Chapter 2

Background

In this chapter we review some basic notation and definitions that will be used in the sequel. In addition, we discuss tractable and intractable problems — one of the basic aspects of not just circuit design but also computer science. Finally, we discuss a state–of–the–art data structure — the reduced ordered binary decision diagram [6] — and justify our use of it as the basic data structure for our algorithms. For more information we refer to [23].

2.1 Notation

We mainly use the notations introduced in [23]. We use braces (i.e., $\{\}$) to denote *unordered* sets and parentheses (i.e., $[\,]$) to denote *ordered* sets. For example, an unordered set of three elements is denoted by $S = \{a, b, c\}$. Vectors of elements are ordered sets. They are denoted by lowercase bold characters. The *cardinality* of a set is the number of its elements. It is denoted by $||$. Given a set S, a *partition* of S is a set of disjoint subsets of S whose union is S. For example, $P_S = \{\{b\}, \{a, c\}\}$ is a partition for our example set. Set membership of an element is denoted by $\in$, set inclusion by $\subset$ or $\subseteq$. The symbol $\forall$ is the *universal quantifier*, the symbol $\exists$ the *existential quantifier*. Implication is denoted by $\Longrightarrow$ and co–implication by $\Longleftrightarrow$. The symbol : means *such that*.

The *Cartesian product* of two sets X and Y, denoted by $X \times Y$, is the set of all ordered pairs (x, y), such that $x \in X$ and $y \in Y$. A *relation* R between two sets X and Y is a subset of $X \times Y$. We write xRy when $x \in X$, $y \in Y$ and $(x, y) \in R$. An *equivalence relation* is a subset R of $X \times X$ which is *reflexive* (i.e., $(x, x) \in R$), *symmetric* (i.e., $(x, y) \in R \Longrightarrow (y, x) \in R$), and *transitive* (i.e., $(x, y) \in R$ and $(y, z) \in R \Longrightarrow (x, z) \in R$). A *partial order* is a relation between X and itself that is reflexive, anti–symmetric (i.e., $(x, y) \in R$ and $(y, x) \in R \Longrightarrow x = y$) and transitive.

A *function (or map)* between two sets X and Y is a relation having the property that each element of X appears as the first element in one and only one pair of the relation. A function between two sets X and Y is denoted by $f : X \longrightarrow Y$. The sets X and Y are called the *domain* and the *co–domain* of the function, respectively.

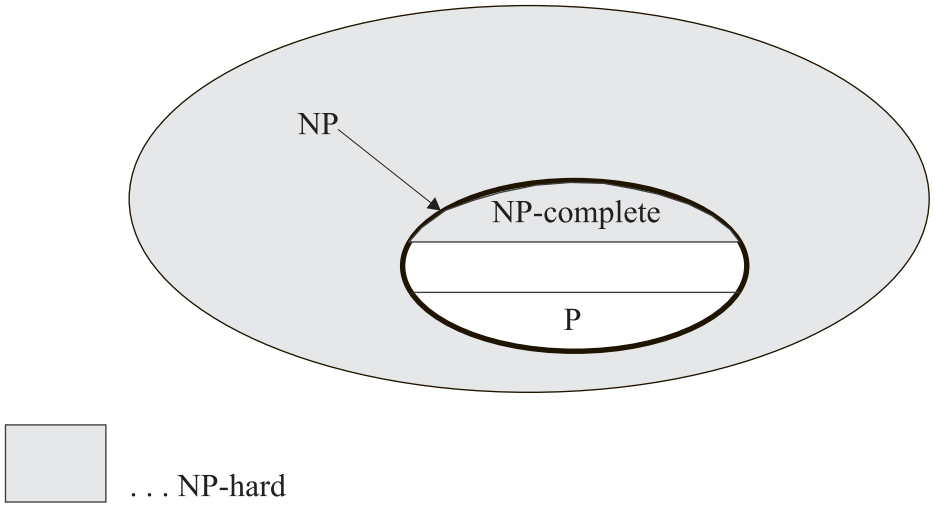

Figure 2.1: The Relation between $\mathcal{P}$ and $\mathcal{NP}$

2.2 The Problem of Complexity

Most problems of computer–aided design tasks for digital circuits are discrete in nature. In other words, it is necessary to solve combinatorial decision and optimization problems. Optimization problems can be reduced to sequences of decision problems. So from now on let us concentrate on decision problems.

