# A New Fault Model and Its Application in Synthesizing Toffoli Networks

by

Jing Zhong
MSc, University of Victoria, 2003
BEng, Beijing University of Posts and Telecommunications, 1998

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of

Doctor of Philosophy

in the Department of Computer Science

# Supervisory Committee

A New Fault Model and Its Application in Synthesizing Toffoli Networks

by

Jing Zhong
MSc, University of Victoria, 2003
BEng, Beijing University of Posts and Telecommunications, 1998

Supervisory Committee

Dr. Jon C. Muzio, Supervisor
(Department of Computer Science)

Dr. D. Michael Miller, Departmental Member
(Department of Computer Science)

Dr. Kin F. Li, Outside Member
(Department of Electrical & Computer Engineering)

Dr. T. Aaron Gulliver, Outside Member
(Department of Electrical & Computer Engineering)

Supervisory Committee

Dr. Jon C. Muzio, Supervisor
(Department of Computer Science)

Dr. D. Michael Miller, Departmental Member
(Department of Computer Science)

Dr. Kin F. Li, Outside Member
(Department of Electrical & Computer Engineering)

Dr. T. Aaron Gulliver, Outside Member
(Department of Electrical & Computer Engineering)

## Abstract

Reversible logic computing is a rapidly developing research area. Both reversible logic synthesis and testing reversible logic circuits are very important issues in this area. In this thesis, we present our work in these two aspects.

We consider a new fault model, namely the crosspoint fault, for reversible circuits. The effects of this kind of fault on the behaviour of the circuits are studied. A randomized test pattern generation algorithm targeting this kind of fault is introduced and analyzed. The relationship between the crosspoint faults and stuck-at faults is also investigated.

The crosspoint fault model is then studied for possible applications in reversible logic synthesis. One type of redundancy exists in Toffoli networks in the form of undetectable multiple crosspoint faults. So redundant circuits can be simplified by deleting those undetectable faults. The testability of multiple crosspoint faults is analyzed in detail. Several important properties are proved and integrated into the simplifying algorithm so

as to speed up the process.

We also provide an optimized implementation of a Reed-Muller spectra based reversible logic synthesis algorithm. This new implementation uses a compact form of the Reed-Muller spectra table of the specified reversible function to save memory during execution. Experimental results are presented to illustrate the significant improvement of this new implementation.

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgments

I would like to sincerely thank my supervisor, Dr. Jon C. Muzio for his time and commitment, his advice and guidance, and his continual support during my graduate studies.

I am very grateful to Dr. Michael Miller and Dr. Dmitri Maslov for their kind help and suggestion to this work. My thankfulness also goes to everybody who has provided valuable feedback and comments on my papers and presentations.

I am also grateful to my supervisory committee, Dr. Miller, Dr. Gulliver, Dr. Li, and Dr. Dueck for giving time reading this thesis and offering support and valuable advice.

I extend my thanks to my family, all members in Digital System Design (DSD) group, all my friends, for their support and encouragement.

# Dedications

To my family

# Chapter 1 Introduction

With much faster and more complex digital systems being built, power consumption of CMOS circuits has become a major concern. In [25] Landauer proved that whenever we perform a logically irreversible operation, there is an unavoidable minimum energy dissipation into the environment, regardless of the realization of the circuit. The logical irreversibility associates with the physical irreversibility in that whenever an irreversible computation is performed, the physical computer erases information about its previous state. Physically the lost information is dissipated into the environment in the form of heat. For every bit of information which is thrown away, the physical computer must generate at least $kT\ln 2$ of energy, where $k$ is Boltzmann's constant, and $T$ is the absolute temperature of the environment where the computation is conducted. This is about $2.9\times10^{-21}$ joule for each bit of information loss at room temperature. In reality, computers consume much more energy than this. Therefore the power dissipation is absolutely a critical issue for the development of modern Very Large Scale Integration (VLSI) technologies.

In order to reduce or remove the arbitrary heat release, researchers have been studying the area of reversible computing. In [7] Bennett showed that, under ideal physical circumstances, the power dissipation of reversible computing is zero. In his work, Bennett also introduced approaches to implementing reversible computation both logically and physically. This is why reversible computing has been gaining more and more attention in recent years.

There are two types of reversibility: physical reversibility and logical reversibility. Physical reversibility refers to the physical implementation of reversible computing. During the computation process the mechanisms of the physical system are managed in a way such that there is no energy release. Existing reversible technologies include quantum technology [22], CMOS technology [4][14], DNA technology [41], nanotechnology [30], optical technology [40], thermal technology [24] as well as other approaches.

Quantum technology is an evolving technology. Quantum computing [34][38] is well-known for being able to exponentially speed up the computation for classical computationally complex problems. The basic principle of quantum computing is to use quantum properties to describe data and use quantum mechanisms to conduct operations on the data. Unlike traditional computers which work with information represented in a series of bits, quantum computers encode information with qubits. Quantum computation is reversible.

In order for a computation to be implemented as physically reversible, the computation must also be logically reversible. A function is logically reversible if its output-to-input mapping is a one-to-one function. Reversible logic gates are fundamental components used to implement reversible logic functions. Commonly used reversible logic gates include the CNOT gate, Toffoli gate, Fredkin gate, SWAP gate, and the classical NOT gate. Other traditional gates such as NAND, EXOR, and NOR gates are irreversible. The reversible logic circuits (or reversible networks) have a very simple cascade structure since neither feedback nor fanout is allowed.

For a good introduction to reversible logic computing, refer to [35]. One of the most

important aspects in this area is reversible logic synthesis. Since reversible logic circuits are constructed with reversible logic gates which are quite different from traditional irreversible gates, many of the traditional synthesis approaches do not work for reversible logic synthesis. Recently a large variety of new synthesis methods are proposed to generate reversible logic circuits with minimum cost or with small numbers of extra outputs. These extra outputs, labeled garbage outputs, are produced during the synthesis process. The following are some of the reversible logic synthesis techniques: exhaustive enumeration [46], ESOP approach (Exclusive OR Sum Of Product) [2][33], transformation-based technique [32], spectral techniques [31], and the use of templates [28].

While the synthesis of reversible logic circuits has been explored in some detail, the testing of reversible circuits is also a critical aspect in this area, yet this aspect is still underdeveloped. Related work has focused on fault detection and fault localization in reversible logic circuits under different fault models. Besides the classical stuck-at fault model, several new faults models, such as the cell fault mode [37] and the partially missing gate fault model [19], are proposed and analyzed. Different detection approaches and Automatic Test Pattern Generation (ATPG) algorithms for these fault models are also described in related work.

In this thesis, we show that the classical stuck-at fault model is not adequate in representing a variety of fault effects in reversible logic circuits and we define a new fault model, the crosspoint fault model for reversible circuits. The main focus of our work is on the analysis of the properties of this new fault model, including ATPG strategies for it and its relationship to the classical stuck-at fault model. We also investigate how this

fault model can be used in simplifying Toffoli networks. A new implementation of a spectral-based reversible logic synthesis algorithm is also developed to speed up the synthesis process.

The rest of this thesis is organized as follows:

- In Chapter 2, 'Background', we give an overview of the basic background for this thesis, which includes: Boolean functions and Boolean algebra; different aspects related to digital system testing such as system structure, fault modeling, test pattern generation, and fault simulation; and spectral techniques, especially the Reed-Muller spectra of Boolean functions.
- The next chapter, 'Reversible Logic Circuits', focuses on the background in reversible logic computing. Basic definitions and concepts in reversible logic, commonly used reversible logic gates, the structure of reversible logic circuits, especially the Toffoli networks, are introduced. In this chapter, we also give an overview of reversible logic synthesis and the testing of reversible logic circuits. Different approaches and techniques, as well as related work in these aspects, are also included.
- In Chapter 4, 'Crosspoint Fault Model', we show that the classical stuck-at fault model is inadequate for reversible logic circuits. We propose a new fault model, the crosspoint fault model, to represent a type of internal defect in reversible logic gates. Detailed properties of reversible circuits under this new fault model are analyzed. A randomized automatic test pattern generation (ATPG) algorithm is proposed to generate a minimum or near-minimum test set for crosspoint faults in Toffoli networks. Simulation is conducted to test the performance of this ATPG

algorithm. In this chapter, we also investigate the relationship between our new crosspoint fault model and the classical stuck-at fault model. Some of the results are presented in our paper [50].

- Fault models sometimes are useful in logic synthesis. In Chapter 5, 'Crosspoint Faults in Simplifying Toffoli Networks', we introduce a new method of simplifying Toffoli networks using our new crosspoint fault model. We study the relationship between the multiple disappearance faults and a certain type of redundancy in reversible logic circuits and use this connection to reduce redundancy in the circuits by deleting independent undetectable multiple disappearance faults. Some important properties related to the testability of double crosspoint faults are provided. These properties can be integrated into the synthesis algorithm to speed up the simplifying process. The simplifying algorithm is given and analyzed. Experimental results are presented to illustrate its performance. Some results in this chapter can be found in our paper [51].
- In Chapter 6, 'An Improved Implementation of a Reversible Synthesis Algorithm', we provide an optimized implementation for a Reed-Muller spectra based reversible logic synthesis algorithm proposed by Maslov et al. in [29]. The original implementation works on the whole Reed-Muller spectra table of the given reversible logic function, which requires a large amount of memory during execution. In our implementation, we reduce the memory usage sharply by storing a compact form of the Reed-Muller spectra table, which contains only the non-zero Reed-Muller spectra rows and counts for a very small fraction of the whole Reed-Muller spectra table. Simulation is conducted to compare the performance

of the new implementation and the old one. Some of the results are given in our paper [52].

- In Chapter 7, ‘Conclusions’, we conclude our work and explain the contributions in testing reversible logic circuits and simplifying Toffoli networks described in this thesis. Possible future research topics in the area of reversible logic are also discussed in this chapter.

# Chapter 2 Background

In this chapter we give the relevant background material related to this thesis. It includes a brief introduction to Boolean algebra and Boolean functions, basic concepts on digital circuits and testing digital systems, as well as some background on spectral techniques.

## 2.1 Boolean Algebra and Boolean Function

First we give a brief description of Boolean algebra [10] and Boolean functions.

A **Boolean algebra** is a set $A$, with three binary operations $\wedge$(AND), $\vee$(OR), and $\neg$ (NOT), and two distinct elements 0 and 1, such that, for all elements $a, b, c$ of set A, the following axioms hold:

| | | |
|---|---|---|
| *Associativity:* | $a \vee (b \vee c) = (a \vee b) \vee c$ | $a \wedge (b \wedge c) = (a \wedge b) \wedge c$ |
| *Commutativity:* | $a \vee b = b \vee a$ | $a \wedge b = b \wedge a$ |
| *Absorption:* | $a \vee (a \wedge b) = a$ | $a \wedge (a \vee b) = a$ |
| *Distributivity:* | $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ | $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ |
| *Complements:* | $a \vee \neg a = 1$ | $a \wedge \neg a = 0$ |

A **Boolean function** is a function $f(x_1, x_2, \cdots, x_n)$ assembled by applying the operations $\wedge$(AND), $\vee$(OR), and $\neg$ (NOT) to the variables $x_1, x_2, \cdots, x_n$, within the domain of Boolean algebra.

In this thesis, we use simplified notation to represent binary operations:

$ab = a \wedge b$; $a + b = a \vee b$; $\overline{a} = \neg a$.

A Boolean function can be described with many different methods including truth tables [1, page 10], primitive cubes [1, page 11], state tables [1, page 13], and binary decision diagrams (BDD) [1, page 17]. One of the most common and easy representations is the truth table, which lists the corresponding output value for every input vector. For instance, the truth table of the Boolean function $y = f(x_1, x_2, x_3) = x_1 x_2 + x_3$ is shown in Table 2-1.

TABLE 2-1 TRUTH TABLE OF A BOOLEAN FUNCTION

| $x_1$ | $x_2$ | $x_3$ | $y$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

A Boolean function can have more than one output. For instance, we can have a Boolean function as follows:

$$
\begin{aligned}
&(y_1, y_2, y_3, y_4) \\
&= (f_1(x_1, x_2, x_3),\ f_2(x_1, x_2, x_3),\ f_3(x_1, x_2, x_3),\ f_4(x_1, x_2, x_3)) \\
&= (x_1 x_2 + x_3,\ x_1 x_2 x_3,\ x_1 x_2 + x_1 x_3,\ x_1 + x_2 + x_3)
\end{aligned}
$$

The truth table of this 4-output Boolean function is shown in Table 2-2.

TABLE 2-2 TRUTH TABLE OF A BOOLEAN FUNCTION

| $x_1$ | $x_2$ | $x_3$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Obviously, for a single output Boolean function with $n$ input variables, it can have a total of $2^{2^n}$ distinct outputs.

## 2.2 Testing Digital Circuits

Since a major part of this thesis is concerned with the testing of reversible logic circuits, in this section we give a brief introduction to testing digital circuits. It includes the general model for the testing of digital systems, concepts and definitions related to testing digital systems, and other important aspects such as fault modeling, fault simulation, and test pattern generation (TPG).

### 2.2.1 Basic Concepts

The testing of a digital circuit is the process of exercising a digital system with stimuli and then analyzing the output response to see whether the circuit worked correctly. In testing, an **error** refers to an instance of an incorrect operation of the circuit and a **fault** is

the cause of an error, which represents the physical difference between a good system and a bad one [21].

Figure 2-1 shows the general model for testing digital circuits.

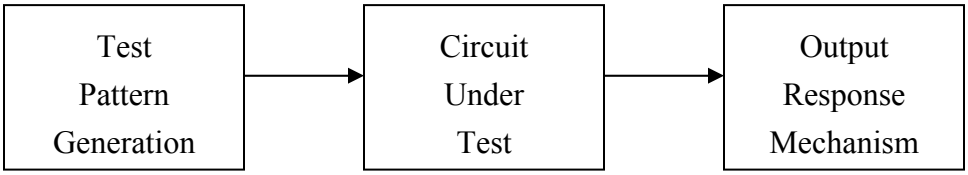



Figure 2-1 Testing digital systems

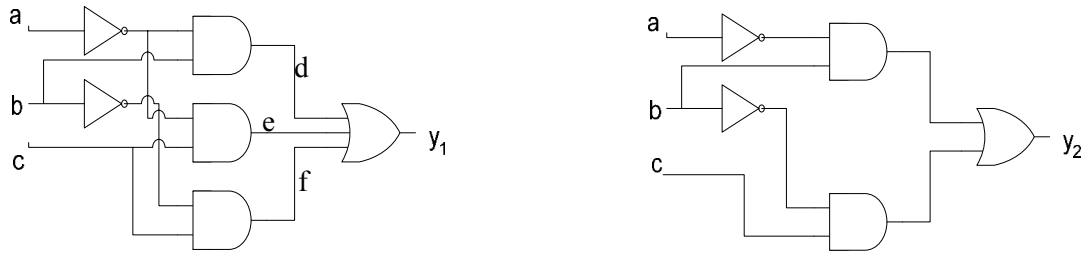
First test patterns (or test vectors) are generated (pseudo-randomly or deterministically). These patterns are used to exercise the Circuit Under Test (CUT). Here the CUT can be a traditional irreversible circuit or a reversible one. When all test patterns have been applied, a mechanism is used to analyze the results and verify whether they correspond to the correct output patterns and hence to decide whether there is any fault in the CUT or not.

Testing digital circuits involves many important aspects such as fault modeling, fault simulation, and test pattern generation (TPG) for different fault models and different circuits. We explain these topics in the following section.

Below we give definitions of some important concepts related to digital circuits or testing digital circuits.

A circuit is called **redundant** if one or more gates or gate inputs can be deleted without changing the logical function of the circuit [1, page 100]. For instance, in Figure 2-2a the circuit can be simplified by removing an AND gate and an input from the OR gate. The

simplified circuit (Figure 2-2b) is irredundant and implements the same function as the original one.




(a) (b)

Figure 2-2 Redundancy in a digital circuit

Redundancy in circuits makes the testing process considerably more difficult, but is often required for other purposes, such as fault tolerance.

Let $p$ and $q$ be the logic functions realized by two circuits $P$ and $Q$ respectively. If $p = q$, we say that the two circuits $P$ and $Q$ are **functionally equivalent**. The circuit in Figure 2-2a and the circuit in Figure 2-2b are functionally equivalent circuits since they implement the same logic function:

$$y_1 = \bar{a}b + \bar{a}c + \bar{b}c$$

$$y_2 = \bar{a}b + \bar{b}c$$

$$y_1 = y_2$$

It could also be verified by listing the truth tables of functions $y_1$ and $y_2$ (see Table 2-3).

TABLE 2-3 TRUTH TABLE OF TWO FUNCTIONALLY EQUIVALENT CIRCUITS

| $a$ | $b$ | $c$ | $y_1$ | $y_2$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 |

Given a circuit $Z$ with function $z$, assume there is a fault $f$ in the circuit and the faulty circuit has function $z_f$. If $z \neq z_f$, the fault $f$ is **detectable**, otherwise it is not detectable. Given a test vector $t$, if $z(t) \neq z_f(t)$, we say the test vector $t$ **detects** the fault $f$. For example, in the circuit in Figure 2-2a, the fault 'line a is stuck at logic value 0' is detectable. It can be detected by the test vector "111". However, in the same circuit, the fault 'line e is stuck at logic value 0' is not detectable.

Similarly, a test set $T$ is said to be able to **detect** a fault $f$ in a circuit $Z$ if $z(T) \neq z_f(T)$.

### 2.2.2 Fault Modeling

**Fault models** describe the effect of physical faults on the behavior of the modeled system [1, page 93]. For traditional irreversible circuits, there are some commonly used fault models, with stuck-at faults, bridging faults and delay faults being the most

common.

The **stuck-at fault model** is one of the very important fault models for conventional irreversible logic circuits. It represents a line of a circuit being stuck at a fixed logic value (logic 0 or logic 1) [1, page 94]. It is denoted by $s-a-v$, $v \in \{0,1\}$. For instance, if a wire is shorted with the power (or ground), we say that this line is stuck-at logic 1 (or logic 0). Figure 2-3 shows an AND gate whose input A is stuck-at 0. It causes the output value to be logic 0. This fault 'line A s-a-0' can be easily detected by applying vector "11" to the inputs.

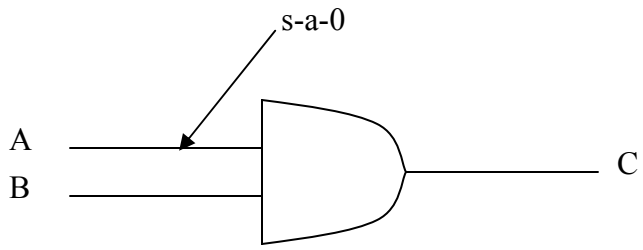



Figure 2-3 A stuck-at fault

The stuck-at fault model does not only cover the situations where a line is physically stuck at some fixed value, it also covers many other realistic defects such as physical defects within a TTL (Transistor-Transistor Logic) gate [21], where internal defects in the gate could be mapped to stuck-at faults on gate inputs or outputs.

The stuck-at fault model is a classical approach to many faults in digital systems. However, there are situations where faults are not covered by this classical fault model. In these situations we need to define different fault models to describe the effect of the faults.

The **bridging fault model** denotes a short between two lines which creates a new logic function [1, page 94]. It can be classified into AND bridging faults and OR bridging faults depending on the new function created by the short. Figure 2-4 shows an example of an OR bridging fault. Suppose line A is one output of block 1 and line B is one output of block 2. If there is a short between A and B which results in an OR bridging fault, the value of both line A and line B would be A+B (A OR B).

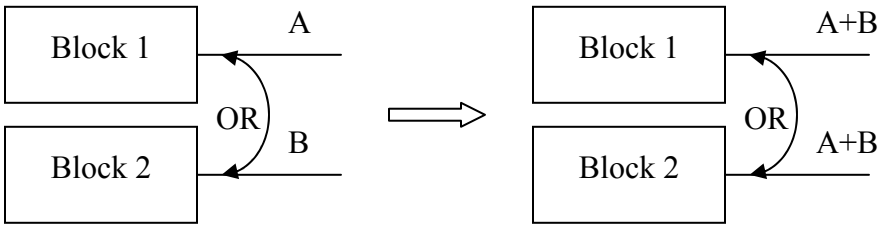



Figure 2-4 An OR bridging fault

The **delay fault model** represents signal delays in the circuit [1, page 52]. There are two types of delay faults: slow-to-rise transition (the transition from logic 0 to logic 1 is slower than expected) and the slow-to-fall transition (the transition from logic 1 to logic 0 is slower than expected). In order to catch the effect of the fault at the output, two vectors are needed to detect it. For example, as shown in Figure 2-5, to detect the slow-to-fall transition on line A, a transition '1 → 0' is required to be applied to the input A. The patterns (11, 01) can detect this fault since when the second pattern "01" is applied, there will be a delay at output C before it changes from logic 1 to logic 0.

