Chapter 2
Preliminaries

This chapter provides the basic definitions and notations to keep the remaining book self-contained. The chapter is divided into three parts. In the first section, Boolean functions, reversible functions, and the respective circuit descriptions are introduced. This builds the basis for all approaches described in this book. Since many of the proposed techniques exploit decision diagrams and satisfiability solvers, respectively, the basic concepts of these core techniques are also introduced in the last two sections. All descriptions are thereby kept brief. For a more in-depth treatment, references to further reading are given in the respective sections.

2.1 Background

Reversible logic realizes bijective Boolean functions. Thus, first the basics regarding Boolean functions are revisited and further extended by a description of the properties specifically applied to reversible functions. Then, reversible circuits as well as quantum circuits are introduced which are used as realizations of reversible functions.

2.1.1 Reversible Functions

Every logic computation can be defined as a function over Boolean variables $\mathbb{B} \in \{0, 1\}$. More precisely:

Definition 2.1 A *Boolean function* is a mapping $f : \mathbb{B}^n \to \mathbb{B}$ with $n \in \mathbb{N}$. A function f is defined over its *input* variables $X = \{x_1, x_2, \ldots, x_n\}$ and hence is also denoted by $f(x_1, x_2, \ldots, x_n)$. The concrete mapping is described in terms of Boolean expressions which are formed over the variables from X and the operations $\wedge$ (*AND*), $\vee$ (*OR*), as well as $\bar{\cdot}$ (*NOT*).

R. Wille, R. Drechsler, *Towards a Design Flow for Reversible Logic*,
DOI 10.1007/978-90-481-9579-4_2,

Table 2.1 Boolean functions

(a) AND

$x_1 x_2$	$x_1 \wedge x_2$
0 0	0
0 1	0
1 0	0
1 1	1

(b) OR

$x_1 x_2$	$x_1 \vee x_2$
0 0	0
0 1	1
1 0	1
1 1	1

(c) NOT

x_1	$\overline{x}_1$
0	1
1	0

Example 2.1 Table 2.1 shows the truth tables of the operations AND, OR, and NOT, respectively. Each truth table has 2^n rows, showing the mapping of each input pattern to the respective output pattern.

Taking AND, OR, and NOT as a basis, every Boolean function can be derived. For example, the often used functions *XOR*, *implication*, and *equivalence* are derived as follows:

- XOR: $x_1 \oplus x_2 := (x_1 \wedge \overline{x}_2) \vee (\overline{x}_1 \wedge x_2)$
- Implication: $x_1 \Rightarrow x_2 := \overline{x}_1 \vee x_2$
- Equivalence: $x_1 \Leftrightarrow x_2 := \overline{x_1 \oplus x_2}$

So far, *single-output functions* have been introduced. However, in practice also *multi-output functions* are widely used.

Definition 2.2 A *multi-output Boolean function* is a mapping $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ with $n, m \in \mathbb{N}$. More precisely, it is a system of Boolean functions $f_i(x_1, x_2, \ldots, x_n)$ with $1 \leq i \leq m$. In the following multi-output functions are also termed as n-input, m-output functions or $n \times m$ functions, respectively.

Example 2.2 Table 2.2(a) shows the truth table of a 3-input, 2-output function representing the adder function.

This book considers reversible functions. Reversible functions are a subset of multi-output functions and are defined as follows:

Definition 2.3 A multi-output function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ is a *reversible function* iff

- its number of inputs is equal to the number of outputs (i.e. $n = m$) and
- it maps each input pattern to a unique output pattern.

In other words, each reversible function is a bijection that performs a permutation of the set of input patterns. A function that is not reversible is termed *irreversible*.

Example 2.3 Table 2.2(c) shows a 3-input, 3-output function. This function is reversible, since each input pattern maps to a unique output pattern. In contrast the function depicted in Table 2.2(a) is irreversible, since $n \neq m$. Moreover, also the

Table 2.2 Multi-output functions

(a) Irrev. (Adder)

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

(b) Irreversible

x_1	x_2	x_3	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

(c) Reversible

x_1	x_2	x_3	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	0	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	1	1	1

function in Table 2.2(b) is irreversible. Here, the number n of inputs indeed is equal to the number m of outputs, but there is no unique input-output mapping. For example, both inputs 000 and 001 map to the output 000.

Quite often, (irreversible) multi-output Boolean functions should be represented by reversible circuits. This necessitates the irreversible function to be *embedded* into a reversible one which requires the addition of constant inputs and garbage outputs defined as follows:

Definition 2.4 A *constant input* of a reversible function is an input that is set to a fixed value (either 0 or 1).

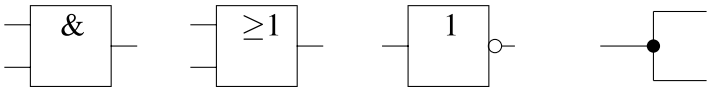
Definition 2.5 A *garbage output* of a reversible function is an output which is a don't care for all possible input conditions.

The problem of embedding is an integral part of synthesis which is described later in this book. In particular, Sect. 3.1.1 and Chap. 5 cover the respective aspects in detail.

Reversible functions can be realized by reversible logic. Due to its special properties, reversible logic found large interest in several domains like low-power design or quantum computation (see Chap. 1). As a result, synthesis of reversible functions has become an intensively studied topic in the last years. Therefore, new kinds of circuits have been proposed that are introduced and compared to traditional circuits in the next section.

2.1.2 Reversible Circuits

A circuit realizes a Boolean function. Usually, a circuit is composed of signal lines and a set of basic gates (called *gate library*). For traditional circuits, often the gate



(a) AND gate (b) OR gate (c) NOT gate (d) Fanout

Fig. 2.1 Traditional circuit elements

library depicted in Fig. 2.1 is used. This includes gates for the operations AND, OR, and NOT, based on which any Boolean function can be realized. Furthermore, fanouts are applied to use signal values more than once.

