自動合成量子循序電路

Automatic Synthesis of Sequential Quantum Boolean Circuits Based on Self-Timed Specifications

研究生：張立楷 (Li-Kai Chang)

指導教授：鄭福炯 (Prof. Fu-Chiung Cheng)

大同大學

資訊工程研究所

碩士論文

Thesis for Master of Science

Department of Computer Science and Engineering
Tatung University

中華民國九十三年八月

August 2004

大同大學
資訊工程研究所
碩士學位論文

自動合成量子循序電路

張立楷

經 考 試 合 格 特 此 證 明

碩士學位論文考試委員

蔡一鳴

鄭福炯

林慧珍

指導教授

鄭福炯

所長

包蒼龍

中 華 民 國 93 年 7 月 21 日

中文摘要

本篇論文提出了一個以基於 Toffoli gate 的量子循序電路的轉換方式，在轉換電路的過程中，我們可以使用 state graph (SG)來描述電路的行爲，再利用本篇論文所提出的轉換方式將自時電路轉換成量子電路。此外，我們利用 IP reuse 的概念讓產生的量子電路具有可組合及可重複使用的特性，因此我們就可使用這些如同模組化的量子電路元件來加以組合成爲更複雜的量子電路。

在這篇論文中，我們不只提出了將 SG 轉換成量子循序電路的方法與演算法，同時也以 Java 實作了一個電腦輔助軟體來設計與自動合成量子循序電路及具可組合特性的量子電路。

利用本篇論文所提出的合成方法，我們成功的將自時電路的 universal set 加以合成，並將其用於組合非同步電路中的控制(control-path)電路。

INDEX

Abstract

This thesis presents a methodology to transfer self-timed circuit specifications into sequential quantum Boolean circuits (SQBCs). State graphs (SGs) are used to describe the behaviors of self-timed circuits and then are translated into SQBCs based on Toffoli gates.

The concept of IP (Intellectual Property) reuse is applied to the constructed SQBCs to produce reusable and composable quantum Boolean circuits (CQBCs). Therefore, these reusable CQBCs as basic modular components can be exploited to construct more complicated quantum Boolean circuits.

Based on our methodology a CAD tool written in Java to automatically synthesize SQBCs and CQBCs is designed and implemented.

A universal set of self-timed components is successfully and automatically synthesized into CQBCs by using our CAD tool. These CQBCs can be used as building blocks to compose all control-path components of self-timed systems.

Chapter 1.

Introduction

Due to the discovery of Shor's prime factorization and Grover's fast database search algorithm [12, 13] quantum computing becomes one of the most rapidly expanding research fields. To perform quantum algorithms, required unitary operations should be expressed as a sequence of basic operations which can be implemented by a quantum computer. To implement a quantum computer, quantum Boolean circuits which consist of quantum gates need to be constructed first [1].

According the current computer architecture, conventional computers are constructed by many different circuit components such as control units, registers and ALUs. Thus if quantum computers are to be realized, quantum Boolean circuits must be implemented first.

The major differences between conventional circuits and quantum ones are their logic gates and wires [6]. Firstly, conventional circuits are based on AND, OR and NOT gates and quantum Boolean circuits are based on NOT, Controlled-Not and Controlled-Controlled-Not gates (i.e. Toffoli gates) [8]. Secondly, the wires in conventional circuits are used to connect components. This is very different with the one in quantum Boolean circuits because wires represent the time evolution.

Due to the above differences, a completely different methodology to synthesize quantum Boolean circuits must be investigated and proposed. Tsai and Kuo [1] propose a methodology to synthesize combinational quantum Boolean circuits based on transformation tables. Any general *m*-to-*n* bit combinational Boolean logic can be synthesized by using Toffoli gates. Iwama et al. [6] propose transformation rules for optimize Controlled-Not-based combinational quantum Boolean circuits and point out a design theory for a sequential quantum circuit is very interesting. To the best of our knowledge there are no related works on synthesizing sequential circuit behaviors into quantum Boolean circuits yet.

This thesis presents a methodology to transfer self-timed circuit specifications into sequential quantum Boolean circuits. State graphs [3, 4] are used to describe the behaviors of self-timed circuits and then are translated into SQBCs based on Toffoli gates. Furthermore, the concept of IP reuse is applied to the constructed SQBCs to produce reusable and composable quantum Boolean circuits (CQBC). Therefore, reusable CQBCs as basic modular components can be exploited to construct more complicated quantum Boolean circuits.

A CAD tool based on our methodology to automatically synthesize SQBCs and CQBCs is also designed and implemented. A universal set of self-timed components is successfully and automatically synthesized into both SQBCs and CQBCs by using

our CAD tool. These CQBCs can be used as building blocks to compose all control-path components of self-timed systems.

This thesis is organized as follows. Chapter 2 introduces basic knowledge on quantum systems and self-timed systems. Tsai and Kuo's [1] methodology of synthesizing combinational quantum Boolean circuits is presented in chapter 3. Chapter 4 and 5 present our methodologies to synthesize sequential quantum Boolean circuits (SQBCs) and reusable CQBC based state graph specifications, respectively. The experimental results of SQBCs and CQBCs synthesis are given in chapter 6. The design flow of our CAD tool is presented in chapter 7. Chapter 8 concludes this thesis as a whole and provides some suggestions for future work.

Chapter 2.

Background

In this chapter we provide the background knowledge on quantum systems and self-timed systems. For quantum systems quantum bits, quantum gates and quantum circuits are briefly described and for self-timed system trace theory, state graph and basic self-timed elements are presented and used in this thesis.

2.1 Fundamental of Quantum Systems

Circuit design and data representation in conventional computers and quantum ones are different in nature. In classical computers data are represented by bits and circuits are network of logic gates while in quantum computers data are represented by quantum bits (Qubits) and quantum circuits are made of a sequence of unitary operations which are represented by quantum gates and quantum wires [2]. Table 2.1 summarizes the differences between classical and quantum computers.

Table 2.1: The differences between classical and quantum computers

Classical Computer	***Quantum Computer***
Bit	Qubit
Logic gates	Quantum gates
Wires	Quantum wires
Circuits	Quantum circuits

2.1.1 Quantum Bits

The basic unit of information in quantum circuit model is quantum bit (qubit). A qubit is simply a two level quantum system. States $|0\rangle$ and $|1\rangle$ are two possible states for a qubit . The notation like '$\rangle$' is called *Dirac notation* and $|0\rangle$ and $|1\rangle$ are *computational basis states* [8].

A qubit is so important because it can store huge information [11]. It differs from a conventional digital bit in that it can store values 'intermediate' between 0 and 1 as shown in Fig. 2.1.

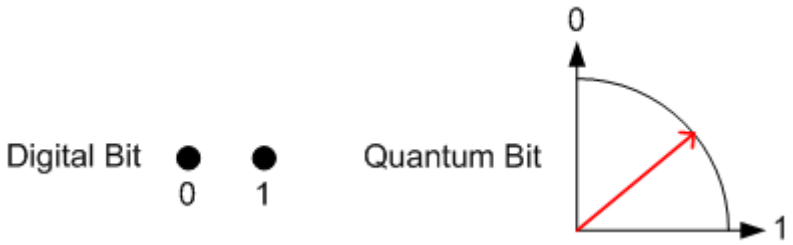


Figure 2.1: Digital bits and qubits

A qubit can be completely represented by a unit vector in a two dimensional Hilbert space. It is also possible to form *linear combinations* of states, often called *superposition*:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle. \qquad (2.1)$$

where α and β are complex numbers and $|\psi\rangle$ is the quantum state of qubit ψ which satisfies that

$$|\alpha|^2 + |\beta|^2 = 1. \tag{2.2}$$

The quantum states $|0\rangle$ and $|1\rangle$ can be measured with the probability $|\alpha|^2$ and $|\beta|^2$ respectively. This means that a qubit can store a superposition of the states $|0\rangle$ and $|1\rangle$ unlike a conventional bit.

2.1.2 Single Qubit Gates

The quantum NOT gate (NOT gate) is the simplest single qubit gate. The process of NOT gate takes state $|0\rangle$ to state $|1\rangle$, and vice versa. In fact, the NOT gate acts linearly, that is, take this state, $\alpha|0\rangle + \beta|1\rangle$, to the corresponding state, $\alpha|1\rangle + \beta|0\rangle$, in which the role of $|0\rangle$ and $|1\rangle$ have been interchanged.

Quantum NOT gate is linear and can be described by a matrix form conveniently. Let X be the matrix of an quantum Not Gate:

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{2.3}$$

If the quantum state $\alpha|0\rangle + \beta|1\rangle$ is written in a vector notation as

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}. \tag{2.4}$$

The corresponding output from the quantum NOT gate is

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}. \tag{2.5}$$

Other possible single qubit gates such as Hadamard, rotation, phase shift gates [8,

12, 13] are not used and thus not discussed in this thesis.

2.1.3 Multiple Qubit Gates

In classical gates, any function can be realized by NAND gates alone, which is thus known as a *universal* gate. In quantum Boolean circuits, any multiple qubit logic can be composed from *controlled-NOT* (CNOT) type logic gates.

CNOT type logic gates is denoted by [t, C], where t is an integer and C is a finite set of integers ($t \notin C$). $|x_t\rangle$ is called a *target bit* and $|x_k\rangle$ is called a *control bit* if $k \in C$.

This kind of gates is also called n-bit Toffoli gates in *reversible computing*. Also the literature refers to [t, C] with only one/two control bit(s) as a Controlled-NOT gate and Controlled-Controlled-NOT gate, respectively [6].

A controlled-NOT gate, shown in Fig. 2.2, has two qubits: one control bit ($|a\rangle$) and one target bit ($|b\rangle$).

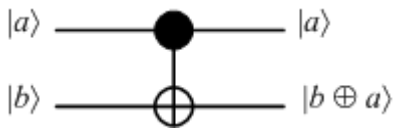


Figure 2.2: The circuit representation of controlled-Not gate

The top line represents the control qubit and the bottom line represents the target qubit. If the control qubit is set to 0, the target gate is unchanged. And, if the control qubit is set to 1, the target qubit is flipped. Thus when applying input pattern $|00\rangle$ to

the CNOT gate the output state is $|00\rangle$ denoted as $|00\rangle \rightarrow |00\rangle$. Some other possible transformations are

$$|01\rangle \rightarrow |01\rangle; |10\rangle \rightarrow |11\rangle; |11\rangle \rightarrow |10\rangle. \tag{2.6}$$

Summarily, the function of CNOT gates can map to $|a\rangle|0\rangle \rightarrow |a\rangle|a\rangle$ and $|a\rangle|1\rangle \rightarrow |a\rangle|\mathrm{NOT}(a)\rangle$.

The CNOT gate can be regarded as the classical XOR gate since the action of the gate can be summarized as

$$|a, b\rangle \rightarrow |a, b \oplus a\rangle, \tag{2.7}$$

where $\oplus$ is *addition modulo-two*, which is exactly what the XOR gate does [8].

In order to construct quantum Boolean circuits, any classical circuit can be replaced by an equivalent circuit only reversible element, by making use of a reversible gate known as the *Toffoli gate* [8].

The Toffoli gate, shown in Fig. 2.3, has three qubits. The third qubit is the target qubit which is flipped when both control qubits (i.e. the first two qubits) are set to 1.

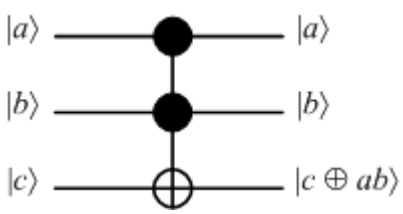


Figure 2.3: The circuit representation of Toffoli gate

The action of the Toffoli gate can be summarized as

$$|a, b, c\rangle \rightarrow |a, b, c \oplus ab\rangle. \tag{2.8}$$

Furthermore, applying the Toffoli gate twice has the effect $|a, b, c\rangle \rightarrow |a, b, c \oplus ab\rangle \rightarrow |a, b, c\rangle$, and thus the Toffoli gate is reversible.

Other possible multiple qubit gates such as Control-unitary and Control-Control-unitary gates [8, 12, 13] are not used in our synthesis algorithm and thus not discussed in this thesis.

2.1.4 Quantum Boolean Circuits

In quantum computing, the behaviors of a quantum circuit are represented by a sequence of unitary operations applied to the qubits of the quantum circuit. The results can be read out by measuring the quantum states of the qubits. That is quantum circuits consist of a sequence of unitary operations represented by quantum gates and quantum wires.

Figure 2.4 shows a quantum Boolean circuit with N qubits, denoted by $|\chi_1\rangle|\chi_2\rangle\ldots|\chi_N\rangle$. The sequence of unitary operations are applied from left to right to corresponding qubits $|\chi_1|$ to $|\chi_N|$. $|\chi_1\rangle|\chi_2\rangle\ldots|\chi_N\rangle$ on the left side is regarded as the input to the quantum circuit, and the states on the right side keep the final result.

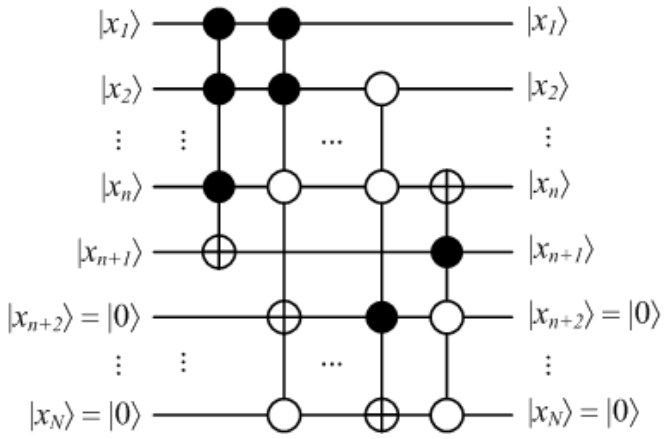


Figure 2.4: A quantum Boolean circuit

Taking the first operation as an example, the $(n + 1)$-st qubit $|\chi_{n+1}\rangle$ is called a work bit and is changed into $|\chi_{n+1} \oplus f(\chi_1, \ldots, \chi_n)\rangle$, which is used for obtaining the value of Boolean function f. Furthermore, qubits $|\chi_{n+2}\rangle|\chi_{n+3}\rangle\ldots|\chi_N\rangle$ are auxiliary qubits used for storing intermediate states. A quantum Boolean circuit can use any finite number of auxiliary qubits [6].