A *decision problem* is a problem with a binary–valued solution, i.e., TRUE or FALSE. For instance, such a problem is the formal verification of digital circuits, i.e., the question whether the implementation and the specification of a digital circuit describe the same function.

Some of these decision problems can be solved by algorithms with polynomial complexity, i.e., the number of elementary operations which are repeated in the algorithm is polynomial in the size of some input to the algorithm. This class of problems is known as $\mathcal{P}$ or the class of *tractable* problems [23]. Unfortunatly, it covers only a small part of the problems in the synthesis and optimization of digital circuits. Then, there is another class of problems that could be solved by polynomial algorithms on non–deterministic machines. These are machines which can start with a guess before performing a deterministic algorithm. We call this class $\mathcal{NP}$. Obviously $\mathcal{P} \subseteq \mathcal{NP}$. The question of whether $\mathcal{P} = \mathcal{NP}$ is still unsolved.

However, there is a class of problems for which it has been shown, that if any of these problems can be solved with a polynomial algorithm, then $\mathcal{P} = \mathcal{NP}$. This class of problems is called $\mathcal{NP}$–hard, and the subclass of these problems which is also in $\mathcal{NP}$ is called $\mathcal{NP}$–complete. Figure 2.1 shows the relations among them.

The named property of $\mathcal{NP}$–hard problems implies that if a polynomial algorithm for one of these problems could be found then many other problems for which no polynomial algorithm has been known so far could be solved by polynomial algorithms as well. In other words, it is very unlikely

that there *are* polynomial algorithms for deterministic machines. That is, why these problems are called *intractable*.

Unfortunately, most of the problems that have to be solved in computer–aided design of microelectronic circuits belong to these class of intractable problems. So it is necessary to think about alternative solution possibilities for $\mathcal{NP}$–hard problems. The basic solution idea is as follows: if it is not possible to find an *exact* solution for a problem in reasonable time, i.e., if there is no polynomial algorithm to solve this problem, then try to find polynomial algorithms which are not guaranteed to find the exact solution for all problem instances but are able to provide good approximations to the exact solution for practical applications. Since these kind of algorithms work with heuristics, i.e., problem–solving techniques which are developed based on experiences, they are called *heuristic* algorithms. In other words, for $\mathcal{NP}$–hard problems the effort of research is to find heuristic algorithms that are expected to have polynomial complexity with small exponents and provide good solutions for a lot of practical problem instances.

2.3 Boolean Functions

First, let us consider the *Boolean n–space*. This is the multi–dimensional space spanned by n binary–valued Boolean variables and is denoted by $B^n = \{0,1\}^n$. A point in this space is referred to as a *minterm* and is denoted $c_1 \dots c_n$ with $c_i \in \{0,1\}$.

A *completely–specified Boolean function* is a mapping between two Boolean spaces. A Boolean function with n input and m output variables is a mapping $f : B^n \longrightarrow B^m$. An *incompletely–specified Boolean function* is defined over a subset of B^n. The minterms where the function is not defined are called *don't care* conditions. If we consider multiple–output functions, i.e., $m > 1$, the *don't care* components may differ for each output of the function. Therefore, incompletely–specified Boolean functions are represented as $f : B^n \longrightarrow \{0,1,*\}^m$, where $*$ represents a *don't care* condition. For each output of f, we can divide its domain into three subsets: the *off–set* includes all minterms for that the function value is 0, the *on–set* (also referred to as the *satisfy set*) those minterms for that the function value is 1, and finally, the *dc–set* contains those minterms for that the function value is $*$. A completely–specified Boolean function can also be described as the set of its on–set minterms.