In contrast, to realize reversible logic some restrictions must be considered: fanouts and feedback are not directly allowed, since they would destroy the reversibility of the computation [NC00]. Also, the gate library from above as well as the traditional design flow cannot be utilized. As a result, a cascade structure over reversible gates is the established model to realize reversible logic.

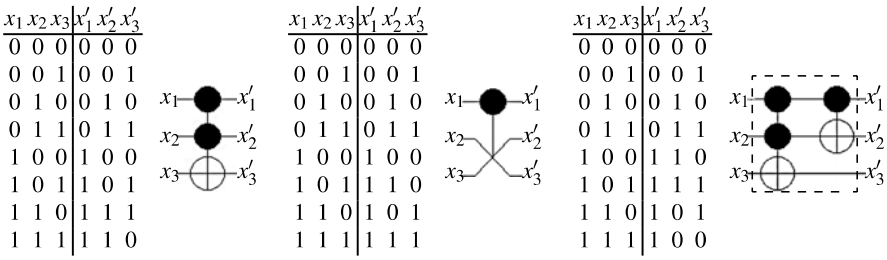
Definition 2.6 A *reversible circuit* G over inputs $X = \{x_1, x_2, \ldots, x_n\}$ is a cascade of reversible gates g_i, i.e. $G = g_0 g_1 \cdots g_{d-1}$ where d is the number of gates. A *reversible gate* has the form $g(C, T)$, where $C = \{x_{i_1}, \ldots, x_{i_k}\} \subset X$ is the set of *control lines* and $T = \{x_{j_1}, \ldots, x_{j_l}\} \subset X$ with $C \cap T = \emptyset$ is the set of *target lines*. C may be empty. The gate operation is applied to the target lines iff all control lines meet the required control conditions. Control lines and unconnected lines always pass through the gate unaltered.

In the literature, three types of reversible gates have been established:

- A (*multiple control*) *Toffoli gate* (MCT) [Tof80] has a single target line x_j and maps $(x_1, x_2, \ldots, x_j, \ldots, x_n)$ to $(x_1, x_2, \ldots, x_{i_1} x_{i_2} \cdots x_{i_k} \oplus x_j, \ldots, x_n)$. That is, a Toffoli gate inverts the target line iff all control lines are assigned to 1.
- A (*multiple control*) *Fredkin gate* (MCF) [FT82] has two target lines x_{j_1} and x_{j_2}. The gate interchanges the values of the target lines iff the conjunction of all control lines evaluates to 1.
- A *Peres gate* (P) [Per85] has a control line x_i, a target line x_{j_1}, and a line x_{j_2} that serves as both, control and target. It maps $(x_1, x_2, \ldots, x_{j_1}, \ldots, x_{j_2}, \ldots, x_n)$ to $(x_1, x_2, \ldots, x_i x_{j_2} \oplus x_{j_1}, \ldots, x_i \oplus x_{j_2}, \ldots, x_n)$ and thus is a cascade of two MCT gates.

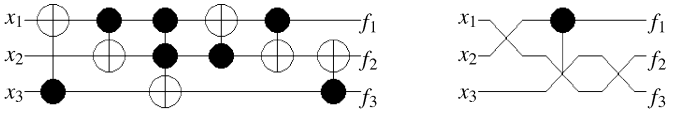
Example 2.4 Figure 2.2 shows a Toffoli gate (a), a Fredkin gate (b), and a Peres gate (c) together with a truth table of its functionality. A ● is used to indicate a control line, while an $\oplus$ ($\times$) is used for denoting the target line of a Toffoli and Peres gate (Fredkin gate).

Remark 2.1 These definitions also provide the basis for other gate types. For example, the Toffoli gate builds the basis for the *NOT* gate (a Toffoli gate with no control lines, i.e. with $C = \emptyset$), for the *controlled-NOT* gate (a Toffoli gate with one control



(a) Toffoli gate (b) Fredkin gate (c) Peres gate

Fig. 2.2 Reversible gates



(a) Toffoli gate circuit (b) Fredkin gate circuit

Fig. 2.3 Reversible circuits

line),[1] as well as for the Toffoli gate as originally proposed in [Tof80]. In contrast, the Fredkin gate builds the basis for a *SWAP* gate (a Fredkin gate with $C = \emptyset$, i.e. an interchanging of two lines).

In the following, the notations $MCT(C, x_j)$, $MCF(C, x_{j_1}, x_{j_2})$, and $P(x_i, x_{j_1}, x_{j_2})$ are used to denote a Toffoli, Fredkin, and Peres gate, respectively. The number of control lines, a Toffoli (Fredkin) gates consists of, defines the *size* of the gate.

Using these gate types, *universal* libraries can be composed. A gate library is called universal, if it enables the realization of any reversible function. For example, it has been proven that every reversible function can be realized using MCT gates only [MD04b]. Also, the gate library consisting of NOT, CNOT, and two-controlled Toffoli gates is universal [SPMH03]. In contrast, a library including only CNOT gates allows the realization of linear reversible functions only [PMH08].

Example 2.5 Figure 2.3 shows reversible circuits realizing the function depicted in Table 2.2(c) with the help of Toffoli and Fredkin gates, respectively.

As for their traditional counterparts, the complexity of reversible circuits is measured by means of different cost metrics. More precisely, the cost of the respective circuits is defined as follows:

[1] The controlled-NOT gate is also known as *CNOT* or *Feynman gate*.

Table 2.3 Quantum cost for Toffoli and Fredkin gates

No. of control lines	Quantum cost	
	of a Toffoli gate	of a Fredkin gate
0	1	3
1	1	7
2	5	15
3	13	28, if at least 2 lines are unconnected 31, otherwise
4	26, if at least 2 lines are unconnected 29, otherwise	40, if at least 3 lines are unconnected 54, if 1 or 2 lines are unconnected 63, otherwise
5	38, if at least 3 lines are unconnected 52, if 1 or 2 lines are unconnected 61, otherwise	52, if at least 4 lines are unconnected 82, if 1, 2 or 3 lines are unconnected 127, otherwise
6	50, if at least 4 lines are unconnected 80, if 1, 2 or 3 lines are unconnected 125, otherwise	64, if at least 5 lines are unconnected 102, if 1, 2, 3 or 4 lines are unconnected 255, otherwise

Definition 2.7 A reversible circuit $G = g_0 g_1 \cdots g_{d-1}$ has *cost* of

$$c = \sum_{i=0}^{d-1} c_i,$$

where c_i denotes the cost of gate g_i.