There are three different kinds of logic gates based on Toffoli gates: one-active controlled gates (denoted by *closed circles*), zero-active controlled gates (denoted by *open circles*) and target gates which are similar to sum (*mod 2*). When all controlled gates in the same wire (i.e. quantum operation) are active, the target gate flips [8].

2.2 Self-Timed Systems

The operations of a quantum Boolean circuit are quite different from those of a classical synchronous circuit which are controlled by a global clock. In quantum

Boolean circuits, a sequence of operations are applied to the qubits and are not controlled by a global clock. An operation can not be applied to a QBC unless the previous operation is complete and the quantum system is stable. This behavior is similar to the fundamental mode of asynchronous circuits.

In the fundamental mode of asynchronous circuits, the input changes can not occur unless the system reaches a stable state. Thus this thesis exploits the specifications of asynchronous circuit design to construct sequential quantum Boolean circuits.

Figure 2.5 shows an asynchronous sequential state machine in fundamental mode [9]. When the inputs of logic block are triggered, outputs are changed by the inputs and current states, and the next states of circuits are stored in the latches. The changes in inputs are forbidden until the system is stable.

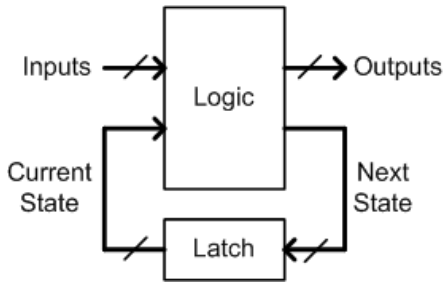


Figure 2.5: Fundamental mode of asynchronous circuits

Asynchronous circuits have no global clock [4]. Local handshaking signals among asynchronous components are used to synchronize operations. Figure 2.6

shows an example of the communication of two asynchronous components.

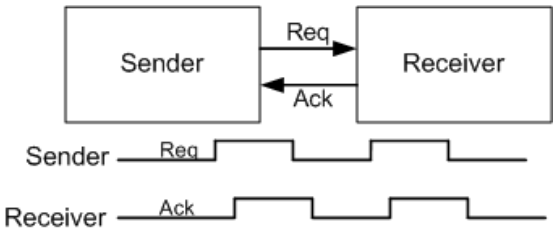


Figure 2.6: Communication of Asynchronous circuits

A request (req) signal from the sender is sent to the receiver to initiate an action and the receiver responses an acknowledgment (ack) signal when this action is complete.

Asynchronous circuits can also operate in input-output mode [9]. For more information on this topic please refer to [9, 10].

2.2.1 Trace Theory for Self-Timed Systems

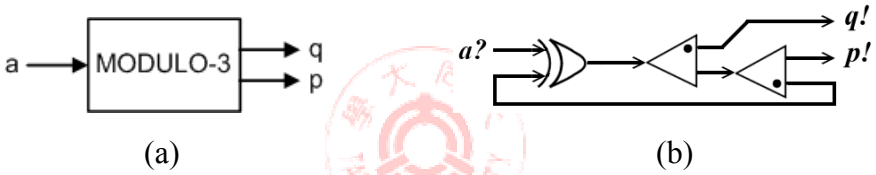
In trace theory, symbols represent events and traces represent behaviors. An event is an occurrence of the corresponding action which is usually the signal transition. '?' or '!' is appended to a symbol-name to denote an input or output action respectively.

The applicable operations in trace theory are as follows:

- 'Pref' is prefix-closure,
- ';' is sequential composition,

- '||' is parallel composition,
- '|' is non-deterministic choice, and
- '*' is Kleene-closure or repetition.

For example, a modulo-3 element [7], shown in Fig 2.7(a), has one input *a* and two outputs, *p* and *q*. The specification of the module-3 in prefix-closure form is *Pref*(*a*?*q*!*a*?*q*!*a*?*p*!)*. That is output *p* is triggered when input *a* is triggered three times. Valid partial behaviors are: *a*, *a q*, *a q a*, *a q a q*, *a q a q a*, *a q a q a p* …



(a) (b)

Figure 2.7: (a) The block diagram and (b) the classic circuit implementation of a modulo-3

The circuit behavior is an interleaving of input and output events in self-timed systems. The formal specification of a circuit can be described by trace theory [5, 7] and this thesis uses trace theory to specify the functionality of a circuit.

2.2.2 State Graph

Trace theory are used to specify the functionality of a circuit while state graphs [3, 4] are used to specify the behaviors of the circuit. State graphs are directed binary coded graphs containing states (or nodes) and directed edges.

Given a set of input or output symbols $J = \{j_1, j_2, \ldots, j_n\}$, a state machine (state graph), M, is define as a 4-tuple $M = \{S, T, s0, NS\}$, where

- S is a finite set of states, defined as $S = \{s \mid s: J \leftarrow \{0, 1\}\}$; $\langle s(j_1), s(j_2), \ldots, s(j_n)\rangle$ is the binary vector of a given state s,
- $T = J \times \{+, -\}$ is a finite set of input or output transitions,
- $s0 \in S$ is a distinguished state known as the initial state,
- NS: $S \times T \rightarrow S$ is the next-state function, having the following property that $\forall s$, $s' \in T$, and $NS(s, t) = s'$,

  if $t == j+$ then $s(j) = 0$ and $s'(j) = 1$,

  if $t == j-$ then $s(j) = 1$ and $s'(j) = 0$.

An edge in SGs is labeled with input or output *signal transitions*. Each signal transition can be represented as x_i+ or x_i- for the rising ($0\rightarrow1$) or falling ($1\rightarrow0$) transition of signal x_i.

A node in SGs is used to describe one state of the circuit. Each state $s \in S$ is labeled with *binary code* $\langle s(1), s(2), \ldots, s(n)\rangle$, and the value of $s(i)$ is 0 or 1. The state binary code is formed by an input binary code and an output binary code. Suppose the circuit has m-bits input and n-bits output. The input and output binary codes of node i are defined as follows:

- $ib(i) \in \{0, 1\}^m$ is the *input binary code function*,

- ob(i) $\in \{0, 1\}^n$ is the *output binary code function*.

The state binary code of node i, sbc(i), can be defined as ib(i) + ob(i) where the symbol '+' denotes the concatenation. And, the k-th state bit of node i is denoted as sbc(i, k).

For example, the SG of the modulo-3, shown in Fig. 2.8, has 6 states and 6 edges. The state binary code of node 1 is "000" since the ib(1) is '0' (i.e. a=0) and ob(1) is "00" (i.e. p=0, q=0).

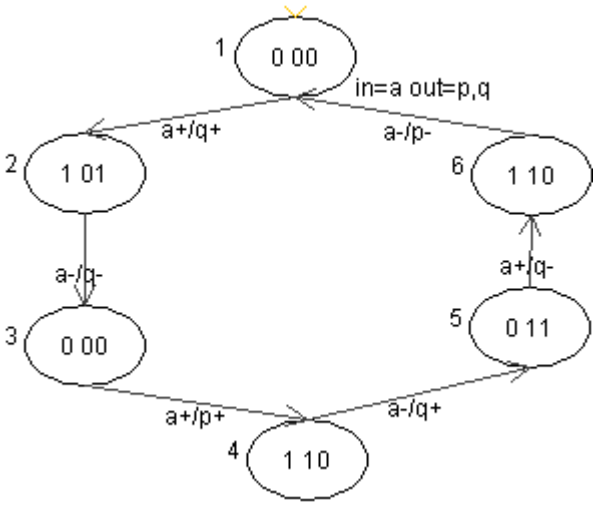


Figure 2.8: The SG of modulo-3

2.2.3 Complete State Coding

An unambiguous state assignment is required for deriving logic. This requirement is called *Complete State Coding* (CSC) [3]. A state graph has a CSC conflict if any two states in the state graph have the same state binary code. To be synthesizable a state graph specification of a circuit must satisfy the CSC requirement.

2.2.4 Basic Self-Timed Elements

A simple universal set of self-timed components [14] is introduced in this subsection. More complex self-timed systems can be composed by using only these basic components: fork, merge, join and toggle elements.

- Fork element: A fork element is simply a wire fan-out as shown in Fig 2.9. It is similar to the 'fan-out' gate in quantum circuits. The circuit specification of a fork is: *Pref*(*a*? (*b*! || *c*!))*.

(a) (b)

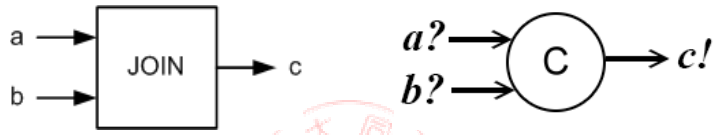
Figure 2.9: (a) The block diagram and (b) implementation of a fork element

- Merge element: A merge element is an 'OR' function of transition signals. A transition on either input causes a transition on the output. The implementation of the merge element is an 'Exclusive-OR' gate as shown in Fig 2.10. The circuit specification of a toggle is: *Pref*((*a*? | *b*?) *c*!)*.

(a) (b)

Figure 2.10: (a) The block diagram and (b) implementation of a merge element

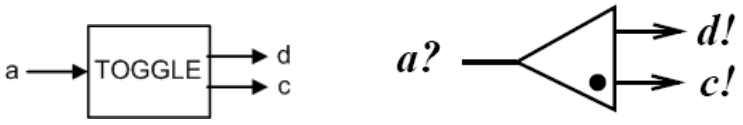
- Join element: A join element is an 'AND' function of transition signals. A transaction on all inputs will cause a transition on the output, otherwise the output is unchanged. A join element is implemented by Muller C-element [9] which has two input signals and one output signal. The symbol and implementation of a join element are shown in Fig 2.11. The circuit specification of a join is: *Pref*((*a*? || *b*?) *c*!)*.



(a) (b)

Figure 2.11: (a) The block diagram and (b) implementation of a join element

- Toggle element: A toggle element shown in Fig 2.12 is used to trigger two output transition signals alternatively. It has one input *a* and two outputs, *c* and *d*. The odd/even number of transition on input *a* causes a transition on the output *c/d* respectively. The circuit specification of a toggle is *Pref*(*a*?*c*!*a*?*d*!)*.



(a) (b)

Figure 2.12: (a) The block diagram and (b) implementation of a toggle element

Chapter 3.

Synthesis of Combinational Quantum Boolean Circuits

Tsai and Kuo [1] proposed a methodology to synthesize combinational quantum Boolean circuits. *Classical Transformation Tables* (i.e. a kind of truth tables) for describing the behavior of combinational circuits are used as the input specifications.

An *m*-to-*n* bit combinational circuit can be specified by a class transformation table containing two parts, a 2^m-by-*m* α table for input, and a 2^m-by-*n* β table for output. $\alpha[i][*]$ is used to denote the *i*-th row and $\alpha[i][j:k]$ to denote the *i*-th row with column index starting from *j* to *k*. β table is defined similarly. Each row of the α table, $\alpha[i][*]$, contains an *m*-bit input pattern, while the same row of the β table, holds the corresponding *n*-bit output.

Taking a half-adder as an example, it has two inputs (*a*, *b*) and two outputs (*s*, *c*). The specification of the half-adder can be described by the α table and the β table shown in Fig. 3.1. The α table and the β table can be regarded as the input and output parts of the truth table of the half-adder, respectively.

b	***a***
0	0
0	1
1	0
1	1
α	

c	***s***
0	0
0	1
0	1
1	0
β	

Figure 3.1: The truth-table of half-adder

A *Quantum Transformation Table* is used to describe a *t*-bit quantum Boolean logic. A quantum transformation table consists of two parts, a 2^t-by-t φ table for input, and a ϕ table of the same size for output. Each row of the φ table, $\varphi[i][*]$, contains an *t*-bit input pattern, while the same row of the ϕ table, holds the corresponding *t*-bit output.

The methodology of synthesizing combinational QBC based on transformation tables contains the following steps:

Step I. Preserve the input qubits, if necessary

The *preserved* qubits are defined to be the input qubits that have to stay unchanged after the operation, while the *volatile* qubits are the input qubits that can be over-written by the output qubits. Assume qubits 0 to $p - 1$ ($0 \le p \le m$) are the qubits to be preserved and qubits p to $m - 1$ are the volatile qubits. Now prepare two empty tables, φ and ϕ, which are both of size 2^m-by-m. For each row i ($0 \le i \le 2^m - 1$), copy

$\alpha[i][0 : m - 1]$ to $\varphi[i][0 : m - 1]$. If $p \neq 0$, also copy the preserved qubits from $\alpha[i][0 : p - 1]$ to $\phi[i][0 : p - 1]$.

Taking the above half-adder as an example, the qubit x and qubit y of ϕ table are corresponding to the input a and input b of α table respectively. Assume the qubit x is to be preserved and the result is shown in Fig. 3.2(a).

y	*x*
0	0
0	1
1	0
1	1
φ	

	x
	0
	1
	0
	1
ϕ	

y	*x*
0	0
0	1
1	0
1	1
φ	

c	*s*	*x*
0	0	0
0	1	1
0	1	0
1	0	1
ϕ		

(a) Preserve input qubits (b) Add output qubits

a	*y*	*x*
0	0	0
0	0	1
0	1	0
0	1	1
φ		

c	*s*	*x*
0	0	0
0	1	1
0	1	0
1	0	1
ϕ		

(c) Transformation table

Figure 3.2: The quantum transformation table of half-adder

Step II. Assign the output qubits

Since qubits 0 to p - 1 are used to preserve the input qubits, assign qubits p to $p + n - 1$ to hold the output qubits. Expand the width of the ϕ table whenever needed. For each row i ($0 \leq i \leq 2^m - 1$), copy $\beta[i][0 : n - 1]$ to $\phi[i][p : p + n - 1]$.

Taking the half-adder as an example, according to β table, the qubit c and s are put in ϕ table for assigning the output qubits and the result is shown in Fig. 3.2(b).

Step III. Distinguish each output state

For a unitary quantum evolution, the quantum transformation table needs to be one-to-one and onto. If for every $a, b \in \{0, 1\}^{p+n}$ in ϕ table, $a \neq b$, then set $d = 0$, go to next step. Otherwise, set

$$d = \lceil \log_2 M \rceil \tag{3.1}$$

where M is the maximum number of occurrences for a repeat pattern and add extra d columns to the ϕ table.

In the half-adder example, no extra columns are added since all entries in the ϕ table are different.