The Boolean n–space can be graphically represented as a hypercube. Here, a point in B^n is represented by a binary–valued vector of dimension n. Now, when the binary input variables of a Boolean function f are associated with the components of B^n, a point in this Boolean space can be identified by the values of the corresponding variables. A *literal* is a variable x_i or its complement $\bar{x_i}$, and a product of n literals denotes a point (also referred to as a *vertex*) in B^n. Figure 2.2 (a) shows this cube for the three–dimensional Boolean space, B^3 with the three Boolean variables x_1, x_2, and x_3. In this cube of Figure 2.2(a) we can put the description of any Boolean function f with three input variables and one output variable. Let us do this for the example function $f = x_1x_2 + \bar{x_1}x_3$. Let the black dots indicate those points of the Boolean three–space that belongs to the on–set, and

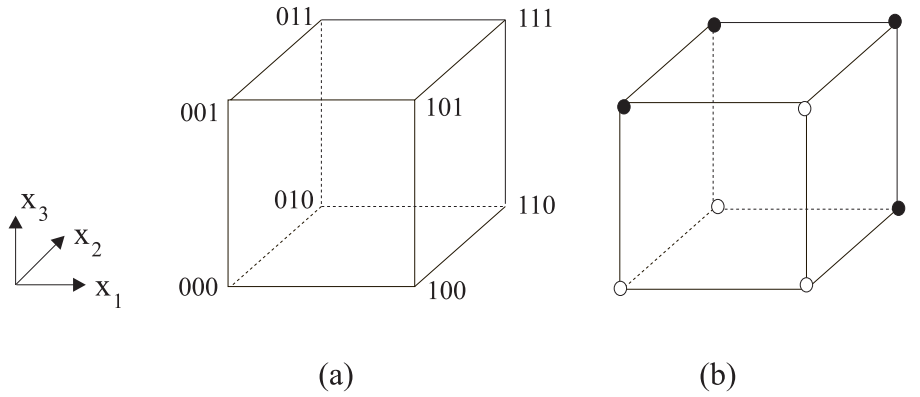

Figure 2.2: The Boolean 3–Space (a) and the Description of $f(x_1, x_2, x_3) = x_1x_2 + \bar{x_1}x_3$ (b)

white dots the points of the off–set of function f, then the graphical description of function f in a cube looks as described in Figure 2.2 (b). The generalization for incompletely–specified Boolean functions and/or functions with more than one output variable is straightforward: use a separate cube for each output, and denote those points of the domain that belongs to the dc–set with an extra sign like $\times$.

The *distance (or Hamming distance)* between two vertices v_1 and v_2 in the Boolean n–space is the number of components that their coordinates differ on. Considering for instance two points of the Boolean three–space, $v_1 = 000$ and $v_2 = 011$, we see that the distance between these two points is 2.

In the following, we will consider completely–specified Boolean functions. We denote the set of all those completely–specified Boolean functions with n input variables and m output variables by $\mathcal{B}_{n,m}$, the set of input variables by $X = [x_1, x_2, \ldots, x_n]$, and the set of output variables by $Y = [y_1, y_2, \ldots, y_m]$. Note that we use these sets as *ordered* sets of variables.

For the sake of simplicity, we will mostly consider a completely–specified Boolean function with n input variables and one output variable. For any of these $f \in \mathcal{B}_{n,1}$, we use the following basic definitions and notations.

The *lexicographical order relation* for Boolean functions is defined as follows: the Boolean function f is lexicographical smaller than a Boolean function $g \in \mathcal{B}_{n,1}$ ($f <_L g$) iff $f(c_1, \ldots, c_n) = 0$ and $g(c_1, \ldots, c_n) = 1$ and the binary vector $(c_1, \ldots, c_n)$ is an encoding of the smallest integer, such that $f(c_1, \ldots, c_n) \neq g(c_1, \ldots, c_n)$[8].

