Une Approche de Calcul Stochastique pour l'Evaluation de Fiabilité Précis et Efficace: Une Implementation Python

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

November 27, 2014

1 Abstract

Cet article détaille mon travail relatif à l'approche de l'évaluation de la fiabilité détaillé par J.Han et al dans leur document *A Stochastic Computational Approach for Accurate and Efficient Reliability Evaluation*[1]

Versions de ce travail sont disponibles en français et en anglais à ma page Github `github.com/EightAndAHalfTails/elec-221-stochastic`

2 Le Papier en Question

Dans cette section, je discuterai des détails relatifs au papier en cours de discussion.

2.1 Résumé

J.Han et al enquêter sur les méthodes d'évaluation de fiabilité existants, tels que l'approche de la modèle de la porte probabiliste, qui tente de tirer analytiquement la fiabilité d'un circuit. Ils découvrent que cette approche est difficile, voire impossible pour les grands circuits, car les corrélations entre les signaux et les nœuds dans le circuit augmentent de façon exponentielle la complexité.

Ils passent ensuite à examiner les algorithmes existants non déterministes fondés sur la simulation, comme la simulation Monte Carlo, ou méthodes dérivées du calcul stochastique. Bien que ceux-ci traitent intrinsèquement des signaux corrélés, ils nécessitent toujours une grande surcharge de calcul pour la génération des nombres pseudo-aléatoires.

Dans le papier, J.Han et al proposer leur propre méthode dérivée du calcul stochastique qui nécessite beaucoup moins de nombres pseudo-aléatoires à

générer. Ils y parviennent en générant des flux binaires d'entrée déterministe puis les permutant au hasard.

3 Mon Travail

Dans cette section, je discuterai des détails du mini-projet que j'ai fait en concernant le document de sujet.

3.1 Résumé

J'ai décidé de mettre en œuvre la solution décrite dans le papier, d'écrire un programme qui tiendrait dans une liste d'interconnexions de circuit (en Verilog HDL), et les probabilités pour les entrées et les erreurs des portes, et imprimer la fiabilité de chaque sortie.

J'ai pensé à l'écrire en C ++ et/ou Verilog/VHDL, mais a décidé sur Python pour la facilité de l'écriture. C'est aussi une preuve de concept, et non un programme particulièrement solide ou efficace.

3.2 l'Algorithme Proposé

Algorithme 1 décrit mon algorithme proposé. La double boucle finale est inefficace en ce que l'évaluation des sorties vers l'arrière à travers le circuit pour chaque sortie consiste à calculer de nombreuses valeurs intermédiaires redondante. Une méthode préférable basée sur la simulation serait de suivre les entrées vers l'avant pour les sorties, ce qui serait seulement $O(\#portes \times \#vecteurs d'entrée)$ as opposed to $O(\#portes \times \#vecteurs d'entré \times \#sorties)$. Cependant, j'ai choisi cette méthode pour simplifier la fonction d'évaluation.

```
Data: circuit logique à tester
Result: fiabilités pour chaque sortie
for chaque porte dans le circuit do
|  représenter porte dans le circuit défectueux;
end
for chaque entrée du circuit défectueux do
|  générer une suite non-Bernoulli;
end
for chaque sortie dans le circuit do
|  for chaque vecteur d'entrée do
|  |  if les sorties sont les mêmes pour chaque circuit then
|  |  |  ajouter 1/n à la fiabilité de sortie ;
|  |  end
|  end
end
```

Algorithm 1: Mon pseudocode mise en œuvre de la solution proposée

3.3 Détails de travail

Le code Python a été inclus ici à l'Annexe A. Il peut également être trouvé à ma page Github, l'URL pour laquelle a été donnée dans la section 1, avec présentation de diapositives et un script tous les deux en anglais et en français.

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.