Bi-Decomposition of Function Sets in Multi-Valued Logic

Ch. Lang

March 8, 2002

Abstract

This dissertation presents a theory for bi-decomposition of multi-valued (MVL) functions. MVL functions are applied in MVL logic design and machine learning applications. The bi-decomposition decomposes a function into two decomposition functions that are connected by a binary operator called gate. Each of the decomposition functions depends on fewer variables than the original function. Recursive bi-decomposition represents a function as a structure of interconnected gates. For the logic design, the type of the gate can be chosen so that it has an efficient hardware representation. For machine learning, gates are selected so that they represent simple and understandable rules in a decision tree.

To describe the MVL bi-decomposition theory, the notion of incompletely specified functions is generalized to function sets. Particular classes of function sets, such as function lattices and intervals are identified, and properties of bi-decompositions are demonstrated with the help of these function sets.

Algorithms are presented for non-disjoint bi-decomposition, where variables may be shared by both decomposition functions. Bi-decomposition is discussed for monotone operators which are of special interest in machine learning. They lead to rules that can easily be understood by humans. For the special case of min- and max-decomposition, the theory of bi-decomposition for Boolean functions using Boolean differential calculus is extended, leading to particular efficient algorithms. The thesis also introduces new classes of operators, such as simple operators, and presents the corresponding decomposition algorithms.

The decomposition algorithms are implemented in the decomposition system YADE. Several test functions from machine learning applications are decomposed. It is demonstrated that decompositions of lower complexity can be found by decomposition of function sets instead of functions. The results are compared to other decomposers.

Contents

List of Symbols

$\{a_1, \ldots, a_n\}$	set of elements $a_1, \ldots, a_n$
$:=$	equal by definition
if_{DEF}	if and only if by definition
$\mathbf{\Phi}$	don't care value
$\mathbf{F}(A)$	function set
$R\langle A, v\rangle$	MVL relation
$[f_l(A), f_u(A)]$	function interval
F_C	care set
F_{DC}	don't care set
F_{ON}	on-set
F_{OFF}	off-set
inf	greatest lower bound of a partially ordered set
sup	least upper bound of a partially ordered set
$\wedge$	AND-operator
$\vee$	OR-operator
$\oplus$	EXOR-operator
$\oplus_m$	sum modulo m or modsum-operator
$\ominus_m$	difference modulo m
$\cap$	intersection of sets
$\cup$	union of sets
$\times$	Cartesian product of sets
$\min^{\text{k}}{}_A$	k-times minimum over A
$\max^{\text{k}}{}_A$	k-times maximum over A
$\lvert X \rvert$	number of elements in the set X
round(r)	rounds the real number r to the nearest integer
$\lceil r \rceil$	smallest integer k with $k \geq r$
$\mathbb{B}^n$	n-dimensional Boolean space
$\mathbb{M}_A$	set of all MVL minterms of the set of variables A
$\mathbb{F}_m(A)$	set of all functions $f(A)$ with output cardinality m
$\mathcal{S}^c_{z=k}$	σ-transformation
$f(A)\lvert_{A_i}$	the function $f(A)$ evaluated for the minterm A_i
const^c_B	const-operator

List of Acronyms

BDC	Boolean Differential Calculus
BDD	Binary Decision Diagram
BEMDD	Binary Encoded Multi-valued Decision Diagram
CISF	Combinated Incompletely Specified Function
CMOS	Complementary Metal-Oxide Semiconductor
DB	Data Base
DC	Don't Care
DFC	Discrete Function Cardinaltity
ISF	Incompletely Specified Function
MDD	Multi-valued Decision Diagram
MM	Min and Max
MOS	Metal-Oxide Semiconductor
MVL	Multi-Valued Logic
STL	Standard Template Library
VLSI	Very Large Scale Integration
YADE	Yet Another DEcomposer

Chapter 1

Introduction

1.1 Motivation

The divide and conquer principle has been recognized as an important and powerful tool in many areas of computer science. Applied to the area of digital circuits, this strategy has been implemented by decomposition of functions. Given a function as a behavioral description of the circuit, the function is decomposed into interconnected smaller functions, and thus, revealing the structure of the circuit. Although the idea of *multi-valued logic (MVL)* circuits is not new, the overwhelming majority of discrete circuits today is two-valued. Two-valued systems are best described by the Boolean logic. Synthesis and analysis of two-valued systems have been a research topic for more than forty years. Methods for decomposition of Boolean functions are well developed.

Advances in the field of semi-conductors indicate an increasing interest in circuits with more than two values. The number of interconnections is becoming an increasing problem in modern VLSI chips [13]. MVL signals can carry more information than two-valued signals and hence, the number of interconnections can be reduced. MVL circuits can be processed in a variety of technologies including Current-Mode CMOS [43] and neuron MOS [12, 11]. Future quantum electronic circuits may exploit tunneling effects [45, 46].

Research on MVL circuits has led to comercial applications. Intel has designed a RAMBUS interface that doubles the bandwidth of the communication between the processor and the memory by application 4-valued signals that transmit 2 bits per clock cycle over a single wire [31]. In modern memories, the size of a memory cell reaches physical limits, and density of the memory can be increased by storing more than one bit per cell, which implies more that two states. The application of MVL circuits requires design methods for MVL systems.

Bi-decomposition of Boolean functions has compared well to other methods of decomposition [24], such as Curtis decomposition. Therefore, decomposition of MVL functions should also be investigated.

Synthesis of logic by functional decomposition involves a decomposition step and a technology mapping, see Figure 1.1(a). A logic function is decomposed into smaller blocks by functional decomposition. Then, a mapping with a technology library synthesizes each block from the gates of the library. The result is a netlist of gates that can be realized in the chosen technology.

Bi-decomposition gives complete control over the type of gates that is produced during decomposition. The bi-decomposition process can be controlled by the technology library of available gates directly and no technology mapping is necessary, see Figure 1.1(b).

Recently, MVL decomposition has found applications in the field of *data mining* [49]. Over the past decade much data has been stored digitally and huge data bases have been created. There arises the question of how to retrieve important information from the large amount of unstructured data.

Example 1.1. Consider the simplified lenses problem from the UCI data base [44]. There are guidelines for a doctor which type of contact lenses should be prescribed for a patient. In this example, the prescription depends on three independent parameters, the age of the patient, the spectacle prescription, and the tear production rate. The parameters are encoded by the input variables *age*, *prescription* and *tear* respectively. The codes and values of these variables are

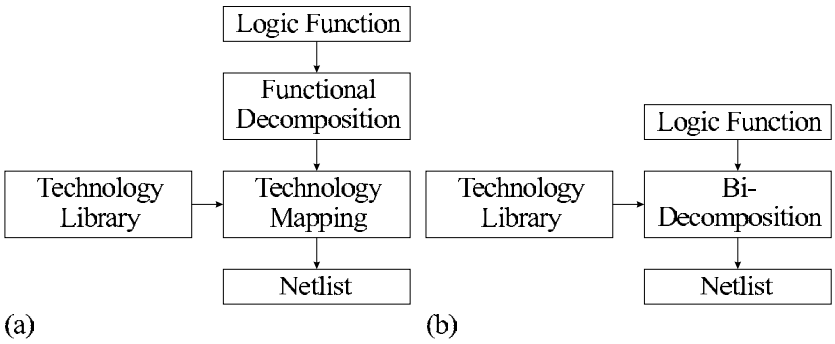

Figure 1.1: Design flow in logic synthesis. **(a)** Design by functional decomposition with technology mapping. **(b)** Design by bi-decomposition without technology mapping.

Table 1.1: Lenses problem: Codes for the variables of the lens prescription rules.

Variable	Type	Meaning	Code	Value
age	input	age of the patient	0 1 2	young pre_presbyopic presbyopic
prescription	input	spectacle prescription	0 1	myope hypermetrope
tear	input	tear production rate	0 1	reduced normal
lens	output	type of contact lenses	0 1 2	hard soft no_contacts

shown in table 1.1. The type of contact lenses, the goal variable, is encoded by the variable *lens* also shown in the table. The guidelines for the prescription are given in table 1.2.For a patient the doctor can determine the values of the variables *age*, *prescription* and *tear*. Then, with help of table 1.2, the doctor can look up the type of contact lenses that should be prescribed.

Consider a patient of presbyopic age, with a myope spectacle prescription and with normal tear production rate. According to table 1.1, this case corresponds to the set of input variables $\{age = 2, prescription = 0, tear = 1\}$. It can be derived from table 1.2 (bold line) that $lens = 2$, and hence, this patient should not be prescribed contact lenses.

However, there are difficulties when looking up a solution from a table.

- The data is not structured. Each case is treated independently, no relations between similar cases are apparent from the table.
- The solution is difficult to understand by humans. The table just tells the solution without explanation.
- In many problems of the real world there is a great number (up to several hundred) input variables. Because the number of parameter combinations grows exponentially with the number of variables, it is impossible to list all combinations in a table. The reason is not primarily the storage capacity of computers, but the fact that knowledge is often incomplete. In many cases the value of the goal variable is not known for some input combinations. In fact, there are many examples where the value of the goal variable is not known for more than 99.9 % of the input combinations [44].

To solve these problems, some structure must be introduced into the data. The structure should reveal hidden relationships between cases and should be understandable by humans. Furthermore, it should be possible to correctly extrapolate from the known values of the goal variable to the unknown cases. In the machine learning theory there are many ways to introduce such a structure into a data base. Data mining includes pattern recognition, statistical analysis

Table 1.2: Lenses problem: Rules for prescription of contact lenses.

age	*prescription*	*tear*	*lens*
0	0	0	2
0	0	1	0
0	1	0	2
0	1	1	1
1	0	0	2
1	0	1	1
1	1	0	2
1	1	1	1
2	0	0	2
2	**0**	**1**	**2**
2	1	0	2
2	1	1	1

and machine learning as methods of data analysis. Given a set of training examples, there are two major goals for data mining. The first one is to partition new examples with a small classification error. The second one is to derive a simple model of the data that can easily be understood by humans.

Usually, data analysis starts with a model developed by a human domain expert. Then, the parameters of the model are adjusted according to the given training examples. This method requires an expert with detailed knowledge about the laws and dependencies of the data. The structure of the model is induced from the knowledge of the expert rather than from the data.

To be independent from (difficult to find) experts and to make it possible to find new structures that were not previously known by experts, machine learning systems were developed that derive a model of the data directly from the set of training examples by data analysis algorithms. Recently, functional decomposition of MVL functions has successfully been applied to solve machine learning problems [49]. The set of training examples is transformed into a discrete function. The input variables are the attributes of the examples and the output variable is the desired classification of the examples. The function is recursively decomposed into small blocks. The result is a structure of interconnected blocks where each block is associated with a function that represents a rule.

This work introduces a new methods of decomposition, therefore some of the existing decomposition theory is introduced here. One method of decomposition is *Curtis-like decomposition*. Originally, Curtis developed a decomposition method for Boolean function [10]. Recently, these ideas have been extended to MVL functions [28], and are called Curtis-like decomposition, where a function is decomposed into smaller decomposition functions by introduction of intermediate variables. The decomposition functions are computed by the decomposition algorithm so that they depend on fewer variables than the original function. The benefit is a small number of decomposition functions because the functions automatically adopt to the structure of the data.

Example 1.2. The data from Table 1.2 can be decomposed into two decomposition functions as shown in Figure 1.2. The value of the intermediate variable $c = 2$ and the value of the goal variable $lens = 2$ for the set of input variables $\{age = 2, prescription = 0, tear = 1\}$ are shown at the connections between the blocks.

By Curtis-like decomposition a structure is introduced that shows rules and relationships between variables. However, the structure may not easily be understandable by humans. The decomposition functions are determined by the decomposition algorithm and can be arbitrary functions. Therefore, their representation still requires tables, even if these tables are much smaller than the original table. After the complete decomposition, these tables only depend on two input or intermediate variables.

For MVL functions, there are many discrete functions with two input variables. Only a few functions (i.e. sum, product, min and max) can easily represented by natural language. Many other functions require complicated explanations or tables to be understood. Therefore, if the solution should be simple from a human point of view, there should be some way to control the decomposition functions.

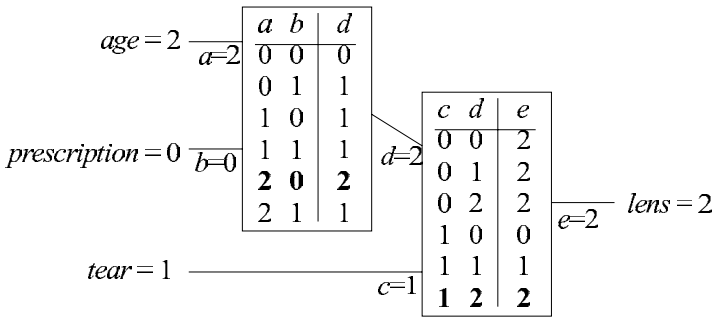

Figure 1.2: Curtis-like decomposition of the lenses prescription rules. with the input variables *age*, *prescription* and *tear* and the output variable *lens*. The variable d represents an intermediate goal.

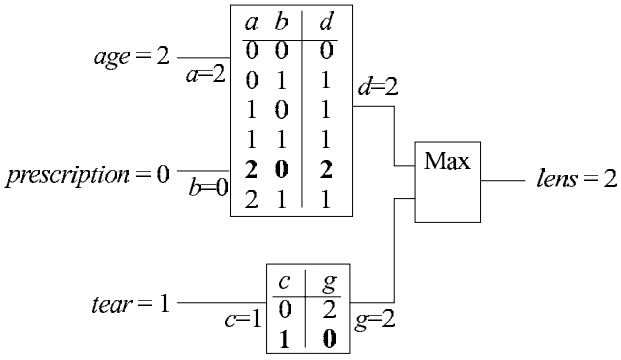

Figure 1.3: Bi-decomposition of the lenses prescription rules.

This work suggests *bi-decomposition* as a way to control the decomposition functions. Bi-decomposition decomposes a function into two decomposition functions that are combined by an operator function. There are different decomposition algorithms for different operator functions. Therefore the operator function can be chosen so that simple to understand decompositions are produced.

Example 1.3. Figure 1.3 shows a bi-decomposition of the function from table 1.2. The operator functions the max-gate. The figure also shows the values of the intermediate and goal variables for the input vector $[age = 2, prescription = 0, tear = 1]$. Note that the original table is split into two smaller tables that are connected by a max-gate. The structure shows that the value $tear = 0$ implies the value $lens = 2$. This translates to the rule "If a patient has a reduced tear production rate, then she should not be prescribed contact lenses". Recursive decomposition of the upper table results in a network that consists of only min- and max-gates, which reveals further rules.

Because bi-decomposition has been shown to give solutions of low complexity in the Boolean case, it can be assumed that bi-decomposition is also well suitable to machine learning applications, provided efficient bi-decomposition algorithms for large MVL functions are known.

1.2 Related Work

Traditionally, most work on decomposition relates to Boolean functions. There are two basic types of decompositions for Boolean functions, Curtis- and bi-decomposition.

Curtis-decomposition has the form $f(A,B,C) = h(\vec{g}(A,C),B,C)$, where A, B and C are disjoint sets of variables; and $\vec{g}(A,C)$ is a vector of functions [10]. The function $h(D,B,C)$ depends on a set of intermediate variables D. The number of variables in D is equal to the size of the vector $\vec{g}(A,C)$. To ensure that the decomposition functions $\vec{g}(A,C)$ and $h(D,B,C)$ depend on fewer variables, it is required that the size of vector $\vec{g}$ be smaller than the number of variables in the set A.

The first who mentioned this type of decomposition was Ashenhurst who only considered the case where $\vec{g}(A,C)$ consists of a single function $g(A,C)$ [1]. A special case of the Ashenhurst decomposition is the Powarov decomposition, where set C of shared variables is the empty set, i.e. $f(A,B) = h(g(A),B)$ [30].

There is a great variety of algorithms that compute Curtis-decompositions. The algorithms can be divided into two classes depending their handling of incompletely specified functions (ISFs). Pedram developed decomposition algorithms by cutting *binary decision diagrams (BDDs)* [15]. BDD cutting algorithms can decompose very large functions, but they are only suitable to decompose completely specified functions. Decomposition of ISFs has a large potential for optimization. However, there are algorithmic difficulties. All known Curtis decomposition algorithms for ISFs depend on the solution of an NP-hard problem whose size grows exponentially with the number of functions in $\vec{g}(A,C)$. Łuba applied set covering algorithms for the decomposition [21], while Perkowski used graph coloring [47]. Perkowski also discussed the possibility of approximation algorithms for the NP-hard problem [22]. He also extended Curits-decomposition to the decomposition of relation and generalized the decomposition by dropping the assumption that the number of variables in function $h(D,C)$ should decrease [27].

Boolean bi-decomposition has the form $f(A,B,C) = g(A,C) \circ h(B,C)$ where '$\circ$' is one of the Boolean operations AND, OR or EXOR. Bi-decomposition has first been mentioned by Böhlau [7]. Boolean differential calculus (BDC), developed by Bochmann and Posthoff [4], has been used to formulate efficient decomposition algorithms for AND- and OR-decomposition by Bochmann and Steinbach [6]. These algorithms have been extended to handle the case of non-decomposable functions. In this case, weak decomposition $f(A,C) = g(A,C) \circ h(C)$ is applied. While there is a closed solution within the BDC for AND- and OR-decomposition, only iterative solutions are possible for EXOR-decompositions [41]. It has been shown that the computation of EXOR-decompositions can be translated into the computation of *Ashenhurst decompositions* and vice versa [18, 16]. Bochmann and Steinbach demonstrate a complete decomposition system using BDC and *lists of ternary vectors (TVLs)* as the central data structure [6].

Zakrevski developed bi-decomposition algorithms by solving systems of Boolean equations [42]. A method of successive removal of values for was developed by Sasao [32]. The separation algorithms by Łuba and Shestakov can also be applied to compute bi-decompositions. In this case however there is no control over the type of operator $\circ$.

Some of the above methods have been extended to MVL functions. The graph-coloring algorithms have been extended to the MVL case in [28]. MVL Curtis-like decomposition and Bi-decomposition using double separation by blanket algebra was shown in [20]. The method of successive value removal also applies to the MVL case [39].

This dissertation develops algorithms for bi-decomposition of MVL functions. It is based on fundamental results in discrete mathematics [5], especially in lattice theory [34] and integer programming [9]. Boolean differential calculus [4] will be extended to the MVL case as needed to develop the results. For a more detailed discussion on MVL differential calculus see [48]. A central part of the dissertation are function sets which were first introduced in [4], and were used to solve Boolean differential equations in [35]. The BEMDD package by Mishchenko [25] was used to verify the algorithms experimentally using machine learning functions from the POLO Internet page [29].

1.3 Contributions of the Dissertation

This dissertation contributes to the following areas of research:

- discrete mathematics
- logic synthesis
- machine learning

The major and original contributions of this dissertation are:

- the application of results from lattice theory, MVL differential calculus and integer programming to MVL logic design problems

- the extension of the concept of function sets from the Boolean to the MVL domain
- the identification of several classes of MVL function sets
- a taxonomy for MVL function sets
- a comparison between MVL and Boolean function sets
- the proof of properties of bi-decompositions for several operators and function sets
- the definition of separation to simplify non-decomposable functions
- a development of two separation methods for MVL function sets
- a development of bi-decomposition algorithms for MVL function sets with respect to several classes of operators
- a development of several strategies for the variable selection problem
- an implementation of the object-oriented decomposition system YADE
- the decomposition of test functions by YADE
- a comparison of several decomposition strategies
- a comparison of the results by YADE with other decomposers

Other contributions include

- an extension of Boolean differential calculus to the MVL domain and the development of properties for the MVL differential operators
- a systematic description of Boolean function sets and decomposition algorithms from the generalized view of multi-valued logic
- a comparison of several strategies for elimination of inessential variables

1.4 An Overview of the Dissertation

The dissertation is organized as follows.

Basic terms and theorems are given in Chapter 2. The chapter defines terms from lattice theory and states elementary theorems. Boolean functions and operators are introduced, and properties of the Boolean derivation operators are given. The discussion of Boolean functions concludes with the introduction of incompletely specified Boolean functions. The second part of Chapter 2 extends the definitions and theorems from Boolean to MVL logic. MVL functions, their operators, some differential operators, ISFs and MVL relations are introduced. The chapter concludes with a brief introduction of data structures for Boolean and MVL functions.

Bi-decomposition algorithms for Boolean ISFs are introduced in Chapter 3. The decomposition theory to find and compute bi-decompositions is presented. The chapter also includes algorithms that recursively decompose Boolean ISFs into two-input gates.

The concept of function sets is extended to the MVL case in Chapter 4. Specific classes of function sets are identified as needed by the decomposition algorithms. Basic properties and a taxonomy of the classes of function sets are shown.

In Chapter 5 properties of MVL bi-decompositions are shown that allow the extension of the bi-decomposition algorithms from Chapter 3 to the MVL domain as well as the construction of new decomposition algorithms. The chapter starts with a brief discussion of bi-decomposition with generic operators. Then, theorems are developed specific classes of operators such as monotone operators, simple operators, min- and max-operators and the modsum-operator. Separation is introduced to simplify functions that are not bi-decomposable.

Various bi-decomposition algorithms are shown in Chapter 6. The properties of function sets from Chapter 4 are applied to simplify the function sets. The bi-decomposition algorithms are based on the results of Chapter 5. Algorithms for the selection of decomposition operators and variable sets are also discussed here.

The extensible and object-oriented decomposition system YADE is documented in Chapter 7. The input of the decomposition system is a set of MVL relations, which are decomposed into a network of gates. The set of gates and selection strategies for the types of decomposition can be configured as parameters of YADE.

Experimental results of decomposition of functions from the machine learning domain are presented in Chapter 8.

The dissertation concludes with a summary of the proposed decomposition methods and the results in Chapter 9. The chapter also provides ideas for further work within this area.

Chapter 2

Mathematical Foundation

2.1 Basic Lattice Theory

2.1.1 Partially Ordered Sets

Some definitions and properties of partially ordered sets and lattices are introduced here. A good summary of lattice theory can be found in [34].

Sets of functions will frequently be used throughout the next chapters. Therefore, there must be a way to describe the structure of sets. The easiest method to introduce an order is a relation, say $\leq$, that is a *partial ordering*. The elements of the set can then be compared by this relation. A relation $\leq$ is a *partial order* if it is

- *reflexive*, i.e. $a \leq a$,
- *antisymmetric*, i.e. $a \leq b$ and $b \leq a$ implies $a = b$ and
- *transitive*, i.e. $a \leq b$ and $b \leq c$ implies $a \leq c$.

A set P and a partial ordering relation $\leq$ over P are called a *partially ordered set*. A partially ordered set can be visualized by a chart. The chart is a graph with one node for each element of the set and a directed edge from element a to b for each pair (a, b) with $a \leq b$. By convention, reflexive and transitive edges are omitted. An example of a partially ordered set $S = \{A, B, C, D, E\}$ with the ordering relation $\leq$ is shown in Figure 2.1.

The *dual relation* of $\leq$, the relation $\geq$, is defined by $a \geq b$ if$_{\text{DEF}}$ $b \leq a$. It is easy to see that if relation $\leq$ is a partial ordering, then relation $\geq$ is also a partial ordering [34]. For every theorem about partially ordered sets there is a *dual theorem* in terms of the dual relation. The poof of a dual theorem is analogous to the proof of the original theorem, with the partial ordering relation and all lemmas replaced by their dual counterparts. Therefore, the dual form of theorems will only be stated and in the proof the duality principle is cited.

Let P be a partially ordered set. The following terms are defined for P.

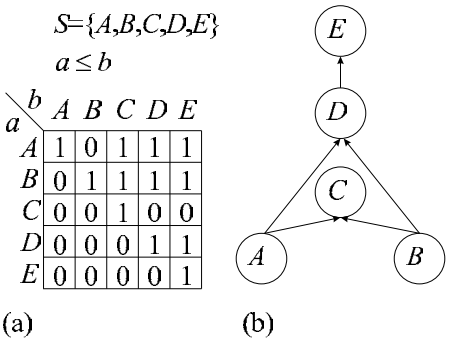

Figure 2.1: Partially ordered sets. **(a)** Partial ordering relation. **(b)** Chart of the partially ordered set.

Incomparable Elements. Set P may have elements $a, b \in P$ that have neither $a \leq b$ nor $b \leq a$. Such elements are *incomparable.*

Totally Ordered Set. A set P is *totally ordered* if$_{\text{DEF}}$ the set P does not contain incomparable elements.

Interval. Let $a, b \in P$ be elements with $a \leq b$. The set $[a, b] := \{x \mid a \leq x \leq b\}$ is an *interval* of P.

Lower Bound. Let $Q \subseteq P$, $Q \neq \emptyset$. An element $a \in P$ is a *lower bound* of Q if $a \leq x$ for all elements $x \in Q$.

Greatest Lower Bound. The element b is the *greatest lower bound* of $Q \subseteq P$, $Q \neq \emptyset$ if element b is a lower bound of Q and if for every lower bound v of Q there is $v \leq b$. The greatest lower bound of Q is denoted by $b = \inf Q$.

Upper Bound. Let Q be a nonempty subset of the set P. An element $a \in P$ is an *upper bound* of Q if $a \geq x$ for all elements $x \in Q$.

Least Upper Bound. The element b is the *least upper bound* of $Q \subseteq P$, $Q \neq \emptyset$ if element b is an upper bound of Q and if for every upper bound v of Q there is $v \leq b$. The least upper bound of Q is denoted by $b = \sup Q$.

Example 2.1. Consider the partially ordered set $S = \{A, B, C, D, E\}$ from Figure 2.1. There is the interval $[A, E] = \{A, D, E\}$ because only D has $A \leq D \leq E$. The greatest lower bound of the set $\{A, D\}$ is $A = \inf\{A, D\}$ because A is the only element with $A \leq A$ and $A \leq D$. The element $\sup\{A, B\}$ does not exist. Although elements C, D and E are upper bounds of $\{A, B\}$, none of these elements is a least upper bound because D and C are incomparable, there is neither $C \geq D$ nor $D \geq C$. Therefore, neither C nor D is a least upper bound of $\{A, B\}$. Similarly, E is incomparable to C and cannot be the least upper bound.

A *partition* divides a set into subsets.

Definition 1. A *partition* of a set A is a set of sets $\{A_1, A_2, \ldots, A_n\}$ so that each element $a \in A$ is element of exactly one set $a \in A_i, 1 \leq i \leq n$.

2.1.2 Lattices

Among rings and groups, lattices are a very general algebraic structure in discrete mathematics. Throughout this work, lattices will be used to show structures of discrete functions.

Definition 2. A *lattice* is a partially ordered set, where for every pair of two elements there exist the greatest lower bound and the least upper bound. Two operations can be introduced for the elements of lattices, the *product* or *intersection* $a \cdot b := \inf\{a, b\}$ and the *sum* or *union* $a + b := \sup\{a, b\}$, where $\{a, b\}$ denotes the two-element set that constists of the elements a and b.

An example of a lattice is any subset of integers ordered by the size of the integers.

Lemma 2.1. *Let S be an arbitrary subset of the set of all integers, and let S be ordered by the natural order of integers. Then, the set S is a lattice with* $\inf(a, b) = \min(a, b)$ *as the product operation and* $\sup(a, b) = \max(a, b)$ *as the sum operation.*

Proof. Let a and b be two arbitrary elements with $a, b \in S$. Because of $\min(a, b) \leq a$ and $\min(a, b) \leq b$, the element $\min(a, b)$ is a lower bound of the set $\{a, b\}$. For every element $l \in S$ with $l \leq a$ and $l \leq b$ there is $l \leq \min(a, b)$. Therefore, $\min(a, b)$ is a greatest lower bound of the set $\{a, b\}$. Dual arguments show that $\max(a, b)$ is a least upper bound of $\{a, b\}$. Because $\min(a, b) \in S$ and $\max(a, b) \in S$, the set S is a lattice. □

It is easy to show that the product and sum operation defined by Definition 2 have the following three properties [34]:

1. the *commutative laws*, $a \cdot b = b \cdot a$ and $a + b = b + a$,

2. the *associative laws*, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ and $(a + b) + c = a + (b + c)$ and

3. the *absorption law* $a \cdot (a + b) = a + a \cdot b = a$.

In fact, it is possible to define a lattice by these laws without using the term of partially ordered set.

Theorem 1. *Let L be a set with two operations $+$ and $\cdot$ on its elements that satisfy the commutative, associative and the absorption laws. Then, the relation $a \leq b := (a + b = b)$ defines a partial ordering in L, and the partially ordered set is a lattice. Furthermore, $a + b = \sup\{a, b\}$ and $a \cdot b = \inf\{a, b\}$.*

Proof. The proof can be found in [34]. □

Unfortunately, the term lattice has also been used to describe a regular arrangement of points. It is important not to confuse these two meanings. In this work, a lattice always denotes a partially ordered set with above mentioned properties.

Several terms related to lattices will be used throughout this work.

Sublattice. A nonempty subset P of the lattice L is a *sublattice* if for all elements $a, b \in P$ there is $a \cdot b \in P$ and $a + b \in P$.

Convex Sublattice. A sublattice P of the lattice L is *convex* if for every $a, b \in P$, $a \leq b$ there is $[a, b] \subseteq P$.

Distributive Lattice. A lattice L is *distributive* if for all elements $a, b, c \in L$ there is $(a+b) \cdot c = a \cdot c + b \cdot c$.

Zero-Element. A *zero-element* o is an element with $a \cdot o = o$ and $a + o = a$ for all elements $a \in L$ of the lattice L.

One-Element. The *one-element* e satisfies $a \cdot e = a$ and $a + e = e$ for all elements $a \in L$ of the lattice L.

Complemented Lattice. A lattice L is *complemented* if it has an one- and a zero-element, and if each element $a \in L$ has a complement $\overline{a}$ so that $a \cdot \overline{a} = o$ and $a + \overline{a} = e$.

Boolean Lattice. A *Boolean lattice* is a complemented and distributive lattice.

Although the term Boolean algebra is widely used to denote a Boolean lattice, the structure is a lattice, and the term Boolean lattice will be preferred.

The lemma below shows that the lattice from Lemma 2.1 is a distributive lattice.

Theorem 2. *Let S be an arbitrary subset of the set of all integers, and let S be ordered by the natural order of integers. Then, the set S is a distributive lattice with $\inf(a, b) = \min(a, b)$ as the product operation and $\sup(a, b) = \max(a, b)$ as the sum operation.*

Proof. Because it was shown in Lemma 2.1 that S is a lattice, it remains to show that the min- and max-operator satisfy the distributive law

$$\min(\max(a, b), c) = \max(\min(a, c), \min(b, c)) \tag{2.1}$$

for all elements $a, b, c \in S$. Two cases are distinguished on the value of $\max(a, b)$.

1. If $\max(a, b) \leq c$ there is $\min(\max(a, b), c) = \max(a, b)$. Furthermore, there is $a \leq c$ and $b \leq c$, which implies $\min(a, c) = a$ and $\min(b, c) = b$. Therefore, there is

$$\max(\min(a, c), \min(b, c)) = \max(a, b), \tag{2.2}$$

   and (2.1) holds.

2. Otherwise there is $c < \max(a, b)$ which implies $\min(\max(a, b), c) = c$. Without loss of generality can be assumed that $a \leq b$. Then, there is $c < \max(a, b) = b$, and hence, $\min(b, c) = c$. Because of $\min(a, c) \leq c$ there is $\max(\min(a, c), \min(b, c)) = c$, and (2.1) holds.

From the cases 1. and 2. can be concluded that (2.1) is satisfied for all $a, b, c \in S$. □

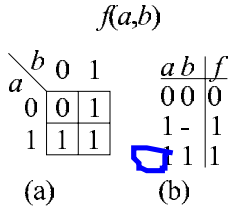

Figure 2.2: Representation of the Boolean function $f(a,b) = a \vee b$. **(a)** Displayed as a Karaugh-Map. **(b)** Displayed as a Table.

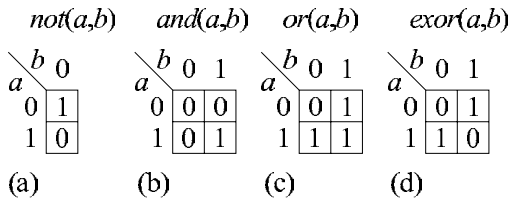

Figure 2.3: Boolean composition operators. **(a)** NOT-operator. **(b)** AND-operator. **(c)** OR-operator. **(d)** EXOR-operator.

2.2 Boolean Functions and Operations

2.2.1 Definition of Boolean Functions

Because of their simplicity, Boolean functions have found many applications in the design of combinatorial logic, finite state machines and other areas.

A *Boolean variable* is a variable that can assume any value from the set $\mathbb{B} = \{0,1\}$, the one-dimensional *Boolean space*. A set $A = \{a_1, a_2, \ldots, a_n\}$ of Boolean variables can assume any element from the set $\mathbb{B}^n$, the n-dimensional *Boolean space*. An element of $\mathbb{B}^n$ is a *Boolean minterm*. The Boolean space $\mathbb{B}^n$ contains $|\mathbb{B}^n| = 2^n$ minterms.

Definition 3. Let A be a set of n Boolean variables. A *Boolean function* $f(A)$ is a many-to-one mapping $\mathbb{B}^n \rightarrow \{0,1\}$ from the Boolean space of n variables into the one-dimensional Boolean space. The set of variables A is called the *support* of the function. The set of minterms A_i with $f(A_i) = 0$ is called *OFF-set*, the set of minterms A_j with $f(A_j) = 1$ is called *ON-set* of $f(A)$.

The set of all Boolean functions that depend on the set of Boolean variables A is denoted by $\mathbb{F}_2(A)$. The index 2 denotes the two valued output of the Boolean function. If the set A contains $|A| = n$ variables there are

$$|\mathbb{F}_2(A)| = 2^{|\mathbb{B}^n|} = 2^{2^n}$$

Boolean functions that depend on A. If A consist of a single variable a for example, there are $2^{2^1} = 4$ Boolean functions from a, namely 0, 1, a and $\overline{a}$.

Boolean functions can be represented as tables of function values, or as Karnaugh maps (see Figure 2.2). Rows of a table can be merged if they differ only in the value of a single input variable. The merged row contains a '-' for the input variable that was different.

2.2.2 Boolean Composition Operators

New functions can be created from existing functions by operators. Well known for Boolean functions are the negation $\overline{f(A)}$, the OR-operator $f(A) \vee g(A)$, the AND-operator $f(A) \wedge g(A)$ and the EXOR-operator $f(A) \oplus g(A)$. These operators are called *composition operators* because they can be defined in terms of an operator function (see Figure 2.3). The result of a composition operator is the composition of the operator function with the argument functions, for instance, $f(A) \vee g(A) = or(f(A), g(A))$, where the function $or(x,y) = x \vee y$ is the composition function shown in Figure 2.3(c).

Theorem 3. *The set $\mathbb{F}_2(A)$ of all Boolean functions $f(A)$, and the AND-, OR- and NOT-operator form a Boolean lattice.*

Proof. The properties of the Boolean lattice can easily be derived from the laws of the Boolean operators [4]. □

It follows from Theorems 1 and 3 that the relation

$$f(A) \leq g(A) \;:=\; (f(A) = f(A) \wedge g(A)) \tag{2.3}$$

is a partial ordering of the set of all Boolean functions $f(A)$.

2.2.3 Derivation Operators

Composition operators transform a single value of the argument function into a single value of the result function. The value of $\overline{f(1,0,1)}$ for instance, can be computed from the value of $f(1,0,1)$; no other value of $f(A)$ must be known.

The *Boolean differential calculus (BDC)* defines operators where the values of the result function, depend on more than a single value of the argument function. Several operators have been defined and applied to describe problems from various areas including logic and finite state machine design, circuit testing, graph processing and many others. For an overview on Boolean differential calculus see [4, 6].

The BDC includes two types of operators, differential and derivation operators. Differential operators introduce new variables (the differentials) into the result function. The values of the differentials encode the direction of the change between the function values. In contrast, the results of the derivation operators are computed for a fixed direction of change between the function values, and the the result of the derivation operators only depends on a subset of the variables of the argument function [4]. Throughout this work only the derivation operators are applied and introduced here.

Definition 4. The *maximum* of a function $f(A,b)$ over a variable b is defined as

$$\max_b f(A,b) := f(A,b=0) \vee f(A,b=1). \tag{2.4}$$

The *k-times maximum* of a function $f(A,B)$ over a set of variables $B = \{b_1, b_2, \ldots, b_k\}$ is defined as

$$\max_B^{\mathrm{k}} f(A,B) := \max_{b_1}(\max_{b_2}(\ldots \max_{b_k} f(A,B)\ldots)). \tag{2.5}$$

To save space, the variables of the k-times derivation operators are written as an index to the operator if the operator is used in a formula embedded into text, i.e. $\max^{\mathrm{k}}{}_B f(A,B)$. The variables are centered below the operator if the formula is displayed on a separate line (2.5). The meaning of both notations is the same.

Note that $f(A,b=0)$ and $f(A,b=1)$ do not depend on the variable b. Therefore, $\max_b f(A,b)$ does not depend on the variable b, and $\max^{\mathrm{k}}{}_B f(A,B)$ does not depend on the set of variables b. This means the function $\max^{\mathrm{k}}{}_B f(A,B)$ does not contain the variables B anymore. The function $\max^{\mathrm{k}}{}_B f(A,B)$ describes the largest value of $f(A,B)$ in the Boolean subspace defined by B.

A minimum operator that finds the minimal value of $f(A,B)$ in the subspace of B can be defined dually.

Definition 5. The *minimum* of a function $f(A,b)$ over a variable b is defined as

$$\min_b f(A,b) := f(A,b=0) \wedge f(A,b=1). \tag{2.6}$$

The *k-times minimum* of a function $f(A,B)$ over a set of variables $B = \{b_1, b_2, \ldots, b_k\}$ is defined as

$$\min_B^{\mathrm{k}} f(A,B) := \min_{b_1}(\min_{b_2}(\ldots \min_{b_k} f(A,B)\ldots)). \tag{2.7}$$

The change of function values is indicated by the Boolean derivation

Definition 6. The *Boolean derivation* of a function $f(A,b)$ over a variable b is defined as

$$\frac{\partial f(A,b)}{\partial b} := f(A,b=0) \oplus f(A,b=1). \tag{2.8}$$

			Characteristic function of		
	ISF $F(A)$	Set of minterms A	ON-set $q(A)$	OFF-set $r(A)$	don't care-set $\varphi(A)$
min-terms A_i	0	OFF-set	0	1	0
	1	ON-set	1	0	0
	Φ	DC-set	0	0	1

Figure 2.4: Characterization of ISFs.

2.2.4 Incompletely Specified Boolean Functions

Some applications do not specify the values of functions for all possible inputs. It may be known for instance, that some inputs cannot occur in a particular application. The values of a function for these inputs are of no interest and can be chosen arbitrarily to optimize the design. The functions of these applications can be described within the theory of *Incompletely Specified Functions (ISF)*. There are many applications for ISFs in synthesis and minimization of systems. In finite state machines, for instance, not all states of the flip-flops are reachable, and the value of the transition function for these states are of no interest. The design of combinatorial logic combines functions by OR- and AND-gates. If one function of an OR-gate has a value of 1, the values of the other functions do not change the output value of the gate and can be chosen arbitrarily.

Although ISFs are frequently used, there are various definitions for ISFs. This section will give a brief introduction into the dependencies between these definitions.

Definition 7. An *ISF* $F(A)$ is a mapping $C \rightarrow \{0,1\}$ from a subset $C \subseteq \mathbb{B}^n$ of the n-dimensional Boolean space into the one-dimensional Boolean space. The set C is called *care set*, the complement set $D = \mathbb{B}^n \setminus C$ is the *don't care set (DC-set)*. The set of minterms A_i with $F(A_i) = 1$ is the *ON-set*, the set of minterms A_j with $F(A_j) = 0$ is the *OFF-set* of the ISF $F(A)$.

Because the union of OFF-set, ON-set and DC-set is the n-dimensional Boolean space $\mathbb{B}^n$ and the sets do not overlap each other, only two of the sets are needed to specify an ISF. ISFs can be visualized by Karnaugh-maps and tables similar to Boolean functions. The elements of the ON-set are denoted by 1, the elements of the OFF-set by 0 and the elements of the DC-set by the *don't care symbol* Φ.

An ISF can also be specified by the *characteristic functions* of ON-set, OFF-set, and DC-set. The characteristic function of a set L of minterms is a Boolean function

$$f(A_i) = \begin{cases} 1 & \text{for } A_i \in L, \\ 0 & \text{otherwise} \end{cases}$$

Three functions can be defined for an ISF:

1. The characteristic function of the ON-set $q(A)$.
2. The characteristic function of the OFF-set $r(A)$.
3. The characteristic function of the DC-set $\varphi(A)$.

The values of an ISF, the corresponding sets of minterms, and their characteristic functions are shown in Figure 2.4. An ISF describes a characteristic set of Boolean functions that have the same values for the care set of the ISF.

Definition 8. The characteristic function set $\mathbf{F}(A)$ of an ISF $F(A)$ with the care set C is the set of all Boolean functions $f(A)$ with $f(A_i) = F(A_i)$ for all minterms $A_i \in C$.

It is necessary to distinguish between ISFs $F(A)$ which is a partially defined function and its characteristic function set $\mathbf{F}(A)$ which is a set of Boolean functions. ISFs are always denoted by slanted capital letters $F(A)$, while its characteristic function set $\mathbf{F}(A)$ is denoted by the same letter printed in bold face.

A function that is element of the characteristic function set of an ISF is called *member function* $f(A) \in \mathbf{F}(A)$ of the ISF. The characteristic function set of an ISF is not an arbitrary subset of the set of all Boolean functions, but has a well-defined structure.

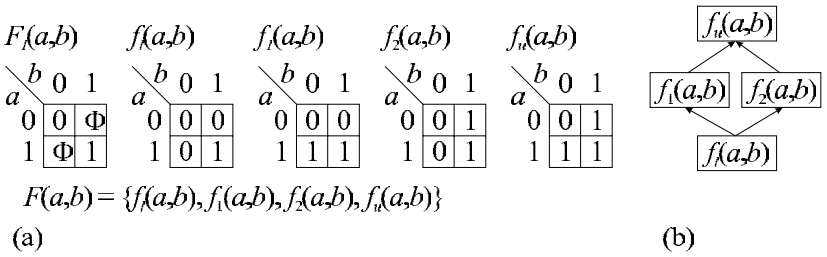

Figure 2.5: Example of a Boolean ISF. **(a)** ISF $F_I(a,b)$, its lower bound $q(a,b)$, upper bound $p(a,b)$, and member functions $f_1(a,b)$ and $f_2(a,b)$. **(b)** Boolean lattice of the characteristic function set.

Theorem 4. *The characteristic function set* $\mathbf{F}(A)$ *of an ISF* $F(A)$ *forms a Boolean lattice with the negation of Boolean functions as complement operation, and the Boolean disjunction and conjunction as the lattice sum and product operation respectively. Furthermore, the characteristic function set* $\mathbf{F}(A)$ *is a convex sublattice of the lattice of all Boolean functions,* $\mathbf{F}(A) = [f_l(A), f_u(A)]$ *where the lower bound* $f_l(A) = q(A)$ *is the characteristic function of the ON-set and the upper bound* $f_u(A) = \overline{r(A)}$ *is the complement of the characteristic function of the OFF-set of the ISF* $F(A)$*.*

Proof. The properties of the Boolean lattice follow directly from the properties of the Boolean operations [4].

It remains to show that the sublattice is convex. Let $\mathbf{I}(A) = [q(A), \overline{r(A)}]$. For any function $f(A)$ there is $f(A) \in I(A)$ if and only if $q(A) \leq f(A) \leq \overline{r(A)}$. Two cases are distinguished on the minterms A_i:

1. If A_i is an element of the don't care set of $F(A)$, the inequality $q(A_i) = 0 \leq f(A_i) \leq 1 = \overline{r(A_i)}$ is always satisfied.

2. Otherwise A_i is an element of the care set of $F(A)$ and there is $q(A_i) = \overline{r(A_i)} = F(A_i)$.

From the cases 1. and 2. follows that $f(A) \in I(A)$ if and only if $f(A_i) = F(A_i)$ for all minterms A_i of the care set of $F(A)$ which implies that $f(A) \in \mathbf{F}(A)$. It follows that $\mathbf{I}(A) = \mathbf{F}(A)$. □

The variables of the interval bounds are omitted in the definition of an interval if the interval bounds depend on the same variables as the interval, i.e. $\mathbf{F}(A) = [f_l(A), f_u(A)]$ is abbreviated by $\mathbf{F}(A) = [f_l, f_u]$.

For ISFs a value variable can be introduced. The Boolean ISF can be described as a relation between the variables of the ISF and its value variable. In [4] this is called a phase list. This idea is extended in Section 2.3.5 to MVL relations.

It can be summarized that there are four different definitions of Boolean ISFs:

1. a partially defined Boolean function,

2. an interval of functions,

3. a relation between Boolean minterms and a value variable and

4. a lattice of Boolean functions.

Example 2.2. Figure 2.5(a) shows an ISF $F_I(a,b)$ with the lower bound $q(a,b)$ and the upper bound $p(a,b)$. The characteristic function set consists of the four functions $q(a,b)$, $p(a,b)$, $f_1(a,b)$ and $f_2(a,b)$ as shown in the figure. The Boolean lattice of these functions is drawn in Figure 2.5(b).

These definitions of Boolean ISFs lead to four different classes of multi-valued function sets in Chapter 4.

2.3 Multi-Valued Functions and Operations

2.3.1 Definition of Multi-Valued Functions

Signals in digital circuits have exactly one of two values. Therefore, Boolean functions are ideal for modeling these systems. In some applications, such as machine learning or multi-valued logic (MVL) synthesis, functions with more than two values are applied. To model these systems, MVL functions, where more than two values can be assigned to the variables, are necessary.

It is always possible to employ a vector of Boolean functions instead of an MVL function by a proper encoding of the values. However, there are some problems with this method.

- The Boolean representation introduces operations that cannot be interpreted in the MVL domain. Decomposition is an example. Boolean variables that represent a single MVL variable can be separated by Boolean decomposition. This decomposition cannot be translated into a corresponding MVL decomposition.
- Which encoding of the values to Boolean vectors should be chosen? Some coding schemes will introduce unnecessary complexity into the system.
- The number of values of MVL variables is, in general, not a power of two. Therefore, some values will not be used and there is the question of what to do with these values.
- In some cases, as in machine learning, it is very important to produce solutions that can easily be understood by humans. The solution produced by replacing MVL by Boolean functions will be in terms of Boolean operators. These operators may not be easily understood in the context of the MVL system.

To avoid these problems, the theory of Boolean functions has been extended to MVL functions, which are a generalization of two-valued Boolean functions. Some properties of Boolean functions cannot be generalized to MVL functions. However, as this work will show, many properties of Boolean functions can be directly extended to the MVL case. Also, investigation of MVL systems emphasizes some properties that were not very apparent in Boolean systems.

A Boolean variable can only assume one of two values. For an MVL variable the number of possible values must be defined. An MVL variable a can assume any value from the set $\{0, 1, \ldots, m-2, m-1\}$, where the integer $m \geq 2$ is called the *cardinality* of the variable a. A set $A = \{a_1, a_2, \ldots, a_n\}$ of MVL variables, with cardinalities $m_1, m_2, \ldots, m_n$ respectively, can assume any element of

$$\mathbb{M}_A = \{0, \ldots, m_1 - 1\} \times \{0, \ldots, m_2 - 1\} \times \ldots \{0, \ldots, m_n - 1\}, \tag{2.9}$$

where $\times$ denotes the Cartesian product of sets. The set $\mathbb{M}_A$ is the *MVL space* of the set of variables A. An element of $\mathbb{M}_A$ is an *MVL minterm*. The MVL space $\mathbb{M}_A$ has $|\mathbb{M}_A| = m_1 * m_2 * \cdots * m_n$ minterms.

Sometimes, the cardinalities of all variables are assumed to be equal and constant. This assumption is called *fixed cardinality* and enables elegant theoretical formulations. However, this assumption is only valid for some applications, like MVL logic. In machine learning the cardinalities of attributes differ from each other. Therefore, in this work cardinalities of variables are assumed to be different, which is called *variable cardinality*. However, for a single variable, its cardinality is assumed to be constant. The case of fixed cardinality can be treated as a special case of variable cardinality.

It is useful to define a partial order over the set $\mathbb{M}_A$ of all minterms.

Definition 9. Assume $A = \{a_1, a_2, \ldots, a_n\}$ and let $A_i, A_j \in \mathbb{M}_A$ be minterms with $A_i = \{a_{1i}, a_{2i}, \ldots, a_{ni}\}$ and $A_j = \{a_{1j}, a_{2j}, \ldots, a_{nj}\}$. Then $A_i \leq A_j$ if$_{\text{DEF}}$ $a_{ki} \leq a_{kj}$ for all $k = 1, 2, \ldots, n$.

The definition of MVL functions is a direct extension of Definition 3:

Definition 10. Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of MVL variables with the cardinalities $m_1, m_2, \ldots, m_n$. An *MVL function* $f(A)$ is mapping

$$\mathbb{M}_A \rightarrow \{0, \ldots, m_o - 1\} \tag{2.10}$$

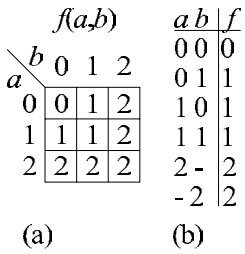

Figure 2.6: MVL-functions. (a) Display as a map. (b) Display as a table.

from the set of minterms $\mathbb{M}_A$ into the set of integers $\{0, 1, \ldots, m_o - 1\}$ The integers $m_i, i = 1, \ldots, n$ are called the *input cardinalities* of the variables a_i, and m_o is called the *output cardinality* of the function $f(A)$.

The set of all MVL functions that depend on the set of MVL variables A and have the output cardinality m_o is denoted by $\mathbb{F}_{m_o}(A)$. If set A contains $|A| = n$ variables, with cardinalities $m_1, m_2, \ldots, m_n$, there are

$$|\mathbb{F}_{m_o}(A)| = m_o{}^{|\mathbb{M}_A|} = m_o^{m_1 * m_2 * \cdots * m_n} \tag{2.11}$$

MVL functions with the output cardinality m_o that depend on A. Like Boolean functions, MVL functions can be visualized by maps and tables (see Figure 2.6). In contrast to Karnaugh maps, the order of minterms need not be gray code for the maps of MVL functions. A dash '-' in the column of the variable a of a table stands for any value in the range $[0, m_a - 1]$ where m_a is the cardinality of the variable a. In Figure 2.6(b) for instance, the last row, '-2 2' abbreviates the three rows '02 2', '12 2' and '22 2'. Note, that a dash can only be used to abbreviate minterms that exist for all values of the variable a. Two minterms '02 2' and '22 2' for instance cannot be replaced by a row containing a dash.

A simple measure of the complexity of an MVL function is the discrete function cardinality .

Definition 11. The *discrete function cardinality (DFC)* $D(f)$ of an MVL function $f(a_1, \ldots, a_n)$ is the product

$$D(f) = m_1 * m_2 * \cdots * m_n \tag{2.12}$$

of the cardinalities m_i of the input variables a_i, $i = 1, \ldots, n$.

The DFC of the function $f(a, b)$ from Figure 2.6 is $D(f) = 3 * 3 = 9$.

2.3.2 Composition Operators

New MVL functions can be created from existing functions by composition. In the Boolean case there are only two functions that depend on one variable, the identity function and negation. Hence, there are only two unary composition operators, identity x and negation $\overline{x}$. For MVL functions, the number of functions depends on the input and output cardinalities. Therefore, there are many unary composition operators. In this work, unary MVL functions are called *literal functions*. The following literal functions were defined in [48], where m_o is the output cardinality:

- the *complement* operator $\overline{x} := (m_o - 1) - x$,
- the *r-order cylic inversion* $\overset{r}{x} := x + r \pmod{m_o}$,
- the *window literal* ${}^{a}x^{b} := \begin{cases} m_o - 1 & \text{for } a \le x \le b, \\ 0 & \text{otherwise.} \end{cases}$

Observe that operators depend on the output cardinality m_o of the composed function. The output cardinality must be given in order to apply these operators. Examples of the unary composition operator functions for the output cardinality $m_o = 3$ are depicted in Figure 2.7. Note, that the complement operator is identical with the Boolean negation if input and output cardinalities are 2. Many other unary operators could be defined. The number of binary operators is even greater. Examples for binary operators from [48] are:

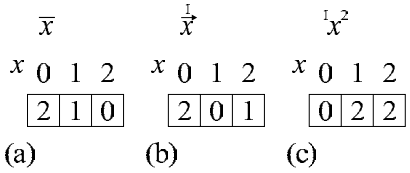

Figure 2.7: Unary MVL-operators. (a) Complement. (b) r-order cyclic inversion. (c) Window literal.

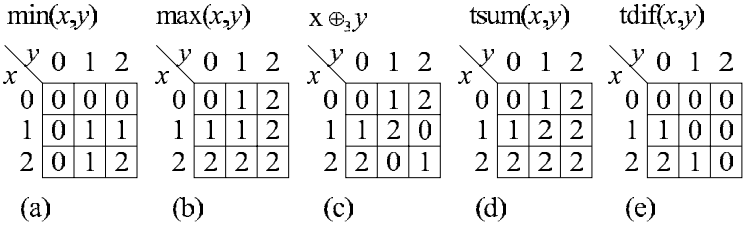

Figure 2.8: Binary MVL-operators from [48]. **(a)** Min-operator. **(b)** Max-operator. **(c)** Modsum-operator. **(d)** Truncated-sum-operator. **(e)** Truncated-difference-operator.

- the arithmetic operators $+$ (addition) and $*$ (multiplication) for integers,
- the *min-operator* $\min(x, y)$,
- the *max-operator* $\max(x, y)$,
- the *modsum-operator* $x \oplus_{m_o} y = (x + y) \pmod{m_o}$,
- the *truncated-sum-operator* $\text{tsum}_{m_o-1}(x, y) = \min(x + y, m_o - 1)$,
- the *truncated-difference-operator* $\text{tdif}(x, y) = \max(x - y, 0)$.

The MVL-functions for these operators are displayed for output cardinality $m_o = 3$ in Figure 2.8. Other operators include:

- the *truncated-product-operator* (see [39]): $\text{tprod}_{m_o-1}(x, y) = \min(x * y, m_o - 1)$,
- the *moddif-operator* $x \ominus_{m_o} y = (x - y + m_o) \pmod{m_o}$,
- the *less-than-or-equal-operator* from [39]: $\text{leq}(x, y) = \begin{cases} 1 & \text{for } x \leq y, \\ 0 & \text{otherwise,} \end{cases}$
- the *equal-operator*: $\text{eq}(x, y) = \begin{cases} 1 & \text{for } x = y, \\ 0 & \text{otherwise,} \end{cases}$
- the *if-less-or-equal-then-zero-operator* $\text{leq}_0(x, y)$ $\text{leq}_0(x, y) = \begin{cases} 0 & \text{for } x \leq y, \\ x & \text{otherwise,} \end{cases}$
- the *if-greater-or-equal-then-m-operator* $\text{geq}_m(x, y)$ $\text{geq}_m(x, y) = \begin{cases} m & \text{for } x \geq y, \\ x & \text{otherwise.} \end{cases}$

The functions of these operators, for the output cardinality $m_o = 3$, are shown in Figure 2.9. Note, that the moddif-operator is the inverse of the modsum-operator. However, the truncated-difference-operator is not the inverse of the truncated-sum-operator. The operators $\text{leq}_0(x, y)$ and $\text{geq}_m(x, y)$ are applied in the computation of decomposition problems in Sections 5.5 and 5.6.