The *satisfy count* of function f is the number of on–set minterms of function f and is denoted as follows:

$$|f| = |\{c_1 \ldots c_n \in B^n : f(c_1, \ldots, c_n) = 1\}|.$$

Definition 2.1 *[23] The* **cofactor** *of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to x_i is $f_{x_i}(x_1, x_2, \ldots, x_i, \ldots, x_n) = f(x_1, x_2, \ldots, 1, \ldots, x_n)$. The* **cofactor** *of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to $\bar{x_i}$ is $f_{\bar{x_i}}(x_1, x_2, \ldots, x_i, \ldots, x_n) = f(x_1, x_2, \ldots, 0, \ldots, x_n)$.*

The cofactor functions f_{x_i} and $f_{\bar{x}_i}$ are considered as functions with the same number of input variables as the function f, i.e., as functions with n input variables.

The logical composition of cofactors of a Boolean function f is denoted as a *subfunction* of f.

The *essential variables* of a Boolean function f are those variables from that f depends on, i.e., x_i is an essential variable of function f iff $f_{x_i} \neq f_{\bar{x}_i}$.

Definition 2.2 *[23] A function* $f(\mathbf{x}) = f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ *is (positive/negative)* **unate in variable** x_i *if* $f_{x_i}(\mathbf{x}) \geq f_{\bar{x}_i}(\mathbf{x})$ *(*$f_{x_i}(\mathbf{x}) \leq f_{\bar{x}_i}(\mathbf{x})$*) for all possible assignments to the other variables* x_j *with* $j \neq i, j = 1, 2, \ldots, n$*. Otherwise it is* **binate** *in that variable.*

A function is (positive/negative) **unate** *if it is (positive/negative) unate in all essential variables. Otherwise it is binate.*

Besides the cofactor function, there are three other functions that will be used with respect to a Boolean function f: the existential abstraction, the universal abstraction, and the Boolean difference.

Definition 2.3 *[23] The* **existential abstraction (or smoothing)** *of function* $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ *with respect to a variable* x_i *is* $\exists_{x_i} f := f_{x_i} + f_{\bar{x}_i}$.

A minterm $c_1 \ldots c_i \ldots c_n$ belongs to the on–set of $\exists_{x_i} f$ if there exists an assignment to x_i which satisfies f. That is, $f(c_1, \ldots, 0, \ldots, c_n) = 1$ or $f(c_1, \ldots, 1, \ldots, c_n) = 1$. In other words, the existential abstraction represents those minterms for which the function is true for at least one assignment to x_i.

Definition 2.4 *[23] The* **universal abstraction (or consensus)** *of function* $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ *with respect to a variable* x_i *is* $\forall_{x_i} f := f_{x_i} \cdot f_{\bar{x}_i}$.

A minterm $c_1 \ldots c_i \ldots c_n$ belongs to the on–set of $\forall_{x_i} f$ if all assignments to x_i satisfy f. That is, both $f(c_1, \ldots, 0, \ldots, c_n) = 1$ and $f(c_1, \ldots, 1, \ldots, c_n) = 1$. So the universal abstraction of a function are those minterms for which the function is true for all assignments to x_i.

Definition 2.5 *[23] The* **Boolean difference** *of a function* $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ *with respect to a variable* x_i *is* $\frac{\partial f}{\partial x_i} := f_{x_i} \oplus f_{\bar{x}_i}$. $\oplus$ *is the exclusive-or operator.*

The Boolean difference represents the minterms for which f changes when variable x_i changes, i.e., it represents the minterms for which x_i is observable at f. When it is zero, then the function does not depend on x_i.

Now let us define a special subset of Boolean functions from $\mathcal{B}_{n,n}$. We denote with $\mathcal{P}_n \subset \mathcal{B}_{n,n}$ the set of all possible permutations on the set of input variables $X = [x_1, x_2, \ldots, x_n]$, i.e., one-to-one mappings of X onto itself. $\mathcal{P}_n$ is a group [17], and we call it *permutation group* in the sequel.