The concrete cost for a single gate of course depends on the respective type but also on the addressed technology. In this book, the following cost metrics are used:

- *Gate count* denotes the number of gates the circuit consists of (i.e. $c_i = 1$ and $c = d$).
- *Quantum cost* denotes the effort needed to transform a reversible circuit to a quantum circuit (see also next section). Table 2.3 shows the quantum cost for a selection of Toffoli and Fredkin gate configurations as introduced in [BBC+95] and further optimized in [MD04a] and [MYDM05]. As can be seen, gates of larger size are considerably more expensive than gates of smaller size. The Peres gate

represents a special case, since it has quantum cost of 4, while the realization with two Toffoli gates would imply a cost of 6.
- *Transistor cost* denotes the effort needed, to realize a reversible circuit in CMOS according to [TG08]. The transistor cost of a reversible gate is $8 \cdot s$ where s is the number of control lines.

Example 2.6 Consider the circuits from Example 2.5 depicted in Fig. 2.3. The Toffoli circuit has a gate count of 6, quantum cost of 10, and transistor cost of 56, while the Fredkin circuit has a gate count of 3, quantum cost of 13, and transistor cost of 8, respectively.

As can be seen, the costs significantly differ depending on the applied cost model. Even if the number of gates in a cascade is a very simple measure of its complexity, it is the most technology-independent metric. Thus, the gate count is often used to evaluate the quality of a reversible circuit. Besides that, also the quantum cost metric is popular because it represents a measure for the most intensely studied application (namely quantum computation) and considers larger gates to be more costly. The transistor cost model is a relatively new model that arose with the application of reversible circuits to the area of low-power CMOS design. In this book, gate count and quantum cost are primarily considered as it allows a fair comparison of synthesis results with respect to previous work. Transistor costs are additionally addressed where appropriate.

Finally, a special property of reversible logic is reviewed:

Lemma 2.1 *If the cascade of MCT gates $G = g_0 g_1 \cdots g_{d-1}$ realizes a reversible function f, then the reverse cascade $G = g_{d-1} g_{d-2} \cdots g_0$ realizes the inverse function f^{-1}.*

Proof Each reversible gate realizes a reversible function. That is, for each input pattern a unique output pattern, i.e. a one-to-one mapping, exists. Thus, calculating the inverse of the function f for an output pattern is essentially the same operation as propagating this pattern backwards through the circuit. □

This lemma is particularly exploited during synthesis of reversible logic as described later in this book.

The next section considers quantum circuits and how they are derived from reversible logic.

2.1.3 Quantum Circuits

Quantum computation [NC00] is a promising application of reversible logic. Every quantum circuit works on qubits instead of bits. In contrast to Boolean logic, qubits do not only allow to represent Boolean 0's and Boolean 1's, but also the superposition of both. More formally:

Definition 2.8 A qubit is a two level quantum system, described by a two dimensional complex Hilbert space. The two orthogonal quantum states

$$|0\rangle \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad |1\rangle \equiv \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

are used to represent the Boolean values 0 and 1. Any state of a qubit may be written as $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers with $|\alpha|^2 + |\beta|^2 = 1$. The quantum state of a single qubit is denoted by the vector

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

The state of a quantum system with $n > 1$ qubits is given by an element of the tensor product of the respective state spaces and can be represented as a normalized vector of length 2^n, called the state vector. The state vector is changed through multiplication of appropriate $2^n \times 2^n$ unitary matrices. Thus, each quantum computation is inherently reversible but manipulates qubits rather than pure logic values. At the end of the computation, a qubit can be measured. Then, depending on the current state of the qubit a 0 (with probability of $|\alpha|^2$) or a 1 (with probability of $|\beta|^2$) returns, respectively. After the measurement, the state of the qubit is destroyed.

In other words, using quantum computation and qubits in superposition, functions can be evaluated with different possible input assignments in parallel. But, it is not possible to obtain the current state of a qubit. Instead, if a qubit is measured, either 0 or 1 is returned depending on the respective probability. Nevertheless, researchers exploited quantum computation (in particular superposition) to solve many practically relevant problems faster than by traditional computing machines. For example, it was possible to solve the factorization problem in polynomial time—for traditional machines only exponential algorithms are known. Even if the research in this area is still at the beginning (so far, quantum algorithms with only up to 28 qubits have been implemented), these first promising results motivate further research in this area.

The focus of this book is how to design reversible and quantum circuits, respectively. Thus, in the following the model for quantum circuits as used in this book is introduced. For a more detailed treatment of the respective physical background, the reader is referred to [Pit99, NC00, Mer07].

Definition 2.9 A *quantum circuit* Q is a cascade of *quantum gates* q_i, i.e. $Q = q_0 \cdots q_{d-1}$.

In this book, the following quantum gates are considered:

- Inverter (NOT): A single qubit is inverted.
- Controlled inverter (CNOT): The target qubit is inverted if the control qubit is 1.
- Controlled V gate: A V operation is performed on the target qubit if the control qubit is 1. The V operation is also known as the square root of NOT, since two consecutive V operations are equivalent to an inversion.

