Reversible Circuits

draft

May 2023

0.1 Introduction

The main purpose of studying reversible circuits is the theoretical possibility of being able to perform computations at almost zero energy cost.
Bennett demonstrated that it is possible to construct a reversible turing machine that does not destroy information and thus can be operated with very little energy.
Also, according to the Landauer principles, energy is always dissipated when a logically irreversible action is performed.

A logical circuit is reversible if it is a bi-jective (one-to-one and onto) function, which means every input combination is uniquely mapped with an output combination. This effectively enables us to determine the input using the output which is not always possible in classical computation.
Reversible Circuits are built using gates similar to how non-reversible circuits are made. These gates are reversible in nature as well.
A very well-known reversible gate is the Toffoli gate. The Toffoli gate has 2 control lines and 1 target line, if the controls are represented using a, b and the target is represented using c. After passing through the Toffoli gate, the target line becomes $c \oplus a \wedge b$ where $\oplus$ represents the XOR operation and $\wedge$ represents the AND operation. In simpler terms, if a and b are 1, c gets flipped.
Reversible circuits has gained importance in fields such as cryptography, communications and digital signal processing where it requires computations that transforms the data without erasing any of the original information. However, the applicability of reversible circuits is not limited to applications that are inherently reversible.*[Taken from Introduction section of Ketan N. Patel, John P. Hayes, Fellow, IEEE, and Igor L. Markov, Member, IEEE]*. It is can also be used in conventional irreversible computation.

The testing of conventional reversible circuits has been primarily considered in respect to fault models. Our approach involves a popular fault model, called the missing gate fault model, which is best suited to quantum technologies. We strive to find a minimum test vector that will determine the correctness of a reversible circuit, which covers all the possible missing gate faults.

0.2 Faults

0.2.1 What is a Fault?

A fault is a disturbance in the system which causes the circuit to deviate from it's intended behaviour. A certain input is said to detect a fault when it's resultant output is different in the absence and presence of the fault. Faults may or may not be detectable. Some faults do not affect the final output of the circuit and they are called redundant faults.

0.2.2 Types of Faults:

There are many types of faults but for our purposes we will be focusing on these three types:

Single Missing Gate Fault

1. **Definition**
   Abbreviated as SMGF, as the name suggests, single missing gate fault refers to the absence of a single gate from the given circuit.

2. **Detection**
   An input which produces 1's at every target line for the gate in question can be used to detect Single Missing Gate Faults for that gate. For a n-Gate circuit, there can be a maximum of n SMGFs.

Partial Missing Gate Faults

1. **Definition**
   Abbreviated as PMGF, partial missing gate fault refers to the absence of one or more controls from a given gate. For our purposes, we will only be talking about 1° partial missing gate faults.

2. **Detection**
   If a given gate has n-controls, an input that produces 1 at n-1 such controls can be used to detect partial missing gate faults for that gate. For a n-Gate circuit and every gate having c_i controls where $i \in [1, n]$, the total number of 1° PMGFs can be written as $\sum_{i=1}^{n} c_i$.

Multiple Missing Gate Faults

1. **Definition**
   Abbreviated as MMGF, multiple missing gate fault refers to the absence of more than one, continuous gates for a circuit.

2. **Detection**
   If a given gate has n-controls, an input that produces 1 at n-1 such controls can be used to detect partial missing gate faults for that gate. For a n-Gate circuit the total number of MMGFs can be written as $\frac{(n) \times (n-1)}{2}$.

0.3 Motivation

The number of input combination grows exponentially with the number of input lines. For example for 5 input lines there can be 2^5 input combinations. As the number of input line grows, it becomes infeasible to detect flaws in the circuit efficiently. Hence there arises a need for a minimal test set of input vectors that covers most if not all faults possible for a given circuit.
Finding such a minimal set is a variation of the set-cover problem. The set-cover problem is a NP-Hard problem which implies there is no optimal way to determine if a such a given input vector is the minimal set or not.
We will only be considering SMGF, 1° PMGF and MMGF. Not all MMGFs are detectable, some can go completely undetected.