Step IV. Add auxiliary qubits

If $m = p + n + d$, no auxiliary qubit is needed, go to next step. Otherwise, if $m < p + n + d$, let $e = (p + n + d) - m$ and add e auxiliary qubits to the φ table. Assign these qubits to be all 0's. In both cases the total number of qubits, t, equals $p + n + d$.

Taking the above half-adder as an example, an auxiliary qubit a is added to the φ table. The resulting quantum transformation table is shown in Fig. 3.2(c).

Step V. Expand the quantum transformation table

If auxiliary qubits are used, expand both φ and ϕ tables to be 2^t rows in length.

For φ table, repeat the original block 2^e times, and for each block, fill the auxiliary qubits with a unique e-bit pattern. For ϕ table, leave the new entries blank. Thus the expanded result is shown in Fig. 3.3(a).

a	*y*	*x*
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
φ		

c	*s*	*x*
0	0	0
0	1	1
0	1	0
1	0	1
ϕ		

(a)

a	*y*	*x*
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
φ		

c	*S*	*x*
0	0	0
0	1	1
0	1	0
1	0	1
1	0	0
1	1	1
1	1	0
0	0	1
ϕ		

(b)

Figure 3.3: The (a) expanded (b) completed quantum transformation table

Based on the constraints derived from the classical Boolean circuits, the quantum transformation table is now partially constructed.

Step VI. Find possible permutations

Since quantum operations are reversible, the φ and ϕ tables must be one-to-one and onto mapping. This problem can be modeled as permutation finding problem.

Proposition I. Given a general cycle $C = (p_0, p_1, p_2, \ldots, p_{n-1})$, C can be constructed using $n - 1$ transpositions, i.e. $C = (p_0, p_1)(p_1, p_2)(p_2, p_3)\ldots(p_{n-3}, p_{n-2})(p_{n-2}, p_{n-1})$.

Proposition II. Given any two general states p and q, with $\Delta(p, q) = d$, the transposition $U = (p, q)$ can be decomposed into $2d - 1$ adjacent state decompositions.

A permutation consists of one or multiple disjoint cycles. Since disjoint cycles commute, each cycle in the permutation can be implemented individually. To do this, a *quantum state transformation digraph*, $G = (V, E)$, where

$$V = \{v_i \mid v_i = \varphi[i][*]\}$$

$$E = \{(v_s, v_d) \mid v_s = \varphi[i][*], v_d \cong \phi[i][*]\} \tag{3.2}$$

and $0 \le i \le 2^t - 1$. The digraph has 2^t vertices, corresponding to each of the 2^t rows in the φ table. An edge is defined from v_s to v_d if it is possible for permutation Q to map v_s to v_d. The notation $v_d \cong \phi[i][*]$ is used to denote, where only u ($u < t$) qubits are specified in $\phi[i][*]$, all states that are *compatible* to the entry. This results in 2^{t-u} edges to be generated for each of the possible (v_s, v_d) pairs. Filling in the $t - u$ blank qubits in the ϕ table selects one of the possible edges and delete others.

According the quantum state transformation digraph shown in Fig. 3.4, the resulting quantum transformation table is shown in Fig. 3.3(b).

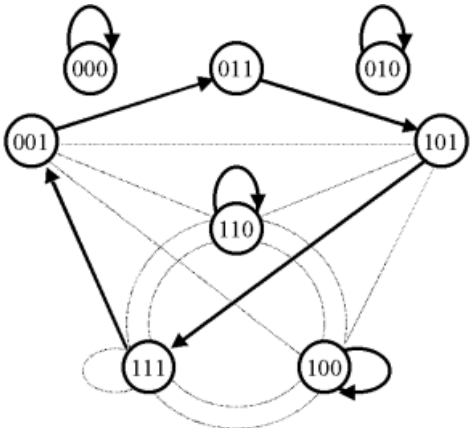


Figure 3.4: The quantum state transformation digraph of a half-adder

Step VII. Implement the permutation using elementary quantum gates

The generalized *n*-bit Toffoli gate [1, 8] is used to implement an adjacent state transposition. Assume *S* (i.e. set), *R* (i.e. reset), *I* (i.e. inversion) $\in \{0, 1\}^n$ and $S \wedge R = R \wedge I = S \wedge I = 0$, and the Hamming distance between *I* and $\{0\}^n$ is 1, an *n*-bit Toffoli gate can be represented by $T(S, R, I)$.

In the notation, *S* and *R*, if expressed in binary digits, mark the positions of control bits. The bits that are set in *S* specify the control bits have to be 1's to activate the logic. Similarly, the bits that are set in *R* specify the control bits have to be 0's to activate the logic. *I* simply represents the target bit to be inverted when the conditions of *S* and *R* are satisfied. The notation is used by the following proposition.

Proposition III. Given any two states s_1 and s_2 with $\Delta(s_1, s_2) = 1$, the transposition can be implement using a $T(S, R, I)$ gate with

$$S = s_1 \wedge s_2,\ R = \neg(s_1 \vee s_2),\ I = s_1 \oplus s_2. \tag{3.3}$$

Now the $T(S, R, I)$ gate can be further decomposed into one target gate and two control gates. This completes the quantum Boolean circuit construction.

Taking the half-adder as an example, the cycle can be represented as a permutation Q, $Q = (011, 001, 111, 101)$. The permutation Q can be implemented using three state transpositions thus the permutation Q becomes (011, 001)(001, 111)(111, 101). In the permutation Q, the non-adjacent state transposition, (001, 111), should be decomposed into adjacent state transpositions, (001, 011)(011, 111)(001, 011) and the permutation Q becomes (011, 001)(001, 011)(011, 111)(001, 011)(111, 101). Thus the quantum Boolean can be constructed using the Toffoli gates and the result is shown in Fig. 3.5.

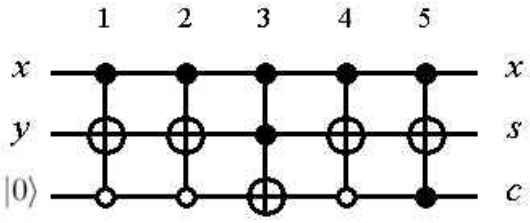


Figure 3.5: The quantum Boolean circuit of a half-adder

Chapter 4.

Synthesis of Sequential Quantum Boolean Circuits

For synthesizing sequential quantum Boolean circuits (SQBCs) state graphs are used to specify the behaviors of a circuit and are transferred to SQBCs automatically. The complete workflow of our synthesis methodology, shown in Fig. 4.1, consists of five main steps: First SGs (i.e. classic circuit specifications) are transformed to STTGs which are quantum circuit specifications. Second, STTGs are transformed to decomposed STTGs. Third, the STTGs can be optionally transformed into composable and thus reusable STTGs. Forth, the STTGs are synthesized into SQBCs based on Toffoli gates. Finally, the SQBCs are optimized based on reduction rules [1]. The first, second and forth steps are described in details in the section 4.1 and the fifth step is described in details in the section 4.2. The issues of CQBCs construction are given in detail in the chapter 5.

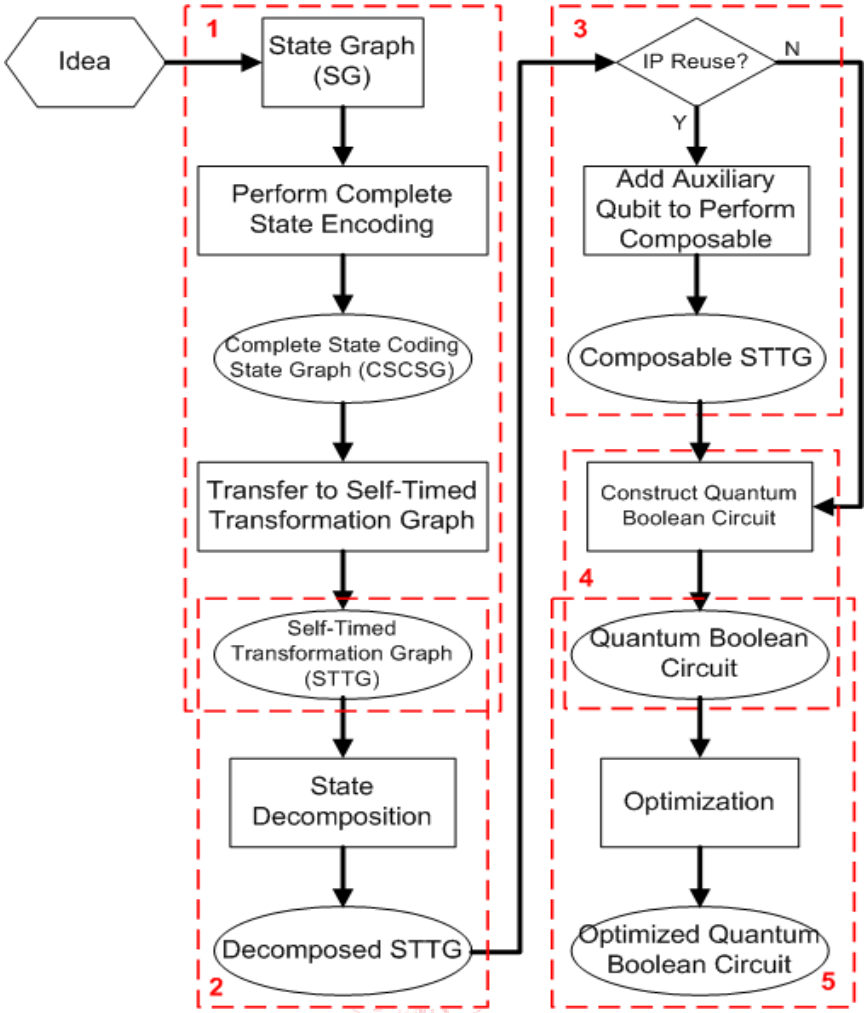


Figure 4.1: The workflow of synthesis of SQBCs

4.1 Transformation of Self-Timed Transformation Graph

For a state graph specifying a sequential circuit with *m*-bit input, *n*-bit output and *e* edges (i.e. *e* next-state functions), a self-timed transformation graph (STTG) corresponding the state graph has *e* transformation sub-graphs (TSG) and each TSG consists of two nodes connecting to each other by two direct edges. If the nodes in a TSG indicate the same state the sub-graph degenerates into a self-loop sub-graph. Thus a STTG is a hyper-graph.

The transformation from SGs to STTGs contains the following steps:

Step I. perform complete state encoding

Since an unambiguous state assignment is needed to construct the SQBCs, complete state encoding algorithm is applied first to avoid CSC conflict.

In order to satisfy the CSC requirement, different *state bits* are appended to the original state binary code to distinguish duplicated states in SGs. If there are *s* states in the SG and the number of duplicated states for each state binary code is *di* ($0 \le i < s$), then the number of state bits needed is

$$k = \log \lceil max(d_i) \rceil. \tag{4.1}$$

The algorithm of this step is presented in Algorithm 1.

For example, the state 1, 3 and state 4, 6 in the SG of modulo-3 of Fig. 2.9, have the same state binary codes. Thus one state bit is appended to each node in the SG and the state bit appended to states 1 and 6 is '0' and to states 3 and 4 is '1'. The unambiguous SG for modulo-3 is shown in Fig. 4.2 which has CSC property. Note that there are some other ways to form a CSCSG.

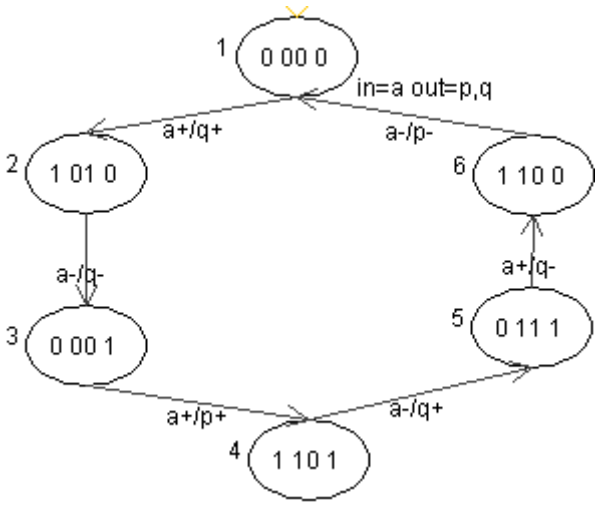


Figure 4.2: The CSCSG of modulo-3 example

Algorithm 1 Complete State encoding

Input: A state graph, SG

Output: A complete state coding SG, CSCSG

Remark: The state binary code of a node is defined as sbc(node).

Method:

```
  CompleteStateEncoding()
  {
    let duplicated-state-list be an empty list;
    md = 0; // the max number of duplicated states;
    sb = 0; // the number of state bits needed to be added;

    // find the max number of duplicated states
    for each node, n, in SG
    {
        collect all the states whose state binary code is equal to sbc(n) into a list;
      add the list to duplicated-state-list;
let di be the number of duplicated nodes for n;
      if (di > md)
          md = di;
    }
    sb = ⌈log(md)⌉;
    // assign different state binary code for the duplicated states
    for each list, l, in duplicate-state-list
    {
```

```
        append different state bits to state binary code of each node in l;
    }
    return SG;
}
```

Step II. Transfer CSCSG to the self-timed transformation graph (STTG)

Each edge in the STTG represents a transition from one state to another. For each edge *e* with source node *i* and target node *j* in a CSCSG a corresponding transformation sub-graph (TSG) is formed for the target node *j*. The TSG consists of two new nodes *source* and *target* connecting to each other by two direct edges (i.e. *source* → *target*, *target* → *source*). These two direct edges are called *quantum links* (marked in dashed lines). The state binary codes of the *source* and *target* nodes are labeled with ib(*j*) + ob(*i*) + sb(*i*) and ib(*j*) + ob(*j*) + sb(*j*), respectively. The function sb(*i*) is the *state bits binary code function* which is similar to ib(*i*) and ob(*i*) in the section 2.2.2. The algorithm of transforming CSCSGs to STTGs is presented in Algorithm 2.

For example, the CSCSG of the modulo-3, shown in Fig. 4.2, can be transferred to the STTG, shown in Fig. 4.3, which contains 6 TSGs. The dashed line connected two nodes in a TSG are the *quantum links*.

For the edge in node 1 and node 2 in Fig 4.2, a TSG is created for node 2. The TSG (see Fig 4.3) contains two nodes, *s* and *t*. The state binary code of the node *s* is

ib(node 2) + ob(node 1) + sb(node 1) = “1000” and the state binary code of the node *t* is ib(node 2) + ob(node 2) + sb(node 2) = “1010.”