Fig. 2.4 Quantum gates

GATE	NOTATION	MATRIX
NOT		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
CNOT		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$
V		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} \\ 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} \end{pmatrix}$
V+		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} \\ 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} \end{pmatrix}$

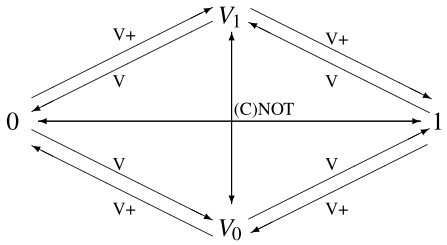


Fig. 2.5 State transitions for NOT, CNOT, V, and V+ operations

- Controlled V+ gate: A V+ operation is performed on the target qubit if the control qubit is 1. The V+ gate performs the inverse operation of the V gate, i.e. $V+ \equiv V^{-1}$.

The notation for these gates along with their corresponding $2^n \times 2^n$ unitary matrices is shown in Fig. 2.4.

In the following, the input to a quantum circuit as well as to each control line of a gate is restricted to 0 and 1. This has the effect that the value of each qubit is restricted to one value of the set $\{0, 1, V_0, V_1\}$, i.e. a 4-valued logic with

$$V_0 = \frac{1+i}{2}\begin{pmatrix} 1 \\ -i \end{pmatrix} \quad \text{and} \quad V_1 = \frac{1+i}{2}\begin{pmatrix} -i \\ 1 \end{pmatrix}$$

is applied. Figure 2.5 shows the resulting transitions with respect to the possible NOT, CNOT, V, and V+ operations. By restricting the quantum circuit model in this way, physical effects like superposition (and entanglement [NC00]) are excluded from the following consideration so that automated approaches (e.g. for synthesis, optimization, verification, etc.) become applicable. Nevertheless, the restricted model remains realistic for many applications. As an example, many of today's quantum algorithms (e.g. Deutsch's algorithm or Grover's algorithm [NC00]) include quantum realizations of reversible (Boolean) functions. Thus, the mentioned restrictions are common in the design of quantum circuits.

Fig. 2.6 Quantum circuit

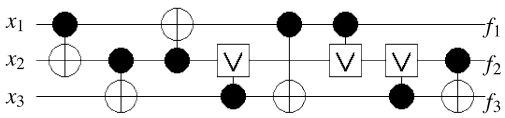
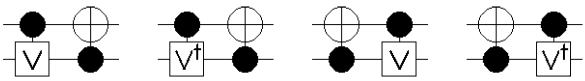


Fig. 2.7 Pairs of quantum gates with unit cost



Example 2.7 Figure 2.6 shows a quantum circuit realizing the reversible function depicted in Table 2.2(c).

All quantum gates are assumed to be the basic blocks of each quantum computation. This is also reflected in the cost metric.

Definition 2.10 Each quantum gate has cost of 1. Thus, the cost of a quantum circuit is defined by the number d of its gates.

Remark 2.2 In previous work, also an extended cost metric has been applied: When a CNOT and a V (or V+) gate are applied to the same two qubits, the cost of the pair can be considered unit as well [SD96, HSY+06]. The possible pairs (denoted by *double gates* in the following) are shown in Fig. 2.7. In this book, primarily the cost metric from Definition 2.10 is applied. However, all approaches can also be extended to consider unit cost of double gates. Exemplarily, this is shown for exact synthesis of quantum circuits in Sect. 4.2.2.

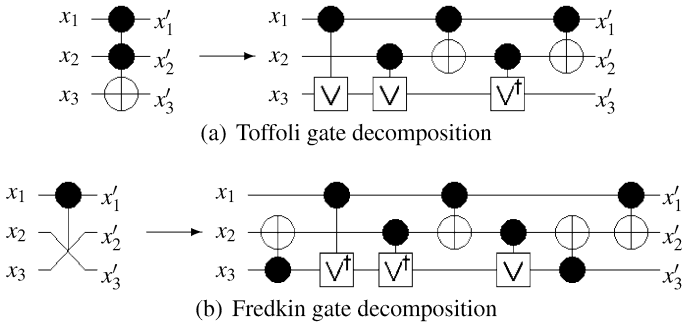
Since quantum circuits are inherently reversible, every reversible circuit can be transformed to a quantum circuit. To this end, each gate of the reversible circuit is *decomposed* into a cascade of quantum gates.

Example 2.8 Figure 2.8(a) (Fig. 2.8(b)) shows the quantum gate cascade which can be used to transform a Toffoli (Fredkin) gate to a quantum circuit. As can be seen, the number of required quantum gates is equal to the quantum cost of the Toffoli (Fredkin) gate as introduced in Table 2.3.

Exploiting these decompositions, synthesis of quantum circuits can be approached from two different angles: (1) Targeting quantum gates directly during the synthesis process or (2) synthesizing reversible circuits first that are later mapped into quantum circuits.

2.2 Decision Diagrams

To represent Boolean (including reversible) functions and circuits, decision diagrams can be applied. They provide an efficient data-structure that can represent



(a) Toffoli gate decomposition

(b) Fredkin gate decomposition

Fig. 2.8 Decomposition of reversible gates to quantum circuits

large functions in a more compact way than truth tables. In the past, several types of decision diagrams have been introduced. In this book, *Binary Decision Diagrams* (BDDs) [Bry86] are considered to represent Boolean functions. *Quantum Multiple-valued Decision Diagrams* (QMDDs) [MT06, MT08] are used to represent reversible functions that may include quantum operations. Both are briefly introduced in this section.

2.2.1 Binary Decision Diagrams

A Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ can be represented by a graph-structure defined as follows:

Definition 2.11 A *Binary Decision Diagram* (BDD) over Boolean variables X with terminals $T = \{0, 1\}$ is a directed acyclic graph $G = (V, E)$ with the following properties:

1. Each node $v \in V$ is either a terminal or a non-terminal.
2. Each terminal node $v \in V$ is labeled by a value $t \in T$ and has no outgoing edges.
3. Each non-terminal node $v \in V$ is labeled by a Boolean variable $x_i \in X$ and represents a Boolean function f.
4. In each non-terminal node (labeled by x_i), the Shannon decomposition [Sha38]

$$f = \overline{x}_i f_{x_i=0} + x_i f_{x_i=1}$$

is carried out, leading to two outgoing edges $e \in E$ whose successors are denoted by $low(v)$ (for $f_{x_i=0}$) and $high(v)$ (for $f_{x_i=1}$), respectively.

