduction to EDIF views, structures and syntax", *Proc. IEE Colloquium on Electronic Interchange Format - EDIF*, London, UK, November 16, 1988, 2/1-2/4.

153. M. Starkey, R. E. Bryant, *Using Ordered Binary Decision Diagrams for Compressing Images and Image Sequences*, Technical Report, CMU-CS, 1995, 95-105.

154. R. S. Stanković, J. T. Astola, *Spectral Interpretation of Decision Diagrams*, Springer, 2003.

155. R. S. Stanković, M. S. Stanković, D. Janković, *Spectral Transforms in Switching Theory, Definitions and Calculations*, Nauka, Belgrade, Serbia, 1998.

156. S. Stanković, J. Astola, "Calculating Entropy Estimate Using Binary Decision Diagrams", Proc. XI International Symposium on Problems of Redundancy in Information and Control Systems, 02 - 06 July, 2007, Saint Petersburg, Russia, 32-36.

157. S. Stanković, J. Astola, "Method for automatic generation of branching programs using decision diagrams", Proc. The 2007 Int. TICSP Workshop on Spectral Methods and Multirate Signal processing, SMMSP 2007, Moscow, Russia, September 3-4, 2007, paper cr1023.

158. S. Stanković, J. Astola, "QDD Based Method of Automatic Circuit Design for Xilinx Virtex-5 FPGA Devices", *Journal of Multiple-Valued Logic and Soft Computing*, accepted for publicaiuton.

159. S. Stanković, J. Astola, "XML framework for various types of decision diagrams for discrete functions", *IEICE Trans. Inf. and Syst.*, Vol. E90-D, No. 11, 2007, 1731-1740.

160. S. Stanković, J. Astola, "XSLT Based Method for Automatic Generation of a Graphical Representation of a Decision Diagram Represented using XML", 7th International Workshop on Boolean Problems, Freiberg, Germany, 21-22 Sept. 2006.

161. S. Stanković., J. Takala, J. Astola, "Method for Automatic Generation of RTL in VHDL Using Decision Diagrams", Proc. The 2006 Int. TICSP Workshop on Spectral Methods and Multirate Signal processing, SMMSP, Florence, Italy, 2006.

162. S. Stojković, "UDDP Universal Decision Diagram Package", *Acta Electrotechnica et Informatica*, Vol. 5, No. 1, 2005, Košice, Slovakia.

163. *System Compiler: Compiling ANSI C/C++ to Synthesis-ready HDL. Whitepaper. C level Design Incorporated*, http://www.cleveldesign.com.

164. M. A. Thornton, D. M. Miller, R. Drechsler, *Spectral Techniques in VLSI CAD*, Kluwer Academic Publishers, 2001.

165. M. A. Thornton, V. S. S. Nair, "Efficient calculation of spectral coefficients and their applications", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-14, No. 11, 1995, 1328-1341.

166. H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, A. Sangiovanni-Vincentelli, "Implicit state enumeration for finite state machines using BDDs", *Digest of Technical Papers of IEEE Int. Conf. on Computer-Aided Design, ICCAD'90*, Santa Clara, California, USA, November 11-15, 1990, 130-133.

167. S. Trimberger, (ed.), *Field-Programmable Gate Arrays*, Kluwer Academic Publisher, Boston, MA, USA, 1994.

168. S. Trimberger, “Effect of FPGA architecture to FPGA routing”, *Proc. 32nd Design Automation Conference, DAC-1995*, San Francisco, CA, USA, June 12-16, 1995, 574-578.

169. K. Tsuchiya, Y. Tekefuji, “A neural network approach to PLA folding problems”, *IEEE Trans. Computer-Aided Design of Integral Circuits*, Vol. CAD-15, No. 10, 1996, 1299-1305.

170. R. Ubar, “Test Generation for Digital Circuits Using Alternative Graphs (in Russian)”, in Proc. Tallinn Technical University, No.409, 7581, Tallinn, Estonia, 1976.

171. S. H. Unger, *Asynchronous Sequential Switching Circuits*, John Wiley & Sons, 1969.

172. *Virtex-5 SXT Platform Technical Background*, Xilinx, inc., 5 February 2007.

173. S. B. K. Vrudhula, M. Pedram, Y. T. Lai,“Edge-valued Binary decision diagrams”, in [145], 109-132.

174. *W3C XML Schema Definition Language (XSDL) 1.1 Part 1: Structures*, W3C Working Draft, http://www.w3.org/TR/xmlschema11-1/, 30 August 2007.