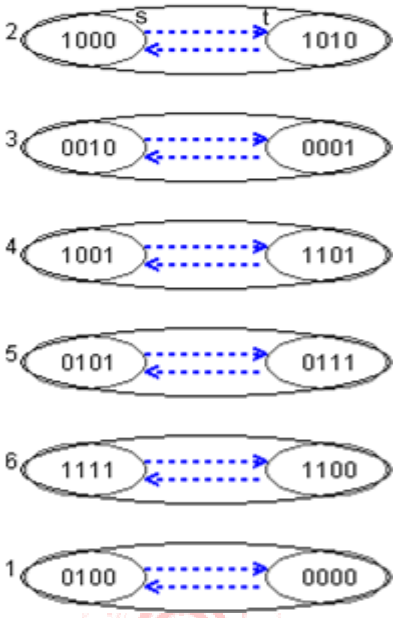


Figure 4.3: The STTG of a modulo-3

Algorithm 2 Transfer to self-timed transformation graph
Input: A complete state coding state graph, CSCSG
Output: A self-timed transformation graph, STTG
Remark: 1. A STTG is a hyper-graph containing many TSGs.
2. The state binary code of a node is defined as ib(node) + ob(node) + sb(node).
Method:

```
TransferToSelfTimedTransformationGraph()
{
  for each edge, e, in CSCSG
  {
    let i and j be the source and target nodes of the edge e, respectively;
    for target node, j, create a TSG with two nodes, s labeled with ib(j) + ob(i) + sb(i) and t
labeled with ib(j) + ob(j) + sb(j);
      create two quantum links for the s and t nodes;
  }
}
```

Step III. Perform State decomposition

In order to construct SQBCs based on Toffoli gates, the state binary codes of adjacent nodes in transformation sub-graphs must differ in only one bit [1]. This property can be retained by performing state decomposition.

If two nodes in a TSG have Hamming distances more than one, they have to be decomposed and some appropriate states are added between them so that any adjacent states differ only one bit. Furthermore, the added states must be never used in STTGs. The algorithm of state decomposition is shown in Algorithm 3.

For example, the state transposition of a TSG, shown in Fig. 4.4(a), contains "1000" and "1011" differ two bits. Two possible state decompositions are shown in Fig. 4.4(b) and Fig. 4.4(c). The states 1001 or 1010 could be added between 1000 and 1011 and the states of the TSG becomes (1000, 1001 / 1010, 1011).

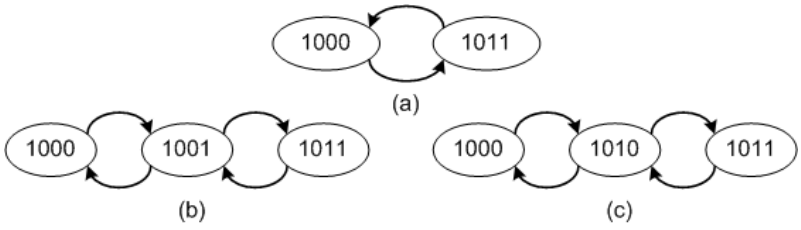


Figure 4.4: (a) A transformation sub-graph, (b) one possible decomposition and (c) the other possible decomposition

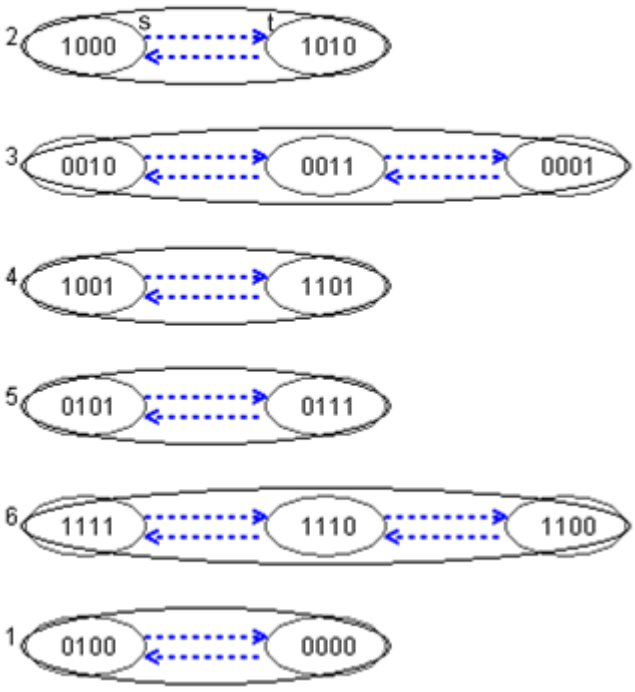


Figure 4.5: The decomposed STTG of a modulo-3

Taking the STTG of a modulo-3, shown in Fig. 4.3, as an example, the state decompositions of the 3rd and 6th TSGs have to be decomposed. The state transposition, (0010, 0001), of the 3rd TSG becomes (0010, 0011, 0001) and the state transposition, (1111, 1100), of 6th TSG becomes (1111, 1110, 1100). The complete STTG of modulo-3 is shown in Fig. 4.5.

Algorithm 3 Perform State Decomposition
Input: A self-timed transformation graph, STTG
Output: A decomposed STTG, DSTTG
Remark: 1.A STTG is a hyper-graph containing many TSGs.
2. The state binary code of a node is defined as sbc(node).
3. The Hamming distance of two codes is defined as H(code1, code2).
Method:
PerformStateDecomposition()

```
{
  for each TSG, tsg, in STTG
  {
    let s and t are the two nodes in tsg;
    hammingDistance = H(sbc(s), sbc(t));

    while (hammingDistance > 1)
    {
      find the unused adjacent node, n, with sbc(s) and sbc(t);
      insert n into tsg between s and t;
      hammingDistance = hammingDistance – 1;
    }
  }
  return STTG;
}
```

4.2 Construction of Quantum Boolean Circuit

Using the rule of the proposition III for implementing state transpositions, introduced in the chapter 3, one quantum wire (i.e. quantum operation) based on the Toffoli gates can be generated by each state transposition of TSGs. The algorithm of this step is shown in Algorithm 4.

Taking the first TSG of STTG, shown in Fig. 4.5, as an example, the states in the TSG are 1000 and 1010. $S = 1000 \wedge 1010 = 1000$ so the 1^{st} qubit uses the one-active controlled-gate. Similarly, since $R = \neg(1000 \vee 1010) = 0101$, the 2^{nd} and 4^{th} qubit use the zero-active controlled-gate. Finally, $I = 1000 \oplus 1010 = 0010$ so the 3^{rd} qubit uses the target gate. Therefore a quantum operation can be formed by the above logic gates in the QBC.

Finally, SQBC of a modulo-3 is constructed and shown in Fig. 4.6, where the qubit *a* is the input, the qubits *p* and *q* are the output and the qubit *s0* is the auxiliary qubit for assisting in the work of circuits.

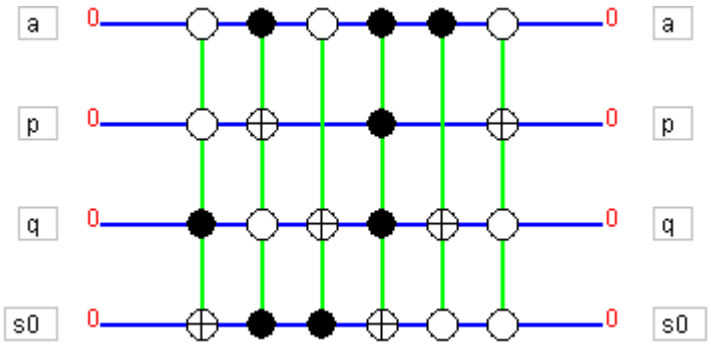


Figure 4.6: The SQBC of a modulo-3

Algorithm 4 Construct Quantum Boolean Circuit
Input: A decomposed STTG, DSTTG
Output: A quantum Boolean circuit, QBC
Remark: 1. QBC contains quantum operations.
2. A quantum operation contains logic gates.
3. A STTG is a hyper-graph containing many TSGs.
4. The state binary code of a node is defined as sbc(node).
5. The i-th bit of the state binary code of a node is defined as sbc(node, i).
6. The operator '+' between two state binary codes indicates the concatenation of codes.

Method:

```
ConstructQBC()
{
  let gate be an empty logic gate;

  for each TSG, tsg, in DSTTG
  {
    for each two consecutive nodes, s0 and s1, in tsg
    {
      create an empty quantum operation as op;
```

```
            for each bit index, i, in sbc(s0) and sbc(s1)
            {
                if (sbc(s0, i) + sbc(s1, i) == "00")
                    gate = one-active controlled gate;
                else if (sbc(s0, i) + sbc(s1, i) == "01" or "10")
                    gate = zero-active controlled gate;
                else if (sbc(s0, i) + sbc(s1, i) == "11")
                    gate = target gate;
                set gate to the i-th qubit of op;
            }
            push op to QBC;
        }
    }
    return QBC;
}
```

4.3 Optimization

The goal of the optimization of QBCs is to simplify and merge the logic gates and wires in QBCs and thus reduce the complexity of circuits. Two reduction rules [1] are used to optimize QBCs:

(1) For any two quantum operations in a QBC, if they are identical then they can be removed from the QBC.

(2) For any two quantum operations in a QBC, if they are identical except one qubit in which one of the logic gates is one-active controlled gate and the other is zero-active controlled gate then these two logic gates can be removed.

The algorithm of reduction rules is shown in Algorithm 5.

For example, the QBC shown in Fig. 4.7(a) contains 5 quantum operations and

15 logic gates. Block 1 of Fig. 4.7(a) contains two identical quantum operations except third qubit. Thus logic gates on the third qubit of block 1 can be removed and the result is shown in the block 1of Fig. 4.7(b). Block 2 of Fig. 4.7(a) contains two identical quantum operations. Thus these two quantum operations can be removed and the result is shown in the block 2 of Fig. 4.7(b). The optimized QBC contains only 2 quantum operations and 5 logic gates.

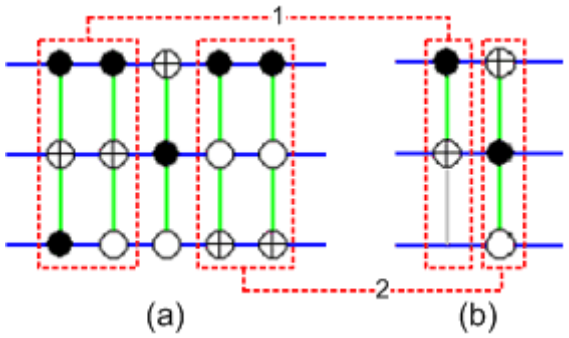


Figure 4.7: The (a) QBC and (b) Optimized QBC

Algorithm 5 *Optimization*
Input: A quantum Boolean circuit, QBC
Output: An optimized QBC, OQBC
Remark: 1. A QBC contains quantum operations.
2. A quantum operation contains logic gates.
Method:

```
Optimization()
{
  for any two quantum operations, op0 and op1, in QBC
  {
    compare logic gates one-by-one in op0 and op1;
    if (all logic gates of are the same)
      remove op0 and op1;
```

```
        else if (only one logic gate (controlled-gate) is different)
        {
          remove op0 or op1;
          remove the different logic gate of op0(op1);
        }
    }
    return QBC;
}
```

Chapter 5.

Synthesis of Composable Quantum Boolean Circuits

For classic circuits, an input changes may cause some output and state changes. When a circuit is stable the same input pattern reapplying again to the circuit can change neither output nor state signals.

This is not true in quantum circuits. Applying the same input pattern twice in quantum circuits may result in the different or wrong state because quantum operations are reversible. This makes QBCs not reusable and large quantum Boolean circuits cannot be composed by basic QBCs like classic circuits.

For example of Fig. 5.2 of a toggle element, initially the QBC is in the target node of state 1 with the state code “000.” If *t* is turned on then it goes to the target node of state 2 with the state code “110.” Now if *t* is applied one again then it goes to the source node of state 2 with state code “100.”

In order to solve the composable problem and make QBCs reusable, a new methodology for synthesizing composable QBCs (CQBCs) is proposed. These CQBCs can be exploited to construct a larger and more complicated QBC rapidly.

5.1 Construction of CQBCs

To make QBCs composable and reusable, only one extra auxiliary qubit added to

QBCs is sufficient. The idea is to reset the auxiliary qubit before applying operations so that the quantum system will not go to the wrong state.

To synthesize CQBCs, the composable STTGs are transferred from decomposed STTGs. An auxiliary qubit with initial value 0's is appended to each state binary code of all nodes in STTGs. An auxiliary node with sbc(the corresponding target node) is added for each TSG in a STTG and then set the auxiliary bit of the auxiliary node to 1. The new STTGs is called composable STTGs. The algorithm of generating composable STTG is shown in Algorithm 6.

The composable STTGs contains two operation blocks: a *core operation block* representing the same circuit behavior of non-composable QBCs and a *composable operation block* preventing going to a wrong state if repeated operations are applied. The core operation block is made by the quantum operations constructed with the source and target nodes and the composable operation block is made by the quantum operations with the target and auxiliary nodes.

The algorithm of constructing CQBCs is shown in Algorithm 7.

Algorithm 6 Transfer to Composable STTG
Input: A decomposed STTG, DSTTG
Output: A composable STTG, CSTTG
Remark: 1. A STTG is a hyper-graph containing many TSGs.
2. The state binary code of a node is defined as sbc(node).
Method:

```
TransferToComposableSTTG()
{
  for each TSG, tsg, in DSTTG
  {
    let s and t be the source and target nodes in tsg;
    append an auxiliary bit, aux, with 0's to all nodes in tsg;
    add a new auxiliary node, a, to tsg and sbc(a) = sbc(t);
      set the bit aux of sbc(a) to 1;
      create two quantum links from the nodes t/a to the nodes a/s, respectively;
  }
  return DSTTG;
}
```

Algorithm 7 Construct Composable Quantum Boolean Circuits

Input: A composable STTG, CSTTG

Output: A composable QBC, CQBC

Remark: 1. A STTG is a hyper-graph containing many TSGs.