The *size* of a BDD is defined by the number of its (non-terminal) nodes.

Example 2.9 Figure 2.9 shows a BDD representing the function $f = x_1 \oplus x_2 \cdot x_3$. Edges leading to a node $f_{x_i=0}$ ($f_{x_i=1}$) are marked by a 0 (1). This BDD has a size of 5.

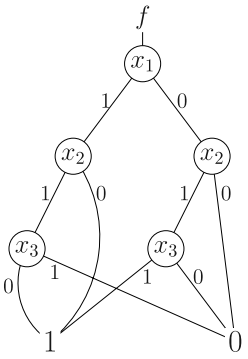


Fig. 2.9 BDD representing $f = x_1 \oplus x_2 \cdot x_3$

A BDD is called *free* if each variable is encountered at most once on each path from the root to a terminal node. A BDD is called *ordered* if in addition all variables are encountered in the same order on all such paths. The respective *order* is defined by $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$. Finally, a BDD is called *reduced* if it does neither contain isomorphic sub-graphs nor redundant nodes. To achieve reduced BDDs, *reduction rules* as depicted in Fig. 2.10 are applied. Applying the reduction rules leads to *shared nodes*, i.e. nodes that have more than one predecessor.

Example 2.10 Figure 2.11 shows two reduced ordered BDDs representing the function $f = x_1 \cdot x_2 + x_3 \cdot x_4 + \cdots + x_{n-1} \cdot x_n$. For the order $x_1, x_2, \ldots, x_{n-1}, x_n$, the BDD depicted in Fig. 2.11(a) has a size of $O(n)$, while the BDD depicted in Fig. 2.11(b) with the order $x_1, x_3, \ldots, x_{n-1}, x_2, x_4, \ldots, x_n$ has size of $O(2^n)$.

Remark 2.3 In the following, reduced ordered binary decision diagrams are called BDDs for brevity. BDDs are canonical representations, i.e. for a given Boolean function and a fixed order, the BDD is unique [Bry86].

As shown by Example 2.10, BDDs are very sensitive to the chosen variable order. It has been shown in [BW96] that proving the existence of a BDD with a lower number of nodes (i.e. proving that no other order leads to a smaller BDD size) is NP-complete. As a consequence, several heuristics to find good orders have been proposed. In particular, *sifting* [Rud93] has been shown to be quite effective.

Further reductions of the BDD size can be achieved, if *complement edges* [BRB90] are applied. They allow to represent a function as well as its complement by one single node only. BDDs can also be used to represent multi-output functions. Then, all BDDs for the respective functions are shared, i.e. isomorphic sub-functions are represented by a single node as well.

For a more comprehensive introduction into BDDs, the reader is refereed to [DB98, EFD05]. For the application of BDDs in practice, many well-engineered BDD packages (e.g. *CUDD* [Som01]) are available.

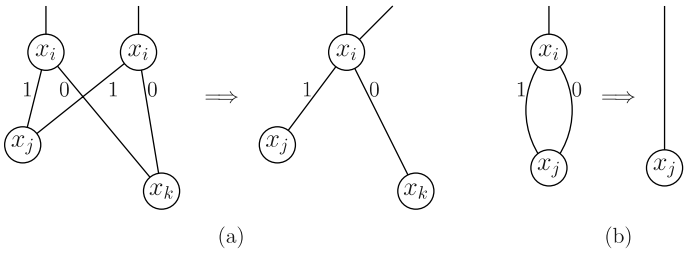


Fig. 2.10 Reduction rules for BDDs

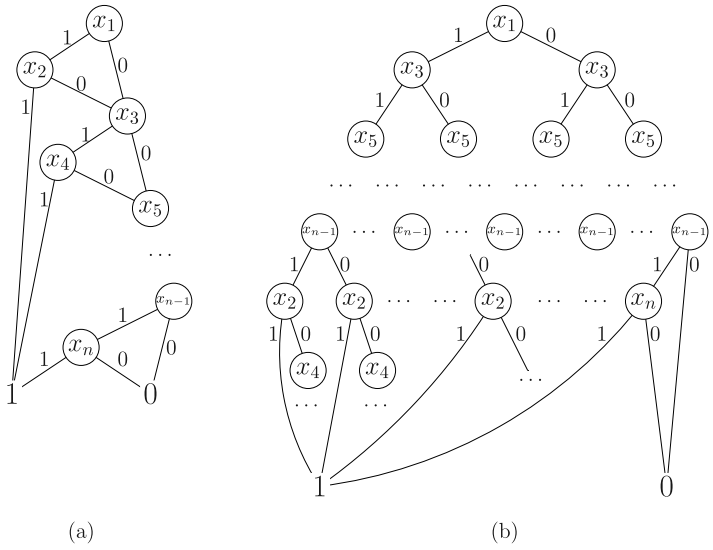
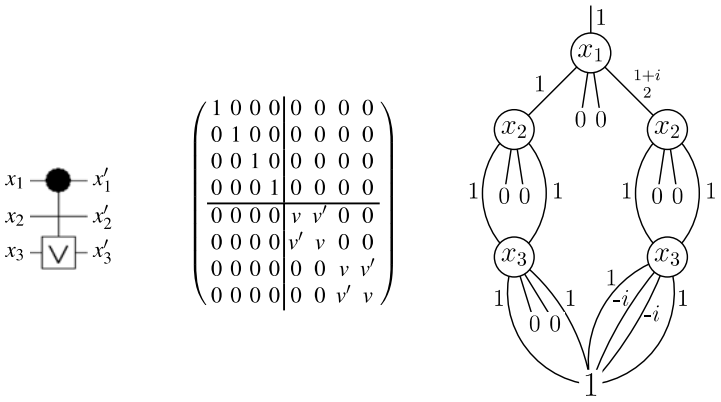


Fig. 2.11 BDDs with different variable orders

2.2.2 *Quantum Multiple-valued Decision Diagrams*

As described in Sect. 2.1.3, quantum operations are defined by $2^n \times 2^n$ unitary matrices (consider again Fig. 2.4 on p. 15 for examples). Thus, to represent functions including quantum operations, an adjusted data-structure is needed. *Quantum Multiple-valued Decision Diagrams* (QMDDs) [MT06, MT08] provide for the representation and manipulation of $r^n \times r^n$ complex-valued matrices with r pure logic states. This includes unitary matrices and thus QMDDs can be applied to represent quantum gates and circuits. Since in this book QMDDs are used as black box only (in contrast to BDDs), a formal definition of QMDDs is omitted and instead they are introduced by exemplarily describing the general idea.



