Sistema especialista para diagnósticos em circuitos lógicos defeituosos

Jordão Memória, José Everardo Bessa Maia

[1]Mestrado Acadêmico em Ciências da Computação (MACC)
Universidade Estadual do Ceará (UECE)

jordaomemoria@gmail.com, jose.maia@uece.br

Resumo. *Este trabalho implementa uma série de funcionalidades para realizar diagnósticos em circuitos lógicos defeituosos. Como linguagem de descrição do circuito é utilizado o formato JSON bastate popular e intuitivo. A partir do formato descrevendo o circuito, inicialmente é fornecida a expressão lógica e tabela verdade do mesmo considerando que não há defeitos no circuito. Um circuito defeituoso é fornecido e em seguida são testadas todas as possibilidades de defeitos considerando até 3 defeitos simultâneos em cada entrada ou saída de porta ou do circuito. Isso é computado obtendo a combinação de 1, 2 e 3 elementos da lista de saídas e entradas do circuito. Por fim, é mostrado o diagnóstico com todas as possibilidade de até 3 defeitos no circuito.*

1. Introdução

Este trabalho foi desenvolvido com o intuito de entender na prática como funciona um sistema especialista. A utilização de circuitos lógicos digitais [Taub 1984] foi escolhida para demonstrar a eficiência de um sistema especialista quando aplicado a diferentes áreas, pois portas lógicas abstraem diversos tipos de problemas do mundo real [Hill et al. 1981]. Os testes deste trabalho foram feitos utilizando 4 circuitos e 1 último maior composto por esses 4.

Inicialmente será evidenciada a formulação do problema. Depois será fornecida uma explicação sobre o padrão da linguagem de descrição dos circuitos. Em seguida será explicado como funciona o algoritmo de geração de expressão lógica do circuito. Na ordem serão abordadas as funções referentes a geração de tabelas verdade. Ademais serão explicadas as funções referentes as falhas simuladas e ao diagnóstico. Por último será mostrado como os testes e resultados foram feitos.

2. Formulação do problema

Problemas de mundo simplificado e de mundo real podem ser definidos a partir de 6 descrições: estados; estado inicial; ações; modelo de transição; teste de objetivo; e custo de caminho. A seguir serão mostradas cada uma das descrições e todos os conceitos ainda não comentados no decorrer das subseções desta seção serão explicados ao longo desse trabalho.

2.1. Estados

Um estado é representado por uma possibilidade de combinação dos defeitos. Vamos supor um circuito apenas com uma porta AND de duas entradas. O circuito possui entradas

de nomes A e B; e saída de nome S. Nesse caso existem 6 ligações que podem dar defeitos: A, B, S, duas entradas da porta e a saída da porta. Para 1 falha simultânea existem $2 \times 6 = 12$ possibilidades, e logo para um circuito de n ligações temos $2n$ possibilidades. Ainda deve-se contabilizar possibilidades com 2 e 3 falhas simultâneas. Nesse caso, utiliza-se a princípio a formula da combinação com k igual a 2 e 3:

$$C_{(n,k)} = \frac{n!}{k! \cdot (n-k)!} \tag{1}$$

Considerando duas falhas simultâneas, para cada ligação haverá quatro possibilidades gerada com dois defeitos em uma única ligação: ([L preso em 0 e L preso em 0][L preso em 0 e L preso em 1][L preso em 1 e L preso em 0][L preso em 1 e L preso em 1]). Logo esse caso deve ser subtraído do número de estados, tendo para um circuito com n ligações:

$$C_{(n,2)} = \frac{n!}{2! \cdot (n-2)!} - 4n \tag{2}$$

Com 3 falhas simultâneas o problema se repete, mas são 12 possibilidades proibidas para cada ligação ao invés de 4. Temos então como número total de estados do problema baseado no número de ligações n do circuito:

$$E(n) = 2n + \frac{n!}{2! \cdot (n-2)!} - 4n + \frac{n!}{3! \cdot (n-3)!} - 12n \tag{3}$$

2.2. Estado Inicial

Qualquer estado pode ser designado como o estado inicial logo o estado inicial é a primeira possibilidade de combinação dos defeitos gerada.