If operators are applied to fixed cardinality, the output cardinality of the operator must equal its input cardinality. Examples for such operators are truncated product, truncated sum, modsum, min or max. Variable cardinality does not restrict the output cardinality of the operators, and operators such as sum or product can also be used.

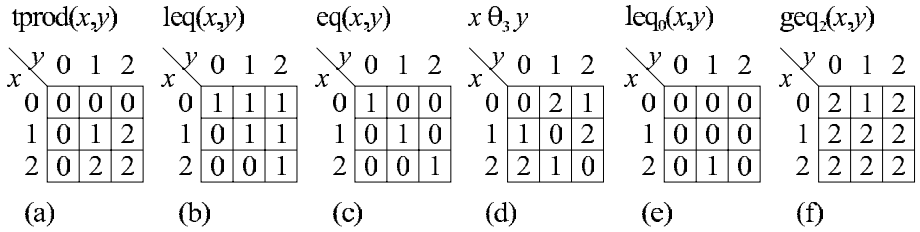

Figure 2.9: Binary MVL-Operators. **(a)** Truncated-product-operator. **(b)** Less-than-or-equal-operator. **(c)** Equal-operator. **(d)** Moddif-operator.**(e)** If-Less-or-equal-then-zero-operator. **(f)** If-greater-or-equal-then-m-operator.

An n-ary operator is the ite-*Operator*, which is an MVL generalization of the Boolean if-then-else operator. The ite-operator selects one of the values from the second to the n-th argument according to the value of the first argument

$$\text{ite}(x, y_0, y_1, \ldots, y_{n-2}) = y_x.$$

It is important to distinguish between the predicate $f(A) \leq g(A)$ for MVL-functions and the binary operator $\text{leq}(f(A), g(A))$. The former is either true or false, whereas the latter is an MVL-function with the output cardinality $m_o = 2$. The result of $\text{leq}(f(A), g(A))$ is still not a Boolean function because the variables of A are not necessarily Boolean variables, i.e. the variables can have an input cardinality of $m_i > 2$.

For the special case of input and output cardinalities of 2, the min-operator becomes the Boolean AND, the max-operator becomes the Boolean OR and the modsum-operator becomes the Boolean EXOR.

The Boolean operators AND, OR, NOT satisfy the laws of a Boolean lattice [4]. For the MVL operators min and max it was shown in Theorem 2 that they satisfy the laws of a distributive lattice. Because the cardinality of MVL variables is in general not a power of 2, the MVL operators in general do not satisfy the laws of a Boolean lattice. However, the properties of the distributive lattice can be extended to MVL functions.

Theorem 5. *The min- and max-operators for MVL functions satisfy the commutative, associative, absorption and distributive laws.*

Proof. It follows from the Theorems 1 and 2 that the min- and max-operator satisfy the commutative, associative, absorption and distributive laws for arbitrary subsets of integers. Let $g(A)$ and $h(A)$ be two arbitrary MVL functions. Then, these laws are satisfied for the integers $g(A_i)$ and $h(A_i)$, where A_i is an arbitrary minterm. Therefore, these laws are also satisfied for the composition of the function $g(A)$ and $h(A)$ by the min- and max-operators. □

All of the associative binary operators can be extended to n-ary operators by repeated application of the binary operator. For the max-operator, for instance, there is $\max(x_1, x_2, \ldots, x_n) = \max(x_1, \max(x_2, \ldots \max(x_{n-1}, x_n) \ldots))$.

2.3.3 MVL Derivation Operators

There are several ways to extend the BDC to MVL functions [48]. Decomposition problems can easily be described in terms of the max- and min-operators over variables.

Definition 12. The *minimum* of an MVL function $f(A, b)$ over the variable b with the cardinality m_b is defined as the MVL function

$$\min_b f(A, b) := \min(f(A, 0), f(A, 1), \ldots, f(A, m_b - 1)). \tag{2.13}$$

The *k-times minimum* of a function $f(A, B)$ over a set of variables $B = \{b_1, b_2, \ldots, b_k\}$ is defined as the MVL function

$$\min_B^k f(A, B) := \min_{b_1}(\min_{b_2}(\ldots \min_{b_k} f(A, B) \ldots)). \tag{2.14}$$

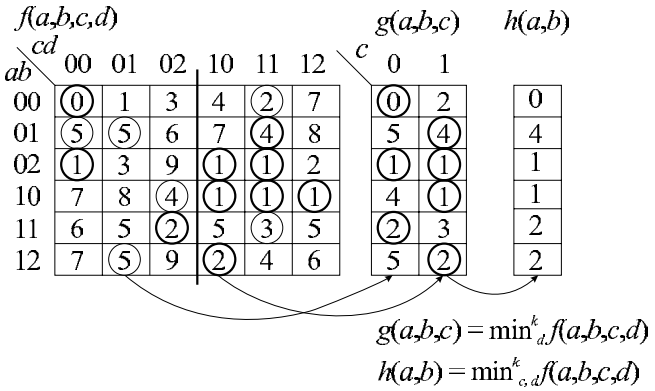

Figure 2.10: Minimum operator of an MVL function. The values of $g(a,b,c)$ are indicated by circles in $f(a,b,c,d)$. The values of $h(a,b)$ are indicated by bold circles in $f(a,b,c,d)$ and circles in $g(a,b,c)$.

Definition 13. The *maximum* of an MVL function $f(A,b)$ over the variable b with the cardinality m_b is defined as the MVL function

$$g(A) = \max_b f(A,b) = \max(f(A,0), f(A,1), \ldots, f(A,m_b-1)). \tag{2.15}$$

The *k-times maximum* of a function $f(A,B)$ over a set of variables $B = \{b_1, b_2, \ldots, b_k\}$ is defined as the MVL function

$$h(A) = \max_B{}^{\mathrm{k}} f(A,B) = \max_{b_1}(\max_{b_2}(\ldots \max_{b_k} f(A,B)\ldots)). \tag{2.16}$$

Similar to the Boolean derivation operators, there are different notations for the derivation operators. The variables of the derivation operators are centered below the operator, such as in (2.16), if the formula is displayed on a separate line, and the variables are set as an index of the operator, such as $\max^{\mathrm{k}}{}_B f(A,B)$, if the operator is embedded into text. The meaning of both notations is the same.

Note that the minimum and maximum over a variable does not depend on this variable anymore, and the minimum and maximum over a set of variables does not depend on this set of variables. The function $\min^{\mathrm{k}}{}_B f(A,B)$ is the projection of the smallest value of the function $f(A,B)$ over the MVL space $\mathbb{M}_B$. Similarly, $\max^{\mathrm{k}}{}_B f(A,B)$ is the projection of the greatest value of function $f(A,B)$ over the MVL space $\mathbb{M}_B$.

Example 2.3. Figure 2.10 shows the function $f(a,b,c,d)$ and the functions

$$g(a,b,c) = \min_d f(a,b,c,d),$$
$$h(a,b) = \min_d g(a,b,c) = \min_{c,d}{}^{\mathrm{k}} f(a,b,c,d).$$

The values of the first column of the function $g(a,b,c)$ are the minimum values of the first three columns of the function $f(a,b,c,d)$. The second column of $g(a,b,c)$ consists of the minimum values of the last three columns of $f(a,b,c,d)$. The circles around the values of function $f(a,b,c,d)$ indicate the values of function $g(a,b,c)$. Because variable d assumes all values in the first and the last three columns in the chart of $f(a,b,c,d)$, these columns span the MVL space $\mathbb{M}_d$.

The values of function $h(a,b)$ are the minimum values over a row of function $g(a,b,c)$ (indicated by bold faced circles) which is the same value as the minimum value over a row of function $f(a,b,c,d)$ (also indicated by bold faced circles). Because the variables $\{c,d\}$ address the columns of the chart of $f(a,b,c,d)$, a row spans the MVL space $\mathbb{M}_{\{c,d\}}$.

The min-operator is characterized by two properties.

1. The minimum is smaller than or equal to its arguments,

$$\min(a_1, a_2, \ldots, a_n) \leq a_i, \quad i = 1, \ldots, n.$$

2. There exists at least one argument of the min-operator that equals the minimum,

$$\exists i : a_i = \min(a_1, a_2, \ldots, a_n), \quad 1 \leq i \leq n.$$

These properties can be extended to the $\min^k$ and $\max^k$ operators. See Figure 2.10 for instance. Property 1 translates into the observation that the values of function $h(a,b)$ are smaller than or equal to the values of function $f(a,b,c,d)$ in the same row. According to property 2, a value of function $h(a,b)$ appears at least once in the same row in the chart of the function $f(a,b,c,d)$.

Below, Lemma 2.2 states property 2 for the minimum over a single variable. In Theorem 6 the property is generalized for the $\min^k$ operator over arbitrary sets of variables. Theorem 8 shows that the $\min^k$ operator satisfies property 1.

In the following, the notation $\max^k{}_B f(A,B)|_{A_i}$ denotes k-times maximum of $f(A,B)$ over B evaluated for the minterm A_i, i.e. for $g(A) = \max^k{}_B f(A,B)$ there is $g(A_i) = \max^k{}_B f(A,B)|_{A_i}$.

Lemma 2.2. *Let $f(A,b)$ be an MVL function, where A is a set of variables, and $b \notin A$ a variable not in A. For every minterm A_i of the variables A there is a value $b = m$ with $\min_b f(A,b)|_{A_i} = f(A_i,m)$.*

Proof. Let $b = m$ be the value of b that minimizes the term $f(A_i,b)$. Then it follows from (2.13)

$$\min_b f(A,b)|_{A_i} = \min(f(A_i,0), f(A_i,1), \ldots, f(A_i,m_b-1)) = f(A_i,m),$$

where m_b is the cardinality of the variable b. □

Theorem 6. *Let $f(A,B)$ be an MVL function, and A and B be disjoint sets of variables. For every minterm A_i there is a minterm B_j with*

$$\min_B{}^k f(A,B)|_{A_i} = f(A_i,B_j). \tag{2.17}$$

Proof. The theorem is proven is by induction on the number of variables in B. If B consists of a single variable, then the theorem follows directly from Lemma 2.2.

Assume the theorem has been proven for every set of variables with $n-1$ variables. It will be shown that the theorem holds for every set of variables with n variables. Let B be a set of variables with $|B| = n$ variables, and $B = \{b\} \cup B'$ be a partition of B into a set of variables B' and variable $b \notin B'$. Then by (2.14)

$$\min_B{}^k f(A,B) = \min_b \min_{B'}{}^k f(A,B),$$

and by Lemma 2.2, there is a value $b = m$ so that

$$\min_B{}^k f(A,B)|_{A_i} = \min_{B'}{}^k f(A,B)|_{A_i,b=m}, \tag{2.18}$$

Furthermore, $|B'| = n-1$, and it follows from the induction hypothesis that there is a set of variables B'_j with

$$\min_{B'}{}^k f(A,B)|_{A_i,b=m} = f(A_i,B'_j,b=m)$$

Let $B_j = B'_j \cup \{b=m\}$, then

$$\min_{B'}{}^k f(A,B)|_{A_i,b=m} = f(A_i,B_j), \tag{2.19}$$

and substitution of (2.19) into (2.18) gives (2.17). □

Theorem 7. *Let $f(A,B)$ be an MVL function, and A and B be disjoint sets of variables. For every minterm A_i there is a minterm B_j with*

$$\max_B{}^k f(A,B)|_{A_i} = f(A_i,B_j). \tag{2.20}$$

Proof. The proof is dual to the proof of Theorem 6. □

Theorem 8. *For every MVL function $f(A,B)$ with disjoint sets of variables A and B there is $\min^k{}_B f(A,B) \leq f(A,B)$.*

Proof. If B consists of a single variable b with cardinality m_b, there is

$$\min(f(A,b=0),f(A,b=1),\ldots,f(A,b=m_b-1)) \leq f(A,b=i) \quad \text{for } i=1,\ldots,m_b, \qquad (2.21)$$

and the result follows by substitution of (2.13) into (2.21). If B consists of more than a single variable, the result can be obtained by induction over the number of variables in B. □

Theorem 9. *For every MVL function $f(A,B)$ with disjoint sets of variables A and B there is* $\max^k{}_B f(A,B) \geq f(A,B)$.

Proof. The proof is dual to the proof of Theorem 8. □

Theorems 10 and 11 provide test criteria for a partial order between functions, when these functions depend on different sets of variables. Such comparisons are frequently required for bi-decomposition tests.

Theorem 10. *Let $f(A,B)$ be an MVL function where A and B are disjoint sets of variables. A function $g(A)$, that does not depend on the set of variables B, satisfies $g(A) \leq f(A,B)$ if and only if $g(A) \leq \min^k{}_B f(A,B)$.*

Proof. It follows directly from Theorem 8 that $g(A) \leq \min^k{}_B f(A,B)$ implies $g(A) \leq f(A,B)$.

Assume there exists a function $h(A) \leq f(A,B)$ with $h(A) \not\leq \min^k{}_B f(A,B)$. Then, there is a minterm A_i with $h(A_i) > \min^k{}_B f(A,B)|_{A_i}$. Because of Theorem 6 there is a minterm B_j with $\min^k{}_B f(A,B)|_{A_i} = f(A_i,B_j)$. Therefore, there is $h(A_i) > f(A_i,B_j)$. This contradicts the assumption $h(A) \leq f(A,B)$ and $g(A) \leq f(A,B)$ only if $g(A) \leq \min^k{}_B f(A,B)$. □

Theorem 11. *Let $f(A,B)$ be an MVL function where A and B are disjoint sets of variables. A function $g(A)$, that does not depend on the set of variables B, satisfies $f(A,B) \leq g(A)$ if and only if* $\max^k{}_B f(A,B) \leq g(A)$.

Proof. The proof is dual to the proof of Theorem 10. □

2.3.4 Incompletely Specified MVL Functions

Incompletely specified MVL functions (MVL ISFs) are heavily used in machine learning. For a given set of examples there is a mapping from attributes to a goal. Examples are given only for a small subset of all possible combinations of attributes. The aim of machine learning is to determine values of the goal for attribute combinations that are not given. The set of examples can be modeled by MVL ISFs which are a direct extension of Boolean ISFs.

Logic synthesis for MVL circuits also applies MVL ISFs. If a certain combination of inputs cannot occur in an application, the value of the synthesized function for this input combination can be any value which is modeled by a don't care.

Definition 14. An *MVL ISF* F(A) is a mapping $C \rightarrow \{0,\ldots,m_o-1\}$ from a subset $C \subseteq \mathbb{M}_A$ of all minterms into the set of integers $\{0,\ldots,m_o-1\}$. The set C is called *care set*, the complement set $D = \mathbb{M}_A \setminus C$ is the *don't care set (DC-set)*.

MVL ISFs can be visualized by maps and tables similar to MVL functions (see Section 2.3.1). The elements of the care set are denoted by their respective values, the elements of the DC-set by Φ.

2.3.5 Multi-Valued Relations

MVL ISFs either specify the value of a function for a particular minterm completely, or the value is undefined. Sometimes, values of a function are partially specified so that there is a choice between several, but not all values. In machine learning application, for instance, the color classification of an example may be known to be 'yellow' or 'green', but not 'black'. These cases cannot be modeled by MVL ISFs where the color must be specified to be one color (a care), or can be any color (a don't care). Therefore, the notion of ISFs has been extended to *MVL relations*.

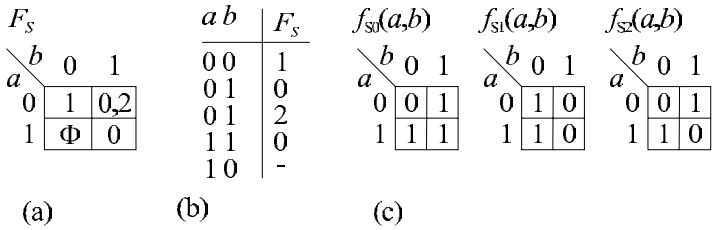

Figure 2.11: MVL relation $F_S\langle(a,b),v\rangle$ with output cardinality $m_v = 3$. **(a)** Display as a map. **(b)** Display as a Table. **(c)** Specification by two-valued functions.

Definition 15. An *MVL relation* $R\langle A,v\rangle$ is a many-to-many mapping, or relation, between the set of minterms $\mathbb{M}_A$ of the variables A and the set of output values $\mathbb{M}_v = \{0,\ldots,m_o-1\}$

$$R\langle A,v\rangle \subseteq \mathbb{M}_A \times \mathbb{M}_v, \tag{2.22}$$

where at least one value is specified for each minterm A_i,

$$\forall A_i : \exists v_j : \langle A_i, v_j\rangle \in R\langle A,v\rangle . \tag{2.23}$$

If the relation $R\langle A,v\rangle$ specifies all values from $\mathbb{M}_v$ for a minterm A_i, the relation has a *don't care* at A_i. If the relation $R\langle A,v\rangle$ specifies more than one, but not all, values from $\mathbb{M}_v$ for a minterm A_j, the relation has a *special don't care* at A_j. If only one value is specified for a minterm A_k, the relation has a care at A_k.

An MVL relation $R\langle A,v\rangle$ can be visualized as a map or table similar to ISFs. For a minterm A_i, all values v_j with $\langle A_i,v_j\rangle \in R\langle A,v\rangle$ are written into the field of the map for the minterm A_i. Don't cares can be abbreviated in the map by the don't care symbol Φ. Figure 2.11(a) shows the map of an MVL relation with a special don't care at $ab = 01$ and a don't care at $ab = 10$.

A table of an relation $R\langle A,v\rangle$ contains one row for each pair $\langle A_i,v_j\rangle$ that is element of the relation $R\langle A,v\rangle$ (see Figure 2.11(b)).

Sometimes it is easier to define MVL relations by a characteristic function.

Definition 16. The *characteristic function* $f(A,v)$ of the relation $R\langle A,v\rangle$ is a two-valued function with

$$f(A_i,v_j) = \begin{cases} 1 & \text{for } \langle A_i,v_j\rangle \in R\langle A,v\rangle \\ 0 & \text{otherwise.} \end{cases} \tag{2.24}$$

MVL relations are a generalization of ISFs. For an ISF $F(A)$ with the output cardinality m_o, the relation $R\langle A,v\rangle$ of the corresponding MVL relation can be constructed by the two rules:

1. If $F(A_i) = v_j$ is a care of ISF $F(A)$, the pair $\langle A_i,v_j\rangle$ is element of $R\langle A,v\rangle$.
2. If $F(A_i)$ is a don't care of ISF $F(A)$, the pairs $\langle A_i,0\rangle, \langle A_i,1\rangle, \ldots \langle A_i,m_o-1\rangle$ are elements of the relation $R\langle A,v\rangle$.

For every minterm, MVL relations specify a set of values. In the Boolean case there are only three possibilities, 0, 1 and $\Phi \rightarrow \{0,1\}$. For MVL functions with an output cardinality m_o, an MVL relation can specify any of the $2^{m_o}-1$ nonempty subsets of the set $\{0,1,\ldots,m_o-1\}$ as the set of possible function values for a minterm. An MVL relation $R\langle A,v\rangle$ can thus be specified by m_o two-valued MVL functions $f_{S0}(A),\ldots,f_{Sm_o-1}$ defined by

$$f_{Si}(A) = \begin{cases} 1 & \text{if } \langle A,i\rangle \in R\langle A,v\rangle, \\ 0 & \text{otherwise.} \end{cases}$$

See Figure 2.11(c) for example. The relation $R\langle(a,b),v\rangle$ from Figure 2.11(a) is specified by the functions $f_{S0}(a,b)$, $f_{S1}(a,b)$ and $f_{S2}(a,b)$. The function $f_{S2}(a,b)$ for instance has value 1 for all minterms where the relation allows the value 2, i.e. the minterms $ab = 01$ and $ab = 10$.

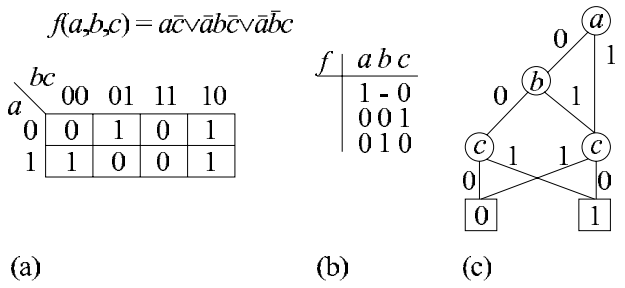

Figure 2.12: Data structures for Boolean functions. **(a)** Karnaugh map, **(b)** Table, **(c)** BDD.

2.4 Data Structures

2.4.1 Tables for Boolean Functions

Many data structures have been developed to store and manipulate Boolean functions. All of these data structures offer specific benefits in terms of memory size for certain classes of functions, or in terms of computational complexity for particular operations. Therefore, the optimal data structure depends on the structure and algorithmic requirements of the task that must be solved.

Most data structures are based on either tables or decision diagrams. There has also been some work on a combination of these two ideas [27, 14]. This section gives a short introduction into orthogonal tables [6], while binary decision diagrams [8] are discussed in Section 2.4.2.

A Boolean function can be stored in a table by enumerating all minterms of the ON-set. Each minterm is stored as a binary vector containing a '1' for each non-negated variable and a '0' for each negated variable of the minterm. In general, the number of binary vectors is too large to be stored directly. Therefore, the table is compressed by merging vectors whose values differ in only one place. The two vectors are replaced by a single ternary vector containing a dash '-' for the place with different values. The vectors '110' and '100' for instance can be merged to '1-0'. The three symbols '1', '0' and '-' are encoded by two bits. All vectors are arranged in a table. It has been shown that algorithms are much more efficient if the table is orthogonal, i.e. each minterm is contained in exactly one vector of the table [6]. Figure 2.12(a) displays a Boolean function in a Karnaugh map, and Figure 2.12(b) shows an orthogonal table of the function. Note that tables are not canonical, there can be more than one table for a Boolean function.

Most algorithms for tables iterate over the vectors of the table from the beginning to the end. Therefore, an efficient and flexible data structure is a list. *Ternary vector lists* (TVL) store tables of Boolean functions as a list of boxes, where each box contains an array of integers that encodes the values of the table [6]. This encoding benefits from the size of the computer registers. On a 32-bit processor for instance, 32 variables of the table can be processed in parallel.

2.4.2 Binary Decision Diagrams

Another representation of a Boolean function is a *decision tree*. The constant functions '0' and '1' are the leaves of the tree. The node representing $f(a, B)$ is labeled by a. The node has two child nodes representing the decision trees of $f(a = 0, B)$ and $f(a = 1, B)$ respectively. There are different decision trees for different orders of variables. A decision tree is *ordered* if the same order of variables is encountered on each path from the root to the leaves. To reduce the size of a decision tree, there are two *reduction rules* that transform the tree into an acyclic directed graph, or *decision diagram*.

1. If the two outgoing edges of a node P point to the same child node C, the node P is removed and the edges from the parents to the node P are redirected to the node C.

2. If two nodes P_1 and P_2 are labeled by the same variable and point to the same child nodes C_1, C_2, the node P_1 is removed and the edges from the parents to the node P_1 are redirected to the node P_2.

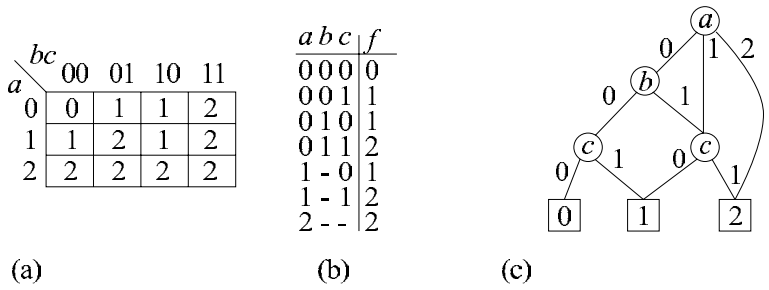

Figure 2.13: Data structures for MVL functions. **(a)** Display as a map. **(b)** Table. **(c)** MDD.

A decision diagram is called reduced if none of the two reduction rules can be applied to any node of the graph. Reduced ordered decision diagrams are called *binary decision diagrams (BDD)*. The BDD of the function $f(a, b, c)$ from Figure 2.12(a) is depicted in Figure 2.12(c).

2.4.3 Tables for MVL Functions

Similar to Boolean functions, there are several data structures to store MVL functions. Most of the data structures are based on tables or decision diagrams. Tables for MVL functions are introduced here, while decision diagrams for MVL functions are discussed in Sections 2.4.4 and 2.4.5.

A Boolean function has only two possible values, 0 and 1. It is possible to represent a Boolean function $f(A)$ by a table that includes all minterms A_i with $f(A_i) = 1$. If a minterm A_j is not element of the table, it can be concluded that $f(A_j) = 0$. A MVL function can have more than two values. Therefore, it is not possible to store an MVL function by a table representing a set of minterms. The table has to be extended by another column encoding the output value v of the function. A table of an MVL function has one column for each input variable in A and one column for the output value v of the function. A row $[A_i, v_j]$ with minterm A_i and output value v_j is element of the table if $f(A_i) = v_j$. This has the side effect that relations and ISFs can also be represented tables. Similar to Boolean functions, the table can be compressed by introduction of the dash symbol '-', meaning that the variable can assume any value. Formally, the row $[a =$ '-', $A_i \setminus a, v_j]$ can be replaced by the set of rows

$$\{[a = 0, A_i \setminus a, v_j], [a = 1, A_i \setminus a, v_j], \ldots, [a = m_a - 1, A_i \setminus a, v_j]\},$$

where m_a is the cardinality of the variable a. Figure 2.13(a) displays an MVL function in a chart. The table representation of the function is shown in Figure 2.13(b).

Not that dash elements cannot occur in the value column of function. However, in the table of MVL ISFs a dash in the value column indicates a don't care of the ISF.

2.4.4 Multi-Valued Decision Diagrams

MVL functions can be represented by *multi-valued decision trees*, which are an extension of binary decision trees. The decision tree of a constant function $f = c$ is a leaf node labeled with the value c. The decision tree of a function $f(a, B)$ is represented by a root node R labeled with the variable a and edges pointing from node R to the decision trees of the functions $f(a = i, B)$, $i = 0, \ldots, m_a - 1$, where m_a is the cardinality of the variable a. The edge from node R to the node of function $f(a = i, B)$ is labeled with i.

Similar to binary decision trees (see Section 2.4.2), multi-valued decision trees can be ordered and reduced. A multi-valued decision tree is ordered if there is the same order of variables on every path from the root node to any leaf node. Two reduction rules reduce an ordered multi-valued decision tree to a *Multi-valued Decision Diagram (MDD)*.

1. If all outgoing edges of a parent node P point to the same child node C, the incoming edges of P are redirected to C and the node P is deleted.

2. If two parent nodes P_1 and P_2 point to the same child nodes $C_1, \ldots, C_m$, the incoming edges of the node P_1 are redirected to the node P_2 and the node P_1 is deleted.

An MDD is called *reduced* if none of the two rules can be applied. In the remaining text only reduced and ordered MDDs will be discussed and the term MDD is used to denote reduced, ordered MDDs. Figure 2.13(c) displays the function from Figure 2.13(a) as an MDD. Similar to BDDs, MDDs are a canonical representation of MVL functions for a given order of variables.

Composition operators in MDDs can be computed recursively. The algorithm is very similar to the computation of the AND- and OR-operators in BDDs. For an operator $\pi(x, y)$ and two functions $g(a, B)$ and $h(a, B)$ assume that the variable a is located at the root of the MDD. The computation of $f(a, B) = \pi(g(a, B), h(a, B))$ applies the identity

$$f(a = i, B) = \pi(g(a = i, B), h(a = i, B)) \quad \text{for } a = 0, \ldots, m_a - 1.$$

Note, that $g(a = i, B)$, $h(a = i, B)$ and $f(a = i, B)$ are the child nodes of the MDD nodes representing $g(a, B)$, $h(a, B)$ and $f(a, B)$ respectively. The children of the node $f(a, B)$ are computed by recursive application of the operator $\pi(x, y)$ to the children of the nodes $g(a, B)$ and $h(a, B)$. The recursion terminates at the leaves where g and h are constant functions and $\pi(g, h)$ also is a constant, represented by a leaf node. The speed of the computation can be dramatically increased by storing intermediate results in a cache. If the same result is needed by another branch of the recursion, the result is taken from the cache, instead of computing it again.

The recursive algorithm must be modified to compute the derivation operators. Consider the minimum over a single variable a, $g(B) = \min_a f(a, B)$. If the variable a is located at the root of the MDD of $f(a, B)$, the minimum can be computed directly from the child nodes $f(a = i, B)$ using (2.13). The minimum operator in (2.13) is computed by the recursive algorithm, described in the previous paragraph. If a is not the root node of the MDD of $f(a, B)$, another variable, say $b \neq a$ is the root of the MDD. Then, there is the identity

$$g(b = i, B \setminus b) = \min_a f(a, b = i, B \setminus b) \quad \text{for } i = 0 \ldots m_b - 1$$

and the recursive algorithm can be applied. The algorithm is terminated, when a is the top-level node in $f(a, B)$. A similar algorithm computes the maximum over a variable.

The k-times minimum and maximum over a vector of variables can be computed by repeated computation of the differential operator over a single variable by (2.14) and (2.16).

ISFs can be represented by MDDs if an additional leaf node labeled with Φ is introduced. This node is used to terminate branches that lead to don't cares of the ISF. The same reduction rules as for MDDs of functions apply to MDDs of ISFs.

2.4.5 Binary Encoded Multi-Valued Decision Diagrams

MDDs are an efficient data structure to represent MVL functions and ISFs. However, there are some drawbacks:

- MDDs cannot naturally represent MVL relations. To represent a relation, more than one MDD is necessary.
- MDDs are a relatively new data structure, and there are not many efficient implementations of MDD packages on computers.

In contrast, there are many highly optimized BDD packages available. *Binary Encoded Multi-valued Decision Diagrams (BEMDD)* are a data structure that applies BDDs to represent MVL relations [23, 25]. BEMDDs store the characteristic function of an MVL relation (see Definition 16) in a BDD. The characteristic function is two-valued, but the input and output variables of the relation are multi-valued. Therefore, these variables must be encoded by Boolean variables to store an MVL relation as a Boolean function in a BDD. The most obvious way to encode an MVL variable by Boolean variables is a set of Boolean variables with a suitable code. The number of Boolean variables that are needed to encode an MVL variable a with cardinality m_a is $n_a = \lceil \log_2(m_a) \rceil$. The most natural encoding is a binary encoding. An MVL variable a is represented by a set $B = \{b_0, b_1, \ldots, b_{n_a-1}\}$ of Boolean variables by the equation

$$a = \sum_{i=0}^{n_a-1} 2^i * b_i, \tag{2.25}$$

Table 2.1: Encoding of a 5-valued MVL variable a by the set of Boolean variables $\{b_0, b_1, b_2\}$.

a	b_0	b_1	b_2
0	0	0	-
1	0	1	-
2	1	0	-
3	1	1	0
4	1	1	1

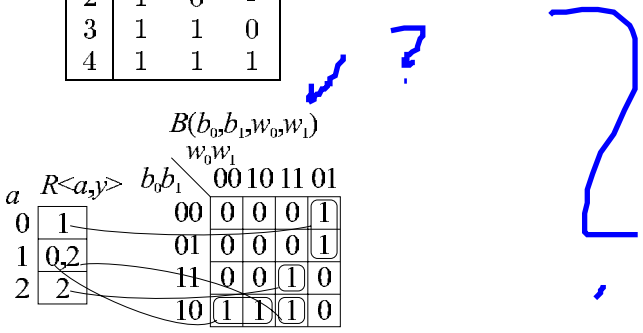

Figure 2.14: Representation of an MVL relation by a Boolean function. **(a)** Maps of the MVL relation $R\langle a, v\rangle$ and the corresponding Boolean function $B(b_0, b_1, w_0, w_1)$. **(b)** Mapping between the MVL and Boolean minterms.

where the summation and product have their usual meaning for integers. In general, the cardinality of MVL variables is not a power of two. In this case, some minterms of the Boolean variables do not correspond with MVL minterms. These unused Boolean minterms are called *spare values*. Research by Mishchenko and Perkowski indicates that more efficient BEMDDs can be found if the spare values are also assigned to MVL minterms. A particular efficient assignment of these spare values is to encode the first s MVL minterms by pairs of adjacent Boolean minterms [23].

Example 2.4. The encoding of a five-valued MVL variable a by three Boolean variables $B = \{b_0, b_1, b_2\}$ for example, is shown in Table 2.1. There are eight Boolean minterms for five MVL values. Therefore, there are three spare values. The table indicates that the first three MVL values (0, 1 and 2) are each represented by a disjunction of two Boolean minterms that differ only in the variable b_2.

In general, there are $s = 2^{n_a} - m_a$ spare values. The value of the MVL variable a is computed by

$$a = \begin{cases} \sum_{i=0}^{n_a-2} 2^i * b_i & \text{for } a < s, \\ \left(\sum_{i=0}^{n_a-1} 2^i * b_i\right) - s & \text{otherwise.} \end{cases} \tag{2.26}$$

The number of BEMDD nodes is reduced by this encoding because the cubes representing the first s values of the MVL variable a do not depend on the variable b_{n_a-1}. This reduces the number of levels for these branches in the BEMDD.

Example 2.5. The relation $R\langle a, v\rangle$, shown in Figure 2.14(a), is represented by the Boolean function $B(b_0, b_1, w_0, w_1)$. The three valued variables a and v are represented by the Boolean sets of variables $\{b_0, b_1\}$ and $\{w_0, w_1\}$ respectively. The mapping between the MVL and the Boolean variables is shown in Figure 2.14(b).The mapping between the values of the MVL relation and Boolean cubes is indicated by lines from the map of relation $R\langle a, v\rangle$ to the map of the Boolean function $B(b_0, b_1, w_0, w_1)$.

BEMDD operations can be performed by BDD operations. In this work, the usual mathematical notation will be used for the BDD operations, see Table 2.2 for a summary of the standard BDD operations.

Functions and ISFs can be stored in BEMDDs as special cases of relations. Composition operators over functions can be computed by composing the BEMDDs representing the MVL functions with the corresponding operator function. Nontrivial is the computation of the derivation operators $\min^k{}_B f(A, B)$ and $\max^k{}_B f(A, B)$. These derivation operators can be computed very efficiently by a slight modification of BEMDDs [25]. A function $f(A, B)$ is stored as a

Table 2.2: Notation for the standard BDD operations.

Notation	Operation
0	constant BDD 0
1	constant BDD 1
$g(A)$	BDD with support A
$\overline{g(A)}$	negation
$g(A) \wedge h(A)$	AND-operation
$g(A) \vee h(A)$	OR-operation
$g(A) \oplus h(A)$	EXOR-operation
$\min^{\mathrm{k}}{}_B f(A,B)$	universal quantification over B
$\max^{\mathrm{k}}{}_B f(A,B)$	existential quantification over B

lower bound interval, which is the relation that contains all values smaller than or equal to the function values $f(A,B)$

$$R\langle(A,B),y\rangle = \{\langle(A_i,B_j),y_k\rangle \mid y_k \leq f(A_i,B_k)\}. \tag{2.27}$$

The lower bound interval of the derivation $\max^{\mathrm{k}}{}_B f(A,B)$ can now be computed by existential quantification of the BEMDD corresponding to the lower bound interval of the function $f(A,B)$. Similarly, k-times minimum $\min^{\mathrm{k}}{}_B f(A,B)$ can be computed by existential quantification of the *upper bound interval* of the function $f(A,B)$. MVL functions can be converted to their lower bound interval by composition with the relation

$$R\langle x,y\rangle = \{\langle x_i,y_j\rangle \mid x \leq y\}. \tag{2.28}$$

In many applications, such as decomposition, functions are stored directly as their lower or upper bound interval.

Other operations on relations, such as intersection and union for example, can easily be computed by Boolean operations on the BEMDDs of the relations.

The major benefits of BEMDDs are the efficient representation of relations and the application of BDD as underlying data structure. BDD packages are not only highly optimized, but the constant size of BDD nodes enables very fast calculations on BEMDDs. MDDs in contrast have nodes of varying sizes for different cardinalities. Therefore, an efficient storage of MDD nodes in an array is impossible. Also, traversal routines (that visit every node of the decision diagram) on BDDs have a simpler structure than MDD traversal routines.

Drawback of BEMDDs are

- the necessary overhead to store functions and ISFs as relations and
- complicated traversal algorithms (that visit each node of the BEMDD) because of the encoding of spare values.

In Chapter 8 there is a comparison between BEMDDs and MDDs applied to bi-decomposition of MVL relations.

Chapter 3

Overview of Bi-Decomposition Techniques

3.1 The Boolean Bi-Decomposition Problem

This chapter gives an overview of theory and algorithms that have been used to bi-decompose functions. Traditionally, most of the work has been done for Boolean functions, but recently some extensions to MVL functions have been published.

This section starts with the definition of the bi-decomposition problem. Section 3.2 describes the bi-decomposition theory for OR, AND, EXOR and weak bi-decomposition of Boolean functions. The results of the theory are applied to design decomposition algorithms for Boolean ISFs in Section 3.3.

The theory shown in Section 3.2 is generalized to MVL bi-decomposition in the Sections 5.5 and 5.7. The algorithms of Section 3.3 provide are extended to decomposition of MVL functions in Chapter 6. The algorithms are applied in an extensible and object-oriented bi-decomposer presented in Chapter 7.

Given a function $f(X)$, the *bi-decomposition problem* is to find an operator function $\pi(x,y)$ and a partition of the set of variables $X = A \cup B \cup C$ into disjoint sets A, B and C so that there are functions $g(A,C)$ and $h(B,C)$ with

$$f(A,B,C) = \pi(g(A,C),h(B,C)). \tag{3.1}$$

The sets A and B should not be empty. The set A is called the *free set*, the set B is called the *bound set* and the set C is called the *shared set*. The functions $g(A,C)$ and $h(B,C)$ are *decomposition functions* of $f(X)$. The pair $\langle g(A,C),h(B,C)\rangle$ is a *bi-decomposition* of $f(X)$ with respect to the operator $\pi(x,y)$, or shorter, a *π-decomposition* of $f(X)$. If the shared set C is the empty set, the bi-decomposition is *disjoint*. If the bi-decomposition is non-disjoint, the shared set C can be computed by $C = X \setminus (A \cup B)$.

Example 3.1. Consider the function $f(a,b,c) = ac \vee bc$. An OR-decomposition of $f(a,b,c)$ with respect to the free set $A = \{a\}$, the bound set $B = \{b\}$ and the shared set $C = \{c\}$ is the pair of decomposition functions $g(a,c) = ac$ and $h(b,c) = bc$.

There are functions that are not bi-decomposable, i.e. there is no operator $\pi(x,y)$ that satisfies (3.1) for any partition of X into a non-empty free set A and a non-empty bound set B. A function for which there exists a solution to the bi-decomposition problem is called *bi-decomposable* . If the function is bi-decomposable with respect to the operator function $\pi(x,y)$, the function is said to be *π-decomposable* . The *bi-decomposition test problem* is to decide if a function is π-decomposable with respect to given free and bound sets.

For Boolean functions there are ten functions that truly depend on two variables (see bold faced functions in Table 3.1), and hence, are candidates for the operator function $\pi(x,y)$.However, it has been shown, that it is sufficient to consider only the OR-decomposition and the EXOR-decomposition [32]. All other decomposition types can be derived from these two types by either decomposing the complemented function, or by decomposing into complemented decomposition functions. The AND-decomposition for example, can be computed from the OR-decomposition

Table 3.1: Operator functions for bi-decomposition.

x	y	0	AND		x		y	EXOR	OR			$\overline{y}$		$\overline{x}$			1
0	0	0	**0**	**0**	0	**0**	0	**0**	**0**	**1**	**1**	1	**1**	1	**1**	**1**	1
0	1	0	**0**	**0**	0	**1**	1	**1**	**1**	**0**	**0**	0	**0**	1	**1**	**1**	1
1	0	0	**0**	**1**	1	**0**	0	**1**	**1**	**0**	**0**	1	**1**	0	**0**	**1**	1
1	1	0	**1**	**0**	1	**0**	1	**0**	**1**	**0**	**1**	0	**1**	0	**1**	**0**	1

by the identity $x \wedge y = \overline{\overline{x} \vee \overline{y}}$. Most decomposition systems additionally consider the AND-decomposition by algorithms dual to the OR-decomposition algorithm because these algorithms are faster.

Above terms can be generalized to bi-decomposition of ISFs. For an ISF $F(X)$ with the characteristic function set $\mathbf{F}(X)$ the bi-decomposition problem is to find an operator function $\pi(x,y)$ and a partition of the set of variables X into nonempty sets A, B and a set C so that there are functions $g(A,C)$ and $h(B,C)$ with

$$\pi(g(A,C),h(B,C)) \in \mathbf{F}(X). \tag{3.2}$$

All other terms introduced in this section, including the weak bi-decomposition can also be extended to ISFs.

A solution to the bi-decomposition problem for an ISF $F(X)$ is a solution to the bi-decomposition problem for any member function of $\mathbf{F}(X)$.

It has been shown that there are ISFs that are not bi-decomposable [38]. If the bound set B can be empty, every ISF is decomposable. For an ISF $F(X)$, the *weak bi-decomposition problem* is to find a member function $f(X) \in \mathbf{F}(X)$, an operator function $\pi(x,y)$ and a partition of the set of variables X into nonempty sets A and C so that there are functions $g(A,C)$ and $h(C)$ with

$$f(A,C) = \pi(g(A,C),h(C)). \tag{3.3}$$

An ISF with a solution to the weak bi-decomposition problem is said to be *weakly bi-decomposable* . Every ISF is weakly EXOR-decomposable because $f(A,C) = g(A,C) \oplus h(C)$ is satisfied for all function $h(C)$ if $g(A,C) = f(A,C) \oplus h(C)$. However, this fact does not lead to useful decompositions because the decomposition function $g(A,C)$ depends on the same variables as the original ISF. It has been proven that all Boolean ISFs are either bi-decomposable or weakly OR-decomposable, or weakly AND-decomposable [38]. Furthermore, it has been shown that weak OR- and AND-decomposition simplifies the ISFs by introducing additional don't cares. Therefore, recursive bi-decomposition and weak bi-decomposition of ISFs terminates.

The bi-decomposition problem can be solved by developing a decomposition test that determines the π-decomposability of an ISF $F(X)$ for all combinations of an operator $\pi(x,y)$, a free set $A \subset X$ and a bound set $B \subset X$. For Boolean functions there is a constant number of operators $\pi(x,y)$, therefore, making decomposition tests for each operator is easy. However, the number of different free and bound sets, grows exponentially with the number of variables in X, and testing all combinations would be computationally expensive or impossible. However, it is possible to start with small free and bound sets, and to move variables successively from the shared set to either the free or the bound set.

Theorem 12. *If an ISF $F(X)$ is π-decomposable with respect to the free set A and the bound set B, it is also π-decomposable with respect to any free set $A' \subseteq A$ and any bound set $B' \subseteq B$.*

Proof. For a diagram of the sets of variables see Figure 3.1. ISF $F(X)$ is π-decomposable with respect to the sets of variables A and B. Therefore, there is a π-decomposition $\langle g(A,C),h(B,C)\rangle$ with $f(X) = \pi(g(A,C),h(B,C))$ and $f(X) \in \mathbf{F}(X)$. Two new functions are defined by

$$g'(A',C') = g'(A,C,C_A) = g(A,C)$$
$$h'(B',C') = h'(B,C,C_B) = h(B,C).$$

Then, there is $f(X) = \pi(g'(A',C'),h'(B',C'))$ and $g'(A',C')$ and $h'(B',C')$ are decomposition functions of $F(X)$. Therefore, $F(X)$ is π-decomposable with respect to the free set A' and the bound set B'. □

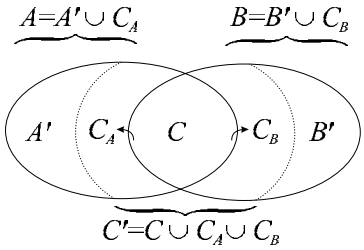

Figure 3.1: Sets of variables when moving variables from the shared set to free and bound set.

Because of Theorem 12, it is possible to check the π-decomposability of an ISF $F(X)$ by checking its π-decomposability for each pair of single variables. There are only $n(n-1)/2$ such pairs for n variables in X. The bi-decomposition can then be improved by shifting more variables from the shared set to either the free or the bound set.

3.2 Boolean Bi-Decomposition Strategy

3.2.1 OR-Decomposition

OR-Decomposition Test

This section gives an introduction into the theory that solves the OR-decomposition problem for ISFs. In this section a test criterion will be developed that can decide whether an ISF contains OR-decomposable functions. Then, it is shown, how to compute the decomposition functions of an OR-decomposition.

The solution of the bi-decomposition problem can be reduced to the test for decomposability for a given free and bound set (see Section 3.1). The development of a test for OR-decomposability is illustrated by *decomposition charts* which are Karnaugh maps with the variables of the free set at the rows and the variables of the bound set at the columns. In the case of non-disjoint decomposition, a decomposition chart consists of one Karnaugh map for each minterm of the shared variables.

There are various methods of OR-decomposition of ISFs, such as the *differential method* by Steinbach et.al. [3], the *pattern method* by Perkowski [27] and the *value removal method* by Sasaso [32]. All these methods are based on the same idea of identifying rows and columns in the decomposition chart that do not contain the value '0', i.e. that consist of only values '1' and 'Φ'.

Example 3.2. Figure 3.2(a) shows the decomposition chart for the disjoint bi-decomposition of an ISF $F(a,b,c,d)$ with respect to the free set $A=\{a,b\}$ and the bound set $B=\{c,d\}$. To keep the example simple, the shared set C is empty.

There is $F(abcd=1100)=0$, and hence, $g(ab=11)=h(cd=00)=0$ for all OR-decomposition functions $g(a,b)$ and $h(c,d)$. Upper bounds on the decomposition functions are the functions $g_u(a,b)$ and $h_u(c,d)$ that have the value '0' only if F(a,b,c,d) has the value '0' in the same row or column of the decomposition chart respectively, see the maps of $g_u(a,b)$ and $h_u(c,d)$ in Figure 3.2(a).

A possible OR-decomposition can be computed by the disjunction of the functions $g_u(a,b)$ and $h_u(c,d)$. The function $f_p(a,b,c,d)=g_u(a,b)\vee h_u(c,d)$ is shown in Figure 3.2(d). Comparison with the function $f_t(a,b,c,d)$ in Figure 3.2(b) verifies that $f_t(a,b,c,d)\leq f_p(a,b,c,d)$ and the pair $\langle g_u(a,b),h_u(c,d)\rangle$ is an OR-decomposition of the ISF $F(a,b,c,d)$.

Because the differential method gives an elegant formal description using the BDC, this method will be explained here. In the following, a formula will be developed that decides if there is an OR-decomposition of the ISF $\mathbf{F}(A,B,C)=[f_t,f_u]$, with respect to the free set A and the bound set B.

The ISF $F(A,B,C)$ is OR-decomposable if there are functions $g(A,C)$ and $h(B,C)$ so that

$$f(A,B,C)=g(A,C)\vee h(B,C), \tag{3.4}$$

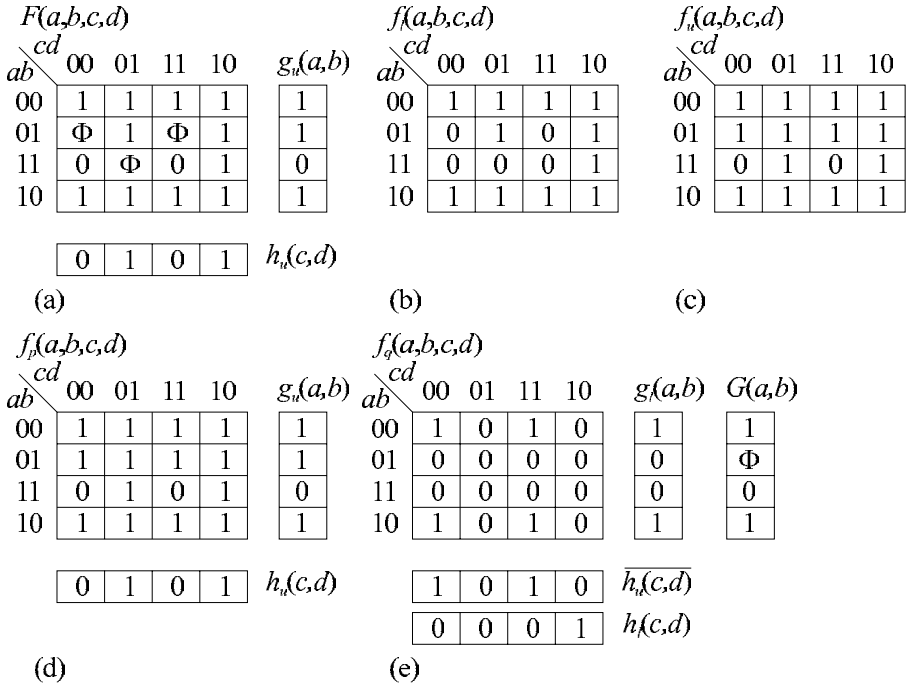

Figure 3.2: OR-Decomposition test of ISF. **(a)** Decomposition chart of an ISF $F(a,b,c,d)$ with upper bounds $g_u(a,b)$ and $h_u(c,d)$ on the decomposition functions. **(b)** Lower bound $f_l(a,b,c,d)$ of the ISF $F(a,b,c,d)$. **(c)** Upper bound $f_u(a,b,c,d)$ of the ISF $F(a,b,c,d)$. **(d)** Function $f_p(a,b,c,d) = g_u(a,b) \vee h_u(c,d)$ **(e)** Lower bound $g_l(a,b) = \max^k_{\{a,b\}} f_q(a,b,c,d)$ with $f_q(a,b,c,d) = f_l(a,b,c,d) \wedge \overline{h_u(c,d)}$, and lower bound $h_l(c,d)$.

with $f(A,B,C) \in \mathbf{F}(A,B,C)$, which is equivalent to

$$f_l(A,B,C) \leq f(A,B,C) \quad \text{and} \tag{3.5}$$

$$f(A,B,C) \leq f_u(A,B,C). \tag{3.6}$$

Consider the minterms (A_i,B_j,C_k) with $F(A_i,B_j,C_k) = 0$. Because $f(A,B,C) \in \mathbf{F}(A,B,C)$, there is $f(A_i,B_j,C_k) = 0$. From (3.4) follows that

$$g(A_i,C_k) = h(B_j,C_k) = 0. \tag{3.7}$$

Because $g(A,C)$ does not depend on the set of variables B, it can be concluded that $g(A_i,C_k) = 0$ for every row (A_i,C_k) of the decomposition chart of $F(A,B,C)$ that contains at least one function value $F(A_i,B_j,C_k) = 0$. Let $g_u(A,C)$ be the function with

$$g_u(A_i,C_k) = \begin{cases} 0 & \text{if } \exists B_j : F(A_i,B_j,C_k) = 0 \\ 1 & \text{otherwise.} \end{cases} \tag{3.8}$$

It follows from the previous discussion that

$$g(A,C) \leq g_u(A,C) \tag{3.9}$$

is a necessary condition for a decomposition function $g(A,C)$. The function $g_u(A,C)$ can be computed with the help of the derivation operator $\min^k_B$. Observe that there is $f_u(A_i,B_j,C_k) = 0$ if and only if $F(A_i,B_j,C_k) = 0$. Therefore, $g_u(A,C)$ can be computed by the projection

$$g_u(A,C) = \min_B{}^k f_u(A,B,C). \tag{3.10}$$

A similar derivation for the columns of the decomposition chart gives

$$h(B,C) \leq h_u(B,C) \quad \text{with} \tag{3.11}$$

$$h_u(B,C) = \min_A{}^k f_u(A,B,C) \tag{3.12}$$

as necessary condition for the decomposition functions $h(B,C)$ of the ISF $F(A,B,C)$.

Now let

$$f_p(A,B,C) = g_u(A,C) \vee h_u(B,C). \tag{3.13}$$

Because of the construction of $g_u(A,C)$ and $h_u(B,C)$ there is

$$f_p(A,B,C) \leq f_u(A,B,C). \tag{3.14}$$

It can easily be verified that the OR-operation is monotone, i.e. $g_1(X) \leq g_2(X)$ and $h_1(X) \leq h_2(X)$ implies

$$g_1(X) \vee h_1(X) \leq g_2(X) \vee h_2(X).$$

Now, $g_u(A,C)$ and $h_u(B,C)$ are upper bounds of the OR-decomposition functions of $F(A,B,C)$. Therefore, the function $f_p(A,B,C)$ is an upper bound on the composition of any two OR-decomposition functions, i.e. if $g_i(A,C)$ and $h_j(B,C)$ are decomposition functions, there is

$$g_i(A,C) \vee h_j(B,C) \leq f_p(A,B,C).$$

The upper bound $f_p(A,B,C)$ should satisfy (3.5). It follows that $f_l(A,B,C) \leq f_p(A,B,C)$, and substitution of (3.13) results in

$$\boxed{f_l(A,B,C) \leq \min_B{}^{\mathrm{k}} f_u(A,B,C) \vee \min_A{}^{\mathrm{k}} f_u(A,B,C)} \tag{3.15}$$

is a necessary, and because of (3.14), sufficient condition for the OR-decomposability of $F(A,B,C)$.

Computation of OR-Decompositions

It can be concluded that the OR-decomposition test problem can be solved by checking inequality (3.15). If the given ISF is OR-decomposable, then a bi-decomposition can be computed by (3.10) and (3.12).

To optimize further decomposition steps, it would be of advantage to compute not just one decomposition function $g(A,C)$, but an ISF $G(A,C)$ where each member function is a decomposition function. It has been shown that $g_u(A,C)$ from (3.10) is the upper bound on the set of decomposition functions. It is now shown how to compute the lower bound $g_l(A,C)$.

Let $h(B,C)$ be an OR-decomposition function of $F(A,B,C)$. It has been shown that $h(B_j,C_k) = 0$ for every column (B_j,C_k) of the decomposition chart that contains the value '0'. Therefore, if such a column contains the value '1', in say row (A_i,C_k), there must be $g(A_i,C_k) = 1$ for all decomposition functions $g(A,C)$.

