A Stochastic Computational Approach for Accurate and Efficient Reliability Evaluation: A Python Implementation

Jake Humphrey
Department of Electronic and Electrical Engineering
Imperial College London
jbh111@ic.ac.uk

November 27, 2014

1 Abstract

This paper details my work relating to the reliability evaluation approach detailed by J.Han et al in their paper *A Stochastic Computational Approach for Accurate and Efficient Reliability Evaluation*[1]

Versions of this work are available in both French and English at my Github page `github.com/EightAndAHalfTails/elec-221-stochastic`

2 The Subject Paper

In this section I will discuss details relating to the paper under discussion.

2.1 Summary

J.Han et al investigate existing reliability evaluation methods, such as the Probabilistic Gate Model approach, which attempts to analytically derive the reliability of a circuit. They discover that this approach is difficult or impossible for large circuits, as correlations between signals and nodes in the circuit exponentially increase the complexity.

They then move on to examining existing non-deterministic simulation-based algorithms, such as Monte Carlo Simulation, or methods derived from Stochastic Computing. While these intrinsically handle correlated signals, they still require a large computation overhead for the generation of pseudorandom numbers.

In the paper, J.Han et al propose their own method derived from Stochastic Computing which requires far fewer pseudorandom numbers to be generated. They achieve this by generating input bitstreams deterministically and then randomly permuting them.

3 My Work

In this section I will discuss details of the mini-project I did relating to the subject paper.

3.1 Summary

I decided to implement the solution described in the paper, to write a program that would take in a circuit netlist (in Verilog HDL), and probabilities for inputs and gate errors, and print out the reliability of each output.

I considered writing this in C++ and/or Verilog/VHDL, but decided on Python for ease of writing. This is also a proof of concept, and not a robust or efficient program.

3.2 The Proposed Algorithm

Algorithm 1 describes my proposed algorithm. The final double loop is inefficient in that evaluating the outputs backwards through the circuit for each output entails calculating many intermediate values redundantly. A better simulation-based method would be to trace the inputs forwards to the outputs, which would be only $O(\#gates \times \#inputvectors)$ as opposed to $O(\#gates \times \#inputvectors \times \#outputs)$. However, I chose this method to simplify the evaluation function.

```
Data: Logic circuit to be tested, input and gate eror probabilities
Result: Reliabilities for each output
for each gate in the circuit do
    represent gate in faulty circuit;
end
for each input to the faulty circuit do
    generate a non-Bernoulli sequence;
end
for each output in the circuit do
    for each input vector do
        if outputs are the same for each circuit then
            add 1/n to reliability of that output;
        end
    end
end
```

Algorithm 1: My pseudocode implementing the proposed solution

3.3 Work Details

The Python code has been included here in Appendix A. It can also be found at my Github page, the url for which was given in Section 1, along with presentation slides and a script in both English and French.