Let $f \in \mathcal{B}_{n,1}$ be a Boolean function and $\pi \in \mathcal{P}_n$ be a permutation of the input variables in X. Then we define:

$$f \circ \pi(x_1, \ldots, x_i, \ldots, x_n) = f(x_{\pi(1)}, \ldots, x_{\pi(i)}, \ldots, x_{\pi(n)}).$$

Often, we will consider a permutation $\pi \in \mathcal{P}_n$ as a map $\pi : X \longrightarrow X$ defined with $\pi(x_i) = x_{\pi(i)}$. Note that then $\pi_1 \circ \pi_2(x_i) = \pi_2(\pi_1(x_i))$ according to this definition of $\pi_1, \pi_2 \in \mathcal{P}_n$. Furthermore, we describe with

$$\pi(x_1, x_2, \ldots, x_n) = (\pi(x_1), \pi(x_2), \ldots, \pi(x_n))$$

the complete permutation π and denote by

$$\mathbf{1}(x_1, x_2, \ldots, x_n) = (x_1, x_2, \ldots, x_n)$$

the identity.

2.4 Reduced Ordered Binary Decision Diagrams

There are different ways to represent Boolean functions. We are especially interested in the representation by binary decision diagrams.

A *binary decision diagram* (BDD) represents a set of binary–valued decisions resulting in an overall decision which can be TRUE or FALSE. They are represented by trees or rooted, directed, and acyclic graphs, where the decisions are associated with the vertices. A special kind of BDDs are *ordered* BDDs which were introduced by Randal E. Bryant in 1986 [6].

Definition 2.6 *[23] An* **OBDD** *is a rooted directed graph with vertex set V. Each non–leaf vertex has as attributes a pointer $index(v) \in \{0, 1, \ldots, n-1\}$ to an input variable in the set $\{x_1, x_2, \ldots, x_n\}$, and two children $low(v)$ and $high(v) \in V$. A leaf vertex v has as attribute a value $value(v) \in \{0, 1\}$.*

For any vertex pair $\{v, low(v)\}$ (and $\{v, high(v)\}$) such that no vertex is a leaf,
$index(v) < index(low(v))$ (and $index(v) < index(high(v))$).

The restriction on the variable ordering guarantees that the graph is acyclic. Considering a vertex v and its two children $low(v)$ and $high(v)$, we also call $low(v)$ the 0–branch and $high(v)$ the 1–branch of vertex v. We associate a Boolean function with an OBDD as follows.