The lower bound $g_l(A,C)$ of the set of decomposition functions $g(A,C)$ can be computed by setting $g_l(A_i,C_k) = 1$ for rows (A_i,C_k) of the decomposition chart that contain the value '1' in a column (B_j,C_k) with values $h_u(B,C) = 0$. There is $f_l(A,B,C) = 1$ iff ISF $F(A,B,C) = 1$. Therefore, the values $F(A_i,B_j,C_k) = 1$ that are in columns (B_j,C_k) with $h_u(B_j,C_k) = 0$ can be computed by

$$f_q(A,B,C) = f_l(A,B,C) \wedge \overline{h_u(B,C)} = f_l(A,B,C) \wedge \overline{\min_A{}^{\mathrm{k}} f_u(A,B,C)}\,, \tag{3.16}$$

and the lower bound of the set of decomposition functions $g(A,C)$ is

$$\boxed{g_l(A,C) = \max_B{}^{\mathrm{k}} f_q(A,B,C) = \max_B{}^{\mathrm{k}} (f_l(A,B,C) \wedge \overline{\min_A{}^{\mathrm{k}} f_u(A,B,C)}\,).} \tag{3.17}$$

The ISF $\mathbf{G}(A,C) = [g_l, g_u]$ contains exactly the OR-decomposition functions $g(A,C)$ of the ISF $F(A,B,C)$.

Example 3.3. See Figure 3.2(a) for example. Column $cd = 11$ contains a '0' at $ab = 11$ and values '1' in the same column at $ab = 10$ and $ab = 00$. Therefore, there is $h(cd = 11) = 0$, $g(ab = 10) = 1$ and $g(ab = 00) = 1$ for all decomposition functions $h(c,d)$ and $g(a,b)$. The function $f_q(a,b,c,d)$ that indicates all '1' in columns with $h(c,d) = 0$ is shown together with the lower bound $g_l(a,b)$ in Figure 3.2(e).

The ISF of all decomposition functions $h(B,C)$ can be computed by exchanging the variables A and B in (3.17). Note, although all functions in $G(A,C)$ and $H(B,C)$ are decomposition functions, not all pairs of functions $\langle g(A,C), h(B,C)\rangle$ with $g(A,C) \in \mathbf{G}(A,C)$ and $h(B,C) \in \mathbf{H}(B,C)$ are OR-decompositions of $F(A,B,C)$.

Example 3.4. See Figure 3.2(d,e). The functions $g_l(a,b)$, $g_u(a,b)$, $h_l(c,d)$ and $h_u(c,d)$ are decomposition functions because $g_u(a,b) \vee h_l(c,d) \in \mathbf{F}(a,b,c,d)$ and $g_l(a,b) \vee h_u(c,d) \in \mathbf{F}(a,b,c,d)$ as can easily be verified by computing the OR-composition. However, the pair $\langle g_l(a,b), h_l(c,d)\rangle$, is not an OR-decomposition because for $abcd = 0101$, there is $g_l(ab = 01) \vee h_l(cd = 01) = 0$ while $F(abcd = 0101) = 1$.

There is the problem of finding all OR-decompositions from the two sets $G(A,C)$ and $H(B,C)$. In Section 3.3 is shown how to select a proper function $g_0(A,C) \in \mathbf{G}(A,C)$ by recursive decomposition of the ISF $G(A,C)$. It remains the question, which ISF $H(B,C)$ contains decomposition functions with respect to the function $g_0(A,C)$, i.e. which functions $h(B,C) \in \mathbf{H}(B,C)$ satisfy

$$g_0(A,C) \vee h(B,C) \in \mathbf{F}(A,B,C)? \tag{3.18}$$

The ISF $H(B,C)$ can be found by ideas similar to the computation of the decomposition ISF $G(A,C)$ in (3.17). The upper bound still is the function $h_u(B,C)$ from (3.12) while the lower bound $h_l(B,C)$ can be computed by

$$\boxed{h_l(B,C) = \max_A{}^{\mathrm{k}}(f_l(A,B,C) \wedge \overline{g_0(A,C)}).} \tag{3.19}$$

The ISF $\mathbf{H}(B,C) = [h_l, h_u]$ contains exactly the functions $h(B,C)$ that satisfy (3.18).

3.2.2 AND-Decomposition

The AND-decomposition is dual to the OR-Decomposition. An ISF

$$\mathbf{F}(A,B,C) = [f_l, f_u] \tag{3.20}$$

is AND-decomposable iff

$$\boxed{f_u(A,B,C) \geq \max_B{}^{\mathrm{k}} f_l(A,B,C) \wedge \max_A{}^{\mathrm{k}} f_l(A,B,C).} \tag{3.21}$$

If the ISF $F(A,B,C)$ is AND-decomposable with respect to the free set A and the bound set B, the ISF $\mathbf{G}(A,C) = [g_l, g_u]$ of all decomposition functions $g(A,C)$ can be computed by

$$\boxed{\begin{aligned} g_l(A,C) &= \max_B{}^{\mathrm{k}} f_l(A,B,C) \\ g_u(A,C) &= \min_B{}^{\mathrm{k}}(f_u(A,B,C) \vee \overline{\max_A{}^{\mathrm{k}} f_l(A,B,C)}). \end{aligned}} \tag{3.22}$$

Given an AND-decomposable ISF $F(A,B,C)$ and a decomposition function $g_0(A,C)$, the ISF of all functions $h(B,C)$ that are AND-decompositions $\langle g_0(A,C), h(B,C)\rangle$ can be found by

$$\boxed{\begin{aligned} h_l(B,C) &= \max_A{}^{\mathrm{k}}(f_l(A,B,C)) \\ h_u(B,C) &= \max_A{}^{\mathrm{k}}(f_u(A,B,C) \vee \overline{g_0(A,C)}). \end{aligned}} \tag{3.23}$$

3.2.3 EXOR-Decomposition

Bi-decomposition with respect to the EXOR-operator is more complex than OR- and AND-decomposition. It was shown that the problem of EXOR-decomposition of Boolean ISFs is equivalent to the problem of finding Ashenhurst decompositions [18, 16]. A closed solution within the BDC is only known for the special case, where the bound set of the decomposition consists of a single variable [6]. The decomposition of the ISF $F(A,b,C)$ with respect to the free set A and the bound set $\{b\}$ has the form

$$f(A,b,C) = g(A,C) \oplus h(b,C) \tag{3.24}$$

with $f(A,b,C) \in \mathbf{F}(A,b,C)$, where A and C denote sets of variables, while b denotes only a single variable.

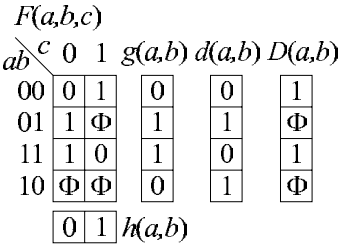

Figure 3.3: EXOR-decomposition of the ISF $F(a,b,c)$.

Example 3.5. The EXOR-decomposition algorithm is illustrated by the decomposition of an ISF $\mathbf{F}(a,b,c) = [f_l, f_u]$ with respect to the free set $A = \{a,b\}$ and the variable of the bound set c (see decomposition chart of Figure 3.3).For simplicity, the shared set C in this example is empty. Because there is only one variable in the bound set, there exist only two values for the decomposition function $h(c)$. Now, consider the function $f(a,b,c) = g(a,b) \oplus h(c)$. For $h(c=0) \neq h(c=1)$ there is $f(a,b,c=0) = \overline{f(a,b,c=1)}$, and for $h(c=0) = h(c=1)$ (not present in the figure) there is $f(a,b,c=0) = f(a,b,c=1)$. In the first case, the second column is the negation of the first column, while in the latter case the two columns of the decomposition chart are equal.

If a row of the decomposition chart contains only cares, the cares determine whether or not the two columns can be made equal or negated. If a row contains at least one don't care, both cases are possible depending on the value that is assigned to the don't care. Now, an auxiliary ISF $D(A,C)$ is computed. The ISF should be zero if the row (A,C) of the decomposition chart contains two cares of equal value, one if the row contains two cares of different values and don't care if the row contains at least one don't care. The ISF $\mathbf{D}(A,C) = [d_l, d_u]$ can be computed by the operations of the BDC. For the function

$$d(A,C) = \max_b{}^{\mathrm{k}}(\overline{f_l(A,b,C)} \vee f_u(A,b,C))$$

there is $d(A_i,C_j) = 1$ if and only if the row (A_i,C_j) of the decomposition chart of $F(A,b,C)$ contains at least one don't care, indicated by $f_l(A_i,C_j) = 0$ and $f_u(A_i,C_j) = 1$. The auxiliary ISF $D(A,C)$ is computed by the Boolean derivative:

$$d_l(A,C) = \frac{\partial f_u(A,b,C)}{\partial b} \vee \overline{d(A,C)} \tag{3.25}$$

$$d_u(A,C) = \frac{\partial f_u(A,b,C)}{\partial b} \wedge d(A,C). \tag{3.26}$$

The ISF $F(A,b,C)$ is decomposable if the auxiliary ISF $D(A,C)$ does not contain values '0' ($d_u(A_i,C_k) = 0$) and '1' ($d_l(A_j,C_k) = 1$) together for the same minterm C_k, i.e. if the columns of the decomposition chart can be made either equal or exact negations of each other. The resulting decomposition criteria is

$$\overline{\min_A{}^{\mathrm{k}} d_u(A,C)} \wedge \max_A{}^{\mathrm{k}} d_l(A,C) = 0. \tag{3.27}$$

Example 3.6. For the ISF $F(a,b,c)$ from Figure 3.3, the auxiliary function $d(a,b)$ and the auxiliary ISF $D(a,b)$ are also shown in the figure. The ISF $D(a,b)$ contains only the values '1' and 'Φ'. Therefore, the ISF $F(a,b,c)$ is EXOR-decomposable.

Unfortunately, this decomposition criterion cannot be generalized to the case where free and bound set contain more than one variable. Although it has not been proven, it seems unlikely that there is a closed solution within the BDC for the general case of vector-vector EXOR-decomposition because the decomposition has a rather complicated structure.

Decomposition algorithms for the general case of EXOR-decomposition can best be formulated by Zakrevskis *method of solving systems of equations* [42]. The EXOR-decomposition problem for ISFs is equivalent a system of linear equations. Consider the EXOR-decomposition of the ISF $F(A,B,C)$ with respect to the sets of variables A and B. For each element (A_i,B_j,C_k) of the care set of $F(A,B,C)$, the equation

$$g(A_i,C_k) \oplus h(B_j,C_k) = F(A_i,B_j,C_k) \tag{3.28}$$

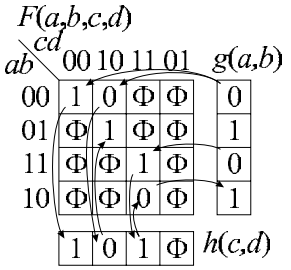

Figure 3.4: EXOR-decomposition of the ISF $F(a, b, c, d)$.

must hold for all EXOR-decompositions $\langle g(A, C), h(B, C)\rangle$ and each care (A_i, B_j, C_k) of the ISF $F(A, B, C)$.

Example 3.7. The EXOR-decomposition of ISF $F(a, b, c, d)$ (see Figure 3.4)with respect to the free set $A = \{a, b\}$ and the bound set $B = \{c, d\}$ defines the following system of equations:

$$g(ab = 00) \oplus h(cd = 00) = 1 \tag{3.29a}$$
$$g(ab = 00) \oplus h(cd = 10) = 0 \tag{3.29b}$$
$$g(ab = 01) \oplus h(cd = 10) = 1 \tag{3.29c}$$
$$g(ab = 11) \oplus h(cd = 11) = 1 \tag{3.29d}$$
$$g(ab = 10) \oplus h(cd = 11) = 0 \tag{3.29e}$$

The system of equations (3.28) is solved by arbitrarily choosing and initial solution for one value of either $g(A, C)$ or $h(B, C)$. Then, all values of the functions $g(A, C)$ and $h(B, C)$ that depend on this assignment are computed. This process is repeated until all equations of the system are satisfied, or a contradiction is found. In the first case an EXOR-decomposition has been computed, in the second case it was shown that the ISF is not EXOR-decomposable. Note, that the decomposition test is integrated into the computation of the EXOR-decomposition.

Example 3.8. The solution of the system (3.29) is indicated by the arrows in Figure 3.4. To solve (3.29a), the assignment $g(ab = 00) = 0$ is set. Then, the dependent value $h(cd = 00) = 1$ can be computed from (3.29a), and $h(cd = 10) = 0$ can be computed from (3.29b). Because $h(cd = 10)$ is now known, $g(ab = 01) = 1$ can be computed from (3.29c). No other value depends on these assignments. Therefore, a second assignment, for instance $g(ab = 11) = 0$ is made. From (3.29d) and (3.29e) follow the values $h(cd = 11) = 1$ and $g(ab = 10) = 1$. With these assignments, all equations (3.29a)–(3.29e) are satisfied. Therefore, the ISF $F(a, b, c, d)$ is EXOR-decomposable. Note that the value $h(cd = 01)$ is still undefined. This value is a don't care of the decomposition ISF.

The EXOR-decomposition algorithm TEST-EXOR-DEC(F, A, B) gets an ISF F, the free set A and the bound set B. The algorithm returns the decomposition ISF G if F is decomposable and the constant NOT-DECOMPOSABLE otherwise.

```
TEST-EXOR-DEC(F, A, B)
 1  S ← System of equations G(A_i, C_k) ⊕ H(B_j, C_k) = F(A_i, B_j, C_k)
 2  while S contains unknown values for G(A, C) or H(B, C)
 3        Find an unknown value G(A_m, C_n)
 4        G(A_m, C_n) ← 0
 5        Compute all dependent values of H(B, C) and G(A, C) in S
 6        if Contradiction found in S
 7           then return NOT-DECOMPOSABLE
 8  return G
```

When this algorithm is implemented on computers, the system of equations is not stored explicitly. Instead, the coefficients of the system are stored as sets of minterms in a suitable data structure, such as BDDs or TVLs. Then, the system is solved by manipulation of these data structures [41]. During the solution of the system of equations, some arbitrary choices for the solution must be made. These choices restrict the solution to a subset of all decomposition functions.

Table 3.2: Decomposition functions of EXORΦdecomposition.

$F(A,B,C)$	0	1	Φ	0	1	Φ
$g_0(A,C)$	0	0	0	1	1	1
$H(B,C)$	0	1	Φ	1	0	Φ

Example 3.9. In the decomposition of the ISF $F(a,b,c,d)$ from Figure 3.4, the choices of $g(ab=00)=0$ and $g(cd=11)=0$ were made arbitrarily. Other choices, such as $g(ab=11)=1$ for instance would result in other solutions.

If only one ISF is computed some space for optimization in further decompositions is lost. Therefore, all possible solutions should be computed and passed to the next stage of decomposition. The set of all solutions of EXOR-decomposition is not an ISF. Instead a new data structure, so called *combinated ISFs* (CISFs) must be applied to describe the set of decomposition functions of EXOR-decomposition [36, 37]. This data structure is discussed in detail in Section 4.3.5.

If one of the decomposition functions, say $g_0(A,C)$ is known, the ISF $\mathbf{H}(B,C)=[h_l,h_u]$ of all decomposition functions $h(B,C)$ can be computed from the values of the ISF $F(A,B,C)$ as shown in Table 3.2 (see also [6]).A care value of the ISF $F(A,B,C)$ causes a care value in decomposition ISF $H(B,C)$ while the don't cares of $F(A,B,C)$ can become don't cares of $H(B,C)$. The bounds of the ISF $H(B,C)$ can be computed by the elimination of the free set of variables A by the derivation operators

$$h_l(B,C)=\max_A{}^{\mathrm{k}}((f_l(A,B,C)\wedge\overline{g_0(A,C)})\vee(\overline{f_u(A,B,C)}\wedge g_0(A,C))) \tag{3.30}$$

$$h_u(B,C)=\min_A{}^{\mathrm{k}}((f_u(A,B,C)\wedge\overline{g_0(A,C)})\vee(\overline{f_l(A,B,C)}\wedge g_0(A,C))). \tag{3.31}$$

3.2.4 Weak Bi-Decomposition

Weak Bi-Decomposition Types

There are functions that are not bi-decomposable with respect to the OR-, AND- and EXOR-operators. To decompose these functions, weak bi-decomposition has been defined. Weak bi-decomposition is an extension of bi-decomposition, where the bound set B is empty. A weak bi-decomposition of the ISF $F(A,C)$ with respect to the operator $\pi(x,y)$ and the free set A is a pair of functions $\langle g(A,C),h(C)\rangle$ with

$$\pi(g(A,C),h(C))\in\mathbf{F}(A,C). \tag{3.32}$$

Usually, the operator $\pi(x,y)$ is either the OR- or the AND-operator. Weak bi-decomposition with respect to the EXOR-operator is trivial because any function $h(C)$ is a decomposition function. No application for weak EXOR-decomposition is known. Other operator functions $\pi(x,y)$ than OR, AND, or EXOR can be reduced to these three by negation the decomposition functions $g(A,C)$ and/or $h(C)$.

Weak OR- and AND-decomposition do not simplify the given ISF in terms of the number of variables. The function $g(A,C)$ depends on the same variables as the ISF $F(A,C)$. It can be shown however that the ISF $G(A,C)$ of all decomposition functions $g(A,C)$ contains more don't cares than ISF $F(A,C)$. Therefore, freedom is added to the decomposition ISF, which increases the chances of finding a bi-decomposition in the next step of decomposition. Functions which are neither weakly OR-decomposable nor weakly AND-decomposable are EXOR-decomposable [38]. Hence, after a weak AND-decomposition and a weak OR-decomposition, there always is an EXOR-decomposition possible.

Weak OR-Decomposition

Consider the case of weak OR-decomposition of an ISF $F(A,C)$,

$$g(A,C)\vee h(C)\in\mathbf{F}(A,C). \tag{3.33}$$

Additional don't cares are introduced into the ISF $G(A,C)$ of all decomposition functions $g(A,C)$ by values $h(C_k)=1$ because there is $g(A,C_k)\vee h(C_k)=1$ independent of the value of $g(A,C_k)$.

An upper bound $h_u(C)$ on the ISF $H(C)$ of all decomposition functions $h(C)$ can be computed by (3.12) with $B = \emptyset$. Additional don't cares are introduced for minterms (A_i, C_j) with $h_u(C_j) = 1$ and $F(A_i, C_j) = 1$, because there is $G(A_i, C_j) = \Phi$. Therefore, ISF $\mathbf{F}(A, C) = [f_l, f_u]$ is weakly OR-decomposable iff

$$\boxed{f_l(A, C) \wedge \min_A{}^{\mathrm{k}} f_u(A, C) \neq 0.} \tag{3.34}$$

The ISF $G(A, C)$ of all decomposition function $g(A, C)$ can be computed by setting the lower bound of $G(A, C)$ to '0' for every value $h_u(C_k) = 1$. It follows that

$$\boxed{\begin{aligned} g_l(A, C) &= f_l(A, C) \wedge \overline{\min_A{}^{\mathrm{k}} f_u(A, C)} \\ g_u(A, C) &= f_u(A, C). \end{aligned}} \tag{3.35}$$

The ISF $H(C)$ of all decomposition functions $h(C)$ depends on the choice of $g_0(A, C) \in \mathbf{G}(A, C)$. If the decomposition function $g_0(A, C)$ is given, the ISF $\mathbf{H}(C) = [h_l, h_u]$ can be computed by (3.19) and (3.12).

Weak AND-Decomposition

Weak AND-decomposition is dual to weak OR-decomposition. An ISF $\mathbf{F}(A, C) = [f_l, f_u]$ is weakly AND-decomposable iff

$$\boxed{f_u(A, C) \vee \max_A{}^{\mathrm{k}} f_l(A, C) \neq 1.} \tag{3.36}$$

The ISF $G(A, C)$ of all decomposition function $g(A, C)$ can be computed by

$$\boxed{\begin{aligned} g_u(A, C) &= f_u(A, C) \vee \overline{\max_A{}^{\mathrm{k}} f_l(A, C)} \\ g_l(A, C) &= f_l(A, C) \end{aligned}} \tag{3.37}$$

The ISF $H(C)$ of all decomposition functions $h(C)$ depends on the choice of $g_0(A, C) \in \mathbf{G}(A, C)$. If the decomposition function $g_0(A, C)$ is given, the ISF $\mathbf{H}(C) = [h_l, h_u]$ can be computed by (3.23).

3.3 Decomposition Systems

3.3.1 Operator and Variable Set Selection for Boolean ISFs

With the theory developed in section 3.2, it is possible to develop algorithms that recursively decompose a given ISF. These algorithms must address a number of problems, such as

1. the selection of free and bound sets,
2. the selection of a decomposition type (OR, AND, EXOR, weak OR, weak AND),
3. the computation of decompositions and
4. the control of the recursive decomposition.

Algorithms for the problems 1.–3. are presented here, while the recursive decomposition algorithm is shown in the next section.

The selection of the free and bound sets is a difficult problem because the number of partitions of a set of variables grows exponentially with the number of variables. It has been shown that decompositions of low complexity are found if the free and bound sets A and B are chosen so that they contain nearly the same number of variables [40]. The algorithm in [40] approximates this decomposition by a greedy algorithm and is presented here.

The algorithm FIND-VARIABLE-SETS gets an operator π and an ISF F as input. The result of the algorithm is a pair $\langle A, B \rangle$ of the variables of the free set A and the bound set B of a π-decomposition of the ISF F. The pair $\langle \emptyset, \emptyset \rangle$ is returned if the function set F is not decomposable with respect to the operator π.

Table 3.3: Formulas to test and compute bi-decompositions of Boolean ISFs.

Type	Test	Free Set	Bound Set
OR	(3.15)	(3.17), (3.10)	(3.19) ,(3.12)
AND	(3.21)	(3.22)	(3.23)
EXOR	TEST-EXOR-DEC	TEST-EXOR-DEC	(3.30), (3.31)
Weak OR	(3.34)	(3.35)	(3.19) ,(3.12)
Weak AND	(3.36)	(3.37)	(3.23)

```
FIND-VARIABLE-SETS(π, F)
 1  X ← SUPPORT(F)
 2  for all a ∈ X do
 3      A ← {a}
 4      for all b ∈ X, b ≠ a do
 5          B ← {b}
 6          if DECOMPOSITION-TEST(π, F, A, B)
 7            then for all c ∈ (X \ A) \ B do
 8                     if DECOMPOSITION-TEST(π, F, A ∪ {c}, B)
 9                       then A ← A ∪ {c}
10                       else if DECOMPOSITION-TEST(π, F, A, B ∪ {c})
11                       then B ← B ∪ {c}
12                     if |A| > |B|
13                       then ⟨A, B⟩ ← SWAP(A, B)
14          return ⟨A, B⟩
15  return ⟨∅, ∅⟩
```

The algorithm SUPPORT(F), called in line 1, returns the support of F, i.e. the set of variables X of ISF $F(X)$. The algorithm FIND-VARIABLE-SETS first looks for an initial solution a single variables in both, the free and bound set respectively (line 6). Then, all remaining variables are tested if they can be moved from the shared set to the free or bound set (lines 6 and 8), where the smaller of the two sets is tested first because of the swap of variables in line 13. The algorithm DECOMPOSITION-TEST called in lines 6, 8 and 10 tests whether the ISF F has a π-decomposition with respect to the free set A and the bound set B by the decomposition criteria that were developed in section 3.2. The decomposition formulas and algorithms of this section are summarized in Table 3.3.

The type of the decomposition is selected by testing each type of decomposition. The strategy for selection of a decomposition type depends on the goal of decomposition. To ensure termination of the decomposition process OR-, AND and EXOR-decompositions should be preferred to weak decompositions. In circuit synthesis OR- and AND-gates are often smaller and faster than EXOR-gates. Therefore, OR- and AND-decompositions should be preferred for these applications. In other applications it may be necessary to minimize the total number of gates. A simple selection algorithm that performs well in many cases was presented in [41]. The best operator is selected by applying the algorithm FIND-VARIABLE-SETS to OR-, AND- and EXOR-decomposition. The sizes of the free and bound sets returned by the algorithm FIND-VARIABLE-SETS are evaluated. The type of decomposition is chosen so that the minimum number of variables in the free and bound sets is minimized. If no OR-, AND- and EXOR-decomposition exists, weak bi-decomposition is applied.

The algorithm FIND-DECOMPOSITION gets an ISF F and returns a decomposition operator π, the free set A and the bound set B.

```
FIND-DECOMPOSITION(F)
 1  X ← SUPPORT(F)
 2  best_cost ← ∞
 3  for π ∈ {'OR', 'AND', 'EXOR'} do
 4      ⟨A, B⟩ ← FIND-VARIABLE-SETS(π, F)
 5      cost ← |X| * min(|A|, |B|) + max(|A|, |B|)
 6      if cost < best_cost
```

```
 7          then best_cost ← cost
 8               best_A ← A
 9               best_B ← B
10               best_π ← π
11  if ⟨best_A, best_B⟩ = ⟨∅, ∅⟩
12     then ⟨best_π, best_A⟩ = FIND-WEAK-DECOMPOSITION(F)
13  return ⟨best_π, best_A, best_B⟩
```

If a function is not bi-decomposable, a weak bi-decomposition is found by the algorithm FIND-WEAK-DECOMPOSITION. The number of variables in the decomposition ISF is not reduced, but additional don't cares are introduced. Most don't cares are introduced if the free set is small. Therefore, only single variables are tried for the free set.

The algorithm FIND-WEAK-DECOMPOSITION gets an ISF $F(A, C)$, the algorithm returns the type π of decomposition and the free set A.

```
FIND-WEAK-DECOMPOSITION(F)
 1  X ← SUPPORT(F)
 2  π ← 'Weak OR'
 3  A ← ∅
 4  for ∀a ∈ X do
 5     if WEAK-OR-TEST(F, {a})
 6        then A ← {a}
 7  if A = ∅
 8     then π ← 'Weak AND'
 9          for ∀a ∈ X do
10             if WEAK-AND-TEST(F, {a})
11                then A ← {a}
12  return ⟨π, A⟩
```

3.3.2 Recursive Bi-Decomposition of Boolean ISFs

After an operator and appropriate free and bound sets have been selected, the decomposition functions must be computed and recursively be decomposed. The decomposition terminates if the decomposition functions depend on only one variable.

The algorithm DECOMPOSE-SIMPLE gets a function f as input and recursively decomposes the function.

```
DECOMPOSE-SIMPLE(f)
1  if f depends on more than one variable
2     then ⟨π, A, B⟩ ← FIND-DECOMPOSITION(f)
3          ⟨g, h⟩ ← COMPUTE-DECOMPOSITION(π, f, A, B)
4          DECOMPOSE-SIMPLE(g)
5          DECOMPOSE-SIMPLE(h)
```

The algorithm first checks for the terminal case in line 1. If the function is not simple, algorithm FIND-DECOMPOSITION in line 2 solves the decomposition problem, and COMPUTE-DECOMPOSITION in line 3 computes a bi-decomposition of f. Lines 4 and 5 recursively decompose the decomposition functions. To simplify the algorithm, the output of the netlist is not shown.

Algorithm DECOMPOSE-SIMPLE can also be applied to decomposition of ISFs. For the decomposition of an ISF F, a function $f \in F$ is selected and passed to the algorithm. This solution however, is not very efficient because the selection of a function f is difficult. There are no efficient algorithms known that can select a function from an ISF so that the function has a simple representation by decomposition.

The selection of a particular function from an ISF can be postponed by decomposition of ISFs. An ISF is decomposed into decomposition ISFs, where the don't cares are restricted to cares only as needed by the decomposition. Each decomposition step takes a small bit from the freedom provided by the don't cares and passes as many don't cares as possible to further stages

of the decomposition. This way the don't cares are used more efficiently than by restricting all don't cares to cares before decomposition.

New problems arise from the decomposition of ISFs. First, decomposition algorithms for ISFs have to be developed. Section 3.2 presented properties for OR-, AND- and EXOR-decomposition of Boolean ISFs. Second, the computation of a decomposition as in line 3 of algorithm DECOMPOSE-SIMPLE is difficult because instead of a pair $\langle g, h \rangle$ of functions, now a pair of ISFs $\langle G, H \rangle$ must be computed. In general, there are many different pairs of such ISFs. An exchange of don't cares between the ISFs G and H is possible. More don't cares in G require fewer don't cares in H and vice versa. In [3] this problem has been solved by first computing ISF G with as many don't cares as possible. Then, the ISF G is completely decomposed and the result, a function $g \in G$ is used to compute the ISF H with as many don't cares as permitted by function the g.

The algorithm DECOMPOSE-ISF gets an ISF F as input. The ISF is fully decomposed and the decomposed function $f \in F$ is returned.

```
DECOMPOSE-ISF(F)
 1  if F depends on less than two variables
 2     then f ← select a member function of F
 3     else ⟨A, B, π⟩ ← FIND-DECOMPOSITION(F)
 4          G ← COMPUTE-FREE-ISF(π, F, A, B)
 5          g ← DECOMPOSE-ISF(G)
 6          H ← COMPUTE-BOUND-ISF(π, F, g, A)
 7          h ← DECOMPOSE-ISF(H)
 8          f ← π(g, h)
 9  return f
```

The algorithm DECOMPOSE-ISF starts with the check for the terminal case in line 1. If the given ISF F depends on less than two variables, the algorithm SIMPLE-DESIGN returns a function $f \in \mathbf{F}$, i.e. one of the four functions 0, 1, a, or $\overline{a}$. If F depends on two or more variables, the algorithm FIND-DECOMPOSITION in line 3 solves the decomposition problem. The algorithm gets the ISF F and returns the free and bound sets A and B respectively, as well as the decomposition operator π of a, possibly weak, bi-decomposition of the ISF. The algorithm COMPUTE-FREE-ISF computes the decomposition ISF G as described in Section 3.2 (see third column of Table 3.3). The ISF G is then recursively decomposed in line 5, with the function g as result. The second decomposition ISF H is computed from the result function g by algorithm COMPUTE-BOUND-ISF (see fourth column of Table 3.3). The second recursive decomposition of H in line 7 gives the result function h. The composition of the decomposition results g and h with the decomposition operator π is returned as the designed result function f in line 9.

Chapter 4

Properties of Discrete Function Sets

4.1 Application of Function Sets

In the past, function sets have found numerous applications. Boolean functions have been generalized to ISFs in many applications. A Boolean ISF can be considered a function interval or a Boolean lattice of functions [4, 6]. These sets of functions have been further extended to general function sets for the solution of Boolean differential equations [35]. Recently, special classes of function sets have been applied to EXOR-decomposition of Boolean ISFs [37].

In this chapter special classes of sets of MVL functions are introduced. Furthermore, basic properties of function sets are shown. The results of this chapter are applied to various decomposition methods in Chapter 5.

MVL ISFs extend MVL functions so that they can describe sets of MVL functions. In some applications, function sets are needed that cannot be described by ISFs. Modsum-decomposition is an example for such an application.

Example 4.1. Consider the function $f(a,b)$ from Figure 4.1(a). Decomposition of $f(a,b)$ results in three pairs of decomposition functions $\langle g_1(a), h_1(b)\rangle$, $\langle g_2(a), h_2(b)\rangle$ and $\langle g_3(a), h_3(b)\rangle$ shown in Figure 4.1(a). There is $f(a,b) = g_i(a) \oplus_3 h_i(b)$ for $i = 1,2,3$. For the synthesis of $f(a,b)$ any of the functions $g_1(a)$, $g_2(a)$ and $g_3(a)$ can be used. However, no single ISF can describe the set of functions $\mathbf{G}(a) = \{g_1(a), g_2(a), g_3(a)\}$. The only ISF that contains $\mathbf{G}(a)$ as a subset is $G'(a) = \Phi$. Obviously, this ISF contains functions, such as $g'(a) = 0$, that cannot be used for synthesis of $f(a,b)$ by the modsum-operator.

There are several ways to solve this problem. One way would be just to ignore additional choices and pick one representative member, say $g_1(a)$, from the set for synthesis. In fact, this method is applied by many synthesis tools. The quality of the solution greatly depends on the choice of the representative. An improvement of this method is the invocation of a heuristic algorithm to

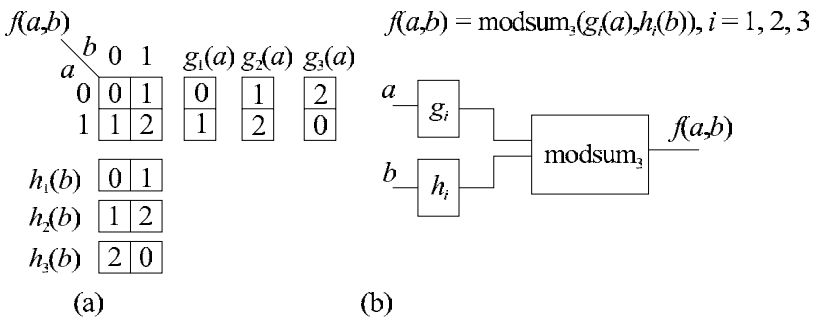

Figure 4.1: Three decompositions of an MVL function $f(a,b)$ by the modsum-Operator. **(a)** Function $f(a,b)$ and the decomposition functions $g_i(a)$ and $h_i(b)$, $i = 1,2,3$. **(b)** Circuit for the decomposition.

guide the choice of the representative. The design of functions by decomposition however, is a process with many stages. In general, it is difficult to estimate the quality of the final solution by inspection of the functions during the early stages of this process.

A more successful way to solve the problem of choice in decomposition is the simultaneous decomposition of many functions from the set of choices. Decomposition of ISFs is an example for this method. The member functions of the ISF are decomposed in a single step and the set is restricted to the subset of decomposable functions. The choice of a promising representative is not made once at the beginning of the design process. Instead, the set is partially restricted to subsets in each decomposition step.

Decomposition of ISFs is very efficient if the function sets can be described by ISFs. As shown in Figure 4.1, there are sets of functions that cannot be described by ISFs. A straightforward approach would describe such sets by a set of ISFs. However, the number of MVL functions is huge, even for a small number of variables. Therefore, the description of function sets by ISFs can require a very large number of ISFs. In many applications, the sets of functions have a very specific structure. Then, it is possible to exploit the properties of this structure and store only the information that is needed.

Example 4.2. It is not necessary to store three functions to describe the set $\mathbf{G}(a)$ from Example 4.1. Instead, it is sufficient to store one function $g(a) = g_1(a)$ and the integer $k = 3$ with the semantics that $\mathbf{G}(a) = \{g(a), g(a) \oplus_3 1, g(a) \oplus_3 1, g(a) \oplus_3 2\}$.

Function sets that share a special structure and have similar properties are called a *class of function sets*. The general definition of function sets is given in Section 4.2. Important classes of function sets that are introduced in Section 4.3. Besides ISFs and MVL relations that have been introduced in Sections 2.3.4 and 2.3.5, function intervals, function lattices and CISFs will be defined. The chapter concludes with a discussion of the relations between these classes of function sets in Section 4.4.

4.2 General Function Sets

The most general form to describe a collection of MVL functions is a *function set*.

Definition 17. A *function set* $\mathbf{F}(A)$ is a nonempty subset $\mathbf{F}(A) \subseteq \mathbb{F}_{m_o}(A)$, of the set of all MVL functions $\mathbb{F}_{m_o}(A)$ that depend on the set of variables A and that have the output cardinality m_o.

Function sets for Boolean functions have first been mentioned in [4]. General function sets describe the solution of Boolean differential equations [35]. Therefore, function sets are of interest, whenever a problem, such as decomposition, can be described by Boolean differential equations. Some properties for sets of Boolean functions can be extended to MVL function sets. It has been shown in Theorem 3 that the set of Boolean functions, the AND, OR and NOT operations form a Boolean lattice. The max- and min-operator are extensions of the Boolean OR and AND to MVL functions. Theorem 3 can be extended to the MVL case.

Theorem 13. *The set $\mathbb{F}_{m_o}(A)$ of all MVL functions that depend on the set of variables A and have the output cardinality m_o is a distributive lattice with the max-operator as sum operation and the min-operator as product operation.*

Proof. Clearly, if $f(A), g(A) \in \mathbb{F}_{m_o}(A)$, then $\max(f(A), g(A))$ and $\min(f(A), g(A))$ have the output cardinality m_o, and therefore, $\max(f(A), g(A)) \in \mathbb{F}_{m_o}(A)$ and $\min(f(A), g(A)) \in \mathbb{F}_{m_o}(A)$.

By Theorem 5 the min- and max-operations satisfy the commutative, associative, absorption and distributive laws. Then, by Theorem 1, the set $\mathbb{F}_{m_o}(A)$ and the min- and max-operators form a distributive lattice. □

Example 4.3. The lattice of all functions $\mathbb{F}_3(a)$ is shown in Figure 4.2. All the functions depend on one variable a with input cardinality $m_a = 2$ and have the output cardinality $m_o = 3$. There are $m_o^{m_a} = 3^2 = 9$ functions that are shown as nodes of the graph. An arrow is drawn from function $f(a)$ to function $g(a)$ for each pair of functions with $f(a) \leq g(a)$ (without reflexive and transitive edges). The smallest function $f_m(a) = 0$ is at the bottom and the largest function $f_M(a) = 2$ is at the top of the graph. Figure 4.3 shows the lattice of all functions $\mathbb{F}_3(\{a, b\})$ that

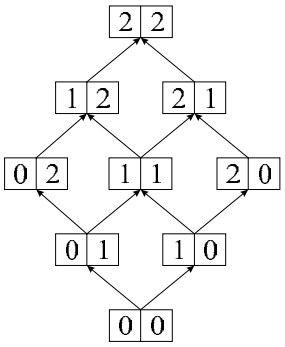

Figure 4.2: Lattice of all functions $\mathbb{F}_3(a)$. The nodes are labeled with $\overline{\lfloor f(0) \mid f(1) \rfloor}$.

depend on the set $\{a, b\}$ of variables with input cardinality $m_a = m_b = 2$ and output cardinality $m_o = 3$. There are $m_o^{m_a * m_b} = 3^{2*2} = 81$ functions. The nodes are labeled with

In contrast to Boolean functions, the set of MVL functions is not a Boolean lattice because the set has no complement operation. The number of elements of a finite Boolean lattice always is a power of 2. The number of MVL functions that depend on a set of variables X is, in general, not a power of 2. Therefore, the set of MVL functions cannot be a model of a Boolean lattice.

It is possible that a function set $\mathbf{F}(A, B)$ contains functions that depend only a subset A of all variables. The function set can then be simplified by restricting the set $\mathbf{F}(A, B)$ to subset $\mathbf{G}(A)$ that contains only the functions that depend on the set of variables A.

Definition 18. A set of variables B is called *inessential* in the function set $\mathbf{F}(A, B)$ if the set contains at least one function $f(A)$ that does not depend on B.

Because function sets are a special class of sets, all operations on sets can be applied to function sets. Especially interesting is the case of intersection. Assume that all functions of the function set $\mathbf{G}(A)$ satisfy a property P_G and all functions of the function set $\mathbf{H}(A)$ satisfy the property P_H. The set $\mathbf{F}(A)$ of all functions that satisfy P_G and P_H is described by the intersection $\mathbf{F}(A) = \mathbf{G}(A) \cap \mathbf{H}(A)$. In Section 4.3 it is shown that many classes of function sets are *closed over set intersection*, i.e. the result of the intersection is a function set of the same class. The intersection of two ISFs for instance is an ISF (or the empty set).

Unfortunately, most function sets are not *closed over set union*. Consider classes of function sets, such as ISFs and relations, that include the sets of a single functions $\{f(A)\}$. If these function sets were closed over set union, it would be possible to construct arbitrary function sets by successive union of single functions. Therefore, only class of all function sets includes the set of single functions and is closed over set union.

For function sets it is necessary characterize the amount of don't cares in the set to develop cost functions for the decomposition algorithms. For ISFs it is easy to compute the number of don't cares. However, the notion of don't care is not defined for function sets. Therefore, a more general definition of the the amount of freedom in a function set must be developed. A very good measure would the size of a function set, i.e. the number of functions contained in the set. This number can become very large and difficult to compute and store. Therefore, formulas are presented that relate the number of don't cares of ISFs to the size of function sets.

Consider a function set $\mathbf{F}(X)$ that depends on the variable set $X = \{x_1, x_2, \ldots, x_n\}$ with the output cardinality m_F and the input cardinalities $m_1, m_2, \ldots, m_n$. How many don't cares does an MVL ISF $\mathbf{I}(X)$ (with the same cardinalities as $\mathbf{F}(X)$) contain that has the same size as the function set $\mathbf{F}(X)$, i.e. $|\mathbf{I}(X)| = |\mathbf{F}(X)|$?

Because a don't care of an MVL ISF can assume any value from the set $\{0, 1, \ldots, m_F - 1\}$, an MVL ISF with k cares contains ${m_F}^k$ functions. Therefore, a function set with $|\mathbf{F}(X)|$ function exhibits a freedom equivalent to an ISF with

$$N_{dc} = \log_{m_F} |\mathbf{F}(X)| \tag{4.1}$$

don't cares.

To compare the amount of don't cares between functions with different support, the number of equivalent don't cares is normalized maximum number don't cares. There are $M = m_1 * m_2 *$

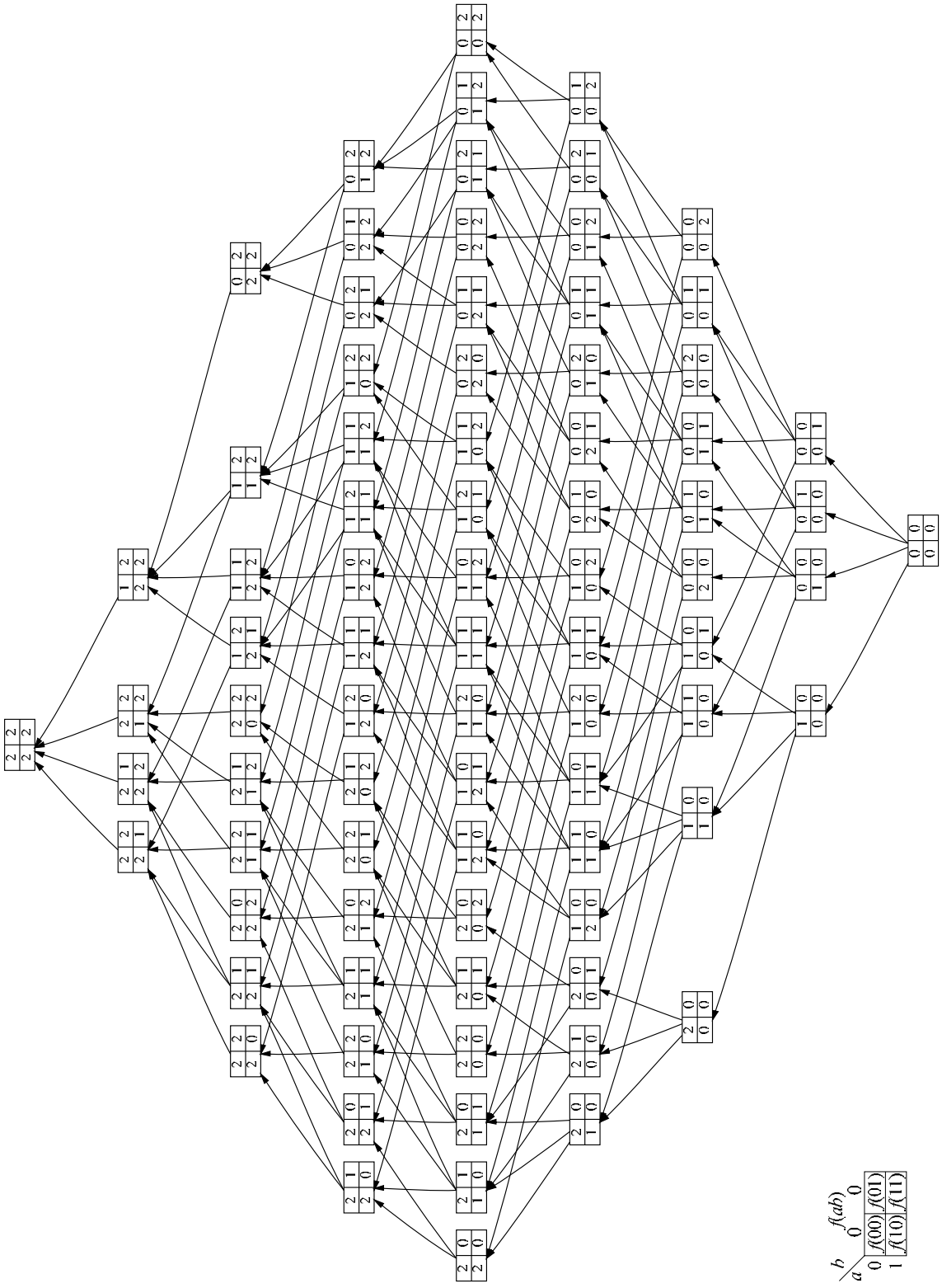

Figure 4.3: Lattice of all functions $\mathbb{F}_3(\{a, b\})$.

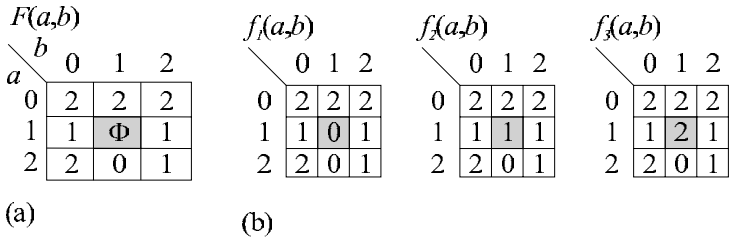

Figure 4.4: An ISF and its characteristic function set **(a)** The ISF $F(a,b)$. **(b)** The characteristic function set $\mathbf{F}(a,b) = \{f_1(a,b), f_2(a,b), f_3(a,b)\}$ of the ISF $F(a,b)$.

$\cdots * m_n$ minterms for the variable set X. Therefore, a function set with support X can contain at most M don't cares and the *ratio of don't cares* becomes

$$R_{dc} = \log_{m_F}|\mathbf{F}(X)|/(m_1 * m_2 * \cdots * m_n). \tag{4.2}$$

Because some function sets in machine learning are very weakly specified, the ratio of equivalent don't cares is very close to 1. On computers with limited precision these ratios would be rounded to 1. Therefore, the *ratio of cares* $R_c = 1 - R_{dc}$ is used as a measure of freedom for function sets:

$$R_c = 1 - \log_{m_F}|\mathbf{F}(X)|/(m_1 * m_2 * \cdots * m_n). \tag{4.3}$$

A value $R_c \lesssim 1$ indicates a completely specified function. A value $R_c = 0$ indicates the set of all functions $\mathbf{F} = \Phi$

4.3 Classes of Function Sets

4.3.1 ISFs as Function Sets

Although MVL ISFs and relations describe sets of MVL functions, they are defined in terms of incompletely specified functions and relations. To relate these sets of functions with other function sets, the term *characteristic function set* is defined. The characteristic function set of an ISF or MVL relation describes the set of all functions that are compatible with the ISF or MVL relation respectively.

The definition and properties of the characteristic function set for ISFs is discussed in this section. The characteristic function set of MVL relations is the topic of Section 4.3.3.

Definition 19. The *characteristic function set* $\mathbf{F}(A)$ of an MVL ISF $F(A)$ with the care set C is the set of all MVL functions $f(A)$ with $F(A_i) = f(A_i)$ for all minterms $A_i \in C$:

$$\mathbf{F}(A) = \{f(A) \mid \forall A_i \in C : f(A_i) = F(A_i)\}. \tag{4.4}$$

Example 4.4. Figure 4.4(a) shows an ISF $F(a,b)$. The characteristic function set

$$\mathbf{F}(a,b) = \{f_1(a,b), f_2(a,b), f_3(a,b)\}$$

of ISF $F(a,b)$ is shown in Figure 4.4(b).

If there is no confusion possible, the term "ISF" will be used as an abbreviation of the term "characteristic function set of the ISF". Both terms are still distinguished by their notation. The ISF $F(A)$ denotes a mapping from a subset $C \subseteq \mathbb{M}_A$ of the set of minterms $\mathbb{M}_A$ into the set $\{0, 1, \ldots m_o - 1\}$, while its characteristic function set $\mathbf{F}(A)$ denotes a set of MVL functions.

Because the characteristic function set of ISFs is a set of functions, the intersection of these sets is well defined.

Example 4.5. Figure 4.5(a) displays two ISFs $G_1(a,b)$ and $H_1(a,b)$ whose intersection is empty. All functions $g_1(a,b) \in \mathbf{G}_1(a,b)$ have $g_1(ab = 11) = 0$ while all functions $h_1(a,b) \in \mathbf{H}_1(a,b)$ have $h_1(ab = 11) = 1$. Therefore, there is $\mathbf{G}_1(a,b) \cap \mathbf{H}_1(a,b) = \emptyset$. Figure 4.5(b) shows two ISFs $G_2(a,b)$ and $H_2(a,b)$ whose intersection is the ISF $F_2(a,b)$. For all functions $g_2(a,b) \in \mathbf{G}_2(a,b)$ and $h_2(a,b) \in \mathbf{H}_2(a,b)$ there is $g_2(ab = 11) = h_2(ab = 11) = 0$. The remaining cares of $\mathbf{G}_2(a,b)$ and $\mathbf{H}_2(a,b)$ restrict the don't cares of the other ISF.

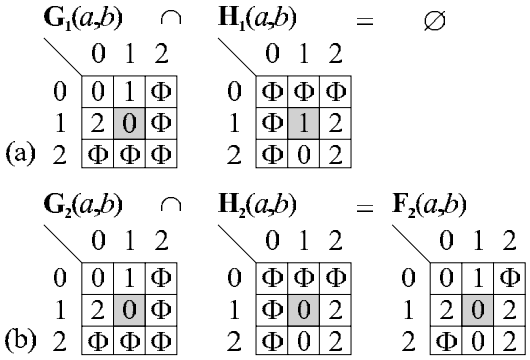

Figure 4.5: Intersection of ISF. **(a)** Empty intersection of $G_1(a,b)$ and $H_1(a,b)$. **(b)** Intersection of $G_2(a,b)$ and $H_2(a,b)$ is specialization of don't cares.

Theorem 14. *Let $G(A)$ and $H(A)$ be two ISFs, and let the function set $\mathbf{F}(A)$ be the intersection $\mathbf{F}(A) = \mathbf{G}(A) \cap \mathbf{H}(A)$ of the two ISFs. Then, there is*

1. *$\mathbf{F}(A) \neq \emptyset$ if and only if for all minterms A_i with $G(A_i) \neq \Phi$ and $H(A_i) \neq \Phi$ there is*

$$G(A_i) = H(A_i). \tag{4.5}$$

2. *If $\mathbf{F}(A) \neq \emptyset$, then there is*

$$F(A_j) = \begin{cases} G(A_j) & \text{for } H(A_j) = \Phi, \\ H(A_j) & \text{otherwise.} \end{cases} \tag{4.6}$$

Proof. Obviously, if there is a minterm A_i where $G(A_i)$ and $H(A_i)$ have cares with different values, no function $f(A)$ can be element of both ISFs and the intersection is empty.

If the values of all cares of $G(A)$ and $H(A)$ agree, the cares of one ISF specialize the don't cares of the other ISF. The result is the ISF $F(A)$ whose cares are the cares of $G(A)$ and $H(A)$ as shown in (4.6). □

The first part of Theorem 14 can be stated informally that the intersection of two ISFs is not empty if the values of all cares of the ISFs agree. The second part of Theorem 14 states that the intersection of two ISFs is the union of their cares.

4.3.2 Function Intervals

A Boolean ISF can be described as an interval of functions $F(A) = [f_l, f_u]$, see theorem 4. An interval of functions can also be defined for MVL functions. It will be shown in this section that MVL function intervals generalize MVL ISFs. As demonstrated in Sections 5.5 and 5.6, these function sets from the basis of min- and max-decomposition of MVL functions.

Definition 20. A *function interval* $\mathbf{F}(A) = [f_l, f_u]$ with $f_l(A) \leq f_u(A)$ is the set of MVL functions

$$\mathbf{F}(A) = \{f(A) | f_l(A) \leq f(A) \leq f_u(A)\}. \tag{4.7}$$

A function interval $\mathbf{F}(A) = [f_l, f_u]$ can be visualized by a map similar to ISFs. For a minterm A_i, the bounds $f_l(A_i)$ and $f_u(A_i)$ are shown as an interval $[f_l(A_i), f_u(A_i)]$ in the field A_i of the map. If $f_l(A_i) = f_u(A_i)$, then the minterm A_i specifies a *care* and only the value $f_l(A_i)$ of the care is shown in the field A_i.

Example 4.6. Figure 4.6(a) shows the map of the function interval $\mathbf{F}(a,b) = [f_1, f_4]$, where the functions $f_1(a,b)$ and $f_4(a,b)$ are displayed in Figure 4.6(b). The functions of the characteristic function set

$$\mathbf{F}(a,b) = \{f_1(a,b), f_2(a,b), f_3(a,b), f_4(a,b)\}$$

are also displayed in Figure 4.6(b).

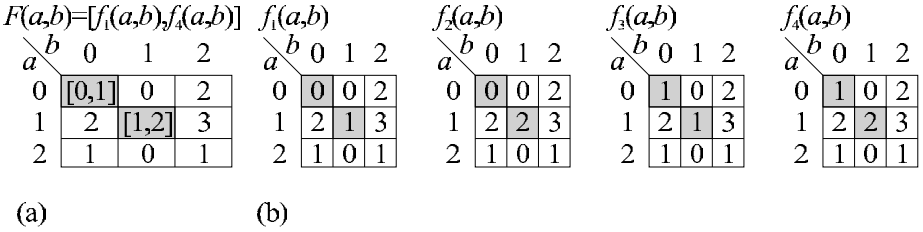

Figure 4.6: Characteristic function set of a function interval. **(a)** Display of $\mathbf{F}(a,b) = [f_1, f_4]$ as map. **(b)** Characteristic function set $\mathbf{F}(a,b) = \{f_1(a,b), f_2(a,b), f_3(a,b), f_4(a,b)\}$.

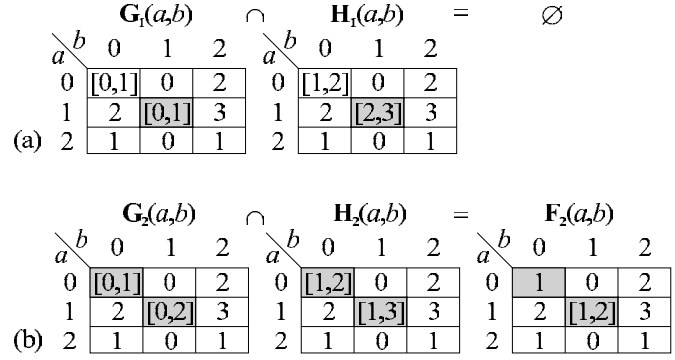

Figure 4.7: Intersection of function intervals. **(a)** Empty intersection of $\mathbf{G}_1(a,b)$ and $\mathbf{H}_1(a,b)$. **(b)** Function interval $\mathbf{F}_2(a,b)$, intersection of $\mathbf{G}_2(a,b)$ and $\mathbf{H}_2(a,b)$.

Analogous to ISFs, the intersection of two function intervals is a function interval (or empty).