2.3. Ações

Nesse ambiente, a única ação usada para qualquer estado é gerar a tabela verdade defeituosa e comparar com a tabela verdade defeituosa escolhida para representar o defeito real. Caso sejam iguais o conjunto de falhas é adicionado na lista de diagnósticos. Caso contrário um novo estado é testado.

2.4. Modelo de transição

Esse sistema aborda um problema determinístico e possui apenas 1 ação para qualquer estado, o que simplifica bastante o modelo de transição. Depois que uma ação é executada, um novo estado é testado. A figura 1 mostra uma representação do modelo de transição.

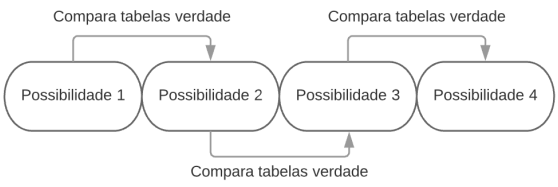

Figura 1. Representação do modelo de transição

2.5. Teste de objetivo

O teste de objetivos, como já comentado, trata-se de comparar a tabela verdade defeituosa referente ao estado e a tabela verdade defeituosa escolhida para representar o defeito real para descobrir se elas são iguais e assim gerar uma solução ou diagnóstico.

2.6. Custo de caminho

Cada estado testado custa 1 e, assim, o custo do caminho é o número de estados testados. Nota-se então que o custo do caminho é diretamente proporcional com o tamanho do circuito e para um mesmo circuito, o custo do caminho será sempre o mesmo, já que as combinações são sempre as mesmas e todas são visitadas.

3. Linguagem de descrição dos circuitos

Antes de mais nada é importante entender como a linguagem de descrição de circuitos usando o JSON funciona, pois a partir dela será possível utilizar o sistema com qualquer circuito [JSON 2013] .

Um circuito funciona como uma estrutura que possui atributos. Os atributos de um circuito são: uma lista de entradas; uma lista de saídas; e uma lista de portas. Uma porta é uma estrutura que possui: uma lista de entradas; uma única saída; um nome; e um tipo. Os tipos podem ser: AND; OR; NOT; NAND; NOR; e XOR. Entradas e saídas são estruturas idênticas e possuem os mesmos atributos que são: nome; valor; e defeito. O atributo “defeito”possui o valor “null”caso a estrutura esteja funcionando corretamente, “1”caso a estrutura esteja presa em 1 e 0 caso a estrutura esteja presa em 0. As ligações do circuito são representadas da seguinte maneira: o valor de cada entrada ou saída conterá o nome da entrada ou saída que se liga a ele.

O formato JSON opera da seguinte maneira: uma estrutura é representada por chaves ({}); uma lista é representada por colchetes ([]) e um atributo é representado da seguinte forma: ”nomeDoAtributo”:valorDoAtributo. O valor do atributo aceita vários tipos de dados mas utilizaremos apenas “null”, “String”, “int”, listas e estruturas. A vírgula é utilizada para separar estruturas, listas e atributos.

A titulo de exemplo vamos supor um circuito com apenas uma porta lógica AND com entradas de nomes A e B, e saída de nome S. A figura 2 mostra a representação em JSON.