175. J. L. Walsh, “A closed set of orthogonal functions”, *American Journal of Mathematics*, Vol. 55, 1923, 5-24.

176. A. H. Watt, C Lilley, *SVG Unleashed*, Sams, 20 September 2002.

177. R. A. Wod, “A high density programmable logic array chip”, *IEEE Trans. Computers*, Vol. C-28, No. 9, 1979, 602-608.

178. W. Wolf, *FPGA-Based System Design*, Prentice-Hall, Englewood Clifs, NJ, USA, 2004.

179. J. Wu, K. Chung, “A new binary image representation: Logicodes”, *J. Vis. Commu. Image Represent.*, Vol. 8, No. 3, 1997, 291-298.

180. *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, http://www.w3.org/TR/xslt, 16 November 1999.

181. S. Yang, “Logic Synthesis and Optimization Benchmarks, Version 3.0,” Tech. Report, Microelectronics Center of North Carolina, 1991.

182. S. N. Yanushkevich, D.M. Miller, V.D. Shmerko, R.S. Stanković, *Decision Diagram Techniques for Micro- and Nanoelectronics Design Handbook*, CRC Press/Taylor & Frencis, 2006.

183. B. Zeidman, *Design with FPGAs and CPLDs*, CMP Books, Manharst, NY, USA, 2002.

184. *XML Binary Characterization*, W3 Consortium Recommendation, http://www.w3.org/TR/xbc-characterization/, 31 March 2005.

185. *XML Path Language (XPath) Version 1.0*, W3C Recommendation, http://www.w3.org/TR/1999/REC-xpath-19991116, 16 November 1999.

186. *XML Schema 1.1 Part 2: Datatypes*, W3C Working Draft, http://www.w3.org/TR/xmlschema11-2/, 17 February 2006.

Appendix A

XML Schema for XML representation of decision diagrams.

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.stane-island.net/theLab/XML"
 elementFormDefault="qualified"
 attributeFormDefault="unqualified"
 xmlns:dd="http://www.stane-island.net/theLab/XML">

 <xsd:complexType name="PointType">
  <xsd:choice>
   <xsd:element name="next_child" type="dd:PointType"
   minOccurs="0" maxOccurs="1" nillable="true" />
   <xsd:element name="next_parent" type="dd:PointType"
   minOccurs="0" maxOccurs="1" nillable="true" />
  </xsd:choice>
  <xsd:attribute name="point" type="xsd:integer" />
  <xsd:attribute name="variables" type="xsd:string" />
 </xsd:complexType>

 <xsd:complexType name="NodeType">
  <xsd:sequence>
   <xsd:element name="next" type="dd:NodeType"
   minOccurs="0" maxOccurs="1" nillable="true" />
   <xsd:element name="parents" type="dd:PointType"
   minOccurs="0" maxOccurs="1" nillable="true" />
   <xsd:element name="children" type="dd:PointType"
   minOccurs="0" maxOccurs="1" nillable="true" />
  </xsd:sequence>
  <xsd:attribute name="terminal" type="xsd:integer" />
  <xsd:attribute name="id" type="xsd:integer" />
  <xsd:attribute name="level" type="xsd:integer" />
  <xsd:attribute name="constant" type="xsd:integer" />
  <xsd:attribute name="rule" type="xsd:string" />
 </xsd:complexType>

  <xsd:complexType name="TreeType">
  <xsd:all>
   <xsd:element name="root" type="dd:NodeType"
   minOccurs="1" maxOccurs="1" />
  </xsd:all>
  <xsd:attribute name="type" type="xsd:string" />
  <xsd:attribute name="num_levels"
  type="xsd:integer" />
 </xsd:complexType>