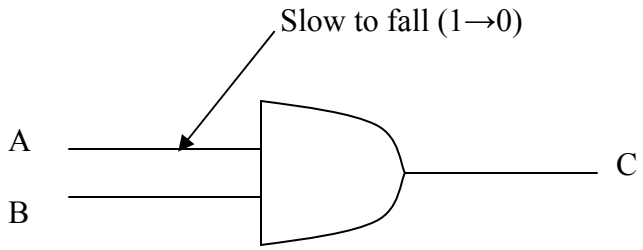



Figure 2-5 Example of a delay fault

Fault models can also be divided into single fault models and multiple fault models, depending on how many faults are allowed in the circuit at a time. For instance, the stuck-at fault model includes the single stuck-at fault (SSF) model and the multiple stuck-at fault (MSF) model.

### 2.2.3 Fault Simulation and Test Pattern Generation

**Fault simulation** is used to simulate a digital circuit in the presence of faults [1, page 131]. It applies a test set to a circuit and then collects the outputs. By comparing the outputs to those from simulating the same circuit without introducing faults, one can determine whether the faults are detected by the test set.

Fault simulation is also used to assess the quality of a test set by computing the fault coverage of this test set. **Fault coverage** is defined as the percentage of faults covered or detected by the test set.

**Test pattern generation (TPG)** is the process of generating test patterns (test vectors) for the CUT. TPG is usually classified into three categories: exhaustive TPG, random TPG, and deterministic TPG.

- Exhaustive TPG is to test a combinational circuit with all possible input vectors. For a circuit with $n$ inputs, there are a total of $2^n$ distinct input vectors. So in this technique all $2^n$ patterns are used to test the CUT. Obviously, exhaustive testing yields maximum fault coverage. All detectable faults could be detected using this method. But as $n$ grows, the number of test vectors increases exponentially, which makes it impractical for a circuit of any significant size.
- Deterministic TPG produces test patterns deterministically. It can be divided into two groups: fault-oriented or fault-independent. The former generates a test set targeting specified faults while the latter approach does not. Some famous deterministic TPG algorithms for testing single stuck-at faults in combinational circuits include the D-algorithm [1, page 198], the 9-V algorithm [1, page 199], and PODEM [1, page 203].
- Random TPG means to generate the test patterns randomly. In reality, we can only generate patterns pseudo randomly, which means the patterns are produced with a deterministic algorithm but they have many characteristics of randomness. Random TPG is easy to implement. Test patterns are often generated by some Linear Finite State Machine (LFSM) such as a Linear Feedback Shift Register (LFSR) [[23]] or a Linear Hybrid Cellular Automata (LHCA) [11]. Random TPG usually generates test sets with much larger size than that generated by deterministic algorithms, but it is much better for Built-In Self-Test (BIST) [5] because of the minimal storage requirements.

Generating test vectors for a specified fault in a fan-out-free combinational circuit

usually involves two fundamental steps: line justification and propagation. **Line justification** means to set primary input values such that the input vector excites the specific fault. **Propagation** means to propagate the effect of the fault to the primary outputs. We give an example to illustrate these two steps.

Assume there is a single stuck-at fault in the circuit shown in Figure 2-6. The circuit has one output g and four inputs a, b, c, and d. Two internal lines are e and f. The single stuck-at fault is line e s-a-1.

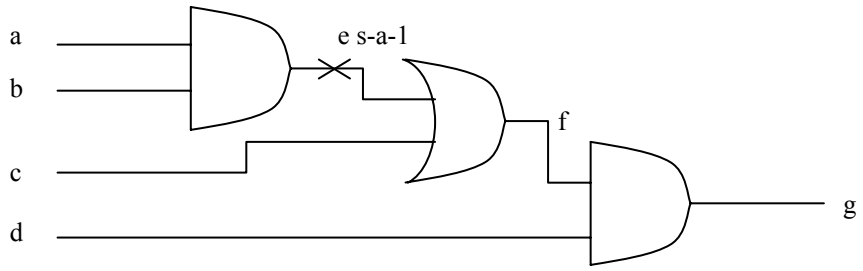



Figure 2-6 Generate a test vector for a single stuck-at fault

Line-justification: Since the fault is line e s-a-1, which means that in all cases line e has a constant value – logic 1, in order to excite the fault, we need a test vector which results in a logic 0 at e. Thus for the left AND gate, at least one of its inputs must have value of 0. So at the primary inputs line a and line b could take any of the three values: “00”, “01”, and “10”.

Propagation: It is obvious that in order to pass the fault effect to the primary output g, the input c must have value 0, d must have value 1.

From the analysis above, we conclude that any of the three vectors “0001”, “0101”, and

"1001" detects the fault line e s-a-1.

Generating test vectors for sequential circuits and circuits with fan-out is more complex. We do not explain these techniques here since reversible circuits are fan-out-free and loop-free. Further detailed information regarding testing sequential circuits can be found in Section 6.3, [1] and Chapter 5, [21].

## 2.3 Spectral Techniques

In this thesis we introduce an improved implementation of a Reed-Muller spectra based approach for reversible logic synthesis, so here we discuss spectral techniques, especially Reed-Muller spectra.

The spectral domain is another domain for analyzing digital circuits for further properties besides those obtained in the Boolean domain. The transformation between the Boolean domain and the spectral domain can be accomplished using this equation: $S = TF$, where $F$ is the column output vector of the Boolean function, $T$ is the transform matrix, $S$ is the spectrum vector of the given Boolean function. Each entry (or spectral coefficient) in $S$ gives some global information about that function, while in Boolean domain a single entry only gives local information [20]. Spectral techniques have been widely used in digital logic, including digital logic synthesis and fault diagnosis.

Boolean data can be transformed to a different domain using a transform matrix. There are many different binary transform matrices [20], such as the Hadamard Orthogonal Transform Matrix, the Walsh and Rademacher-Walsh Transform Matrices, the Harr

Transform Matrix, the Reed-Muller Transform Matrix. Different information is provided in the different spectral domains. Below we show how the Reed-Muller spectra are derived.

The RM spectrum of a Boolean function can be easily computed using the RM transformation matrix $M^n$:

$$R = M^n f \,. \qquad (2.1)$$

Here $M^n = \begin{bmatrix} M^{n-1} & 0 \\ M^{n-1} & M^{n-1} \end{bmatrix}$ and $M^0 = [1]$. The operation is over GF(2).

For instance, given a Boolean function $f(x_1, x_2, x_3) = x_1 x_2 + x_3$, it is easy to compute its output vector $F = [0,0,0,1,1,1,1,1]^T$. According to Equation 2.1, the RM spectrum of this function can be computed:

$$R = M^n F = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Note that the operation '+' in the evaluation of the matrix multiplication is mod2 addition.

The elements in the derived Reed-Muller spectrum represent the coefficients $(a_0, a_1, a_2, \cdots, a_{2^n-1})$ in the Positive Polarity Reed-Muller form (PPRM) of the Boolean function. Any Boolean function $f(x_1, x_2, \cdots, x_n)$ of $n$ variables can be uniquely

represented by the **Positive Polarity Reed-Muller form (PPRM)** $a_0 \oplus a_1 x_1 \oplus a_2 x_2 \oplus \cdots \oplus a_{2^n-1} x_1 x_2 \cdots x_n$ with $a_i \in \{0,1\}$. Here $\oplus$ denotes the Exclusive OR (EXOR) operation. It is positive polarity since the complement of a variable is not allowed in the expression. A Boolean function can also be described in Fixed Polarity Reed-Muller form (FPRM) or Mixed Polarity Reed-Muller form (MPRM). In FPRM form each variable can appear as either complemented or not complemented, while in MPRM form a variable can appear as both. Each Boolean function has more than one FPRM or MPRM form. More properties regarding the FPRM and MPRM forms of a Boolean function can be found in [15].

For a 3-variable Boolean function the RM spectra correspond to the coefficients of the following terms in the PPRM form of the given function: $1, x_1, x_2, x_1x_2, x_3, x_1x_3, x_2x_3, x_1x_2x_3$. So in the above example, the PPRM form of the Boolean function $f(x_1, x_2, x_3) = x_1x_2 \oplus x_3 \oplus x_1x_2x_3$.

We can also get back the Boolean specification of the function by transforming the RM spectrum using the inverse of the RM transform matrix:

$f = (M^n)^{-1} M^n f = (M^n)^{-1} R$.

Since the RM transform matrix is self-inverse, which means $(M^n)^{-1} = M^n$, the Boolean specification could also be computed from $f = M^n R$, with mod-2 addition.

A fast Reed-Muller transform algorithm is provided by Thornton et al. in [47]. Figure 2-7 shows an example of this algorithm. We compute the RM spectra of the Boolean function $f(x_1, x_2, x_3) = x_1x_2 + x_3$ with its output vector $F = [0,0,0,1,1,1,1,1]^T$.

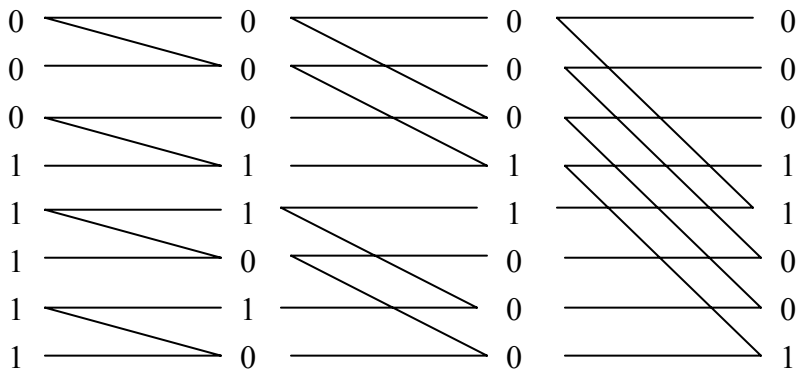



Figure 2-7 Example of fast Reed-Muller transform

Using this fast RM transform algorithm we obtain the RM spectrum of the specified function:

$$R = [0,0,0,1,1,0,0,1]^T .$$

Figure 2-8 shows the interpretation of a "Butterfly" signal flowgraph for the Reed-Muller transform, where $\oplus$ denotes the XOR operation.

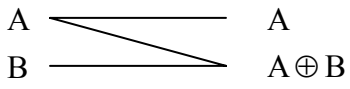



Figure 2-8 Interpretation of a signal flowgraph for the RM transform

The following properties can be used to compute the composite RM spectra of two functions [13].

Given two $n$-variable Boolean functions $f(X)$ and $g(X)$, assume that $\hat{f}(W)$ and $\hat{g}(W)$ are the RM spectra of $f(X)$ and $g(X)$, respectively, i.e.,

$\hat{f}(W) = M^n f(X)$ and

$\hat{g}(W) = M^n g(X)$,

The composite spectra of these two functions $f(X)$ and $g(X)$, can be computed using the following equations, where $\oplus$, $\wedge$, and $\vee$ denote componentwise XOR, AND, and OR operations.

1) Let $\varphi(X) = f(X) \oplus g(X)$, then

$$\hat{\varphi}(X) = \hat{f}(X) \oplus \hat{g}(X). \tag{2.2}$$

2) Let $\varphi(X) = f(X) \wedge g(X)$, then

$$\hat{\varphi}(X) = \overset{\oplus}{\underset{U \vee V = W}{}} \hat{f}(U) \cdot \hat{g}(V). \tag{2.3}$$

3) Let $\varphi(X) = f(X) \vee g(X)$, then

$$\hat{\varphi}(X) = \hat{f}(W) \oplus \hat{g}(W) \overset{\oplus}{\underset{U \vee V = W}{}} \hat{f}(U) \cdot \hat{g}(V). \tag{2.4}$$

Equations 2.2, 2.3, and 2.4 show that the composite RM spectra of two Boolean functions can be computed in the RM spectral domain if the RM spectra of each of the Boolean functions are provided. However, only XOR operations can be easily computed in the spectral domain. For other operations such as AND and OR, sometime it is easier to apply the RM transform to the RM spectra of both individual functions, then perform the specified operation, and finally get back to the RM spectra domain by applying the fast RM transform again.

## 2.4 Conclusion

In this chapter we touched upon fundamental background knowledge related to this dissertation. First we explained the concepts of Boolean algebra and Boolean functions, as well as the basic axioms in Boolean algebra.

We also gave a brief introduction to digital system testing, which included the general model of digital testing, fault modeling and commonly used fault models for conventional irreversible circuits, fault simulation methods, and different TPG approaches for digital systems.

Spectral techniques, especially the Reed-Muller transform and the PPRM form of a Boolean function, were given in detail. Fast RM transform algorithm, as well as methods of computing composite RM spectrum, was also provided.

In the following chapter we give the background on reversible logic circuits.

# Chapter 3 Reversible Logic Circuits

In this chapter we introduce basic definitions and related work in reversible logic [17][53], reversible logic synthesis, and testing reversible logic circuits.

## 3.1 Reversible Logic

First we give the definition of a reversible function.

A Boolean function is **reversible** if ([35]):

1. It has the same number of inputs and outputs;
2. There is a one-to-one correspondence between the input vectors and the output vectors, i.e. in the truth table of the function, there is a distinct output row matching each distinct input row.

As we can see, many traditional Boolean functions, such as AND, OR, XOR, NAND, are irreversible since all of them have more than one input but only one output. Also there is no one-to-one matching between the input vectors and the output vectors. However, the NOT function is reversible since it satisfies the definition above: it has one input and one output, and for each input vector, there is a distinct output vector corresponding to it so that the computation could be done from input to output, and inversely, from output to input as well.

Table 3-1 lists the truth table of a 2-input reversible gate called Controlled-NOT gate (or CNOT, Feynman gate) [16]. CNOT has two inputs ( $X$ and $Y$ ) and two outputs ( $X'$

and $Y'$). It is shown from the truth table that there is a unique output vector corresponding to each input vector.

TABLE 3-1 TRUTH TABLE OF CNOT

| $X$ | $Y$ | $X'$ | $Y'$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 |

Figure 3-1 gives the structure of the CNOT gate, where $\oplus$ denotes the Exclusive OR (XOR) operation. The value of output $X'$ is always the same as that of input $X$, while the value of output $Y'$ equals to the value of input $Y$ if $X = 0$, otherwise $Y' = \overline{Y}$. Here input $X$ is called a control input and input $Y$ is called a target input.

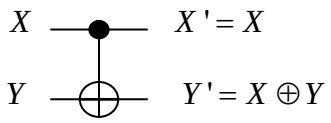



Figure 3-1 The CNOT gate

The function of the CNOT gate is:

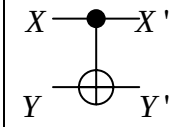
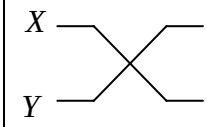
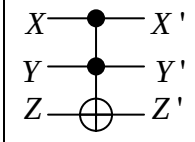
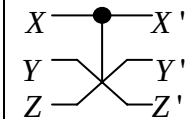
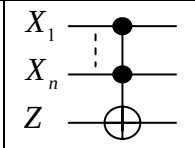
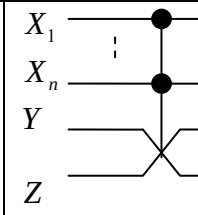
$$X' = X$$

$$Y' = X \oplus Y .$$

Other commonly studied reversible gates include: SWAP gate, Toffoli gate [48],

Fredkin gate [18], generalized Toffoli gate, and generalized Fredkin gate. The structure and function of each reversible gate are listed in Table 3-2, where $\oplus$ denotes the XOR operation.

TABLE 3-2 REVERSIBLE LOGIC GATES

| Gate Name | Structure | Function |
|---|---|---|
| NOT |  | $X' = \bar{X}$ |
| CNOT (Feynman) |  | $X' = X$<br>$Y' = X \oplus Y$ |
| SWAP |  | $X' = Y$<br>$Y' = X$ |
| Toffoli |  | $X' = X$<br>$Y' = Y$<br>$Z' = XY \oplus Z$ |
| Fredkin |  | $X' = X$<br>$Y' = \bar{X}Y \oplus XZ$<br>$Z' = \bar{X}Z \oplus XY$ |
| Generalized Toffoli |  | $X_i' = X_i$<br>$Z' = X_1 \cdots X_n \oplus Z$ |
| Generalized Fredkin |  | $X_i' = X_i$<br>$Y' = \bar{M}Y \oplus MZ$<br>$Z' = \bar{M}Z \oplus MY$ ,<br>$M = X_1 X_2 \cdots X_n$ |

The SWAP gate has two inputs and two outputs. It switches the input values at the outputs.

The Toffoli gate is a three-input three-output reversible gate. The structure of the Toffoli gate is very similar to that of the CNOT gate except that it contains two control inputs instead of one. When both control inputs have logic value '1', the target output takes the complement value of the target input, otherwise it takes the same value of the target input.

The Fredkin gate is also a three-input three-output reversible logic gate. It is different from the Toffoli gate in that it contains only one control input $X$ but two target inputs $Y$ and $Z$. When the control input of the Fredkin gate $X = 0$, outputs $Y' = Y$ and $Z' = Z$. Otherwise the target outputs swap the value of the target inputs, meaning that $Y' = Z$ and $Z' = Y$.

Both the Toffoli gate and the Fredkin gate are universal reversible gates because any logic function can be implemented by using only Toffoli gates or only Fredkin gates [18].

In this thesis we are only concerned with **Toffoli networks**, which are synthesized in the model of generalized Toffoli gates, i.e., Toffoli networks are constructed with only generalized Toffoli gates. The formal definition of a generalized Toffoli gate is as follows:

"For the set of domain variables $\{x_1, x_2, \cdots, x_n\}$ the **generalized Toffoli gate** has the form $TOF(C;T)$, where $C = \{x_{i1}, x_{i2}, \cdots, x_{ik}\}$, $T = \{x_j\}$ and $C \cap T = \phi$. It maps a Boolean pattern $(x_1^0, x_2^0, \cdots, x_n^0)$ to $(x_1^0, x_2^0, \cdots, x_{j-1}^0, x_j^0 \oplus x_{i1}^0 x_{i2}^0 \cdots x_{ik}^0, x_{j+1}^0, \cdots, x_n^0)$" ([32]). Here $C$ is the control set and $T$ is the target. The target output is of the opposite value of the target input value if all control inputs have logic value '1', otherwise it is of the same

value as that of the target input.

The NOT, CNOT, and Toffoli gates are special cases of generalized Toffoli gates. The NOT gate does not have any control input. The CNOT has one control input and the Toffoli gate has two.

Similarly, the generalized Fredkin gate can be defined as a reversible gate with two target inputs and $m$ control inputs, where $m \geq 0$. When all control inputs have logic value '1', the target outputs swap the value of the target inputs, otherwise the target outputs take the same value of the corresponding target inputs.

The SWAP gate and the Fredkin gate can be viewed as special cases of generalized Fredkin gates in that there is no control input in the SWAP gate and there is one control input in the Fredkin gate.

It is conventionally agreed that no fan-out or feedback is allowed in reversible logic circuits [34], so they are defined as a cascade structure (Figure 3-2).

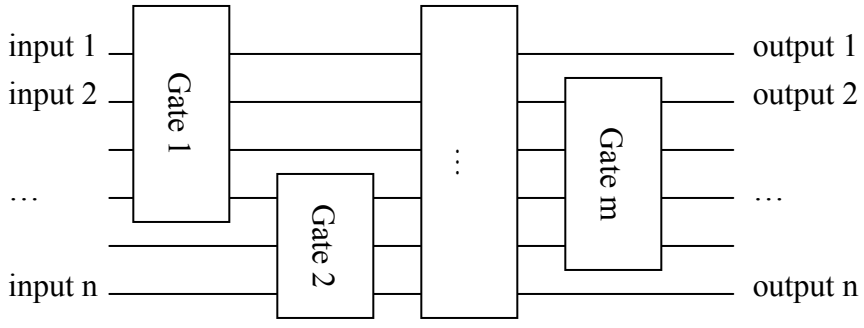



Figure 3-2 Reversible circuit: a cascade structure

The function of the circuit $f$ can be computed by propagating the signals from left

(inputs) to right (outputs). The inverse ( $f^{-1}$) of function $f$ can be easily computed by propagating the signals backwards from right (outputs) to left (inputs) because the inverse function of each reversible gate is exactly the same as its forward logic function ([26]).

A Toffoli network has a grid-like regular structure. Figure 3-3 shows the structure of a Toffoli network, which implements the function of a full adder. The circuit consists of two Toffoli gates and two CNOT gates.

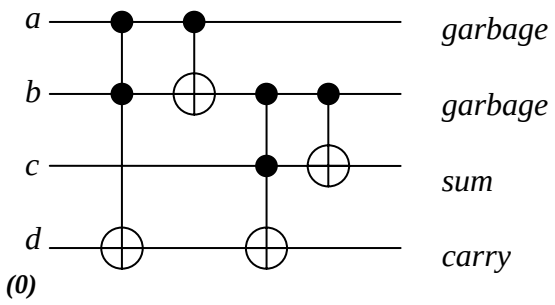



Figure 3-3 A Toffoli network: full adder

It is known that the original specification of a full adder contains only three inputs (two data bits *a, b*, and the carry in bit *c*) and two outputs (*sum* and *carry*). But in order to implement this function with reversible gates, one more input *d*, which has a constant logic value '0', and two more outputs, which are labeled as the garbage outputs, have to be added.