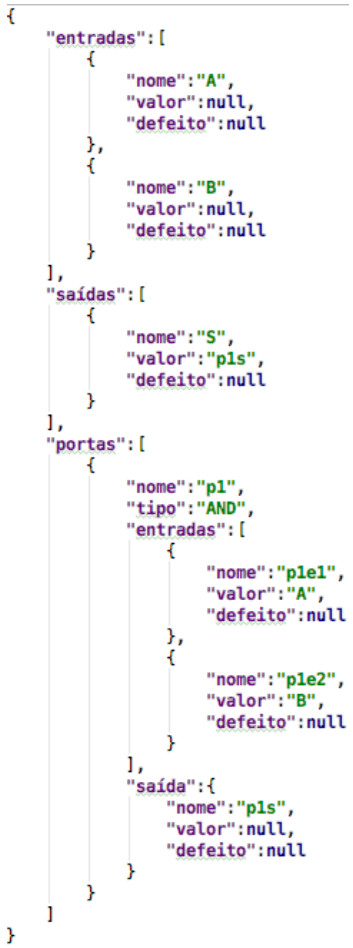

Figura 2. JSON representando circuito com apenas uma porta lógica AND com entradas de nomes A e B, e saída de nome S

4. Expressão lógica do circuito

O arquivo "logicFunctionGenerator.py"é o responsável por gerar a expressão lógica do circuito e o mesmo possui 2 funções: "generateLogicFunctions(circuit)"e "findLogic-Function(resp,circuit, porta)". Boa parte deste sistema utiliza a biblioteca numpy para manipular os números, arrays e utilizar métodos numéricos [Oliphant 2015].

A função "generateLogicFunctions(circuit)"recebe o circuito em formato de classe e percorre todas as saídas do circuito. Para cada saída do circuito, todas as entradas de circuito e saídas das portas são verificadas no intuito de encontrar qual se liga a saída de circuito em questão. Caso a ligação encontrada seja uma entrada de circuito, então a iteração termina com uma expressão do tipo "S = A", "Cout = Bin"ou de forma geral "nomeDaSaida = nomeDaEntrada". Caso a ligação encontrada seja uma saída de porta, a função "findLogicFunction(resp,circuit, porta)"é chamada contendo uma resposta de string vazia, o circuito e a porta encontrada. No fim de todas as iterações o resultado é uma lista de strings contendo a expressão lógica de cada saída. A figura 3 mostra a função "generateLogicFunctions(circuit)".

```
def generateLogicFunctions(circuit):
    resps = []
    for saída in circuit.saídas:
        for entrada in circuit.entradas:
            if saída.valor == entrada.nome:
                return saída.nome + " = " + entrada.nome
        for porta in circuit.portas:
            if saída.valor == porta.saída.nome:
                resps.append(saída.nome+" = "+findLogicFunction("", circuit, porta))
    return resps
```

Figura 3. Funcão generateLogicFunctions(circuit)

A função “findLogicFunction(resp,circuit, porta)”retorna o valor lógico da porta informada. O atributo “resp”é um acumulador que à medida que essa função vai sendo chamada recursivamente, “resp”vai compondo a expressão lógica final. Essa função opera da seguinte forma: todas as entradas da porta informada são percorridas. Para cada entrada da porta existem duas possibilidades: ou essa está ligada a uma entrada do circuito ou está ligada a uma saída de outra porta. Caso esteja ligada a uma entrada do circuito, o atributo “resp”é atualizado adicionando no fim o nome da entrada e uma vírgula. Caso esteja ligada a uma saída de porta, então essa função é chamada recursivamente com a porta da saída encontrada e o atributo “resp”atual. Quando todas as entradas são percorridas, “resp”é atualizado fechando um parêntese e assim a expressão lógica está finalizada. A figura 4 mostra a funcão descrita.

```
def findLogicFunction(resp,circuit, porta):
    list = resp + porta.tipo+"("
    for entradaPorta in porta.entradas:
        for entradaCircuito in circuit.entradas:
            if entradaPorta.valor == entradaCircuito.nome:
                list += entradaCircuito.nome + ","
    for entradaPorta in porta.entradas:
        list2 = []
        for portaCircuito in circuit.portas:
            if entradaPorta.valor == portaCircuito.saída.nome:
                list2.append(findLogicFunction("", circuit, portaCircuito)+",")
        for r in list2:
            list += r
    list += ")"
    list = list.replace(",)", ")")
    return list
```