(a) V gate (b) Matrix (c) QMDD

Fig. 2.12 QMDD representing the matrix of a single V gate

A QMDD structure is based on partitioning an $r^n \times r^n$ matrix M into r^2 sub-matrices, each of dimension $r^{n-1} \times r^{n-1}$ as shown in the following equation:

$$M = \begin{pmatrix} M_0 & M_1 & \cdots & M_{r-1} \\ M_r & M_{r+1} & \cdots & M_{2r-2} \\ \vdots & \vdots & \ddots & \vdots \\ M_{r^2-r} & M_{r^2-r+1} & \cdots & M_{r^2-1} \end{pmatrix}.$$

In the following, the concepts of QMDDs are briefly presented by way of the following example of a single V gate.

Example 2.11 Figure 2.12(a) shows a V gate in a 3-line circuit. The unitary matrix describing the behavior of this gate is given in Fig. 2.12(b) where $v = \frac{1+i}{2}$ and $v' = \frac{1-i}{2}$. The QMDD for this matrix is given in Fig. 2.12(c). The edges from each non-terminal node point to four sub-matrices indexed 0, 1, 2, 3 from left to right. Each edge has a complex-valued weight. For clarity, edges with weight 0 are indicated as stubs. In fact, they point to the terminal node.

The key features of QMDD are evident in this example. There is a single terminal node. Furthermore, each edge has a complex-valued weight. Each non-terminal node represents a matrix partitioning. For example, the top node in Fig. 2.12(c) represents the partitioning shown in Fig. 2.12(b). The non-terminal nodes lower in the diagram represent similar partitioning of the resulting sub-matrices. The representation of common sub-matrices is shared. To ensure the uniqueness of the representation, edges with weight 0 must point to the terminal node and normalization is applied to non-terminal nodes so that the lowest indexed edge with non-zero weight has weight 1.

As for BDDs, an efficient implementation exists also for QMDDs. However, since QMDD involve multiple edges from nodes and are applicable to both binary

and multiple-valued problems, the QMDD package is not built using a standard decision diagram package. Nevertheless, the implementation employs well-known decision diagram techniques like sharing, reordering, and so on. For a more comprehensive introduction into QMDDs, the reader is referred to [MT08].

2.3 Satisfiability Solvers

The methods described in this book make use of techniques for solving the *Boolean satisfiability problem* (SAT problem). The SAT problem is one of the central NP-complete problems. In fact, it was the first known NP-complete problem that was proven by Cook in 1971 [Coo71]. Despite this proven complexity, efficient solving algorithms have been developed that found great success as proof engines for many practically relevant problems. Today there exists algorithms exploiting SAT that solve many practical problem instances, e.g. in the domain of automatic test pattern generation [Lar92, DEF+08], logic synthesis [ZSM+05], debugging [SVAV05], and verification [BCCZ99, CBRZ01, PBG05].

In this section, the SAT problem, the respective solving algorithm, and its application are introduced. Furthermore, extended SAT solvers additionally exploiting bit-vector logic, quantifiers, or problem-specific modules, respectively, are briefly reviewed. These engines are used later as core techniques for selected steps in the proposed flow for reversible logic.

2.3.1 Boolean Satisfiability

The *Boolean satisfiability problem* (SAT problem) is defined as follows:

Definition 2.12 Let $h : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function. Then, the SAT problem is to find an assignment to the variables of h such that h evaluates to 1 or to prove that no such assignment exists.

In other words, SAT asks if $\exists X h$ for an h over variables X and determines a satisfying assignment in this case. In this context, the Boolean formula h is often given in *Conjunctive Normal Form* (CNF). A CNF is a set of clauses, each clause is a set of literals, and each literal is a Boolean variable or its negation. The CNF formula is satisfied if all clauses are satisfied, a clause is satisfied if at least one of its literals is satisfied, and a variable is satisfied when 1 is assigned to the variable (the negation of a variable is satisfied under the assignment 0).

Example 2.12 Let $h = (x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_3)(\overline{x}_2 + x_3)$. Then, $x_1 = 1, x_2 = 1$, and $x_3 = 1$ is a satisfying assignment for h. The values of x_1 and x_2 ensure that the first clause becomes satisfied, while x_3 ensures this for the remaining two clauses.

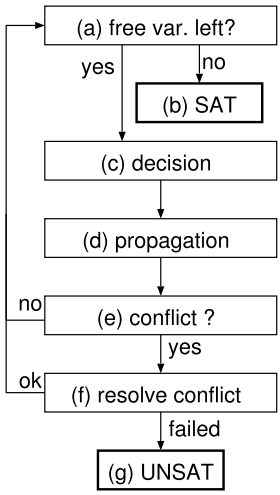


Fig. 2.13 Solving algorithm in modern SAT solvers

To solve SAT problems, in the past several (backtracking) algorithms (or *SAT solvers*, respectively) have been proposed [DP60, DLL62, MS99, MMZ+01, GN02, ES04]. Most of them apply the steps depicted in Fig. 2.13: While there are free variables left (a), a decision is made (c) to assign a value to one of these variables. Then, implications are determined due to the last assignment (d). This may cause a conflict (e) that is analyzed. If the conflict can be resolved by undoing assignments from previous decisions, backtracking is done (f). Otherwise, the instance is unsatisfiable (g). If no further decision can be made, i.e. a value is assigned to all variables and this assignment did not cause a conflict, the CNF is satisfied (b). Advanced techniques like e.g. *efficient Boolean constraint propagation* [MMZ+01] or *conflict analysis* [MS99] as well as efficient decision heuristics [GN02] are common in state-of-the-art SAT solvers today.