In the next section we explain the process of reversible logic synthesis. Here we only compute the logic function of outputs *sum* and *carry* in Figure 3-3 to verify that the circuit implements the desired specification.

$$sum = (a \oplus b) \oplus c = a \oplus b \oplus c$$

$$carry = ((a \oplus b)c) \oplus ((ab) \oplus 0) = (ac \oplus bc) \oplus (ab) = ac \oplus bc \oplus ab$$
$$= ab + bc + ac$$

Hence the output function is verified.

## 3.2 Reversible Logic Synthesis

Given the specification of a logic function, how can we implement it with reversible logic gates? In this section we give a basic idea about the process of reversible logic synthesis as well as several reversible logic synthesis approaches proposed by other researchers.

The first thing we need to do is build a reversible truth table based on the given specification of a function. During this step, inputs with constant value (logic '0' or logic '1') and/or garbage outputs might need to be added to the specification. Then a synthesis algorithm for reversible logic circuits should be used to generate the circuit with reversible gates. Different circuit structures may result from different synthesis algorithms. Also different synthesis algorithms are developed based on the availability of reversible gate libraries. For instance, some reversible logic synthesis algorithms aim on generating Toffoli networks. Others can generate reversible circuits with a larger variety of gates including Toffoli gates and Fredkin gates.

Below an example is given to illustrate the synthesis procedure.

We consider the full adder mentioned in Section 3.3. The specification contains three inputs (*a*, *b*, and *c*) and two outputs (*carry* and *sum*). The truth table of the full adder is

shown in Table 3-3.

TABLE 3-3 TRUTH TABLE OF A FULL ADDER

| *c* | *b* | *a* | *carry* | *sum* |
|---|---|---|---|---|
| 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 |

Obviously the function is not reversible since the number of outputs is not equal to the number of inputs. So at least one more output should be added to the function. This extra output is called a **garbage output** since it is not part of the specification of the function and it is used only for constructing a reversible function based on the original specification of the function. But is it enough to add just one garbage output? We notice that in the truth table of the full adder three output vectors are of the same pattern, which is "10" as shown in shaded area. So if only one garbage output is added, at least two output vectors would remain identical, either "100" or "101", meaning that the derived three-input three-output function would still be irreversible since there is no possible one-to-one matching between the input vectors and the output vectors. In this case, the solution is to introduce an additional input with constant value and one more garbage output to the function and construct the truth table in a way such as to make sure that there is a unique output pattern corresponding to each input pattern. So we could get a

truth table like the one shown in Table 3-4.

TABLE 3-4 TRUTH TABLE OF A FULL ADDER (REVERSIBLE LOGIC IMPLEMENTATION)

| *d* | *c* | *b* | *a* | *carry* | *sum* | *garbage* | *garbage* |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

Now it is a four-input four-output reversible function. The part in the shaded area is the original desired function of a full adder. As we can see, there are many different ways to arrange the output vectors so as to construct a reversible function. For instance, we can swap any two output vectors as long as they do not contain any of the shaded cells. Consequently, these actions will lead to different reversible functions and thus different implementations.

With a reversible function, the next step is to construct a circuit with reversible gates. This is reversible synthesis. There are many reversible techniques investigated recently by researchers. In Chapter 6 we explain our improved reversible synthesis algorithm in

detail. Here only a short description is given.

Maslov classified the synthesis methods in detail in his Ph.D. dissertation [26] into the following categories: composition methods, decomposition methods, factorization methods, EXOR logic based methods, genetic algorithms, spectral techniques, exhaustive search, etc. Most of the existing reversible logic synthesis algorithms fall in one or more of these categories. Please refer to his dissertation and the literature listed there for more detailed information.

Below we introduce the basic idea of several synthesis algorithms for reversible logic circuits proposed by other researchers.

- Exhaustive enumeration [46]: Shende, Prasad, Markov, and Hayes investigated this heuristic reversible logic synthesis method to construct reversible circuits with the CNT gate library (including the NOT gate, the CNOT gate, and the Toffoli gate) based on permutation theory. For any reversible logic function, the output vectors are a permutation of the input vectors. The authors gave some propositions on the relationship between permutation and the constructability of the given function with CNT gate library. This synthesis method generates optimal realizations for given reversible functions.
- ESOP approach [2]: This reversible logic synthesis algorithm proposed by Agrawal and Jha uses an Exclusive OR Sum of Products (ESOP) expression of the output function to construct the circuit. Since any Boolean function can be represented uniquely by a Positive-Polarity Reed-Muller (PPRM) form, and one output of the Toffoli gate implements an ESOP function, this algorithm uses a priority queue based search tree and selects successive Toffoli gates according

to the priority order of each output variable. This method produces near-optimal outputs for reversible functions of three inputs and can be extended to functions with more inputs.

- Transformation-based technique [32]: Miller, Maslov, and Dueck designed this transformation-based synthesis approach. It works on the reversible function in Boolean domain. The synthesis algorithm applies a generalized Toffoli gate to either the inputs or the outputs iteratively until the resulting reversible function represents the identity function. The identity function is defined as a function whose output is identical to its input. In this algorithm, the synthesis process can be done from inputs to outputs, from outputs to inputs, or in both directions. This transformation-based technique does not involve extensive searching and thus is good for synthesizing very large reversible logic functions.
- Spectral techniques [31]: Miller proposed an approach to synthesize Toffoli networks in the Rademacher-Walsh spectral domain. The algorithm iteratively applies a single generalized Toffoli gate to either an input or an output depending as to which leads to the better improvement in the resulting complexity metric. The synthesis ends when the derived Rademacher-Walsh spectra reach a unique variable or its complement. This approach generates good results for small functions.
- Use of templates [28]: Maslov, Dueck, and Miller used the technique of templates in their transformation-based reversible logic synthesis algorithms. A template is defined as a network which implements the identity function. It consists of two sequences of reversible gates. The main purpose of templates is

to find out the first sequence of reversible gates in the network generated by the synthesis algorithm and then replace it with the second sequence of reversible gates which is functionally equivalent, but simpler or smaller than the first sequence. Use of templates in synthesis algorithms usually produces more optimized outputs than the original synthesis algorithms.

In order to evaluate the performance of different reversible logic synthesis algorithms, the following criteria are often used:

- Number of garbage outputs: Although more garbage outputs does not necessarily result in more gates needed in implementing the given function or a higher cost of the implementation, in some cases the number of inputs/outputs are constrained by the implementation technologies as one more input requires one extra bit during computation. So some of the reversible synthesis algorithms target specifically towards generating reversible circuits with minimum garbage outputs.
- Number of gates: The number of gates required to implement a given function, of course, is directly related to how much hardware resource will be needed for the implementation. A circuit is said to be optimal if none of its functionally equivalent alternatives has a lower cost. In many cases, a circuit generated with the minimum number of gates is considered to be the optimal realization [46] of the given function. So the number of gates generated is also an important aspect to consider when evaluating the quality of different synthesis approaches.
- Quantum cost of the circuit: We know that different gates require different area

sizes and consume different amounts of energy. Also a reversible logic function can be implemented with different libraries which contain different sets of reversible logic gates. So the number of gates used in realizing a function is not accurate in representing the cost of the circuit. The quantum cost [6][26] for each reversible gate is calculated based on how many one-bit and two-bit quantum gates from a universal gate set are needed to implement it. The sum of the quantum cost of all gates in the circuit, or the quantum cost of the whole circuit, is a more accurate measure in defining the cost of the circuit. The quantum costs of different generalized Toffoli gates are summarized in Maslov's reversible benchmarks page [27].

- Runtime: Simulation runtime is widely used to assess the efficiency of different algorithms. It is often used to compare the performance among a variety of synthesis algorithms. It also implies other features of the synthesis algorithms such as whether they are scalable.

Benchmark functions are often used to carry out the assessment of different synthesis algorithms. One set of benchmark functions and some of their implementations can be found at Maslov's website [27]. In this thesis, all our simulation work is conducted using this set of benchmark circuits.

## 3.3 Testing Reversible Circuits

While research on reversible logic synthesis has been popular recently, the issue of testing reversible logic circuits also has gained some attention. Since reversible logic is synthesized with reversible gates whose logic structure is quite different from that of traditional irreversible gates such as NAND and XOR, the implementation technologies of reversible circuits may also be different from those of irreversible circuits. It raises a question as to whether the traditional fault models such as the stuck-at fault model are still adequate to represent the effect of errors on reversible circuits. If not, shall we define new fault models for reversible circuits? How to detect these faults? Before introducing our work in testing reversible logic circuits, we first summarize briefly the previous work related to this area.

Recent work related to testing reversible logic circuits includes ATPG techniques for classical stuck-at faults in reversible circuits, new fault models proposed for reversible circuits and ATPG methods targeting these new fault models, testing strategies for quantum circuits, etc.

As mentioned in Section 3.1, the structure of reversible logic circuits is regular and much simpler than that of conventional irreversible circuits. Also there is no fan-out or feedback in the system. These features make it easier to test reversible circuits.

Patel et al. [37] study the stuck-at fault model in reversible logic circuits and introduce two approaches to test stuck-at faults in reversible circuits. Their work shows that the TPG problem is significantly simplified for reversible circuits because of reversibility. The two methods they use to generate minimum test set for all stuck-at faults in

reversible circuits are integer linear programming (ILP) and decomposition. The former approach formulates an ILP with binary variables in order to generate minimum test sets. The latter approach works together with the former one to find minimum test sets for large circuits since the ILP method itself is not computationally feasible for large circuits.

Some important properties related to testing stuck-at faults in reversible logic circuits [37] are listed below as they are used in later parts of this thesis.

- A test set which covers all single stuck-at faults also detects all multiple stuck-at faults.
- A test set is complete for all single stuck-at faults in a reversible logic circuit if and only if this test set can set each line in the circuit to both logic 0 and logic 1.
- Each test vector detects exactly half of all possible single stuck-at faults. Each single stuck-at fault is detected by exactly half of all possible test vectors.

Patel et al. also introduced a new fault model, the cell fault model, for reversible logic circuits. The cell fault model is defined as situations where a reversible logic gate changes arbitrarily from its desired function. The ILP method previously mentioned for stuck-at faults can be modified to generate minimum test sets for the cell faults in reversible logic circuits.

Another new fault model, the missing-gate fault (MGF) model, is proposed by Hayes et al. [19][44] based on quantum technology. It is defined as the complete removal of a gate from the circuit. It was shown that missing-gate faults are highly detectable. In [19] the authors also defined two other fault models in the same family based on possible physical defect in quantum technology: the repeated gate fault (RGF) model and the partially missing-gate fault (PMGF) model. Obviously, the repeated gate fault model represents

situations where a gate is duplicated and the partially missing-gate fault represents situations where the size of a gate is reduced since part of the gate is missing.

Allen et al. [3] brought up two more new fault models for reversible logic circuits: the lock gate fault (LGF) model and the control switch stuck fault (CSSF) model. These two fault models are defined based on the CMOS technology used to physically construct reversible logic circuits. ATPG techniques for these fault models are also provided and compared in [3].

Fault localization in reversible circuits is analyzed in [45]. The authors use an adaptive tree to detect and locate the single stuck-at faults in a reversible logic circuit. Their investigation shows that the problem of fault localization is easier for reversible logic circuits than for classical irreversible logic circuits because each test vector covers exactly half of the single stuck-at faults in reversible logic circuits and thus the fault table contains a large number of '1's.

Pierce et al. [42][43] also proposed a method to generate fault localization trees for faults in reversible logic circuits in a fault model independent environment. In this method, the fault tables are stored with decision diagrams which saves memory such that it could be applied to much larger circuits.

Chakraborty investigated the testability of Toffoli networks under the stuck-at fault model in [12]. The testability for the single stuck-at fault model is examined by modeling a generalized Toffoli gate as a multiple input AND gate and a two-input XOR gate. It was proved that a $n$-input reversible logic circuit has a universal test set of size $n^2 + 2n + 2$. Here a universal test set is defined as the family of test sets which covers all Toffoli networks with the same number of inputs.

The problem of fault tolerance in reversible logic circuits is addressed in [49]. The authors propose three new reversible logic gates and illustrate that reversible logic circuits constructed with these reversible gates are on-line testable using two pair rail checkers which are also implemented with these new reversible gates. Other research with a similar focus is reported in [36].

Since quantum computing is reversible, the testing of quantum circuits has also been the subject of considerable research. Previous work related to testing quantum circuits includes fault detection and localization [8][9][39], and fault modeling [19].

## 3.4 Conclusion

In this chapter we gave an overview of reversible logic, basic reversible logic gates, and structure of reversible logic circuits. We particularly discussed the definition of generalized Toffoli gates and Toffoli networks which are constructed with generalized Toffoli gates, as our main concern in this thesis is fault modeling and detection in Toffoli networks.

We also explained the steps of reversible logic synthesis. Different approaches to reversible logic synthesis, as well as existing work in reversible logic synthesis, were presented briefly.

Related work in testing reversible logic circuits, such as fault models and fault-detection strategies, fault localization methods, design for testability and fault tolerance in reversible logic circuits, were also included so as to give a background for the coming chapters.

# Chapter 4 Crosspoint Fault Model

In Chapter 3 we introduced background on reversible logic circuits including the definition of reversible logic, commonly used reversible logic gates, reversible synthesis techniques, and existing work in testing reversible circuits, especially the testing of stuck-at faults in reversible circuits. In this chapter, we show that the stuck-at fault model itself is not adequate in representing effects of different faults in the circuit. So we present a new fault model, namely the crosspoint fault model, for reversible logic circuits. Properties of reversible circuits under the crosspoint fault model are investigated and a randomized TPG algorithm targeting this specific fault model is explained and analyzed in detail. Finally, experimental results are presented to support the hypothesis that our TPG algorithm yields very good performance in testing crosspoint faults in reversible logic circuits.

## 4.1 Crosspoint Fault Model

In this section we demonstrate that the stuck-at fault model itself is not adequate to represent all error effects in reversible circuits. After that we introduce a new fault model, namely the crosspoint fault, for reversible logic circuits based on the logic structure of generalized Toffoli gates.

It was mentioned in Section 2.2.2 that in traditional irreversible circuits, stuck-at faults do not only represent errors which happen to the wires in a circuit, they also cover other situations such as internal defects of a gate. However, in reversible logic circuits such as

in Toffoli networks, stuck-at faults do not cover the situations where some of the gates themselves do not function properly. For example, Figure 4-1 shows the structure of a simple reversible circuit, which contains two generalized Toffoli gates. It is easy to verify that the test set {0110, 1001, 1000} is complete for all stuck-at faults in the circuit. But assume that there are two errors in the first gate, which are: (1) the second control point in the gate is missing; (2) the third control point is missing. Assume that these two errors do not happen at the same time. It can be verified that the test set {0110, 1001, 1000} detects neither of them. This example illustrates the deficiency of the stuck-at fault model in describing different errors in reversible circuits.

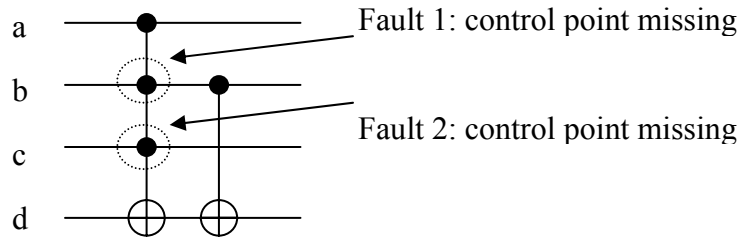



Figure 4-1 Stuck-at fault model is inadequate for reversible circuits

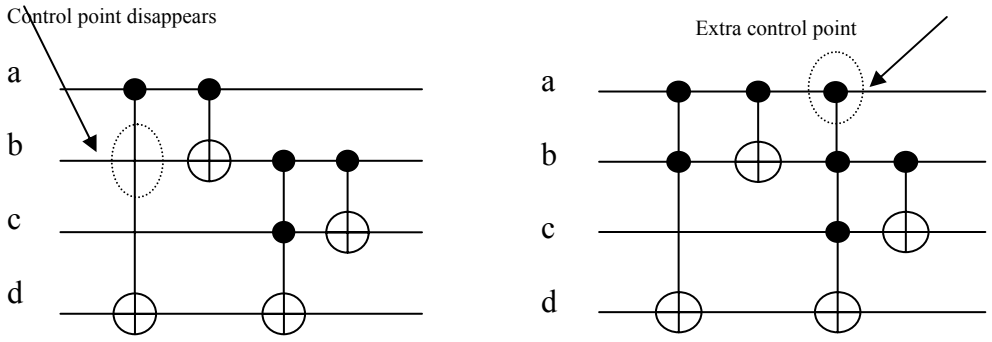
In this case, a new fault model is needed to represent the behavior of these errors.

As introduced in Section 2.2.2, fault models map the physical faults to the logical behavior of the system. However, the implementation technologies of reversible circuits are not yet fully developed, so we think it is valuable to define a new fault model for reversible logic circuits based on their logic structure. Given that Toffoli networks have a grid-like or array-based regular structure, we are proposing a new fault model, namely the crosspoint fault model, which maps the situations where internal defects exist in a

gate such that the control points do not function correctly. The faulty situations can be either missing or insertion of some control points in a generalized Toffoli gate.

A **crosspoint fault** in a reversible logic circuit is defined as either an extra or a missing control point in a generalized Toffoli gate. It is classified into two categories: disappearance faults and appearance faults.

A **disappearance fault** happens when a control point in a Toffoli gate missing. For example, consider the structure of the full adder shown in Figure 3-3, in the first Toffoli gate a disappearance fault happens if the control point from line b is missing. This fault changes the Toffoli gate into a CNOT gate (as shown in Figure 4-2a). It transpires that this disappearance fault model is the same as the partial missing-gate fault model introduced in [19].




a. Disappearance fault b. Appearance fault

Figure 4-2 Crosspoint faults

Similarly, an **appearance fault** happens when an extra control point is introduced to a Toffoli gate. For instance, consider the same fuller adder in Figure 3-3, a possible appearance fault can be adding one additional control point from line a to the third gate,

(see Figure 4-2b). This appearance fault results in transforming the 2-input Toffoli gate into a 3-input Toffoli gate.

**Multiple crosspoint faults** are defined as situations where two of more single crosspoint faults exist in a reversible circuit at the same time. They are categorized into three types: multiple disappearance faults, multiple appearance faults, and mixed multiple faults. A **multiple disappearance fault** happens when at least two control points disappear from the circuit at the same time. Similarly, a **multiple appearance fault** means that two or more extra control points are added to the circuit. And a **mixed multiple crosspoint fault** happens when both appearance fault and disappearance fault exist in a reversible circuit at the same time.

We can also group multiple crosspoint faults into double faults, triple faults, etc., based on the multiplicity of the faults.

The crosspoint fault model is suitable for describing the faults that can happen on the control points of generalized Toffoli gates, but it is also suitable for other reversible logic gates with control points such as the Fredkin gates. In this thesis, we are only concerned with Toffoli networks, thus the following properties related to crosspoint faults are derived for Toffoli networks. However, some of the properties may also be relevant for reversible logic circuits which contain other types of gates such as Fredkin gates.

## 4.2 Testing Crosspoint Faults

As the crosspoint fault model has been introduced in the previous section, in this section we discuss how the crosspoint faults affect the behavior of reversible circuits and

how to test for these faults. A randomized ATPG algorithm targeting crosspoint faults is proposed and analyzed in detail. The performance of the ATPG algorithm is evaluated by simulation studies.

### 4.2.1 Properties

Since there is a one-to-one mapping between the input vectors and the output vectors of a reversible gate, it is always possible to activate a crosspoint fault and propagate its effect to the primary outputs. The following three properties are straightforward based on this fact.

**Property 1: A crosspoint fault in a reversible circuit results in a reversible circuit.**

This property is trivial, since the disappearance of one control point or the addition of one more control point to the generalized Toffoli gate makes the new gate the same type, which is still a generalized Toffoli gate except that the number of control points is changed. The new gate is still reversible.

**Property 2: All single crosspoint faults are detectable.**

This property is also obvious. For any single crosspoint fault, we can always find an input vector at the corresponding faulty gate so as to detect the fault at that stage (line-justification). And, it is straightforward to track this justified vector backward to the primary inputs (line-justification) as well as to propagate the fault effect to the primary outputs (propagation). Thus all single crosspoint faults are detectable.

***Line-justification:***

Figure 4-3 shows a generalized Toffoli gate at a certain stage in a circuit, where $c_1$ to $c_n$

are control inputs of the gate, d is the target input or data input of the gate, $a_1$ to $a_m$ are wires not connected to the gate. The justification rules used to detect the single crosspoint faults at the corresponding stage are summarized in Table 4-1.