Figura 4. Funcão generateLogicFunctions(circuit)

Agora que o código foi explicado, é interessante fornecer uma explicação sobre como funciona a sintaxe da expressão lógica. Nesse caso é muito mais fácil explicar a partir de exemplos. S = AND(A,B) é um exemplo de um circuito contendo apenas uma porta AND, com entradas de nomes A e B e saída de nome S. Um exemplo mais complexo pode ser visto da figura 5, que representa o circuito 1 e mostra as seguintes características: o circuito 1 possui duas entradas (S e Cout); S está ligado a uma porta do tipo XOR, que por sua vez possui duas entradas, uma entrada ligada em Cin e outra ligada em outra porta XOR, que por sua vez possui duas entradas ligadas em entradas do circuito cujos nomes são A e B. Cout pode ser interpretada da mesma forma.

```
S = XOR(Cin,XOR(A,B))
Cout = OR(AND(Cin,XOR(A,B)),AND(B,A))
```

Figura 5. Expressão lógica do circuito 1

5. Tabela verdade do circuito

O arquivo “tableGenerator.py”é o responsável por gerar a tabela verdade do circuito e o mesmo possui 4 funções principais: “simulateLogic(resps, typePort)”, “getResult(circuit, porta)”, “getResults(circuit)”e “generateTable(circuit)”.

A função mais trivial é a “simulateLogic(resps, typePort)”, que recebe uma lista de 0 e 1; o tipo da porta (AND, OR, NOT, NAND, NOR e XOR) e retorna o resultado da lógico da porta.

A função “getResults(circuit)”é muito similar a função “generateLogicFunctions(circuit)”, que já foi explicada. A única diferença entre essa e aquela é que essa retorna uma lista de strings contendo a expressão lógica e aquela retorna uma lista de 0 e 1, para cada saída do circuito, mas ambas percorrem todas as saídas do circuito.

A função “getResult(circuit, porta)”é muito similar a função “findLogicFunction(resp,circuit, porta)”, que já foi explicada. A única diferença entre essa e aquela é que essa retorna uma string contendo a expressão lógica da porta e aquela retorna 0 ou 1 como resultado lógico da porta.

```
def generateTable(circuit):
    lenEntradas = len(circuit["entradas"])
    sequence = ["".join(seq) for seq in itertools.product("01", repeat=lenEntradas)]
    table = []
    cols = []
    i = 0
    while i < lenEntradas:
        cols.append(circuit["entradas"][i]["nome"])
        i += 1
    for s in circuit["saídas"]:
        cols.append(s["nome"])
    for s in sequence:
        line = []
        i = 0
        while i < lenEntradas:
            if circuit["entradas"][i]["defeito"] == None:
                circuit["entradas"][i]["valor"] = int(s[i])
            line.append(int(s[i]))
            i += 1
        results = getResults(circuit)
        for r in results:
            line.append(r)
        table.append(line)
    df = DataFrame(table,columns=cols)
    return df
```

Figura 6. Funcão generateTable(circuit)

A função “generateTable(circuit)”cria uma lista contendo a combinação binária referente ao número de entradas do circuito. Por exemplo, caso o circuito possua apenas 2 entradas, então a combinação binária gerada será (00, 01, 10, 11), e assim por diante aumentando a lista conforme cresce o número de entradas. Com a lista criada, a mesma é percorrida. Para cada iteração, cada algarismo do elemento é atribuído a uma entrada e

assim o atributo “valor”de cada entrada é atualizado com 0 ou 1. Feito isso, a função “getResults(circuit)”é chamada retornando a lista dos resultados das saídas. A cada iteração uma linha da tabela verdade é criada. A figura 6 mostra a função descrita.