4 Appendices

A Python Code

```
#!/bin/env python2

import random
import timeit
from collections import defaultdict
ITER = 10000

class Circuit:
  def __init__(self, i=[], o=[], g=dict()):
    self.inputs = i # list of strings containing input node names
    self.outputs = o # list of strings containing output node names
    self.gates = g # dictionary keyed by output of all gates in the circuit
  def printout(self):
    print "inputs", self.inputs
    print "outputs", self.outputs
    print "gates:"
    [x.printout() for x in self.gates.values()]

class Gate:
  def __init__(self, name, op, ins, out):
    self.name = name
    self.operation = op
    self.inputs = ins
    self.output = out
  def printout(self):
    print "name: {}, op: {}, {} -> {}".format(self.name, self.operation, self.inputs, self.o

def parse(filename):
  """reads in a verilog file and returns a Circuit instance containing
     the circuit descibed by the file"""
  gates = dict()
  with open(filename) as f:
    for l in f.readlines():
      l = l.replace('\n', '')
      if l == "":
        continue
      if l[:2] == "//":
        continue
      command = l.split(" ")[0].lower()
      if command == "module":
        pass
      elif command == "input":
        inputs = l.replace("input ", "")
        inputs = inputs.replace(';', '').replace(" ", "") # strip the semicolon and whitespa
        inputs = inputs.split(',') #extract input tokens
      elif command == "output":
        outputs = l.replace("output ", "")
        outputs = outputs.replace(';', '').replace(" ", "") # strip the semicolon and whites
        outputs = outputs.split(',') #extract output tokens
      elif command == "wire":
        pass
      elif command in ["nand", "and", "or", "nor", "xor", "xnor", "not"]:
        connections = l.split('(')[1].replace(" ", "").replace(");", "").split(',')
        out = connections[0]
        ins = connections[1:]
        name = l.split(" ")[1]
        gates[out] = Gate(name, command, ins, out)
      else:
        pass
        #print "cannot parse line: " + l
  return Circuit(inputs, outputs, gates)

def evaluate(node, circuit, inputval):
"""Given a Cicuit instance and input vector, this function
derives the value at a given node in the circuit"""
  if node in circuit.inputs:
    return inputval[node]
  #circuit.printout()
  #print node
  gate = circuit.gates[node] #find gate whose output is node. O(1)
  inputs = [evaluate(i, circuit, inputval) for i in gate.inputs] # get values at input
  if gate.operation == "and":
    return all(inputs)
  elif gate.operation == "nand":
    return not all(inputs)
  elif gate.operation == "or":
    return any(inputs)
  if gate.operation == "nor":
    return not any(inputs)
  elif gate.operation == "xor":
    # true if no. of Trues is odd
    return inputs.count(True)%2 == 1
  elif gate.operation == "xnor":
    # true if all inputs are the same
    return all(inputs) or not any(inputs)
    #return inputs.count(True) == 0 or inputs.count(False) == 0
  elif gate.operation == "not":
    return not inputs[0]
  else:
    print "Error at gate", gate.name
    return False

def getVNFaulty(circuit):
"""Given a circuit instance, this function creates a new Circuit instance, representing
the original with each gate in the circuit appended with an xor gate"""
  faulty_gates = dict()
  error_ins = []
  for ideal_gate in circuit.gates.values():
    intermed_node = ideal_gate.name + "__inter__"
    faulty_gate = Gate(name = ideal_gate.name,
                       op = ideal_gate.operation,
                       ins = ideal_gate.inputs,
                       out = intermed_node)
    error_name = ideal_gate.name + " VNE"
    xor_gate = Gate(name = error_name,
                    op = "xor",
                    ins = [intermed_node, error_name],
                    out = ideal_gate.output)
    error_ins.append(error_name)
    faulty_gates[faulty_gate.output] = faulty_gate
    faulty_gates[xor_gate.output] = xor_gate
  faulty_circuit = Circuit(circuit.inputs + error_ins, circuit.outputs, faulty_gates)
  return faulty_circuit

def exhaust(circuit):
"""This function exhaustively tries all inputs to a circuit and prints the corresponding ou
  inputval = dict()
  for iteration in range(2**len(circuit.inputs)):
    input_vector = [(iteration>>i)&1 for i in xrange(len(circuit.inputs)-1,-1,-1)]
    for i in range(len(circuit.inputs)):
      inputval[circuit.inputs[i]] = input_vector[i]==1
    print input_vector, [evaluate(n, circuit, inputval) for n in circuit.outputs]

def getNonBernoulliSequences(circuit):
"""This fuction, given a circuit, returns a dictionary instance mapping input node names to
  inputval = dict()
  for input_node in circuit.inputs:
    prob = float(raw_input("Enter probability of {}: ".format(input_node)))
    num_ones = int(prob*ITER)
    non_ber = [True]*num_ones + [False]*(ITER-num_ones)
    random.shuffle(non_ber)
    inputval[input_node] = non_ber
  return inputval

def findReliabilities(filename):
  # create circuit from file
  ideal = parse(filename)

  # create error-prone version of the circuit
  faulty = getVNFaulty(ideal)

  # create input sequences for all inputs (including gate errors)
  inputval = getNonBernoulliSequences(faulty)

  outputs_same = defaultdict(list)

  # for each interation
  for i in range(ITER):
    # get the ith element of each input bitstream
    inputvec = {x:inputval[x][i] for x in inputval.keys()}

    # or each output
    for output_node in ideal.outputs:
      # find the value of the output in each circuit
      ideal_output = evaluate(output_node, ideal, inputvec)
      faulty_output = evaluate(output_node, faulty, inputvec)

      # success if they are the same
      outputs_same[output_node].append(ideal_output == faulty_output)

  #print reliability of that output
  for n, v in outputs_same.iteritems():
    print "reliability of {} is {}".format(n, 1.0*v.count(True)/ITER)
  # Complexity = O(ITER * len(circuit.outputs) * len(circuit.gates))

#reps = 1
#time = timeit.timeit('findReliabilities("./c17.v")', setup = "from __main__ import findRel
#time = timeit.timeit('findReliabilities("./c432.v")', setup = "from __main__ import findRe
#print time

findReliabilities("./c17.v")
#findReliabilities("./c432.v")
```

References

[1] J. Han, H. Chen, J. Liang, P. Zhu, Z. Yang, and F. Lombardi, *A Stochastic Computational Approach for Accurate and Efficient Reliability Evaluation*, 2012.