2. A QBC contains quantum operations.

3. A TSG in CSTTG contains the source, target and auxiliary nodes.

Method:

```
ConstructComposableQBC()
{
  let composableQBC and coreQBC be the empty QBCs;
  split CSTTG into composable operation block STTG (STTG0) and core operation block
STTG (STTG1);
  composableQBC = ConstructQBC(STTG0);
  coreQBC = ConstructQBC(STTG1);
   CQBC = cascade composableQBC and coreQBC;
  return CQBC;
}
```

5.1.1 Example

For example, toggle [7, 10] has one input *t* and two outputs *a* and *b*. The circuit specification of a toggle is *Pref*(*t*?*a*!*t*?*b*!)*, and its SG is shown in Fig. 5.1. If the first

transition on *t* triggers the output signal *a* and the second transition on *t* triggers the output signal *b*. Thus a toggle element can trigger two transition signals alternatively.

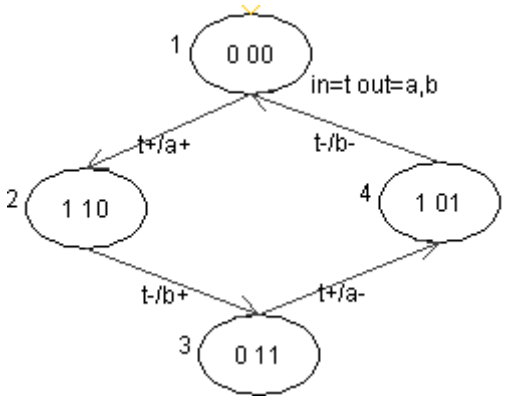


Figure 5.1: The SG of a toggle

After the steps introduced in the section 4.1 and 4.2, the STTG of a toggle transferred from the SG is shown in Fig. 5.2.

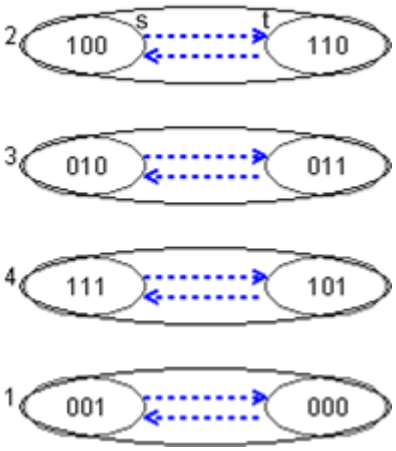


Figure 5.2: STTG of a toggle

The STTG of a toggle is transferred by the composable synthesis methodology proposed in the section 5.1 and the resulting composable STTG is shown in Fig. 5.3.

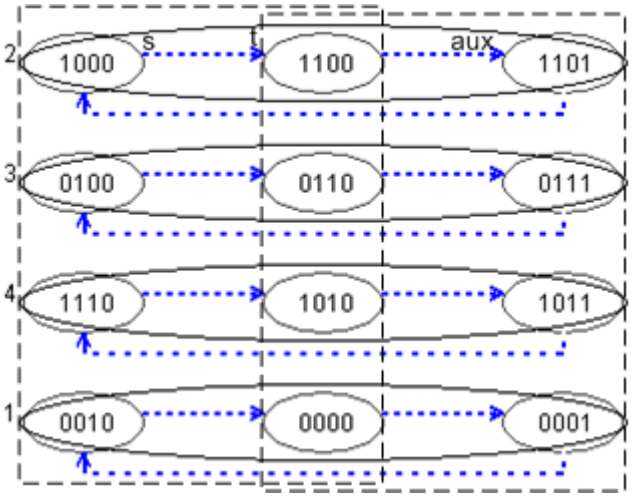


Figure 5.3: The composable STTG of a toggle

The CSTTG, shown in Fig. 5.3, can be split into a core operation block STTG (STTG0) which is composed by the source and target nodes and a composable operation block STTG (STTG1) which is composed by the target and auxiliary nodes. Thus by using Algorithm 4 a core QBC and composable QBC are constructed. A composable QBC is generated by cascading STTG1 with STTG0. The resulting CQBC of the toggle through optimizations is shown in Fig. 5.4 and the composable operation block and core operation block are from the 0^{th} to 3^{rd} and 4^{th} to 5^{th} quantum operations, respectively.

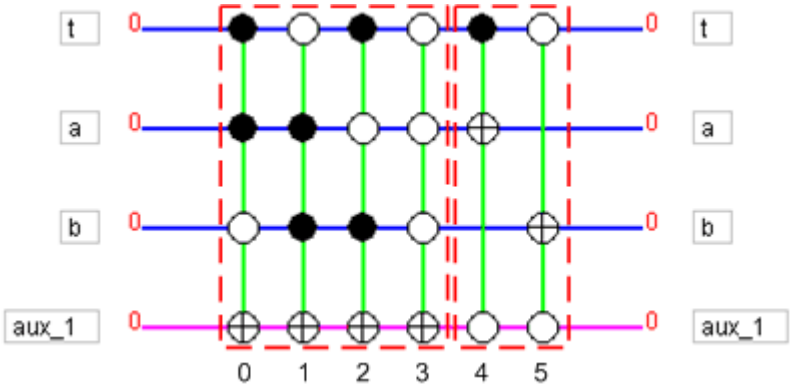


Figure 5.4: The CQBC of a toggle

The composable operation block is used to present go the wrong state when an input is applied to a QBC twice. For the CQBC in Fig. 24, initially all input, output and auxiliary qubits are set to zero. Suppose the input *t* is turned on, the system will go to the state with state binary code "1101." If "1101" is applied again, the system will go to the state with state binary code "1000." Now if the input *t* is turned off, the system ends up with the state with state binary code "0001" which is a wrong state. Since the auxiliary qubits are to prevent the system goes to the wrong state, the correct operation is to reset the auxiliary qubits when applying any input pattern to the system. Thus the correct input for the system is "1100" (by resetting the fourth qubit) and the system stays in the state with state binary code "1101."

5.2 Composition of Composable Quantum Boolean Circuits

Once CQBCs are synthesized by the above methodology they can be exploited to construct a larger quantum Boolean circuit rapidly like classic circuits.

For example, mod-4 counter can be constructed by two composable toggle QBCS shown in Fig. 5.5. Figure 5.5 (a) and (b) shows the classic and quantum Toggle components, respectively.

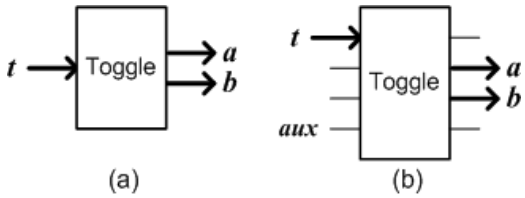


Figure 5.5: The block diagram of a toggle in (a) classical version (b) CQBC version

The circuit implementation of classical mod-4 counter can be directly composed by two toggle elements, shown in Fig. 5.6(b). The quantum version of mod-4 counter can be constructed similarly, shown in Fig. 5.6(b).

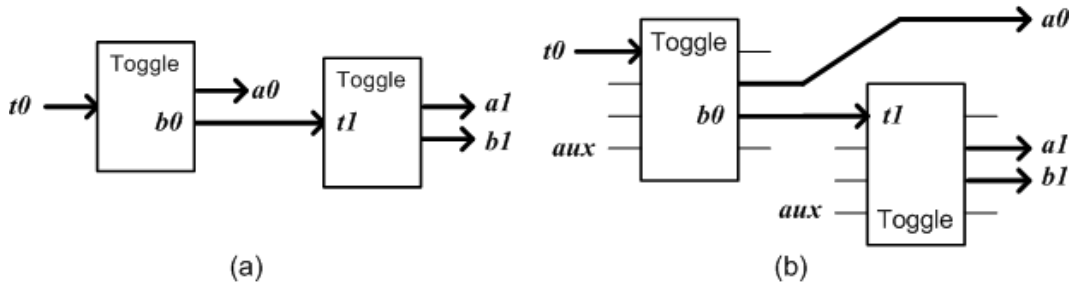


Figure 5.6: The circuit implementation of a mod-4 counter in (a) classical version and (b) CQBC version

Figure 5.7 shows the QBC of mod-4 counter. If the output (i.e. *b0*) of first toggle element and the input (i.e. *t1*) of second toggle element are connected, they share the same qubit (i.e. *b0*/*t1*) in the CQBC. The CQBC of the mod-4 counter has 7 qubits, 12

quantum operations and 44 logic gates. Furthermore, mod-4 counter is composable since the toggle element is composable.

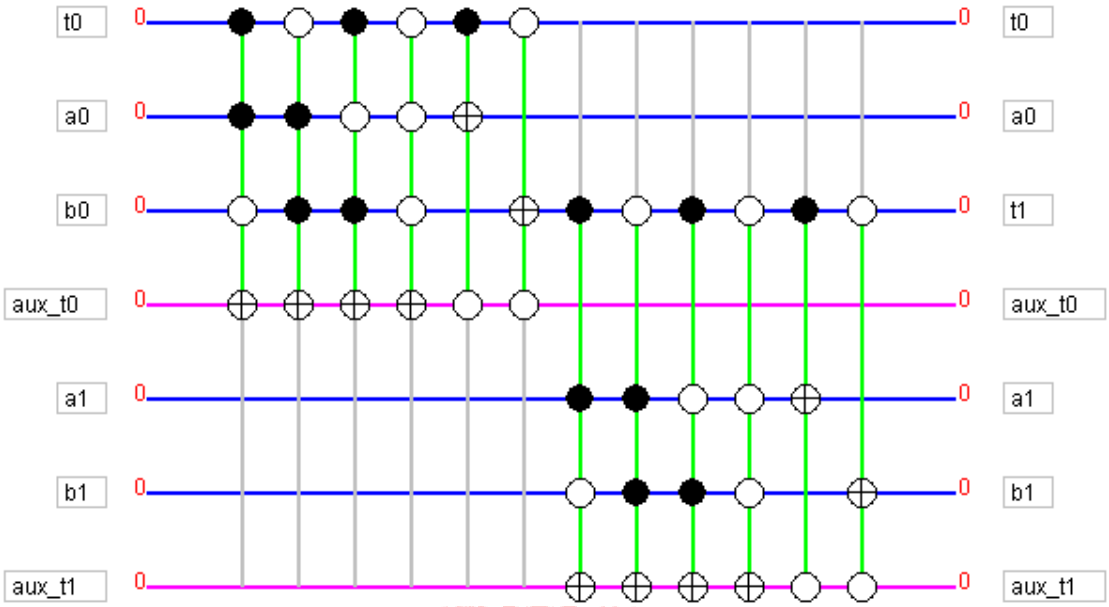


Figure 5.7: The QBC of a mod-4 counter

Chapter 6.

Experimental Results

A simple universal set of self-timed components [14] is used to test our CAD tool. The results are shown in Table 1. Columns 2 and 3 (i.e. *#N* and *#E*) are the numbers of nodes (states) and edges (transitions) of SGs, respectively. Column 4 shows the number of total qubits required for the QBC containing input qubits, output qubits and auxiliary qubits. Columns 5 and 6 show the number of quantum operations and logic gates of the synthesized QBC and the optimized QBC, respectively. The module-3 example is also shown in the row 6. The results of Table 1 show that the optimization algorithm can reduce a great deal of the numbers of quantum operations and logic gates for the Fork, Merge and Toggle elements.

Table 6.1: The result of SQBC synthesis

Circuit	***#N***	***#E***	***Qubit***			***Quantum Operation***		***Logic Gate***	
			I	***O***	***Aux***	***Original***	***Optimized***	***Original***	***Optimized***
Fork	2	2	1	1	0	2	1	4	1
Merge	4	8	2	1	0	4	1	12	1
Join	6	12	2	1	1	2	2	8	8
Toggle	4	4	1	2	0	4	2	12	4
Modulo-3	6	6	1	2	1	8	6	32	22

Table 6.2 shows the results of composable QBC synthesis.

Table 6.2: The result of CQBC synthesis

Circuit	***Qubit***			***Quantum Operation***		***Logic Gate***	
	I	***O***	***Aux***	***Original***	***Optimized***	***Original***	***Optimized***
Fork	1	1	1	4	3	12	8
Merge	2	1	1	8	5	32	18
Join	2	1	2	4	4	20	20
Toggle	1	2	1	8	6	32	22
Modulo-3	1	2	2	14	10	70	46
Modulo-4	1	2	2	12	12	44	44

Since CQBCs have an additional composable operation block the number of quantum operations of CQBCs is much larger than the non reusable QBCs. This is because the optimization algorithm can not reduce any operations or logic gates in the composable operation block.

Chapter 7.

CAD Tool Overview

The CAD tool to automatically synthesize SQBCs and CQBCs is implemented in Java programming language based the methodologies proposed in chapters 4 and 5. Our CAD tool implements the following functions: translation from SGs to corresponding SQBCs or CQBCs, design entry for SGs and QBCs, optimization for QBCs and simulation of QBCs.

The workflow of our CAD tool is shown in Fig. 7.1 and it consists of four major subsystems:

1. *Design Entry*: entry design for both and QBC.
2. *Transformation*: Transformations of SGs to self-timed transformation graphs (STTGs), STTGs to decomposed STTG or composable STTG and constructions of QBCs.
3. *Optimization*: optimized QBCs for less qubits, logic gates and/or wires.
4. *Simulation*: testing and simulation of QBCs.

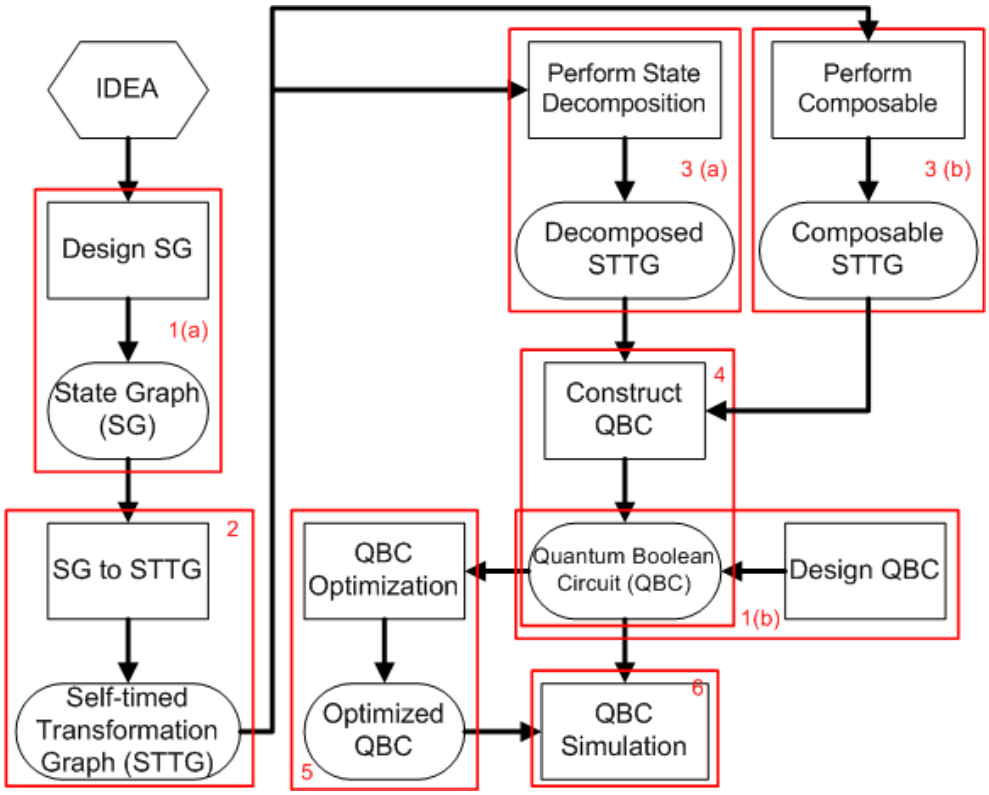


Figure 7.1: The overview of CAD tool

7.1 State Graph Design Entry

The SG design entry facilitates editing of state graph. The components (e.g. nodes and edges) in SGs can be placed, moved and labeled easily. Besides, this tool can decide that if this circuit is composable and it will be transferred to the corresponding QBCs. The GUI of CAD tool is shown in Fig. 7.2.