Example 4.7. Figure 4.7(a) displays two function intervals $\mathbf{G}_1(a,b)$ and $\mathbf{H}_1(a,b)$ whose intersection is empty.All functions $g_1(a,b) \in \mathbf{G}_1(a,b)$ have $g_1(ab=11) \leq 1$ while all functions $h_1(a,b) \in \mathbf{H}_1(a,b)$ have $h_1(ab=11) \geq 2$. Therefore, there is $g_1(a,b) \notin \mathbf{H}_1(a,b)$ for all functions $g_1(a,b) \in \mathbf{G}_1(a,b)$ which implies $\mathbf{G}_1(a,b) \cap \mathbf{H}_1(a,b) = \emptyset$.

Figure 4.7(b) shows two function intervals $\mathbf{G}_2(a,b)$ and $\mathbf{H}_2(a,b)$ whose intersection is the function interval $\mathbf{F}_2(a,b)$. For all functions $g_2(a,b) \in \mathbf{G}_2(a,b)$ and $h_2(a,b) \in \mathbf{H}_2(a,b)$ there is $0 \leq g_2(ab=11) \leq 2$ and $1 \leq h_2(ab=11) \leq 3$. Therefore, for all functions $f_2(a,b) \in \mathbf{G}_2(a,b) \cap \mathbf{H}_2(a,b)$ there is $1 \leq f_2(ab=11) \leq 2$. Similarly, there is $g_2(ab=00) \leq 1$ and $h_2(ab=00) \geq 1$ it follows that the only value for $f_2(ab=00)$ is $f_2(ab=00) = 1$.

Theorem 15. *Let* $\mathbf{G}(A) = [g_l, g_u]$ *and* $\mathbf{H}(A) = [h_l, h_u]$ *be two non-empty function intervals, and let the function set* $\mathbf{F}(A)$ *be the intersection* $\mathbf{F}(A) = \mathbf{G}(A) \cap \mathbf{H}(A)$ *of the two intervals. Then, there is* $\mathbf{F}(A) \neq \emptyset$ *if and only if*

$$g_l(A) \leq h_u(A) \text{ and} \tag{4.8a}$$
$$h_l(A) \leq g_u(A). \tag{4.8b}$$

Furthermore, if $\mathbf{F}(A) \neq \emptyset$ *then, there is* $\mathbf{F}(A) = [f_l, f_u]$ *with*

$$f_l(A) = \max(g_l(A), h_l(A)) \text{ and} \tag{4.9a}$$
$$f_u(A) = \min(g_u(A), h_u(A)). \tag{4.9b}$$

Proof. A function $f(A)$ is element of $\mathbf{G}(A) \cap \mathbf{H}(A)$ if and only if there is

$$g_l(A) \leq f(A), \tag{4.10a}$$
$$h_l(A) \leq f(A), \tag{4.10b}$$
$$f(A) \leq g_u(A) \text{ and} \tag{4.10c}$$
$$f(A) \leq h_u(A). \tag{4.10d}$$

Substitution of (4.10a) into (4.10d) results in (4.8a), and substitution of (4.10b) into (4.10c) results in (4.8b). Therefore, if there is a function $f(A) \in \mathbf{G}(A) \cap \mathbf{H}(A)$, then the inequalities (4.8) are satisfied.

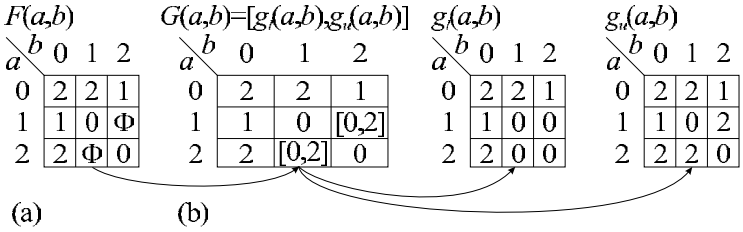

Figure 4.8: ISF and function interval with the same characteristic function set **(a)** ISF $F(a,b)$. **(b)** Function interval $\mathbf{G}(a,b) = [g_l, g_u]$, and bounds $g_l(a,b)$ and $g_u(a,b)$.

The function $f(A) = \max(g_l(A), h_l(A))$ satisfies the inequalities (4.10a) and (4.10b). Assume that the inequalitites (4.8) are satisfied. Because $\mathbf{G}(A)$ is non-empty, there is $g_l(A) \leq g_u(A)$. Together with (4.8b) it is shown that $f(A)$ satisfies (4.10c). Similarly, (4.10d) follows from $h_l(A) \leq h_u(A)$ and (4.8a). Therefore, if (4.8) is satisfied, $\mathbf{G}(A) \cap \mathbf{H}(A)$ is not empty.

Equations (4.10a) and (4.10b) are equivalent to $\max(g_l(A), h_l(A)) \leq f(A)$ and equations (4.10a) and (4.10b) are equivalent to $f(A) \leq \min(g_u(A), h_u(A))$. Therefore, there is $f(A) \in \mathbf{G}(A) \cap \mathbf{H}(A)$ iff $f(A) \in \mathbf{F}(A)$ as defined by (4.9). □

Compared in terms of the characteristic function set, function intervals generalize MVL ISFs.

Example 4.8. The ISF $F(a,b)$ from Figure 4.8(a) can be described by the function interval $\mathbf{G}(a,b) = [g_l, g_u]$ shown in Figure 4.8(b). The cares $F(a_i, b_j) = v_k$ of the ISF $F(a,b)$ are replaced by cares of the function interval with $g_l(a_i, b_j) = g_u(a_i, b_j) = v_k$. The don't cares $F(a_k, b_l) = \Phi$ are translated into intervals with $g_l(a_k, b_l) = 0$ and $g_l(a_k, b_l) = m_o - 1$ where $m_o = 3$ is the output cardinality of the ISF $F(a,b)$.

Theorem 16. *Let $F(A)$ be an ISF and $\mathbf{G}(A) = [g_l, g_u]$ be a function interval with*

$$g_l(A_i) = \begin{cases} 0 & \text{for } F(A_i) = \Phi, \\ F(A_i) & \text{otherwise,} \end{cases} \tag{4.11a}$$

$$g_u(A_i) = \begin{cases} m_o - 1 & \text{for } F(A_i) = \Phi, \\ F(A_i) & \text{otherwise,} \end{cases} \tag{4.11b}$$

where m_o is the output cardinality of the ISF $F(A)$. Then, there is $\mathbf{F}(A) = \mathbf{G}(A)$.

Proof. Let $f(A)$ be a member function $f(A) \in \mathbf{F}(A)$ of ISF $F(A)$. Two cases are distinguished by the values of $F(A_i)$.

1. For $F(A_i) = \Phi$, by (4.11), there is $g_l(A_i) = 0 \leq f(A_i) \leq m_o - 1 = g_u(A_i)$.
2. Otherwise there is $F(A_c) \neq \Phi$ and

$$F(A_i) = f(A_i) = g_l(A_i) = g_u(A_i) \tag{4.12}$$

From the cases 1 and 2 follows $g_l(A) \leq f(A) \leq g_u(A)$ and $f(A) \in \mathbf{G}(A)$. It follows that

$$\mathbf{F} \subseteq \mathbf{G} \tag{4.13}$$

Now let $g(A)$ be a member function $g(A) \in \mathbf{G}(A)$ of the function interval $\mathbf{G}(A)$. For $F(A_c) \neq \Phi$, (4.12) still holds. Therefore, there is $g(A_c) = F(A_c)$ for all elements A_c of the care set of ISF $F(A)$, and $g(A) \in \mathbf{F}(A)$. It follows that $\mathbf{G}(A) \subseteq \mathbf{F}(A)$. Together with (4.13) there is $\mathbf{F}(A) = \mathbf{G}(A)$. □

The converse of Theorem 16 is not true. There are function intervals whose characteristic function set cannot be defined by an ISF.

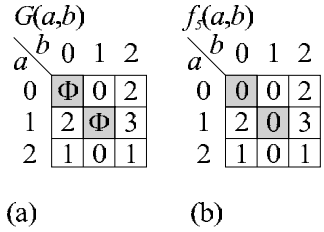

Figure 4.9: ISF $F_S(a,b)$ containing the characteristic function set $\mathbf{F}(a,b)$ of interval $F(a,b)$ from Figure 4.6(a). **(a)** ISF $F_N(a,b)$. **(b)** Member function $f_5(a,b) \in \mathbf{F_S}(a,b)$, that is not element of $\mathbf{F}(a,b)$.

Example 4.9. Consider the function interval $\mathbf{F}(a,b) = [f_1, f_2]$ from Figure 4.6(a). Obviously an ISF $F_S(a,b)$ that contains the member functions $f_1(a,b) \dots f_4(a,b)$ of the interval $\mathbf{F}(a,b)$ must have don't cares at $ab = 00$ and $ab = 11$ where the interval $F(a,b)$ specifies more than one function value. The ISF $F_S(a,b)$ containing all functions from $\mathbf{F}(a,b)$ is shown in Figure 4.9. However, the ISF $F_S(a,b)$ contains the function $f_5(a,b)$ as shown in Figure 4.9(b), and function $f_5(a,b)$ is not an element of the function interval $\mathbf{F}(a,b) = \{f_1(a,b) \dots f_4(a,b)\}$ shown in Figure 4.6(b).

The theorem below demonstrates a method to simplify function intervals by removing inessential variables.

Theorem 17. *The set of variables B is inessential in the function interval $\mathbf{F}(A,B) = [f_l, f_u]$ iff*

$$\max_B{}^{\mathrm{k}} f_l(A,B) \leq \min_A{}^{\mathrm{k}} f_u(A,B). \tag{4.14}$$

Furthermore, the subset $\mathbf{G}(A) \subseteq \mathbf{F}(A,B)$ of all functions that does not depend on the set of variables B is a function interval

$$\mathbf{G}(A) = [\max_B{}^{\mathrm{k}} f_l(A,B), \min_B{}^{\mathrm{k}} f_u(A,B)] \tag{4.15}$$

Proof. The theorem follows directly from the Theorems 10 and 11. □

The author of this work believes that function intervals will find many applications [17]. The use of function intervals in decomposition is demonstrated in Chapter 5. It is also possible that function intervals arise from applications in machine learning. The value of an attribute (such as age for instance) may not be known precisely but can be specified as an interval. In Chapter 8 test cases from real world applications are shown that can be described by function intervals, but not by ISFs.

4.3.3 MVL Relations as Function Sets

The don't care of an ISF lets the function value completely undefined for that minterm. Function intervals give more control to specify partially known values by intervals. The most general way to specify the value for a minterm is an arbitrary subset of of the set of all possible values by MVL relations. To relate the functions specified by MVL relations, to other function sets, a characteristic function set is defined for MVL relations.

Definition 21. The *characteristic function set* $\mathbf{R}(A)$ of an MVL relation $R\langle A, v\rangle$ is the set of all functions $f(A)$ whose function values agree with the values specified by the relation $R\langle A, v\rangle$,

$$\mathbf{R}(A) = \{f(A) \,|\, \forall A_i \in \mathbb{M}_A : \langle A_i, f(A_i)\rangle \in R\langle A, v\rangle\}. \tag{4.16}$$

Example 4.10. The characteristic function set

$$\mathbf{F}(a,b) = \{f_1(a,b), f_2(a,b), f_3(a,b), f_4(a,b)\} \tag{4.17}$$

of the MVL relation $F\langle(a,b), v\rangle$ from Figure 4.10(a) is shown in Figure 4.10(b). The functions of the relation can have two values ($y = 1$ or $y = 3$) at the special don't care at $ab = 00$ and two values ($y = 0$ and $y = 2$) at the special don't care at $ab = 11$. Therefore, the characteristic function set $\mathbf{F}(a,b)$ of the relation $F\langle(a,b), v\rangle$ contains $2 * 2 = 4$ functions.

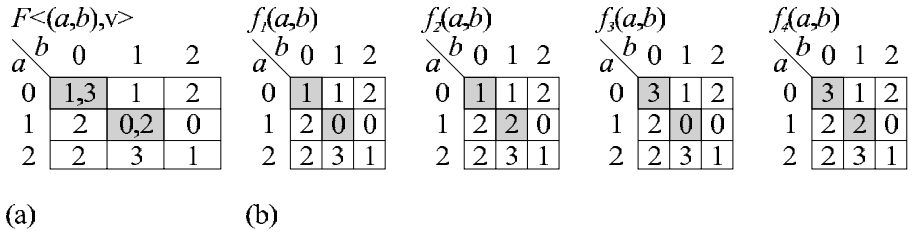

Figure 4.10: Characteristic function set of an MVL relation **(a)** MVL relation $F\langle(a,b),v\rangle$. **(b)** Characteristic function set $\mathbf{F}(a,b) = \{f_1(a,b), f_2(a,b), f_3(a,b), f_4(a,b)\}$ of the relation $F\langle(a,b),v\rangle$.

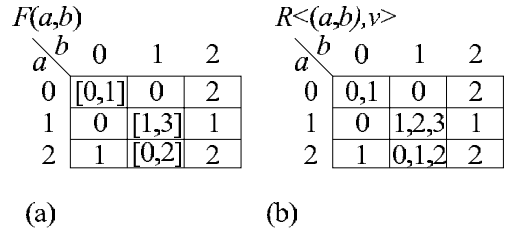

Figure 4.11: Function interval and MVL relation with the same characteristic function set. **(a)** Function interval $F(a,b)$. **(b)** MVL relation $R\langle(a,b),v\rangle$.

Similar to ISFs, the term "MVL relation" will be used instead of the term "characteristic function set of the relation" if there is no confusion possible. However, it is important, to distinguish between the relation $R\langle A,v\rangle$ which is a set of pairs of minterms A_i and MVL constants v_i, and the function set $\mathbf{R}(A)$ which is a set of MVL functions.

It is easy to see that MVL relations generalize function intervals.

Example 4.11. Consider the function interval $\mathbf{F}(a,b)$ in Figure 4.11(a). The characteristic function set of interval $\mathbf{F}(a,b)$ can be described by an MVL relation $R\langle(a,b),v\rangle$ by enumerating the values of the intervals as shown in Figure 4.11(b).

Theorem 18. *For every function interval* $\mathbf{F}(A) = [f_l, f_u]$ *there is an MVL relation* $R\langle A,v\rangle$ *with the same characteristic function set, i.e.* $\mathbf{F}(A) = \mathbf{R}(A)$, *where*

$$R\langle A,v\rangle = \{\langle A_i, v_j\rangle \mid f_l(A_i) \le v_j \le f_u(A_i)\}. \tag{4.18}$$

Proof. For every minterm A_i the values of the interval $[f_l(A_i), f_u(A_i)]$ can be enumerated and transformed into a relation between the minterm A_i and the MVL value v_j, which is expressed by (4.18). □

The converse of theorem 18 is not true. There are MVL relations whose characteristic function set cannot be described by a function interval.

Example 4.12. Consider the MVL relation $F\langle(a,b),y\rangle$ from Figure 4.10(a). To include all member functions of $F\langle(a,b),y\rangle$, a function interval must specify for the lower bound $f_l(ab = 00) = 1$ and for the upper bound $f_u(ab = 00) = 3$. However, the interval also includes the value $f_l(ab = 00) \le 2 \le f_u(ab = 00)$, which is not an element of the member functions of the MVL relation $F\langle(a,b),y\rangle$, see Figure 4.10(b). Therefore, the MVL relation cannot be described by a function interval.

In some data structures, such as MDDs, function intervals are easier to store than MVL relations. For each minterm a function interval is specified by its lower and upper bound, i.e. by two integers in the range $[0, m_o - 1]$. The encoding of these bounds on a computer takes $2\lceil \log_2 m_o \rceil$ bits. A value of an MVL relation requires one bit for each possible output value, i.e. m_o bits, which is, for $m_o \ge 4$, more than for the bounds of function intervals. If a particular data structure, such as BEMDD for instance, has an efficient representation for relations, this data structure can also be applied to store function intervals [25].

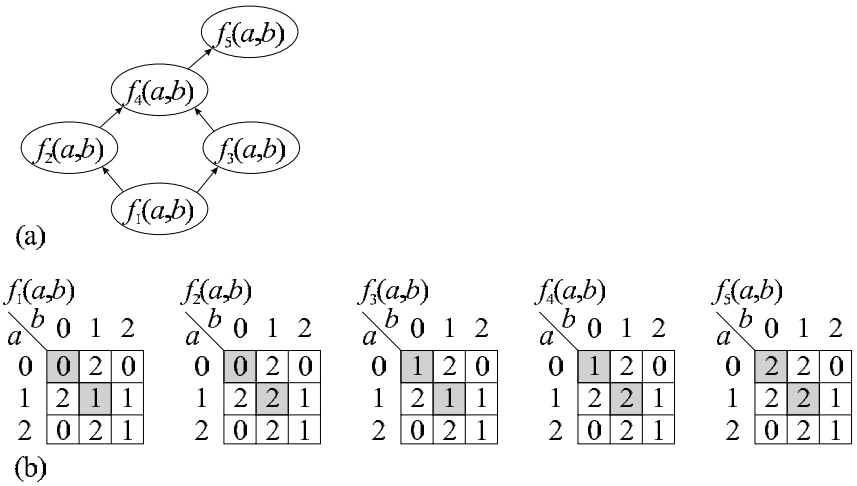

Figure 4.12: Function lattice $\mathbf{F}(a,b)$ **(a)** Chart of the function lattice $\mathbf{F}(a,b)$. **(b)** Member functions of the function lattice $\mathbf{F}(a,b) = \{f_1(a,b), f_2(a,b), f_3(a,b), f_4(a,b), f_5(a,b)\}$.

4.3.4 Function Lattices

In Theorem 4 it was shown that the characteristic function set of a Boolean ISF forms a Boolean lattice. For MVL functions, it is possible to embed a sublattice into the lattice $\mathbb{F}_{m_o}(X)$ of all functions.

Definition 22. An *MVL function lattice* is a sublattice of the lattice of all functions $\mathbb{F}_{m_o}(X)$ with respect to the min- and max-operations.

Example 4.13. Figure 4.12(a) shows a function lattice $\mathbf{F}(a,b)$ that consists of the five functions shown in Figure 4.12(b)

$$\mathbf{F}(a,b) = \{f_1(a,b), f_2(a,b), f_3(a,b), f_4(a,b)\}. \tag{4.19}$$

It can easily be verified that min and max of any two functions of $\mathbf{F}(a,b)$ also are elements of $\mathbf{F}(a,b)$, such as $\min(f_2(a,b), f_3(a,b)) = f_1(a,b)$.

Theorem 19. *All function lattices are distributive.*

Proof. The distributivity follows directly from the distributive laws for the min- and max-operators on integers in Theorem 2. □

It is easy to show that the intersection of two function lattices is either empty or a function lattice.

Theorem 20. *If the intersection of two function lattices is not empty, the result of the intersection is a function lattice.*

Proof. Let $\mathbf{G}(A)$ and $\mathbf{H}(A)$ be two arbitrary function lattices, and assume that their intersection $\mathbf{F}(A) = \mathbf{G}(A) \cap \mathbf{H}(A)$ is not empty. Let $f_1(A)$ and $f_2(A)$ be two arbitrary member functions of the set $\mathbf{F}(A)$. Because $\mathbf{F}(A)$ is the intersection of $\mathbf{G}(A)$ and $\mathbf{H}(A)$, $f_1(A)$ and $f_2(A)$ are also member functions of $\mathbf{G}(A)$ and $\mathbf{H}(A)$. Now, $\mathbf{G}(A)$ and $\mathbf{H}(A)$ are function lattices. Therefore, minimum and maximum of $f_1(A)$ and $f_2(A)$ also are members of $\mathbf{G}(A)$ and $\mathbf{H}(A)$, and hence, are members of $\mathbf{F}(A)$. Therefore, $\mathbf{F}(A)$ is a function lattice. □

It is shown in Chapter 5.3 that function lattices arise during bi-decomposition of function intervals. For the Boolean case ISFs are Boolean lattices of functions. Similarly, MVL relations are generalized by function lattices.

Example 4.14. Figure 4.13(a) shows an MVL relation $F\langle a, v\rangle$. The characteristic function set $\mathbf{F}(a) = \{f_1(a), \ldots, f_6(a)\}$ is shown in Figure 4.13(b). Figure 4.13(c) shows the graph of the function lattice of $\mathbf{F}(a)$. It is easy to verify that min and max of any two functions from the set $\mathbf{F}(a)$ are also elements of $\mathbf{F}(a)$. Therefore, the characteristic function set of the relation $F\langle a, v\rangle$ is a sublattice of the lattice of all functions.

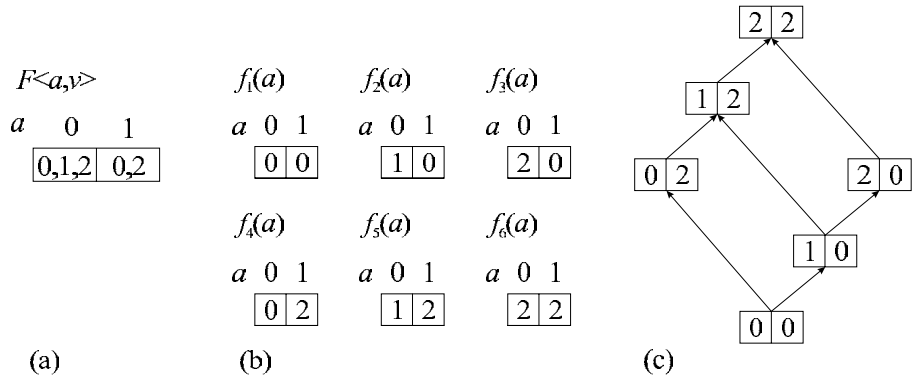

Figure 4.13: Function lattice of an MVL relation. **(a)** MVL relation $F\langle a, v\rangle$. **(b)** Characteristic function set $\mathbf{F}(a) = \{f_1(a), \ldots, f_6(a)\}$. **(c)** Chart of the function lattice $\mathbf{F}(a)$.

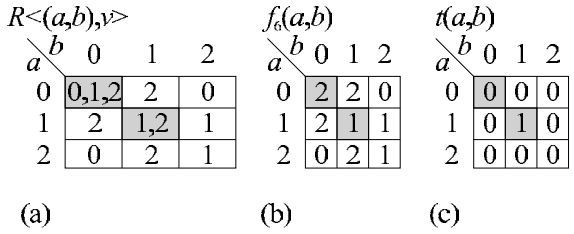

Figure 4.14: Properties of function lattices. **(a)** MVL relation $R\langle (a, b), v\rangle$ containing the functions of lattice $\mathbf{F}(a, b)$ from Figure 4.12. **(b)** Function $f_6(a, b) \in \mathbf{R}(a, b)$ that is not element of $\mathbf{F}(a, b)$. **(c)** Parameter function $t(a, b)$ to show that function lattices are not input independent.

Theorem 21. *The characteristic function set* $\mathbf{R}(A)$ *of an MVL relation* $R\langle A, v\rangle$ *forms a function lattice.*

Proof. The proof is based on the property that the maximum of two integers a and b is equal to one of the integers, e.g. $\max(a, b) \in \{a, b\}$ and $\min(a, b) \in \{a, b\}$.

Let $g(A)$ and $h(A)$ be two functions with $g(A), h(A) \in \mathbf{R}(A)$, and let $f(A) = \max(g(A), h(A))$. For every minterm A_i there is $\langle A_i, g(A_i)\rangle \in R\langle A, v\rangle$ and $\langle A_i, h(A_i)\rangle \in R\langle A, v\rangle$ Therefore, there is

$$\langle A_i, \max(g(A_i), h(A_i)\rangle \in R\langle A, v\rangle \tag{4.20}$$

which implies $\langle A_i, f(A_i)\rangle \in R\langle A, v\rangle$ for all minterms A_i. By definition 21 there is $f(A) \in \mathbf{R}(A)$.

A proof with dual arguments shows that $\min(g(A), h(A)) \in \mathbf{R}(A)$. Together with Theorems 1 and 5 it follows that the characteristic function set $\mathbf{R}(A)$ is a function lattice. □

As shown in Example 4.13, the number of functions in an MVL relation is generally not a power of two, and the characteristic function set of an MVL ISF cannot be a model of a Boolean lattice.

The converse of Theorem 21 is not true. There are function lattices that cannot be described by MVL relations.

Example 4.15. See the function lattice $\mathbf{F}(a, b)$ from Figure 4.12. For the minterm $ab = 00$, the functions of the lattice assume the values 0, 1 and 2; and for the minterm $ab = 11$ the values 1 and 2 are possible. The corresponding MVL relation $R\langle (a, b), v\rangle$ is shown in Figure 4.14(a). However, the MVL relation $R\langle (a, b), v\rangle$, contains a function $f_6(a, b)$, shown in Figure 4.14(b) that is not element of the function lattice $\mathbf{F}(a, b)$. Therefore, it is not possible to describe the functions of the function lattice $\mathbf{F}(a, b)$ by an MVL relation.

There is an important difference between MVL relations and function lattices. For an MVL relation it is possible to combine the function values of two member functions and the result also is a member function of the MVL relation. The definition below gives a formal definition of the combination process, where the selection is done by a two-valued parameter function $t(A)$ that is a mapping $\mathbb{M}_A \rightarrow \{0, 1\}$.

Definition 23. A function set $\mathbf{F}(A)$ is *input independent* if$_{\text{DEF}}$ for any two member functions $g(A), h(A) \in \mathbf{F}(A)$ and any two-valued parameter function $t(A)$, the function

$$f(A) = g(A) * t(A) + h(A) * (1 - t(A)) \tag{4.21}$$

also is a member function $f(A) \in \mathbf{F}(A)$.

The function value $f(A_i)$ is either $g(A_i)$, for $t(A) = 1$, or $h(A)$, for $t(A) = 0$. Input independence is an important property because it allows local optimization of a function. For the minterm A_i the function value of $f(A_i)$ can be taken from the set $\{g(A_i), h(A_i)\}$ without looking at other minterms A_j.

Theorem 22. *The characteristic function set of MVL relations is input independent.*

Proof. Consider the MVL relation $R\langle A, v\rangle$. Let $g(A), h(A) \in \mathbf{R}(A)$ be two member functions of $R\langle A, v\rangle$, and let $t(A)$ be an arbitrary two-valued function. It will be shown that $f(A)$ defined by (4.21) also is a member function of $R\langle A, v\rangle$. Two cases are distinguished by the values of $t(A_i)$.

1. If $t(A_i) = 0$, there is $f(A_i) = h(A_i)$, and because of $\langle A_i, h(A_i)\rangle \in R\langle A, v\rangle$, there is $\langle A_i, f(A_i)\rangle \in R\langle A, v\rangle$.
2. Otherwise there is $t(A_i) = 1$. Then, $f(A_i) = g(A_i)$, and because of $\langle A_i, g(A_i)\rangle \in R\langle A, v\rangle$, there is $\langle A_i, f(A_i)\rangle \in R\langle A, v\rangle$.

It follows that $\langle A_i, f(A_i\rangle \in R\langle A, v\rangle$ for every minterm A_i. Therefore, $f(A) \in \mathbf{R}(A)$ is a member function of $R\langle A, v\rangle$, and the MVL relation is input independent. □

In contrast to MVL relations, this property does not hold for function lattices.

Example 4.16. Consider the function lattice $F(a, b)$ from Figure 4.12. For the parameter function $t(a, b)$ shown in Figure 4.14(c) there is

$$f_6(a, b) = f_3(a, b) * t(a, b) + f_5(a, b) * (1 - t(a, b)), \tag{4.22}$$

where the function $f_6(a, b)$ is shown in Figure 4.14(b). However, the function $f_6(a, b)$, is not an element of the function lattice $\mathbf{F}(a, b)$. Therefore, the function lattice is not input independent.

Because function lattices are not input independent, they will be much more difficult to optimize than MVL relations.

Below, Theorem 23 shows that MVL relations and input independence are equivalent. Before that, Lemma 4.1 extends the definition of input independence from two to n functions.

Lemma 4.1. *Let $\mathbf{F}(A)$ be an input independent function set, and let $g_0(A), g_1(A), \ldots, g_{n-1}(A) \in \mathbf{F}(A)$ be n member functions of $\mathbf{F}(A)$. For any n-valued parameter function $t(A)$, the function*

$$f(A) = \begin{cases} g_0(A_i) & \text{for } t(A_i) = 0, \\ g_1(A_i) & \text{for } t(A_i) = 1, \\ \vdots & \\ g_{n-1}(A_i) & \text{for } t(A_i) = n - 1 \end{cases} \tag{4.23}$$

is a member function $f(A) \in \mathbf{F}(A)$ of the function set.

Proof. The lemma is proven by induction on the number n of functions. For $n = 2$ the hypothesis follows from (4.21). Assume the hypothesis has been proven for $n = k - 1$. It remains to show that $f(A)$ from (4.23) is a member function of $\mathbf{F}(A)$ for $n = k$.

The function $f(A)$ can be composed in two stages. The first stage combines the function values of the first $k - 1$ functions $g_0(A), g_1(A), \ldots, g_{k-2}(A)$ into a function $h(A)$. The second stage composes function $f(A)$ from the functions $h(A)$ and $g_{k-1}(A)$.

Given an arbitrary k-valued parameter function $t(A)$, let

$$r(A) = \begin{cases} 0 & \text{for } t(A_i) = k - 1, \\ t(A_i) & \text{otherwise} \end{cases} \tag{4.24}$$

be a $(k-1)$-valued function. Then, by the induction hypothesis, the function

$$h(A) = \begin{cases} g_0(A_i) & \text{for } r(A_i) = 0, \\ g_1(A_i) & \text{for } r(A_i) = 1, \\ \vdots & \\ g_{k-2}(A_i) & \text{for } r(A_i) = k-2 \end{cases} \tag{4.25}$$

is a member function $h(A) \in \mathbf{F}(A)$ of the function set. Let

$$s(A) = \begin{cases} 0 & \text{for } t(A_i) = k-1, \\ 1 & \text{otherwise} \end{cases} \tag{4.26}$$

be a two-valued parameter function. Then, there is $f(A) = h(A) * s(A) + g_{k-1}(A) * (1 - s(A))$. Because $h(A)$ and $g_{k-1}(A)$ are member functions of $\mathbf{F}(A)$, by definition of the input independence, $f(A)$ is also a member function of $\mathbf{F}(A)$. □

Theorem 23. *A function set $\mathbf{F}(A)$ is input independent if and only if it is the characteristic function set of an MVL relation $R\langle A, v\rangle$, i.e. $\mathbf{R}(A) = \mathbf{F}(A)$.*

Proof. If the function set $\mathbf{F}(A)$ is not input independent, theorem 22 shows that there is no MVL relation that has $\mathbf{F}(A)$ as characteristic function set. It remains to show that there is an MVL relation $R\langle A, v\rangle$ with $\mathbf{R}(A) = \mathbf{F}(A)$ if the function set $\mathbf{F}(A)$ is input independent.

Let $\mathbf{F}(A) = \{f_0(A), f_1(A), \ldots, f_{n-1}(A)\}$ have n member functions. The MVL relation $R\langle A, v\rangle$ is constructed as the union of the function values of the member functions of $\mathbf{F}(A)$

$$R\langle A, v\rangle = \bigcup_{k=0}^{n-1} \bigcup_{\forall A_i} \langle A_i, f_k(A_i)\rangle . \tag{4.27}$$

Obviously, there is $\langle A_i, f_k(A_i)\rangle \in R$ for all minterms A_i and $k = 0, \ldots, n-1$. Therefore, $f_k(A) \in \mathbf{R}(A)$ for $k = 0, \ldots, n-1$, and

$$\mathbf{F}(A) \subseteq \mathbf{R}(A). \tag{4.28}$$

Let $g(A) \in \mathbf{R}(A)$ be a member function of the MVL relation. Because of (4.27), there is for every minterm A_i a function $f_k(A)$ with $g(A_i) = f_k(A_i)$. Let $t(A)$ be the function of the indices k, i.e. let $g(A_i) = f_{t(A_i)}(A_i)$. Then there is

$$g(A) = \begin{cases} f_0(A_i) & \text{for } t(A_i) = 0 \\ f_1(A_i) & \text{for } t(A_i) = 1 \\ \vdots & \\ f_{n-1}(A_i) & \text{for } t(A_i) = n-1. \end{cases}$$

It follows from lemma 4.1 that $g(A)$ is a member function $g(A) \in \mathbf{F}(A)$ of the function set. Therefore, there is $\mathbf{R}(A) \subseteq \mathbf{F}(A)$. Together with (4.28) there is $\mathbf{R}(A) = \mathbf{F}(A)$. □

Theorem 23 implies that the class of input independent function sets is equal to the class of characteristic function sets of MVL relations. Because function lattices are not input independent, they are difficult to store on computers. No efficient data structure for general function lattices is known. However, in real applications, function lattices never are computed explicitly. Instead properties from lattice theory are applied to obtain the desired results. The knowledge, that a function lattice has unique lower and upper bounds for instance, leads to efficient decomposition algorithms for monotone operators in Section 5.3.

4.3.5 Combinated ISFs

In Section 3.2.3 it was shown that Boolean ISFs are not sufficient to handle function sets that arise in EXOR-decomposition of ISFs. Sets of *combinated ISFs (CISFs)* have been defined to optimize solutions of EXOR-decomposition [37]. In this section, these results are extended to the MVL case. The function set MVL CISF is defined here, while its application in modsum-decomposition is discussed in Section 5.7.

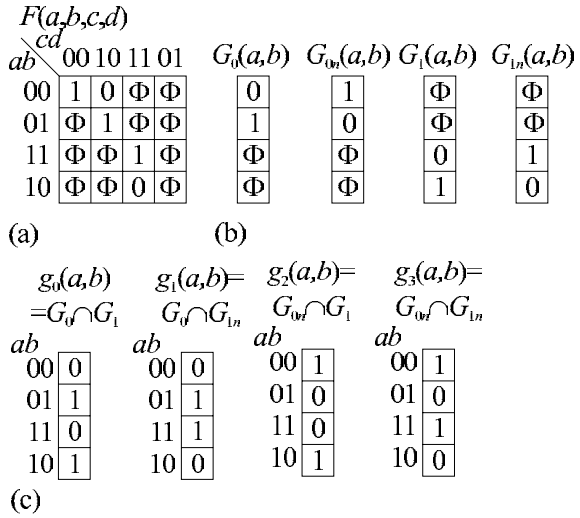

Figure 4.15: Decomposition functions of EXOR-decomposition. **(a)** ISF $F(a,b,c,d)$. **(b)** Components of the decomposition functions **(c)** Decomposition functions.

Example 4.17. The creation of a Boolean CISF is illustrated by the EXOR-decomposition of the ISF $F(a,b,c,d)$ from Example 3.8, see Figure 4.15(a).During the solution of the system of equations (3.29), two choices for the function $g(a,b)$ were made. The first choice $g(ab = 00) = 0$ results in the ISF $G_0(a,b)$ as shown in Figure 4.15(a). If the choice is made differently, $g(ab = 00) = 1$, the solution of (3.29) results in the ISF $G_{0n}(a,b)$. If compared to the ISF $G_0(a,b)$ it can be see that the cares of the ISF $G_{0n}(a,b)$ are negated. Similarly, the two possible choices for $g(ab = 11)$ result in the ISFs $G_1(a,b)$ and its negation $G_{1n}(a,b)$. The set of all decomposition functions can now be computed by the intersection of the characteristic function sets of the ISFs $G_0(a,b)$ and $G_{0n}(a,b)$ with the ISFs $G_1(a,b)$ and $G_{1n}(a,b)$. The result is shown in Figure 4.15(b). Note that all four decomposition functions can be derived from the two ISFs $G_0(a,b)$ and $G_1(a,b)$ by negation and intersection of the ISFs. The resulting function set is a Boolean CISF.

Boolean CISFs are the result of EXOR-decompositions of Boolean ISFs. In section 5.7 it will be shown that this result can be extended to modsum-decomposition of MVL ISFs, where *MVL CISFs* are applied. Decomposition functions that arise in the modsum-decomposition of a function have been shown in Section 4.1. Consider an MVL function $f(A,B)$, and assume the function is modsum-decomposable into $f(A,B) = g(A) \oplus_k h(B)$. Then, there is

$$f(A,B) = (g(A) \oplus_k c) \oplus_k (h(B) \ominus_k c), \tag{4.29}$$

for any integer $c = 0,\ldots,k-1$. Therefore, the functions $g_c(A) = g(A) \oplus_k c$ can also be used to decompose the function $f(A,B)$. The set of functions

$$\mathbf{G}(A) = \{g(A) \oplus i, \text{ for } i = 0,\ldots,k-1\} \tag{4.30}$$

describes the functions $g_c(A)$ that can be used for modsum-decomposition of $f(A,B)$. In Section 5.7 this result will be generalized to modsum-decomposition of ISFs. First, it is necessary to define the modsum-operator between ISFs and integer constants.

Definition 24. For an ISF $F(A)$ and an integer constant c, the modsum $G(A) = F(A) \oplus_k c$ is an ISF $G(A)$ defined by

$$G(A_i) := \begin{cases} \Phi & \text{for } F(A_i) = \Phi, \\ F(A_i) \oplus_k c & \text{otherwise.} \end{cases} \tag{4.31}$$

Now, it is possible to generalize the function sets described by (4.30) to ISFs.

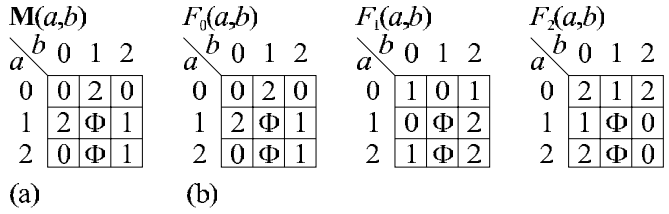

Figure 4.16: Characteristic function set of a modular ISF with module 3. **(a)** Modular ISF $\mathbf{M}(a,b) = F_0(a,b) \cup F_1(a,b) \cup F_2(a,b)$. **(b)** Sub-ISFs $F_0(a,b)$, $F_1(a,b) = F_0(a,b) \oplus_3 1$ and $F_2(a,b) = F_0(a,b) \oplus_3 2$.

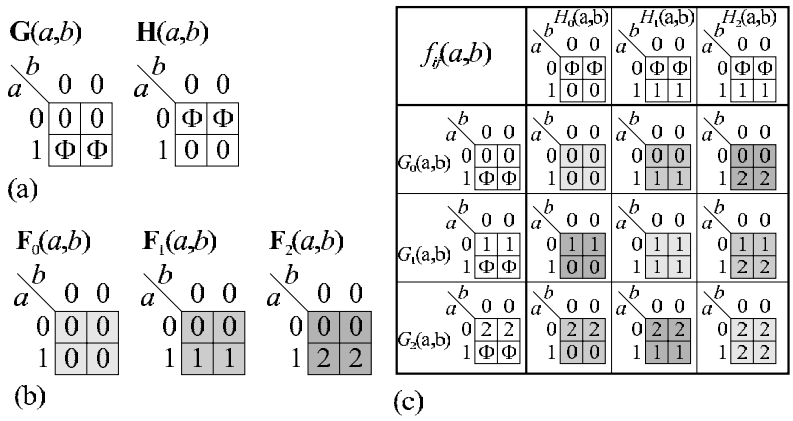

Figure 4.17: Intersection of modular ISFs. **(a)** Modular ISFs $\mathbf{G}(a,b)$ and $\mathbf{H}(a,b)$. **(b)** Modular ISFs of the intersection $\mathbf{G}(a,b) \cap \mathbf{H}(a,b)$. **(c)** Functions of the intersection $\mathbf{G}(a,b) \cap \mathbf{H}(a,b)$.

Definition 25. Let $F(A)$ be an ISF, and let $F_c(A) = F(A) \oplus c$ for $c = 0, 1, \ldots k-1$. The *modular ISF* $\mathbf{M}(A)$ is the set of functions

$$\mathbf{M}(A) := \mathbf{F}_0(A) \cup \mathbf{F}_1(A) \cup \cdots \cup \mathbf{F}_{k-1}(A). \tag{4.32}$$

The integer k is the *module* and the ISF $F(A)$ is a core of the modular ISF.

Example 4.18. A modular ISF $\mathbf{M}(a,b)$ with module $k = 3$ is shown in Figure 4.16(a), the corresponding sub-ISFs $F_i(a,b), i = 0,1,2$ are displayed in Figure 4.16(b). A core of $\mathbf{M}(a,b)$ is the ISF $F_0(X)$. A function $f(a,b)$ is element of the modular ISF $\mathbf{M}(a,b)$ if and only if it is a member function of one of the three ISFs $F_0(X)$, $F_1(X)$, or $F_3(X)$.

In general, modular ISFs are not function lattices.

Example 4.19. The modular ISF $\mathbf{M}(a,b)$ from Figure 4.16 is not a function lattice. For each minterm (a_i, b_j) there is a function $f_{ij}(a,b) \in \mathbf{M}(a,b)$ with $f_{ij}(a_i,b_j) = 0$ (see Figure 4.16). Therefore, the minimum over all member functions of the modular ISF is the constant function '0', which is not an element of the modular ISF $\mathbf{M}(a,b)$.

There are only very few modular ISFs that are function lattices. As Example 4.19 indicates the minimum over all functions of a modular ISF is the constant function '0', and hence, only modular ISFs that contain constant functions are function lattices. Unlike function lattices, modular ISFs are not closed over set intersection.

Example 4.20. The intersection of the modular ISFs $\mathbf{G}(a,b)$ and $\mathbf{H}(a,b)$ from Figure 4.17(a) consists of the nine functions $g_{ij}(a,b)$ for $i,j = 0,1,2$ shown in Figure 4.17(c).To illustrate the construction of the intersection, the component ISFs $G_i(a,b)$ and $H_j(a,b)$ of the modular ISFs $\mathbf{G}(a,b)$ and $\mathbf{H}(a,b)$ are also shown in the figure. No single modular ISF can describe this set of functions. The function set is union of three modular ISFs $\mathbf{F}_k(a,b), k = 0,1,2$ shown in Figure 4.17(b). The correspondence between the modular ISFs and the intersection functions is indicated by shaded boxes.

Table 4.1: Properties of MVL function sets.

Class	Values $m_o = 4$	Notation	Input independence	Closed over intersection
ISF	0 1 2 3	Φ	yes	yes
Interval	0 1 2 3	$[1,2]$	yes	yes
Relation	0 1 2 3	$0,1,3$	yes	yes
Lattice			no	yes
CISF			no	no

It will be shown in Section 5.7 that the functions arising from modsum-decomposition of ISFs can be described by intersections of modular ISFs. Therefore, the corresponding class of function sets is defined here.

Definition 26. Let $\mathbf{M}_1(A), \mathbf{M}_2(A), \ldots, \mathbf{M}_n(A)$ be modular ISFs with the same module k whose core ISFs have disjoint care sets. The intersection

$$\mathbf{F}(A) = \mathbf{M}_1(A) \cap \mathbf{M}_2(A) \cap \cdots \cap \mathbf{M}_n(A) \tag{4.33}$$

is called a *combinated ISF* or *CISF*.

Because the care sets of the modular ISFs $\mathbf{M}_i(A)$ are disjoint, the intersection in (4.33) is non-empty.

Because CISFs differ from all other classes of function sets shown here, new decomposition algorithms are needed which were discussed in [37] for Boolean CISFs. Heuristic decomposition algorithms for MVL CISFs are presented in Section 6.3.7.

4.4 Relations between Classes of Function Sets

With the properties shown in the previous sections, it is possible to relate various classes of function sets. First consider the function sets arising from the generalization of Boolean ISFs to the MVL domain. As shown in Section 2.2.4, the characteristic function set of a Boolean ISF can be described by

1. Boolean ISFs,
2. function intervals,
3. relations or
4. lattices.

The cases 1., 2. and 3. are equivalent descriptions of the same class of function sets. Lattices of Boolean functions generalize Boolean ISFs, but there always is an one-to-one mapping between Boolean lattices and ISFs. Different data structures were derived from the representations shown in the table. ISFs for instance are represented by an extension of BDDs with a third terminal node representing the don't care value Φ. A pair of two BDDs or TVLs is used to represent a Boolean ISF as an interval of functions, or an additional variable is introduced into the data structure to store an ISF as a relation. Even though there are significant differences between these data structures in terms of the algorithmic complexity, all of these data structures describe the characteristic function set of Boolean ISFs.

In the MVL case these definitions of Boolean ISFs can be generalized to different classes of function sets with different properties. Table 4.1 summarizes these properties.The second and third column show examples for the values and notation of the function set for a particular minterm. This notion is not defined for function lattices and CISFs because these function sets are not input independent. The table also indicates which classes of function sets are closed over set intersection.

A taxonomy of the classes of function sets introduced in this chapter is shown in Figure 4.18. The most general class of function sets is the class of all function sets. Theorems 16, 18, and 21

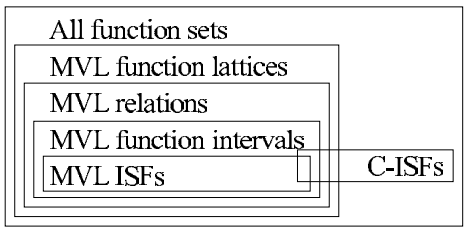

Figure 4.18: Taxonomy of classes of MVL function sets.

prove that there is a hierarchy among the classes ISF, function intervals, relations and lattices. The class of function sets that are function lattices and CISFs is limited to very few trivial cases, such as modular ISFs that contain constant functions.

There is a tradeoff between choosing a more general class of function sets, such as function lattices for instance, or choosing a specialized class of functions set such as ISFs for instance. On the one hand, more general classes of function sets allow greater flexibility in the optimization of functions because the search space is increased. On the other hand, more specialized class of function sets exhibit additional properties that allow the construction of more efficient optimization algorithms that run faster and require less memory.

Chapter 5

Properties of Bi-Decompositions

5.1 The MVL Bi-Decomposition Problem

Bi-decomposition of Boolean ISFs was discussed in Chapter 3. The notion of Boolean ISFs was generalized to MVL function sets in Chapter 4. In this chapter bi-decomposition is generalized so that it can be applied to decompose MVL function sets. For Boolean bi-decomposition there are only two interesting decomposition operators, OR and EXOR (see Section 3.1). For MVL functions there are may decomposition operators (see Section 2.3.2). Therefore, there are many different bi-decomposition algorithms. For the Boolean case, decomposition algorithms were only considered for Boolean ISFs. For the MVL case there is a variety of function sets. Therefore, the MVL bi-decomposition algorithms are classified not only by the decomposition operator, but also by the class of function set they decompose. Similar to Boolean bi-decomposition, there are MVL function sets that are not bi-decomposable. The concept of weak bi-decomposition (see Section 3.2.4) is extended so that MVL function sets can be simplified and the termination of recursive decomposition is ensured.

Basic terms are introduced in this section to generalize the bi-decomposition problem to bi-decomposition of MVL function sets. Properties that apply to all MVL decomposition operators are discussed in section 5.2. Sections 5.3–5.7 discuss the bi-decomposition problem for particular combinations of decomposition operators and function sets. Two extensions of weak bi-decomposition are presented in Section 5.8.

The definition of Boolean bi-decomposition for ISFs (3.2) can be generalized to bi-decomposition of MVL function sets.

Definition 27. A *bi-decomposition of an MVL function set* $\mathbf{F}(A,B,C)$ with respect to a *decomposition operator* $\pi(x,y)$, the *free set* A and the *bound set* B is a pair of functions $\langle g(A,C),h(B,C)\rangle$ with

$$\pi(g(A,C),h(B,C)) \in \mathbf{F}(A,B,C). \tag{5.1}$$

Neither the free nor the bound set may be empty. The set of variables C is called the *shared set*. The functions $g(A,C)$ and $h(B,C)$ are called the *free and bound decomposition function* respectively.

A function set $\mathbf{F}(A,B,C)$ is called *decomposable* if there exists a bi-decomposition. In the following, the term "bi-decomposition with respect to the operator $\pi(x,y)$" is abbreviated by the term π-decomposition. Although the free and bound set of a decomposition can be exchanged for symmetric operators, they must be distinguished if the operator $\pi(x,y)$ is not symmetric with respect to the variables x and y.

Example 5.1. Figure 5.1(a) gives an example of the bi-decomposition of the function $f(a,b)$ with respect to the variable partition $A=\{a\},B=\{b\},C=\emptyset$, and the operator $\pi(x,y)$ shown in Figure 5.1(b). Which functions $g(a)$ and $h(b)$ satisfy $f(a,b)=\pi(g(a),h(b))$? There is $f(1,0)=0$. The only minterm (x,y) with $\pi(x,y)=0$ is $x=y=0$. Therefore, there is $g(1)=h(0)=0$. Now there is $f(0,0)=1$. Because of $h(0)=0$ and $\pi(1,0)=\pi(2,0)=1$, the value of $g(0)$ can be either 1 or 2 resulting in the functions $g_1(a)$ and $g_2(a)$ as shown in the Figure. Similarly, because of $f(1,1)=1$, $g(1)=0$ and $\pi(0,1)=\pi(0,2)=1$, the value of $h(1)$ can be 1 or 2 resulting in the functions $h_1(b)$ and $h_2(b)$ as shown.

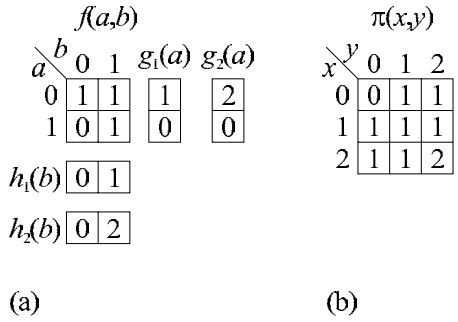

Figure 5.1: Decomposition functions for the bi-decomposition of $f(a,b)$ with respect to $\pi(x,y)$. (a) Function $f(a,b)$. (b) Operator $\pi(x,y)$.

It can easily be verified that $f(a,b) = \pi(g_1,h_1) = \pi(g_2,h_1) = \pi(g_1,h_2)$. Note however, that $f(a,b) \neq \pi(g_2,h_2)$ because $f(0,1) = 1 \neq \pi(g_2(0),h_2(1)) = \pi(2,2) = 2$.

It can be summarized that g_1, g_2, h_1 and h_2 are decomposition functions; and $\langle g_1,h_1\rangle$, $\langle g_1,h_2\rangle$, and $\langle g_2,h_1\rangle$ are p-decompositions of $f(a,b)$.

For Boolean ISFs it is possible to exchange the don't cares between the decomposition ISF $G(A,C)$ and $H(B,C)$ (see Section 3.3.2). For MVL function sets, the notion of don't cares is not always defined. To describe the relation between the decompositions functions, additional terms must be introduced. In general, there are many bi-decompositions $\langle g_i(A,C),h_j(B,C)\rangle$, as shown in Example 5.1. The set of all such pairs is described by a *decomposition relation.*

Definition 28. The *pi-decomposition relation* $D\,\langle g(A,C),h(B,C)\rangle$ of a function set $\mathbf{F}(A,B,C)$ with respect to the free set A, the bound set B is the set of all π-decompositions of $\mathbf{F}(A,B,C)$

$$D\,\langle g(A,C),h(B,C)\rangle := \{\langle g(A,C),h(B,C)\rangle \mid \pi(g(A,C),h(B,C)) \in \mathbf{F}(A,B,C)\}. \tag{5.2}$$

The *free decomposition set* is the set $\mathbf{G}(A,C)$ of all free decomposition functions $g(A,C)$

$$\mathbf{G}(A,C) := \{g(A,C) \mid \exists h(B,C) : \langle g(A,C),h(B,C)\rangle \in D\,\langle g(A,C),h(B,C)\rangle\}. \tag{5.3}$$

The *bound decomposition set* is the set $\mathbf{H}(B,C)$ of all bound decomposition functions $h(B,C)$

$$\mathbf{H}(B,C) := \{h(B,C) \mid \exists g(A,C) : \langle g(A,C),h(B,C)\rangle \in D\,\langle g(A,C),h(B,C)\rangle\}. \tag{5.4}$$

The *decomposition set with respect to the function* $g_0(A,C)$ is the set of decomposition functions $\mathbf{H}_{g_0}(B,C)$ that decompose $\mathbf{F}(A,B,C)$ with respect to $g_0(A,C)$

$$\mathbf{H}_{g_0}(B,C) = \{h(B,C) \mid \langle g_0(A,C),h(B,C)\rangle \in D\,\langle g,h\rangle\}. \tag{5.5}$$

Decomposition relations can be displayed by bipartite graphs with the functions $g(A,C)$ and $h(B,C)$ as nodes. An edge is drawn between the pairs of functions that are bi-decompositions.

Example 5.2. The decomposition relation of the decomposition from Example 5.1 is shown in Figure 5.2. The lattice $\mathbb{F}_3(a)$ of all functions that depend on a with output cardinality 3 is shown at the left side, the lattice $\mathbb{F}_3(b)$ of all functions that depend on b with output cardinality 3 is shown to the right. Solid edges are drawn between the functions of the three decomposition pairs $\langle g_1,h_1\rangle$, $\langle g_1,h_2\rangle$ and $\langle g_2,h_1\rangle$.

The function set $\mathbf{F}(a,b)$ has the free decomposition set $\mathbf{G}(a) = \{g_1(a),g_2(a)\}$ and the bound decomposition set $\mathbf{H}(b) = \{h_1(b),h_2(b)\}$. The decomposition set $\mathbf{H}_{g_2}(b)$ with respect to the function $g_2(a)$ is the set $\mathbf{H}_{g_2}(b) = \{h_1(b)\}$.

The properties developed in Sections 5.3–5.7 aim at the solution of three problems for the π-decomposition of the function set $\mathbf{F}(A,B,C)$ with resepect the free set A and the bound set B.

1. **Decomposition test.** Is the function set π-decomposable with respect to the free and bound set?

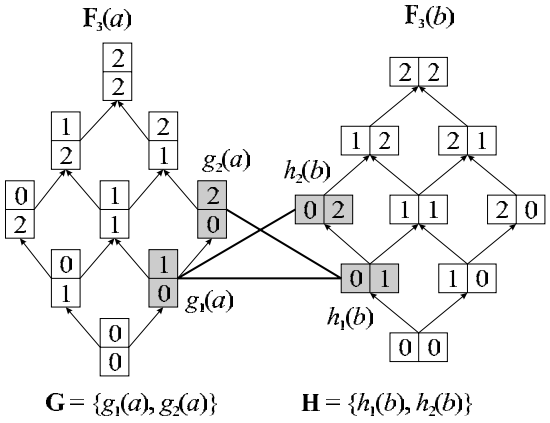

Figure 5.2: Decomposition relation for the bi-decomposition from Figure 5.1.

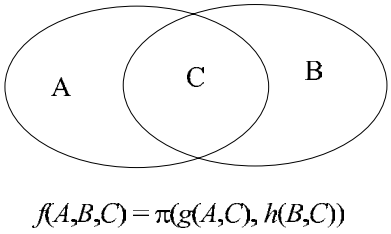

Figure 5.3: Variable sets of a bi-decomposition.

2. **Free decomposition set.** Wich functions does the free decomposition set $\mathbf{G}(A, C)$ contain?

3. **Decomposition set with respect to a function.** Which functions does the decomposition set $\mathbf{H}_{g0}(B, C)$ with respect to the given decomposition function $g_0(A, C)$ contain?

Bi-decomposition can be further generalized with respect to the variable sets involved [16]. Bi-decomposition as introduced in Section 3.1 considers the case where the decomposition functions g and h only depend on variables that are present in the original function (see Figure 5.3).