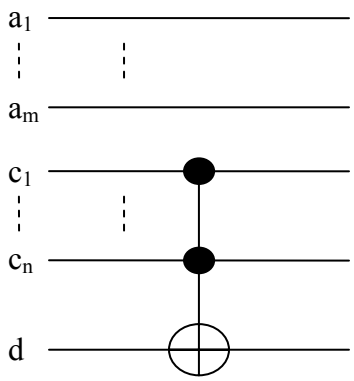



Figure 4-3 Generalized Toffoli gate at a certain stage in a circuit

TABLE 4-1 JUSTIFICATION RULES

| Gate | Fault | Justification rules |
|---|---|---|
| Generalized Toffoli | $c_i$ disappearing | $c_i = 0$<br>$c_j = 1$, j = 1, 2, … , i-1, i+1, … , n<br>$a_q$ : don't care, q = 1, 2, … , m<br>d: don't care |
| | $a_p$ appearing | $a_p = 0$<br>$c_j = 1$, j = 1, 2, … , n<br>$a_q$ : don't care, q = 1, 2, … , p-1, p+1, … , m<br>d: don't care |

Reversibility makes it much easier to find a test vector at the primary inputs in reversible circuits as against conventional irreversible circuits. We can easily compute the inverse function of a reversible gate from its forward function. The inverse of a Generalized Toffoli gate is exactly the same as its forward function. So, from the justified vector at the stage of the faulty gate, by applying the inverse function of each gate backwards, we can find a test vector at the primary inputs which is able to justify the specified crosspoint fault in the circuit.

***Propagation:***

Similarly, the propagation of the fault effect to the primary outputs is always feasible for Toffoli networks. The justified vector at the stage of the faulty gate creates different output vectors for the faulty and fault-free gate. Then, according to the property that there is a one-to-one correspondence between the input vectors and the output vectors in any reversible logic function, the vector at the primary outputs of the faulty circuit is always different from that of the fault-free circuit.

**Property 3: The number of gates in a circuit is an upper bound on the size of a complete test set for all single appearance crosspoint faults in this circuit.**

From Figure 4-3 we can see that, if $c_1$ to $c_n$ are set to logic 1, $a_1$ to $a_m$ set to logic 0, d set to either logic 1 or logic 0, this vector covers all single appearance faults at this stage. So the total number of test vectors needed for all single appearance faults in the circuit does not exceed the number of gates.

### 4.2.2 ATPG Algorithm

There are a variety of methods to generate a complete test set for all single crosspoint faults. A traditional ATPG algorithm could be used to generate a test vector for each individual fault, and then compress the test set (see below).

1. Read in the specification of a circuit, generate a complete fault list F
2. For each fault $f_i$ in F, do:
    a. Find a vector $v_i$ which detects $f_i$
    b. Add $v_i$ to V
    c. Delete $f_i$ from list F
3. Repeat 2 until F is null
4. Compress V
5. Return V

**Traditional ATPG Algorithm**

Obviously, the traditional ATPG algorithm is easy to implement and generates a complete test set for all faults. But usually the test set is not minimal. Exhaustive algorithms are often used to find a minimum test set. However, exhaustive search leads to exponential increase of searching space and execution time when the number of faults grows. In the following, we give a randomized heuristic ATPG algorithm which does not do an exhaustive search yet still generates a minimal (or near-minimal) test set for crosspoint faults. Compared to the greedy approaches, the randomized algorithm greatly reduces complexity and running time. The detailed algorithm is shown below.

1. Read in the specification of a circuit, generate a complete fault list F
2. Get a new fault list F' by randomly choosing N=10 faults from F (if size(F)<=N, then N=size(F)), do
    a. For each fault $f_i$ in F', find a vector $v_i$ which detects $f_i$
    b. Use each $v_i$ to simulate the circuit, find out how many faults can be detected by $v_i$, record this number as $N_i$
    c. Find out the vector $v_j$ which has the maximum $N_j$, if $v_j$ is not in the test set V, add $v_j$ to V
    d. Delete all the faults which are covered by $v_j$ from list F
3. Repeat 2 until F is null
4. Return V

**ATPG Algorithm**

This randomized ATPG algorithm for crosspoint faults takes advantage of both random algorithms and greedy algorithms. To avoid a large amount of computation, instead of testing all faults, in every round we choose a fixed number of faults and generate a test vector for each of them. Among these test vectors, we select one which has the highest fault coverage for all faults remaining on the fault list. Then this test vector is included in the test set. All faults covered by it are deleted from the fault list. These steps are repeated until the fault list is empty. For the purpose of randomization, we include a parameter N, which is the number of faults chosen randomly in each round, in this algorithm. The value of N affects the performance of the algorithm: a bigger N might lead to a smaller test set, but it also means longer running time. Experiments show that 10 is a good value for N since in this case the algorithm generates a near minimum test set in a comparably small time.

### 4.2.3 Experimental Results

Simulation studies were conducted on a set of standard benchmark circuits ([27]) in order to evaluate the performance of our ATPG algorithm. Table 4-2 shows the size of the test sets generated by our algorithm. Table 4-3 compares results generated by our ATPG and those generated by the traditional algorithm. It is also illustrated how parameter N affects the performance of the algorithm.

Table 4-2 shows the experimental results of testing single crosspoint faults using our ATPG algorithm. N=10 for these experiments.

The columns of Table 4-2 correspond to the name of the benchmark circuit; the number of inputs of the circuit; the number of gates in the circuit; the number of total single disappearance faults in the circuit; the number of vectors generated by our randomized ATPG algorithm to cover all disappearance faults; the number of total single appearance faults in the circuit; the size of the test set for appearance faults; the size of the test set to cover both types of crosspoint faults.

TABLE 4-2 SIMULATION RESULTS FOR SINGLE CROSSPOINT FAULTS

| Circuit name | inputs | gates | Disappearance faults | | Appearance faults | | Combined |
|---|---|---|---|---|---|---|---|
| | | | # of faults | Test set | # of faults | Test set | Test set |
| 2of5 | 6 | 18 | 24 | **9** | 66 | **6** | **15** |
| 2of5_2 | 7 | 12 | 19 | **4** | 53 | **9** | **8** |
| 3bit adder | 4 | 4 | 6 | **2** | 6 | **3** | **4** |
| 3_17 | 3 | 6 | 7 | **2** | 5 | **2** | **3** |
| 4_49 | 4 | 16 | 24 | **5** | 24 | **5** | **7** |
| 4mod5_1 | 5 | 8 | 12 | **2** | 20 | **4** | **4** |
| 4mod5_2 | 5 | 9 | 11 | **2** | 25 | **6** | **5** |
| 4mod5_3 | 5 | 5 | 6 | **2** | 14 | **3** | **4** |
| 5mod5_1 | 6 | 17 | 38 | **5** | 47 | **9** | **5** |
| 5mod5_2 | 6 | 10 | 18 | **5** | 32 | **5** | **7** |
| cycle10_2 | 12 | 19 | 100 | **16** | 109 | **4** | **18** |
| cycle17_3 | 20 | 48 | 409 | **44** | 503 | **7** | **54** |
| ham3 | 3 | 5 | 6 | **3** | 4 | **3** | **3** |
| ham7 | 7 | 23 | 34 | **5** | 104 | **6** | **6** |
| ham15 | 15 | 13 | 352 | **17** | 1496 | **27** | **33** |
| hwb4 | 4 | 17 | 27 | **5** | 24 | **5** | **5** |
| hwb5 | 5 | 55 | 101 | **10** | 119 | **10** | **13** |
| hwb6 | 6 | 12 | 320 | **17** | 310 | **20** | **21** |
| hwb7 | 7 | 28 | 864 | **27** | 870 | **33** | **37** |
| mod5adder | 6 | 21 | 27 | **6** | 78 | **7** | **12** |
| mod1024adder | 20 | 55 | 220 | **39** | 825 | **19** | **45** |
| mod1048576add | 40 | 21 | 1540 | **175** | 6650 | **45** | **196** |
| rd53_1 | 7 | 30 | 41 | **12** | 139 | **10** | **17** |
| rd53_2 | 7 | 12 | 28 | **8** | 44 | **9** | **9** |
| rd53_3 | 8 | 12 | 20 | **4** | 64 | **9** | **9** |
| rd53_4 | 7 | 28 | 24 | **10** | 144 | **9** | **13** |
| rd53_5 | 7 | 16 | 27 | **5** | 69 | **7** | **7** |
| rd73 | 10 | 20 | 34 | **4** | 146 | **15** | **13** |
| rd84 | 15 | 28 | 49 | **4** | 343 | **22** | **19** |
| xor5 | 5 | 4 | 4 | **1** | 12 | **4** | **4** |
| hwb8 | 8 | 63 | 2214 | **50** | 2245 | **56** | **67** |
| hwb9 | 9 | 15 | 5812 | **85** | 6540 | **98** | **110** |
| hwb10 | 10 | 36 | 15455 | **137** | 17224 | **168** | **180** |
| hwb11 | 11 | 93 | 39211 | **243** | 53929 | **293** | **324** |

From Table 4-2, we can see that, our ATPG algorithm yields very good performance in generating test set for crosspoint faults. For all the benchmark circuits, the size of a complete test set for either all single disappearance crosspoint faults or all single appearance faults is always smaller than the number of gates in the circuit. The test sets produced for some of the benchmark circuits are amazingly small. For instance, for two small circuits, our algorithm shows that only one or two test vectors are sufficient to cover all single disappearance faults in the circuit. Even when testing both types of crosspoint faults, the combined test sets are still very small. Actually, for 33 out of the 34 benchmark circuits, the combined test size is no bigger than the number of gates in the circuit.

As each time N faults are randomly chosen, it could result in situations where the combined test set is smaller than the test set for appearance faults (for instance, circuits 2of5_2 and 5mod5_1) and the combined test set is bigger than combination of the test set for disappearance faults and the test set for appearance faults (for example, circuit cycle17_3).

Table 4-3 shows the superior performance of our randomized ATPG algorithm as compared to the traditional ATPG method described in the beginning of Section 4.2.2. In this table, figures in columns 2 and 5 are test sets generated with the traditional TPG method, figures in columns 3 and 6 are generated with our randomized ATPG algorithm. Columns 4 and 7 give the improvement of our ATPG over the traditional one in percentage.

TABLE 4-3 OUR TPG ALGORITHM VS TRADITIONAL TPG METHOD

| Circuit name | Disappearance faults | | | Appearance faults | | |
|---|---|---|---|---|---|---|
| | Test set traditional | Test set new | Improve-ment (%) | Test set traditional | Test set new | Improve-ment (%) |
| 2of5 | 18 | **9** | 50.0 | 17 | **6** | 64.7 |
| 2of5_2 | 12 | **4** | 66.7 | 12 | **9** | 25.0 |
| 3bit adder | 4 | **2** | 50.0 | 4 | **3** | 25.0 |
| 3_17 | 4 | **2** | 50.0 | 3 | **2** | 33.3 |
| 4_49 | 8 | **5** | 37.5 | 8 | **5** | 37.5 |
| 4mod5_1 | 7 | **2** | 71.4 | 8 | **4** | 50.0 |
| 4mod5_2 | 5 | **2** | 60.0 | 8 | **6** | 25.0 |
| 4mod5_3 | 4 | **2** | 50.0 | 4 | **3** | 25.0 |
| 5mod5_1 | 9 | **5** | 44.4 | 16 | **9** | 43.8 |
| 5mod5_2 | 9 | **5** | 44.4 | 9 | **5** | 44.4 |
| cycle10_2 | 18 | **16** | 11.1 | 19 | **4** | 78.9 |
| cycle17_3 | 46 | **44** | 4.3 | 48 | **7** | 85.4 |
| ham3 | 4 | **3** | 25.0 | 4 | **3** | 25.0 |
| ham7 | 11 | **5** | 54.5 | 18 | **6** | 66.7 |
| ham15 | 74 | **17** | 77.0 | 93 | **27** | 71.0 |
| hwb4 | 8 | **5** | 37.5 | 12 | **5** | 58.3 |
| hwb5 | 21 | **10** | 52.4 | 26 | **10** | 61.5 |
| hwb6 | 41 | **17** | 58.5 | 58 | **20** | 65.5 |
| hwb7 | 83 | **27** | 67.5 | 109 | **33** | 69.7 |
| mod5adder | 16 | **6** | 62.5 | 13 | **7** | 46.2 |
| mod1024adder | 48 | **39** | 18.8 | 55 | **19** | 65.5 |
| mod1048576adder | 193 | **175** | 9.3 | 210 | **45** | 78.6 |
| rd53_1 | 27 | **12** | 55.6 | 26 | **10** | 61.5 |
| rd53_2 | 14 | **8** | 42.9 | 12 | **9** | 25.0 |
| rd53_3 | 12 | **4** | 66.7 | 12 | **9** | 25.0 |
| rd53_4 | 18 | **10** | 44.4 | 17 | **9** | 47.1 |
| rd53_5 | 13 | **5** | 61.5 | 14 | **7** | 50.0 |
| rd73 | 20 | **4** | 80.0 | 20 | **15** | 25.0 |
| rd84 | 28 | **4** | 85.7 | 28 | **22** | 21.4 |
| xor5 | 4 | **1** | 75.0 | 4 | **4** | 0.0 |
| **Average** | | | **50.5** | | | **46.7** |

Obviously, our ATPG algorithm performs much better than the traditional ATPG method. For some of the circuits, the improvement is significant. When testing disappearance faults, for 19 out of the 30 benchmark circuits, the test sets generated by our ATPG algorithm have a size which is no more than half of that generated by traditional ATPG method (improvement >=50%). For the appearance faults, 14 out of 30 benchmark circuits have this property.

For both disappearance faults and appearance faults, the average improvement of our ATPG over the traditional one is about 50%.

In Section 4.1 we noted that the disappearance fault model we proposed includes the missing-gate fault model introduced in [19]. In that paper the author used an ATPG method which is based on linear programming to generate minimum test sets for partial missing-gate faults. Comparison between our results and the minimum test sets from [19] shows that our randomized ATPG algorithm does generate minimal or near-minimal test sets for the benchmark circuits (see Table 4-4). Our ATPG algorithm produces minimum test sets for 14 out of the 21 benchmark circuits. For the remaining 7 benchmark circuits, the difference between the size of test sets generated with our ATPG and that of the minimum test sets is very small.

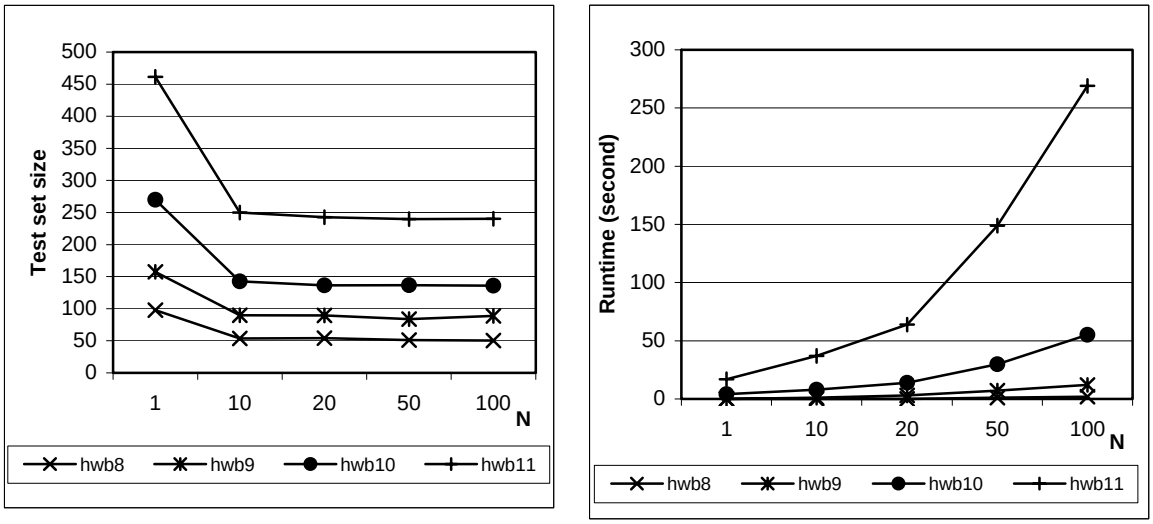
TABLE 4-4 RANDOMIZED ATPG GENERATES NEAR-MINIMAL TEST SET

| Circuit name | Test set size for disappearance faults | |
|---|---|---|
| | our alg. | min [19] |
| 2of5 | 9 | 8 |
| 2of5_2 | 4 | 3 |
| 3_17 | 2 | 2 |
| 4_49 | 5 | 5 |
| 4mod5_1 | 2 | 2 |
| 4mod5_2 | 2 | 2 |
| 5mod5_1 | 5 | 5 |
| ham3 | 3 | 3 |
| ham7 | 5 | 4 |
| hwb4 | 5 | 5 |
| hwb5 | 10 | 9 |
| hwb6 | 17 | 15 |
| hwb7 | 27 | 24 |
| mod5adder | 6 | 6 |
| rd53_1 | 12 | 8 |
| rd53_2 | 8 | 8 |
| rd53_3 | 4 | 4 |
| rd73 | 4 | 4 |
| rd84 | 4 | 4 |
| xor5 | 1 | 1 |

The runtimes of our ATPG algorithm and those of the ATPG method proposed in [19] are not compared in this thesis since in [19] only those relatively small benchmark circuits are tested. It usually takes a very small amount of time to generate minimum test sets for these small circuits. However, we think that our ATPG algorithm is more efficient in simulating much larger circuits because the computational complexity of the randomized approach is smaller than that of the method proposed in [19], which is

basically an exhaustive search. This also implies that our ATPG algorithm could handle much larger circuits than the approach introduced in [19]. In [19], the largest circuits tested are ham15tcl (15 inputs with 162 gates) and hwb7tc (7 inputs with 305 gates). However, our algorithm could work with much larger circuits such as mod1048576adder (40 inputs with 210 gates) and hwb11 (11 inputs and 9314 gates).

Figure 4-4 illustrates how the parameter N affects the performance of our randomized ATPG algorithm. Figure 4-4(a) shows the size of the test set generated for disappearance faults for several large benchmark circuits. Figure 4-4(b) shows the corresponding runtimes. When N equals 1, the test sets generated are always the worst since they have the biggest size. For small circuits, the size of the test sets produced does not vary too much when N is equal to 10, 20, 50, or 100. On the other hand, for large circuits, when N grows from 10 to 20, to 50, and finally to 100, the average size of the test sets becomes slightly smaller, while the execution time increases exponentially. Considering both factors of test set size and runtime, we decide that either 10 or 20 is a good choice for N since in these conditions the algorithm yields quite satisfactory outputs in a comparably small amount of time. In our simulation, we use the value 10 for parameter N.




a. Test set size b. Runtime

Figure 4-4 ATPG algorithm and N

From the runtime curves shown in Figure 4-4(b) we can see that our ATPG algorithm is very efficient. The simulation was conducted on a Sun Ultra 5 with 400 MHz Ultra SPARC II processor and 256 MB memory. For most of the benchmark circuits tested, the runtime is less than one second. For the largest circuit, hwb11, which contains 9314 gates and 93140 single crosspoint faults, the runtime is 37 seconds, 46 seconds, and 60 seconds for all single disappearance faults, all single appearance faults, and all single crosspoint faults, respectively (when N equals to 10). These figures show that our randomized ATPG algorithm for crosspoint faults in reversible circuits is very efficient and is able to work for much bigger circuits.

## 4.3 Crosspoint Faults and Stuck-at Faults

After studying the features of our new fault model, the crosspoint fault model, in reversible logic circuits, in this section we investigate the relationship between crosspoint faults and stuck-at faults and investigate whether this crosspoint fault model is a better fault model than the stuck-at fault model for reversible logic circuits for the perspective of covering other types of faults.

### 4.3.1 Relationship between Crosspoint Faults and Stuck-at Faults

The relationship between the crosspoint fault and the stuck-at faults is summarized in the following property.

**Property 4: A complete test set for all single crosspoint faults covers the majority of the single stuck-at faults. The proportion of stuck-at faults covered is at least (2n+k-1)/(2n+2k), where n is the number of inputs, and k is the number of gates in the circuit.**

The detailed proof of this property is given below.

It is proved in [37] that under the single stuck-at fault model a test set is complete if, and only if, each wire at every level can be set to both logic 0 and logic 1 by the test set. Consequently, in order to prove property 4 we need to show that most wires in a Toffoli network can be set to both logic 0 and logic 1 by the complete test set generated for all single crosspoint faults.

In a reversible circuit implemented with generalized Toffoli gates, a wire exists in one of the following three ways:

1. It is connected to one of the control points of a gate which has at least two control inputs. Consider, for instance, lines *a*, *d*, *f*, and *g* in Figure 4-5, which is a simple Toffoli network consisting of eight wires (*a*, *b*, *c*, *d*, *e*, *f*, *g*, and *h*) and three crosspoints (1, 2, and 3). In this case, each wire can be set to both logic 0 and logic 1 by a complete test set for all single disappearance faults: it is set to logic 0 by any test vector which detects the disappearance fault at the crosspoint it crosses; it is set to logic 1 by any test vector which detects the disappearance fault of a crosspoint other than the one it crosses in the gate. For example, line *d* in Figure 4-5 can be set to logic 0 by a test vector which covers the fault 'crosspoint 3 is missing' and set to logic 1 by a test vector which covers the fault 'crosspoint 2 is missing'. So both single stuck-at faults 'line *d* s-a-1' and 'line *d* s-a-0' are covered by a complete test set for all single crosspoint faults.