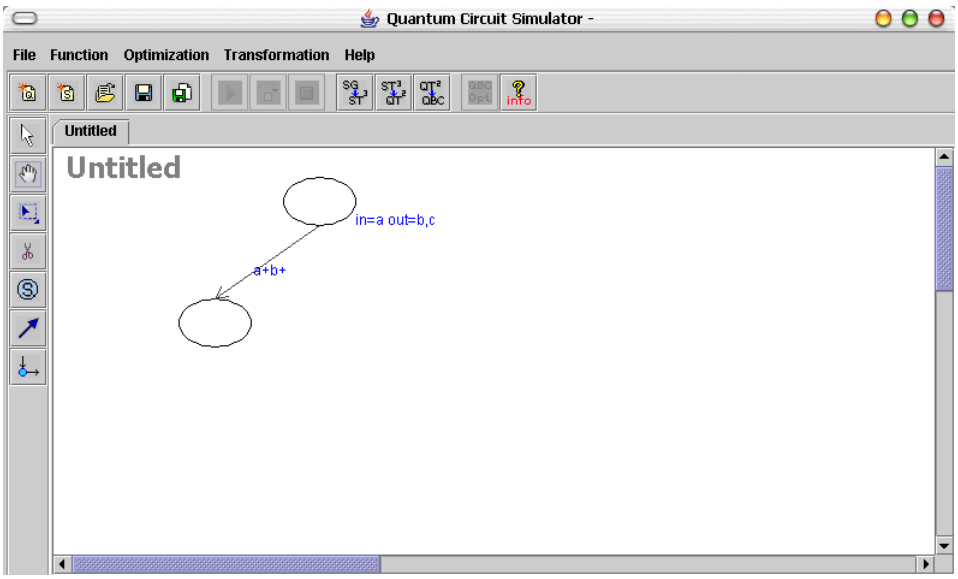


Figure 7.2: The state graph design entry

7.2 QBC Layout Design Entry

The QBC layout design entry helps designers to construct the quantum Boolean circuit layout easily. Qubits, quantum operations, and logic gates (e.g. controlled gates and target gates) can be placed, labeled and configured attributes, such as the state of qubits, auxiliary qubits or not and etc. The GUI of CAD tool is shown in Fig. 7.3.

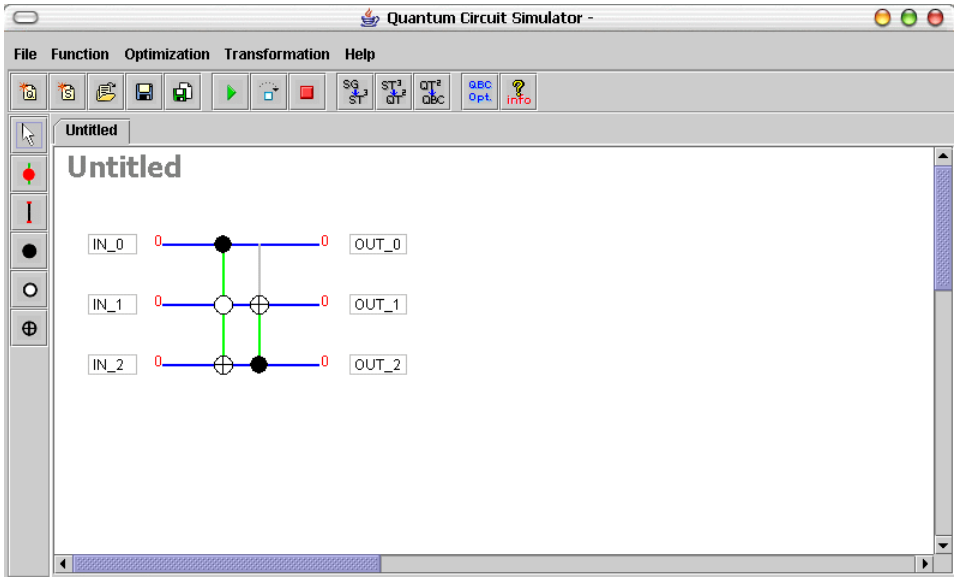


Figure 7.3: The quantum Boolean circuit layout design entry

7.3 State Graph to Self-Timed Transformation Graph

SG to STTG transformation tool transforms state graphs to decomposed or composable STTGs. For example, the SG of a toggle, shown in Fig. 5.1, is transform to the decomposed STTG in Fig. 7.4 or composable STTG in Fig. 5.3.

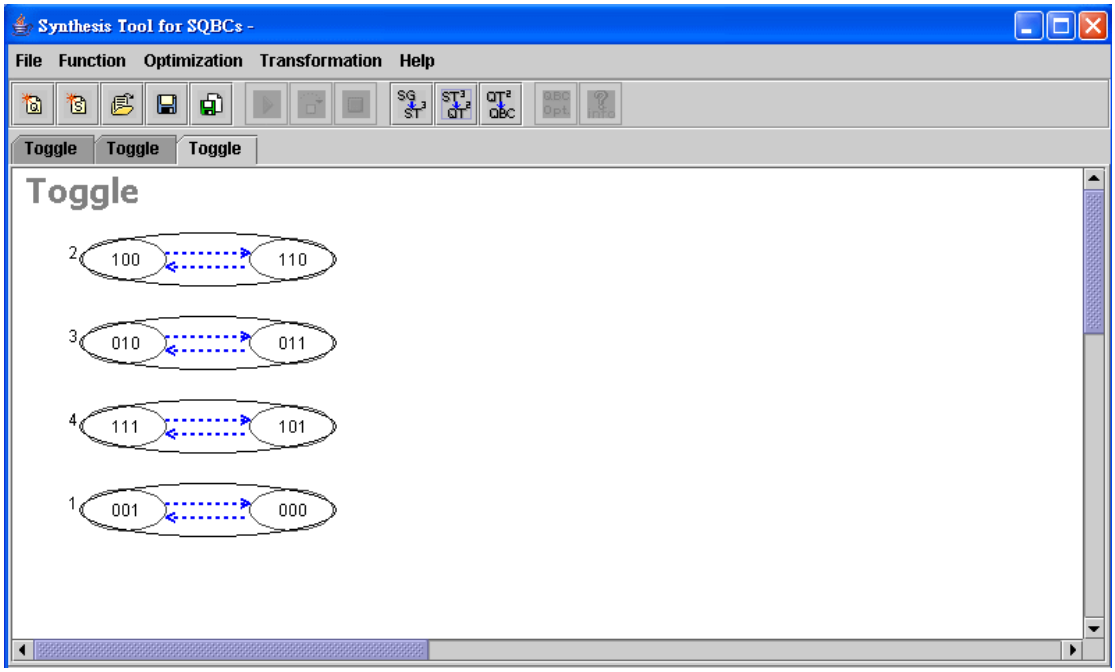


Figure 7.4: The decomposed STTG of a toggle

7.4 Self-Timed Transformation Graph to Quantum Boolean Circuit Construction

STTG to QBC transformation tool transforms STTGs to corresponding quantum Boolean circuits. For example, the decomposed STTG of a toggle, shown in Fig. 7.4, is transformed to the SQBC in Fig. 7.5 or the composable STTG, shown in Fig. 5.3, is transformed the CQBC in Fig. 5.4.

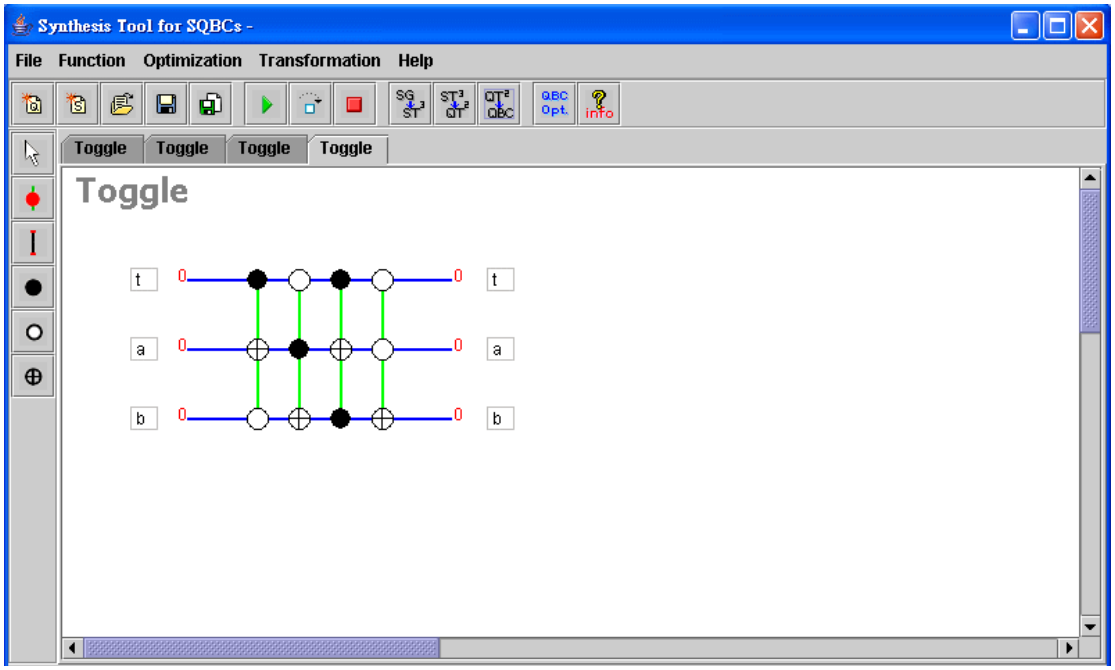


Figure 7.5: The SQBC of a toggle

7.5 Quantum Boolean Circuit Optimization

The optimization tool can take a quantum Boolean circuit and simplify and merge their quantum operations and logic gates in an optimized QBC. For example, the QBC of a toggle, shown in Fig. 7.5, is optimized by our CAD tool and the optimized result is shown in Fig. 7.6.

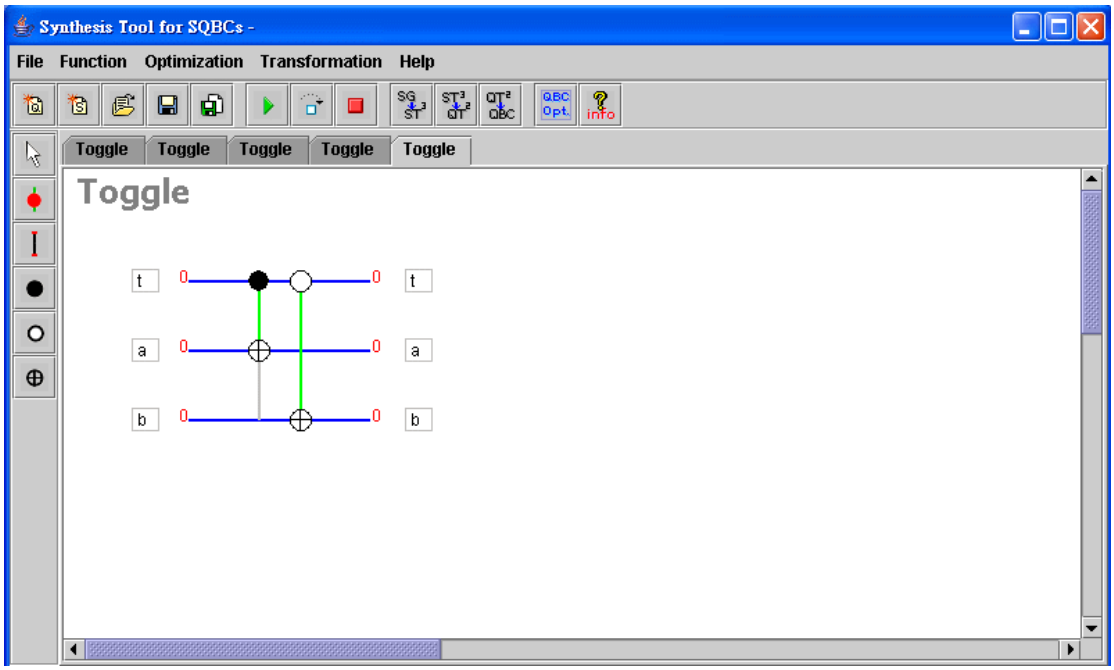


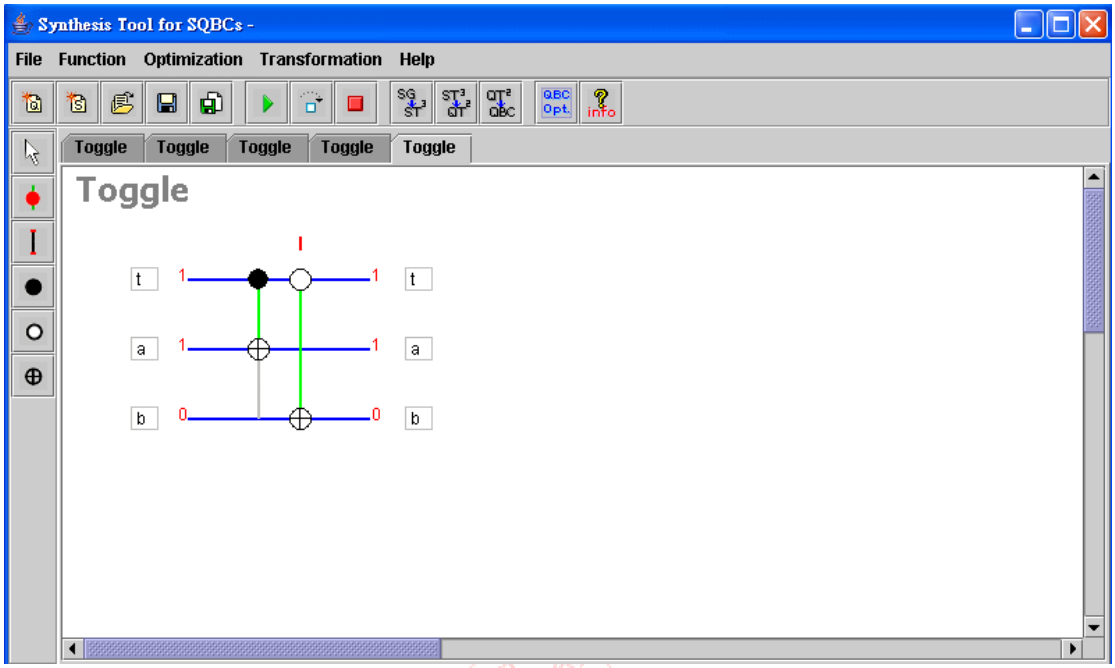
Figure 7.6: The optimized QBC of a toggle