For some applications, it is useful to consider the case, where the decomposition functions depend on variables that are not present in the original function (see Figure 5.4). This decomposition scheme seems to contradict the fundamental principle of decomposition that aims to reduce the number of variables. Generalized decompositions might be useful in cases, where one of the decomposition functions is available without additional cost (because it has already been realized, or taken from a library), and the other function still has fewer variables than the original function.

Definition 29. Given a function set $F(A_1, B_1, C_1)$ a *generalized bi-decomposition* with respect to the operator $\pi(x, y)$ is a pair of functions $g(A_1, A_2, C_1, C_2)$ and $h(B_1, B_2, C_1, C_2)$ so that

$$\pi(g(A_1, A_2, C_1, C_2), h(B_1, B_2, C_1, C_2)) \in \mathbf{F}(A_1, B_1, C_1). \tag{5.6}$$

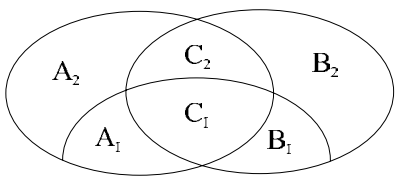

Figure 5.4: Variable sets of a generalized bi-decomposition.

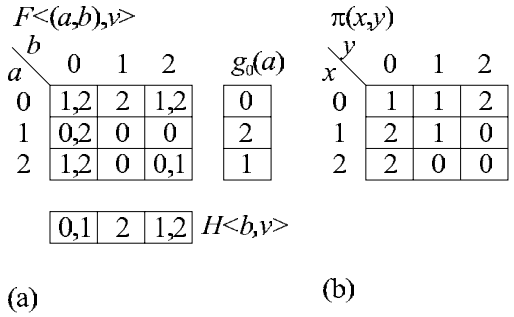

Figure 5.5: Decomposition of the relation $F\langle(a,b),v\rangle$ with respect to function $g_0(a)$ and the operator $\pi(x,y)$. **(a)** Relation $F\langle(a,b),v\rangle$, decomposition function $g_0(a)$ and decomposition set $H_{g_0}(b)$ with respect to $g_0(a)$. **(b)** Operator $\pi(x,y)$.

5.2 Bi-Decomposition of Function Sets

In general, the structure of the decomposition sets depends on the the decomposition operator and on the type of function set that is decomposed. Some properties hold for every operator function $\pi(x,y)$. There is a monotony between function sets and their decomposition sets.

Theorem 24. *Let* $\mathbf{F}(A,B,C)$ *be an arbitrary function set and let* $\mathbf{F}'(A,B,C)$ *be a subset*

$$\mathbf{F}'(A,B,C) \subseteq \mathbf{F}(A,B,C) \tag{5.7}$$

of $\mathbf{F}(A,B,C)$. *If function set* $\mathbf{F}'(A,B,C)$ *is* π*-decomposable with the free set* $\mathbf{G}'(A,C)$ *and the bound set* $\mathbf{H}'(B,C)$, *function set* $\mathbf{F}(A,B,C)$ *is also* π*-decomposable. Furthermore, for the decomposition sets* $\mathbf{G}(A,C)$ *and* $\mathbf{H}(B,C)$ *of* $\mathbf{F}(A,B,C)$ *there is*

$$\mathbf{G}'(A,C) \subseteq \mathbf{G}(A,C) \text{ and} \tag{5.8a}$$
$$\mathbf{H}'(B,C) \subseteq \mathbf{H}(B,C). \tag{5.8b}$$

Proof. The function set $\mathbf{F}'(A,B,C)$ is π-decomposable. Then, for every decomposition function $g'(A,C) \in \mathbf{G}'(A,C)$ of $\mathbf{F}'(A,B,C)$ there is a function $h'(B,C)$ so that

$$f'(A,B,C) = \pi(g'(A,C),h'(B,C)) \in \mathbf{F}'(A,B,C).$$

Because of (5.7) there is $f'(A,B,C) \in \mathbf{F}(A,B,C)$. Hence, $\mathbf{F}(A,B,C)$ is π-decomposable, and $g'(A,C)$ is a decomposition function of $\mathbf{F}(A,B,C)$. Therefore, there is $g'(A,C) \in \mathbf{G}(A,C)$, which implies (5.8a). A similar observation for the decomposition functions $h'(B,C) \in \mathbf{H}'(B,C)$ proves (5.8b). □

If a decomposition function $g_0(A,C)$ is known for an MVL relation, the decomposition set with respect to $g_0(A,C)$ also is a relation.

Example 5.3. The relation $F\langle(a,b),v\rangle$ from Figure 5.5(a) is π-decomposable with respect to operator $\pi(x,y)$ from Figure 5.5(b) and the function $g_0(a)$ from Figure 5.5(a). The decomposition set $H_{g_0}(b)$ with respect to $g_o(a)$ is a relation as shown in Figure 5.5(a).

Theorem 25. *Let* $R\langle(A,B,C),v\rangle$ *be a relation that is* π*-decomposable with respect to free set* A *and the bound set* B. *Let* $g(A,C)$ *be a decomposition function of* R, *and* $\mathbf{H}(B,C)$ *be the set of all functions* $h(B,C)$ *with*

$$\pi(g(A,C),h(B,C)) \in \mathbf{R}(A,B,C). \tag{5.9}$$

Then, the decomposition set $\mathbf{H}(B,C)$ *is a relation.*

Proof. It will be shown that the function set $\mathbf{H}(B,C)$ is input independent (see Definition 23). Then, Theorem 23 shows that the function set is a relation.

Let $h_1(B,C)$ and $h_2(B,C)$ be two arbitrary member functions of the set $\mathbf{H}(B,C)$, and let $t(B,C)$ be an arbitrary two-valued function. Now define

$$h_3(B,C) = h_1(B,C) * t(B,C) + h_2(B,C) * (1 - t(B,C)).$$

It will be shown that $h_3(B,C) \in \mathbf{H}(B,C)$. Two cases are distinguished by the values of $t(B_i,C_j)$.

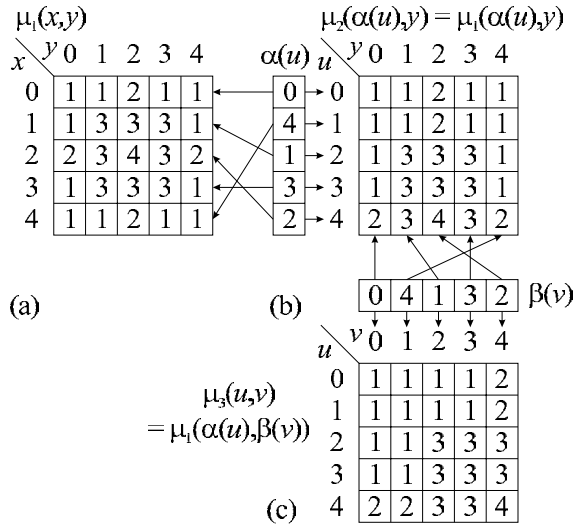

Figure 5.6: Permuting rows and columns of an operator. **(a)** Operator function $\mu_1(x,y)$ of the Gaussian bell function and permutation operator to sort the rows. **(b)** Operator function $\mu_2(u,y)$ with sorted rows and permutation operator to sort the columns. **(c)** Operator function $\mu_3(u,v)$ with sorted rows and columns.

1. If $t(B_i,C_j)=0$, there is $h_3(B,C)=h_2(B,C)$. Because $h_2(B,C)$ is a member of $\mathbf{H}(B,C)$ it satisfies (5.9). Therefore, there is

$$\langle (A,B_i,C_j),\pi(g(A,C_j),h_2(B_i,C_j))\rangle \in \mathbf{R}(A,B,C),$$

   which implies

$$\langle (A,B_i,C_j),\pi(g(A,C_j),h_3(B_i,C_j))\rangle \in \mathbf{R}(A,B,C).$$

2. Otherwise there is $t(B_i,C_j)=1$ and $h_3(B,C)=h_1(B,C)$. Substitution of $h_2(B,C)$ by $h_1(B,C)$ case 1 show that there is

$$\langle (A,B_i,C_j),\pi(g(A,C_j),h_3(B_i,C_j))\rangle \in \mathbf{R}(A,B,C).$$

With the cases 1 and 2 is shown that $h_3(B,C)$ also satisfies (5.9), and $h_3(B,C)\in\mathbf{H}(B,C)$. Therefore, the set $\mathbf{H}(B,C)$ is input independent and a relation. □

While Theorem 25 does not lead directly to a decomposition algorithm because one decomposition function must be known, it can be applied to compute the decomposition set with respect to a give function.

If a decomposition of a function set is known for an operator $\pi(x,y)$, other decompositions can be derived from this decomposition by permutation of the values of $\pi(x,y)$. The permutation of the values of operator $\pi(x,y)$ can be interpreted as the permutation of the rows and columns of the chart of the operator.

Example 5.4. Consider Gaussian bell function

$$\mu_1(x,y)=\text{round}(4*e^{-((x-2)^2-(y-2)^2)/5}) \tag{5.10}$$

is shown in Figure 5.6(a).A decomposition is first computed for the operator $\mu_3(u,v)$ by decomposition algorithms shown in Section 5.3. Then, the decomposition is transformed back into a decomposition for the original operator $\mu_1(x,y)$ by permutation of the values of the decomposition functions. The transformation of the operator $\mu_1(x,y)$ from Figure 5.6(a) is shown in Figure 5.6(b) and (c). The permutations $\alpha(x)$ and $\beta(y)$ in Figure 5.6(a) and (b) transform between the operators $\mu_1(x,y)$ and $\mu_3(x,y)$ shown in Figure 5.6(c).

Theorem 26. *If a function set* $\mathbf{F}(A,B,C)$ *is bi-decomposable with respect to the operator* $\pi(x,y)$*, then* $\mathbf{F}(A,B,C)$ *is* π'*-decomposable, where*

$$\pi'(u,v)=\pi(\alpha(u),\beta(v)). \tag{5.11}$$

and $\alpha(u)$ *and* $\beta(v)$ *are permutations of the values of the variables* x *and* y*.*

Proof. Because $\mathbf{F}(A,B,C)$ is π-decomposable, there are functions $g(A,C)$ and $h(B,C)$ with

$$\pi(g(A,C),h(B,C)) \in \mathbf{F}(A,B,C). \tag{5.12}$$

Denote the inverse permutation of $\alpha(x)$ and $\beta(y)$ with $\alpha^{-1}(u)$ and $\beta^{-1}(v)$ respectively. Let $g'(A,C) = \alpha^{-1}(g(A,C))$ and $h'(B,C) = \beta^{-1}(h(B,C)$, where $\alpha^{-1}(x)$ and $\beta^{-1}(y)$ are the inverse permutations of $\alpha(u)$ and $\beta(v)$ respectively. By (5.11) and the definition of $g'(A,C)$ and $h'(B,C)$ there is

$$\begin{aligned}\pi'(g'(A,C),h'(B,C)) &= \pi(\alpha(g'(A,C)),\beta(h'(B,C))) \\ &= \pi(\alpha\alpha^{-1}(g(A,C)),\beta\beta^{-1}(h(B,C))) \\ &= \pi(g(A,C),h(B,C))\end{aligned}$$

Substitution into (5.12) shows that $\mathbf{F}(A,B,C)$ is π'-decomposable with $g'(A,C)$ and $h'(B,C)$ as decomposition functions. □

5.3 Monotone Operators

In Definition 9, a partial order between minterms was defined. By means of this definition, a special class of functions can be defined.

Definition 30. A *monotone function* $f(X)$ is an MVL function with $f(X_i) \leq f(X_j)$ for all minterms X_i and X_j with $X_i \leq X_j$.

Examples of monotone functions are the minimum $\min(x,y)$, the maximum $\max(x,y)$, the truncated sum $\text{tsum}(x,y)$, or the truncated product $\text{tprod}(a,b)$.

Monotone functions have applications in machine learning. Many processes of the real world are monotone. A larger value of the input to a system often causes a larger value of the output. Therefore, many monotone functions, such as min, max, sum and product are easy to describe and understand in natural language. Machine learning systems that are described in terms of monotone functions will be easy to understand by humans. Therefore, bi-decomposition algorithms for monotone operator functions are of particular interest. This section demonstrates properties of function intervals that allow the construction of efficient algorithms for bi-decomposition of function intervals with respect to arbitrary monotone operators.

Definition 31. A *monotone bi-decomposition* is a bi-decomposition with respect to a monotone operator function $\mu(x,y)$.

Below, Lemma 5.1 shows an important algebraic property of monotone functions.

Lemma 5.1. *A monotone function $\mu(x,y)$ satisfies the distributive laws* (5.13)

$$\mu(a,\max(b,c)) = \max(\mu(a,b),\mu(a,c)), \tag{5.13a}$$
$$\mu(a,\min(b,c)) = \min(\mu(a,b),\mu(a,c)), \tag{5.13b}$$
$$\mu(\max(a,b),c) = \max(\mu(a,c),\mu(b,c)), \tag{5.13c}$$
$$\mu(\min(a,b),c) = \min(\mu(a,c),\mu(b,c)). \tag{5.13d}$$

Proof. The proof starts with (5.13a). Two cases are distinguished:

1. $b \leq c$: There is $\max(b,c) = c$, and hence

$$\mu(a,\max(b,c)) = \mu(a,c). \tag{5.14}$$

   Now there is $\mu(a,b) \leq \mu(a,c)$ because μ is monotone, and thus, $\mu(a,c) = \max(\mu(a,b),\mu(a,c))$. Substitution into (5.14) gives equation (5.13a).

2. $b > c$: There is $\max(b,c) = b$, and hence,

$$\mu(a,\max(b,c)) = \mu(a,b). \tag{5.15}$$

   Now there is $\mu(a,c) \leq \mu(a,b)$ because μ is monotone, and thus, $\mu(a,b) = \max(\mu(a,b),\mu(a,c))$. Substitution into (5.15) gives equation (5.13a).

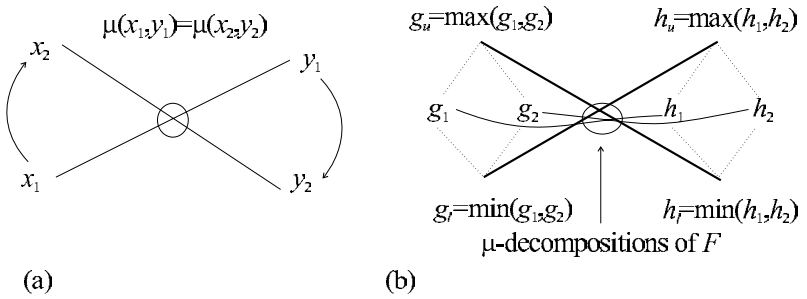

Figure 5.7: (a) Compensation of parameter changes for monotone functions. (b) Decomposition relation for monotone operators.

Equation (5.13b) can be proven by dual arguments. Equations (5.13c) and (5.13d) can be proven similarly after exchanging the roles of the first and second parameter of $\mu(x,y)$. □

Intuitively, the function value of monotone functions increases as their arguments increase. For a monotone operator $\mu(x,y)$ for example, an increase in the first parameter x can be compensated by a decrease in the second parameter y, see Figure 5.7(a).

Example 5.5. Consider the sum function $sum(x,y) = x+y$. Let $z_0 = sum(x_0,y_0)$. If variable x is increased to $x_1 > x_0$, the only way to get z_0 as the result of $z_0 = sum(x_1,y_1)$ is to decrease variable y to $y_1 < y_0$.

Below, Lemma 5.2 extends this principle to monotone bi-decomposition of functions.

Lemma 5.2. *Let* $\mathbf{F}(A,B,C) = [f_l, f_u]$ *be a function interval and* $\mu(x,y)$ *be a monotone operator. If the pairs of functions*

$$\langle g_1(A,C), h_1(B,C)\rangle \text{ and}$$
$$\langle g_2(A,C), h_2(B,C)\rangle$$

are μ*-decompositions of* $\mathbf{F}(A,B,C)$*, then the pairs of functions*

$$\langle g_l(A,C), h_u(B,C)\rangle \text{ and} \tag{5.16}$$
$$\langle g_u(A,C), h_l(B,C)\rangle \tag{5.17}$$

are also μ*-decompositions of* $\mathbf{F}(A,B,C)$*, where*

$$g_l(A,C) = \min(g_1(A,C), g_2(A,C)) \tag{5.18}$$
$$g_u(A,C) = \max(g_1(A,C), g_2(A,C)) \tag{5.19}$$
$$h_l(B,C) = \min(h_1(B,C), h_2(B,C)) \tag{5.20}$$
$$h_u(B,C) = \max(h_1(B,C), h_2(B,C)) \tag{5.21}$$

See Figure 5.7(b).

Proof. It is first proven that the pair from (5.16) is a μ-decomposition of F. Let be

$$f_1(A,B,C) = \mu(g_1(A,C), h_1(B,C)) \tag{5.22}$$
$$f_2(A,B,C) = \mu(g_2(A,C), h_2(B,C)) \tag{5.23}$$
$$f_3(A,B,C) = \mu(g_l(A,C), h_u(B,C)). \tag{5.24}$$

Substitution of (5.21) into (5.24) results in

$$f_3(A,B,C) = \mu(g_l(A,C), \max(h_1(B,C), h_2(B,C)))$$

By (5.13a) if follows that

$$f_3(A,B,C) = \max(\mu(g_l(A,C), h_1(B,C)), \mu(g_l(A,C), h_2(B,C))) \tag{5.25}$$

Because of (5.18) there is $g_l(A,C) \leq g_1(A,C)$ and $g_l(A,C) \leq g_2(A,C)$. Operator $\mu(x,y)$ is monotone. Hence, there is

$$\mu(g_l(A,C),h_1(B,C)) \leq \mu(g_1(A,C),h_1(B,C)) \text{ and} \tag{5.26}$$
$$\mu(g_l(A,C),h_2(B,C)) \leq \mu(g_2(A,C),h_2(B,C)). \tag{5.27}$$

The max-operator is also monotone. Therefore, (5.26) and (5.27) can be substituted into (5.25), which gives the inequality

$$f_3(A,B,C) \leq \max(\mu(g_1(A,C),h_1(B,C)),\mu(g_2(A,C),h_2(B,C)))$$

Substitution of (5.22) and (5.23) gives

$$f_3(A,B,C) \leq \max(f_1(A,B,C),f_2(A,B,C)). \tag{5.28}$$

The function interval $\mathbf{F}(A,B,C)$ is a function lattice (see Theorems 18 and 21). There is $f_1(A,B,C) \in \mathbf{F}(A,B,C)$ and $f_2(A,B,C) \in \mathbf{F}(A,B,C)$. Hence, there is

$$\max(f_1(A,B,C),f_2(A,B,C)) \in \mathbf{F}(A,B,C) \text{ and} \tag{5.29}$$
$$\max(f_1(A,B,C),f_2(A,B,C)) \leq f_u(A,B,C) \tag{5.30}$$

Substitution of (5.30) into (5.28) results in

$$f_3(A,B,C) \leq f_u(A,B,C). \tag{5.31}$$

With the dual arguments it can be shown that

$$f_l(A,B,C) \leq f_3(A,B,C). \tag{5.32}$$

From (5.31) and (5.32) follows that $f_3(A,B,C) \in \mathbf{F}(A,B,C)$. Therefore, the pair (5.16) is a μ-decomposition of $\mathbf{F}(A,B,C)$.

Similar arguments show that pair (5.17) also is a μ-decomposition of $\mathbf{F}(A,B,C)$. □

Example 5.6. Consider the μ-decomposition of the function interval $\mathbf{F}(a,b)$ shown in Figure 5.8(a) with respect to the monotone operator $\mu(x,y)$ shown in Figure 5.8(b). Two μ-decompositions of the function interval $F(a,b)$ are $\langle g_1(a),h_1(b)\rangle$ and $\langle g_2(a),h_2(b)\rangle$ as displayed in Figure 5.8(c) and (d). Therefore, the pairs

$$\langle \max(g_1(a),g_2(a)),\min(h_1(b),h_2(b))\rangle \text{ and}$$
$$\langle \min(g_1(a),g_2(a)),\max(h_1(b),h_2(b))\rangle$$

are also μ-decompositions of $F(a,b)$ as shown in Figure 5.8(e) and (f).

Below, Theorem 27 reveals the structure of the decompositions set of function intervals after a monotone bi-decomposition.

Theorem 27. *If a function interval* $\mathbf{F}(A,B,C)$ *is* μ*-decomposable with respect to the monotone operator* $\mu(x,y)$ *and the variable sets* A *and* B*, then the decomposition sets* $\mathbf{G}(A,C)$ *and* $\mathbf{H}(B,C)$ *are function lattices. Furthermore, the pairs*

$$\langle \inf \mathbf{G}(A,C), \sup \mathbf{H}(B,C)\rangle \text{ and} \tag{5.33a}$$
$$\langle \sup \mathbf{G}(A,C), \inf \mathbf{H}(B,C)\rangle \tag{5.33b}$$

are μ*-decompositions of* $\mathbf{F}(A,B,C)$.

Proof. Because $\mathbf{G}(A,C)$ is a decomposition set, for any two functions $g_1(A,C) \in \mathbf{G}(A,C)$ and $g_2(A,C) \in \mathbf{G}(A,C)$, there are functions $h_1(B,C)$ and $h_2(B,C)$ so that $\langle g_1(A,C),h_1(B,C)\rangle$ and $\langle g_2(A,C),h_2(B,C)\rangle$ are μ-decompositions of $\mathbf{F}(A,B,C)$. Then, by Lemma 5.2, $g_3(A,C) = \max(g_1(A,C),g_2(A,C))$ is a decomposition function of $\mathbf{F}(A,B,C)$, and $g_3(A,C) \in \mathbf{G}(A,C)$. Similarly, by Lemma 5.2, $\min(g_1(A,C),g_2(A,C)) \in \mathbf{G}(A,C)$, and decomposition set $\mathbf{G}(A,C)$ is a function lattice. With similar arguments it can be shown that the decomposition set $\mathbf{H}(B,C)$ is a function lattice.

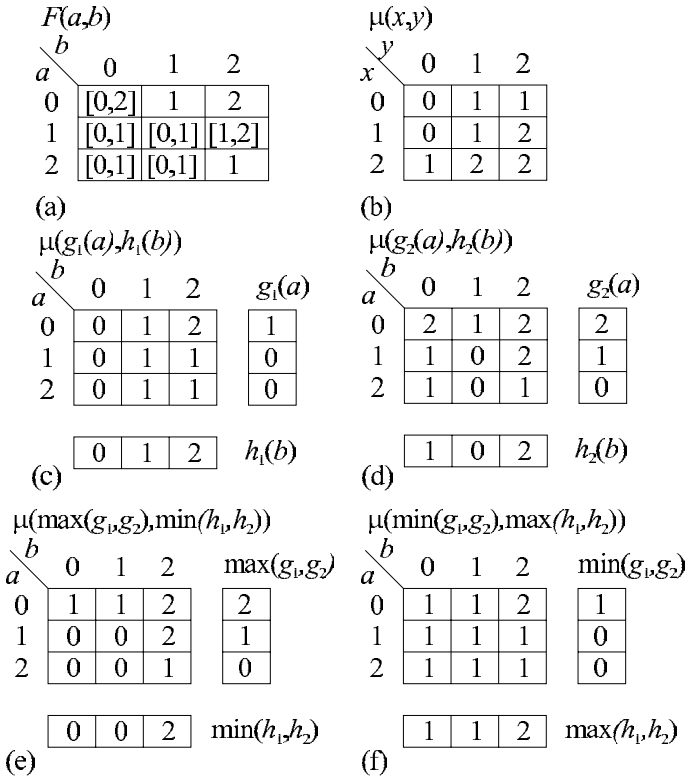

Figure 5.8: Monotone μ-decomposition of a function interval. **(a)** Function interval $\mathbf{F}(a,b)$. **(b)** Monotone operator $\mu(x,y)$. **(c),(d)** Two different μ-decompositions of $\mathbf{F}(a,b)$. **(e),(f)** New μ-decompositions constructed by Lemma 5.2.

It is now shown that $\langle \inf \mathbf{G}(A,C), \sup \mathbf{H}(B,C) \rangle$ is a μ-decomposition of $\mathbf{F}(A,B,C)$. Because $\inf \mathbf{G}(A,C)$ and $\sup \mathbf{H}(B,C)$ are elements of the decomposition sets $\mathbf{G}(A,C)$ and $\mathbf{H}(B,C)$ respectively, there are functions $h(A,C)$ and $g(B,C)$ so that the pairs $\langle g(A,C), \sup \mathbf{H}(B,C) \rangle$ and $\langle \inf \mathbf{G}(A,C), h(B,C) \rangle$ are μ-decompositions of $\mathbf{F}(A,B,C)$. By Lemma 5.2

$$\langle \min(g(A,C), \inf \mathbf{G}(A,C)), \max(h(B,C), \sup \mathbf{H}(B,C)) \rangle$$

also is a μ-decomposition of $\mathbf{F}(A,B,C)$. Because of

$$\begin{aligned} \min(g(A,C), \inf \mathbf{G}(A,C)) &= \inf \mathbf{G}(A,C) \text{ and} \\ \max(h(A,C), \sup \mathbf{H}(B,C)) &= \sup \mathbf{H}(B,C) \end{aligned}$$

the pair $\langle \inf \mathbf{G}(A,C), \sup \mathbf{H}(B,C) \rangle$ is a μ-decomposition of $\mathbf{F}(A,B,C)$. Dual arguments show that the pair $\langle \sup \mathbf{G}(A,C), \inf \mathbf{H}(B,C) \rangle$ is a μ-decomposition of $\mathbf{F}(A,B,C)$. □

The theorem has two conclusions.

1. Because the decomposition sets are function lattices, the decomposition sets have a greatest lower bound and a least upper bound. This property will be used when refining the structure of the decomposition sets for the min- and max-operators in Section 5.5.

2. It is possible to find a bi-decomposition by finding the bounds of the decomposition sets. A decomposition algorithm based on this idea is demonstrated in Section 6.3.3.

Monotone operators as defined here have their minimum value at $\mu(0,0)$ and their maximum value at $\mu(m_x - 1, m_y - 1)$. There are operators that have a maximum or minimum value at another point, while they still show a monotone transition between the maximum and minimum values. Some of such operators can be transformed into monotone operators by permutation of the rows and columns of chart of the operator. A decomposition is first computed for the monotone operator and is then transformed back into a decomposition for the original operator by Theorem 26.

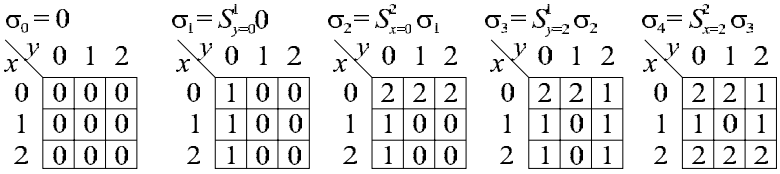

Figure 5.9: Creation of a simple operator.

Example 5.7. Consider the two-dimensional Gaussian bell function $\mu_1(x,y)$ from Example 5.4. There is a maximum at $\mu_1(2,2)$ with monotonically non-increasing values to lower and greater values for x and y respectively. The operator is transformed into the monotone operator $\mu_3(x,y)$ in Example 5.4. Therefore, a μ_1-decomposition can be computed by computing a monotone μ_3-decomposition.

5.4 Simple Operators

Simple operators are a special class of operator functions $\sigma(x,y)$ that allow the application of a decomposition algorithm that is based on the successive replacement of cares by don't cares. The replacement algorithm has been applied to AND- and OR-decomposition of Boolean ISFs [32]. In [39] the Boolean algorithm was extended to MVL ISFs and simple monotone operators, a subclass of monotone operators.

This section starts with a definition of simple operators that are not necessarily monotone. The replacement algorithm is then generalized to bi-decomposition of MVL relations.

Intuitively, a simple operator function is obtained by successively filling rows and columns of the chart of the function with constant values, where new values overwrite previous ones.

Example 5.8. The simple operator function $\sigma_4(x,y)$ shown in Figure 5.9 is obtained by the following sequence

1. Start with a constant function $\sigma_0 = 0$.
2. Fill the column $y = 0$ with 1.
3. Fill the row $x = 0$ with 2.
4. Fill the column $y = 2$ with 1.
5. Fill the row $x = 2$ with 2.

The replacement of rows and columns by constant values is called *σ-transformation.*

Definition 32. The *σ-transformation* $\mathcal{S}^c_{z=k}\,\sigma(X,z)$ of the MVL function $\sigma(X,z)$ is the MVL function

$$\mathcal{S}^c_{z=k}\,\sigma(X,z_i) := \begin{cases} c & \text{for } z_i = k \\ \sigma(X,z_i) & \text{otherwise,} \end{cases} \tag{5.34}$$

where where z is a single variable, and c and k are MVL constants respectively

The MVL constants k and c indicate the the position and the value of the replacement of function values respectively. Now, a simple operator function can be defined as a sequence of σ-transformations.

Definition 33. An MVL function $\sigma(x,y)$ is a *simple operator function* if it is a result of a sequence of σ-transformations applied to a constant σ_0

$$\sigma(x,y) = \mathcal{S}^{c_n}_{z_n=k_n} \ldots \mathcal{S}^{c_2}_{z_2=k_2}\,\mathcal{S}^{c_1}_{z_1=k_1}\,\sigma_0. \tag{5.35}$$

with $z_i \in \{x,y\}, i = 1 \ldots n$ that satisfies two conditions

1. There is at most one σ-transformation for each value of the operator variables, i.e. for $z_i = z_j$ there is $k_i \neq k_j$.

2. There is at least one value x_0 and one value y_0 with no σ-transformation, i.e. for all $z_i = x$ there is $k_i \neq x_0$ and for all $z_j = y$ there is $k_j \neq y_0$.

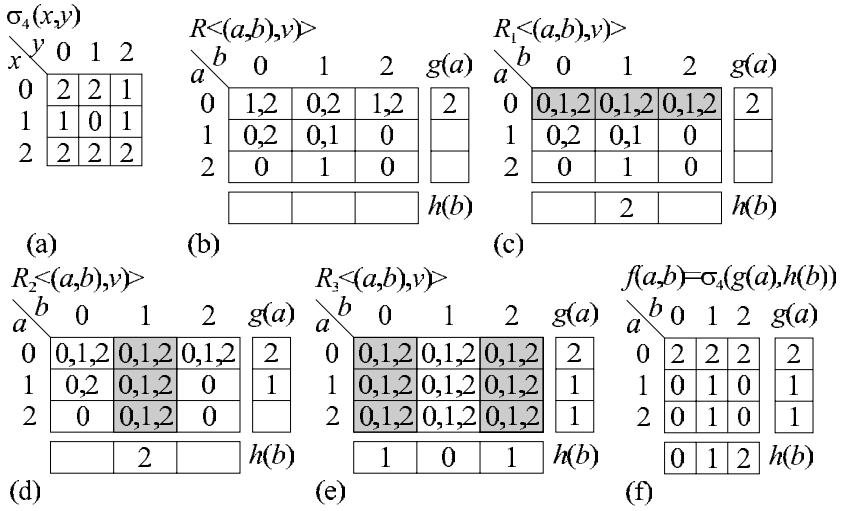

Figure 5.10: Decomposition of a relation with respect to the simple operator $\sigma_4(x,y)$. **(a)** Simple operator $\sigma_4(x,y)$. **(b)** Relation $R\langle(a,b),v\rangle$. **(c)** –**(e)** Decomposition of $R\langle(a,b),v\rangle$ by successive simplification. **(f)** Composition $\sigma_4(g(a),h(b))$.

A relation $R\langle(A,B,C),v\rangle$ can be σ-decomposed by successive computation of the values of decomposition functions $g(A,C)$ and $h(B,C)$.

Example 5.9. Consider the decomposition of the 3-valued relation $R\langle(a,b),v\rangle$ in Figure 5.10(b) with respect to the operator $\sigma_4(x,y)$, shown in Figure 5.10(a). There is $\langle(a=0,b),2\rangle \in R\langle(a,b),v\rangle$ and $\sigma(x=2,y)=2$ for all variables b and y. Therefore, there can be assigned $g(a=0)=2$ and the relation can be simplified by setting $R(a=0,b)$ to don't cares. The result is R_1, see Figure 5.10(c). The σ_4-decomposition problem of R was reduced to the σ_3-decomposition problem of R_1. There is $\langle(a,b=1),1\rangle \in R_1$ and $\sigma_3(x,y=2)=1$ for all variables a and x. Therefore, there can be assigned $h(b=1)=2$ and the relation can be simplified by setting $R_1(a,b=1)$ to don't cares. The result is R_2, see Figure 5.10(d), and the decomposition problem was again simplified to σ_2-decomposition of R_2. Further simplification is shown in Figure 5.10. The composition $f(a,b)=\sigma_4(g(a),h(b))$ shows that there is $f(a,b)\in\mathbf{R}(a,b)$ and the pair $\langle g(a),h(b)\rangle$ is a decomposition of $R\langle(a,b),v\rangle$, see Figure 5.10(f).

In the previous example, a relation was amended with don't cares if an entire row or column of the relation contained a certain constant value. To compute the don't cares, a new operator is defined.

Definition 34. The const-operator $T\langle A,w\rangle = \mathrm{const}^c_B(R\langle(A,B),v\rangle)$ is an operator that transforms a relation $R\langle(A,B),v\rangle$ into a relation $T\langle A,w\rangle$ by

$$T\langle A,w\rangle := \{\langle A_i,w_j\rangle \mid \langle(A_i,B),c\rangle \in R\langle(A,B),v\rangle\}. \tag{5.36}$$

Note that the relation $T\langle A,w\rangle$ is not a complete MVL relation. For minterms where the relation $R\langle(A_i,B),v\rangle$ does not contain the constant value c, the relation $T\langle A_i,w\rangle$ does not contain any value. If $R\langle(A_j,B),v\rangle$ contains the constant value c the relation $T\langle A_j,w\rangle$ contains a don't care. The decomposition of relations by successive computation of the decomposition functions can now be proven.

Theorem 28. *Let*

$$\sigma(x,y) = \mathcal{S}^c_{x=k}\,\sigma'(x,y) \tag{5.37}$$

be a simple operator function, and $R\langle(A,B,C),v\rangle$ be an MVL relation. Then, the relation $R\langle(A,B,C),v\rangle$ is σ-decomposable if and only if the relation

$$R'\langle(A,B,C),v\rangle = R\langle(A,B,C),v\rangle \cup R_A\langle(A,C),v\rangle \tag{5.38}$$

is σ'-decomposable, where

$$R_A\langle(A,C),v\rangle = \mathrm{const}^c_B(R\langle(A,B,C),v\rangle) \tag{5.39}$$

Proof. The proof is split into two parts. Part I) will show that $R'\langle(A,B,C),v\rangle$ is σ'-decomposable if $R\langle(A,B,C),v\rangle$ is σ-decomposable. Part II) will show that the converse is true.

I) Assume that the relation $R\langle(A,B,C),v\rangle$ is σ-decomposable then, there are functions $g(A,C)$ and $h(B,C)$ with $\sigma(g(A,C),h(B,C)) \in R\langle(A,B,C),v\rangle$. Two cases are now distinguished by the values of $g(A_i,C_j)$.

1. If $g(A_i,C_j) = k$, it follows from (5.37) that $\sigma(k,h(B,C)) = c$. Therefore, there is $\langle(A_i,B,C_j),c\rangle \in R$ for all minterms B and because of (5.39) the minterm (A_i,C_j) is a don't care of R_A. Because of (5.38), this minterm also is a don't care of R', and there is $\langle(A_i,C_j),\sigma'(g(A_i,C_j),h(B,C_j))\rangle \in R'$.

2. Otherwise there is $g(A_i,C_j) \neq k$. From (5.37) follows

$$\sigma(g(A_i,C_j),h(B,C_j)) = \sigma(g(A_i,C_j),h(B,C_j)), \tag{5.40}$$

and together with (5.38) there is $\langle(A_i,C_j),\sigma'(g(A_i,C_j),h(B,C_j)\rangle \in R \subseteq R'$.

From the cases 1 and 2 follows that the pair $\langle g(A,C),h(B,C)\rangle$ is a σ'-decomposition of R' hence, R' is σ'-decomposable.

II) Assume that the relation $R'\langle(A,B,C),v\rangle$ is σ'-decomposable then, there are functions $g'(A,C)$ and $h(B,C)$ with $\sigma'(g'(A,C),h(B,C)) \in R'\langle(A,B,C),v\rangle$. Define a function $g(A,C)$ by

$$g(A_i,C_j) = \begin{cases} k & \text{if } \langle(A_i,C_j),c\rangle \in R_A \\ x_0 & \text{if } \langle(A_i,C_j),c\rangle \notin R_A \text{ and } g'(A_i,C_j) = k \\ g'(A_i,C_j) & \text{otherwise,} \end{cases} \tag{5.41}$$

where x_0 is the missing value of x in the sequence of σ-transformations of the operator $\sigma(x,y)$, see Definition 33. Three cases are distinguished by the minterms (A_i,C_j).

1. If $\langle A_i,C_j,c\rangle \in R_A$, the minterm A_i,C_j is a don't care of R_A and because of (5.39), there is $\langle A_i,B,C_j,c\rangle \in R$. Because of (5.37) there is $\sigma(k,y) = c$ for all values of y. Together with $g(A_i,C_j) = k$ there is $\langle(A_i,B,C_j),\sigma(g(A_i,C_j),h(B,C_j)\rangle \in R$.

2. If $\langle A_i,C_j,c\rangle \notin R_A$ and $g'(A_i,C_j) = k$, the minterm A_i,C_j is not a don't care of R_A and $\langle(A_i,C_j),v\rangle \notin R_A$ for all values of v. Because of (5.38) there is

$$\langle A_i,B,C_j,\sigma'(g'(A_i,C_j),h(B,C_j)\rangle \in R.$$

By definition of simple operators no x-value of the σ-transformations occurs twice. Because the final σ-transformation of $\sigma(x,y)$ has $x = k$, there is no σ-transformation with $x = k$ in the sequence of σ-transformations of $\sigma'(x,y)$. Because there is no σ-transformation with $x = x_0$ (the missing value of x), there is $\sigma(x_0,y) = \sigma'(k,y)$. Because of $g(A_i,C_j) = k$ there is $\langle(A_i,B,C_j),\sigma(g(A_i,C_j),h(B,C_j)\rangle \in R$

3. Otherwise there is $\langle A_i,C_j,c\rangle \notin R_A$ and $g'(A_i,C_j) \neq k$. With the arguments from case 2 it follows that $\langle A_i,B,C_j,\sigma'(g'(A_i,C_j),h(B,C_j)\rangle \in R$. Because of (5.37) there is $\sigma(x,y) = \sigma(x,y)$ for $x \neq k$. Because of $g(A_i,C_j) = g'(A_i,C_j)$ there is

$$\langle(A_i,B,C_j),\sigma(g(A_i,C_j),h(B,C_j)\rangle \in R.$$

From the cases 1–3 follows that the pair $\langle g(A,C),h(B,C)\rangle$ is a σ-decomposition of R hence, R is σ-decomposable. □

5.5 Max-Decomposition

5.5.1 Overview

Section 5.3 demonstrated properties of bi-decomposition of function intervals with respect to monotone operators. The max-operator is a special cases of a monotone operator. Therefore, decomposition algorithms for monotone operators also apply to the max-operator. Max-decomposition of function intervals exhibits some additional properties that allow to develop faster

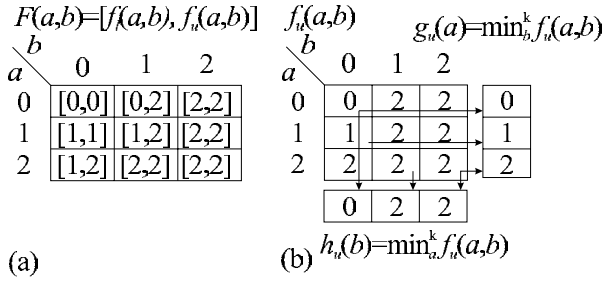

Figure 5.11: Max-decomposition of a function interval. **(a)** Function interval $F(a,b)$. **(b)** Upper bounds $g_u(a)$ and $h_u(b)$ of the max-decomposition functions of $F(a,b)$.

decomposition algorithms. It will also be shown that the decomposition sets have a simpler structure for the max-operator than for general monotone operators.

The properties developed in this section are mostly generalizations of the bi-decomposition theory for Boolean ISFs using the BDC as explained in Section 3.2. The results for the OR-decomposition of Boolean ISFs are generalized to the max-decomposition of MVL function intervals.

First, a criterion is developed that decides whether a function interval is max-decomposable using MVL derivation operators. Then, the structure of the decomposition sets is discussed and formulas for the computation of the decomposition sets are given. Some aspects of the decomposition relation for max-decomposition are also demonstrated. Section 5.6 extends the results of this section to the dual case of min-decompositions.

Definition 35. A function interval $F(A,B,C) = [f_l, f_u]$ is *max-decomposable* if$_{\text{DEF}}$ there exist functions $g(A,C)$ and $h(B,C)$ with

$$\max(g(A,C), h(B,C)) \geq f_l(A,B,C) \text{ and} \tag{5.42}$$

$$\max(g(A,C), h(B,C)) \leq f_u(A,B,C). \tag{5.43}$$

5.5.2 Bounds on the Decomposition Functions

An upper bound on the max-decomposition functions can be developed from the upper bound of the given function interval. A max-decomposition $\langle g(A,C), h(B,C)\rangle$ of the function interval $\mathbf{F}(A,B,C) = [f_l, f_u]$ must satisfy inequality (5.43). Because of $g(A,C) \leq \max(g(A,C), h(B,C))$, it can be concluded that $g(A,C) \leq f_u(A,B,C)$.

Example 5.10. Consider the max-decomposition of the function interval $F(a,b)$ shown in Figure 5.11(a) with respect to the free and bound set $A = \{a\}$ and $B = \{b\}$ respectively.To simplify the example, the shared set C is empty. The values of function $g(a_i)$ should not exceed the values of $f_u(a_i,b)$ as shown in Figure 5.11(b). Therefore, an upper limit on the function values of $g(a)$ is the minimum value of each row of the chart of $f_u(a,b)$.

Lemma 5.3. *Let the function interval $F(A,B,C) = [f_l, f_u]$ be max-decomposable with respect to the free and bound sets A and B respectively. If $\langle g(A,C), h(B,C)\rangle$ is a max-decomposition of $F(A,B,C)$, then there is*

$$g(A,C) \leq g_u(A,C) \leq f_u(A,B,C) \tag{5.44}$$

$$h(B,C) \leq h_u(B,C) \leq f_u(A,B,C), \tag{5.45}$$

where the functions $g_u(A,C)$ and $h_u(B,C)$ are defined by

$$g_u(A,C) = \min_B{}^{\mathrm{k}} f_u(A,B,C) \tag{5.46}$$

$$h_u(B,C) = \min_A{}^{\mathrm{k}} f_u(A,B,C). \tag{5.47}$$

Proof. Assume that $\langle g(A,C), h(B,C)\rangle$ is a max-decomposition of $F(A,B,C)$. Then, (5.43) is satisfied. Now, there is $g(A,C) \leq \max(g(A,C), h(B,C))$. Substitution into (5.43) gives $g(A,C) \leq f_u(A,B,C)$. Inequality (5.44) now results from Theorem 10. A similar proof holds for $h(B,C)$ and $h_u(B,C)$. □

Lemma 5.3 shows that there are upper limits on the decomposition functions. It remains the questions are these limits tight, i.e. are there decompositions that reach these limits? The lemma below gives a positive answer.

Lemma 5.4. *Let $F(A,B,C) = [f_l, f_u]$ be a max-decomposable function interval and the pair $\langle g(A,C), h(B,C)\rangle$ be a max-decomposition of $F(A,B,C)$, then the pairs of functions $\langle g_u(A,C), h(B,C)\rangle$ and $\langle g(A,C), h_u(B,C)\rangle$ are also max-decompositions of $F(A,B,C)$ where*

$$g_u(A,C) = \min_B{}^{\mathrm{k}} f_u(A,B,C) \tag{5.48}$$

$$h_u(B,C) = \min_A{}^{\mathrm{k}} f_u(A,B,C). \tag{5.49}$$

Proof. Assume that $\langle g(A,C), h(B,C)\rangle$ is a max-decomposition of $F(A,B,C)$, then inequality (5.42) is satisfied. From Lemma 5.3 can be concluded that $g_u(A,C) \geq g(A,C)$. Thus, there is $\max(g_u(A,C), h(B,C)) \geq \max(g(A,C), h(B,C))$. Substitution into (5.42) gives

$$\max(g_u(A,C), h(B,C)) \geq f_l(A,B,C) \tag{5.50}$$

Because of (5.44) and (5.45) there is $g_u(A,C) \leq f_u(A,B,C)$ and $h(B,C) \leq f_u(A,B,C)$. Thus, there is

$$\max(g_u(A,C), h(B,C)) \leq f_u(A,B,C).$$

Together with (5.50) it can be concluded that $\langle g_u(A,C), h(B,C)\rangle$ is a max-decomposition of $F(A,B,C)$. A similar proof holds for $\langle g(A,C), h_u(B,C)\rangle$. □

Lemma 5.4 leads to an important consequence. In general, there are many max-decomposition functions, say $g_1, g_2, \ldots, g_m$ and $h_1, h_2, \ldots, h_n$. However, not every pair of functions $\langle g_i, h_j\rangle$, $i = 1 \ldots m$, $j = 1 \ldots n$ is a max-decomposition. The lemma now shows that all pairs $\langle g_u, h_j\rangle$, $j = 1 \ldots n$ and $\langle g_i, h_u\rangle$, $i = 1 \ldots m$ are a max-decompositions.

The development of the lower bound on the decomposition functions is complicated by the fact that the lower bound for the functions $g(A,C)$ depends on the other decomposition function $h(B,C)$. A decomposition must satisfy inequality (5.42) for all minterms (A_i, B_j, C_k). If for a particular minterm (A_1, B_1, C_1) there is $h(B_1, C_1) \geq f_l(A_1, B_1, C_1)$, then inequality (5.42) is satisfied for (A_1, B_1, C_1) independently of the value of $g(A_1, C_1)$, and no restriction results from this minterm for $g(A_1, C_1)$. Otherwise, if $h(B_2, C_2) < f_l(A_2, B_2, C_2)$, there must be $g(A_2, C_2) \geq f_l(A_2, B_2, C_2)$ to satisfy (5.42). Because $g(A,C)$ must satisfy (5.42) for all minterms B, the maximum value over B must be taken.

Example 5.11. Consider decomposition of $F(a,b)$ from Figure 5.11(a) with respect to the function $h_u(b)$ shown below the map of $f_l(a,b)$ in Figure 5.12(a).For the minterm $a,b = 1,1$ there is $h_u(b=1) \geq f_l(a,b=1,1)$ and $\max(g(a=1), h(b=1)) \geq f_l(a,b=1,1)$ is satisfied independently of the value of $g(a=1)$. For the minterm $a,b = 2,0$ there is $h_u(b=0) < f_l(a,b=2,0)$. Therefore, the minimum value of $g(a=2)$ should be $f_l(a,b=2,0) = 1$. The restrictions on $g(a)$ for all minterms b_j are shown in the map of function $f_r(a,b)$ in Figure 5.12(b). The lower bound $g_l(a)$, after taking the maximum over the variable b, is shown to the right of the map.

The restrictions on the lower bound can be expressed with the help of the operator function $\mathrm{leq}_0(x,y)$ (see Section 2.3.2).

Lemma 5.5. *Let the function interval $F(A,B,C) = [f_l, f_u]$ be max-decomposable with respect to the free and bound sets A and B respectively. If $\langle g(A,C), h(B,C)\rangle$ is a max-decomposition of $F(A,B,C)$, then there is*

$$g(A,C) \geq g_l(A,C) \tag{5.51}$$

$$g_l(A,C) = \max_B{}^{\mathrm{k}} \mathrm{leq}_0(f_l(A,B,C), h(B,C)). \tag{5.52}$$

Proof. Let the pair $\langle g(A,C), h(B,C)\rangle$ be a max-decomposition of $F(A,B,C)$. Two cases are distinguished by the minterms (A_i, B_j, C_k).

1. If there is $f_l(A_i, B_j, C_k) \leq h(B_j, C_k)$, it can be concluded that $\mathrm{leq}_0(f_l(A_i, B_j, C_k), h(B_j, C_k)) = 0$, and it follows that $g(A_i, C_k) \geq \mathrm{leq}_0(f_l(A_i, B_j, C_k), h(B_j, C_k))$.

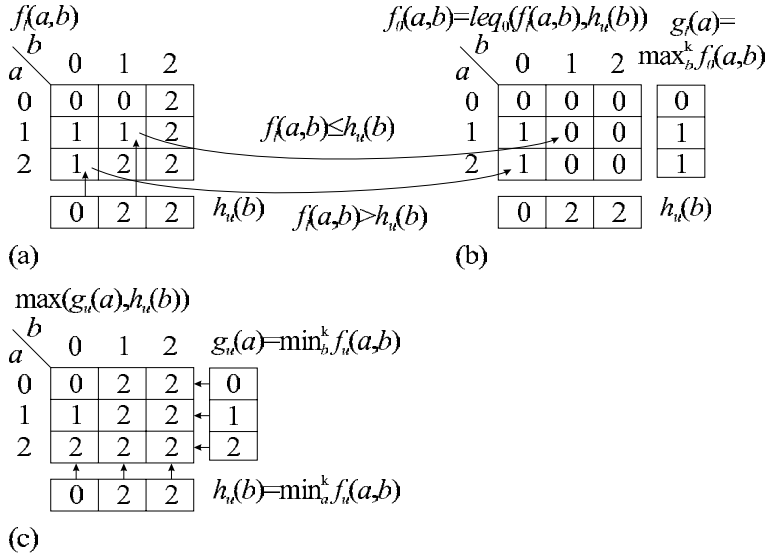

Figure 5.12: Computation of decomposition functions. **(a)** Lower bound $f_l(a,b)$ of the function interval $F(a,b)$. **(b)** Computation of the lower bound $g_l(a)$ of the decomposition set. **(c)** Test for max-decomposability by comparison of $\max(g_u(a),h_u(b))$ with $f_l(a,b)$.

2. Otherwise, there is $f_l(A_i,B_j,C_k) > h(B_j,C_k)$, and it results from (5.42) that $g(A_i,C_k) \geq f_l(A_i,B_j,C_k)$. Because of $\mathrm{leq}_0(f_l(A_i,B_j,C_k),h(B_j,C_k)) = f_l(A_i,B_j,C_k)$, there is $g(A_i,C_k) \geq \mathrm{leq}_0(f_l(A_i,B_j,C_k),h(B_j,C_k))$.

From the cases 1. and 2. can be inferred that there is $g(A,C) \geq \mathrm{leq}_0(f_l(A,B,C),h(B,C))$. Application of Theorem 11 gives $g(A,C) \geq \max^k{}_B \mathrm{leq}_0(f_l(A,B,C),h(B,C))$, which implies (5.51). □

Again, there remains the question if the lower bound is tight, which is positively answered by the following lemma.

Lemma 5.6. *Let $F(A,B,C) = [f_l,f_u]$ be a max-decomposable function interval and $\langle g(A,C),h(B,C)\rangle$ be a max-decomposition of $F(A,B,C)$, then the pair of functions $\langle g_l(A,C),h(B,C)\rangle$ is also a max-decomposition of $F(A,B,C)$, where*

$$g_l(A,C) = \max_B^k \mathrm{leq}_0(f_l(A,B,C),h(B,C)). \tag{5.53}$$

Proof. It results from Lemma 5.5 that there is $g_l(A,C) \leq g(A,C)$. Thus, it can be inferred that $\max(g_l(A,C),h(B,C)) \leq \max(g(A,C),h(B,C))$. Substitution into (5.43) gives

$$\max(g_l(A,C),h(B,C)) \leq f_u(A,B,C). \tag{5.54}$$

Now, two cases are distinguished by the minterms (A_i,B_j,C_k).

1. If there is $h(B_j,C_k) \geq f_l(A_i,B_j,C_k)$, then because of the inequality $\max(g_l(A_i,C_k),h(B_j,C_k)) \geq h(B_j,C_k)$, there is $\max(g_l(A_i,C_k),h(B_j,C_k)) \geq f_l(A_i,B_j,C_k)$.

2. Otherwise, there is $f_l(A_i,B_j,C_k) > h(B_j,C_k)$ and

$$\mathrm{leq}_0(f_l(A_i,B_j,C_k),h(B_j,C_k)) = f_l(A_i,B_j,C_k). \tag{5.55}$$

   From (5.53) and Theorem 11 can be inferred that there is $g_l(A,C) \geq \mathrm{leq}_0(f_l(A,B,C),h(B,C))$. Substitution into to (5.55) yields $g_l(A_i,C_k) \geq f_l(A_i,B_j,C_k)$. Because of $\max(g_l(A_i,C_k),h(B_j,C_k)) \geq g_l(A_i,C_k)$, it can be concluded that $\max(g_l(A_i,C_k),h(B_j,C_k)) \geq f_l(A_i,B_j,C_k)$.

The cases 1. and 2. show that $\max(g_l(A,C),h(B,C)) \geq f_l(A,B,C)$. Together with (5.54) is shown that $\langle g_l(A,C),h(B,C)\rangle$ is a max-decomposition of $F(A,B,C)$. □

5.5.3 Max-Decomposition Test

Now it is possible to derive a test condition for max-decomposability of function intervals from the upper bounds on the decomposition functions. Lemmas 5.3 and 5.4 showed tight upper bounds $g_u(A,C)$ and $h_u(B,C)$ on the decomposition functions. Therefore, a decomposition can exist only if these bounds satisfy (5.42). On the other hand if (5.42) is satisfied by the upper bounds, the functions $g_u(A,C)$ and $h_u(B,C)$ are a decomposition and the function interval is decomposable. The following theorem gives a formal argument based on this intuition.

Theorem 29. *A function interval $F(A,B,C)$ with the bounds $F(A,B,C) = [f_l, f_u]$ is max-decomposable with respect to the free set A and the bound set B iff*

$$f_l(A,B,C) \leq \max(g_u(A,C), h_u(B,C)) \tag{5.56}$$

$$g_u(A,C) = \min_B{}^{\mathrm{k}} f_u(A,B,C) \tag{5.57}$$

$$h_u(B,C) = \min_A{}^{\mathrm{k}} f_u(A,B,C). \tag{5.58}$$

Proof. Assume that (5.56) is satisfied. It results from (5.57), (5.58) and Theorem 10 that there is $g_u(A,C) \leq f_u(A,B,C)$ and $h_u(B,C) \leq f_u(A,B,C)$. Thus, there is

$$\max(g_u(A,C), h_u(B,C)) \leq f_u(A,B,C).$$

Together with (5.56) is shown that $\langle g_u(A,C), h_u(B,C)\rangle$ is a max-decomposition of $F(A,B,C)$. Hence, $F(A,B,C)$ is max-decomposable.