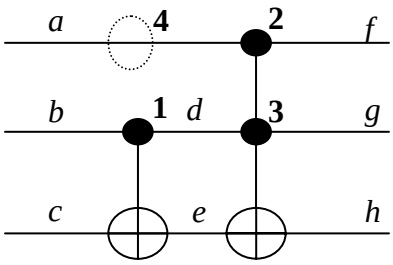



Figure 4-5 Crosspoint faults and stuck-at faults

2. It is connected to the control point of a gate which has only one control input. In this case, the wire can be set to both 0 and 1 by a complete test set for all single

crosspoint faults: it is set to 0 by the test vector which detects the disappearance fault at the crosspoint it crosses; it is set to 1 by the vector which detects the appearance fault of a crosspoint at the same stage as the gate. For example, line *b* in Figure 4-5 can be set to logic 0 by any test vector which detects the disappearance fault 'crosspoint 1 is missing'. It can also be set to logic 1 by any test vector which covers the appearance fault 'crosspoint 4 is added to the CNOT gate'. Consequently, both single stuck-at faults on line *b* are covered by a complete test set for all crosspoint faults in the circuit.

3. It is not connected to any control point of any gate. There are three possibilities: (i) it connects a primary input and a data input of a gate, like line *c* in Figure 4-5; (ii) it connects the data output of a gate and the data input of another gate, like line *e* in Figure 4-5; (iii) it connects the data output of a gate and a primary output, like line *h* in Figure 4-5. These three cases are the only conditions where some (less than 50%) of the single stuck-at faults on these wires may not be covered by a test set generated for all single crosspoint faults. Given that single stuck-at faults are very easy to detect (every fault is covered by half of the total number of input vectors [37]), we expect that in this case most of the single stuck-at faults will still be detected by a complete test set generated for all single crosspoint faults.

Property 4 gives a lower bound of the fault coverage for stuck-at faults by a complete test set for all crosspoint faults. However, in practice, the fault coverage is much higher given that stuck-at faults in reversible circuits are much easier to test than in conventional circuits. This property is well demonstrated by the experimental results we discuss below.

However, it should be noted that there is no guarantee that all single stuck-at faults are

covered by a complete test set for all single crosspoint faults. As a simple illustration, consider the circuit shown in Figure 4-5. {100, 011} is a complete test set for all single crosspoint faults, but it does not cover the single stuck-at faults: line *d* stuck-at-0 and line *e* stuck-at-0.

Based on the relationship between crosspoint faults and stuck-at faults, we propose another method to generate a complete test set for stuck-at faults in Toffoli networks. We could use a complete test set for all single crosspoint faults in a reversible logic circuit as a very good starting point for a test set for all stuck-at faults in the same circuit as it covers a majority of them. For the remaining stuck-at faults not covered, an easy fault-oriented TPG algorithm is able to find the test vectors for them. Combining both test sets we get a complete test set for all stuck-at faults in the circuit.

### 4.3.2 Experimental Results

Experiments were conducted to examine the test sets generated for stuck-at faults and crosspoint faults. First we use a complete test set for all crosspoint faults to test the single stuck-at faults in the same circuit. Then, we use a complete test set for all stuck-at faults to test the single crosspoint faults in the circuit. We use a simple ATPG algorithm[1] to produce a complete test set for stuck-at faults in reversible logic circuits. The detailed algorithm is given below.

---

[1] Different TPG algorithms result in slightly different sized test sets. For instance, J. Biamonte [9] has indicated they use a greedy algorithm and get test sets that are slightly smaller than ours.

1. Read in the specification of a circuit, generate a complete fault list F
2. Generate a random vector $v_1$ and its complement $v_1$'. Add $v_1$ and $v_1$' to test set V. Use these two vectors to stimulate the circuit, and delete all the faults which are covered by these two vectors from list F
3. Randomly choose a fault $f_i$ from F, do:
    a. Find a vector $v_i$ which detects $f_i$
    b. Use $v_i$ to stimulate the circuit. Delete all faults which are covered by $v_i$ from list F
    c. If $v_i$ is not in V, add $v_i$ to V
4. Repeat 3 until F is null
5. Return V

**ATPG Algorithm for Stuck-at Faults**

Simulation results are given in Table 4-5 and Table 4-6. Table 4-5 shows the performance of a complete test set for all crosspoint faults when it is used to test stuck-at faults. Table 4-6 shows the performance of a stuck-at faults test set in detecting crosspoint faults.

TABLE 4-5 CROSSPOINT FAULTS DOMINATE STUCK-AT FAULTS

| Circuit name | Number of inputs | Number of gates | Number of single stuck-at faults | Detected by test set for crosspoint faults | | |
|---|---|---|---|---|---|---|
| | | | | Average | | Maximum number |
| | | | | Number | % | |
| 2of5 | 6 | 18 | 48 | 48 | 100.0 | 48 |
| 2of5_2 | 7 | 12 | 38 | 37.9 | 99.7 | 38 |
| 3bit adder | 4 | 4 | 16 | 16 | 100.0 | 16 |
| 3_17 | 3 | 6 | 18 | 18 | 100.0 | 18 |
| 4_49 | 4 | 16 | 40 | 40 | 100.0 | 40 |
| 4mod5_1 | 5 | 8 | 26 | 23.8 | 91.5 | **25** |
| 4mod5_2 | 5 | 9 | 28 | 27.75 | 99.1 | 28 |
| 4mod5_3 | 5 | 5 | 20 | 18.5 | 92.5 | 20 |
| 5mod5_1 | 6 | 17 | 46 | 45.6 | 99.1 | 46 |
| 5mod5_2 | 6 | 10 | 32 | 32 | 100.0 | 32 |
| cycle10_2 | 12 | 19 | 62 | 61.2 | 98.7 | 62 |
| cycle17_3 | 20 | 48 | 136 | 135.3 | 99.5 | 136 |
| ham3 | 3 | 5 | 16 | 15.6 | 97.5 | 16 |
| ham7 | 7 | 23 | 60 | 58.5 | 97.5 | 60 |
| ham15 | 15 | 132 | 294 | 292.4 | 99.5 | **293** |
| hwb4 | 4 | 17 | 42 | 42 | 100.0 | 42 |
| hwb5 | 5 | 55 | 120 | 120 | 100.0 | 120 |
| hwb6 | 6 | 126 | 264 | 264 | 100.0 | 264 |
| hwb7 | 7 | 289 | 592 | 592 | 100.0 | 592 |
| mod5a | 6 | 21 | 54 | 54 | 100.0 | 54 |
| mod1024a | 20 | 55 | 150 | 149.8 | 99.9 | 150 |
| mod1048576a | 40 | 210 | 500 | 500 | 100.0 | 500 |
| rd53_1 | 7 | 30 | 74 | 74 | 100.0 | 74 |
| rd53_2 | 7 | 12 | 38 | 37.5 | 98.7 | 38 |
| rd53_3 | 8 | 12 | 40 | 39.6 | 99.0 | 40 |
| rd53_4 | 7 | 28 | 70 | 69.7 | 99.6 | 70 |
| rd53_5 | 7 | 16 | 46 | 45.7 | 99.3 | 46 |
| rd73 | 10 | 20 | 60 | 59.8 | 99.7 | 60 |
| rd84 | 15 | 28 | 86 | 85.7 | 99.7 | 86 |
| xor5 | 5 | 4 | 18 | 17.3 | 96.1 | 18 |
| hwb8 | 8 | 637 | 1290 | 1290 | 100.0 | 1290 |
| hwb9 | 9 | 1544 | 3106 | 3106 | 100.0 | 3106 |
| hwb10 | 10 | 3631 | 7282 | 7282 | 100.0 | 7282 |
| hwb11 | 11 | 9314 | 18650 | 18650 | 100.0 | 18650 |

We first take a look at the performance of the test sets for crosspoint faults. It turns out that for almost all benchmark circuits, the combined test set for both types of crosspoint faults covers 100% of the single stuck-at faults in the same circuit (see Table 4-5). Since a test set which covers all single stuck-at faults also covers all multiple stuck-at faults, the combined test set for both types of crosspoint faults covers all stuck-at faults. Columns 5 and 7 are the average and maximum numbers (from 10 different trials) of single stuck-at faults covered by the combined test set for crosspoint faults generated by the randomized ATPG algorithm given in Section 4.2.2. For 30 out of the 34 benchmark circuits, the average number of stuck-at faults covered by the test set for crosspoint faults is no smaller than M-1, where M is the total number of single stuck-at faults in the benchmark circuit. For the remaining 4 benchmark circuits, the number of single stuck-at faults which are not detected by the complete test set for all single crosspoint faults is still less than 3.

Now let us look at the results in percentage form (column 6, Table 4-5). 100% fault coverage is achieved for 16 out of the 34 benchmark circuits. 27 benchmark circuits have more than 99% fault coverage. The average fault coverage of all benchmark circuits is about 99%, which is a good illustration that in most cases, the single crosspoint faults dominate the stuck-at faults.

Results also show that for the majority of benchmark circuits except two (4mod5_1 and ham15), at least in one of the ten trials, the complete test set generated for crosspoint faults covers all of the stuck-at faults in the circuit. For circuits 4mod5_1 and ham15, only one stuck-at fault in each of them is not covered.

TABLE 4-6 STUCK-AT FAULTS DO NOT DOMINATE CROSSPOINT FAULTS

| Circuit name | Test (T) size for stuck faults | Disappearance faults | | | Appearance faults | | |
|---|---|---|---|---|---|---|---|
| | | Total number of faults | Detected by T | | Total # faults | Detected by T | |
| | | | Number | % | | Number | % |
| 2of5 | 3 | 24 | 5 | 20.8 | 66 | 2 | 3.0 |
| 2of5_2 | 5 | 19 | 18 | 94.7 | 53 | 30 | 56.6 |
| 3bit adder | 3 | 6 | 6 | 100. | 6 | 2 | 33.3 |
| 3_17 | 4 | 7 | 6 | 85.7 | 5 | 3 | 60.0 |
| 4_49 | 4 | 24 | 16 | 66.7 | 24 | 16 | 66.7 |
| 4mod5_1 | 4 | 12 | 10 | 83.3 | 20 | 13 | 65.0 |
| 4mod5_2 | 5 | 11 | 11 | 100. | 25 | 15 | 60.0 |
| 4mod5_3 | 3 | 6 | 5 | 83.3 | 14 | 5 | 35.7 |
| 5mod5_1 | 4 | 38 | 20 | 52.6 | 47 | 25 | 53.2 |
| 5mod5_2 | 4 | 18 | 11 | 61.1 | 32 | 20 | 62.5 |
| cycle10_2 | 5 | 100 | 18 | 18.0 | 109 | 44 | 40.4 |
| cycle17_3 | 5 | 409 | 24 | 5.9 | 503 | 104 | 20.7 |
| ham3 | 4 | 6 | 6 | 100. | 4 | 3 | 75.0 |
| ham7 | 6 | 34 | 28 | 82.4 | 104 | 68 | 65.4 |
| ham15 | 7 | 352 | 179 | 50.9 | 1496 | 861 | 57.6 |
| hwb4 | 4 | 27 | 22 | 81.5 | 24 | 13 | 54.2 |
| hwb5 | 6 | 101 | 74 | 73.3 | 119 | 88 | 73.9 |
| hwb6 | 6 | 320 | 189 | 59.1 | 310 | 178 | 57.4 |
| hwb7 | 6 | 864 | 415 | 48.0 | 870 | 380 | 43.7 |
| mod5a | 4 | 27 | 13 | 48.1 | 78 | 10 | 12.8 |
| mod1024a | 5 | 220 | 54 | 24.5 | 825 | 248 | 30.1 |
| mod1048576a | 8 | 1540 | 152 | 9.9 | 6650 | 1689 | 25.4 |
| rd53_1 | 3 | 41 | 17 | 41.5 | 139 | 3 | 2.2 |
| rd53_2 | 5 | 28 | 14 | 50.0 | 44 | 24 | 54.5 |
| rd53_3 | 5 | 20 | 19 | 95.0 | 64 | 38 | 59.4 |
| rd53_4 | 4 | 24 | 13 | 54.2 | 144 | 26 | 18.1 |
| rd53_5 | 5 | 27 | 16 | 59.3 | 69 | 59 | 85.5 |
| rd73 | 5 | 34 | 29 | 85.3 | 146 | 96 | 65.8 |
| rd84 | 5 | 49 | 42 | 85.7 | 343 | 199 | 58.0 |
| xor5 | 4 | 4 | 4 | 100. | 12 | 9 | 75.0 |
| hwb8 | 8 | 2214 | 914 | 41.3 | 2245 | 1024 | 45.6 |
| hwb9 | 11 | 5812 | 3313 | 57.0 | 6540 | 3550 | 54.3 |
| hwb10 | 11 | 15455 | 6961 | 45.0 | 17224 | 7756 | 45.0 |
| hwb11 | 10 | 39211 | 15014 | 38.3 | 53929 | 25118 | 46.6 |

The test set for all stuck-at faults in the circuits (column 2, Table 4-6), on the contrary, does not show good performance in testing crosspoint faults (columns 4, 7, Table 4-6). For most of the benchmark circuits, only a fraction (in many cases less than half) of the single crosspoint faults are covered by the test set for stuck-at faults. In percentage form (columns 5, 8, Table 4-6), the average fault coverage of all benchmark circuits is 61.83%, and 48.98% for disappearance faults and appearance fault respectively. This demonstrates that the stuck-at fault model is unlikely to be the ideal fault model for reversible logic circuits.

## 4.4 Conclusion

In this chapter we introduced a new fault model, the crosspoint fault model, for reversible logic circuits since the traditional stuck-at fault model itself is not adequate in representing different fault effects in reversible logic circuits. The new crosspoint fault model is defined to describe internal defects in generalized Toffoli gates where the crosspoints do not work properly. It includes both situations where some control points are missing (disappearance faults) and where new control points are introduced to the gate (appearance faults). Some properties about how this new fault model affects the behavior of the circuits are investigated.

We proposed a randomized ATPG algorithm to generate a complete test set for crosspoint faults in Toffoli networks. This ATPG algorithm takes advantage of both random algorithms and greedy algorithms in order to produce minimum or near minimum

test set in a comparably short time. Experimental results and comparison with simulation results from other ATPG algorithms have demonstrated the power of our ATPG algorithm.

In this chapter we also studied the relationship between crosspoint faults and stuck-at faults and showed that a complete test set for all single crosspoint faults covers most of the single stuck-at faults in the circuits, while a complete test set for stuck-at faults does not show good performance in testing single crosspoint faults. This illustrates that the new fault model, the crosspoint fault model, is a better fault model than the traditional stuck-at fault model for Toffoli networks.

# Chapter 5 Crosspoint Faults in Simplifying Toffoli Networks

In the previous chapter we introduced a new fault model, namely the crosspoint fault model, for reversible logic circuits. We showed that the crosspoint fault model is a better model than the classical stuck-at fault model in representing the fault effects in Toffoli networks since in most cases the crosspoint faults dominate the stuck-at faults. In this chapter, we investigate some other properties of Toffoli networks under this fault model and study as to whether these properties could contribute to other aspects of reversible logic computing such as reversible logic synthesis. In the remaining part of this chapter, we show how to use this new type of fault model to identify a form of redundancy in reversible circuits, and how to eliminate the redundancy and thus simplify the circuits.

## 5.1 Redundancy in Reversible Circuits

Traditionally, redundancy in circuit design has been closely tied to testability. A combinational circuit that contains an undetectable stuck-at fault is said to be redundant, since such a circuit can always be simplified by removing at least one gate or gate input [1, page 100]. While the presence of redundancy in digital designs may often be intentional, as in fault tolerant systems, or for hazard avoidance, it is obvious that one would normally wish for a synthesis algorithm to avoid the inclusion of unnecessary redundant elements.

For our purposes, a reversible logic circuit is said to be **redundant** if it contains a gate,

or a line (an input-output pair in a reversible logic gate), which can be removed without affecting the logical output of the circuit. So, if one reversible logic circuit is redundant, we can remove at least one gate or gate input-output pair from the circuit in order to reduce cost.

As shown in Figure 5-1, we classify the redundancy in reversible logic circuits into two categories: type-a redundancy and type-b redundancy.

a. Duplicated gates b. Gates with less inputs

Figure 5-1 Redundancy in reversible circuits

Type-a redundancy denotes situations where at least one gate could be removed from the circuit, or replacing a network with one of fewer gates, such as consecutive duplicated gates shown in Figure 5-1(a). Most of the templates introduced by Maslov, et al. ([28]) belong to this category as templates use a simpler network to replace one with more gates.

Type-b redundancy denotes situations where gate inputs and outputs could be removed from the circuit, such as the Toffoli network shown in Figure 5-1(b), where both of the four-input generalized Toffoli gates can be replaced with a three-input Toffoli gate. In this case although the number of gates is not decreased, the new gates have a reduced

number of control inputs and control outputs and thus the cost of the circuit is also reduced.

In our research on reversible logic circuits, it became clear that the implementations of some benchmark functions are redundant. One example is the implementation of benchmark function 2_of_5 (see Figure 5-2), which contains a type-b redundancy. We can delete control points 1 and 7, as shown in Figure 5-2, without changing the function of the circuit. We could also remove control points 3 and 5 to make the circuit not redundant. However, we could not remove both pairs of control points at the same time since they are **dependent**, meaning that deleting one pair of control points would make the other pair irredundant. Only one pair of control points could be removed to make the resulting circuit non-redundant.

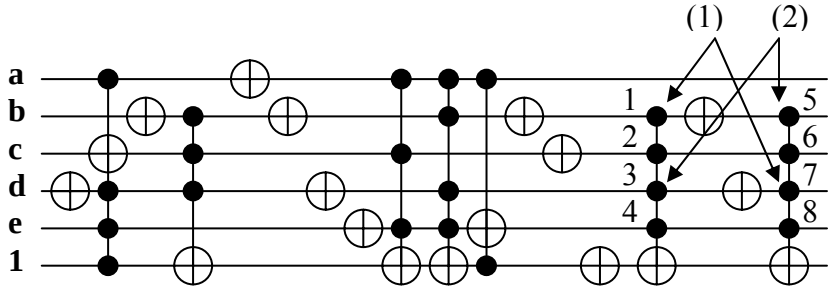



Figure 5-2 Benchmark circuit 2_of_5 is redundant

This raises a question as to how to identify type-b redundancy in reversible logic circuits. In the following section, we show how the crosspoint fault model which is defined in the previous chapter is related to type-b redundancy in reversible logic circuits and how to use the multiple crosspoint faults to identify and reduce type-b redundancy in

Toffoli networks.

## 5.2 Double Crosspoint Faults and Redundancy in Toffoli Networks

As introduced in Section 4.1, a multiple disappearance fault happens when there are at least two single disappearance faults in the circuit. It is obvious that all multiple disappearance faults in a non-redundant reversible circuit are detectable. If there exists an undetectable multiple disappearance fault in a circuit, then the circuit must be redundant. For instance, in the benchmark circuit 2_of_5 (Figure 5-2) shown in Section 5.1, there are two undetectable double disappearance faults: (1) control points 1 and 7 are missing; (2) control points 3 and 5 are missing. Removing either pair of the control points makes the resulting circuit non-redundant. So this type-b redundancy in benchmark circuit 2_of_5 can be interpreted as the existence of undetectable multiple disappearance faults in the circuit.

This connection between type-b redundancy and undetectable multiple disappearance faults in Toffoli networks can be used to simplify the circuits. The basic idea is to first locate type-b redundancy in Toffoli networks by searching for all undetectable multiple disappearance faults and then simplify the circuit by deleting those independent undetectable multiple faults in the circuits.

In Section 4.1 we also mentioned the mixed multiple crosspoint faults. In this thesis we only consider the situation where a single disappearance fault and a single appearance fault happen at the same time, which is labeled as a **mixed double fault**. If there exists an undetectable mixed double crosspoint fault in a reversible circuit, it means that there

exists an alternative circuit which implements exactly the same function as the original one.

The circuits shown in Figure 5-3 give an example of undetectable mixed double crosspoint faults. It is easy to verify that the two circuits are functionally equivalent. The circuit on the right can also be interpreted as the resulting circuit when an undetectable double mixed crosspoint fault happens to the circuit on the left: the control point from line b on the first gate is missing and a control point from line a is introduced to the second gate.