 <xsd:element name="tree" type="dd:TreeType"/>

</xsd:schema>
```

Appendix B

Function 9sym from MCNC set of benchmarks implemented using BDD.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY DD_proba IS
  PORT (con0, con1, x0, x1, x2, x3, x4, x5, x6, x7, x8
  : IN std_logic; o : OUT std_logic);
END ENTITY DD_proba;

ARCHITECTURE DD_arch OF DD_proba IS

COMPONENT my_shannon
    PORT (f0, f1, x : IN std_logic; z : OUT std_logic);
END COMPONENT;

SIGNAL o2416, o1601, o928, o472, o203, o64, o4, o3,
o2, o63, o62, o202, o201, o471, o470, o927, o926, o925,
o459, o188, o39, o1600, o1599, o1598, o1597, o1596, o155,
o2415, o2414, o2413, o2412, o2411, o387 : std_logic;
BEGIN

s2416 : my_shannon
PORT MAP (o1601, o2415, x0, o);
```

```
s1601 : my_shannon
PORT MAP (o928, o1600, x1, o1601);

s928 : my_shannon
PORT MAP (o472, o927, x2, o928);

s472 : my_shannon
PORT MAP (o203, o471, x3, o472);

s203 : my_shannon
PORT MAP (o64, o202, x4, o203);

s64 : my_shannon
PORT MAP (o4, o63, x5, o64);

s4 : my_shannon
PORT MAP (con0, o3, x6, o4);

s3 : my_shannon
PORT MAP (con0, o2, x7, o3);

s2 : my_shannon
PORT MAP (con0, con1, x8, o2);

s63 : my_shannon
PORT MAP (o3, o62, x6, o63);

s62 : my_shannon
PORT MAP (o2, con1, x7, o62);

s202 : my_shannon
PORT MAP (o63, o201, x5, o202);

s201 : my_shannon
PORT MAP (o62, con1, x6, o201);

s471 : my_shannon
PORT MAP (o202, o470, x4, o471);

s470 : my_shannon
PORT MAP (o201, con1, x5, o470);

s927 : my_shannon
PORT MAP (o471, o926, x3, o927);

s926 : my_shannon
PORT MAP (o470, o925, x4, o926);
```

```
s925 : my_shannon
PORT MAP (con1, o459, x5, o925);

s459 : my_shannon
PORT MAP (con1, o188, x6, o459);

s188 : my_shannon
PORT MAP (con1, o39, x7, o188);

s39 : my_shannon
PORT MAP (con1, con0, x8, o39);

s1600 : my_shannon
PORT MAP (o927, o1599, x2, o1600);

s1599 : my_shannon
PORT MAP (o926, o1598, x3, o1599);

s1598 : my_shannon
PORT MAP (o925, o1597, x4, o1598);

s1597 : my_shannon
PORT MAP (o459, o1596, x5, o1597);

s1596 : my_shannon
PORT MAP (o188, o155, x6, o1596);

s155 : my_shannon
PORT MAP (o39, con0, x7, o155);

s2415 : my_shannon
PORT MAP (o1600, o2414, x1, o2415);

s2414 : my_shannon
PORT MAP (o1599, o2413, x2, o2414);

s2413 : my_shannon
PORT MAP (o1598, o2412, x3, o2413);

s2412 : my_shannon
PORT MAP (o1597, o2411, x4, o2412);

s2411 : my_shannon
PORT MAP (o1596, o387, x5, o2411);

s387 : my_shannon
PORT MAP (o155, con0, x6, o387);

END DD_arch;
```