Now, assume that $F(A,B,C)$ is max-decomposable. Then, there exists a max-decomposition $\langle g(A,C), h(B,C)\rangle$. Application of Lemma 5.4 shows that $\langle g_u(A,C), h(B,C)\rangle$ also is a max-decomposition of $F(A,B,C)$. Then, a second application of the Lemma shows that the pair $\langle g_u(A,C), h_u(B,C)\rangle$ is a max-decomposition of $F(A,B,C)$, and (5.56) results from (5.42). □

Example 5.12. The functions $g_u(a)$ and $h_u(b)$ are displayed together with $\max(g_u(a), h_u(b))$ in Figure 5.12(c). Comparison with $f_l(a,b)$ in Figure 5.12(a) shows that (5.56) is satisfied. Additional comparison with $f_u(a,b)$ from Figure 5.11(a) reveals that $\langle g_u(a), h_u(b)\rangle$ is indeed a max-decomposition of $\mathbf{F}(a,b) = [f_l, f_u]$.

5.5.4 Computation of the Decomposition Functions

Once a decomposition is found, the max-decomposition sets must be computed. First, a formula for the decomposition set with respect to a give function is derived in Theorem 30. Then, the free decomposition set is computed in Theorem 31.

Theorem 30. *Let the function interval $\mathbf{F}(A,B,C) = [f_l, f_u]$ be max-decomposable with respect to the free set A and the bound set B, and let function $g_0(A,C)$ be a decomposition function of $\mathbf{F}(A,B,C)$. Then, the pair of functions $\langle g_0(A,C), h(B,C)\rangle$ is a max-decomposition of $\mathbf{F}(A,B,C)$ iff $h(B,C) \in [h_l, h_u]$, where*

$$h_u(B,C) = \min_A{}^{\mathrm{k}} f_u(A,B,C) \tag{5.59}$$

$$h_l(B,C) = \max_A{}^{\mathrm{k}} \mathrm{leq}_0(f_l(A,B,C), g_0(A,C)). \tag{5.60}$$

Proof. First, assume that $\langle g_0(A,C), h(B,C)\rangle$ is a max-decomposition of $\mathbf{F}(A,B,C)$. Then, it results from Lemma 5.5 that $h(B,C) \geq h_l(B,C)$ and from Lemma 5.3 that $h(B,C) \leq h_u(B,C)$. Thus, there is $h(B,C) \in [h_l, h_u]$.

Now, assume that $h(B,C) \in [h_l, h_u]$. Because $g_0(A,C)$ is a decomposition function of $\mathbf{F}(A,B,C)$, Lemma 5.6 shows that $\langle g_0(A,C), h_l(B,C)\rangle$ is a decomposition. It follows that

$$\max(g_0(A,C), h_l(B,C)) \geq f_l(A,B,C). \tag{5.61}$$

Now, there is $h(B,C) \geq h_l(B,C)$ and $\max(g_0(A,C), h(B,C)) \geq \max(g_0(A,C), h_l(B,C))$. Substitution into (5.61) results in

$$\max(g_0(A,C), h(B,C)) \geq f_l(A,B,C) \tag{5.62}$$

Lemma 5.4 shows that$\langle g_0(A,C), h_u(B,C)\rangle$ is a decomposition. It follows that

$$\max(g_0(A,C), h_u(B,C)) \leq f_u(A,B,C). \tag{5.63}$$

Now, there is $h(B,C) \leq h_u(B,C)$ and $\max(g_0(A,C), h(B,C)) \leq \max(g_0(A,C), h_u(B,C))$. Substitution into inequality (5.63) results in $\max(g_0(A,C), h(B,C)) \leq f_u(A,B,C)$ Together with (5.62) is shown that $\langle g_0(A,B), h(B,C)\rangle$ is a max-decomposition of $\mathbf{F}(A,B,C)$. □

Theorem 31. *Let the function interval* $\mathbf{F}(A,B,C) = [f_l, f_u]$ *be max-decomposable with respect to the free set* A *and the bound set* B*. Then, the function* $g(A,C)$ *is a decomposition function of* $\mathbf{F}(A,B,C)$ *iff* $g(A,C) \in [g_l, g_u]$*, where*

$$g_l(A,C) = \max_B{}^{\mathrm{k}}\, \mathrm{leq}_0(f_l(A,B,C), h_u(B,C)) \tag{5.64}$$

$$g_u(A,C) = \min_B{}^{\mathrm{k}} f_u(A,B,C) \tag{5.65}$$

$$h_u(B,C) = \min_A{}^{\mathrm{k}} f_u(A,B,C). \tag{5.66}$$

Proof. Because $\mathbf{F}(A,B,C)$ is max-decomposable, there exists a decomposition function $h(B,C)$. Now, Lemma 5.4 shows that the $h_u(B,C)$ also is a decomposition function.

First assume that $g(A,C)$ is a decomposition function. Theorem 30 shows that $g(A,C) \in [g_l, g_u]$.

Now assume that $g(A,C) \in [g_l, g_u]$. Then, Theorem 30 shows that the pair $\langle g(A,C), h_u(B,C)\rangle$ is a decomposition. Hence, $g(A,C)$ a decomposition function. □

5.6 Min-Decomposition

Min-decomposition is dual to max-decomposition. The results of Section 5.5 are extended and summarized for the case of min-decomposition.

Theorem 32. *A function interval* $\mathbf{F}(A,B,C)$ *with the bounds* $\mathbf{F}(A,B,C) = [f_l, f_u]$ *is min-decomposable with respect to the free set* A *and the bound set* B *iff*

$$f_u(A,B,C) \geq \min(g_l(A,C), h_l(B,C)) \tag{5.67}$$

$$g_l(A,C) = \max_B{}^{\mathrm{k}} f_l(A,B,C) \tag{5.68}$$

$$h_l(B,C) = \max_A{}^{\mathrm{k}} f_l(A,B,C). \tag{5.69}$$

Proof. The proof is dual to the proof of Theorem 29. □

Theorem 33. *Let the function interval* $\mathbf{F}(A,B,C) = [f_l, f_u]$ *be min-decomposable with respect to the free set* A *and the bound set* B*, and let function* $g_0(A,C)$ *be a decomposition function of* $\mathbf{F}(A,B,C)$*. Then, the pair of functions* $\langle g_0(A,C), h(B,C)\rangle$ *is a min-decomposition of* $\mathbf{F}(A,B,C)$ *iff* $h(B,C) \in [h_l, h_u]$*, where*

$$h_l(B,C) = \max_A{}^{\mathrm{k}} f_l(A,B,C) \tag{5.70}$$

$$h_u(B,C) = \min_A{}^{\mathrm{k}}\, \mathrm{geq}_{m_F}(f_u(A,B,C), g_0(A,C)). \tag{5.71}$$

where m_F *is the output cardinality of the function interval* $\mathbf{F}(A,B,C)$*.*

Proof. The proof is dual to the proof of Theorem 30. □

Theorem 34. *Let the function interval* $\mathbf{F}(A,B,C) = [f_l, f_u]$ *be min-decomposable with respect to the free set* A *and the bound set* B*. Then, function* $g(A,C)$ *is a decomposition function of* $\mathbf{F}(A,B,C)$ *iff* $g(A,C) \in [g_l, g_u]$*, where*

$$g_u(A,C) = \min_B{}^{\mathrm{k}}\, \mathrm{geq}_{m_F}(f_u(A,B,C), h_l(B,C)) \tag{5.72}$$

$$h_l(B,C) = \max_A{}^{\mathrm{k}} f_l(A,B,C) \tag{5.73}$$

$$g_l(A,C) = \max_B{}^{\mathrm{k}} f_l(A,B,C). \tag{5.74}$$

Proof. The proof is dual to the proof of Theorem 31. □

5.7 Modsum-Decomposition of ISFs

In this section, properties are shown that allow the construction of efficient algorithms for modsum-decomposition of ISFs. The algorithms are an extension of the EXOR-decomposition algorithms for Boolean ISFs [41, 37]. The decomposition theory below is formulated in terms of systems of linear equations [42].

For the min- and max-operators, the decomposition algorithms were developed for function intervals in Sections 5.5 and 5.6. However, the decomposition algorithms for the modsum-operator can only be extended to the more specialized ISFs. In Section 6.3.6 the properties developed for ISFs are applied in a heuristic modsum-decomposition algorithm for MVL relations. Heuristic means that the algorithm finds all decompositions if the input is an ISF, but some modsum-decomposable MVL relations may be found to be non-decomposable.

A modsum-decomposition of an ISF $F(A,B,C)$ is a pair of functions $\langle g(A,C),h(B,C)\rangle$ so that

$$g(A,C) \oplus_m h(B,C) \in \mathbf{F}(A,B,C), \tag{5.75}$$

where the constant m is the module of the modsum-operation. In this section, it is always assumed that the module is equal to the output cardinality m_F of the ISF $F(A,B,C)$.

First, the modsum-decomposition problem is transformed into the problem of solving a system of linear equations. Then, this system is solved by successively assigning values to the variables of the system and by computing values of dependent variables. It is shown that the decomposition set is a CISF as defined in Section 4.3.5. To transform the modsum-decomposition problem into a system of equations, a special type of equation systems is defined.

Definition 36. A *bilinear system of equations* is a system of equations that consists of two sets $X = \{x_1, x_2, \ldots, x_{n_x}\}$ and $Y = \{y_1, y_2, \ldots, y_{n_y}\}$ of m-valued variables. The system contains n_f equations of the form

$$x_{i_k} \oplus_m y_{j_k} = f_k \tag{5.76}$$

where $k = 1 \ldots n_f$, $0 \leq f_k < m$, $1 \leq i_k \leq n_x$ and $1 \leq j \leq n_y$.

The modsum-problem can be transformed into a bilinear system of equations by associating the values of the unknown decomposition functions $g(A,C)$ and $h(B,C)$ with the variables x_{i_k} and y_{i_k} respectively, and by associating the values of the cares of the ISF $F(A,B,C)$ with the constants f_k.

Lemma 5.7. *The functions $g(A,C)$ and $h(B,C)$ are a modsum-decomposition of the ISF $F(A,B,C)$ with respect to the free set A and the bound set B if and only if the bilinear system of equations*

$$g(A_i,C_k) \oplus_{m_F} h(B_j,C_k) = F(A_i,B_j,C_k) \tag{5.77}$$

is satisfied for all elements (A_i,B_j,C_k) of the care set of $F(A,B,C)$.

Proof. The functions $g(A,C)$ and $h(B,C)$ are a modsum-decomposition of $F(A,B,C)$ if and only if

$$g(A,C) \oplus_{m_F} h(B,C) \in \mathbf{F}(A,B,C). \tag{5.78}$$

By definition of the characteristic function set of an ISF, equation (5.78) is satisfied if and only if (5.77) is satisfied for all cares of $F(A,B,C)$. □

It is straightforward to setup the system of equations (5.77). The modsum-problem for ISFs is now reduced to two questions. What are the solutions of (5.77) and how can these solutions be computed?

Observe that a system of bilinear equations can be partitioned into subsystems with disjoint sets of variables. Different minterms C_i and C_j for instance, define such subsystems because the system (5.77) does not include equations that contain variables $g(A,C_i)$ or $h(B,C_i)$ together with variables $g(A,C_j)$ or $h(B,C_j)$. First, the solution of bilinear systems of equations is developed after partitioning into subsystems with disjoint sets of variables.

Definition 37. A *connected system of bilinear equations* is a system of bilinear equations whose equations cannot be partitioned into subsets so that each subset of equations contains a disjoint set of variables.

$F(a,b,c)$ a \ bc	00	10	20	01	11	21
0	0	2	Φ	Φ	Φ	Φ
1	2	Φ	Φ	0	1	Φ
2	Φ	Φ	1	1	2	0
3	Φ	Φ	0	Φ	Φ	0
4	Φ	Φ	Φ	Φ	Φ	2

Figure 5.13: Connected components of modsum-decomposition of the ISF $F(a,b,c)$ with the free set $A = \{a\}$ and the bound set $B = \{b\}$.

Table 5.1: Variables of the connected components of (5.79).

No.	Variables $g(a,c)$	Variables $h(b,c)$
1	$g(ac=00)$, $g(ac=10)$	$h(bc=00)$, $h(bc=10)$
2	$g(ac=20)$, $g(ac=30)$	$h(bc=20)$
3	$g(ac=11)$, $g(ac=21)$, $g(ac=31)$, $g(ac=41)$	$h(bc=01)$, $h(bc=11)$, $h(bc=21)$

Example 5.13. Consider the modsum-decomposition of the ISF $F(a,b,c)$, shown in Figure 5.13, with respect to the free set $A = \{a\}$ and the bound set $B = \{b\}$. The ISF has 12 cares, therefore, the corresponding system of bilinear equations consists of the 12 equations shown below

$$\begin{array}{rcll}
g(ac=00) \oplus_3 h(bc=00) & = & F(abc=000) & = 0 \\
g(ac=00) \oplus_3 h(bc=10) & = & F(abc=010) & = 2 \\
g(ac=10) \oplus_3 h(bc=00) & = & F(abc=100) & = 2 \\
\hline
g(ac=20) \oplus_3 h(bc=20) & = & F(abc=220) & = 1 \\
g(ac=30) \oplus_3 h(bc=20) & = & F(abc=320) & = 0 \\
\hline
g(ac=11) \oplus_3 h(bc=01) & = & F(abc=101) & = 0 \\
g(ac=11) \oplus_3 h(bc=11) & = & F(abc=111) & = 1 \\
g(ac=21) \oplus_3 h(bc=01) & = & F(abc=201) & = 1 \\
g(ac=21) \oplus_3 h(bc=11) & = & F(abc=211) & = 2 \\
g(ac=21) \oplus_3 h(bc=21) & = & F(abc=221) & = 0 \\
g(ac=31) \oplus_3 h(bc=21) & = & F(abc=321) & = 0 \\
g(ac=41) \oplus_3 h(bc=21) & = & F(abc=421) & = 2
\end{array} \tag{5.79}$$

The system (5.79) can be separated into three connected components, separated by horizontal lines. The corresponding cares of the ISF are indicated by shaded boxes in Figure 5.13. Table 5.1 shows the variables of each of the connected components

Connected systems of bilinear equations have a special structure. All pairs of variables are connected by a *chain of equations.*

Lemma 5.8. *Let S be a connected system of equations in the variables X and Y (of cardinality m). For any two variables $x_1, x_2 \in X$ there is a chain of equations*

$$\begin{array}{rcl}
x_{i0} \oplus_m y_{j0} & = & f_{k0} \\
x_{i1} \oplus_m y_{j0} & = & f_{k1} \\
x_{i1} \oplus_m y_{j1} & = & f_{k2} \\
 & \vdots & \\
x_{in} \oplus_m y_{j(n-1)} & = & f_{k(2n-1)}
\end{array} \tag{5.80}$$

with $x_1 = x_{i0}$ and $x_2 = x_{in}$.

Proof. Assume that there is no such chain between x_1 and x_2. Define the subset R of all equations that are contained in chains that starts at x_1. The system S can now be partitioned into the set of equations R and the remaining equations $S \setminus R$. Both sets of equations contain disjoint sets of variables. However, this contradicts the assumption that S be connected. Therefore, there is a chain between x_1 and x_2. □

If one variable of the solution, is known, other variables can be computed by subtraction. If x_i for instance is known and the equation $x_i \oplus_m y_j = f_k$ is element of the system, then y_j can be computed by

$$y_j = f_k \ominus_m x_i \tag{5.81}$$

For connected systems, all variables of the system can be computed by successive application of equations similar to (5.81).

Lemma 5.9. *Let S be a connected system of bilinear equations. Then for any two solutions (X_a, Y_a) and (X_b, Y_b) (with $X_a = \{x_{a1}, \ldots, x_{an_x}\}$, $Y_a = \{y_{a1}, \ldots, y_{an_y}\}$, $X_b = \{x_{b1}, \ldots, x_{bn_x}\}$ and $Y_b = \{y_{b1}, \ldots, y_{bn_y}\}$) there is*

$$x_{ai} \ominus x_{aj} = x_{bi} \ominus x_{bj} \tag{5.82}$$

for all $1 \leq i \leq n_x$ and $1 \leq j \leq n_y$.

Proof. Because S is connected, there is a chain from x_i to x_j of form (5.80). Alternating subtraction and addition of the equations in (5.80) results in

$$x_i \ominus x_j = f_{k0} \ominus f_{k1} \oplus f_{k2} \ominus \cdots \ominus f_{k(2n-1)}.$$

Therefore, the difference of two variables x_i and x_j is a constant for all solutions of S. □

Assume that one solution (X, Y) of a connected system of bilinear equations is known. Are there other solutions of this system? If one variable of the solution, x_1 for instance, is increased by a constant amount, say c, all variables $x_i, i = 1, \ldots, n_x$ are increased by c (see Lemma 5.9). This implies that all variables y_i of this solution must be decreased by c to satisfy (5.76). Therefore, variables of X are increased by c and all variables in Y are decreased by c. Below, Lemma 5.10 shows the structure of the solution of connected systems of bilinear equations. The notation $X \oplus_m l$ for the set $X = \{x_1, \ldots, x_{n_x}\}$ and the MVL constant l denotes the set

$$X \oplus_m l = \{x_1 \oplus_m l, \ldots, x_{n_x} \oplus_m l\}. \tag{5.83}$$

Lemma 5.10. *If the vectors (X_a, Y_a) ((with $X_a = \{x_{a1}, \ldots, x_{an_x}\}$ and $Y_a = \{y_{a1}, \ldots, y_{an_y}\}$) are a solution of a connected system S of bilinear equations, then the system has m different solutions of the form*

$$X_a \oplus l \quad \text{where } l = 0, \ldots, m-1. \tag{5.84}$$

Proof. It is first shown that (5.84) is a solution of S. Let $x_i \oplus y_j = f_k$ be an arbitrary equation of S. Because (X_a, Y_a) is a solution of S, there is

$$\begin{aligned} f_k &= x_{ai} \oplus y_{aj} \\ &= (x_{ai} \oplus l) \oplus (y_{aj} \ominus l) \end{aligned}$$

Therefore, $(X_a \oplus l, Y_a \ominus l)$ also is a solution of S for $l = 0, \ldots, m-1$.

Now assume that there is a solution $X_b = \{x_{b1}, \ldots, x_{bn_x}\}$ of S that is different from any solution in (5.84). Let $l = x_{a0} \ominus x_{b0}$. Then, $X_c = X_a \oplus l$ also is a solution of S with $X_b \neq X_c$. There is

$$\begin{aligned} x_{c0} &= x_{a0} \oplus l \\ &= x_{a0} \oplus x_{b0} \ominus x_{a0} \\ &= x_{b0} \end{aligned}$$

Because of $X_b \neq X_c$ there is some i with $x_{bi} \neq x_{ci}$. Because of $x_{b0} = x_{c0}$ there is

$$x_{bi} \ominus x_{b0} \neq x_{ci} \ominus x_{c0}$$

This inequality however contradicts the observation in Lemma 5.9. Therefore, the assumption that there is a solution X_b different from (5.84) must be wrong and the lemma is proven. □

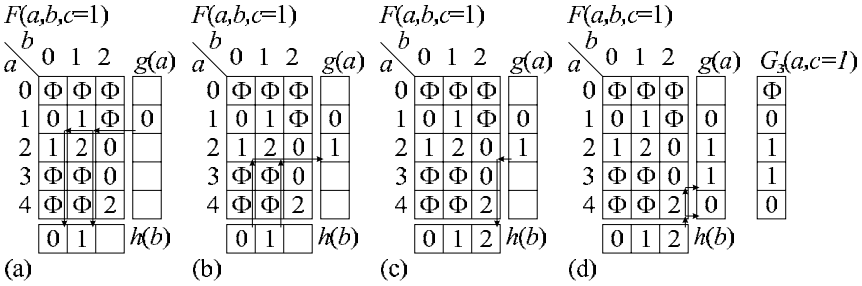

Figure 5.14: Connected component $F(a,b,c=1)$ and solution of the corresponding system of bilinear equations by a chain of equations.

Example 5.14. The solution of a connected system of bilinear equations is shown for the third connected component of $F(a,b,c=1)$ from Figure 5.13. The connected system of bilinear equations consists of the last seven equations from (5.79) (below the second line). An initial solution, say $g(a=1)=0$ is chosen arbitrarily, see Figure 5.14(a). The values $h(b=0)$ and $h(b=1)$ are computed by the equations

$$\begin{array}{rcrcrcl} h(0) & = & F(abc=101) \ominus g(a=1) & = & 0 \ominus 0 & = & 0 \\ h(1) & = & F(abc=111) \ominus g(a=1) & = & 1 \ominus 0 & = & 1 \end{array} \tag{5.85}$$

Now $g(a=2)$ is computed in a similar way from $h(b=0)$ and $h(b=1)$, see Figure 5.14(b). If the equations resulted in different values for $g(a=2)$, the ISF would not be decomposable. Both equations however, result in the value $g(a=2)=1$. All remaining values of $g(a)$ and $h(b)$ are computed in a similar way. The arrows in Figure 5.14 indicate the sequence of computations. Observe that the value $g(a=0)$ remains undetermined. No bilinear equation restricts this value therefore, the value is a don't care in the solution ISF $G_3(a,c=1)$ shown in Figure 5.14(d).

Now, it is possible to compose the solution of a general system of bilinear equations from the solutions of its connected components.

Lemma 5.11. *The set of solutions of a system of bilinear equations is the Cartesian product of the sets of solutions of its connected components.*

Proof. Because each connected component has a disjoint set of variables, every solution of one component can be combined with every solution of another component. □

To solve the EXOR-decomposition problem, the solution of the system of equations must now be translated back into a set of MVL functions. Theorem 35 shows that CISFs are an appropriate class of function sets to describe decomposition set of EXOR-decomposition.

Theorem 35. *The EXOR-decomposition set of an ISF is a CISF.*

Proof. The functions of the EXOR-decomposition set of ISFs are solutions of bilinear systems of equations (Lemma 5.7). The solution of a connected system of bilinear equations consists of vectors that differ only by an additive constant (Lemma 5.10). Therefore, these vectors can be described by a modular ISF (Definition 25) where the cares of the modular ISF correspond with the variables of the connected system of bilinear equations. The solution of a general system of bilinear equations is the Cartesian product of the solution of its connected components (Lemma 5.11). Therefore, the solution of a system of bilinear equations can be described by the intersection of modular ISFs. This set of MVL functions is a CISF (Definition 26) . □

Example 5.15. The EXOR-decomposition set of the ISF $F(a,b,c)$ in Figure 5.13 is the CISF

$$\mathbf{C}(a,c) = \mathbf{M}_1(a,c) \cap \mathbf{M}_2(a,c) \cap \mathbf{M}_3(a,c), \tag{5.86}$$

where $\mathbf{M}_1(a,c)$, $\mathbf{M}_2(a,c)$ and $\mathbf{M}_3(a,c)$ are modular ISFs whose cores, the ISFs $G_1(a,c)$, $G_2(a,c)$ and $G_3(a,c)$, are shown in Figure 5.15(a). An ISF $\mathbf{G}_{021}(a,c) \subset \mathbf{C}(a,c)$ is shown in Figure 5.15(b), where

$$G_{021}(a,c) = (G_1(a,c) \oplus 0) \cap (G_2(a,c) \oplus 2) \cap (G_3(a,c) \oplus 1). \tag{5.87}$$

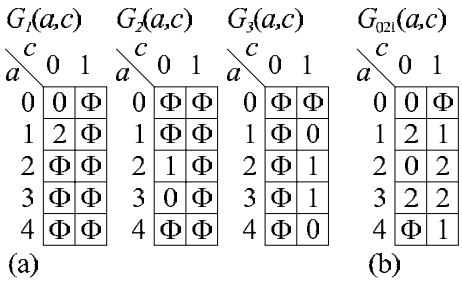

Figure 5.15: Solutions of the connected systems of bilinear equations of the ISF $F(a,b,c)$ in Figure 5.13. **(a)** Component ISFs $G_1(a,c)$, $G_2(a,c)$ and $G_3(a,c)$ of the solution. **(b)**, Member ISF $M(a,c)$ of the resulting CISF.

To compute the EXOR-decomposition on a computer, the corresponding system of bilinear equations is never setup explicitly. Instead, sets of minterms are maintained that store the minterms belonging to the connected components of the system of bilinear equations. One function value is chosen arbitrarily in each connected component and the remaining function values are computed iteratively. Then, the solution is stored in one modular ISF for each connected component. The intersection of these modular ISFs is only computed when the corresponding CISF is further decomposed.

No exact decomposition algorithms are known for CISFs. In Section 6.3.7 algorithms are shown that heuristically search for decomposable ISFs that are contained in CISFs. Because many ISFs are tested during the search, the chances of finding a decomposition by these algorithms is much higher than the decomposition of an arbitrary member ISF of the CISF.

The author believes that it might be possible to extend the ideas of this section to function intervals by application of results from integer programming. However, no efficient algorithm has been found yet.

5.8 Separation of Non-Decomposable Function Sets

5.8.1 Introduction

There are function sets that are not bi-decomposable with respect to a given set of operators. To simplify these function sets, an extension of bi-decomposition is necessary. Bi-decomposition decomposes a given function set $\mathbf{F}(A,B,C)$ into two decomposition sets $\mathbf{G}(A,C)$ and $\mathbf{H}(B,C)$, where each decomposition set depends on fewer variables. In order to simplify non-bi-decomposable function sets, these requirements must be relaxed. The extended bi-decomposition is called *separation*. In this section, two separation methods are introduced.

1. Section 5.8.2 introduces *multi-decomposition* , where an MVL function set is decomposed not only into two decomposition sets, but into n decomposition sets, where the number n is minimized.

2. Section 5.8.3 introduces *set separation*, where a function set is decomposed into a bi-decomposable function set and a superset of the of the original function set.

5.8.2 Multi-Decomposition

In this section it will be shown, that it is always possible to decompose a function set $\mathbf{F}(A,B,C)$ into a set of n bi-decomposable functions $f_1(A,B,C),\ldots,f_n(A,B,C)$. Below, there is a precise definition of this concept.

Definition 38. A *multi-decomposition* of a function set $\mathbf{F}(A,B,C)$ with respect to the sets of variables A and B and the operators $\pi(x_1,\ldots,x_n)$ and $\mu(y,z)$ is a vector of n functions $f_1(A,B,C),\ldots,f_n(A,B,C)$ so that

$$\pi(f_1(A,B,C),\ldots,f_n(A,B,C)) \in \mathbf{F}(A,B,C), \tag{5.88}$$

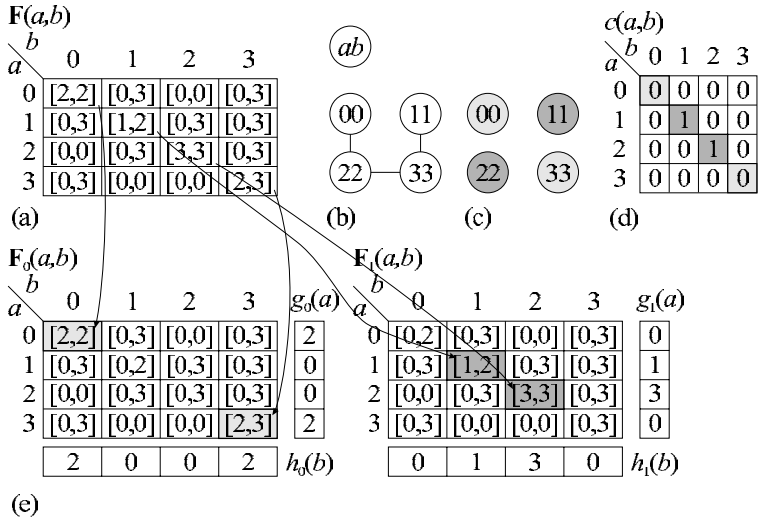

Figure 5.16: Multi-decomposition of function intervals. **(a)** Non-decomposable function interval $\mathbf{F}(a,b)$. **(b)** Incompatibility graph of $\mathbf{F}(a,b)$. **(c)** Coloring of the incompatibility graph. **(d)** Coloring function $c(a,b)$. **(e)** Multi-decomposition of $\mathbf{F}(a,b)$ into $\mathbf{F}_0(a,b)$ and $\mathbf{F}_1(a,b)$.

where the functions $f_1(A,B,C),\ldots,f_n(A,B,C)$ are μ-decomposable with respect to the sets of variables A and B.

To obtain decompositions of low complexity, the number n of decomposition functions should be as small as possible.

In this section, multi-decomposition of function intervals with respect to the max- and min-operators, *max-min-multi-decomposition* is considered. In extension of Section 5.6, another criterion for min-decomposability of function intervals is developed below. Then, it is shown how this criterion can be used to determine the minimum number of functions for max-min multi-decomposition. The min-decomposability of a function interval can be checked locally by the inspection of selected values of the interval bounds.

Example 5.16. Consider the function interval $\mathbf{F}(a,b) = f_l(a,b), f_u(a,b)]$ in Figure 5.16(a).To see that the function interval is not min-decomposable, it is sufficient to consider only three values of the interval bounds. For every min-decomposition function $g(a)$ there is $g(a) \geq f_l(ab = 00) = 2$, and for every min-decomposition function $h(b)$ there is $h(b = 2) \geq f_l(ab = 22) = 3$. It follows that

$$\min(g(a = 0), h(b = 2)) \geq 2. \tag{5.89}$$

This restriction contradicts the condition $f_u(ab = 02) = 0 < 2$. Therefore, the function interval $\mathbf{F}(a,b)$ is not min-decomposable.

Theorem 36. *A function interval $\mathbf{F}(A,B,C) = [f_l, f_u]$ is min-decomposable if and only if for all 5-tuple of minterms A_i, A_j, B_m, B_n, C_k there is*

$$\min(f_l(A_i,B_m,C_k), f_l(A_j,B_n,C_k)) \leq f_u(A_i,B_n,C_k). \tag{5.90}$$

Proof. First, assume that $\mathbf{F}(A,B,C)$ is min-decomposable. It can be derived from Theorem 32 that

$$\min(g_l(A_i,C_k), h_l(B_n,C_k)) \leq f_u(A_i,B_n,C_k) \tag{5.91}$$

$$g_l(A_i,C_k) = \max_B{}^{\mathrm{k}} f_l(A,B,C)|_{(A_i,C_k)} \tag{5.92}$$

$$h_l(B_n,C_k) = \max_A{}^{\mathrm{k}} f_l(A,B,C)|_{(B_n,C_k)}. \tag{5.93}$$

Because of Theorem 9 there is

$$f_l(A_i,B_m,C_k) \leq \max_B{}^{\mathrm{k}} f_l(A,B,C)|_{(A_i,C_k)}$$

$$f_l(A_j,B_n,C_k) \leq \max_A{}^{\mathrm{k}} f_l(A,B,C)|_{(B_n,C_k)}.$$

Substitution into (5.91) results in (5.90).

Now assume that (5.90) is satisfied for all 5-tuple of minterms A_i, A_j, B_m, B_n, C_k. Theorem 7 shows that for all minterms A_0, B_1, C_2 there are minterms B_0 and A_1 with

$$f_l(A_0, B_0, C_2) = \max_B{}^{\mathrm{k}} f_l(A, B, C)|_{(A_0, C_2)} \tag{5.94a}$$

$$f_l(A_1, B_1, C_2) = \max_A{}^{\mathrm{k}} f_l(A, B, C)|_{(B_1, C_2)}. \tag{5.94b}$$

Because (5.90) is satisfied for all 5-tuple of minterms, (5.94) can be substituted in (5.90)

$$\min(\max_A{}^{\mathrm{k}} f_l(A, B, C)|_{(B_1, C_2)}, \max_B{}^{\mathrm{k}} f_l(A, B, C)|_{(A_0, C_2)}) \leq f_u(A_0, B_1, C_2). \tag{5.95}$$

Because (5.95) is satisfied for all triple of minterms A_0, B_1, C_2 there is

$$\min(\max_A{}^{\mathrm{k}} f_l(A, B, C), \max_B{}^{\mathrm{k}} f_l(A, B, C)) \leq f_u(A, B, C). \tag{5.96}$$

It follows from Theorem 32 that $\mathbf{F}(A, B, C)$ is min-decomposable. □

Properties of max-min-multi-decompositions can be shown with the help of *incompatibility graphs*, which is a graph with one node for each minterm and an edge between minterms for every pair of minterms that does not satisfy (5.90).

Definition 39. The *min-incompatibility graph* of a function interval $\mathbf{F}(A, B, C) = [f_l, f_u]$ with respect to the sets of variables A and B is a graph with one node for each minterm (A_i, B_m, C_k). There is an edge between all pairs of minterms (A_i, B_m, C_k) and (A_j, B_n, C_k) with

$$\min(f_l(A_i, B_m, C_k), f_l(A_j, B_n, C_k)) > f_u(A_i, B_n, C_k). \tag{5.97}$$

Example 5.17. The incompatibility graph of the function interval $\mathbf{F}(a, b)$ from Figure 5.16(a) is shown in Figure 5.16(b). The nodes are labeled the with the value ab of the minterms. To simplify the graph, isolated nodes, that do not have any edges are omitted. If there is $f_l(a, b) = 0$ the inequality (5.97) is never satisfied and the node is isolated. Therefore, only four nodes of the incompatibility graph of $\mathbf{F}(a, b)$ are shown. For example, there is an edge between the nodes $ab = 00$ and $ab = 22$ because there is

$$\begin{aligned} \min(f_l(ab = 00), f_l(ab = 22)) &> f_u(ab = 02) \\ \min(2, 3) &> 0. \end{aligned} \tag{5.98}$$

Lemma 5.12. *Let $\mathbf{F}(A, B, C) = [f_l, f_u]$ be a function interval. Then, $\mathbf{F}(A, B, C)$ is min-decomposable if and only if its incompatibility graph does not contain any edges.*

Proof. By Theorem 36, $\mathbf{F}(A, B, C)$ is min-decomposable if and only if (5.90) is satisfied for all 5-tuple of minterms, which equivalent to (5.97) being false. Therefore, the incompatibility graph of $\mathbf{F}(A, B, C)$ does not contain any edges. □

Lemma 5.12 shows that max-min-multi-decomposition problem is equivalent to partitioning the incompatibility graph of a function interval into subsets so that there are no edges between the nodes of the same subset. This is the graph coloring problem, which assigns a color to each node so that no pair of nodes with the same color is connected by an edge. In the following a coloring of a graph is described by a *coloring function*, which is an MVL function where each minterm is assigned to a node of the graph and each function value is assigned to a color of the nodes.

Lemmas 5.13 and 5.14 relate colorings of incompatibility graphs to max-min-multi-decompositions. The results of these Lemmas are summarized in Theorem 37.

Lemma 5.13. *Let $c(A, B, C)$ be a coloring function with n colors of the incompatibility graph of the function interval $\mathbf{F}(A, B, C) = [f_l, f_u]$. Then, $\mathbf{F}(A, B, C)$ is max-min multi-decomposable into n functions.*

Proof. Define n function intervals $\mathbf{F}_i(A, B, C) = [f_{li}, f_{ui}]$, $i = 1, \ldots, n$ by

$$f_{li}(A_r, B_s, C_t) = \begin{cases} f_l(A_r, B_s, C_t) & \text{for } c(A_r, B_s, C_t) = i \\ 0 & \text{otherwise} \end{cases} \tag{5.99}$$

$$f_{ui}(A, B, C) = f_u(A, B, C). \tag{5.100}$$

First, it is shown that the function intervals $\mathbf{F}_1(A,B,C),\ldots,\mathbf{F}_n(A,B,C)$ are min-decomposable. Consider the function interval $\mathbf{F}_i(A,B,C)$. It will be shown that (5.90) is satisfied for all 5-tuple of minterms A_r, A_s, B_u, B_v, C_t. Two cases are distinguished by the values of the coloring function $c(A,B,C)$.

1. For $c(A_r,B_u,C_t)\neq i$ or $c(A_s,B_v,C_t)\neq i$ there is $f_{li}(A_r,B_v,C_t)=0$ or $f_{li}(A_s,B_v,C_t)=0$ and
$$\min(f_{li}(A_r,B_u,C_t),f_{li}(A_s,B_v,C_t))=0\leq f_{ui}(A_r,B_v,C_t).$$

2. Otherwise, there is $c(A_r,B_u,C_t)=c(A_s,B_v,C_t)=i$. Because nodes (A_r,B_u,C_t) and (A_s,B_v,C_t) are colored with the same color, there is no edge between (A_r,B_u,C_t) and (A_s,B_v,C_t) in the incompatibility graph of $\mathbf{F}(A,B,C)$ which implies that
$$\min(f_l(A_r,B_u,C_t),f_l(A_s,B_v,C_t))\leq f_u(A_r,B_v,C_t)$$
It follows from (5.99) and (5.100) that there is
$$\min(f_{li}(A_r,B_u,C_t),f_{li}(A_s,B_v,C_t))\leq f_{ui}(A_r,B_v,C_t).$$

It can be concluded that (5.90) is satisfied for $\mathbf{F}_i(A,B,C)$ for all 5-tuple of minterms. Hence, $\mathbf{F}_i(A,B,C)$ is min-decomposable. Therefore, there are min-decomposable functions $f_i(A,B,C)$, $i=1,\ldots,n$ with $f_i(A,B,C)\in\mathbf{F}_i(A,B,C)$.

Let $f(A,B,C)=\max(f_1(A,B,C),\ldots,f_n(A,B,C))$. It will now be shown that $f(A,B,C)\in\mathbf{F}(A,B,C)$. Because of $f_i(A,B,C)\leq f_{ui}(A,B,C)=p(A,B,C)$ there is $f(A,B,C)\leq f_u(A,B,C)$.

Let A_r,B_s,C_t be arbitrary minterms. Assume without loss of generality that there is $c(A_r,B_s,C_t)=i$. Then, there is $f_{li}(A_r,B_s,C_t)=f_l(A_r,B_s,C_t)$ and $f_l(A_r,B_s,C_t)\leq f_i(A_r,B_s,C_t)$. Hence, there is $f_l(A,B,C)\leq f(A,B,C)$. Therefore, the functions $f_1(A,B,C),\ldots,f_n(A,B,C)$ are a max-min multi-decomposition of $\mathbf{F}(A,B,C)$. □

Lemma 5.14. *If the function interval $\mathbf{F}(A,B,C)=[f_l,f_u]$ is max-min-multi-decomposable into n function, then its incompatibility graph is colorable with n colors.*

Proof. Let the functions $f_1(A,B,C),\ldots,f_n(A,B,C)$ be a max-min-multi-decomposition of $F(A,B,C)$, and let $f(A,B,C)=\max(f_1(A,B,C),\ldots,f_n(A,B,C))$. Let $c(A,B,C)$ be an n-valued function so that $i=c(A_r,B_s,C_t)$ denotes the smallest integer with $f_i(A_r,B_s,C_t)=f(A_r,B_s,C_t)$. It will be shown that $c(A,B,C)$ is a coloring of the incompatibility graph of $\mathbf{F}(A,B,C)$.

Consider two arbitrary minterms (A_r,B_u,C_t) and (A_s,B_v,C_t) with $c(A_r,B_u,C_t)=c(A_s,B_v,C_t)=i$. Because function $f_i(A,B,C)$ is min-decomposable, by Theorem 36, there is
$$\min(f_i(A_r,B_u,C_t),f_i(A_s,B_v,C_t))\leq f_i(A_r,B_v,C_t). \tag{5.101}$$
By the definition of $c(A,B,C)$ there is $f_i(A_r,B_u,C_t)=f(A_r,B_u,C_t)$ and because of $f(A,B,C)\in\mathbf{F}(A,B,C)$ there is $f_l(A_r,B_u,C_t)\leq f_i(A_r,B_u,C_t)$. Similarly, there is $f_l(A_s,B_v,C_t)\leq f_i(A_s,B_v,C_t)$. Because of $f_i(A,B,C)\leq f(A,B,C)$ there is $f_i(A_r,B_v,C_t)\leq f_u(A_r,B_v,C_t)$. Substitution of these inequalities into (5.101) gives
$$\min(f_l(A_r,B_u,C_t),f_l(A_s,B_v,C_t))\leq f_u(A_r,B_v,C_t).$$
Therefore, inequality (5.97) is never satisfied for pairs of minterms with the same value of the function $c(A,B,C)$, and there is no edge between the corresponding nodes of the incompatibility graph of $\mathbf{F}(A,B,C)$. Hence, function $c(A,B,C)$ is a coloring of the incompatibility graph of $\mathbf{F}(A,B,C)$ with n colors. □

Theorem 37. *A function interval $\mathbf{F}(A,B,C)=[f_l,f_u]$ is max-min multi-decomposable into n functions if and only if its incompatibility graph is colorable with n colors.*

Proof. The theorem is a direct consequence of Lemmas 5.13 and 5.14. □

Example 5.18. A coloring of the incompatibility graph of the ISF $\mathbf{F}(a,b)$ from Figure 5.16(a) is drawn in Figure 5.16(c). Two colors are needed to color the graph. The minterms that are not shown in the graph are isolated and do not have any edges. Therefore, they can be colored by any color. Color 0 is assigend to the lightly shaded and the isolated nodes. Color 1 is assigned

to the darker shaded nodes. The coloring function of this coloring is shown in Figure 5.16(d). The max-min-multi-decomposition of $\mathbf{F}(a,b)$ into $\mathbf{F}_0(a,b)$ and $\mathbf{F}_1(a,b)$ that corresponds to this coloring function is shown in Figure 5.16(e). The lower bound of $\mathbf{F}_0(a,b)$ is set to '0' for all minterms (a_i,b_j) with the coloring function $c(a_i,b_j) \neq 0$. A min-decomposition of $\mathbf{F}_0(a,b)$ is shown to the right and at the bottom of the chart of $\mathbf{F}_0(a,b)$. Similarly, the lower bound in $\mathbf{F}_1(a,b)$ is set to '0' for all minterms (a_i,b_j) with $c(a_i,b_j) \neq 1$. A min-decomposition of $\mathbf{F}_1(a,b)$ is also shown.

5.8.3 Set Separation

Non-bi-decomposable function sets can be simplified so that they become bi-decomposable. For Boolean functions, weak bi-decomposition was applied to non-bi-decomposable functions. A weak bi-decomposition of a Boolean ISF $F(A,C)$ with respect to the operator $\pi(x,y)$ is a pair of ISFs $G(A,C)$ and $H(C)$ so that:

1. There is $\mathbf{F}(A,C) \subset \mathbf{G}(A,C)$.
2. For every function $g(A,C) \in \mathbf{G}(A,C)$ there is a function $h(C) \in \mathbf{H}(C)$ with $\pi(g(A,C),h(C)) \in \mathbf{F}(A,C)$.

Weak bi-decomposition is similar to bi-decomposition except that it is not required that $G(A,C)$ depend on fewer variables than $F(A,C)$. Instead, to ensure a simplification of $G(A,C)$, it is required that $\mathbf{G}(A,C)$ be a superset of $\mathbf{F}(A,C)$. By this condition, the set $\mathbf{G}(A,C)$ introduces additional don't cares, and after successive weak bi-decompositions the function set is eventually bi-decomposable. In fact, it has been shown that a Boolean ISF is always bi-decomposable after two successive weak bi-decompositions [38].

The situation is more complicated for MVL function sets. Because MVL variables can have more than two values, the removal of an MVL variable puts stronger restrictions on the function set than the removal of a Boolean variable. A 4-valued variable for instance, can be thought of as being equivalent to two Boolean variables. The above mentioned weak bi-decomposition concept would require the simultaneous removal of the two Boolean variables, which is not always possible. Therefore, another strategy is proposed for MVL function sets.

The concept of weak bi-decomposition can be modified so that it can be applied to MVL function sets. Instead requiring that the function set $\mathbf{H}(C)$ depend on fewer variables, it is only required that $\mathbf{H}(A,C)$ be bi-decomposable. This concept is called *set separation*.

Definition 40. A *set separation* of the function set $\mathbf{F}(A,B,C)$ with respect to the operators $\pi(x,y)$ and $\rho(x,y)$ is a function set $\mathbf{S}(A,B,C) \supset \mathbf{F}(A,B,C)$ so that for all functions $s(A,B,C) \in \mathbf{S}(A,B,C)$ there exist functions $g(A,C)$ and $h(B,C)$ with

$$\pi(s(A,B,C),\rho(g(A,C),h(B,C))) \in \mathbf{F}(A,B,C). \tag{5.102}$$

In the following, set separation of function intervals with respect to the max- and min-operators is discussed, where the function set $\mathbf{S}(A,B,C)$ should be a function interval. This set separation is called *max-min-separation*.

Definition 41. A *max-min-separation* of a function interval $\mathbf{F}(A,B,C)$ is a function interval $\mathbf{S}(A,B,C)$ with $\mathbf{S}(A,B,C) \supset \mathbf{F}(A,B,C)$ where for all functions $s(A,B,C) \in \mathbf{S}(A,B,C)$ there exist functions $g(A,C)$ and $h(B,C)$ with

$$\max(s(A,B,C),\min(g(A,C),h(B,C))) \in \mathbf{F}(A,B,C). \tag{5.103}$$

A max-min-separation can be computed backwards by the choice of a min-decomposable function $d(A,B,C) = \min(g(A,C),h(B,C))$. The separation set $\mathbf{S}(A,B,C)$ can then be computed from the set of all functions $s(A,B,C)$ that satisfy (5.103). Theorem 38 shows how to compute the function interval $\mathbf{S}(A,B,C)$.

Theorem 38. *Let $\mathbf{F}(A,B,C) = [f_l,f_u]$ be a function interval and $d(A,B,C) \leq f_u(A,B,C)$ be an MVL function. Then, for all functions $s(A,B,C) \in [s_l(A,B,C),f_u(A,B,C)]$ there is*

$$\max(s(A,B,C),d(A,B,C)) \in \mathbf{F}(A,B,C), \tag{5.104}$$

where

$$s_l(A,B,C) = \mathrm{leq}_0(f_l(A,B,C),d(A,B,C)). \tag{5.105}$$

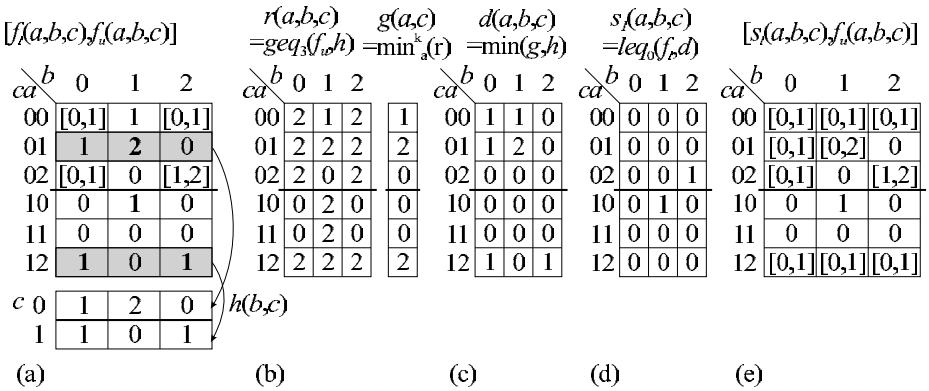

Figure 5.17: Min-max-separation of the function interval $\mathbf{F}(a,b,c) = [f_l, f_u]$. **(a)** Function interval $\mathbf{F}(a,b,c)$ and function $h(b,c)$. **(b)** Computation of function $g(a,c)$. **(c)** Separation function $d(a,b,c) = \min(g(a,b), h(b,c))$. **(d)** Lower bound $s_l(a,b,c)$ of the separation interval $\mathbf{S}(a,b,c)$. **(e)** Separation interval $\mathbf{S}(a,b,c)$.

Proof. The theorem follows directly from Theorem 30. □

The upper bound of the separation interval $\mathbf{S}(A,B,C)$ is the same as in the original function interval $\mathbf{F}(A,B,C)$ independent of the choice of the function $d(A,B,C)$. Therefore, the only way to increase the function interval $\mathbf{S}(A,B,C)$ so that it contains more functions than the function interval $\mathbf{F}(A,B,C)$ is to decrease the lower bound $s_l(A,B,C)$ of the separation interval $\mathbf{S}(A,B,C)$.

Consider a minterm (A_i, B_j, C_k) where

$$f_l(A_i,B_j,C_k) \leq d(A_i,B_j,C_k). \tag{5.106}$$

It follows from (5.105) that $s_l(A_i,B_j,C_k) = 0$. If there is $f_l(A_i,B_j,C_k) > 0$ then, the interval $\mathbf{S}(A_i,B_j,C_k)$ is larger than the interval $\mathbf{F}(A_i,B_j,C_k)$, and hence, $\mathbf{S}(A,B,C)$ contains more functions than $\mathbf{F}(A,B,C)$. A minterm (A_i,B_j,C_k) that satisfies (5.106) is said to be *covered* by the separation function $d(A,B,C)$.

Example 5.19. Consider the max-min-separation of the function interval $\mathbf{F}(a,b,c)$, shown in Figure 5.17(a) with respect to the variables a and b. The function interval $\mathbf{F}(a,b,c)$ is depicted in two decomposition charts separated by the bold line for $c = 0$ and $c = 1$. A min-decomposable function is the function $d(a,b,c)$ shown in Figure 5.17(c). The lower bound $s_l(a,b,c)$ of the separation interval $\mathbf{S}(a,b,c) = [s_l(a,b,c), f_u(a,b,c)]$ can be computed by (5.105) (Figure 5.17(d)). The function $d(a,b,c)$ covers all minterms except $(abc) = (220)$ and $(abc) = (011)$ of the function interval $\mathbf{F}(a,b,c)$.

There remains the question how the separation function $d(A,B,C) = \min(g(A,C), h(B,C))$ should be chosen so that the interval $\mathbf{S}(A,B,C)$ contains as many functions as possible. No efficient algorithm is known to find a separation function $d(A,B,C)$ that maximizes the number of functions in the separation interval $\mathbf{S}(A,B,C)$. Therefore, a heuristic algorithm for the computation of the separation function $d(A,B,C)$ is described.

The number of functions in the separation interval $\mathbf{S}(A,B,C)$ is increased by the minterms of $\mathbf{F}(A,B,C)$ that are covered by the separation function $d(A,B,C)$. Intuitively, the separation function $d(A,B,C)$ should cover

1. many minterms of the interval $F(A,B,C)$,
2. minterms with a large value of $f_l(A,B,C)$.

Note that the function $d(A,B,C)$ is min-decomposable if and only if for all minterms C_k the functions $d_k(A,B) = d(A,B,C_k)$ are min-decomposable. Therefore, the functions $d_k(A,B)$ can be considered separately. Every function $d_k(A,B)$ is depicted in a decomposition chart with the variables A at the columns and the variables B at the rows of the chart. A first heuristic selects the functions $d_k(A,B)$ so that a complete row containing the maximum value of the chart of

$f_l(A,B,C_k)$ is covered. The selection of rows that are covered can be described by a selection function $\alpha(C)$ which assumes the value of the variables A of the selected row.

Example 5.20. The maximum values of $f_l(a,b,c_k)$ from Figure 5.17(a) are shown as bold printed numbers. The rows that will be covered by $d(a,b,c)$ are indicated by shaded cells of the chart. The selection function $\alpha(c)$ has the values $\alpha(c=0)=1$ because the row $a=1$ was selected for $c=0$, and there is $\alpha(c=1)=2$ because the row $a=2$ was selected for $c=1$. The function $h(b,c)=f_l(\alpha(c),b,c)$ is shown at the bottom of Figure 5.17(a).

A min-decomposable function $d(A,B,C)$ that covers the selected rows of the decomposition chart can be constructed by setting $d(A_i,B_j,C_k)=f_l(A_i,B_j,C_k)$ if the minterm (A_i,B_j,C_k) is covered and $d(A_l,B_m,C_n)=0$ otherwise:

$$d(A_i,B_j,C_k)=\begin{cases} f_l(A_i,B_j,C_k) & \text{if } \alpha(C_k)=A_i, \\ 0 & \text{otherwise.} \end{cases} \tag{5.107}$$

The function $d(A,B,C)$ constructed by (5.107) is always min-decomposable. One min-decomposition of $d(A,B,C)$ is a function $g(A,C)$ that is equal to the maximum value of the non-zero row of $d(A,B,C)$ and zero everywhere else. The function $h(B,C)$ has the values of the non-zero row of $d(A,B,C)$:

$$g(A,C)=\max_B{}^{\mathrm{k}} d(A,B,C), \tag{5.108a}$$

$$h(B,C)=d(\alpha(C),B,C). \tag{5.108b}$$

In some cases it is possible to cover more values of $\mathbf{F}(A,B,C)$ by increasing the function $g(A,C)$. The upper bound for the function $g(A,C)$ is given by the restriction $d(A,B,C)\le f_u(A,B,C)$. Therefore, the function $g(A,B)$ can be computed by (5.71) to be

$$g(A,C)=\min_B{}^{\mathrm{k}}\,\mathrm{geq}_{m_o}(f_u(A,B,C),h(B,C)). \tag{5.109}$$

The separation set $\mathbf{S}(A,B,C)$ can then be computed by equation (5.105) with $d(A,B,C)=\min(g(A,C),h(B,C))$.

Example 5.21. For the function interval $\mathbf{F}(a,b,c)$ (Figure 5.17(a)) there is

$$g(a,c)=\min_a{}^{\mathrm{k}}\,\mathrm{geq}_3(f_u(a,b,c),h(b,c))$$

(Figure 5.17(b)). The lower bound $s_l(a,b,c)$ (Figure 5.17(d)) of the separation interval $\mathbf{S}(a,b,c)$ (Figure 5.17(e)) is computed from the function $d(a,b,c)=\min(g(a,c),h(b,c))$ (Figure 5.17(c)).

Chapter 6

Decomposition Algorithms

6.1 Overview

The properties shown in Chapter 5 are applied in this chapter to develop decomposition algorithms. The computational complexity of algorithms greatly depends on the underlying data structure. Therefore, BEMDD based data structures for function sets are introduced in Section 6.2 before bi-decomposition and separation algorithms are developed in Sections 6.3 and 6.4 respectively.

The final goal is not the computation of a single decomposition but the complete decomposition of function sets which requires additional algorithms. The selection of the free and bound sets for a decomposition is shown in Section 6.5. Strategies for the selection of the decomposition operator are discussed in Section 6.6. The recursive decomposition algorithm that completely bi-decomposes function sets is the topic of Section 6.7. Algorithms that convert and simplify function sets are shown in Sections 6.8 and 6.9.

6.2 Data structures

There are MVL decomposition systems that are based on MDDs [17]. However, decomposition systems that apply BEMDDs have been proven to be orders of magnitudes faster than MDD-based systems [25]. Therefore, the algorithms in this chapter are based on BEMDDs.

BEMDDs represent MVL relations by a special encoding of a BDD (see Section 2.4.5). Because function intervals, ISFs and functions are special cases of MVL relations, it would be possible to represent these function sets by a single BEMDD. Especially for function intervals there is a representation that requires slightly more memory, but allows much more efficient algorithms.

A function interval $\mathbf{F}(A) = [f_l, f_u]$ is represented by a pair $F(A) = \langle F.l(A), F.u(A) \rangle$ of BEMDDs [25]. The BEMDD $F.l(A)$ is the upper bound interval of $f_l(A)$ the BEMDD $F.u(A)$ is the lower bound interval of $f_u(A)$. This representation allows the fast computation of derivation operations on the bounds of the interval. This is important because the min- and max-decomposition (MM-decomposition) from Sections 5.5 and 5.6 apply these derivation operators during test and computation of decompositions.