6. Diagnóstico e geração de faltas

O arquivo “faultsGenerator.py”é o responsável por tratar de todas as funções referentes a defeitos e diagnósticos. Antes de explicar as funções deste arquivo é necessário enfatizar que ele possui uma classe chamada “Fault”. Essa classe possui 2 atributos: “nome”e “defeito”. Ao atributo “nome”deve ser atribuído o nome da entrada ou saída onde o defeito irá existir. Ao atributo “defeito”deve ser atribuído 0 para a entrada ou saída presa em 0, e 1 para a entrada ou saída presa em 1. A figura 7 mostra a classe descrita.

```
class Fault:
    nome = None
    defeito = None

    def __init__(self, nome, defeito):
        self.nome = nome
        self.defeito = defeito
```

Figura 7. Classe Fault

Com a classe explicada, pode-se dar início ao esclarecimento da principal função deste trabalho: “generateListOfFaults(circuit)”. O primeiro passo para gerar uma lista com todas as possibilidades de combinações de falhas é possuir uma lista com todas as falhas possíveis para então fazer a combinação. As primeiras linhas de código percorrem todas as entradas do circuito, saídas do circuito, entradas das porta, saída das portas e o nome de cada uma é adicionado em uma lista. Confira o trecho de código referente a explicação acima na figura 8.

```
l = []
faults = []
for e in circuit.entradas:
    l.append(e.nome)
for s in circuit.saídas:
    l.append(s.nome)
for p in circuit.portas:
    for e in p.entradas:
        l.append(e.nome)
    l.append(p.saída.nome)
for f in l:
    fault1 = Fault(f, 1)
    fault0 = Fault(f, 0)
    faults.append(fault0)
    faults.append(fault1)
```

Figura 8. Trecho de código inicial da funcão generateListOfFaults(circuit)

Em seguida, a lista é percorrida. Para cada iteração é criado dois objeto da classe “Fault”, um com o defeito 0 e outro com o defeito 1. Isso pode ser observado ainda na figura 8. Com a lista de defeitos preenchida as combinações podem ser feitas com o

module “itertools”nativo da linguagem Python [Rossum 1995]. Combinações de 1, 2 e 3 elementos são geradas e podem ser observadas na figura 9.

```
fault1 = list(combinations(faults, 1))
fault2 = list(combinations(faults, 2))
fault3 = list(combinations(faults, 3))
```

Figura 9. Combinações de 1, 2 e 3 elementos

Existem detalhes bastante sutis que devem ser enfatizados a essa altura do trabalho. É admissível que a combinação de apenas 1 elemento nos retornará a mesma lista de falhas. As complicações aparecem então para combinações de 2 e 3 elementos. O primeiro desafio a se vencer é evitar a repetição de defeitos apenas com a ordem invertida. Por exemplo, entrada 1 da porta 1 e entrada 2 da porta 3 pode ser considerado uma possibilidade de diagnóstico. Outra possibilidade seria entrada 2 da porta 3 e entrada 1 da porta 1. Perceba que esses diagnósticos são idênticos, apenas com a ordem invertida. Esse tipo de repetição aconteceria se fosse obtido a permutação dos defeitos. Felizmente está sendo usada uma função que retorna a combinação, ou sejam, ordens de elementos diferente estão dentro do mesmo diagnóstico. Como segundo e ultimo desafio a se ultrapassar, temos o seguinte problema: uma mesma entrada ou saída não pode apresentar mais de um defeito, isto é, não é possível que a entrada 1 da porta 1 esteja presa em 0 e presa em 1 ao mesmo tempo. Apenas com a combinação mostrada na figura 9 esse tipo de caso acontece, logo deve-se percorrer a lista de defeitos gerada e remover os casos que uma mesma entrada ou saída apresente mais de um defeito. A figura 10 mostra o trecho de código realizado para cumprir o que foi dito no último período.

```
toRemove = []
for i in range(len(fault2)):
    f0, f1 = fault2[i]
    if f0.nome == f1.nome:
        toRemove.append(fault2[i])
for f in toRemove:
    fault2.remove(f)
toRemove = []
for i in range(len(fault3)):
    f0, f1, f2 = fault3[i]
    if f0.nome == f1.nome or f0.nome == f2.nome or f1.nome == f2.nome:
        toRemove.append(fault3[i])
for f in toRemove:
    fault3.remove(f)
```