7.6 Quantum Boolean Circuit Simulation

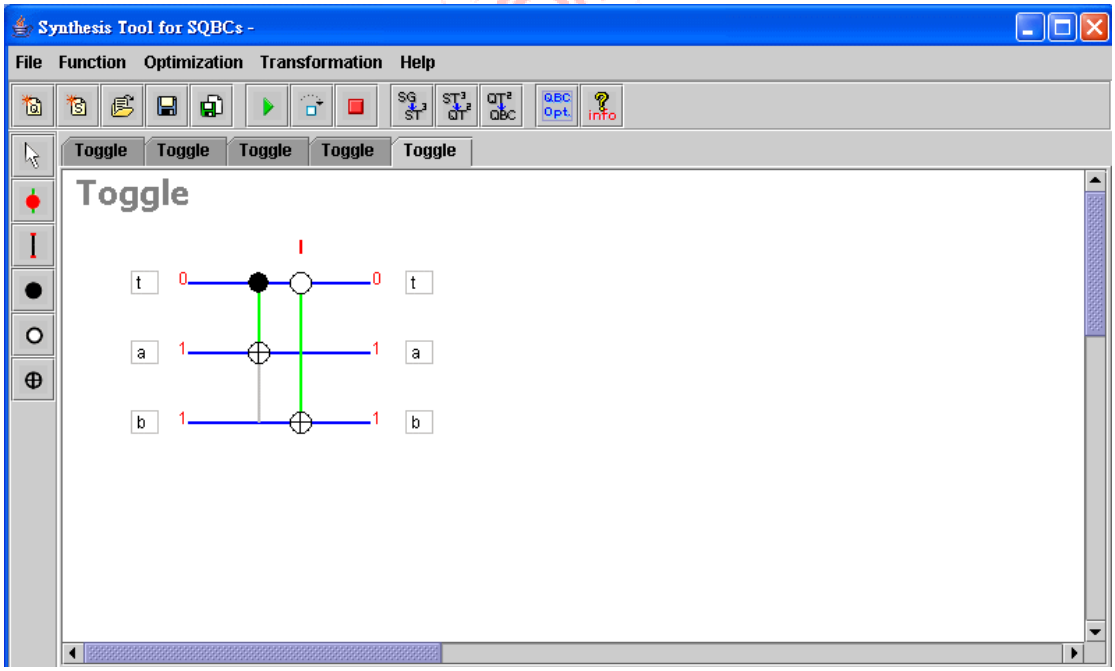
In order to verify the correctness of QBCs, our CAD tool can simulate and test QBCs. Only the value of qubits in QBCs has to be set and then the results (states) will be known after simulating. Finally the correctness of QBCs can be proved by the results.

For example, the simulation results of the optimized SQBC of a toggle, shown in Fig. 7.6, are shown in Fig. 7.7. The initial state of the toggle is shown in Fig. 7.6. If *t* is turned on, the state of the output *a* is changed to 1 and the result is shown in Fig. 7.7(a). Then *t* is turned off, the state of the output *b* is changed to 1 and the result is shown in Fig. 7.7(b). If *t* is turned on again, the state of the output *a* is changed to 0, shown in Fig. 7.7(c). Finally, *t* is turned off, the state of the output *b* is changed to 0 and the system goes to the initial state, shown in Fig. 7.6.

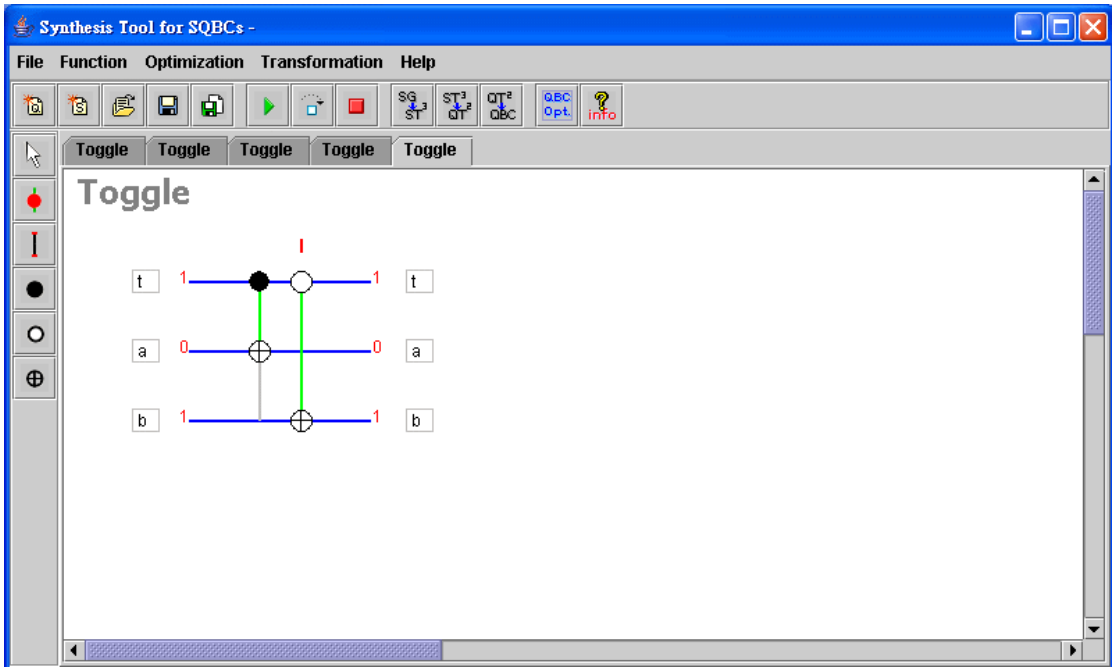
Thus, the correctness of SQBCs can be proved by the simulation tool.



(a)



(b)



(c)

Figure 7.7: The simulation results of SQBC of a toggle

Chapter 8.

Conclusions and Future Works

8.1 Conclusions

This thesis presents a novel methodology to transfer self-timed circuit specifications into sequential quantum Boolean circuits (SQBCs). State graphs (SGs) used for self-timed system design are exploited to describe the behaviors of circuits and then are automatically translated into SQBCs based on Toffoli gates.

The concept of IP reuse is also applied to the constructed SQBCs to produce reusable and composable quantum Boolean circuits (CQBCs). These reusable CQBCs as building blocks can be exploited to construct more complicated quantum Boolean circuits.

Based on our methodology a CAD tool written in Java to automatically synthesize SQBCs and CQBCs is also designed and implemented.

A universal set of self-timed components is successfully and automatically synthesized into CQBCs by using our CAD tool. These CQBCs can be used as building blocks to compose all control-path components of self-timed systems.

8.2 Future works

There are still some issues to be resolved and some suggestions for future research are

as follows:

(1) **Feedback wire problem**: A classis circuit can be easily composed by basic components with feedback wires. These feedback wires forms internal states. It is not clear how to construct the same circuit with composable QBCs.

(2) **Quantum algorithms to QBCs problem**: To be able to realize quantum algorithm a methodology has to be investigated and designed to transfer these algorithms into state graphs used in our CAD tool and make use of our CAD tool to automatically synthesize quantum algorithms into QBCs.

(3) **Optimization problem:** There are many different solutions during the state decomposition step. Thus it is worth investigating which solution(s) may result in better SQBCs.

Bibliography

[1] Tsai, I.-M.; Kuo, S.-Y.; "Quantum Boolean Circuit Construction and Layout under Locality Constraint", *Nanotechnology, 2001. IEEE-NANO 2001. Proceedings of the 2001 1st IEEE Conference on* , 28-30 Oct. 2001 Pages:111 - 116

[2] Wei-Min Zhang, *Introduction to Quantum Information Processing*, Lecture notes in the 2003 Symposium on Digital Life and Internet Technologies

[3] Cortadella, J.; Kishinevsky, M.; Kondratyev, A.; Lavagno, L.; Yakovlev, A.; "Complete state encoding based on the theory of regions", A*dvanced Research in Asynchronous Circuits and Systems, 1996. Proceedings, Second International Symposium on*, 18-21 March 1996 Pages: 36 – 46

[4] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. *Technical report*, 1996

[5] David L. Dill, *Trace theory for automatic hierarchy verification of speed-independent circuits*. Pages 51-65, 1987

[6] Iwama, K.; Kambayashi, Y.; Yamashita, S.; "Transformation Rules for Designing CNOT-based Quantum Circuits", *Design Automation Conference, 2002. Proceedings. 39th*, 10-14 June 2002 Pages:419 - 424

[7] Priyadarsan Patra, Donald Fussell, "Building-blocks for Designing DI Circuits", *Technical report tr93-23*, Dept. of Computer Sciences, The Univ. of Texas at Austin, November 1993

[8] M. A. Nielsen and I. L. Chung. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000

[9] Chris J. Myers, *Asynchronous Circuit Design*, Wiley & Sons, Inc., 2001

[10] Jens Sparso, Steve Furber, *Principle of Asynchronous Circuit Design*, Kluwer Academic Publishers, 2001

[11] Julian Brown. *Minds, Machines, and the Multiverse*. Simon & Schuster, 2000

[12] Arthur O. Pitternger, *An Introduction to Quantum Computing Algorithm*. Birkjauser, 1998

[13] Michael Brooks, *Quantum Computing and Communications*. Springer, 1998.

[14] J.C. Ebergen, *Translating programs into delay-insensitive circuits*. Stichting Mathematisch Centrum, Amsterdam, 1989.