Functions are stored by a single BEMDD. Functions are mainly stored as the result of decompositions for later reuse. Therefore, there can be very many functions and memory is important. A function that is the result of a complete decomposition can be reused if it is a member function of a function set that still must be decomposed. As shown below, this test can be performed very efficiently if a function is represented by a single BEMDD.

CISFs were defined in Section 4.3.5 as intersection of modular ISFs. A naive data structure for a CISF is a vector of BEMDDs encoding the core ISFs of the modular ISFs as relations. This approach is inefficient if the number of modular ISFs is large. In Section 5.7 it was shown that CISFs arise in modsum-decomposition of ISFs and that each minterm C_i of the shared variables C generates a set of modular ISFs. In general, the vector C may contain many variables and the number of minterms C_i can be very large which implies a great number of modular ISFs.

Table 6.1: Decomposition algorithms by function set and operator

Section	max, min	Monotone	Simple	modSum	Separation	Multi-Dec
ISF				6.3.6		
Interval	6.3.5	6.3.3			6.4.2	6.4.1
Relation	6.3.4		6.3.4	(6.3.6)		
CISF	6.3.7					

Therefore, the variables C are used to encode a modular ISF for every minterm C_i in a single BEMDD, which reduces the number of BEMDDs significantly.

A modular ISF $\mathbf{M}(A)$ can be represented by its core ISF $F(A)$. A member ISF of $\mathbf{M}(A)$ can be computed by the modsum $F(A) \oplus_m k$ of the ISF $F(A)$ with the m-valued constant k. This representation can be extended. A set of modular ISFs $\mathbf{M}(A, D)$, i.e. one modular ISF for every minterm D_k can be represented by an ISF $F(A, D)$. Member ISFs of the set $\mathbf{M}(A, D)$ can be computed by the modsum $F(A, D) \oplus_m k(D)$ of the ISF $F(A, D)$ with the m-valued function $k(D)$. In this representation the variables D are called *independent*.

A CISF $\mathbf{S}(A, D)$ can be represented by a set of modular ISFs $S.F(A, D)$ with the set of independent variables D. The set of modular ISFs $S.F(A, D)$ is represented as relations by a set of BEMDDs $S.F(A, D) = \{S.F_1(A, D), \ldots, S.F_n(A, D)\}$.

No efficient data structures are known for function lattices or general function sets. If necessary, the decomposition algorithms of Section 6.3 approximate function lattices by an MVL relation.

In order to reuse functions that have already been synthesized, it is necessary to check if these functions are elements of a given function set that still needs to be synthesized. If a function $f(A)$ is represented by the BEMDD relation $f\langle A, z\rangle$, its membership $f(A) \in \mathbf{R}(A)$ in a relation $R\langle A, z\rangle$ can be tested by a single BDD operation

$$f\langle A, z\rangle \cap R\langle A, z\rangle = f\langle A, z\rangle . \tag{6.1}$$

If a function interval $\mathbf{F}(A)$ is represented by a pair of BEMDDs $\langle F.l\langle A, z\rangle , F.u\langle A, z\rangle\rangle$, a function $f(A) \in \mathbf{F}(A)$, represented by the BEMDD $f\langle A, z\rangle$ must be contained in both relations $F.l\langle A, z\rangle$ and $F.u\langle A, z\rangle$ respectively which results in the membership test

$$F.l\langle A, z\rangle \cap F.u\langle A, z\rangle \cap f\langle A, z\rangle = f\langle A, z\rangle . \tag{6.2}$$

6.3 Bi-Decomposition Algorithms

6.3.1 Classes of Decomposition Algorithms

This section applies the properties from Chapter 5 to develop decomposition algorithms. The input to the algorithms is an operator $\pi(x, y)$, a function set $\mathbf{F}(A, B, C)$, the free set A and the bound set B of the decomposition. Three classes of algorithms are developed in the following sections.

1. **Test.** The algorithm class TEST$(\pi, \mathbf{F}, A, B)$ returns **true** iff a π-decomposition of the function set $\mathbf{F}(A, B, C)$ with respect to the free set A and the bound set B exists.

2. **Computation of the decomposition set.** The algorithm class FREE-SET$(\pi, \mathbf{F}, A, B)$ returns the decomposition set $\mathbf{G}(A, C)$ of the π-decomposition of the function set $\mathbf{F}(A, B, C)$ with respect to the free set A and the bound set B.

3. **Computation of the decomposition set with respect to a function.** The algorithm class BOUND-SET$(\pi, \mathbf{F}, A, g)$ returns the decomposition set $\mathbf{H}(B, C)$ of the π-decomposition of the function set $\mathbf{F}(A, B, C)$ with respect to the free set A, the bound set B and the decomposition function $g(A, C)$.

The algorithms depend on the type of operator $\pi(x, y)$ and on the class of function set $F(X)$. Table 6.1 gives an overview of the algorithms presented in the following sections. If there is no

algorithm for a certain combination of a function set and an operator in Table 6.1 (e.g. monotone operators for MVL relations), a decomposition can still be computed by conversion of the function set to another class of function sets with a suitable decomposition algorithm (e.g. function interval). If the destination class of function sets is more general (e.g. conversion from ISF to function interval), all possible decompositions will be found. If the conversion is to a more specialized class of function sets, some function sets may be found to be not decomposable, although they are decomposable. This is the case when all decomposable functions contained in the original function set were lost during the conversion process. Conversion algorithms between function sets are discussed in Section 6.8.

The modsum-decomposition algorithm from Section 6.3.6 is exact only for ISFs. The algorithm has been extendend to decompose MVL relations, but then the algorithm does not find all decompositions. However, the algorithm is still more efficient than a decomposing the result of the conversion from the relation to an ISF.

To avoid cluttering the algorithms with the variables of the BEMDD relations, these variables are only shown in the pseudo code when the BEMDD is first introduced.

6.3.2 Computation of Decomposition Sets for Relations

For some operators (e.g. monotone operators) there exist efficient algorithms for the decomposition test, but no efficient algorithms are known to compute the decomposition sets. If a relation is decomposed, Theorem 25 shows that the decomposition set of a relation $R\langle(A,B,C),z\rangle$ with respect to a decomposition function $g(A,C)$ and an operator $\pi(x,y)$ is relation $H\langle(B,C),y\rangle$. There is $\langle(B_j,C_k),y_l\rangle \in H\langle(B,C),y\rangle$ iff there is

$$\forall A_i : \langle(A_i,B_j,C_k),\pi(g(A_i,C_k),y_l)\rangle \in R\langle(A,B,C),z\rangle\,. \tag{6.3}$$

The algorithm Bound-Set-MVR(π,R,g,A) computes the bound set of the π-decomposition of the relation $R\langle(A,B,C),z\rangle$ with respect to the decomposition function $g(A,C)$ and the free set of variables A. The operator π, the relation R and the function $g(A,C)$ are represented by their BEMDD relations $\pi\langle(x,y),z\rangle$, $R\langle(A,B,C),z\rangle$ and $g\langle(A,C),x\rangle$ respectively. The algorithm returns the BEMDD of the relation $H\langle(B,C),y\rangle$ that is the decomposition set of the relation R.

Bound-Set-MVR$(\pi\langle(x,y),z\rangle, R\langle(A,B,C),z\rangle, g\langle(A,C),x\rangle, A)$
1 $G\langle(A,C,y),z\rangle = \max^k{}_x(\pi \wedge g)$
2 $F\langle(A,B,C),y\rangle = \max^k{}_z(G \wedge R)$
3 $H\langle(B,C),y\rangle = \min^k{}_A F$
4 **return** H

The BEMDD $G\langle(A,C,y),z\rangle$ is a relation between the values y_l and z_m so that $\pi(g(A_i,C_k),y_l) = z_m$. The BEMDD $F\langle(A,B,C),y\rangle$ is a relation of all values y_l that satisfy (6.3) for a particular minterm A_i. The bound decomposition set $H\langle(B,C),y\rangle$ is obtained by universal quantification over the variables A.

A similar algorithm can be applied to approximate the free set of a relation. Instead of passing a function from the free set, a decomposition function $h(B,C)$ from the bound set is given to the algorithm Free-Set-MVR(π,R,h,B) to compute an approximation of the free set $G((A,C),x)$.

Free-Set-MVR$(\pi\langle(x,y),z\rangle, R\langle(A,B,C),z\rangle, h\langle(B,C),y\rangle, B)$
1 $H\langle(B,C,x),z\rangle = \max^k{}_y(\pi \wedge h)$
2 $F\langle(A,B,C),x\rangle = \max^k{}_z(H \wedge R)$
3 $G\langle(A,C),x\rangle = \min^k{}_B F$
4 **return** G

The relation G returned by algorithm Free-Set-MVR is only an approximation of the free set of R because there may be functions $g_i(A,C) \notin G$ that are decomposition functions with respect to a bound function $h_1(B,C) \neq h(B,C)$.

6.3.3 Monotone Operator Decomposition

The decomposition of a function interval $\mathbf{F}(A,B,C) = [f_l, f_u]$ with respect to a monotone operator $\mu(x,y)$ is computed by relaxation, see Section 5.3. Initially the free function $g_l(A,C)$ is set to the smallest possible function $g_l(A,C) = 0$ Iteratively, the function $h_u(A,C)$ is computed to be the greatest function that satisfies the decomposition criteria

$$\mu(g_l(A,C), h_u(B,C)) \leq f_u(A,B,C) \tag{6.4}$$

and the function $g_l(A,C)$ is computed to be the smallest function that satisfies the decomposition criteria

$$\mu(g(A,C), h(B,C)) \geq f_l(A,B,C). \tag{6.5}$$

The function $g_l(A,C)$ is monotonically increased, while the function $h_u(B,C)$ is montonically decreased. If the function interval $\mathbf{F}(A,B,C)$ is decomposable, there must be an iteration, where $g_l(A,C) = \inf \mathbf{G}(A,C)$ is the lower bound of the decomposition set $\mathbf{G}(A,C)$ and $h_u(B,C) = \sup \mathbf{H}(B,C)$ is the upper bound of the decomposition set $\mathbf{H}(B,C)$. Then, the pair $\langle g_l(A,C), h_u(B,C)\rangle$ is a decomposition of $\mathbf{F}(A,B,C)$ (see Theorem 27), which can be checked by

$$\mu(g(A,C), h(B,C)) \in \mathbf{F}(A,B,C) \tag{6.6}$$

to termintate the iteration. If $\mathbf{F}(A,B,C)$ is not decomposable, there must be an iteration, where (6.4) or (6.5) cannot be satisfied which also terminates the iteration.

The algorithm MON-DECOMPOSITION$(\mu\langle(x,y),z\rangle, F\langle A,B,C,z\rangle, A, B)$ gets the BEMDD of a monotone operator mu, a function interval F represented by a pair of BEMDDs and two sets of functions A and B. The algorithms returns the BEMDDs $\langle g\langle(A,C),x\rangle, h\langle(B,C),y\rangle\rangle$ of a decomposition if F is decomposable, or the constant NOT-DECOMPOSABLE otherwise.

```
MON-DECOMPOSITION(μ⟨(x,y),z⟩, F⟨(A,B,C),z⟩, A, B)
 1  g_l⟨(A,C),x⟩ ← 0
 2  r ← u ← true
 3  while r = true do
 4        if u
 5          then g_y⟨(A,C,y),z⟩ ← max^k_x(μ ∩ g_l)
 6               f_y⟨(A,B,C),y⟩ ← max^k_z(g_y ∩ F.u)
 7               h_y⟨(B,C),y⟩ ← min^k_A f_y
 8               h_u⟨(B,C),y⟩ ← UPPER-BOUND(h_y)
 9               if max^k_y h_u ≠ 1
10                 then r ← false
11          else h_x⟨(B,C,x),z⟩ ← max^k_y(μ ∩ h_u)
12               f_x⟨(A,B,C),x⟩ ← max^k_z(h_x ∩ F.l)
13               g_x⟨(A,C),x⟩ ← min^k_B f_x
14               g_l ← LOWER-BOUND(g_x)
15               if max^k_x g_l ≠ 1
16                 then r ← false
17        if μ(g_l, h_u) ∈ F
18          r ← false
19        u ← ū
20  if μ(g_l, h_u) ∈ F
21    then return ⟨g_l, h_u⟩
22    else return NOT-DECOMPOSABLE
```

The algorithms starts with $g_l = 0$ (line 1) and then alternates between decreasing h_u (lines 5–10) and increasing g_l (lines 11–16). A relation g_y is computed that contains all pairs $\langle y,z\rangle$ with $\mu(g_l,y) = z$ (line 5). The intersection with the lower bound interval $F.u$ of the upper bound of $\mathbf{F}(A,B,C)$ results in the relation f_y that contains all values y with $\mu(g,y) \leq f_u$ (line 6). Universal quantification over A and selection of the upper bound function by the algorithm UPPER-BOUND results in the function h_u that is the greatest function that satisfies (6.4) (lines 7 and 8). The algorithm terminates if the function h_u is incomplete, i.e. (6.4) could not be satisfied for at least one minterm (lines 9 and 10). The computation of the function g_l

(lines 11–16) is dual to the computation of the function h_u. The loop in line 3 is also terminated if a decomposition was found (lines 17 and 18). The algorithm returns the decomposition in line 21.

Test and computation of a decomposition are merged into one algorithm. Because the free decomposition set is a function lattice (see Theorem 27), which is difficult to store, only an approximation of the free decomposition set can be computed. If a decomposition $\langle g, h\rangle$ is found by the algorithm MON-DECOMPOSITION, an approximation of the free decomposition set can be computed by the algorithm FREE-SET-MVR from Section 6.3.2. The bound decomposition set with respect to a function $g(A, C)$ can be computed exactly by the algorithm BOUND-SET-MVR.

6.3.4 Simple Operator Decomposition

Simple operators are defined in Section 5.4 as a sequence of σ-transformations. A σ-transformation is a replacement or a row or column of the chart of the operator function by a constant value. In the algorithms below, a σ-transformation is stored as a structure $S = \{d, v, c\}$ of three values:

1. The scalar d indicates the direction of the replacement. Its value can be either 'COLUMN' or 'ROW'.
2. The integer v is the value of the variable where the replacement takes place.
3. The integer c is the constant value that is substituted during the replacement.

The σ-transformation $\mathcal{S}^1_{y=0}$ and $\mathcal{S}^2_{x=0}$ shown in the second and the third chart of in Figure 5.9 for example, are stored in the structures $S_1 = \{$'COLUMN', 0, 1$\}$ and $S_2 = \{$'ROW', 0, 2$\}$.

The algorithm MAKE-SIMPLE-OPERATOR(S) computes the operator function of a simple operator from a vector S of σ-transformations.

```
MAKE-SIMPLE-OPERATOR(S)
 1  for i ← 0...LENGTH(S) − 1 do
 2      if S[i].d = 'ROW'
 3         then for y ← 0...m_y − 1 do
 4                  σ[S[i].v, y] ← S[i].c
 5         else  for x ← 0...m_x − 1 do
 6                  σ[x, S[i].v] ← S[i].c
 7  return σ
```

It was shown in Theorem 28 that the decomposability of a relation does not change if don't cares are added to the relation, where the relation contains the constant value c, and the the first of the σ-transformations is removed from the operator. This simplification is applied to compute decompositions of relations with respect to simple operators.

The input of the algorithm SIMPLE-DECOMPOSITION(S, R, A, B) is the vector S of σ-transformations of a simple operator σ, a relation $R\langle(A, B, C), z\rangle$ and two sets of variables A and B. If the relation R is σ-decomposable, the algorithm returns a pair of functions $\langle g, h\rangle$ that is a σ-decomposition of R with respect to the sets of variables A and B, otherwise the constant NOT-DECOMPOSABLE is returned.

```
SIMPLE-DECOMPOSITION(S, R⟨(A, B, C), z⟩, A, B)
 1  d(A, B, C) ← g⟨(A, C), x⟩ ← h⟨B, C, y⟩ ← 0
 2  x0 ← x : ∀S[i].d = ROW : S[i].v ≠ x
 3  y0 ← y : ∀S[i].d = COLUMN : S[i].v ≠ y
 4  for i ← LENGTH(S) − 1...0 do
 5      if S[i].d = ROW
 6         then e(A, B, C) ← max^k_z(min^k_B R ∧ S[i].c) ∧ d̄
 7              g ← g ∨ (e ∧ S[i].v)
 8         else  e ← max^k_z(min^k_A R ∧ S[i].c) ∧ d̄
 9              h ← h ∨ (e ∧ S[i].v)
10      R ← R ∨ e
11      d ← d ∨ e
```

```
12  g ← g ∨ (d̄ ∧ x0)
13  h ← h ∨ (d̄ ∧ y0)
14  if R = 1
15     then return ⟨g, h⟩
16     else  return NOT-DECOMPOSABLE
```

The neutral elements of the simple operator are computed in lines 2 and 3. The relation R is simplified for each σ-transformation of the vector S in line 10. The corresponding values of the decomposition functions are stored in lines 12 and 13. The minterms for which the relation R can be simplified are represented by the two-valued function $e(A, B, C)$ in lines 6 and 8. The two-valued function $d(A, B, C)$ indicates the minterms where the relation has been simplified by previous σ-transformations (see line 11) to prevent overwriting of values of the decomposition functions g and h.

Similar to monotone decomposition, the free set can be approximated by the algorithm FREE-SET-MVR from Section 6.3.2. The bound decomposition set with respect to a function $g(A, C)$ can be computed exactly by the algorithm BOUND-SET-MVR.

6.3.5 Min- and Max-Decomposition

Min- and Max decompositions can be computed directly by the formulas shown in Sections 5.5 and 5.6. Therefore, no special pseudo code for these algorithms is shown here.

6.3.6 Modsum-Decomposition

It was shown in Section 5.7 that modsum-decomposition of ISFs can be computed by solving systems of bilinear equations. Such a system of equations can be solved by repeatedly assigning a constant to one variable of the system and then computing the values of all dependent variables. This algorithm can be extended to relations. The resulting decomposition algorithm may not find all possible decomposition of the relation, but the chances of finding a decomposition are still better than by decomposing the ISF that results from a conversion of the relation to a single ISF. Many different ISFs are inspected during the solution of a system of equations defined by a relation.

The algorithm MODSUM-DECOMPOSITION($R\langle(A, B, C), v\rangle, A, B$) gets the BEMDD of the relation $R\langle(A, B, C), v\rangle$ and the sets of variables A and B of the free and bound set respectively. If a decomposition was found, the algorithm returns a pair of CISFs $\langle g, h\rangle$ of a modsum-decomposition, otherwise the constant NOT-DECOMPOSABLE is returned.

```
MODSUM-DECOMPOSITION(R⟨(A, B, C), v⟩, A, B)
 1  i ← 0
 2  while R ≠ 1 do
 3        gr⟨(A, C), v⟩ ← hr⟨(B, C), v⟩ ← 1
 4        gt⟨(A, C), v⟩ ← PICK-START(R, A)
 5        do
 6           gr ← gr ∧ gt
 7           ⟨ok, R, ht⟩ ← TRANSFORM(R, gt, A)
 8           if ok = false
 9              then return NOT-DECOMPOSABLE
10              else  if ht ≠ 1
11                       then hr ← hr ∧ ht
12                            ⟨ok, R, gt⟩ ← TRANSFORM(R, ht, B)
13                            if ok = false
14                               then return NOT-DECOMPOSABLE
15        while ht ≠ 1 and gt ≠ 1
16        g[i] ← gr
17        h[i] ← hr
18        i ← i + 1
19  return ⟨g, h⟩
```

The algorithm successively chooses a partial solution of the system of bilinear equations by the algorithm PICK-START in line 4 and computes all dependent values of the decomposition functions in the loop in lines 5–15. The values of the decomposition functions are collected in the variables gr and hr and are stored as a CISF in lines 16 and 17. The algorithm TRANSFORM called in line 7 gets a set of minterms gt where the decomposition function gr is known and returns the set of minterms ht where the values of the decomposition function hr depend on gr. The algorithm also returns a Boolean ok that indicates that no a contradiction was found. The algorithm TRANSFORM masks the minterms of R, where a solution was computed by don't cares and returns the modified relation. Similarly, the call of the algorithm TRANSFORM in line 12 computes the minterms gt where the decomposition function gr depends on the known values of function hr. If a contradiction was found, algorithm MODSUM-DECOMPOSITION returns NOT-DECOMPOSABLE. If no contradiction was found, further dependent values are computed until all dependent equations of have been solved and and the loop in line 15 exits. This process is repeated until all values of the decomposition functions are known which results in the ISF Φ represented by the BEMDD $R = 1$. In this case the loop in line 2 exits and the computed CISF is returned.

6.3.7 Decomposition of CISF

A CISF is the intersection of a set of modular ISFs, see Definition 26. The non-empty intersection of ISFs is an ISF, see Theorem 14. Therefore, if the modules of the modular components of the CISF are known, the decomposition problem of CISF is reduced to the decomposition problem of ISF. A greedy algorithm is applied to compute the modules of the CISF. The algorithm starts with the first modular ISF and increases the module from 0 until a decomposable ISF is found. Then, this procedure is repeated for the second component, but the decomposability is checked for the intersection of both ISFs. The process is repeated for all modular ISFs of the CISF until all modules are determined and a decomposable ISF, which is contained in the CISF, is found.

The input of the algorithm CISF-MAX-DECOMPOSITION(S, A, B) is a CISF S and two sets of variables A and B. The algorithm returns a max-decomposable ISF if it can find one, or the constant NOT-DECOMPOSABLE otherwise.

```
CISF-MAX-DECOMPOSITION(S ⟨(A,B,C),z⟩, A, B)
 1  E ← SUPPORT(S) \ S.D
 2  R⟨(A,B,C),z⟩ ← 1
 3  for i ← 0,...,LENGTH(S.F) − 1 do
 4      T⟨(A,B,C),z⟩ ← S.F[i]
 5      for j ← 0,...,MVL(S) − 1 do
 6          R_T⟨(A,B,C),z⟩ ← R ∧ (T ⊕_MVL(R) j)
 7          I⟨(A,B,C),z⟩ ← ISF-TO-FI(R_T)
 8          D(A,B,C) ← max^k_z((min^k_A I.u ∨ min^k_B I.u) ∧ I.l)
 9          F(C \ E) ← min^k_{A∪B∪E} D
10          R ← R ∧ (R_T ∨ F̄)
11          T ← T ∨ F
12      if T ≠ 1
13        then return NOT-DECOMPOSABLE
14  return R
```

The algorithm tries to find a max-decomposable ISF R that is contained in the the CISF S. The loop in line 3 iterates over all components $F[i]$ of S. All modules j are tested in the loop starting at line 5. The ISF R_T contains the intersection of the component ISF $T \oplus_m j$ and ISF R (see line 6). The minterms $C \setminus E$ where the ISF R_T is max-decomposable are computed in lines 7–9. The function F has value 1 for each minterm of $C \setminus E$ where the ISF R_T is max-decomposable. For these minterms the ISF R_T is added to the result R in line 10. These minterms are also removed from the modular ISF T in line 11. The remaining cares of T are then tested with next module j in the next interation of the loop in line 5. If all cares could be added to the ISF R, the relation T contains only don't cares, i.e. its BEMDD is '1'. This condition is tested in line 12. If cares remain in T, no decomposable ISF could be found and the algorithm returns the constant NOT-DECOMPOSABLE in line 13. If all components could be added to the ISF R, the ISF is returned in line 14.

If the algorithm CISF-MAX-DECOMPOSITION(S, A, B) returns an ISF, the ISF is always max-decomposable. Therefore, a heuristic decomposition test algorithm can be formulated. Algorithms for the computation of the free and bound sets of functions of the CISF can be derived from the corresponding algorithms for the max-decomposition of function intervals. Similar algorithms can be developed for the test and computation of min-decompositions of CISFs.

6.4 Separation Algorithms

6.4.1 Multi-Decomposition

Multi-decomposition of function intervals involves three steps, see Section 5.8.2:

1. Computation of the incompatibility graph.
2. Coloring of the incompatibility graph.
3. Selection of a decomposable subfunction.

The incompatibility graph is stored as a two-valued function where the variables of the free and bound sets are duplicated. Source and destination nodes of the graph are encoded by distinct sets of variables S and D. A graph $G(S, D)$ has an edge from node S_i to node D_j iff there is $G(S_i, D_j) = 1$.

In multi-decomposition of a function interval $F(A, B, C)$, a node is encoded by the set of variables (A, B, C). To represent the incompatibility graph by a BEMDD, these variables must be duplicated to have variables A_S, B_S, C_S for the source nodes and variables A_D, B_D, C_D for the destination nodes. There are no incompatibility edges between nodes with different values of the shared variables C. Therefore, these variables are not duplicated. There is $C = C_S = C_D$.

In Definition 39 an incompatibility graph is defined as a set of nodes for each pair minterms where (5.97) is satisfied. To compute the incompatibility graph efficiently by BEMDD operations, this condition must be transformed. Inequality (5.97) is equivalent to

$$(f_l(A_i, B_m, C_k) > f_u(A_i, B_n, C_k)) \wedge (f_l(A_j, B_n, C_k) > f_u(A_i, B_n, C_k)) \tag{6.7}$$

Given BEMDDs $F.l\,\langle (A_S, B_S, C), z\rangle$ and $F.u\,\langle (A_S, B_D, C), z\rangle$ for the interval bounds $f_l(A, B, C)$ and $f_u(A, B, C)$ respectively, the term $f_l(A_i, B_m, C_k) \leq f_u(A_i, B_n, C_k)$ can be computed by the term

$$\max_z{}^{\mathrm{k}}(F.l\,\langle (A_S, B_S, C), z\rangle \wedge F.u\,\langle (A_S, B_D, C), z\rangle). \tag{6.8}$$

It follows that the negation, $f_l(A_i, B_m, C_k) > f_u(A_i, B_n, C_k)$, can be computed by the term

$$\min_z{}^{\mathrm{k}}\overline{F.l\,\langle (A_S, B_S, C), z\rangle \wedge F.u\,\langle (A_S, B_D, C), z\rangle}. \tag{6.9}$$

The input of the algorithm MAKE-MAX-INCOMPATIBILITY-GRAPH(F, A_S, B_S, A_D, B_D) is a function interval represented by a pair of BEMDDs $F\,\langle (A_S, B_S, C), z\rangle$ and four sets of variables A_S, B_S, A_D and B_D as input. The algorithm returns the incompatibility graph of the max-min-separation of F where A_S and B_S encode the source nodes and A_D and B_D encode the destination nodes of the graph.

MAKE-MAX-INCOMPATIBILITY-GRAPH$(F\,\langle (A_S, B_S, C), z\rangle, A_S, B_S, A_D, B_D)$
1 $F_{lSS}\,\langle (A_S, B_S, C), z\rangle \leftarrow F.l(A_S, B_S, C)$
2 $F_{lDD}\,\langle (A_D, B_D, C), z\rangle \leftarrow F.l(A_S \rightarrow A_D, B_S \rightarrow B_D, C)$
3 $F_{uSD}\,\langle (A_S, B_D, C), z\rangle \leftarrow F.u(A_S, B_S \rightarrow B_D, C)$
4 $F_{uDS}\,\langle (A_D, B_S, C), z\rangle \leftarrow F.u(A_S \rightarrow A_D, B_S, C)$
5 $G_1(A_S, A_D, B_S, B_D, C) \leftarrow \min^{\mathrm{k}}{}_z\overline{F_{uSD} \wedge F_{lSS}} \wedge \min^{\mathrm{k}}{}_z\overline{F_{uSD} \wedge F_{lDD}}$
6 $G_2(A_S, A_D, B_S, B_D, C) \leftarrow \min^{\mathrm{k}}{}_z\overline{F_{uDS} \wedge F_{lSS}} \wedge \min^{\mathrm{k}}{}_z\overline{F_{uDS} \wedge F_{lDD}}$
7 **return** $G_1 \vee G_2$

The bounds of the interval are transformed into BEMDDs in terms of of the destination variables in lines 1–4. The edges of the incompatibility graph are computed by (6.7) and (6.9) in line 5. The reflexive edges of the incompatibility graph are computed in line 6. The union of both sets of edges is returned in line 7.

The input of the algorithm MAX-MULTI-DECOMPOSITION(F, A, B) is a function interval F and two sets of variables A and B. The algorithm returns a function interval that is a subfunction of a max-min-separation of the function interval F.

MAX-MULTI-DECOMPOSITION$(F\langle(A,B,C),z\rangle, A, B)$
1 $A_D \leftarrow$ DUPLICATE-VARIABLES(A)
2 $B_D \leftarrow$ DUPLICATE-VARIABLES(B)
3 $G(A, A_D, B, B_D, C) \leftarrow$ MAKE-INCOMPATIBILITY-GRAPH(F, A, B, A_D, B_D)
4 $L\langle(A,B,C),z\rangle \leftarrow$ COLORIZE$(G, A \cup B, A_D \cup B_D)$
5 $R.l\langle(A,B,C),z\rangle \leftarrow \text{ite}(L, F.l, 0, \ldots, 0)$
6 $R.u\langle(A,B,C),z\rangle \leftarrow F.u$
7 **return** R

The algorithm COLORIZE(G, S, D) (line 4) returns a coloring of the graph G where the the source and destination nodes encoded by the variables S and D respectively. The Color '0' is selected for the function interval of the multi-decomposition in line 5. A discussion of graph coloring algorithms applied to functional decomposition can be found in [22]. In this work the method of dominant nodes from [26, 27] was applied.

6.4.2 Set Separation

A set separation of a function interval can be computed by the properties developed in Section 5.8.3.

The input of the algorithm MAX-SET-SEPARATION(F, A, B) is the pair of BEMDDs $F\langle(A,B,C),z\rangle$ of a function interval $[f_l(A,B,C), f_u(A,B,C)]$ and two sets of variables A and B. The algorithm returns the free function interval $G\langle(A,B,C),z\rangle$ of a max-min-separation of the function interval F with respect to the free and bound set of variables A and B respectively.

MAX-SET-SEPARATION$(F\langle(A,B,C),z\rangle, A, B)$
1 $g_0(A,C) \leftarrow$ SELECT-MAX-CORE$(F.u, A, B)$
2 $h\langle(B,C),z\rangle \leftarrow \max^k{}_A(F.l \wedge g0)$
3 $f_c(A,B,C) \leftarrow \max^k{}_z(F.u \wedge h)$
4 $g\langle(A,C),z\rangle \leftarrow \min^k{}_B(F.u \vee f_c)$
5 $f_g(A,B,C) \leftarrow \max^k{}_z(F.l \wedge g)$
6 $f_h(A,B,C) \leftarrow \min^k{}_z(F.l \vee \overline{h})$
7 $G.l\langle(A,B,C),z\rangle \leftarrow F.l \vee (f_g \wedge f_h)$
8 $G.u\langle(A,B,C),z\rangle \leftarrow F.u$
9 **return** G

The algorithm SELECT-MAX-CORE$(F.u, A, B)$ in line 1 returns a function $g_0(A, C_k)$ that has value '1' for exactly one minterm A_i so that the row A_i of the decomposition chart of $F(A,B,C_k)$ contains the maximum value of $f_u(A,B,C_k)$. The function $h(B,C)$ from (5.108b) is computed in line 2, and the function $g(A,C)$ from (5.109) is computed in lines 3 and 4. The lower bound $g_l(A,C)$ of the separation interval $\mathbf{G}(A,C)$ is computed by (5.105) in lines 5–7. The interval is returned in line 9.

6.5 Variable Set Selection

Decomposition test algorithms have been developed in Section 6.3. Now, suitable sets of variables for the bound and free set must be selected. Given a decomposition operator $\pi(x,y)$, there are several strategies to select the sets of variables for bound and free set. There are heuristics which are guided by parameters computed from the function set. Backtracking algorithms are also possible. This section introduces two algorithms for the selection of the free and bound sets. The first algorithm FIND-BEST-VARIABLE-SETS is an extension of the greedy algorithm

FIND-VARIABLE-SETS from Section 3.3.1. The second algorithm FIND-MAX-VARIABLE-SETS tries to maximize the number of functions that are contained in the decomposition sets.

The algorithm FIND-VARIABLE-SETS from Section 3.3.1 looks for a decomposition with single variables in both, the free and bound set respectively. Then, this decomposition is extended so that the free and bound set are of approximately equal size. The intial pair of single variables is chosen randomly (depending on the seqence of variables in the input). The algorithm can be impoved by considering all pairs of single variables as cores for a decomposition. If the function set has only very few decompositions, this search strategy does not increase the computation time very much because many pairs of variables must be tested by both algorithms before a decomposition is found. If the function set has many decompositions, the time for finding the best decomposition by the new algorithm would significantly increase because many core decompositions must be extended to find the best decomposition. To speed up the tests, a cache is added so that the result of previous decomposition tests are stored and looked up rather than computed again.

The algorithm FIND-BEST-VARIABLE-SETS gets an operator $\pi\langle(x,y),z\rangle$ and a function set $\mathbf{F}(A,B,C)$ as input. The result of the algorithm is a pair $\langle A,B\rangle$ of the variables of the free set A and the bound set B if a decomposition was found, or the pair $\langle\emptyset,\emptyset\rangle$ otherwise.

```
FIND-BEST-VARIABLE-SETS(π⟨(x,y),z⟩, F(A,B,C))
 1  A_b ← B_b ← ∅
 2  X ← SUPPORT(F)
 3  for all a ∈ X do
 4      A ← {a}
 5      for all b ∈ X, b ≠ a do
 6          B ← {b}
 7          if CACHE-TEST(F, π, A, B)
 8            then for all c ∈ X \ (A ∪ B) do
 9                     if CACHE-TEST(F, π, A ∪ {c}, B)
10                       then A ← A ∪ {c}
11                       else if CACHE-TEST(F, π, A, B ∪ {c})
12                              then B ← B ∪ {c}
13                                   if |A| > |B|
14                                     then ⟨A,B⟩ ← SWAP(A,B)
15          if |X|*|A| + |B| > |X|*|A_b| + |B_b|
16            then A_b ← A
17                 B_b ← B
18  return ⟨A_b, B_b⟩
```

The algorithm iterates over all pairs $\langle a,b\rangle$ of single variables (see the loops in lines 3 and 5). The algorithm CACHE-TEST(F,π,A,B) test whether a function set F is π-decomposable with respect to the free and bound sets A and B respectively. The algorithm also has a side effect, it stores the result of the decomposition test in a cache. A decomposition test is only performed if the result cannot be derived from the cache by the following two rules

1. If F is decomposable with respect to supersets of A and B, then F is also decomposable respect to A and B (see Theorem 12).

2. If F is not decomposable with respect to subsets of A and B, then F is not decomposable with respect to A and B.

The quality of a solution is evaluated in line 15. The formula maximizes the smaller of the free and bound set. Depending on the application, other quality measures might be applicable.

A different strategy for the selection of variables is followed in the algorithm FIND-MAX-VARIABLE-SETS. Similar to algorithm FIND-BEST-VARIABLE-SETS, all pairs of variables are testet and possible decompositions are extended. However, the algorithm does not maximize the number of variables in a particular set, but it maximizes the number of functions that are contained in the decomposition sets.

```
FIND-MAX-VARIABLE-SETS(π(x,y), F(A,B,C))
 1  X ← SUPPORT(F)
 2  S ← 0
 3  for ∀a,b ∈ X do
 4      r ← DEC-SET-TEST(π,F,{a},{b})
 5      if r > S
 6         then S ← r
 7              A ← {a}
 8              B ← {b}
 9  if S = 0
10     then return ⟨∅,∅⟩
11  A_a ← B_a ← X \ (A ∪ B)
12  while A_a ≠ ∅ or B_a ≠ ∅ do
13        S ← 0
14        for ∀a ∈ A_a do
15            r ← DEC-SET-TEST(π,F,A ∪ {a},B)
16            if r > S
17               then S ← r
18                    A_b ← A ∪ {a}
19                    B_b ← B
20            if r = 0
21               then A_a ← A_a \ {a}
22        for ∀b ∈ B_a do
23            r ← DEC-SET-TEST(π,F,A,B ∪ {b})
24            if r > S
25               then S ← r
26                    A_b ← A
27                    B_b ← B ∪ {b}
28            if r = 0
29               then B_a ← B_a \ {b}
30        A ← A_b
31        B ← B_b
32  return ⟨A,B⟩
```

The algorithm first tests all pairs of variables $\langle a, b\rangle$ (line 4 and stores the best decomposition in the sets of variables A and B (lines 5–8). The algorithm DEC-SET-TEST(π, F, A, B) (called in lines 4, 15 and 23) tests the π-decomposability of the function set $\mathbf{F}$ with respect to the free and bound sets A and B respectively. The algorithm returns a quality measure (e.g. the number of functions in the decomposition sets) of the decomposition. A value of 0 indicates that the function set is not decomposable. More positive values indicate better decompositions. If no decomposition was found after testing all pairs of single variables, empty sets are returned in line 10

Two sets of variables A_a and B_a are maintained to store the variables that are candidates to can be added to the free and bound set respectively. Initially, all remaining variables of the support are considered (line 11). If a decomposition is not possible, the corresponding variable is removed from the set (lines 21 and 29).

All candidate variables are tested to extend the current decomposition $\langle A, B\rangle$ (lines 15 and 23). If a better decomposition is found, this is stored by the variables A_b and B_b. After all candidate variables have been tested the best decomposition becomes the current decomposition for further extension (lines 30 and 31). If no further extension is possible, the candidate sets A_a and B_b are empty, and the loop in line 12 terminates. The best decomposition is returned in line 32.

6.6 Operator Selection

The selection of the best decomposition operator $\pi(x, y)$ can be guided by parameters computed from the function. If the number of operators is small, all operators can be tried and the best decomposition is chosen. There are several ways to compare decompositions. Possible cost

functions include the number of variables of a decomposition or the number of functions in the decomposition sets. Different classes of operators may be difficult to compare to each other. Bi-decomposition for instance is difficult to compare to separation in terms of the size of free and bound set. For MM-decomposition the decomposition sets can be computed exactly as opposed to monotone decomposition where only approximations of the decomposition sets are possible. Therefore, an MM-decomposition, with the same size of free and bound set, should be preferred to a monotone decomposition. To solve these problems, the operators are divided into groups. The groups of operators are tested sequentially. If at least one decomposition exists in a group, the best decomposition from this group is selected. The best operator is selected from the first group that has a decomposition.

The input of the algorithm SELECT-OPERATOR(Π, F) is a vector of operators Π and a function set $F(X)$. The algorithm returns the free and bound sets of a π-decomposition of F where π is an element of Π. The groups in Π are separated by the constant SEPARATOR as an element of the vector Π.

```
SELECT-OPERATOR(Π, F(X))
 1  S ← 0
 2  i ← 0
 3  while S = 0 do
 4        ⟨A, B⟩ ← FIND-VARIABLE-SETS(Π[i], F)
 5        r ← GET-QUALITY(Π[i], F, A, B)
 6        if r > S
 7          then S ← r
 8               π ← Π[i]
 9               A_b ← A
10               B_b ← B
11        i ← i + 1
12        if Π[i] = SEPARATOR
13          then i ← i + 1
14               if S > 0
15                 then return ⟨π, A_b, B_b⟩
```

The algorithm iterates over the elements of the vector Π (line 3). It is tested if a decomposition with respect to the operator $\Pi[i]$ exists in line 4. A quality measure is computed by the algorithm GET-QUALITY (line 5). The algorithm GET-QUALITY returns a positive quality measure of the decomposition, or '0' if there was no decomposition ($A = \emptyset, B = \emptyset$). If the decomposition is better than the best decomposition found so far, the decomposition is stored by the variables π, A_b and B_b in lines 8–10. If the next element in the vector Π is a separator and a decomposition was found, the best decomposition is returned in line 15, otherwise, the next group of operators is tested.

6.7 Recursive Decomposition

The algorithm DECOMPOSE gets a vector of operators Π and a function set $\mathbf{F}$. One function $f \in \mathbf{F}$ of the set is completely decomposed and returned by DECOMPOSE. The algorithm is a direct extension of the Boolean decomposition algorithm DECOMPOSE-ISF presented in Section 3.3 to the MVL domain.

```
DECOMPOSE(Π, F)
1  if |SUPPORT(F)| < 2
2    then return SIMPLE(F)
3  ⟨π, A, B⟩ ← SELECT-OPERATOR(F)
4  G ← COMPUTE-FREE-SET(π, F, A, B)
5  g ← DECOMPOSE(G)
6  H ← COMPUTE-BOUND-SET(π, F, g, A)
7  h ← DECOMPOSE(H)
8  return π(g, h)
```

6.8 Conversion Algorithms

Some algorithms are needed to convert one class of function set into another. The algorithms presented in this Chapter apply to function intervals and MVL relations. No conversion algorithms are presented for CISFs because the conversion to a function interval or relation would restrict the function too much. Instead, CISFs are decomposed by specialized decomposition algorithms that select a decomposable ISF within the CISF, see Section 6.3.7.

There remains the conversion between function intervals and relations. Conversions in both directions are necessary because the decomposition algorithms accept either relations or function intervals (see Table 6.1), but the other class may have been produced during the previous decomposition.

The conversion of from function intervals to relations is straight forward. Because a relations are a more general class of function sets (see Theorem 18) all functions of the interval can be contained in one relation. If a function interval is represented by a pair $F(A) = [F.l\,\langle A, z\rangle, F.u\,\langle A, z\rangle]$ of BEMDDs (see Section 6.2), the relation $R\langle A, z\rangle$ representing the function interval can be computed by

$$R\langle A, z\rangle = F.l\,\langle A, z\rangle \wedge F.u\,\langle A, z\rangle \tag{6.10}$$

The conversion of a relation $R\langle A, z\rangle$ into a function interval $[F.l\,\langle A, z\rangle, F.l\,\langle A, z\rangle]$ is more difficult. Because a relation is not always representable by a function interval (see Example 4.12), some functions of the relation will be lost during the conversion. The question is which functions should be discarded by the conversion? An obvious strategy is to include as many functions in the interval as possible. This means to approximate the special don't cares of the relation by intervals that are as large as possible. For a single minterm, the special don't care of the relation can be represented by a set of integers. For this set it is possible to to find the largest interval contained in the special don't care by a single sweep of the values. The function interval contained in a relation can now be computed by a recursive algorithm that traverses the BEMDD of the relation. Whenever the algorithm encounters the value variables of the relation, the sweep algorithm converts the special don't care into the function interval.

6.9 Support Minimization

Some function set contain functions that depend on fewer variables than whole the function set. These variables are called inessential variables (see Definition 18). Such function sets arise in machine learning or logic synthesis applications. They can also be the result of certain decomposition algorithms (e.g. set separation).

It is possible to simplify the function sets by selection of the subset of functions that depend on only a subset of variables. The selection of a nonempty subset of functions that depends on the smallest number of variables is called *support minimization*, which is a difficult problem. Various search strategies have been proposed [23]. The support minimization problem can be divided into three parts:

1. Test whether a set of variables is inessential in a function set.
2. Computation of the simplified subset of function.
3. Selection of a minimum support set of variables by a search algorithm.

The first two problems must be solved for each class of function sets, while the last problem is independent of the class of function set.

A method of computation of the inessential variables for function intervals is shown in Theorem 17. For the BEMDD representation of function intervals, the computation is straight forward. A set of variables B is inessential in a function interval $F(A, B) = [F.l\,\langle (A, B), z\rangle, F.u\,\langle (A, B), z\rangle]$ iff

$$\max_z{}^{\mathrm{k}}(\min_B{}^{\mathrm{k}} F.l \wedge \min_B{}^{\mathrm{k}} F.u) = 1. \tag{6.11}$$

If the variables B are inessential, the simplified interval is computed by

$$G(A) = [\min_B{}^{\mathrm{k}} F.l, \min_B{}^{\mathrm{k}} F.u]. \tag{6.12}$$

A set of variables B is inessential in a relation $R\langle(A,B),z\rangle$ if the relation contains functions that do not depend on B, i.e. that are constant over B. This property can be checked by the following BEMDD operations

$$\max_z{}^{\mathrm{k}} \min_B{}^{\mathrm{k}} R\langle(A,B),z\rangle = 1. \tag{6.13}$$

If the relation is independent the simplified relation can be computed by

$$G\langle A,z\rangle = \min_B{}^{\mathrm{k}} R\langle(A,B),z\rangle\,. \tag{6.14}$$

The general support minimization problem is NP-hard. Three algorithms are presented to find a set of inessential variables:

1. The algorithm ELIMINATE-SIMPLE(F) tries to remove the variables of the support of F in the sequence of the input. Because the algorithm is a simple loop over all variables, no pseudo code is presented for this algorithm.

2. The algorithm ELIMINATE-MAX-SET(F) successively removes the variable from the support so that the maximum number of functions remains in the simplified function set.

3. The algorithm ELIMINATE-TIME-OUT(F) is a memorized search strategy that terminates at the exact minimum support. Because the search can take very long, the algorithm has a time out. After the time out, the best solution, found so far, is extended by the algorithm ELIMINATE-SIMPLE.

The algorithm ELIMINATE-MAX-SET(F) gets a function set F and returns a simplified function set where inessential variables are removed so that the simplified function set contains many functions.

```
ELIMINATE-MAX-SET(F)
 1  X ← SUPPORT(F)
 2  while |X| > 0 do
 3        S ← 0
 4        for ∀x ∈ X do
 5           G ← REMOVE-VAR(F, {x})
 6           if |G| > S
 7              then S ← |G|
 8                   x_b = x
 9           if |G| = 0
10              then X ← X \ {x}
11        if S > 0
12          then F ← REMOVE-VAR(F, {x_b})
13  return F
```

A set X of candidate variables for removal is maintained. The set is initialized with the support of F (line 1). All variables of X are tried to be removed (lines 4 and 5). The algorithm REMOVE-VAR(F,X) removes the set of variables X from the function set F. If X is inessential, the simplified set, or otherwise, an empty function set is returned. If the size of the simplified set G is larger than the best set found so far, the variable is stored by x_b (line 8). If a variable was not inessential, it is removed from the set X of candidate variables (line 10). If x_b contains the best inessential variable, it is removed from the function set F (line 12), and the process is repeated for the remaining candidate variables. The removal terminates if no candidate variables are left (line 2), and the simplified function set is returned (line 13).

The algorithm ELIMINATE-TIME-OUT maintains a cache with all inessential function sets found so far by the algorithm. The cache is sorted by the size of the inessential function sets. The information from the cache is used to reduce the number of tests whether a set of variables is inessential. The algorithm IS-INESSENTIAL-CACHE(F,X,C) gets a function set F, a set of variables X and a set C of all inessential sets of variables of size $|X|-1$. The algorithm returns **true** if the set X is inessential in F.

```
IS-INESSENTIAL-CACHE(F, X, C)
 1  for ∀x ∈ X do
 2      if (X \ x) ∉ C
 3         then return false
 4  return REMOVE-VAR(F, X)
```

If a function has very few inessential variables, there only few sets of inessential variables. In this case it is possible to minimize the support exactly by search algorithms. However, these search algorithms take too much time if there are many inessential variables. Therefore, a search algorithm was implemented that tries to find an exact solution. If too much time was spent in finding the exact solution, an approximation of the exact solution is computed based on the results found until the time out.

The algorithm ELIMINATE-TIME-OUT starts the search with single variables for the set of inessential variables. The results are stored in the cache. Then, the size of the set of inessential variables is successively increased until no larger set of inessential variables can be found.

```
ELIMINATE-TIME-OUT(F)
 1  X ← SUPPORT(F)
 2  C[1] ← ∅
 3  for ∀x ∈ X do
 4      if REMOVE-VAR(F, {x}) ≠ ∅
 5         then C[1] ← C[1] ∪ {{x}}
 6  i ← 2
 7  while C[i − 1] ≠ ∅ do
 8          C[i] ← ∅
 9          for ∀Y ∈ C[i − 1] do
10              if TIME-OUT()
11                 then for ∀x ∈ X \ Y do
12                              if REMOVE-VAR(F, Y ∪ {x}) ≠ ∅
13                                 then Y ← Y ∪ {x}
14                      return REMOVE-VAR(F, Y)
15              for ∀x ∈ X \ Y do
16                  Z ← Y ∪ {x}
17                  if IS-INESSENTIAL-CACHE(F, Z, C[i − 1]) ≠ ∅
18                     then C[i] ← C[i] ∪ {Z}
19          i ← i + 1
20  if i > 2
21     then Y ← some element of C[i − 1]
22     else  Y ← ∅
23  return REMOVE-VAR(F, Y)
```

All single inessential variables are stored in the cache $C[1]$ (lines 1–5). Then, the algorithm iterates over all sets of inessential variables of size $i-1$ (line 7) and tries to increase these sets by a single variable (lines 15–18). If too much time has passed, algorithm TIME-OUT returns true (line 10), and the current set of inessential variables Y is extended by testing the remaining variables in the sequence of the input (lines 11–13). The simplified function set is returned in line 14 if there was a time out, or in line 23 if the algorithm found the exact solution.

Chapter 7

Decomposition System YADE

7.1 Overview

This chapter presents the implementation of the bi-decomposition system YADE (Yet Another DEcomposer) on a computer. The decomposition system gets an MVL function set as input. The output of the decomposition system is the representation of a member function of the function set as a network of MVL gates. The types of gates can be chosen as parameters of the decomposition system.

The network of MVL gates is computed by recursive bi-decomposition of function sets. Each decomposition can be considered as a node of a tree where the decomposition sets are the children of the node. Literal functions of single variables form the leaves of the tree. The decomposition tree of a function $\mathbf{F}(a, b, c)$ is displayed in Figure 7.1.

The decomposition strategy implemented in YADE is a depth first traversal of the tree, indicated by the numbered arrows in Figure 7.1. Function sets are passed downward the tree. The function sets are decomposed until only literal functions of single variables remain at the leaves of the tree. Then, these functions are passed upward the tree and composed to more complex decomposition results at the nodes of the tree. Each non-leaf node is traversed three times.

1. The first traversal of a node is initiated by the parent of the node (e.g. along the edges 1, 2, 4, 7, 8 and 10 in Figure 7.1). A function set is passed from the parent of the node. If the node is the root of the tree, the function set is the input to the decomposition algorithm (edge 0 in Figure 7.1). If the function depends on more than one variable, a decomposition operator and a partition of variables are chosen. The decomposition of the function set is computed and the free decomposition set is passed to the left child node of the decomposition tree. If the function depends on a single variable, a literal function is computed and returned to the parent of the node.

2. The second traversal of a node is initiated by the left child of the node (e.g. along the edges 3, 6 and 9 in Figure 7.1). A function that represents the result of the decomposition of this subtree is passed to the node. The bound decomposition set is computed and passed to the right child of the node.

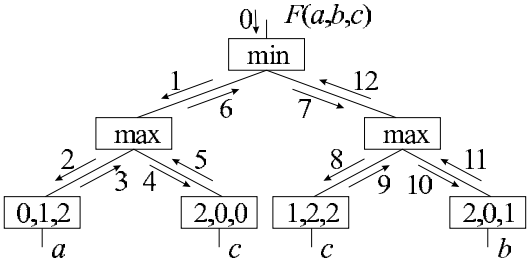

Figure 7.1: Decomposition tree of the function $\mathbf{F}(a, b, c)$.

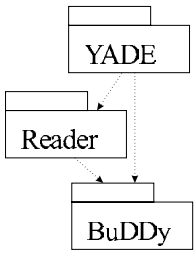

Figure 7.2: Package diagram of YADE.

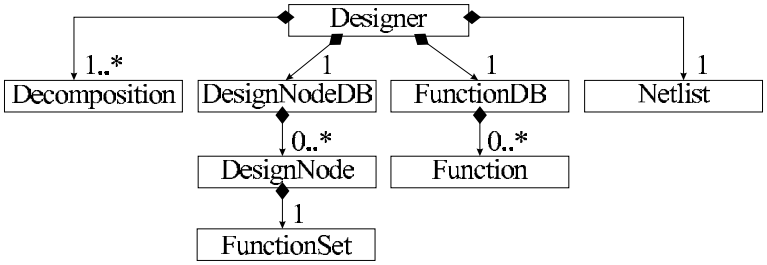

Figure 7.3: Class diagram of YADE.

3. The third traversal is initiated by return of the decomposition result from the right child of the node (e.g. along the edges 5, 11 and 12 in Figure 7.1). The decomposition result is computed by the composition of the two decomposition results from the children by the decomposition operator. The result is then passed to the parent of the node.

7.2 YADE Program Structure

7.2.1 Packages and Classes

The package structure of YADE is shown in Figure 7.2. The YADE package is based on the file reader by Mishchenko [23]. Both packages use the BuDDy BDD package as data structure [19].

The static structure of the YADE program is displayed in the class diagram in Figure 7.3. The top-level class of YADE is the class **Decomposer**. The only public method of the the class, the function DECOMPOSE($fvec$), accepts a set of function sets $fvec$ as input and returns the netlist of gates representing the decomposed functions. The decomposition algorithms are contained in a set of objects of class **Decomposition**. This class and its derived classes contain the decomposition algorithms whose theoretical foundation was developed in Chapter 5 and which are described in Section 6.3. The remaining members of **Decomposer** maintain the data during the decomposition process. The class **DesignNodeDB** contains a data base of function sets that have not yet been fully decomposed. Each function set is represented by an object of class **DesignNode**, where the actual function set is stored as an instance of the class **FunctionSet**. The member **FunctionDB** of **Designer** stores all the functions that have been fully decomposed as a set of objects of class **Function**. These functions are stored for possible reuse later in the decomposition process. The output of the decomposition is stored and returned as objects of class **Netlist**.

The YADE program contains algorithms for all the decomposition methods that were discussed in Chapter 5. Functions common to all decomposition algorithms are implemented in the base class **Decomposition**, classes with actual decomposition algorithms are derived from the class **Decomposition**. The decomposition methods and the function sets that are decomposed by the derived classes are shown in Table 7.1. The decomposition algorithms are documented in Section 6.3.

In Chapter 4 various classes of function sets were introduced. All classes that are applied by the decomposition algorithms must be represented by suitable classes. To have a common interface for all classes of function sets they are all derived from the base class **FunctionSet**, see Figure 7.4. The YADE program supports single functions (class **Function**), function intervals

Table 7.1: Classes that are derived from the class **Decomposition**.

Class Name	Description type	Function set
DecMonotone	monotone operators	function intervals
DecSimple	simple operators	MVL relations
MinaxDecomp	MM-decomposition	function intervals
MinaxWeak	weak MM-decomposition	function intervals
MinaxSep	set separation	function intervals
MultiDec	multi-decomposition	function intervals
DecModsum	modsum-decomposition	function intervals
DecCISFMinaxDec	MM-decomposition	CISF
DecCISFMinaxSep	set separation	CISF

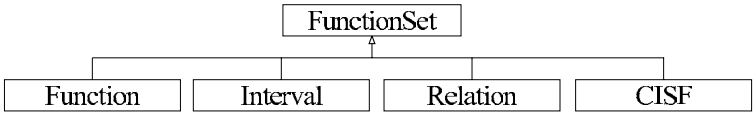

Figure 7.4: Hierarchy of classes representing function sets.