Figura 10. Laços que removem entradas ou saídas com mais de um defeito

Feito isso as 3 listas são retornadas contendo combinações de 1, 2 e 3 defeitos simultâneos e a função “generateListOfFaults(circuit)”termina. Continuando a explicação das principais funções temos a “updateFault(circuit, fault)”que simplesmente aplica o defeito recebido ao circuito recebido e retorna o circuito atalizado com o defeito. Por exemplo, supondo que uma entrada de uma porta possuía o valor “p5s”, mostrando assim que o seu valor estava conectado a saída de nome “p5s”, como foi descrito na seção 3. Depois que a função “updateFault(circuit, fault)”for executada, o valor da suposta porta

será atribuído a 0 ou 1 conforme o defeito. Esse mecanismo funciona muito bem com todo código implementado. A figura 11 mostra a função descrita.

```
def updateFault(circuit, fault):
    for entrada in circuit["entradas"]:
        if fault.nome == entrada["nome"]:
            entrada["defeito"] = fault.defeito
            entrada["valor"] = fault.defeito
            return circuit
    for saida in circuit["saídas"]:
        if fault.nome == saida["nome"]:
            saida["defeito"] = fault.defeito
            saida["valor"] = fault.defeito
            return circuit
    for porta in circuit["portas"]:
        for entrada in porta["entradas"]:
            if fault.nome == entrada["nome"]:
                entrada["defeito"] = fault.defeito
                entrada["valor"] = fault.defeito
                return circuit
        if fault.nome == porta["saída"]["nome"]:
            porta["saída"]["valor"] = fault.defeito
            porta["saída"]["defeito"] = fault.defeito
            return circuit
```

Figura 11. Função updateFault(circuit, fault)

Como forma de representar as tabelas-verdade dos circuitos foi utilizado o tipo DataFrame do pacote pandas existente na linguagem Python. Essa estrutura funciona como uma matriz contendo o nome das colunas e diversos métodos que facilitam a análise de dados. Como última função a ser comentada, temos a "checkFaults(circuitName, faults, dfDefeito)". A mesma recebe o nome do circuito, a lista de falhas a ser testada e o DataFrame contendo a tabela verdade do circuito defeituoso. A lista de falhas é composta por tuplas. Cada tupla possui um conjunto de defeitos. Para cada falha, ou seja, para cada conjunto de defeitos que representam uma possibilidade de diagnóstico, um novo circuito é instanciado sem defeito nenhum e em seguida o conjunto de defeitos é inserido nesse circuito. Depois uma tabela verdade é gerada a partir desse circuito defeituoso. Então as duas tabelas são comparadas: a que foi recebida como parâmetro de entrada da função e a que foi gerada a partir do conjunto de defeitos. Caso sejam iguais, a tupla é inserida em uma lista contendo todos os diagnósticos. Caso contrário a próxima iteração começa. No fim de todas as iterações a lista contendo os diagnósticos é retornada. A figura 12 mostra a função descrita.

```
def checkFaults(circuitName, faults, dfDefeito):
    diagList = []
    for faultTuple in faults:
        circuit = getEditableJson(circuitName)
        for f in faultTuple:
            circuit = updateFault(circuit, f)
        df = generateTable(circuit)
        if df.equals(dfDefeito):
            diagList.append(faultTuple)
    return diagList
```

Figura 12. Função checkFaults(circuitName, faults, dfDefeito)

7. Resultados

Como forma de testar o sistema foi utilizada a seguinte estratégia. Foram utilizados 4 circuitos relativamente pequenos contendo de 5 a 7 portas cada. Além desses foi utilizado um quinto circuito composto pela junção dos 4. Esse circuito é consideravelmente maior pois possui 24 portas.