Appendix

A. State Graph

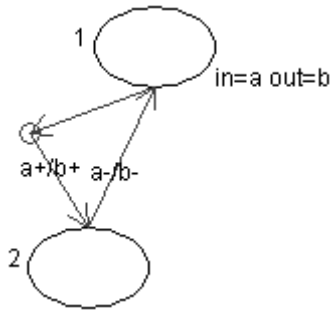


Figure A.1: The SG of a fork

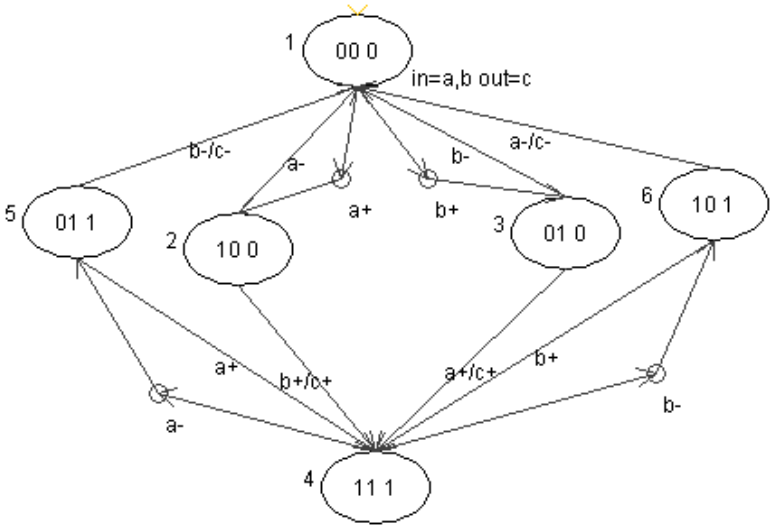


Figure A.2: The SG of a join

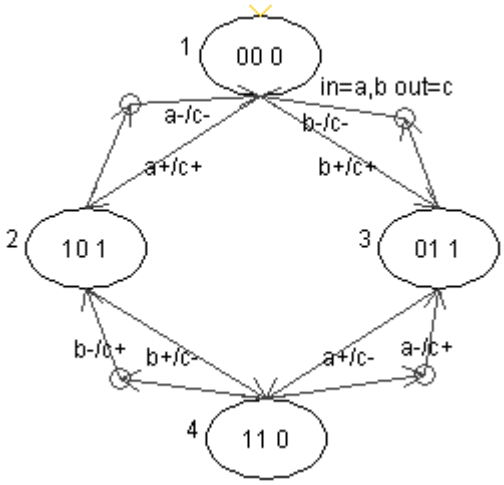


Figure A.3: The SG of a merge

B. Self-Timed Transformation Graph

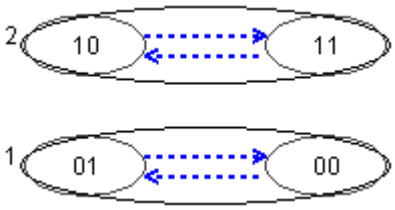


Figure B.1: The STTG of a fork

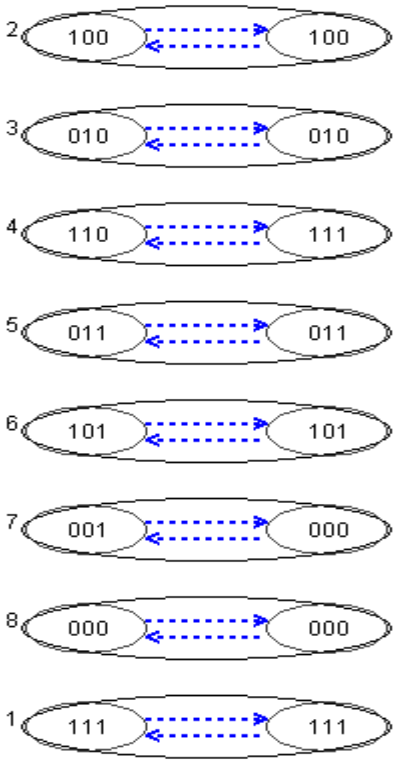


Figure B.2: The STTG of a join

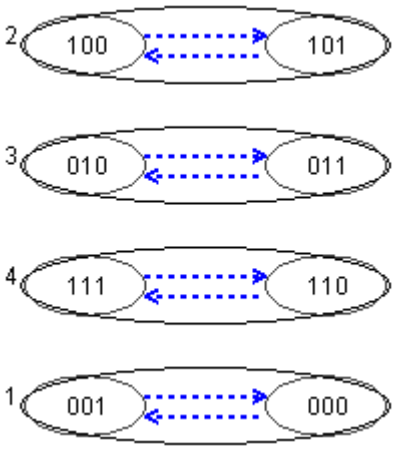


Figure B.3: The STTG of a merge

C. Composable Self-Timed Transformation Graph

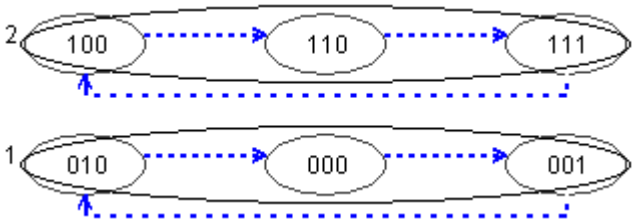


Figure C.1: The CSTTG of a fork

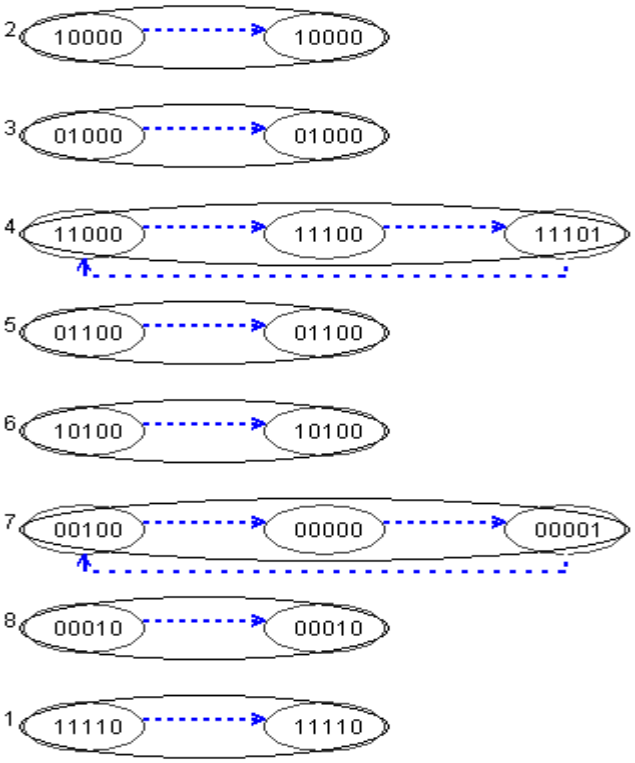


Figure C.2: The CSTTG of a join

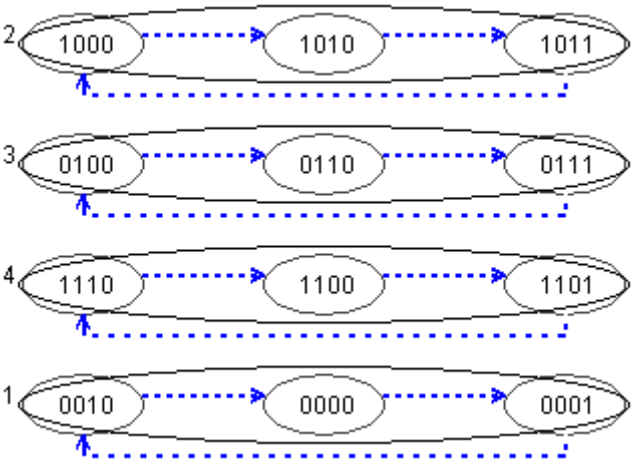


Figure C.3: The CSTTG of a merge

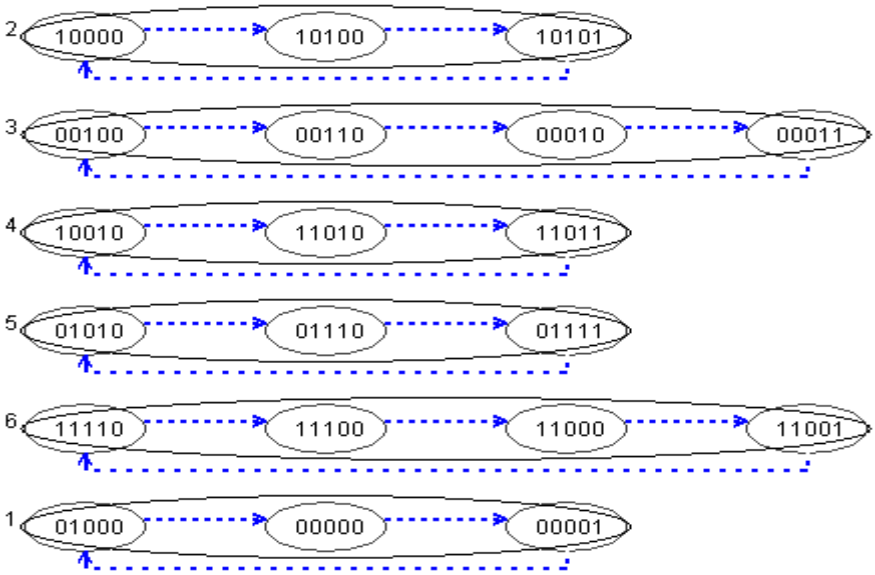


Figure C.4: The CSTTG of a modulo-3

D. Sequential Quantum Boolean Circuits

Figure D.1: The (a) SQBC and (b) the optimized SQBC of a fork

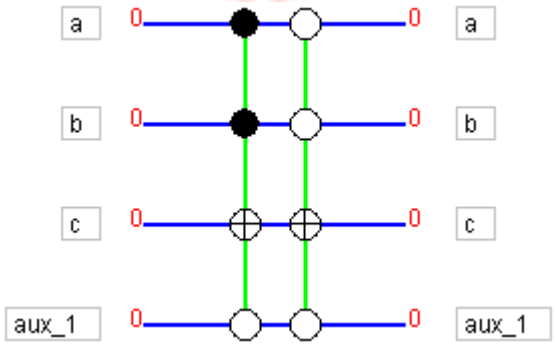


Figure D.2: The SQBC of a join

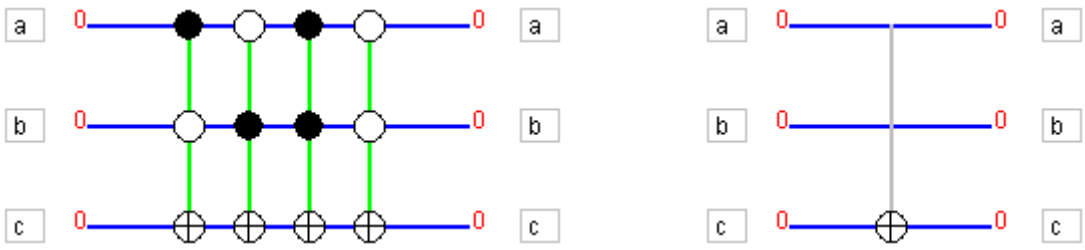


Figure D.3: The (a) SQBC and (b) the optimized SQBC of a merge

E. Composable Quantum Boolean Circuits

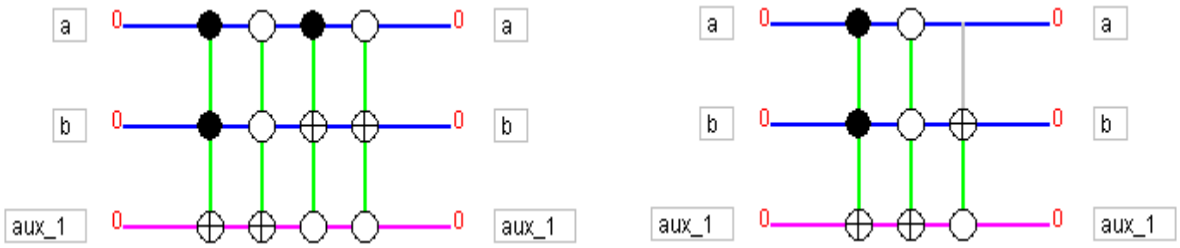


Figure E.1: The (a) CQBC and (b) the optimized CQBC of a fork

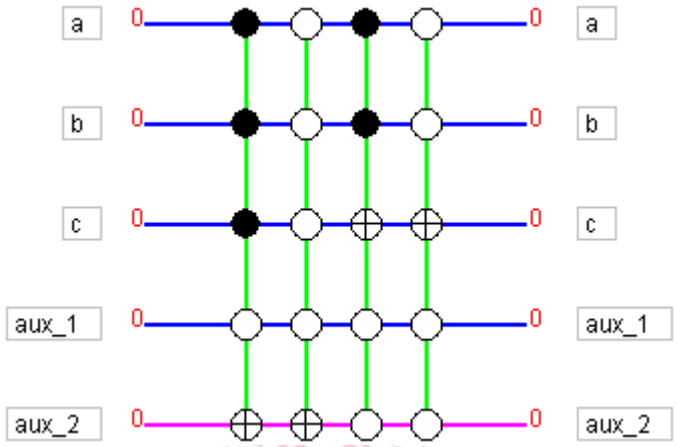


Figure E.2: The CQBC of a join

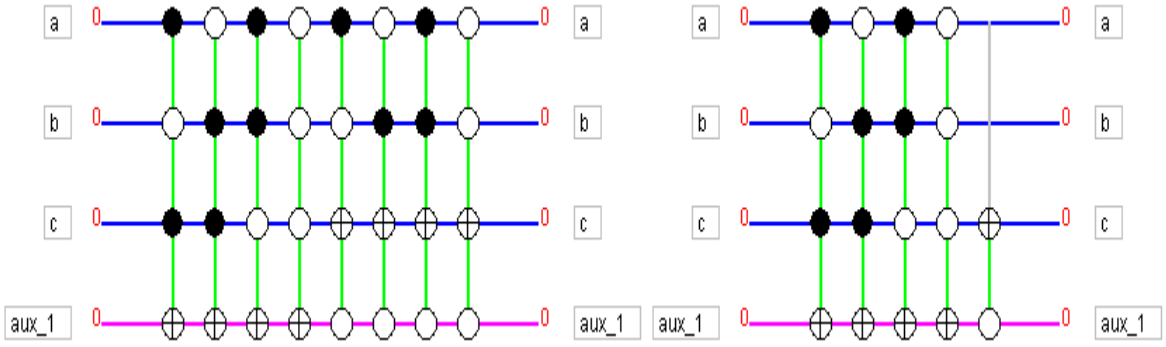


Figure E.3: The (a) CQBC and (b) the optimized CQBC of a merge

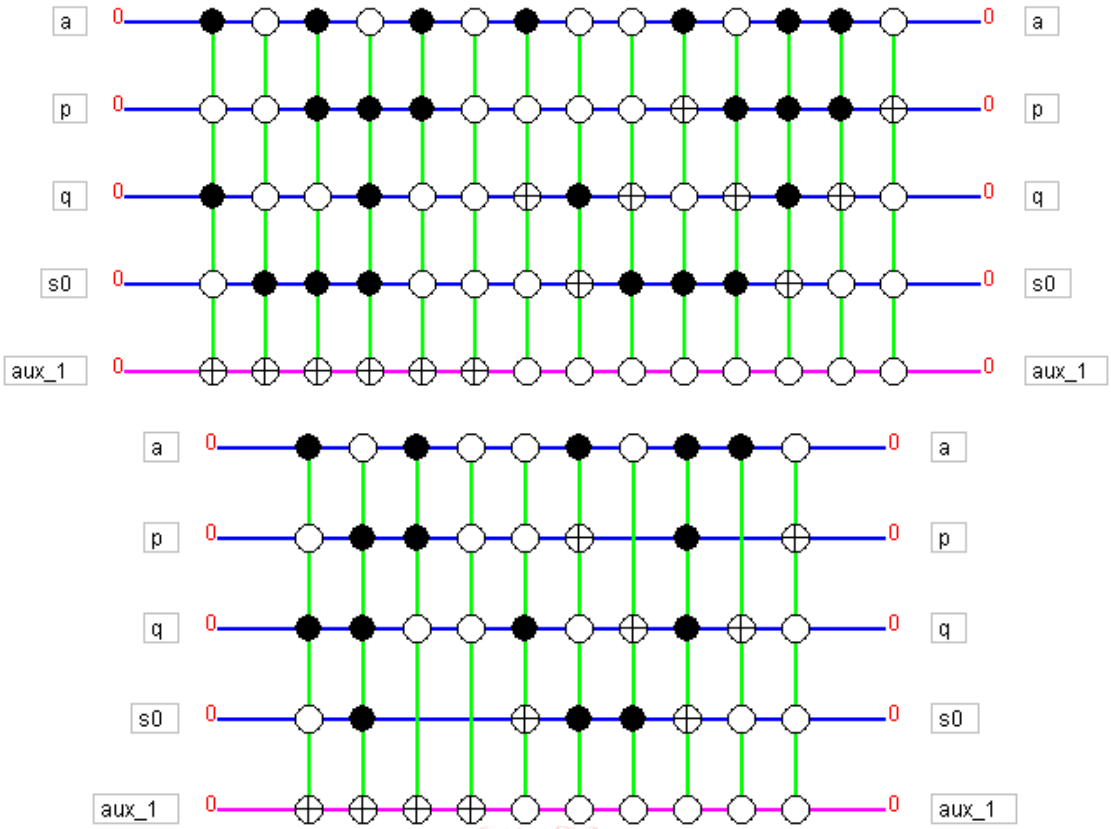


Figure E.4: The (a) CQBC and (b) the optimized CQBC of a modulo-3