(class **Interval**), MVL relations (class **Relation**) and CISFs (class **CISF**). ISFs processed as a special case of function intervals.

7.2.2 Decomposition Process

The general sequence during the decomposition of function sets is displayed in the sequence diagram in Figure 7.5. The decomposition process is started by a call of the method DECOMPOSE($fvec$), where the vector $fvec$ contains all the function sets that will be decomposed. These function sets are inserted as nodes into the data base **designNodeDB**. Then, one node is taken from **designNodeDB** by the method GET(). Two cases are possible.

1. If the node is not simple, e.g. if the function set depends on more than one variable, the function set is decomposed by a call of the method FINDVARIABLES(). When the variables of the free and bound set have been found, the functions of the free decomposition set are computed by FREESET(). Then, two new nodes g and h are created for the free and the bound decomposition set (altough the bound decomposition set is not known yet). These new nodes are entered into the data base **designNodeDB** by the method PUT(g, h).

2. If the design node is simple, it depends on at most one variable and the corresponding function can be computed as a literal. The computed function is entered into the data base **FunctionDB** by the method PUT(), and the resulting gate is written to the **netlist** by the method PUTGATE(). Now, some of the unknown bound sets can be computed by

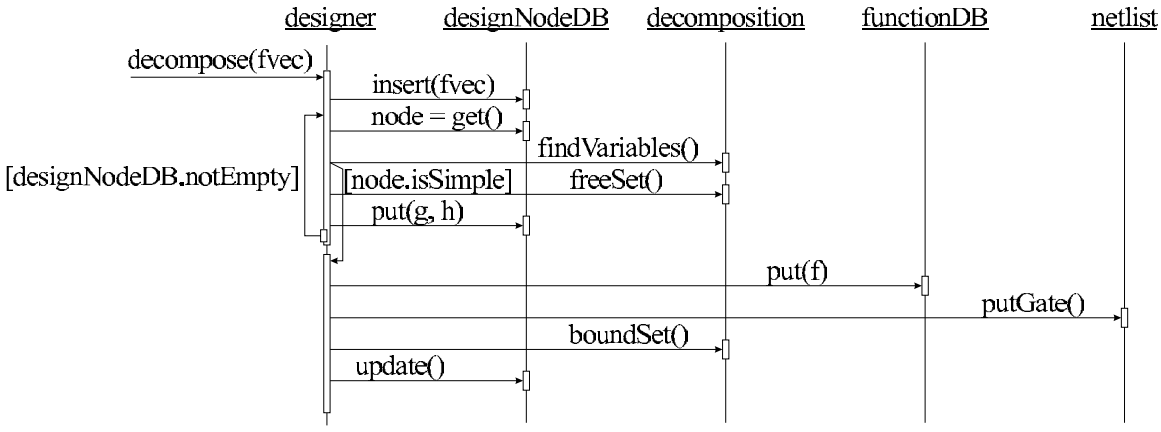

Figure 7.5: Sequence diagram of the decomposition process.

$R\langle(a,b),v\rangle$

a \ b	0	1	2
0	1	1	1,3
1	0,1,2,3	1	2
2	0	0	0

Figure 7.6: Input of YADE: Relation $R\langle(a,b),v\rangle$.

Table 7.2: Commands of the ML-format

Command	Description
.imvl	cardinality of input variables
.omvl	cardinality of output variables
.inputs	names of the input variables
.outputs	names of the output variables
.names	names of the variables in the following table
.mvl	cardinality of the variables in the following table
.end	end of the ML-file

a call of the method BOUNDSET() and the bound sets are updated in **designNodeDB** by the method UPDATE()

The process of getting nodes from **designNodeDB** is repeated until the data base is empty. Then, all nodes are decomposed and the resulting netlist is returned.

7.2.3 Data Structures

The input data of YADE is a file containing all relations that should be decomposed in the *ML-format*. An ML-file contains plain ASCII text that describes MVL relations by a set of tables.

Example 7.1. The relation $R\langle(a,b),v\rangle$ shown in Figure 7.6 can be described by the following ML file.

```
# This is an example ML file.
.imvl 3 3
.omvl 4
.inputs a b
.outputs v
.names a b v
.mvl 3 3 4
0 0 1
0 1 1
0 2 3
0 2 1
1 0 -
1 1 1
1 2 2
2 - 0
.end
```

Lines starting with a # are comments. Lines starting with a dot contain commands. The commands that must occur in an ML-file are listed in Table 7.2. The cardinalities of variables are denoted by space separated lists of integers. Names of variables are denoted by space separated lists of strings. The actual table data immediately follows the .mvl command. Each line describes elements of the relation. The line '0 1 1' in Example 7.1 expresses that $\langle(ab=00, v=1\rangle \in R\langle(a,b),v\rangle$. A dash in the table data denotes a don't care of the corresponding variable. An MVL file can contain more than one table. Each table consists of the commands .names, .mvl and the actual table data. Multiple tables can be used to describes multiple output variables.

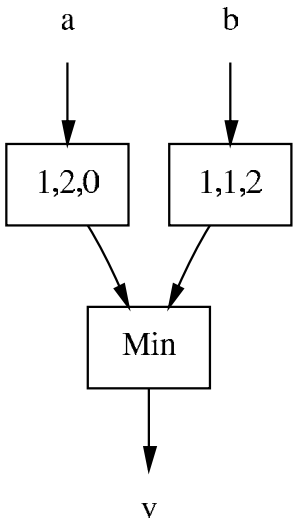

Figure 7.7: The graph drawn by dotty from the YADE output for the relation from Figure 7.6.

The output data of YADE is a file in *dotty-format* that describes the decomposed relations as a netlist of gates. A dotty-file is a plain ASCII file that describes graphs for viewing with the graph printing tool dotty [2]. The dotty package also contains a description of the dotty-format.

Example 7.2. The decomposition of the relation $R\langle(a,b),v\rangle$ from Figure 7.6 results in the graph shown in Figure 7.7.

The inputs of the network are drawn at the top of the graph. While the outputs are located at the bottom of the graph. Literal functions are labeled with the sequence of output values starting from input value '0'. All other gates are labeled with their respective functions.

In YADE ll function sets are stored internally as BEMDDs.

7.3 YADE Implementation

YADE is implemented in ANSI C++. The program was compiled using Microsoft Visual C++ 6.0 (MSVC). Only the standard libraries are referenced. Although other compilers have not been tested, the program should be easily portable to other ANSI C++ compilers, such as GNU C++. In fact, earlier MDD versions of YADE were developed using GNU C++.

The only issue when porting YADE is the use of the standard template library (STL). The STL from MSVC does not have hash tables. Therefore, the STL from Silicon Graphics (SGI) [33] was used for hash tables. The two versions of STL are distinguished by using different C++ namespaces for the respective version of the STL. Compilers with a complete STL, GNU C++ for instance, only need to reference one version of the STL.

Most of the computation time is spent on the BDD operations. Therefore, the performance of YADE is determined by the performance of these BDD operations. The size of tables and caches is crucial for the performance of BDD packages. In general the more memory is available the faster runs the program. This relationship is only valid as long as all memory used is physical memory. The current setup of YADE is optimized for 256 MB of physical RAM. This setup can be changed by overwriting constants in the file g_Cubes.h. Because most operations in the BDD packages are integer operations, the speed of a floating point unit is secondary. In its current version, YADE supports only a single processor. A BDD package that supports parallel processing and implements the interface of the BuDDy BDD package could speed up the program.

Table 7.3: Statistics output by YADE

Key	Description
	name of the ML-file
	decomposition setup
Time	time for decomposition in seconds
DFC	DFC of the netlist
ADFC	DFC as computed in [25]
Const	number of constants
Lit	number of Literals
Buf	number of buffers (identity functions)

7.4 YADE Package

7.4.1 Compact User Manual

The YADE package is called by the following command

```
yade file num
```

The first parameter (`file`) is the name of the ML-file that is decomposed the second parameter (`num`) is the number of a predefined decomposition setup (see Section 7.4.3). The output of YADE is a dotty-file with the same name as parameter `file`, except that the extension is changed to `.dt`.

YADE outputs also information about the decomposition process and statistics of the result netlist. The following statistic is output by YADE during the decomposition of the relation from Figure 7.6.

```
File reading time = 0 sec
Node 1: Min Rel  xg:0  xh:1
g dec 1->2
Node 2: Simple x:0
h dec 1->3
Node 3: Simple x:1
Starting
Verification...
Design Verified

test.ml DES100 Time 1 DFC 15  ADFC 11 Const 0 Lit 2 Buf 0 Min 1
Max 0 ModSum 0 TSum 0 TProd 0 Leq 0 Switch 0 Avg 0 Leq0 0 GeqN 0 IMax 0
```

The first line shows the time that was necessary to read the input and to convert it into the BEMDD data structure. The second line shows the decomposition type (min-decomposition of a relation) and the internal indexes of the variables for the free (xg) and bound (xh) sets of the decomposition. The third line indicates the internal numbers of the decomposition nodes for the original relation (node 1) and the free decomposition set (node 2). The fourth line shows that YADE designed a literal for node 2. Similarly the bound set (node 3) designed as a literal function. After the verification of the decomposition result, statistics of the network is output. The last two lines contain statistics of the netlist as shown in Table 7.3 The numbers of gates are output after the name of the respective gate.

7.4.2 Class Overview

This section briefly describes the classes of YADE that are necessary to apply and extend the decomposition algorithms. The classes are described in the format of the following template:

ExampleClass — short description of the class ExampleClass

Description

Purpose of the class

Fields

Fields contained in the class.

Methods

`void ExampleMethod()`
Description of the ExampleMethod()

Because decomposers are quite complex programs (YADE consists of more than 40 classes), the class reference here is simplified so that the overall structure of YADE and possible extensions are apparent.

Decomposition — base class for decomposition algorithms

Description

Purely virtual class that implements a common interface for all classes implementing decomposition algorithms. Algorithms common to all decomposition classes, such as variable selection is implemented here. A summary of the derived classes can be found in Table 7.1.

Fields

Methods

`bool GetDecomposition(FunctionSet F,VarSet& A, VarSet& B)`
Returns true if a decomposition of the function set `F` can be found. If a decomposition was found, the the free and bound set are stored in the variable sets `A` and `B` respectively.

`void FreeSet(FunctionSet F, VarSet A, VarSet B, FunctionSet& G)`
Computes the free decomposition set `G` of the function set `F` with respect to the variables of the free set `A` and the bound set `B`.

`void BoundSet(FunctionSet F, VarSet A, Function g, FunctionSet& H)`
Computes the bound decomposition set `H` of the function set `F` with respect to the variables of the free set `A` and and the decomposition function `g`.

Designer — decomposes relations

Description

Top-level class of YADE. The class performs a complete bi-decomposition of relations into a network of MVL gates.

Fields

`Decomposition* decomposition`
Array that stores all decomposition methods.

`DesignNodeDB designNodeDB`
Database that contains the design nodes that have not yet been completely decomposed.

`FunctionDB functionDB`
Database that contains the function that have been completely decomposed.

`NetList netlist`
Stores the network of gates that results from bi-decomposition.

Methods

`Design(BDD *relation, NetList& result)`
Performs a complete bi-decomposition of the BEMDDs `relations` into a netlist of gates that is stored in `result`

DesignerConfig — Configuration the decomposition process

Description

The global object `designerConfig` of this class controls the configuration of the decomposition process. Possible values of the fields are described in Section 7.4.3.

Fields

`enum VarElimMeth varElimMeth`
Selects the algorithm to eliminate inessential variables, see Section 6.9.

`int varTimeout`
time out value for the algorithm ELIMINATE-TIME-OUT

`enum SepVarMeth sepVar`
Selects the method of finding the variables for separation.

`enum DecVarSel decVarSel`
Selects the method of finding the variables for bi-decomposition.

`DecTypeVec decSeqFI`
Defines the sequence of decompositions that are tested for function intervals in algorithm FIND-OPERATOR (see Section 6.6)

`DecTypeVec decSeqREL`
Defines the sequence of decompositions that are tested for MVL relations in algorithm FIND-OPERATOR (see Section 6.6)

`DecTypeVec decSeqCISF`
Defines the sequence of decompositions that are tested for CISFs in algorithm FIND-OPERATOR (see Section 6.6)

`bits`
Bits that control the decompositions process, such as printing of debug information and reuse of functions.

Methods

`SetConfig(int n)`
Selects a predefined configuration that can subsequently be modified.

FunctionSet — base class for function sets

Description

Purely virtual class that implements a common interface for all classes that store function sets. Classes that are derived from `FunctionSet` are shown in Figure 7.4.

Fields

`VarSet support`
The variables of the support of the function set.

Methods

`bool Contains(Function f)`
Tests if the function `f` is a member of the function set.

`void MinimizeSupport()`
Minimizes the support of the function set by removal of inessential variables.

`double NumFunc()`
Returns the $\log_2$ of the number of functions that are contained in the function set.

NetList — stores a netlist of gates

Description

Database of a netlist of gates to store the result of the decomposition process.

Fields

`OperatorList m_op`
Array of the types of gates of the netlist.

`NetIDListList m_net`
Two-dimensional array of the connections between the gates.

Methods

`void PushGate(Operator op, NetID in0, NetID in1, NetID out)`
Stores a two-input gate in the netlist. The type of the gate is `op`, the inputs are connected to the nets `in0` and `in1`, and the output is connected to net `out`.

`void PrintDotty(ostream& os)`
Prints the netlist to the stream `os` in dotty-format.

7.4.3 Configuration and Control

The decomposition can be controlled by numerous options. It would too difficult to implement all options at the command line. Therefore, YADE can be configured by slight modifications to the source code and recompilation of the program. These modifications are limited to one location of the source code by a global control object `desingerConfig` of the class `DesignerConfig`. This object is configured before the decomposition. During the decomposition `designerConfig` is queried and the desired decomposition algorithms are selected. For a description of the configuration options. Below, the fields of the class `DesignerConfig` and their possible values are described.

`varElimMeth` Selects the algorithm for elimination of inessential variables, see Section 6.9. Possible values are

VarSimple	algorithm ELIMINATE-SIMPLE
VarMaxSet	algorithm ELIMINATE-MAX-SET
VarTimeOut	algorithm ELIMINATE-TIME-OUT

`varTimeout` Time out value in seconds for the algorithm ELIMINATE-TIME-OUT.

`sepVar` Selects the method of finding the variables for separation. Possible values are

SepFirst	separates the first two variables of the input
SepAll	tests all pairs of variables, selects the best pair

`decVarSel` Selects the method of finding the variables for bi-decomposition. Possible values are

FindVar	algorithm FIND-VARIABLE-SETS (Section 3.3)
FindBestVar	algorithm FIND-BEST-VARIABLE-SETS (Section 6.5)
FindMaxVarSet	algorithm FIND-MAX-VARIABLE-SETS (Section 6.5)

`decSeqFI` Defines the sequence of decompositions that are tested for function intervals in the algorithm FIND-OPERATOR (see Section 6.6). Possible values are shown in Table 7.4 together with the decomposition operator, algorithm and class (see Table 7.1).

`decSeqREL` Defines the sequence of decompositions that are tested for MVL relations in the algorithm FIND-OPERATOR (see Section 6.6) Possible values are shown in Table 7.4.

`decSeqCISF` Defines the sequence of decompositions that are tested for CISFs in the algorithm FIND-OPERATOR (see Section 6.6) Possible values are shown in Table 7.4.

`bits` Miscellaneous bits that control the decompositions process. Values can be combined by the C++ bitwise OR-operator. Possible bits are

REUSE_FUNC	enables the reuse of functions
PRINT_DEBUG	prints information during the decomposition

Table 7.4: Constants to select a decomposition method.

Constant	Operator	Algorithm	Class
Separator		Separator between groups	
Tsum	tsum	monotone decomposition	**DecMonotone**
TProd	tprod	monotone decomposition	**DecMonotone**
Avg$_i$	avg	monotone decomposition	**DecMonotone**
Leq	leq	simple decomposition	**DecSimple**
MinRel	min	simple decomposition	**DecSimple**
MaxRel	max	simple decomposition	**DecSimple**
Switch	switch	simple decomposition	**DecSimple**
Leq0	leq_0	simple decomposition	**DecSimple**
GeqM	geq_m	simple decomposition	**DecSimple**
Imax$_i$	imax	simple decomposition	**DecSimple**
MinFI	min	MM-decomposition of intervals	**MultiDec**
MaxFi	max	MM-decomposition of intervals	**MultiDec**
MinWeak	min	weak bi-decomposition	**MinaxWeak**
MaxWeak	max	weak bi-decomposition	**MinaxWeak**
MinDom	min-max	multi-decomposition	**MultiDec**
MaxDom	max-min	multi-decomposition	**MultiDec**
MinSep	min-max	set separation	**MinaxSep**
MaxSep	max-min	set separation	**MinaxSep**
Modsum	$\oplus_m$	modsum-decomposition	**DecModsum**
MinCISF	min	CISF bi-decomposition	**DecCISFMinaxDec**
MaxCISF	max	CISF bi-decomposition	**DecCISFMinaxDec**
MinSepCISF	min	CISF set separation	**DecCISFMinaxSep**
MaxSepCISF	max	CISF set separation	**DecCISFMinaxSep**

Chapter 8

Experimental Results

8.1 Overview

This section displays the results of decomposition of test functions by various setups of the YADE decomposer. The results are also compared to the results of other decomposers. First, the test functions, the YADE configurations and the quality measures are shown. Then, the results of the decomposition of the test functions are displayed.

8.2 Description of Test Functions

The practical applicability of the algorithms implemented in the YADE decomposer was tested by decomposition of test functions which were take from the POLO web page [29] and the UCI data base [44]. The functions were selected from a variety of machine learning application. Some test functions have been extracted from medical or engineering data bases (DB), others represent optimal strategies of board games or test cases from artificial intelligence applications (see Table 8.1). The table displays the number of inputs N_{in}, the maximum cardinality M_{in} of the inputs, the number B_{in} of Boolean variables to represent the input variables in a BEMDD, the cardinality M_{out} of the output, the number N_c of cubes, the ratio R_c of cares (see (4.3)), the type of function set and, if possible, a description of the data. Most functions have a single output variable. The only exceptions are flare1 and flare2 with three output variables each.

The largest test function has an input equivalent to 112 Boolean variables (lung-cancer). Some functions are completely specified (e.g. car), while others show an extremely low percentage of cares (e.g. audiology). The test functions cover various types of function sets (completely specified functions, ISFs, function intervals and relations).

8.3 Reference Design and Test Setup

This section describes the parameters that are applied to evaluate various decomposition strategies. To have a common basis in this comparison, a *YADE reference setup* is used. The decomposition strategies are compared to the reference setup. The results of the decomposition of the test functions by the YADE reference setup are also shown in this section.

The quality of different decomposition strategies can be compared by several quality measures. The most important measure is the complexity of the network gates created by the decomposition process. The complexity of a Boolean network can be estimated from the number of gates. However, in networks with MVL gates, the input cardinality of the gates must be taken into account. In section 2.3.1 the DFC was defined as a complexity measure of MVL functions. It follows from (2.12) that the DFC of a literal is the cardinality of the variable that feeds the literal. The DFC of a two-input gate is the product of the cardinality of the inputs to the gate. Including the DFC, the following parameters are applied to characterize decomposition network.

DFC The DFC of a network of gates is defined as the sum of the DFCs of the gates of the network, including the literals.

Table 8.1: Description of the test functions.

Name	N_{in}	M_{in}	B_{in}	M_{out}	N_c	R_c	Type	Description
audiology	69	6	80	24	200	1.31E−16	ISF	
balance	4	5	12	5	625	1.00	Func	
breastc	9	10	36	2	699	5.80E−07	ISF	Breast cancer DB
bridges1	9	4	16	7	108	8.01E−03	Rel	
bridges2	10	4	18	7	108	5.89E−03	Rel	
car	6	4	12	4	1728	1.00	Func	
chess1	6	8	16	18	28056	0.43	ISF	Chess end game
chess2	36	3	37	2	3196	3.10E−08	ISF	Chess end game
cloud	6	9	21	2	108	4.52E−04	ISF	
employ1	9	5	15	4	9600	1.00	Func	
employ2	7	5	17	4	18000	1.00	Func	
flag	28	14	57	6	194	1.73E−14	ISF	
flare1	10	7	17	5	969	3.12E−03	Rel	Solar flare DB
flare2	10	7	17	5	3198	5.02E−03	Rel	Solar flare DB
hayes	4	5	11	3	132	0.17	FI	
iris	4	12	13	3	150	0.03	FI	
lensesmv	4	3	5	3	24	1.00	Func	Contact lenses
lung-cancer	56	4	112	3	32	1.40E−16	ISF	
monks1te	6	4	10	2	432	1.00	Func	Monk's problems
monks1tr	6	4	10	2	124	0.29	ISF	Monk's problems
monks2te	6	4	10	2	432	1.00	ISF	Monk's problems
monks2tr	6	4	10	2	169	0.39	ISF	
monks3te	6	4	10	2	432	1.00	Func	Monk's problems
monks3tr	6	4	10	2	122	0.28	ISF	Monk's Problems
mushroom	22	12	54	2	8124	7.98E−11	ISF	
nursery	8	5	16	5	12960	1.00	Func	
post-operative	8	4	14	3	90	0.02	FI	
programm	12	4	24	5	20000	6.68E−03	ISF	
sensory	11	6	19	11	576	2.60E−03	ISF	
ships	4	5	9	4	34	0.17	ISF	
shuttlem	6	4	8	2	15	0.99	ISF	Space shuttle
sleep	9	21	31	5	62	8.30E−05	ISF	
sponge	44	9	86	3	76	1.18E−16	ISF	
tic-tac-toe	9	3	18	2	958	0.05	ISF	Game strategy
trains	32	10	51	2	10	0.00	ISF	Trains data set
zoo	16	9	19	7	101	2.00E−04	ISF	Zoo database

Gate2 The number of two-input gates of the network.

Literal The number of literals of the network.

Reuse The number of times that a function set could be realized by reuse a function that has already been decomposed.

Level The number of gates (including literals) along the longest path from any input any output.

Time The total computation time to compute the bi-decomposition network including file reading and internal verification of the results on a 900 MHz Pentium III computer with 256 MB RAM under Windows NT.

The common basis for the comparison of different decomposition strategies is the YADE reference setup DES100. All other test cases follow the same template, which is shown below for the reference setup.

DES100 — YADE reference setup

YADE Setup

Setup	Parameter	Value
DES100	varElimMeth	VarMaxSet
	sepVar	SepFirst
	decVarSel	FindVar
	decSeqFI	MaxFI MinFI ♦ MaxSep
	decSeqREL	MaxRel MinRel ♦ MaxSep

Results

The decomposition results for the test functions from Table 8.1 are shown in Table 8.2.

Comments

The smallest network consists of only a single literal (trains), while the largest network contains more than 15000 gates (programm). The time needed for the decomposition is less than 10 seconds with two exceptions (chess1 and programm) where approximately 5 minutes were needed.

It can be seen that the size of the network depends very much on the ratio of cares in the function. The test functions audiology and lung-cancer have 69 and 56 input variables respectively, while their networks consists of less than 150 gates. The test functions programm and chess have 12 and 6 input variables respectively, while their networks consist of more than 10000 gates each. This fact can be explained by the very small ratio of cares $R_c < 2\text{E}{-}16$ for audiology and lung-cancer, while programm and chess1 have a ratio of cares of 6.7E−3 and 0.43.

The network of the test function train does not have any two-input gates because it only has one inessential variable. The test function is included in the test functions because it is instructive to see that other variable elimination strategies do not find this result.

The fist section of the template contains the name of the setup (**DES100**) and a short description. The second section (**YADE Setup**) shows the values of the global configuration object `designerConfig` for this setup, see Section 7.4.3. The sequence of decompositions (see algorithm SELECT-OPERATOR in Section 6.6) is shown for the fields `decSeqFI`, `decSeqREL` and `decSeqCISF`. The separator between the groups is denoted by the symbol ♦. Some fields of the object `designerConfig` may be omitted if their values have no influence in this particular setup. The value of `decSeqCISF` for instance, is not shown because CISFs do not appear during the decomposition as modsum-decomposition is not applied. The third section (**Results**) shows the decomposition results of this setup for the test functions. The last section (**Comments**) comments the results.

To compare the quality of the results obtained by different decomposition strategies, some kind of average over various test functions is needed. Because the size of test functions differ by orders of magnitude, an arithmetic mean over all test functions would be dominated by the largest function. Therefore, the parameters described above are normalized to the results of the

Table 8.2: Decomposition results of the YADE reference setup DES100.

Name	DFC	Gate2	Literal	Reuse	Level	Time
audiology	32477	136	91	46	13	7
balance	1546	150	48	103	12	1
breastc	712	61	49	13	10	1
bridges1	1706	48	35	14	10	0
bridges2	2648	78	51	28	12	0
car	724	53	33	21	10	0
chess1	2682211	10653	3378	7276	26	283
chess2	271	86	41	46	9	4
cloud	421	43	32	12	9	0
employ1	428	42	30	13	11	1
employ2	662	38	25	14	10	2
flag	4785	140	87	54	12	3
flare1	546	70	36	37	8	0
flare2	2347	142	56	89	12	1
hayes	96	10	8	3	6	0
iris	148	9	10	0	6	0
lensesmv	88	9	8	2	5	0
lung-cancer	208	26	20	7	8	1
monks1te	46	6	7	0	5	0
monks1tr	46	6	7	0	5	0
monks2te	46	6	7	0	5	0
monks2tr	468	95	27	69	11	1
monks3te	23	3	4	0	3	0
monks3tr	172	28	18	11	8	0
mushroom	113	9	10	0	5	3
nursery	3546	140	40	101	13	1
post-operative	356	52	22	31	9	0
programm	291419	15615	444	15172	28	295
sensory	55885	785	261	525	16	2
ships	232	17	16	2	7	0
shuttlem	57	12	8	5	6	0
sleep	356	17	16	2	6	0
sponge	49	4	5	0	4	1
tic-tac-toe	851	193	41	153	14	1
trains	10	0	1	0	1	0
zoo	147	6	6	1	5	0

Table 8.3: Normalized average results of bi-decomposition with respect to monotone operators.

Setup	DFC	Gate2	Literal	Reuse	Level	Time
MON1	1.15	1.17	1.14	1.31	0.99	1.72
MON2	1.09	1.29	1.24	1.53	1.03	1.93
MON3	0.96	0.99	1.03	0.94	0.98	1.05
MON4	1.08	1.31	1.25	1.59	1.03	1.6

reference design DES100. Then, the average is computed over these normalized parameters. Let $D_{ri}, i = 1 \ldots n$ be the DFCs for n test functions of the YADE reference setup, and D_{ti} be the DFCs for the same test functions of a YADE test setup. The average normalized DFC D_n of the test setup is

$$D_n = \frac{1}{n} \sum_{i=1}^{n} \frac{D_{ri}}{D_{ti}}. \tag{8.1}$$

Other normalized parameters, such as the number of gates, the number of levels, or the decomposition time are computed analogous.

8.4 Selection of Operators

The performance of three different algorithms for different classes of algorithms was tested in addition to MM-decomposition: the relaxation algorithm for monotone operators (Section 6.3.3), the value-removal algorithm for simple operators (Section 6.3.4) and the modsum decomposition algorithm (Section 6.3.6). For each algorithm, different setups are tested that either prefer the MM-operators or the new decomposition operators.

MON — Monotone decomposition

YADE Setup

Setup	Parameter	Value
MON1–4	varElimMeth	VarMaxSet
	sepVar	SepFirst
	decVarSel	FindVar
MON1	decSeqFI	MaxFI MinFI Tprod Tsum Avg ♦ MaxSep
	decSeqREL	MaxRel MinRel Tprod Tsum Avg ♦ MaxSep
MON2	decSeqFI	Tprod Tsum Avg MaxFI MinFI ♦ MaxSep
	decSeqREL	Tprod Tsum Avg MaxRel MinRel ♦ MaxSep
MON3	decSeqFI	MaxFI MinFI ♦ Tprod Tsum Avg ♦ MaxSep
	decSeqREL	MaxRel MinRel ♦ Tprod Tsum Avg ♦ MaxSep
MON4	decSeqFI	Tprod Tsum Avg ♦ MaxFI MinFI ♦ MaxSep
	decSeqREL	Tprod Tsum Avg ♦ MaxRel MinRel ♦ MaxSep

Results

The normalized average results of the four decomposition strategies are shown in Table 8.3.

Comments

Compared to the reference design DES100, the three monotone operators tprod, tsum and avg were added to the sequence of decompositions. The setups MON1–MON4 differ in the sequence and the grouping of the operators. The setups MON1 and MON2 select the best operator among the monotone and the MM-operators. If decompositions of equal quality can be found, MON1 prefers the MM-operators, while MON2 prefers the monotone operators. The setup MON3 selects monotone operators only if there is no MM-decomposition. The setup MON4 selects an MM-operator only if no other monotone decomposition exists.

The table shows that the DFC increases for the MON1, MON2 and MON4, while the DFC decreases for MON3. This implies that MM-decomposition can be improved if a monotone

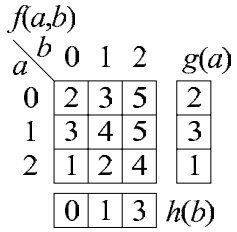

Figure 8.1: Truncated-sum-decomposition of function $f(a,b)$ into two functions $g(a)$ and $h(b)$ with smaller cardinality.

decomposition is selected only if no MM-decomposition exists. All other strategies increase the complexity of the decomposition result.

For the setups MON1, MON2 and MON4 the increase in DFC can be explained by the approximation of the decomposition sets. For the min- and max-decomposition the decomposition algorithm computes the maximal decomposition sets. For all other decomposition algorithms only an approximation of the decomposition set can be computed, which implies that there is less freedom for optimization in the decomposition functions.

As for the DFC, the setup MON3 reduces the number of gates, while the other strategies increase the number of gates. However, for the monotone operators, the reduction in the number of gates is not as significant as the reduction of the DFC. This effect is caused by the structure of the tsum and tprod operators. These operators can produce large output values from relatively small input values. The function $f(a,b)$ (Figure 8.1) can be composed by $f(a,b) = \text{tsum}(g(a), h(b))$ where the cardinality of the functions $g(a)$ and $h(b)$ ($m_g = 3$, $m_h = 5$) is smaller than the cardinality of the function $f(a,b)$ ($m_f = 7$). This is not possible for min- and max-gates.

In general, large values of an operator function cause small function values in the decomposition functions which causes a small DFC of the operator functions.

An interesting aspect is the reduction in the number of levels of the decomposed network for the strategies MON1 and MON3. While the reduction for strategy MON3 can be explained by the smaller number of gates, strategy MON1 produces a larger number of gates while still producing a smaller number of levels. This effect is caused by the additional possibilities for decompositions provided by the additional decomposition operators.

SIMPLE — Simple decomposition

YADE Setup

Setup	Parameter	Value
SIMPLE1–4	varElimMeth	VarMaxSet
	sepVar	SepFirst
	decVarSel	FindVar
SIMPLE1	decSeqFI	MaxFI MinFI GeqM Leq0 Imax ♦ MaxSep
	decSeqREL	MaxRel MinRel GeqM Leq0 Imax ♦ MaxSep
SIMPLE2	decSeqFI	GeqM Leq0 Imax MaxFI MinFI ♦ MaxSep
	decSeqREL	GeqM Leq0 Imax MaxRel MinRel ♦ MaxSep
SIMPLE3	decSeqFI	MaxFI MinFI ♦ GeqM Leq0 Imax ♦ MaxSep
	decSeqREL	MaxRel MinRel ♦ GeqM Leq0 Imax ♦ MaxSep
SIMPLE4	decSeqFI	GeqM Leq0 Imax ♦ MaxFI MinFI ♦ MaxSep
	decSeqREL	GeqM Leq0 Imax ♦ MaxRel MinRel ♦ MaxSep

Results

The normalized average results of the four decomposition strategies are shown in Table 8.4.

Comments

Compared to the reference design DES100, the three simple operators geq_m, leq_0 and imax were added to the sequence of decompositions. Similar to the monotone decomposition four

Table 8.4: Normalized average results of bi-decomposition with respect to simple operators.

Setup	DFC	Gate2	Literal	Reuse	Level	Time
SIMPLE1	0.95	1.00	1.13	0.85	0.99	1.93
SIMPLE2	0.98	1.06	1.24	0.89	1.05	2.12
SIMPLE3	0.97	0.96	1.08	0.83	0.98	1.06
SIMPLE4	0.98	1.01	1.21	0.76	1.04	2.02

strategies were applied to select the decomposition operator. The setups SIMPLE1–SIMPLE4 differ in the sequence and the grouping of the operators. The setups SIMPLE1 and SIMPLE2 select the best operator among the simple and the MM-operators. If decompositions of equal quality can be found, SIMPLE1 prefers the MM-operators, while SIMPLE2 prefers the simple operators. The setup SIMPLE3 selects simple operators only if there is no MM-decomposition. The setup SIMPLE4 selects an MM-operator only if no other simple decomposition exists.

In comparison with the reference design DES100 the DFC is reduced by all four operator selection strategies. The best strategy is SIMPLE1 where the best of all operators is selected and the MM-operators are preferred to the other operators. The better performance of simple operators compared to monotone operators can be explained by the fact that the value-removal algorithm can decompose relations, while the relaxation algorithm for monotone operators can only decompose function intervals. A relation must be converted to a function interval before it can be decomposed by a monotone operator. This conversion loses some freedom for optimization.

Similar to monotone decomposition, the reduction in the number of gates is not as significant as the reduction in the DFC. Similarly to the tsum-operator, the geq_m-operator can decompose a function into decomposition functions of smaller cardinality, which reduces the DFC of the decomposed network.

The number of levels is reduced by the setups SIMPLE1 and SIMPLE3 which prefer MM-operators, but consider simple operators in addition to the MM-operators. This can avoid separation if no MM-decomposition is possible. However, the setups SIMPLE2 and SIMPLE4 increase the number of levels, which is explained by the increased number of gates for these setups.

MODSUM — Modsum decomposition

YADE Setup

Setup	Parameter	Value
MODSUM1–4	varElimMeth	VarMaxSet
	sepVar	SepFirst
	decVarSel	FindVar
	decSeqCISF	MaxCISF MinCISF ♦ MaxSepCISF
MODSUM1	decSeqFI	MaxFI MinFI Modsum ♦ MaxSep
	decSeqREL	MaxRel MinRel Modsum ♦ MaxSep
MODSUM2	decSeqFI	Modsum MaxFI MinFI ♦ MaxSep
	decSeqREL	Modsum MaxRel MinRel ♦ MaxSep
MODSUM3	decSeqFI	MaxFI MinFI ♦ Modsum ♦ MaxSep
	decSeqREL	MaxRel MinRel ♦ Modsum ♦ MaxSep
MODSUM4	decSeqFI	Modsum ♦ MaxFI MinFI ♦ MaxSep
	decSeqREL	Modsum ♦ MaxRel MinRel ♦ MaxSep

Results

The normalized average results of the four decomposition strategies are shown in Table 8.5.

Comments

Compared to the reference design DES100, the modsum-operator was added to the sequence of decompositions. The CISFs that result from modsum-decomposition are decomposed by

Table 8.5: Normalized average results of bi-decomposition with respect to the modsum-operator.

Setup	DFC	Gate2	Literal	Reuse	Level	Time
MODSUM1	1.21	1.22	1.21	1.16	1.01	2.67
MODSUM2	1.29	1.31	1.37	1.15	1.03	2.79
MODSUM3	0.98	0.98	1.03	0.91	0.99	1.44
MODSUM4	1.44	1.42	1.39	1.49	1.06	3.27

the MM-decomposition and set separation of CISFs. Similar to the monotone decomposition four strategies were applied to select the decomposition operator. The setups MODSUM1–MODSUM4 differ in the sequence and the grouping of the operators. The setups MODSUM1 and MODSUM2 select the best operator among the modsum and the MM-operators. If decompositions of equal quality can be found, MODSUM1 prefers the MM-operators, while MODSUM2 prefers the modsum-operator. The setup MODSUM3 selects the modsum-operator only if there is no MM-decomposition. The setup MODSUM4 selects an MM-operator only if no modsum-decomposition exists.

The strategy MODSUM3 decreases the DFC of the circuit. All other strategies show an significant increase in DFC. The relatively poor performance of these the modsum-decomposition compared to monotone and simple decomposition is caused by the fact that modsum-decomposition can only decompose ISFs. For function intervals the algorithm does not find all decompositions. Relations must be converted to function intervals which may lose decomposable functions during the conversion process. Still the setup MODSUM3 shows that the DFC of the decomposed network can be decreased if modsum-decomposition is chosen if no MM-decomposition is possible.

Because the modsum-operator does not offer the property that the decomposition functions have a smaller DFC than the original function, the change in the number of gates is approximately the same as the change in the DFC.

There is a reduction in the number of level only for strategy MODSUM3 where the number of gates is smaller. The additional possibilities to find good decompositions are overwritten by the larger number of gates for the other decomposition strategies.

8.5 Separation Strategies

Several strategies have been proposed to simplify non-decomposable function sets. The reference design DES100 applies max-min set separation to non-decomposable functions. The first two variables of the support are selected for the separation variables. For some non-decomposable functions, weak bi-decomposition can simplify these function sets. Multi-decomposition is another method that can simplify all function sets. A comparison of these separation methods is shown in the test case below.

SEP — Separation methods

YADE Setup

Setup	Parameter	Value
SEP1–4	varElimMeth	VarMaxSet
	sepVar	SepFirst
	decVarSel	FindVar
SEP1	decSeqFI	MaxFI MinFI ♦ MaxWeak MinWeak ♦ MaxSep
	decSeqREL	MaxRel MinRel ♦ MaxWeak MinWeak ♦ MaxSep
SEP2	decSeqFI	MaxFI MinFI ♦ MaxDom
	decSeqREL	MaxRel MinRel ♦ MaxDom
SEP3	decSeqFI	MaxFI MinFI ♦ MaxDom MinDom
	decSeqREL	MaxRel MinRel ♦ MaxDom MinDom
SEP4	decSeqFI	MaxFI MinFI ♦ MaxSep MinSep
	decSeqREL	MaxRel MinRel ♦ MaxSep MinSep

Table 8.6: Normalized average results of the separation methods.

Setup	DFC	Gate2	Literal	Reuse	Level	Time
SEP1	1.14	1.12	1.09	1.17	1.37	0.77
SEP2	1.05	1.01	0.98	1.02	1.05	1.05
SEP3	1.05	1.00	1.01	0.98	1.02	1.26
SEP4	1.09	1.06	1.07	1.08	1.09	1.17

Results

The normalized average results of the four decomposition strategies are shown in Table 8.6.

Comments

The strategy SEP1 applies weak decomposition to simplify non-decomposable function sets. However, weak decomposition is not always possible. Therefore, weak MM-decomposition is applied in addition to the max-min set separation of the reference design DES100. All variables were tested to find a weak decomposition. The weak MM-decomposition with the greatest number of functions in the free set was selected as the best weak bi-decomposition.

Strategy SEP2 applies multi-decomposition with the DOM graph coloring algorithm. Max-min-multi-decomposition with respect to first two variables of the support was applied to all non-decomposable function sets.

The strategy SEP3 applies multi-decomposition with respect to the first two variables of the support to non-decomposable function sets. The max-min- or min-max-multi-decomposition with the smallest number of colors is selected as the best multi-decomposition.

The strategy SEP4 applies set separation with respect to the first two variables of the input to non-decomposable function sets. The max-min or min-max set separation with the greatest number of functions in the free set is selected as the best separation.

The results (Table 8.6) show that none of the separation strategies can reduce the DFC or the number of two-input gates. Only the strategy SEP2 shows a slight reduction in the number of literals.

The large number of gates for strategy SEP1 is caused by many subsequent weak decompositions before a bi-decomposable function set is found. Both set separation and multi-decomposition find a bi-decomposable function set after at most m_a steps, where m_a is the cardinality of the decomposition variable. There is no such guarantee for weak decomposition. Although the strategy SEP1 produces more gates, the decomposition time is significantly smaller because weak decompositions can be computed much faster than separations.

The multi-decomposition algorithms SEP2 and SEP3 perform poor compared to set separation because multi-decomposition reduces the number of non-decomposable functions at the cost of the size of the function sets. Much freedom for optimization of the decomposition functions is destroyed in order to reduce the number of non-decomposable functions during decomposition. As the results show the the size of the decomposition function sets has great influence on the complexity of the decomposed network.

It may surprise that the complexity of circuits increases if the set separation for min-max-separation is considered in addition to max-min set separation in the setup SEP4. However, alternating between max-min and min-max-separations may increase the number of separations before a bi-decomposable function is found.

So far only separations with respect to the first two variables of the support of the function set were considered. It seems likely that better results can be obtained if all pairs of variables are tested the best separation (largest function set or smallest number of colors) is selected. This hypothesis is tested by the following test case.

SEPALL — Separation methods

YADE Setup

Table 8.7: Normalized average results of the separation methods with respect to all pairs of variables.

Setup	DFC	Gate2	Literal	Reuse	Level	Time
SEPALL1	1.00	1.00	1.01	1.02	1.01	0.81
SEPALL2	1.04	1.00	0.99	0.99	1.04	1.66
SEPALL3	1.05	1.03	1.04	1.07	1.08	1.07
SEPALL4	1.03	0.98	1.00	0.95	1.01	2.49

Setup	Parameter	Value
SEPALL1–4	varElimMeth	VarMaxSet
	sepVar	SepAll
	decVarSel	FindVar
SEPALL1	decSeqFI	MaxFI MinFI ♦ MaxSep
	decSeqREL	MaxRel MinRel ♦ MaxSep
SEPALL2	decSeqFI	MaxFI MinFI ♦ MaxDom
	decSeqREL	MaxRel MinRel ♦ MaxDom
SEPALL3	decSeqFI	MaxFI MinFI ♦ MaxSep MinSep
	decSeqREL	MaxRel MinRel ♦ MaxSep MinSep
SEPALL4	decSeqFI	MaxFI MinFI ♦ MaxDom MinDom
	decSeqREL	MaxRel MinRel ♦ MaxDom MinDom

Results

The normalized average results of the four decomposition strategies are shown in Table 8.7.

Comments

The setups SEPALL1–SEPALL4 set the field `sepVar` to SepAll instead of SepFirst. This causes a test of all pairs of variables for the separation. The separation with the largest number of functions in the free decomposition set is selected for set separation. The multi-decomposition with the smallest number of colors is selected among all multi-decompositions. The setups SEPALL1 and SEPALL2 apply max-min set separation and max-min multi-decomposition respectively. The setups SEPALL3 and SEPALL4 also includes min-max set separation and min-max multi-decomposition in the search process.

None of the decomposition strategies could improve the complexity compared to the reference design. Subsequent set separations with respect to different pairs of variables may increase the number of separations before a bi-decomposable function set is found.

Chapter 9

Conclusion and Further Work

9.1 Conclusion

This dissertation introduces the concept of MVL function sets to solve several bi-decomposition problems in multi-valued logic. Function sets are a generalization of MVL relations. Processing of general function sets on computers needs very much memory. Therefore, some subclasses of function sets that occur during bi-decomposition were identified. Relations between these subclasses of function sets were shown and taxonomy of function classes was established. MVL function sets were related to Boolean function sets. MVL ISFs, function intervals and relations are MVL extensions of Boolean ISFs. Boolean function lattices are generalized to MVL function lattices. CISFs were introduced as a novel class of function sets for both the Boolean and the MVL case respectively.

Function sets were applied to compute the free and bound decomposition sets of bi-decompositions. Decomposition algorithms were developed for several classes of operators. Bi-decomposition of function intervals for monotone operators can be computed by iteration algorithms. The decomposition sets are function lattices. To simplify the storage of the decomposition sets, an approximation of these lattices by MVL relations was shown.

Simple operators are defined by a sequence of special if-then-else constructs. Bi-decompositions of relations for simple operators can be computed by successive substitutions of known values of the decomposition functions. The decomposition set was also approximated by MVL relations. The MM-operators are special cases of simple operators. It was shown that in this case the decomposition set is a relation and no approximation of the decomposition set is needed.

A closed solution was presented for bi-decomposition of function intervals with respect to the MM-operators. Operations from the MVL differential calculus were applied to generalize corresponding results for Boolean ISFs. The relation between free and bound decomposition sets was shown. These results were applied to develop efficient bi-decomposition algorithms for function intervals.

The modsum-decomposition problem was solved exactly for ISFs and by approximation for relations. The decomposition sets of modsum-decomposition are CISFs. Decomposition algorithms for CISF were also presented. The algorithms presented are equally applicable to any decomposition operator that has the latin square property.

There are MVL functions that are not decomposable. It was proposed to simplify these functions by separation. The free set of set separation is a superset of the original function set and therefore simpler to decompose. The bound set of set separation is always bi-decomposable. All functions are separable. The max-set separation algorithm is a heuristical algorithm that tries to make the free set as large as possible. The multi-decomposition algorithm applies graph-coloring to separate a function set into a minimal number of bi-decomposable function sets.

The decomposition algorithms were implemented in the decomposition system YADE, which is an extensible and modular platform for the test of a variety of function sets and their application in bi-decomposition algorithms. The underlying data structure are BEMDDs the fastest known data structure for bi-decomposition algorithms. YADE can be configured to decompose MVL functions by different decomposition strategies.

Several test functions from machine learning applications have been decomposed with respect to different decomposition operators, variable selection methods and variable elimination

methods. The basic decomposition strategy applies bi-decomposition and separation for the MM-operators. Even large test functions could be decomposed into a network of MM-gates within a few minutes on a Pentium PC. The inclusion of other decomposition operators reduced the the number of gates as well as the DFC of the decomposed network of gates.

9.2 Further Work

The decomposition algorithms presented in this dissertation can be applied to machine learning problems. In machine learning further work is needed to explore the dependencies between decomposition strategies and the learning error. Also, if data is noisy, decomposition algorithms must be able to process noisy data. In this case each decomposition is assigned a quality measure. For the MM-decomposition of function intervals for instance this quality measure can be the number of minterms, where the decomposition criteria is satisfied.

In logic synthesis of MVL circuits, it is necessary to identify the basic gates of the chosen circuit technology. The algorithms of this work can then be applied or extended to produce networks that can be realized directly as gates of this technology. Further work is needed to develop algorithms for the synthesis of sequential logic.

The decomposition of test functions shows that the decomposition algorithms presented are not optimal. Different decomposition strategies differ very much in terms of number of gates or complexity. There is no single best strategy. Improved algorithms for the selection of decomposition variables or operators could combine the benefits of different strategies. To guide the selection strategies, a complexity measure can be defined for function sets. In this work the number of member functions was applied to measure the complexity of a function set. More sophisticated complexity measures could include some information about the complexity of the individual member function of the set. One such measure is the entropy of a function.

The theory of function sets has its origins in the solution of Boolean differential equations. It can be expected that the solution of MVL differential equations will identify many other classes of function sets. The author of this work believes that the extension Boolean differential equations to multi-valued logic will form the theoretical basis for many problems in combinatorial and sequential logic synthesis and machine learning.

Bibliography

[1] R. Ashenhurst. The decomposition of switching functions. In *International Symposium on the Theory of Switching Functions*, pages 74–116, April 1957.

[2] AT&T Bell Labs, http://www.research.att.com/sw/tools/graphviz/. *Graphviz*.

[3] D. Bochmann, F. Dresig, and B. Steinbach. A new decomposition method for multilevel circuit design. In *European Conference on Design Automation*, pages 374–377, Amsterdam, 1991.

[4] D. Bochmann and C. Posthoff. *Binäre dynamische Systeme*. Oldenbourg Verlag, München, 1981.

[5] D. Bochmann, C. Posthoff, and Haubold. *Diskrete Mathematik*.

[6] D. Bochmann and B. Steinbach. *Logikentwurf mit XBOOLE*. Verlag Technik, Berlin, 1991.

[7] Böhlau and Bochmann. *Gruppierung*.

[8] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1992.

[10] H. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, N.J., 1962.

[11] S. Han, Y. Choi, and H. Kim. A 4-digit cmos quaternary to analog converter with current switch and neuron mos down-literal circuit. In *31th IEEE International Symposium on Multiple-Valued Logic*, pages 67–71, Warsaw, May 2001.

[12] M. Inaba, K. Tanno, and O. Ishizuka. Realization of nmax and nmin functions with multi-valued voltage comparators. In *31th IEEE International Symposium on Multiple-Valued Logic*, pages 27–32, Warsaw, May 2001.

[13] M. Kameyama. Innovation of intelligent integrated systems architecture –future challenge–. In *Extended Abstracts of the 8th International Workshop on Post-Binary Ultra-Large-Scale Integration Systems*, pages 1–4, May 1999.

[14] G. Kempe and C. Lang. Efficient representation of boolean functions by three- and four-function decomposition. In *3rd International Workshop on Boolean Problems*, pages 39–46, Freiberg, September 1998.

[15] Y-T. Lai, K-R. Pan, and M. Pedram. Obdd-based functional decomposition: algorithms and implementation. *IEEE Trans. on Computer-Aided Design*, 15(8), August 1996.

[16] C. Lang. *Mehrfachnutzung von Schaltungsteilen bei der dekompositorischen Synthese von kombinatorischen Schaltungen*. diploma thesis, Technical University Chemnitz, Chemnitz, 1995.

[17] C. Lang and B. Steinbach. Decomposition of multi-valued functions into min- and max-gates. In *31th IEEE International Symposium on Multiple-Valued Logic*, pages 173–178, Warsaw, May 2001.

[18] T. Le. A-Decomposition und ihre Anwendung bei der Synthese mehrstufiger Schaltungen. In *Workshop Boolesche Probleme*, pages 81–89, October 1994.

[19] J. Lind-Nilsen. *BuDDy — A Binary Decision Diagram Package*. IT-højskolen i København, http://www.itu.dk/research/buddy/index.html.

[20] J. Lou and J. Brzozowski. A generalization of shestakov's function decomposition method. In *29th IEEE International Symposium on Multiple-Valued Logic*, pages 66–71, Freiburg, May 1999.

[21] T. Łuba. Decomposition of multiple-valued functions. In *25th IEEE International Symposium on Multiple-Valued Logic*, pages 256–261, 1995.

[22] R. Malvi, M. Perkowski, and L. Jozwiak. Exact graph coloring for functional decomposition: Do we need it? In *3rd International Workshop on Boolean Problems*, pages 1–10, Freiberg, September 1998.

[23] A. Mishchenko, C. Files, M. Perkowski, B. Steinbach, and C. Dorotska. Implicit algorithms for multi-valued input support minimization. In *4th International Workshop on Boolean Problems*, pages 9–21, Freiberg, September 2000.

[24] A. Mishchenko, B. Steinbach, and M. Perkowski. An algorithm for bi-decomposition of logic functions. In *38th Design Automation Conference*, pages 18–22, Las Vegas, June 2001.

[25] A. Mishchenko, B. Steinbach, and M. Perkowski. Bi-decomposition of multi-valued relations. In *10th International Workshop on Logic & Synthesis*, pages 35–40, Granlibakken, June 2001.

[26] M. Perkowski. *A Method of Solving Combinatorial Problems in Automatic Design of Digital Circuits*. PhD thesis, Warsaw Technical University, Warsaw, 1980.

[27] M. Perkowski. A new representation of strongly unspecified switching functions and its application to multi-level and/or/exor synthesis. In *Second Workshop on Applications of Reed-Muller Expansion in Circuit Design*, pages 143–151, Chiba City, August 1995.

[28] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. Zhang. Decomposition of multiple-valued relations. In *27th IEEE International Symposium on Multiple-Valued Logic*, pages 13–18, Halifax, Nova Scotia, Canada, May 1997.

[29] Portland State University, http://www.ee.pdx.edu/polo/. *POLO WebPage*.

[30] Povarov. *Povarov Decomposition*.

[31] Rambus Inc., http://www.rdram.com/. *Rambus Signaling Technologies — RSL, QRSL and SerDes Technology Overview*.

[32] T. Sasao and J. Butler. On bi-decompositions of logic functions. In *International Workshop on Logic Synthesis*, pages 18–21, Lake Tahoe, California, May 1997.

[33] Silicon Graphics Inc., http://www.sgi.com/tech/stl/. *SGI — Services & Support: Standard Template Library Programmer's Guide*.

[34] L. Skornjakow. *Elemente der Verbandstheorie*. Akademie-Verlag, Berlin, 1973.

[35] B. Steinbach. Dissertation, Technical University Karl-Marx-Stadt, Karl-Marx-Stadt.

[36] B. Steinbach and C. Lang. A general data structure for exor-decomposition of sets of switching function. In *3rd International Workshop on Boolean Problems*, pages 59–66, Freiberg, September 1998.

[37] B. Steinbach, C. Lang, and M. Perkowski. Bi-decomposition of discrete function sets. In *4th International Workshop on Applications of the Reed-Muller Expansion*, pages 233–252, Victoria, B.C., 1999.

[38] B. Steinbach and T. Le. Entwurf testbarer Schaltnetzwerke. Technical Report 12, Technical University Chemnitz, 1990.

[39] B. Steinbach, M. Perkowski, and C. Lang. Bi-decomposition of multi-valued functions for circuit design and data mining applications. In *29th IEEE International Symposium on Multiple-Valued Logic*, pages 50–58, Freiburg, May 1999.

[40] B. Steinbach, F. Schuhmann, and M. Stöckert. Functional decomposition of speed optimized circuits. In D. Auvergne and R. Hartenstein, editors, *Power and Timing Modelling for Performance of Integrated Circuits*, pages 65–77. IT Press, Bruchsal, 1993.

[41] B. Steinbach and A. Wereszczynski. Synthesis of multi-level circuits using exor-gates. In *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion*, pages 161–168, 1995.

[42] B. Steinbach and A. Zakrevski. Three models and some theorems on decomposition of boolean functions. In *3rd International Workshop on Boolean Problems*, pages 11–18, Freiberg, September 1998.

[43] H. Teng and R. Bolton. *The Use of Arithmetic Operators in a Self-Restored Current-Mode CMOS Multiple-Valued Logic Design Architecture.* Warsaw, May 2001.

[44] UCI, http://www.ics.uci.edu/~mlearn/MLRepository.html. *UCI data base.*

[45] T. Uemura and T. Baba. A three-valued d-flip-flop and shift register using multiple-junction surface tunnel transistors. In *31th IEEE International Symposium on Multiple-Valued Logic*, pages 89–93, Warsaw, May 2001.

[46] T. Waho, K. Hattori, and Y. Takamatsu. Flash analog-to-digital converter using resonant-tunneling multiple-valued curcuits. In *31th IEEE International Symposium on Multiple-Valued Logic*, pages 94–99, Warsaw, May 2001.

[47] W. Wan and M. Perkowski. A new approach to the decomposition of incompletely specified multi-output functions based on graph coloring and local transformations and its application to fpga mapping. In *European Conference on Design Automation*, pages 230–235, Hamburg, September 1992.

[48] S. Yanushkevich. *Logic Differential Calculus in Multi-Valued Logic Design.* Habilitation thesis, Technical University of Szczecin, Szczecin, 1998.

[49] B. Zupan. *Machine Learning Based on Function Decomposition.* PhD thesis, University of Ljubljana, Ljubljana, 1997.

Index