These techniques as well as the tremendous improvements in the performance of the respective implementations [ES04] enable the consideration of problems with more than hundreds of thousands of variables and clauses. Thus, SAT is widely used in many application domains. Therefore, the real world problem is transformed into CNF and then solved by using a SAT solver as a black box.

2.3.2 Extended SAT Solvers

Despite their efficiency, Boolean SAT solvers have a major drawback: they work on the Boolean level. But, many problems are formulated on a higher level of abstraction and would benefit from a more general description, respectively. As a consequence, researchers investigated the use of more expressive formulations than CNF—by still exploiting the established SAT techniques. This leads (1) to the combination of SAT solvers with decision procedures for decidable theories resulting

in *SAT Modulo Theories* (SMT) [BBC+05, DM06b] and (2) to the application of quantifiers resulting in *Quantified Boolean Formulas* (QBF) [Bie05, Ben05]. Furthermore, problem-specific knowledge is exploited during the solving process by the SAT solver *SWORD* [WFG+07]. The respective concepts are briefly reviewed in the following.

2.3.2.1 SMT Solvers for Bit-vector Logic

An SMT solver integrates a Boolean SAT solver with other solvers for specialized theories (e.g. linear arithmetic or bit-vector logic). The SAT solver thereby works on an abstract representation (still in CNF) of the problem and steers the overall search process, while each (partial) assignment of this representation has to be validated by the theory solver for the theory constraints. Thus, advanced SAT techniques together with specialized theory solvers are exploited.

In this book, the theory of *quantifier free bit-vector logic* (QF_BV) is utilized. This logic is defined as follows:

Definition 2.13 A *bit-vector* is an element $\mathbf{b} = (b_{n-1}, \ldots, b_0) \in \mathbb{B}^n$. The *index* $[\,] : \mathbb{B}^n \times [0, n) \to \mathbb{B}$ maps a bit-vector $\mathbf{b}$ and an index i to the ith component of the vector, i.e. $\mathbf{b}[i] = b_i$. Conversion from (to) a natural number is defined by $\text{nat} : \mathbb{B}^n \to N$ ($\text{bv} : N \to \mathbb{B}^n$) with $N = [0, 2^n) \subset \mathbb{N}$ and $\text{nat}(\mathbf{b}) := \sum_{i=0}^{n-1} b_i \cdot 2^i$ ($\text{bv} := \text{nat}^{-1}$).

Problems can be constraint by using bit-vector operations as well as arithmetic operations. Let $\mathbf{a}, \mathbf{b} \in \mathbb{B}^n$ be two bit-vectors. Then, the *bit-vector operation* $\circ \in \{\wedge, \vee, \ldots\}$ is defined by $\mathbf{a} \circ \mathbf{b} := (\mathbf{a}[n-1] \circ \mathbf{b}[n-1], \ldots, \mathbf{a}[0] \circ \mathbf{b}[0])$. An *arithmetic operation* $\bullet \in \{\cdot, +, \ldots\}$ is defined by $\mathbf{a} \bullet \mathbf{b} := \text{nat}(\mathbf{a}) \bullet \text{nat}(\mathbf{b})$.

Example 2.13 Let $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ be three bit-vector variables with bit-width $n = 3$ and $(\mathbf{a} \vee \mathbf{b} = \mathbf{c}) \wedge (\mathbf{a} + \mathbf{b} = \mathbf{c})$ an SMT bit-vector instance over these variables. Then, $\mathbf{a} = (010)$, $\mathbf{b} = (001)$, and $\mathbf{c} = (011)$ is a satisfying solution of this instance, since it satisfies each constraint.

To solve SMT instances in QF_BV logic either (1) a combination of a traditional SAT solver and a specialized (bit-vector) theory solver is applied (see e.g. [BBC+05, DM06a]), (2) the instance is pre-processed exploiting the higher level of abstraction before the resulting (simplified) instance is bit-blasted to a traditional SAT solver (see e.g. [GD07, BB09]), or (3) a specialized solver that directly works on the bit-level of the problem is used (see e.g. [DBW+07]).

Having an efficient solver available, similar to Boolean SAT the real world problem is transformed into a QF_BV instance. But instead of a description in terms of clauses, the higher level representation in terms of bit-vectors is used. Then, the resulting instance is passed to the solver which is again used as a black box. The higher abstraction which is now available can be exploited to accelerate the solving process.

2.3.2.2 QBF Solvers

Another generalization of the SAT problem is given by *QBF satisfiability*. Here, variables of the Boolean function h additionally can be universally and existentially quantified. More formally:

Definition 2.14 Let $h : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function over the variables X (usually given in CNF). Then, $Q_1 X_1 \dots Q_t X_t h$ with disjunct $X_i \subset X$ and $Q_i \in \{\exists, \forall\}$ is a *Quantified Boolean Formula* (QBF). The QBF problem is to find an assignment to the variables of h such that h evaluates to 1 with respect to the quantifiers or to prove that no such assignment exists.

Example 2.14 Let $\exists x_2, x_3 \forall x_1 (x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_3)(\overline{x}_2 + x_3)$. Then $x_2 = 1$ and $x_3 = 1$ is a satisfying assignment for the QBF h. The value of x_2 ensures that the first clause becomes satisfied, while x_3 ensures this for the remaining two clauses for all possible assignments to x_1.