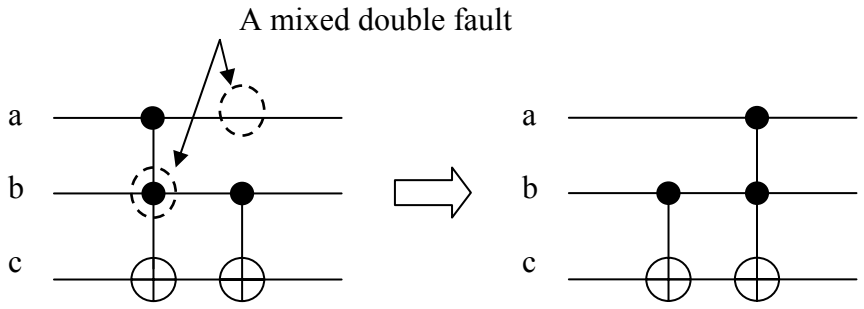



Figure 5-3 An undetectable mixed double crosspoint fault

Functionally equivalent reversible logic circuits might have a different or the same quantum cost as the original reversible circuit, depending on the quantum costs of each gate in the circuits. However, functionally equivalent Toffoli networks derived from introducing one or more undetectable double mixed crosspoint faults to the original circuit have the same quantum costs. We give a proof of this property in Section 5.5 below. Here we just examine the two circuits given in Figure 5-3: they have the same

quantum costs as they contain the same gates. The only difference between them is that the two gates are placed in a different order.

The mixed multiple crosspoint faults are also potentially useful in simplifying Toffili networks. During the process of synthesis or simplification, the original circuit might not easily be further simplified with current methods, but one of its functionally equivalent alternatives might be. Under this circumstance the mixed multiple crosspoint faults can be used to find such equivalent circuits.

However, in this chapter our main interest is in the simplification of Toffoli networks by locating undetectable multiple disappearance faults, especially undetectable double disappearance faults, and then removing type-b redundancy in the circuits. In following sections, we give properties of double disappearance faults and then use these properties to simplify Toffoli networks.

## 5.3 Properties of Double Disappearance Faults

In this section we present several important properties regarding the testability of double disappearance faults in Toffoli networks. These features will be integrated in the simplification algorithm to reduce the costs of the circuits. The detailed algorithm is given in Section 5.4.

**Property 5**: **If a double disappearance fault happens at the same gate, it is detectable.**

This property is obvious, based on the structure of a generalized Toffoli gate.

**Property 6: If two generalized Toffoli gates result in an undetectable double**

**disappearance fault, then the two gates are of the same size, i.e., they have the same number of inputs.**

Assume a double disappearance fault happens at gates G1 and G2 in a Toffoli network, where G1 has control inputs $(x_1, x_2, ..., x_m)$ and data input $x_{m+1}$, G2 has control inputs $(y'_1, y'_2, ..., y'_s)$ and data input $y'_{s+1}$. Without loss of generality, we assume the double fault is the missing control point $x_m$ on G1 and another missing control point $y'_s$ on G2. Figure 5-4 shows the sub-circuit from G1 to G2: G1 is at stage A, G2 is at stage C, and B is the block of reversible gates between G1 and G2. We need to prove that if this double fault is undetectable, then $m = s$.

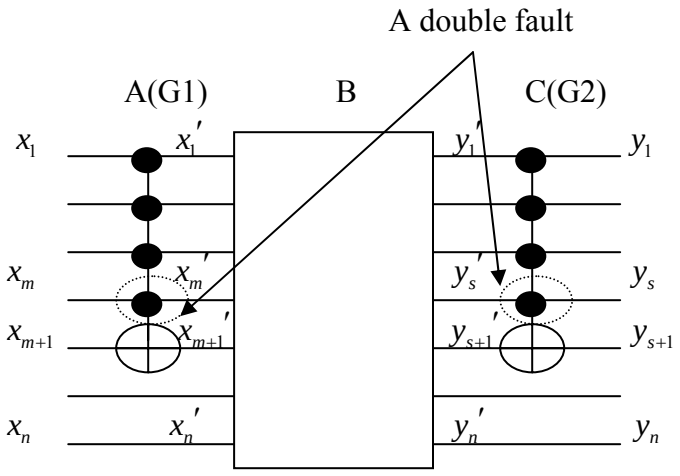



Figure 5-4 A sub-circuit which contains a double disappearance fault

*Proof*:

As shown in Figure 5-4, we divide the sub-circuit into three parts: (i) Part A is stage A which contains the faulty gate G1; (ii) Part B is the fault-free sub-circuit between G1 and G2. This sub-circuit may contain an arbitrarily large number of gates; (iii) Part C is stage

C which contains the faulty gate G2. Clearly all three parts, individually or the concatenation of any two or three of them, in either faulty or fault-free state, are reversible functions with $n$ inputs and $n$ outputs, each implementing a permutation of $2^n$ distinct $n$-bit vectors. In the following, we approach a proof of this property by analyzing the reversible functions implemented by each of these three parts in both faulty and fault-free states. We first show that in part A, the effect of the faulty gate G1 swaps two blocks of output vectors in the truth table. Then, in part C, the faulty gate G2 has to cancel this effect to make the double disappearance fault undetectable.

(1) Part A: Table 5-1 shows the truth table of the function implemented by part A for both the fault-free and the faulty circuits. Without loss of generality, we label $(x_1, x_2, ..., x_m, x_{m+1})$ as the most significant bits. Vectors $v_i$ are the appropriate $n$-bit output vectors which are not relevant to the proof as they are the same for the fault-free circuit and the faulty one.

Clearly, the effect of the missing control point $x_m$ from G1 on the fault-free reversible logic function is that it changes the output vectors when the first $m$ bits in the input vectors are '11…11 0'. In the truth table, it swaps two blocks of vectors (block I and block II, as shown in Table 5-1) in the output set, each block containing $2^{n-m-1}$ vectors. For the remaining $2^n - 2^{n-m}$ input vectors whose first $m$ bits do not have the pattern '11…11 0', the output vectors are exactly the same for the fault-free circuit and the faulty one.

TABLE 5-1 TRUTH TABLE OF PART A

| Input vector $x_1 \ldots x_{m-1}$ | $x_m$ | $x_{m+1}$ | $x_{m+2} \ldots x_n$ | Output vector (fault-free circuit) $x_1' \ldots x_{m-1}'$ | $x_m'$ | $x_{m+1}'$ | $x_{m+2}' \ldots x_n'$ | Output vector (faulty circuit) $x_1' \ldots x_{m-1}'$ | $x_m'$ | $x_{m+1}'$ | $x_{m+2}' \ldots x_n'$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0……0 | 0 | 0 | 0……0 | $v_0$ | | | | $v_0$ | | | | |
| 0……0 | 0 | 0 | 0……1 | $v_1$ | | | | $v_1$ | | | | |
| …… | | | | …… | | | | …… | | | | |
| **1……1** | **0** | 0 | 0……0 | **1……1** | **0** | **0** | **0……0** | **1……1** | **0** | **1** | **0……0** | |
| **……** | | | | **……** | | | **I** | **……** | | | | **II** |
| **1……1** | **0** | 0 | 1……1 | **1……1** | **0** | **0** | **1……1** | **1……1** | **0** | **1** | **1……1** | |
| **1……1** | **0** | 1 | 0……0 | **1……1** | **0** | **1** | **0……0** | **1……1** | **0** | **0** | **0……0** | |
| **……** | | | | **……** | | | **II** | **……** | | | | **I** |
| **1……1** | **0** | 1 | 1……1 | **1……1** | **0** | **1** | **1……1** | **1……1** | **0** | **0** | **1……1** | |
| 1……1 | 1 | 0 | 0……0 | $v_j$ | | | | $v_j$ | | | | |
| …… | | | | …… | | | | …… | | | | |
| 1……1 | 1 | 1 | 1……1 | $v_{(2^n-1)}$ | | | | $v_{(2^n-1)}$ | | | | |

(2) Part B: Part B is also a permutation of $2^n$ distinct $n$-bit vectors. This sub-circuit is fault-free.

Now we consider part (A+B). In this section '+' denotes the operation of concatenation. In step (1) we get that the effect of the missing control point from G1 on part A is swapping two blocks of $2^{n-m-1}$ vectors in the output set of function A. Since there is no fault in part B, and part B only performs a permutation of the $2^n$ distinct vectors, we know that the effect of the missing control point from G1 on part (A+B) is also swapping two blocks of $2^{n-m-1}$ vectors in the output set of function (A+B).

(3) Part C: Similar to the analysis of part A, the missing point at $y'_s$ from G2 simply swaps two blocks of vectors in the output set of reversible function C, each block containing $2^{n-s-1}$ vectors.

(4) For the whole sub-circuit (A + B + C):

From the analysis in (1), (2), and (3) above, we know that the first missing point at $x_m$

swaps two blocks of vectors in the output set of function (A+B), and each block having $2^{n-m-1}$ vectors. The second missing point at $y'_s$ also swaps two blocks of vectors in the output set of function C, each having $2^{n-s-1}$ vectors.

In order to make the double disappearance fault undetectable, which means that the function of the faulty circuit is exactly the same as that of the fault-free one, the swap happening in part C must precisely cancel the effect that of the swap happened in part A+B. This requires that the blocks swapped in part A and part B must have the same length, i.e., $m = s$.

*Hence Property 6 is proved*.

If $m \neq s$, i.e., the blocks swapped by part A and part C have different lengths, then the swap at part C cannot cancel the effect of the swap at part A since there are some output vectors which are switched only by one of the swaps. In this case, the faulty circuit must implement a different function from that of the fault-free one, thus the double fault is detectable.

Property 6 is very important to the algorithm for simplifying Toffoli networks by locating undetectable double disappearance faults since it reduces the search space significantly. Only double faults that occur on two gates of the same size need to be considered and examined to see whether they are detectable.

It should be noted that an undetectable double disappearance fault does not necessarily have to happen on two equivalent Toffoli gates. Here two **equivalent Toffoli gates** in a circuit are defined as follows: both gates have the same size, and the control inputs and data input of one gate are on the same lines as those of the other gate. This is obvious, since in reversible circuits, we could always use a SWAP gate to switch two lines. The

structure of a SWAP gate is introduced in Chapter 3. It can be implemented with three CNOT gates, as shown in Figure 5-5.

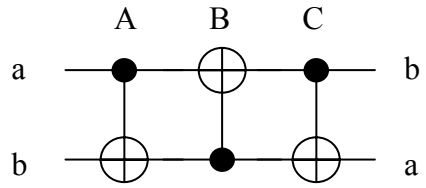



Figure 5-5 Structure of a SWAP gate

The three CNOT gates in the SWAP gate (Figure 5-5) are of the same size. However, only gate A and gate C are considered as equivalent gates since the control input and the data input of gate B are on different lines from those of gate A and C.

Property 7 and property 8 are derived from property 6.

**Property 7**: **Given three single disappearance faults $f_1$, $f_2$, and $f_3$ in a Toffoli network, if both double disappearance faults $\{f_1, f_2\}$ and $\{f_1, f_3\}$ are undetectable, then the single faults $f_2$ and $f_3$ cannot happen on the same gate.**

We prove this property by contradiction.

*Proof:*

Assume single fault $f_1$ happens on gate G1 at stage A, both single faults $f_2$ and $f_3$ happen on a same gate G2, which is at stage C (Figure 5-6). Assume both G1 and G2 have $m$ control inputs (according to property 6, G1 and G2 must have the same number of inputs); $f_2$ happens on the $j$-th input of G2, $f_3$ happens on the $k$-th input of G2, $j \neq k$.

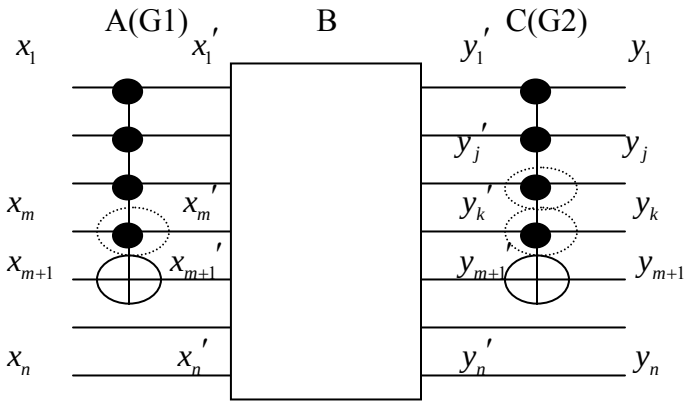



Figure 5-6 A sub-circuit which contains a double disappearance fault

According to part (1) of the proof for property 6, both single faults $f_2$ and $f_3$ individually swap two blocks of output vectors in the truth table of the function of stage C. The two blocks swapped by $f_2$ have the same size as those swapped by $f_3$. But since $j \neq k$, there is no overlap between the blocks moved by fault $f_2$ and those moved by $f_3$. This can be easily examined by listing the truth table of function C with each fault applied.

However, since both double faults $\{f_1, f_2\}$ and $\{f_1, f_3\}$ are undetectable, according to the proof of property 6, fault $f_2$ should switch back the blocks swapped by $f_1$, and fault $f_3$ should also switch back the same blocks swapped by $f_1$. That means fault $f_2$ and $f_3$ must work on the same blocks. This is a contradiction since we showed above that $f_2$ and $f_3$ work on different blocks. So $f_2$ and $f_3$ cannot happen on the same gate, or if they do, $j$ must be equal to $k$, meaning that $f_2$ and $f_3$ represent the same fault.

*Consequently property 7 is proved.*

This property is also very helpful in speeding up the simplification process of Toffoli networks. If a double fault $\{f_1, f_2\}$ is found undetectable, assuming $f_1$ happens on G1, $f_2$ happens on G2, then any other double faults $\{f_1, f_j\}$ and $\{f_i, f_2\}$, where $f_i$ happens on G1, $f_j$ happens on G2, are detectable.

However, two undetectable double disappearance faults can happen on the same gates, i.e., if both double disappearance faults $\{f_1, f_2\}$ and $\{f_3, f_4\}$ are undetectable, it could be true that $f_1$ and $f_3$ happen on the same gate, $f_2$ and $f_4$ both happen on another distinct gate. This can be illustrated by the circuit shown in Figure 5-7. Let $f_1, f_2, f_3, f_4$ denote the four single disappearance faults happening to control points 2, 3, 5, 6, respectively. Then both double disappearance faults $\{f_1, f_4\}$ and $\{f_2, f_3\}$ are undetectable and they happen on the same gates.

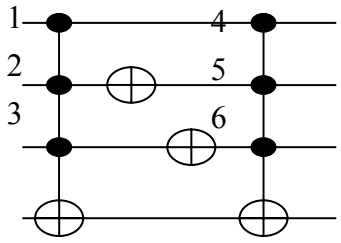



Figure 5-7 Two undetectable double disappearance faults happen on the same gates

**Property 8**: **Given a double disappearance fault $\{f_1, f_2\}$, where $f_1$ and $f_2$ occur on two gates of the same size, let $T$ be the set of all vectors which detect the single fault $f_1$. If none of the vectors in $T$ detects the double fault, then the double fault**

**$\{f_1, f_2\}$ is undetectable.**

*Proof:*

According to the proof of property 6, each single fault swaps exactly two equal-sized blocks of output vectors in the truth table of the given function. If none of the vectors in $T$ detects the double faults, then both single faults have swapped the same blocks of vectors. So for the remaining vectors which are not included in $T$ and do not detect $f_1$, they do not detect the second fault $f_2$ either because $f_2$ does not change their corresponding outputs.

*Property 8 is proved.*

We use this property to quickly find out whether a double fault is undetectable. Instead of going through all possible $2^n$ vectors, we only need to check the vectors which are able to exercise the first single fault. So for any double disappearance fault, in order to test whether it is detectable or not, in the worst case, we only need to check $2^{n-(m-1)} = 2^{n-m+1}$ vectors, where the circuit has $n$ inputs and the gates where the double fault happens have $m$ inputs.

## 5.4 Synthesis Algorithm for Simplifying Toffoli Networks

In this section, our primary focus is on the algorithm for simplifying Toffoli networks using properties on the testability of double disappearance faults as introduced in Section 5.3.

The detailed algorithm is given below.

*1. Read in circuit information. Get a fault list* $F_{list}=\{f_1, f_2, \ldots\}$ *of all single disappearance faults.*

*2. Take two single faults* $f_i$ *and* $f_j$ *from* $F_{list}$ *and form a double fault* $\{f_i, f_j\}$. *Assume* $f_i$ *happens at gate* $g_i$, $f_j$ *happens at gate* $g_j$.

*If Size(*$g_i$*)!=Size(*$g_j$*), mark the double fault* $\{f_i, f_j\}$ *as detectable.*

*Else {Compute the distance of the double fault* $\{f_i, f_j\}$ *as* $d=|g_i-g_j|$. *If* $d=0$, *mark the double fault* $\{f_i, f_j\}$ *as detectable.*

*Else {Simulate the sub-circuit from* $g_i$ *to* $g_j$ *using vectors which detect the single fault* $f_i$. *Break if the double fault* $\{f_i, f_j\}$ *is detectable.*

*Else mark the fault as undetectable. Delete both faults* $f_i$ *and* $f_j$ *from* $F_{list}$, *update circuit information and fault list* $F_{list}$. *}}*

*3. Repeat step 2 until all double faults in* $F_{list}$ *are tested.*

The algorithm checks every double disappearance fault to see whether it is undetectable; if yes, then delete the double fault. Using this technique we can simplify the circuits.

As we can see, while the algorithm is basically exhaustive search, the integration of the properties given in Section 5.3 sharply reduces the size of the search space. Before a large number of input vectors are applied to the circuit, properties 6 and 7 are used to determine many of the double disappearance faults which are detectable. Property 8 is

finally used to check whether a double disappearance fault is detectable by applying vectors to exercise the circuit if all previous steps fail to decide. However, even in this step, the process does not go through all $2^n$ input vectors, where $n$ is the number of inputs of the circuit. Instead, only the vectors which could exercise one of the two single faults are examined.

Another method used in the simplification algorithm to speed up the simulation is to simulate only the sub-circuit from $g_i$ to $g_j$ instead of simulating the whole circuit. This technique is based on the property that concatenating a fault-free reversible circuit B of the same number of inputs before (Figure 5-8(b)) or after (Figure 5-8(c)) another reversible circuit A (Figure 5-8(a)) which contains faults does not change the testability of those faults. This is true since there is a one-to-one matching between the input vectors and the output vectors of circuit B.

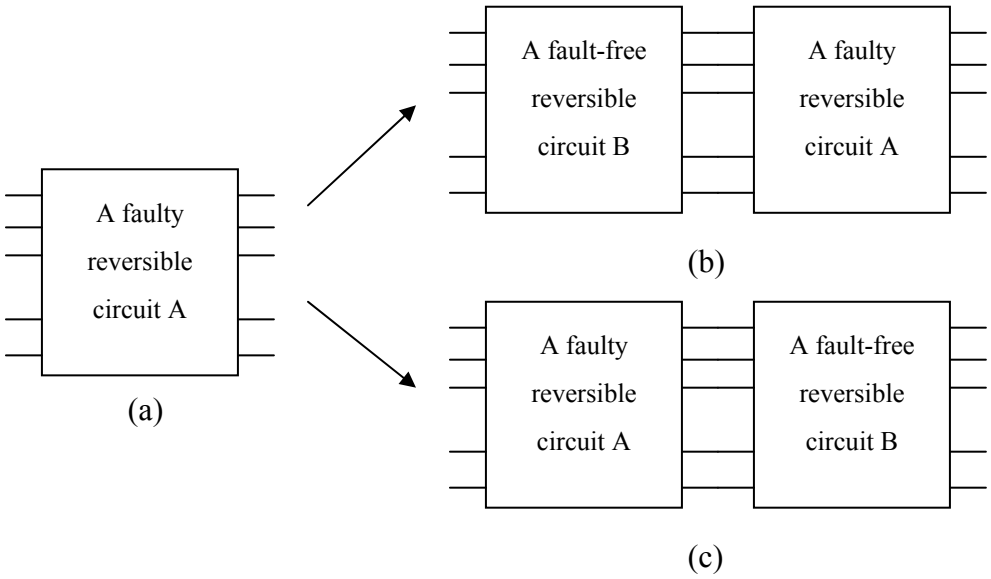



Figure 5-8 Concatenation of a fault-free reversible circuit and a faulty circuit

For a couple of very large circuits, we only examine double disappearance faults with distance less than 100 in the simulation to save time. The distance of a double crosspoint fault is defined in the simplification algorithm as the number of gates between the two single faults plus one.

The simplification algorithm could be easily adjusted to simulate Toffoli networks to find undetectable triple faults or mixed faults. The detailed algorithms are not included here. However, some of the properties may not hold or need to be modified for the triple disappearance faults or mixed faults. In next section, we give modified versions of properties 6, 7, and 8 for double mixed crosspoint faults.

## 5.5 Properties of Double Mixed Crosspoint Faults

Properties 6, 7, 8 for double disappearance faults given in Section 5.3 can be modified as follows (properties 9, 10, and 11) in order to check the testability of double mixed faults.