Appendix C

Code of the SVG document with graphical representation of a ROBDD of the function $f(x_1, x_2, x_3) = x_1\overline{x}_2 \vee x_2x_3$.

```
<svg:svg width="21cm" height="15cm" version="1.1"
xmlns:svg="http://www.w3.org/2000/svg"
xmlns="http://www.w3.org/2000/svg">
    <svg:title>BDD</svg:title>
    <svg:g id="node0" text-align="center">
        <svg:ellipse cx="10.5cm" cy="1.75cm" rx="0.5cm"
        ry="0.5cm" stroke="rgb(0,0,0)" fill="rgb(240,240,240)"
        stroke-width="1" />
        <svg:line x1="10.5cm" y1="2.25cm"
        x2="5.25cm" y2="4.75cm"
        style="fill:none;stroke:rgb(128,128,128);
        stroke-width:1" />
        <svg:text x="7.975cm" y="3.4cm"
        font-family="Times Roman"
        font-size="12" fill="black">x1_comp</svg:text>
        <svg:line x1="10.5cm" y1="2.25cm"
        x2="15.75cm" y2="4.75cm"
        style="fill:none;stroke:rgb(128,128,128);
        stroke-width:1" />
        <svg:text x="13.225cm" y="3.4cm"
        font-family="Times Roman"
        font-size="12" fill="black">x1</svg:text>
        <svg:text x="10.5cm" y="1.95cm"
        style="font-family:TimesRoman;
        font-size:24;fill:black;">S</svg:text>
    </svg:g>
    <svg:g id="node1" text-align="center">
        <svg:ellipse cx="5.25cm" cy="5.25cm"
        rx="0.5cm" ry="0.5cm" stroke="rgb(0,0,0)"
        fill="rgb(240,240,240)"
        stroke-width="1" />
        <svg:line x1="5.25cm" y1="5.75cm"
        x2="5.25cm" y2="11.75cm"
        style="fill:none;stroke:rgb(128,128,128);
        stroke-width:1" />
        <svg:text x="5.35cm" y="8.65cm"
        font-family="Times Roman"
        font-size="12" fill="black">x2_comp</svg:text>
        <svg:line x1="5.25cm" y1="5.75cm"
        x2="10.5cm" y2="8.25cm"
        style="fill:none;stroke:rgb(128,128,128);
        stroke-width:1" />
        <svg:text x="7.975cm" y="6.9cm"
        font-family="Times Roman"
        font-size="12" fill="black">x2</svg:text>
        <svg:text x="5.25cm" y="5.45cm"
        style="font-family:TimesRoman;
        font-size:24;fill:black;">S</svg:text>
    </svg:g>
    <svg:g id="node2" text-align="center">
        <svg:rect x="4.875cm" y="11.75cm"
        width="0.75cm" height="1cm"
        stroke="rgb(0,0,0)" fill="rgb(240,240,240)"
        stroke-width="1" />
        <svg:text x="5.25cm" y="12.45cm"
        style="font-family:TimesRoman;
        font-size:24;fill:black;">0</svg:text>
    </svg:g>
    <svg:g id="node3" text-align="center">
        <svg:ellipse cx="10.5cm" cy="8.75cm"
        rx="0.5cm" ry="0.5cm"
        stroke="rgb(0,0,0)" fill="rgb(240,240,240)"
        stroke-width="1" />
        <svg:line x1="10.5cm" y1="9.25cm"
        x2="5.25cm" y2="11.75cm"
        style="fill:none;stroke:rgb(128,128,128);
        stroke-width:1" />
        <svg:text x="7.975cm" y="10.4cm"
        font-family="Times Roman"
        font-size="12" fill="black">x3_comp</svg:text>
        <svg:line x1="10.5cm" y1="9.25cm"
        x2="15.75cm" y2="11.75cm"
        style="fill:none;stroke:rgb(128,128,128);
        stroke-width:1" />
        <svg:text x="13.225cm" y="10.4cm"
        font-family="Times Roman"
        font-size="12" fill="black">x3</svg:text>
        <svg:text x="10.5cm" y="8.95cm"
        style="font-family:TimesRoman;
        font-size:24;fill:black;">S</svg:text>
    </svg:g>
    <svg:g id="node4" text-align="center">
        <svg:rect x="15.375cm" y="11.75cm"
        width="0.75cm" height="1cm"
        stroke="rgb(0,0,0)" fill="rgb(240,240,240)"
        stroke-width="1" />
        <svg:text x="15.75cm" y="12.45cm"
        style="font-family:TimesRoman;
        font-size:24;fill:black;">1</svg:text>
    </svg:g>
    <svg:g id="node5" text-align="center">
        <svg:ellipse cx="15.75cm" cy="5.25cm"
        rx="0.5cm" ry="0.5cm"
        stroke="rgb(0,0,0)" fill="rgb(240,240,240)"
        stroke-width="1" />
        <svg:line x1="15.75cm" y1="5.75cm"
        x2="10.5cm" y2="8.25cm"
        style="fill:none;stroke:rgb(128,128,128);
        stroke-width:1" />
        <svg:text x="13.225cm" y="6.9cm"
        font-family="Times Roman"
        font-size="12" fill="black">x2</svg:text>
        <svg:line x1="15.75cm" y1="5.75cm"
        x2="15.75cm" y2="11.75cm"
        style="fill:none;stroke:rgb(128,128,128);
        stroke-width:1" />
        <svg:text x="15.85cm" y="8.65cm"
        font-family="Times Roman"
        font-size="12" fill="black">x2_comp</svg:text>
        <svg:text x="15.75cm" y="5.45cm"
        style="font-family:TimesRoman;
        font-size:24;fill:black;">S</svg:text>
    </svg:g>
</svg:svg>
```