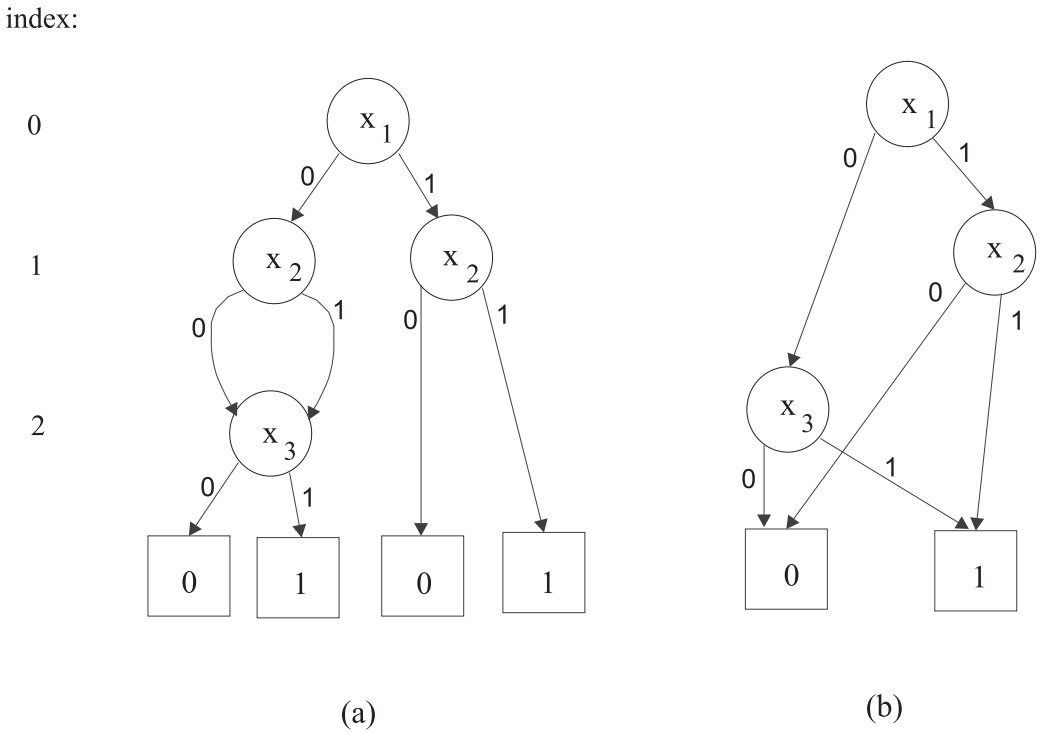

Figure 2.3: Ordered Binary Decision Diagrams for $f(x_1, x_2, x_3) = x_1x_2 + \bar{x_1}x_3$

Definition 2.7 *[23] An* OBDD *with root vertex v denotes a function f^v such that: If v is a leaf with $value(v) = 1$, then $f^v = 1$. If v is a leaf with $value(v) = 0$, then $f^v = 0$. If v is not a leaf and $index(v) = l$, where l is associated with variable x_i, then $f^v = \bar{x_i} \cdot f^{low(v)} + x_i \cdot f^{high(v)}$.*

Figure 2.3 shows two OBDDs of function $f = x_1x_2 + \bar{x_1}x_3$ with the variable ordering x_1, x_2, x_3. I.e., vertex index 0 is associated with variable x_1, index 1 with variable x_2, and index 2 with variable x_3. Changing the variable ordering in an OBDD means to change the relation from the indices to the variables, thus the variables in the ROBDD need to be reordered.

OBDDs have more practical applications than other kinds of BDDs because of two properties. First, OBDDs can be transformed into a unique form for representing a Boolean function $f \in \mathcal{B}_{n,1}$ by reduction of isomorphic subgraphs and redundant vertices.

Definition 2.8 *[23] An* OBDD *is said to be a* **reduced** OBDD *(or* **ROBDD***) if it contains no vertex v with $low(v) = high(v)$, nor any pair $\{u, v\}$ such that the subgraphs rooted in v and in u are isomorphic.*

The OBDD in Figure 2.3 (b) is an ROBDD for function $f = x_1x_2 + \bar{x_1}x_3$. Note that the ROBDD of any Boolean function is unique only with respect to a certain variable ordering, i.e., with resepct to a certain relation of the indices of the vertices of the ROBDD to the variables of the function. In [6], Randal E. Bryant proved that ROBDDs are unique forms for representing Boolean functions.

For this reason, they are especially suited for any problem in the computer–aided design of digital circuits for which equivalence checks between Boolean functions are necessary. Here, equivalence checking reduces to checking if the two unique representations of the two functions are the same.

The other property which has made ROBDDs popular in the circuit design area is that operations on ROBDDs can be done in polynomial time of their size, i.e., the number of vertices [6]. However, it would be a false conclusion that ROBDDs can be used to efficiently solve intractable problems. Unfortunatly, the size of OBDDs may strongly depend on the ordering of the variables. For example, the size of OBDDs for the adder function is very sensitive with respect to the selected variable ordering [23]. While it is polynomial in the number of input variables in the best case, it is exponential in the worst case. Other functions, like the arithmetic multiplier functions, do not have an OBDD representation with polynomial size. They have exponential OBDDs regardless of the selected variable ordering [6].

However, for a lot of common and reasonable practical examples of logic functions, a variable ordering can be found such that the size of their OBDDs is tractable. Many researchers have investigated heuristics to find optimal variable orderings (e.g., [3, 11, 12, 22, 32]). ROBDD packages have been developed and improved that allow an efficient manipulation of Boolean functions with the help of ROBDDs (e.g., [4, 18, 19, 24]). Alltogether, ROBDDs have developed to become the state–of–the–art data structure for representing Boolean functions and have been successfully used in many applications. The development of the ROBDD has led to a big factor in the progression of research in synthesis and verification of digital systems during the last 10 years.