Obviously, solving QBF problems is significantly harder than solving pure SAT instances. In fact, it is PSPACE-complete [Pap93]. Nevertheless, QBF enables the formulation of many problems in a more compact way. In this sense, complexity is moved from the problem formulation to the solving engine, i.e. the task can be formulated in a more compact way resulting in a more complex problem to be solved by the solver. However, since usually solving engines are well-engineered with respect to the dedicated problem, this may lead to a faster solving process. Today, recent solvers (e.g. [Bie05, Ben05]) exploit techniques like symbolic skolemization to solve QBF instances (i.e. converting the instance to a normal form which enables simplifications).

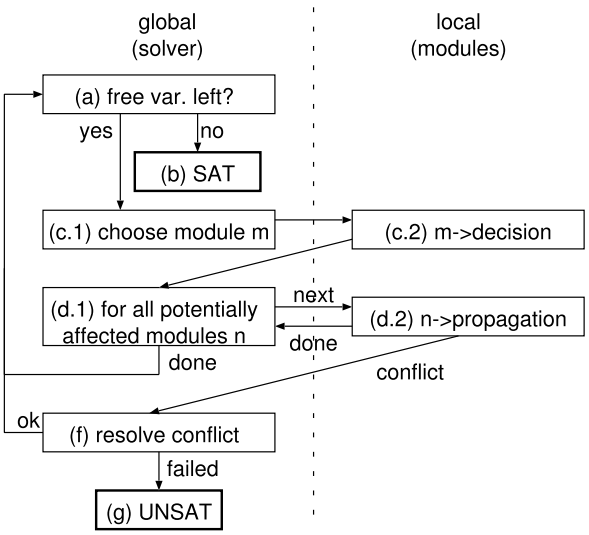
2.3.2.3 SWORD Solver

Due to the translation of the problem into CNF (or QF_BV logic, respectively), problem-specific knowledge is lost. More illustrative, decisions, implications, and learning schemes can only exploit the Boolean (bit-vector) description. In contrast, with more problem-specific knowledge available, more options exist how to control the search space traversion. This observation is exploited by the problem-specific SAT solver *SWORD* [WFG+07].[2]

SWORD represents the problem in terms of so called *modules*. Each module defines an operation over bit vectors of *module variables*. Each module variable is a Boolean variable. By this, structural and semantical knowledge is available which can be exploited by special algorithms for each kind of module. Furthermore, this

[2]SWORD has been co-developed by the authors of this book. Even if SWORD is focused on problem-specific knowledge, it can also be used as an SMT solver and already participated in the respective SMT competitions in 2008 [WSD08] and 2009 [JSWD09], respectively.

Fig. 2.14 SWORD algorithm



leads to a more compact problem formulation, since representing complex operations in terms of modules substitutes a significant amount of clauses.

Example 2.15 Consider an $n \times n$-multiplier. This multiplier can be represented by n^2 AND gates and $n-1$ adders [MK04]. Furthermore, a single AND gate can be modeled by three clauses and requires $\theta(n^2)$ auxiliary variables. Thus, just to encode the AND gates a CNF with $\theta(n^2)$ auxiliary variables and clauses is required, respectively. In contrast, using SWORD only $3n$ module variables (for the two inputs and the output of the multiplication) and a single (multiplier) module are needed to represent the whole multiplication.

Given a SAT instance including modules, the overall algorithm depicted in Fig. 2.14 is used to solve the problem. This algorithm is similar to the procedure as applied in standard SAT solvers (see Fig. 2.13 on p. 22): While free variables remain (a), a decision is made (c), implications resulting from this decision are carried out (d), and if a conflict occurs, it is analyzed (f). The important difference is that SWORD has two operation levels: The *global* algorithm controls the overall search process and calls *local* procedures of modules for decision and implication. Thus, decision making and the implication engine can be adjusted for each type of module.

In more detail, the solver first chooses a particular module based on a *global decision heuristic* (c.1). Then, this module chooses a value for one of its variables according to a *local decision heuristic* (c.2). Afterwards, the solver calls the *local implication procedures* (d.2) of all modules that are potentially affected (d.1) by the previous decision or implication. Here, a *variable watching scheme* similar to the one presented in [MMZ+01] is used which can efficiently determine these modules. The chosen modules imply further assignments and detect conflicts.

Due to the two operation levels, problem-specific strategies e.g. for decision making and propagation can be exploited by the modules. For example, decision making can be prioritized so that modules, which are assumed to be "more important" than others, are selected for a decision with a higher priority than less important modules. Furthermore, different modules can be equipped with different strategies. For a more detailed description of SWORD, the reader is referred to [WFG+07, WFG+09].

Therewith, all preliminaries required for this book have been introduced. Besides an introduction of reversible and quantum logic, also the applied core techniques have been briefly described. With that as a basis, the contributions towards a design flow for reversible logic are proposed in the following chapters. Decision diagrams are thereby applied for synthesis (Sect. 3.2), partially for exact synthesis (Sect. 4.3.2), and for verification (Sect. 7.1.2), while Boolean satisfiability is exploited in exact synthesis (especially in Sect. 4.2 and partially in Sect. 5.3 as well as Sect. 6.3), verification (Sect. 7.1.3), and debugging (Sect. 7.2). The extended SAT solvers (i.e. SMT solvers, QBF solvers, and SWORD) are used to improve exact synthesis (Sect. 4.3).

Note on Benchmarks In the following chapters, the respective contributions are introduced and evaluated in detail. Different scopes are thereby considered that are also reflected in the benchmark sets used to experimentally evaluate the respective methods. Synthesis approaches are evaluated using large functions (for the proposed heuristic method), small functions (for exact synthesis), and irreversible functions (to evaluate the different embeddings). In contrast, the methods targeting optimization, verification, and debugging work on a given circuit description. Furthermore, different timeouts are applied in the respective evaluations (e.g. exact synthesis normally requires more run-time than heuristic synthesis). As already stated in the introduction, the benchmarks used in this book are publicly available at www.revlib.org. The resulting tools can be obtained under www.revkit.org.

http://www.springer.com/978-90-481-9578-7

Towards a Design Flow for Reversible Logic

Wille, R.; Drechsler, R.

2010, XIII, 184 p., Hardcover
ISBN: 978-90-481-9578-7