**Property 9: If two generalized Toffoli gates result in an undetectable double mixed fault, assume that the disappearance fault happens on a gate with $p$ inputs, the appearance fault happens on a gate with $q$ inputs, then $p = q + 1$.**

Proof of property 9 is very similar to that of property 6. We only need to follow the same idea and examine the two blocks of vectors in the output set of the truth table swapped by each single fault. The second swap action must cancel the effect caused by the first swap action. The detailed proof is omitted here.

**Property 10**: **In a Toffoli network, if two double mixed crosspoint faults** $\{f_1, f_2\}$

**and $\{f_1, f_3\}$ are undetectable, then the single faults $f_2$ and $f_3$ cannot occur on the same gate.**

This property holds for either of the following two situations.

- $f_1$ is a single disappearance fault; $f_2$ and $f_3$ are single appearance faults.
- $f_1$ is a single appearance fault; $f_2$ and $f_3$ are single disappearance faults.

**Property 11**: **Given a double mixed fault $\{f_1, f_2\}$, where $f_1$ and $f_2$ occur on two gates of the same size, let $T$ be the set of all vectors which detect the single fault $f_1$. If none of the vectors in $T$ detects the double fault, then the double mixed fault $\{f_1, f_2\}$ is undetectable.**

Similar to properties 7 and 8, the proofs of properties 10 and 11 are very easy based on property 9 and its proof.

Property 12 is derived from property 9.

**Property 12: Given a Toffoli network $N_1$, if network $N_2$ is one of its functional equivalent circuits derived by introducing to or removing from $N_1$ one or more undetectable double mixed crosspoint faults, then circuits $N_1$ and $N_2$ have the same quantum costs.**

We mentioned this property in Section 5.2. It is very easy to prove.

According to property 9, an undetectable double mixed crosspoint fault only happens on two gates whose sizes satisfy the following equation: $p = q + 1$, where $p$ is the number if inputs of the gate where the disappearance fault happens, and $q$ is the number of inputs of the gate where the appearance fault happens. So removing this double mixed fault results in changing the $p$-input Toffoli gate into a $q$-input Toffoli gate and

changing the $q$-input Toffoli gate into a $p$-input Toffoli gate. This operation just switches the positions of the two faulty gates and hence does not affect the quantum costs of the whole circuit.

So the process of locating undetectable double mixed crosspoints in Toffoli networks is not used to identify redundancy and thus reduce quantum costs of the circuits since the functionally equivalent circuits derived in this manner have the same quantum costs as the original circuit. Instead, it aims to find functionally equivalent alternatives for Toffoli networks. Properties 9, 10, and 11 are used to speed up this process. The derived alternative circuits can be considered for further simplification by other methods if possible.

## 5.6 Experimental Results

While the discussion in the previous sections indicates that, in theory, such multiple crosspoint faults could be useful in a synthesis algorithm, it is only through experimental evidence that we can gauge as to how often this is the case. We designed some experiments to investigate whether multiple crosspoint faults are useful in reversible logic synthesis, and whether the simplification algorithm is efficient in processing large circuits:

- Simulate benchmark circuits to find out how many undetectable multiple crosspoint faults exist in the circuits, and, how many of them can be removed without affecting the circuits' function.

- For each undetectable double disappearance fault in benchmark circuits, measure the distance of two faulty gates.
- Measure the run time of our simplification algorithm to evaluate its performance.

In the first part of the experiments, we simulate benchmark circuits posted on [27]. Both multiple disappearance faults and double mixed faults are tested. Two large circuits hwb10 and hwb11 are not simulated for triple disappearance faults and double mixed faults due to the large search space. For double disappearance faults, only those faults that occur with distance less than 100 are simulated.

The experimental results are shown in Table 5-2.

First we discuss the simulation results for multiple disappearance faults. The numbers in columns 2, 3, 4, and 5 of Table 5-2 correspond to the numbers of single disappearance faults in the benchmark circuit; the numbers of undetectable double disappearance faults in the circuit; the numbers of double disappearance faults which we can delete from the circuit to reduce redundancy and costs; and the number of undetectable triple disappearance faults.

TABLE 5-2 SIMULATION RESULTS FOR MULTIPLE CROSSPOINT FAULTS

| Circuit name | Disappearance faults | | | | Mixed double faults |
|---|---|---|---|---|---|
| | Number of single faults | Double faults | | Triple faults | Undetectable |
| | | Undetectable | Removable | Undetectable | |
| 2of5 | 24 | 2 | 1 | 0 | 0 |
| 2of5_2 | 19 | 0 | 0 | 0 | 0 |
| 3bit adder | 6 | 0 | 0 | 0 | 0 |
| 3_17 | 7 | 0 | 0 | 0 | 2 |
| 4_49 | 24 | 0 | 0 | 0 | 7 |
| 4mod5_1 | 12 | 0 | 0 | 0 | 8 |
| 4mod5_2 | 11 | 0 | 0 | 0 | 0 |
| 4mod5_3 | 6 | 0 | 0 | 0 | 3 |
| 5mod5_1 | 38 | 0 | 0 | 0 | 23 |
| 5mod5_2 | 18 | 0 | 0 | 0 | 3 |
| cycle10_2 | 100 | 0 | 0 | 0 | 0 |
| cycle17_3 | 409 | 0 | 0 | 0 | 0 |
| ham3 | 6 | 0 | 0 | 0 | 0 |
| ham7 | 34 | 0 | 0 | 0 | 1 |
| ham15 | 352 | 14 | 7 | 0 | 30 |
| hwb4 | 27 | 0 | 0 | 0 | 3 |
| hwb5 | 101 | 0 | 0 | 0 | 2 |
| hwb6 | 320 | 17 | 7 | 0 | 11 |
| hwb7 | 864 | 22 | 13 | 0 | 21 |
| mod5a | 27 | 0 | 0 | 0 | 1 |
| mod1024a | 220 | 0 | 0 | 0 | 0 |
| rd53_1 | 41 | 0 | 0 | 0 | 0 |
| rd53_2 | 28 | 0 | 0 | 0 | 0 |
| rd53_3 | 20 | 0 | 0 | 0 | 0 |
| rd53_4 | 24 | 0 | 0 | 0 | 3 |
| rd53_5 | 27 | 0 | 0 | 0 | 4 |
| rd73 | 34 | 0 | 0 | 0 | 0 |
| rd84 | 49 | 0 | 0 | 0 | 0 |
| xor5 | 4 | 0 | 0 | 0 | 0 |
| hwb8 | 2214 | 42 | 20 | 0 | 30 |
| hwb9 | 5812 | 298 | 62 | 0 | 64 |
| hwb10 | 15455 | 371* | 119 | - | - |
| hwb11 | 39211 | 613* | 247 | - | - |

* Only those with distance less than 100 were tested

Most small benchmark circuits are likely to be non-redundant as we did not find any undetectable double disappearance faults in them. But most of the large benchmark circuits are redundant since undetectable double disappearance faults exist. For instance, the circuit hwb6 contains 17 undetectable double disappearance faults and 7 of them are removable. Most of the other large circuits are also redundant and can be simplified. Bigger benchmark circuits contain more undetectable double disappearance faults and thus have more room for simplification.

The results (column 5 of Table 5-2) show that none of the benchmark reversible circuits contains any undetectable triple disappearance fault.

Column 6 of Table 5-2 shows the number of undetectable mixed double faults in each benchmark circuit. This means that some benchmark circuits have alternative implementations. During the process of simplification, if the benchmark circuit itself is not easily simplified, some of its alternatives might be. This gives us more opportunities to get better implementation for a function. Results show that some of the benchmark circuits have functionally equivalent circuits.

Of course, in some cases one could envisage that the multiple disappearance faults might be included in the design of templates [28], for instance, the two examples given in Figure 5-1. However, this is only really practical if the faults are comparatively close together. We use the idea of distance, which is defined in the synthesis algorithm, to explore this further.

Table 5-3 shows the number of undetectable double faults which fall into different categories according to their distance for benchmark circuits hwb7, hwb8 and hwb9. For instance, in circuit hwb7, there are 22 undetectable double disappearance faults in total.

Only 2 double faults have distance less than 10. Four of them have distance more than 50. Similarly, for benchmark circuits hwb8 and hwb9, only a small portion of the undetectable double faults have distance less than 10, while many have distance more than 50. In these cases the redundancy would be extremely difficult to identify with methods such as templates.

TABLE 5-3 DISTANCE IN UNDETECTABLE MULTIPLE CROSSPOINT FAULTS

| Distance of double faults | Number of undetectable faults | | |
|---|---|---|---|
| | Hwb7 | Hwb8 | Hwb9 |
| <10 | 2 | 5 | 38 |
| [10, 20) | 5 | 11 | 23 |
| [20, 30) | 4 | 5 | 20 |
| [30, 40) | 6 | 5 | 14 |
| [40, 50) | 1 | 4 | 23 |
| [50, 100) | 4 | 9 | 65 |
| >=100 | 0 | 3 | 115 |
| Total | 22 | 42 | 298 |

In order to illustrate the efficiency of our simplification algorithm (here we refer it as the new algorithm), we also compare its runtime with that of a similar algorithm (referred as the old algorithm) which is also developed by us. The old algorithm has the same procedure as the new algorithm except that the properties presented in Section 5.3 are not integrated in it.

Table 5-4 gives simulation results of some benchmark circuits. The runtime of simplifying each benchmark circuit is recorded for both the old algorithm and the new

one. The simulation was conducted on a Pentium 4 processor with 3.00 GHz frequency and 2.00 GB of RAM memory running Windows XP.

TABLE 5-4 RUNTIME COMPARISON

| Circuit Name | Number of single faults | Runtime (second) | |
|---|---|---|---|
| | | Old alg. | New alg. |
| 2of5 | 24 | 0.000 | 0.000 |
| Hwb6 | 320 | 9.562 | 0.796 |
| Hwb7 | 864 | 266.796 | 14.187 |
| Hwb8 | 2214 | 8554.156 | 214.609 |
| Ham15 | 352 | 10920.171 | 38.843 |

We can see that the new algorithm is very efficient. We evaluated the first 30 benchmark circuits listed in Table 5-2. For most circuits tested, it takes less than a minute. For one of the large circuits, hwb8, which contains 2214 single faults and thus $\binom{2214}{2}$ = 2,449,791 double disappearance faults, the runtime is still less than 4 minutes. Apparently this algorithm can be used for much larger circuits which contain many more faults and/or inputs.

Results show that the new algorithm runs much faster than the old one. For all large circuits, there is a significant improvement: for circuit hwb8, the new program is about 40 times faster than the old one. For circuit ham15, the new algorithm is about 300 times faster. This is because ham15 has 15 inputs thus in the worst case the old algorithm has to simulate a double disappearance fault with all possible $2^{15}$ test vectors, while in the new algorithm, the use of properties could quickly find many double disappearance faults

which are detectable.

## 5.7 Conclusion

In this chapter we studied more properties related to crosspoint faults in Toffoli networks. We introduced two types of redundancy in Toffoli networks and showed that one of the redundancy forms is connected with the undetectable multiple disappearance crosspoint faults. Consequently, we applied this feature in reversible synthesis and proposed an approach to reduce type-b redundancy in Toffoli networks by deleting independent multiple disappearance faults from the circuits.

We investigated several important properties to quickly identify the testability of a double disappearance fault. These properties were integrated into the synthesis algorithm to simplify Toffoli networks. The simplification algorithm was given in detail.

We also gave several properties related to the testability of double mixed crosspoint faults. These properties can be used to speed up the process of finding functionally equivalent circuits for Toffoli networks. However, the functionally equivalent circuits derived in this manner have exactly the same quantum costs as the original circuit so this process of searching equivalent circuits does not itself contribute to removing redundancy or reducing costs of the circuits, but may, of course, lead to a simpler circuit.

Experimental results were provided to show that some benchmark reversible circuits, especially the large ones, could be further simplified since they contain some undetectable double disappearance faults. Deleting the independent undetectable double

disappearance faults reduces the quantum costs of these circuits.

Simulation results also illustrated that after integrating the properties related to the testability of double crosspoint faults, our simplification algorithm was far more efficient than other algorithms and could be used to handle very large circuits.

# Chapter 6 An Improved Implementation of a Reversible Synthesis Algorithm

As mentioned in Chapter 3, spectral technologies are often used in reversible logic synthesis. In this chapter, we give an improved implementation of a Reed-Muller (RM) spectra-based reversible logic synthesis algorithm proposed by Maslov et al. [29] and show that the new implementation runs much faster than the original one.

## 6.1 Reed-Muller Spectra of Reversible Logic functions

In chapter 2 we mentioned that the RM spectrum of a Boolean function is defined as follows:

$R = M^n f$ .

Where $M^n = \begin{bmatrix} M^{n-1} & 0 \\ M^{n-1} & M^{n-1} \end{bmatrix}$ and $M^0 = [1]$.

Since a reversible logic function has the same number of outputs as the number of inputs, its output $f$ is a $2^n \times n$ matrix, where $n$ is the number of inputs or outputs. Consequently, the RM spectrum $R$ of a reversible logic function is also a $2^n \times n$ matrix. For instance, the RM spectrum of the function implemented by a Toffoli gate can be calculated as follows.

$(y_1, y_2, y_3) = (f_1(x_1, x_2, x_3), f_2(x_1, x_2, x_3), f_3(x_1, x_2, x_3)) = (x_1, x_2, x_1x_2 \oplus x_3)$,

$$\text{So } F^T = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix},$$

$$R = M^n F = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Columns 1, 2, and 3 in the RM spectral table correspond to the RM spectra of the individual output function $y_3, y_2, y_1$ respectively.

For the $n$-variable identity function $(y_n, y_{n-1}, \cdots, y_1) = (x_n, x_{n-1}, \cdots, x_1)$, its RM spectral table contains only one non-zero coefficient at row $2^{i-1}$ for each output $y_i$. The remaining RM spectral coefficients are all 0. For instance, the RM spectra of the 3-variable identity function are:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Application of a generalized Toffoli gate to a given function in the RM spectral domain can be done easily from either input or output side or both sides of the specification of the function [29].

## 6.2 A Reversible Synthesis Algorithm Based on Reed-Muller Spectra

In this section we briefly introduce a Reed-Muller spectra-based reversible synthesis algorithm for Toffoli networks.

This Reed-Muller spectra-based reversible synthesis algorithm was proposed by Maslov, Miller, and Dueck in [29]. Since the proof of this algorithm is very complex, we do not explain it in detail. For further information, please refer to the original paper [29] by Maslov et al. Below we only give the basic steps of this synthesis algorithm and provide an example to illustrate it.

In the synthesis algorithm, the RM spectra of the given reversible logic function are first computed. We use $(r_n, r_{n-1}, \cdots, r_1)$ to denote the values in each row of the RM spectral table. For every non-zero RM spectral row, the following steps are applied in order to transform the original RM spectra into that for the identity function by applying Toffoli gates to the original function.

A: Step $i$=0. If this row contains a non-zero element, apply NOT gates $TOF(x_j)$ for every $r_j = 1, j = 1..n$.

B: Step $i = 2^{k-1}, k = 1..n$. If $r_k \neq 1$, assign a gate $TOF(x_s; x_k)$ such that $s = \max\{j \,|\, r_j = 1, j = 1..n\}$ and $s > k$. Apply gates $TOF(x_k; x_j)$ for every $r_j = 1, j = 1..n$.

C: Step $i \neq 2^{k-1}, i > 0$. First find $s = \max\{j \,|\, r_j = 1, j = 1..n\}$ such that item $2^{s-1}$ does not appear in the binary expansion of $i$. Assign gates $TOF(x_s; x_j)$ for every $r_j = 1, j = 1..n$. Apply gate $TOF(X_i; x_s)$, where $X_i$ is a product of variables $x_j$ such that the $j$-th bit the binary expansion of $i$ is one. Undo $TOF(x_s; x_j)$ if such gates changed earlier rows in

the RM spectral table than row $i$.

As the process reaches the RM spectra of the identity function, we link the gates backwards to obtain the circuit.

This reversible logic synthesis algorithm is proved to be able to converge since in B and C-type steps it is always possible to find a value of $s$ which satisfies the requirements ([29]).

Table 6-1 gives an example of synthesizing a reversible function with this algorithm.

TABLE 6-1 SYNTHESIS OF AN EXAMPLE FUNCTION

| Spectral coefficient | Spectrum cba | Step 1 cba | Step 2 cba | Idendity cba |
|---|---|---|---|---|
| 1 | 000 | 000 | 000 | 000 |
| a | 010 | 001 | 001 | 001 |
| b | 001 | 011 | 010 | 010 |
| ab | 111 | 110 | 111 | 000 |
| C | 111 | 110 | 111 | 100 |
| ac | 000 | 000 | 000 | 000 |
| bc | 000 | 000 | 000 | 000 |
| abc | 000 | 000 | 000 | 000 |
| Gate applied | TOF(b;a)<br>TOF(a;b) | TOF(b;a) | TOF(c;b)<br>TOF(c;a)<br>TOF(a,b:c) | |

The second column in Table 6-1 is the RM spectra of the specified function to be synthesized. Now we need to transform the RM spectra into those of the identity function (column 5, Table 6-1) by applying Toffoli gates to it according to the rules given in the synthesis algorithm.

The first row of the RM spectral table is 000, which is the same as that of the identity function, so no action is taken for this row, according to part A of the synthesis algorithm.

The second row is 010, and we need to transform it to 001. According to part B of the algorithm, first TOF(b; a) is assigned to change the least significant bit from 0 to 1; then TOF(a; b) is applied to transform the middle bit from 1 to 0. So these two gates transform the second row from 010 to 001. This step might change some later rows of the RM spectral table, in this case, the 3rd, 4th, and 5th row in the table.

The following two steps are similar, except that in different cases different rules (part B or part C) are applied.

After three steps the RM spectral table reaches those of the identity function. We link the Toffoli gates from the primary outputs to get the desired circuit.

The derived Toffoli network is shown in Figure 6-1.

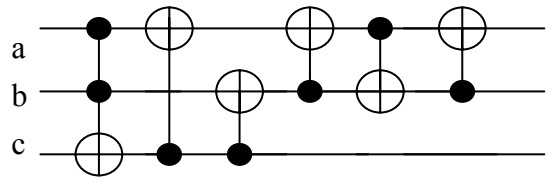



Figure 6-1 Derived Toffoli network for the given function

The original implementation of this synthesis algorithm works with the entire RM spectral table of the specified function during the whole process. It is known that the RM spectral table of an $n$-variables reversible function contains $2^n$ rows. Obviously, the number of rows of the RM spectral table grows exponentially as $n$ increases.

Consequently, the process has large memory requirements during execution.

In the next section, we introduce an improved implementation of this reversible synthesis algorithm by using a compact form of the RM spectral table.

## 6.3 Improved Implementation of the Algorithm

The optimized approach to implementing the synthesis algorithm is based on an important observation we found while investigating the RM spectra of some reversible benchmark functions.

The RM spectral table of some reversible benchmark functions listed on [27] are computed and examined. We find a very interesting and important phenomenon: the RM spectral tables of most of the benchmark functions contain a very large number of zero-rows, meaning that all elements in those rows are of value zero. Only a small number of rows contain non-zero elements. The numbers of non-zero RM spectral rows of each benchmark function are summarized in Table 6-2.

TABLE 6-2 RM SPECTRA OF BENCHMARK FUNCTIONS

| Function | Number of inputs | Number of RM spectra rows | Number of non-zero RM spectra rows |
|---|---|---|---|
| 3-bit adder | 4 | 16 | 7 |
| 4mod5_1 | 5 | 32 | 9 |
| Cycle10_2 | 12 | 4096 | 37 |
| Cycle17_3 | 20 | 1048576 | 122 |
| Ham3 | 3 | 8 | 4 |
| Ham7 | 7 | 128 | 11 |
| Ham15 | 15 | 32768 | 26 |
| Rd53_2 | 7 | 128 | 22 |
| Rd84 | 15 | 32768 | 195 |
| Hwb6 | 6 | 64 | 60 |
| Hwb7 | 7 | 128 | 126 |

As shown in Table 6-2, for all except two of the benchmark functions studied, the non-zero RM rows only count for a very small fraction of the whole RM spectral table. The only type of functions which do not exhibit this phenomenon is the hidden weight bit (hwb) functions, as shown in the shaded area of Table 6-2. Why do the RM spectra of hwb functions contain many non-zero rows while other types of functions do not? Currently we do not have an explanation for this. We are looking into this problem in depth in the future and are hoping to find more properties related to the RM spectra of different reversible functions.

Below we propose a better implementation of the synthesis algorithm described in Section 6.2 to speed up the synthesis process based on this observation.

The basic algorithm remains essentially the same. The main difference is the way of storing the RM spectra of the specified reversible function. In the new implementation,

only the indexed non-zero rows of the RM spectral table instead of the whole table are stored. So the program only needs to go through the compact RM spectral table and need not check every RM spectral row to see whether it is non-zero.

There are some other minor issues related to the new implementation of the RM spectra based reversible logic synthesis algorithm.