0.4 Approaches

0.4.1 First Approach

Lets say the circuit has n input lines and g gates. We start off by simulating the circuit and storing the SMGF, PMGF and MMGF faults that can be identified using the input. This pre-computation takes 2^n cycles since there are 2^n possible input vectors.

A gate can be represented using two integers, a target and a control

$$\begin{aligned}
\textbf{n} &\leftarrow \text{number of input lines for that gate}\\
\textbf{target} &\leftarrow \text{n-length binary number with 1 bit at location of target}\\
\textbf{control} &\leftarrow \text{n-length binary number with 1 bits at control points}\\
\\
\textbf{Gate} &\leftarrow \text{Pair(target, control)}
\end{aligned}$$

As a result, every circuit can be represented by the following properties

$$\begin{aligned}
\textbf{g} &\leftarrow \text{Number of Gates in the circuit}\\
\textbf{n} &\leftarrow \text{Number of Lines in the circuit}\\
\\
\textbf{Circuit} &\leftarrow \text{List of } g \text{ gates with } n \text{ input lines}
\end{aligned}$$

Given a circuit having g gates, we can determine the number of faults that are possible:

Algorithm 0.1: Counting Faults

```
    g ← number of gates

  Single Missing Gate Faults:
    count_smgf ← g

  Partial Missing Gate Faults:
    count_pmgf ← $\sum_{i=1}^{g}$ control points *in* $gate_i$

  Multiple Missing Gate Faults:
    count_mmgf ← $\frac{(g) \times (g-1)}{2}$

 Total Faults:
    count_total ← count_smgf + count_pmgf + count_mmgf
```

A filtered set of input vector that can be used to identify all faults can be generated using the following algorithm:

Algorithm 0.2: Filtered Set of Inputs

```
begin
n ← number of input lines
fault_mapping ← mapping of a fault to unique integer IDs

function filterInputs(n, fault_mapping):
    leftPointer ← 0
    rightPointer ← $2^n - 1$

    faults_covered ← {}
    fault_encountered ← mapping of input vector to set of faults identified by it.
    total_faults ← total number of SMGFs, 1° PMGFs and MMGFs possible

    function simulate_circuit(input_vector):
        simulated_faults ← all faults identified by input_vector
        set_of_IDs ← [fault_mapping[fault] for fault in simulated_faults]
        return set_of_IDs
    end simulate_circuit


    while len(faults_covered) < total_faults AND leftPointer <= rightPointer:
        left_new_faults ← simulate_circuit(leftPointer)
        faults_covered ← faults_covered ∪ left_new_faults
        fault_encountered[leftPointer] ← left_new_faults

        right_new_faults ← simulate_circuit(rightPointer)
        faults_covered ← faults_covered ∪ right_new_faults
        fault_encountered[rightPointer] ← right_new_faults
    end while

    return fault_encountered

end filterInputs
```

After this set has been constructed, we select the input vectors from this set greedily to construct the final set of input vectors that cover all possible faults:

Algorithm 0.3: Greedily selecting input vectors

```
begin

fault_encountered ← mapping between input vector and corresponding fault set identified

function greedySelection(fault_encountered):
    function setDifference(A, B):
        A.fault_set ← A.fault_set \ B.fault_set
        return A
    end setDifference

    fault_list ← list of tuples [(i, f)]
                                for each key−value pair (i, f) in fault_encountered

    selection ← {}
    selected_input_vectors ← []

    while fault_list is NOT empty:

        highest ← max(fault_list) w.r.t fault_set length
        selected_input_vectors ← selected_input_vectors ∪ highest.input_vector
        selection ← selection ∪ highest.fault_set

        fault_list ← [setDifference(element, highest) for element in fault_list]
        fault_list ← fault_list.filter(element.fault_set is not empty)
    end while

    return selected_input_vectors
end greedySelection
```