- In the new implementation we need to keep the indices of the non-zero RM spectral rows, i.e., the positions of the non-zero rows in the original full RM spectral table. This adds a small amount of memory overhead, but it is quite worthwhile compared to the large amount of memory saved by the new implementation.
- In C-type steps, it is possible that new non-zero RM spectral rows are created when the current non-zero row is being processed. So in our implementation, a small routine is introduced to solve this problem by inserting the newly created non-zero row(s) into the compact RM spectral table. After insertion, the indices of the non-zero rows are updated.
- Another different approach used in the new implementation is the method of how to apply generalized Toffoli gates which have more than two inputs to the RM spectral table since application of generalized Toffoli gates with more than two inputs involves the AND operation. In the original implementation, the AND operation is done in the Boolean domain as it is very easy to perform the AND operation in this domain. However, in order to work in the Boolean domain we need to transform the whole RM spectra into its Boolean function. Then after the AND operation in the Boolean domain we need to transform it back into the

RM spectral domain. Consequently in the original implementation each application of a Toffoli gate with more than two inputs involves two RM transformations, which require both time and memory to process. In the new implementation, we take advantage of the compact RM spectral table. Since only a small fraction of the RM spectral table contains non-zero elements, it is easy to carry out the AND operation in the RM spectral domain using Equation 2.3 given in Section 2.3: only the non-zero elements need to be considered. Thus no RM transform is required in the new implementation.

Now we consider the same example as given in Table 6-1. In the new implementation, the compact form of the RM spectral table, as well as the indices of the non-zero RM spectral rows, of the specified function is stored, as shown in the shaded area of Table 6-3. The rest of the process is similar to that of the initial implementation except the way of applying the last gate TOF(a, b; c) in step 2. The final outputs are exactly the same as those of the old implementation.

TABLE 6-3 EXAMPLE: IMPROVED IMPLEMENTATION

| Index | Function | Step 1 | Step 2 | Identity |
|---|---|---|---|---|
| | cba | cba | cba | cba |
| 1 | 010 | 001 | 001 | 001 |
| 2 | 001 | 011 | 010 | 010 |
| 3 | 111 | 110 | 111 | 000 |
| 4 | 111 | 110 | 111 | 100 |
| Gate applied | TOF(b;a)<br>TOF(a;b) | TOF(b;a) | TOF(c;b)<br>TOF(c;a)<br>TOF(a,b:c) | |

In this example, the function has three variables, so the full RM spectral table contains $2^3 = 8$ rows. In our new implementation, four rows are eliminated from the original RM spectral table. This clearly indicates the advantage of the new implementation over the old one in the aspect of saving memory. Now consider a much bigger benchmark function, function Rd84, which has 15 variables. The whole RM spectral table of function Rd84 contains 32768 rows. However in the new implementation, only 195 non-zero RM spectral rows are required to be stored. This memory-saving is significant. And from our observation, this new implementation is suitable for most of the benchmark functions except the hwb ones.

## 6.4 Experimental Results

In this section we give experimental results to illustrate the improved performance of the new implementation of the synthesis algorithm over the old one. Simulation was conducted on some of the benchmark functions, using both the initial and the new approaches of the synthesis algorithm. Execution time is recorded to evaluate the performance. For all the simulation, we use a Pentium 4 processor with 3.00 GHz frequency and 2.00 GB of RAM memory running Windows XP.

Table 6-4 gives the execution time (in seconds) of some benchmark functions for both the original implementation (column 3) and the new one (column 4). In the table, ‘a’ indicates a time which is less than 0.0005 seconds.

TABLE 6-4 RM SPECTRA OF BENCHMARK FUNCTIONS

| Function | Number of inputs | Execution time (seconds) | |
|---|---|---|---|
| | | Original implementation | New implementation |
| 3-bit adder | 4 | a* | a* |
| 4mod5_1 | 5 | a* | a* |
| Cycle10_2 | 12 | 0.051 | 0.015 |
| Cycle17_3 | 20 | 60.125 | 3.843 |
| Ham7 | 7 | 0.062 | 0.062 |
| Rd53_2 | 7 | a* | a* |
| Rd84 | 15 | 0.234 | 0.015 |
| Rd73 | 10 | a* | a* |
| Mod1024adder | 20 | 31.875 | 1.843 |

*: a indicates a time less than 0.0005 secs.

We can see that for small benchmark functions such as 3-bit adder and 4mod5_1, the new implementation of the synthesis algorithm does not show much better performance over the old one. There are two reasons for this: (i) The processor used for simulation is too powerful. It works at a very high clock rate and has more than enough RAM memory. (ii) For the small functions, it takes a small amount of memory to store the whole RM spectral table. So, the compact RM spectral table does not show significant improvement in these cases. However, when we re-run the simulation for benchmark function Rd73 on a less-powerful machine which has only 512 MB memory, the runtime is 0.050s and 0.001s for the old approach and new approach respectively, indicating that the new program is about 50 times faster than the old one for function Rd73. This gives us a better view of the improvement achieved by the new implementation in a resource-limited environment.

As the number of variables in the benchmark functions increases, the new implementation exhibits much improved performance. For instance, for benchmark function Cycle10_2, the new implementation is about 3.4 times faster than the old one. For other large benchmark functions such as Cycle17_3, Rd84, and Mod1024adder, the new program is more than 15 times faster than the old one. These results show that the new implementation of the synthesis algorithm is much more efficient than the old one.

Results also indicate that our new approach can be used to synthesize much larger reversible functions. In our experiment, the longest synthesis process happens for benchmark function Cycle17_3, which has 20 inputs, but the process takes less than 4 seconds. This indicates that our new implementation of the RM spectra based synthesis algorithm has the capacity for much larger reversible functions.

When conducting simulation with the new implementation of the synthesis algorithm we also collected information on the number of new non-zero RM spectral rows which were created and added to the compact RM spectral table during the synthesis process. These results are presented in Table 6-5.

TABLE 6-5 NEW NON-ZERO RM ROWS FOR EACH BENCHMARK FUNCTION

| Function | Number of rows in the whole RM spectra table | Number of non-zero RM rows | |
|---|---|---|---|
| | | Before synthesis | After synthesis |
| 3-bit adder | 16 | 7 | 7 |
| 4mod5_1 | 32 | 9 | 9 |
| Cycle10_2 | 4096 | 37 | 37 |
| Cycle17_3 | 1048576 | 122 | 122 |
| Ham7 | 128 | **11** | **122** |
| Rd53_2 | 128 | 22 | 22 |
| Rd84 | 32768 | **195** | **287** |
| Rd73 | 1024 | **104** | **117** |
| Mod1024adder | 1048576 | 1033 | 1033 |

The results show that for 6 out the 9 benchmark functions, no new non-zero RM spectra row was inserted to the compact RM spectra table. For functions Rd73 and Rd84, although some new non-zero RM rows were introduced, the resulting number of non-zero rows is still quite small compared to the number of rows in their corresponding full RM spectral table, which is equal to $2^n$, where $n$ is the number of inputs of the benchmark function.

However, for the remaining function Ham7, whose original RM spectral table contains only 11 non-zero row, during synthesis, 111 new non-zero rows were created, resulting in the total number of non-zero rows reaching 122, which is very close to $2^7$=128, the total number of rows in the full RM spectral table. In this case the new implementation did not show much advantage over the old one. This explains why in Table 6-4 the execution time of function Ham7 is about the same for both old and new implementations. This

implies that the new implementation has little advantage over the old one for Ham type functions.

The hwb functions were not simulated since their RM spectral tables contain very few zero rows and the size of the RM spectral table is not reduced much if we delete the zero RM rows. Consequently, our new algorithm would show little improvement over the original for these functions.

## 6.5 Conclusion

In this chapter we introduce a new approach to implement a Reed-Muller spectra based algorithm for synthesizing Toffoli networks. This new idea uses a compact RM spectral table instead of the whole RM spectral table of the function since it takes advantage of the phenomenon that the RM spectral tables of many benchmark functions have a large number of zero rows. This new approach greatly reduces runtime memory consumption. Experimental results showed that this technique speeds up the synthesis process significantly.

Our simulation results also showed that the synthesis process of the new implantation took a very small amount of time. Hence we should be able to use this modified algorithm to synthesize much larger functions. More work remains to be done regarding this aspect.

It is mentioned that not all functions showed the phenomenon that their RM spectra table contain a large number of zero rows. For the benchmark circuits we examined, only the hwb functions did not exhibit this property. In the future we will investigate how this

is related to different reversible functions so that we can apply this technique to target function groups.

Future work also includes studying the properties of RM spectra of different reversible logic functions as well as the investigation of possible approaches of combining this technique with other synthesis algorithms.

# Chapter 7 Conclusion

In this chapter we summarize our work in this thesis and give several possible future research directions.

## 7.1 Contributions

Our contributions to the area of reversible logic computing focus on testing reversible logic circuits (mainly Toffoli networks) and reversible logic synthesis. They are summarized as follows:

- We propose a new fault model, labeled as the crosspoint fault model, for reversible circuits. This new fault model represents the situations where the control points in a Toffoli gate do not function properly. We show that this new fault model is a better fault model for reversible circuits than the standard stuck-at fault model defined in conventional irreversible logic circuits. Both theoretical analysis and experimental results show that the crosspoint fault model dominates the stuck-at fault model in most instances.
- We give a randomized ATPG method for the new defined crosspoint faults in reversible circuits. The performance of this ATPG algirithm is compared with other ATPG algorithms. Simulation results show that our randomized ATPG algorithm generates minimum or near-minimum test sets for crossspoint faults in a comparably small amount of time.
- We show that a certain type of redundancy in reversible logic circuits can be related to the crosspoint fault model. Undetectable multiple disappearance faults

represent redundancy in reversible circuits. By deleting the independent multiple crosspoint faults, we are able to reduce the quantum cost of the reversible logic circuits.

- We examine the testability of multiple crosspoint faults extensively and provide an efficient algorithm to simplify redundant Toffoli networks. Experimental results show that many benchmark circuits can be further simplified. We also show that in many cases this type of redundancy cannot be identified or reduced using previous approaches such as templates.
- We study the double mixed crosspoint faults and illustrate that they can be used to find a set of functionally equivalent alternative circuits for a given reversible logic circuit.
- We present an optimized implementation of a Reed-Muller spectra based reversible logic synthesis algorithm proposed by Maslov et al [29]. Our new implementation uses a compact Reed-Muller spectra table to save memory and is proved to be able to speed up the synthesis process significantly. This new implementation can also be used to synthesize much larger reversible logic functions.

## 7.2 Future Work

There is still much research waiting to be undertaken in reversible logic computing,

either related or not related to the work we have presented in this thesis. In the following we give some of the possible further research topics that build directly on our results.

- In the aspect of the crosspoint fault model defined in this thesis for reversible logic circuits, it would be interesting to investigate issues such as the relationship between single crosspoint faults and multiple crosspoint faults, properties of multiple crosspoint faults in Toffoli networks, and more applications of crosspoint faults in reversible logic synthesis.
- We are also considering other possible new fault models for reversible logic circuits, which might be based more closely on the technology with which they are implemented. Investigation of the properties of reversible logic circuits under the new fault models, and the ATPG problem for these models, will also be required.
- It is useful to study reversible logic circuits which are constructed with gates not restricted in the library of generalized Toffoli gates. New fault models might better represent abnormal behaviors of those gates. The relationship between different fault models in reversible logic circuits should also be investigated.
- As mentioned in chapter 7, we can examine the Reed-Muller spectral properties of different reversible logic benchmark functions more extensively. Looking into this problem could classify different reversible logic functions into different groups according to their Reed-Muller spectral properties and thus we could use the optimized synthesis algorithm for the targeted function groups to speed up the synthesis process.
- More research could be undertaken on the properties of the Reed-Muller spectra

of reversible logic functions and other possible application of these properties in reversible logic synthesis and testing reversible circuits.

- It would be interesting to look at the properties of reversible logic functions in other spectral domains.

# Bibliography

[1] Abramovici M., Breuer M. A., and Friedman A. D.: 'Digital Systems Testing and Testable Design', Revised Printing, IEEE Press, New York, 1995

[2] Agrawal A., Jha N. K., 'Synthesis of Reversible Logic', Proceedings of Design, Automation and Test in Europe Conference and Exhibition, pp. 1384-1385, 2004

[3] Allen J., Biamonte J., Perkowski M.: 'ATPG for Reversible Circuits using Technology-Related Fault Models', Proceedings of International Symposium on Representations and Methodologies for Emergent Computing Technologies, pp. 100-107, 2005

[4] Athas W. C., Svensson L. J.: 'Reversible Logic Issues in Adiabatic CMOS', Proceedings of Workshop on Physics and Computation, pp. 111-118, 1994

[5] Bardell P. H., McAnney W. H., Savir J.: 'Built-In Test for VLSI: Pseudorandom Techniques', John Wiley and Sons, 1987

[6] Barenco A., Bennett C. H., Cleve R., DiVincenzo D. P., Margolus N., Shor P., Sleator T., Smolin J., and Weinfurter H.: 'Elementary Gates for Quantum Computation', Phys Rev A. 52, 5, pp. 3457-3467,1995

[7] Bennett C. H.: 'Logical Reversibility of Computation', IBM Journal of Research and Development, 17, pp. 525-532, 1973

[8] Biamonte J. D., Allen J., Lukac M., and Perkowski M.: **'**Principles of Quantum Fault Detection', McNair Research Journal, 2004

[9] Biamonte J. D., Perkowski M.: **'**Testing a Quantum Computer', Proceedings of KAIS- WQIS 5th Workshop on Quantum Information Science, 2004

[10] Boole G.: 'An Investigation of the Laws of Thought', Prometheus Books, 2003

[11] Cattell K., Muzio J. C.: 'Analysis of One-Dimensional Linear Hybrid Cellular Automata over GF(q), IEEE Transactions on Computers, Vol. 45, pp. 782-792, 1996

[12] Chakraborty A.: 'Synthesis of Reversible Circuits for Testing with Universal Test Set and C-Testability of Reversible Iterative Logic Arrays', Proceedings of 18th International Conference on VLSI Design, pp. 249-254, 2005

[13] Damarla T. R., Karpovsky M.: 'Fault Detection in Combinational Networks by Reed-Muller Transforms', IEEE Transactions on Computers, Vol. 36, Issue 6, pp. 788-797, 1989

[14] De Vos A.: 'Towards Reversible Digital Computers', European Conference on Circuit Theory and Design, pp. 923-931, 1997

[15] Falkowski: 'Relationship between Haar and Reed-Muller Spectral and Functional Domains', IEICE Electronics Express, Vol. 2, No. 1, pp. 37-42, 2005

[16] Feynman R.: 'Quantum Mechanical Computers', Optics News, 11, pp. 11-20, 1985

[17] Frank M. P.: 'Introduction to Reversible Computing: Motivation, Progress, and Challenges', Proceedings of the 2nd Conference on Computing Frontiers, pp. 385-390, 2005

[18] Fredkin E., and Toffoli T.: 'Conservative Logic', International Journal of Theoretical Physics, 21, pp. 219-253, 1982

[19] Hayes J. P., Polian I., and Becker B.: 'Testing for Missing-Gate Faults in Reversible Circuits', IEEE Proceedings of the 13th Asian Test Symposium, pp. 100-105, 2004

[20] Hurst S. L., Miller D. M., Muzio J. C.: 'Spectral Techniques in Digital Logic', Academic Press, 1985

[21] Jha N., Gupta S.: 'Testing of Digital Systems', Cambridge University Press, 2003

[22] Kim J., Lee J. S., Lee S.: 'Implementing Unitary Operations in Quantum Computation', Physical Review A, 62, 2000

[23] Koeter J.: 'What's an LFSR?', Application Report, Texas Instruments, SCTA036A, 1996

[24] Koller J. G., Athas W. C., Svensson L. J.: 'Thermal Logic Circuits', Proceedings on Physics and Computation, pp. 119-127, 1994

[25] Landauer R.: 'Irreversibility and Heat Generation in the Computational Process', IBM Journal of Research and Development, 3, pp. 183-191, 1961

[26] Maslov D.: 'Reversible Logic Synthesis', Ph. D. Dissertation, University of New Brunswick, 2003

[27] Maslov D.: 'Reversible Logic Synthesis Benchmarks Page', www.cs.uvic.ca/~dmaslov/, accessed Oct. 2008

[28] Maslov D., Dueck G. W., and Miller D. M.: 'Templates for Toffoli Network Synthesis', Proceedings of International Workshop on Logic Synthesis, pp. 320-326, 2003

[29] Maslov D., Miller D. M., and Dueck G. W.: 'Reed-Muller Spectra Based Synthesis of Reversible Circuits Using a Quantum Cost Metric', 7th International Symposium on Representations and Methodology of Future Computing Technologies (RM), Tokyo, Japan, September 2005

[30] Merkle R. C.: 'Reversible Electronic Logic Using Switches', Nanotechnology, Vol 4, pp. 21-40, 1993

[31] Miller D. M., and Dueck G. W.: 'Spectral Techniques for Reversible Logic Synthesis', 6th International Synposium on Representations and Methodology of Future Computing Technologies, pp. 56-62, 2003

[32] Miller D. M., Maslov D., and Dueck G. W.: 'A Transformation Based Algorithm

for Reversible Logic Synthesis', Proceedings of the Design Automation Conference, pp. 318-323, 2003

[33] Mishchenko A., and Perkowski M.: 'Logic Synthesis of Reversible Wave Cascades', Proceedings of International Workshop on Logic Synthesis, pp. 197-202, 2002

[34] Nielsen M. A., and Chuang I. L.: 'Quantum Computation and Quantum Information', Cambridge University Press, New York, 2000

[35] Pan W. D., and Nalasani M.: 'Reversible Logic', IEEE Potentials, 24, 1, pp. 38-41, 2005

[36] Parhami B.: 'Fault-Tolerant Reversible Circuits', 40th Asilomar Conference on Signals, Systems and Computers, pp. 1726-1729, 2006

[37] Patel K. N., Hayes J. P., and Markov I. L.: 'Fault Testing for Reversible Circuits', IEEE Transaction on CAD, 23, pp. 1220-1230, 2004

[38] Peres A.: 'Reversible Logic and Quantum Computers', Phys Rev A. 32, pp. 3266-3276, 1985

[39] Perkowski M., Biamonte J., and Lukac M.: 'Test Generation and Fault Localization for Quantum Circuits', Proceedings of 35th International Symposium on Multiple-Valued Logic, pp. 62-68, 2005

[40] Picton P.: 'Opoelectronic, Multivalued, Conservative Logic', International Journal of Optical Computing, Vol 2, pp. 19-29, 1991

[41] Picton P.: 'A Universal Architecture for Multiple-valued Reversible Logic', MVL Journal, Vol 5, pp. 27-37, 2000

[42] Pierce D., and Biamonte J. D.: 'Fault Localization for Reversible Circuits', Project Report, Portland Quantum Logic Group, Jan. 2005

[43] Pierce D., and Biamonte J., Perkowski M.: 'Test Set Generation and Fault

Localization Software for Reversible Circuits', Proceedings of International Symposium on Representations and Methodologies for Emergent computing Technologies, pp. 42-49, 2005

[44] Polian I., Hayes J. P., Fiehn T., and Becker B.: 'A Family of Logical Fault Models for Reversible Circuits', IEEE Proceedings on the 10th European Test Symposium, pp. 422-427, 2005

[45] Ramasamy K., Tagare R., Perkins E., and Perkowski M.: 'Fault Localization in Reversible Circuits is Easier than for Classical Circuits', Proceedings of International Workshop on Logic Synthesis, 2004

[46] Shende V. V., Prasad A. K., Markov I. L., and Hayes J. P.: 'Reversible Logic Circuits Synthesis', IEEE/ACM International Conference on Computer Aided Design, pp. 353-360, 2002

[47] Thornton M. A., Miller D. M., Drechsler R., Transformations amongst the Walsh, Haar, Arithmetic and Reed-Muller Spectral Domains, International Workshop on Applications of the Reed-Muller Expansion in Circuit Design, pp. 215-225, August 2001

[48] Toffoli T.: 'Reversible Computing, in Automata, Languages and Programming', 7th Colloquium, J. W. de Bakker and J. van Leeuwen, Eds, New York, Springer-Verlag, Lecture Notes in Computer Science, pp. 632-644, 1980

[49] Vasudevan D. P., Lala P. K., Parkerson J. P.: 'Online Testable Reversible Logic Circuit Design using NAND Blocks', Proceedings of 19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, vol. 00, pp. 324-331, 2004

[50] Zhong J., and Muzio J. C.: 'Analyzing Fault Models for Reversible Logic Circuits', IEEE World Congress on Computational Intelligence, special session on Quantum Computing and Quantum Computational Intelligence, pp 2422-2427, 2006

[51] Zhong J., and Muzio J. C.: 'Using Crosspoint Faults in Simplifying Toffoli Networks', the 4th International IEEE Northeast Workshop on Circuits and Systems,

pp 129-132, 2006

[52] Zhong J., and Muzio J. C.: 'Improved Implementation of a Reed-Muller Spectra Based Reversible Synthesis Algorithm', IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp 202-205, 2007

[53] Zuliani P.: 'Logic Reversibility', IBM Journal of Research and Development, 45, pp. 807-818, 2001