Todos os passos a seguir foram feito para cada circuito citado. O circuito é carregado sem defeitos e sua tabela verdade e expressão lógica são geradas. Em seguida sua expressão lógica e sua tabela verdade são carregadas em um arquivo contendo os resultados dos testes do circuito. No próximo passo, as possibilidades de falhas são geradas com até 3 defeitos simultâneos, gerando 3 listas: faults1, faults2 e faults3. É iniciada então uma iteração com as falhas de 1 defeito, falhas de 2 defeitos e falhas de 3 defeitos. Dentro da citada iteração são executadas 10 repetições do procedimento a seguir.

Uma falha é sorteada para gerar o circuito defeituoso. A partir dessa falha a tabela verdade defeituosa é gerada. A tabela verdade defeituosa é escrita no arquivo de resultados do circuito. Em seguida utilizando as três listas citadas acima e a função "checkFaults(circuitName, faults, dfDefeito)", 3 listas de diagnósticos são geradas e escritas no arquivo de resultados do circuito. É feito um teste para verificar se o defeito sorteado está entre os diagnósticos, que é escrito no arquivo de resultados do circuito. Com isso a iteração termina e o processo se repete. No fim de todas as iterações o arquivo de resultados está completo. A figura 13 mostra o código do que foi descrito.

```
def main():
    circuitList = ["circuito1","circuito2", "circuito3", "circuito4","circuito5"]
    for circuitName in circuitList:
        circuit = getEditableJson(circuitName)
        df = generateTable(circuit)
        circuit = readJson(circuitName)
        expression = generateLogicFunctions(circuit)
        write(circuitName, expression, df)
        faults1, faults2, faults3 = generateListOfFaults(circuit)
        faultsList = [faults1, faults2, faults3]
        for faults in faultsList:
            for i in range(10):
                errorNumber = randint(0, len(faults)-1)
                faultTuple = faults[errorNumber]
                circuit = getEditableJson(circuitName)
                for f in faultTuple:
                    print(f.nome, f.defeito)
                    circuit = updateFault(circuit,f)
                dfDefeito = generateTable(circuit)
                writeFault(circuitName, i, faultTuple, dfDefeito)
                diagList1 = checkFaults(circuitName, faults1, dfDefeito)
                diagList2 = checkFaults(circuitName, faults2, dfDefeito)
                diagList3 = checkFaults(circuitName, faults3, dfDefeito)
                writeDiag(circuitName, len(diagList1), len(diagList2), len(diagList3))
```

Figura 13. Função main

Os resultados gerais podem ser considerados bem sucedidos pois para todos os testes o defeito real está dentro das falhas diagnosticadas. A figura 14 mostra uma parte do arquivo de resultado do circuito 1, que será explicada a seguir.

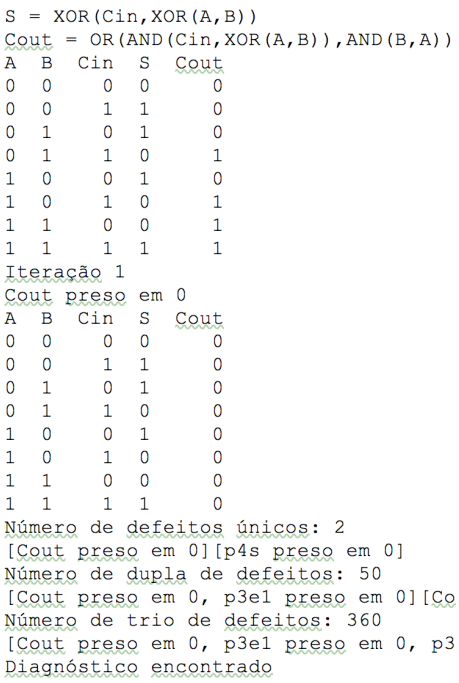

Figura 14. Resultado da primeira iteração do circuito 1

A figura 14 inicialmente mostra as expressões lógicas das saídas do circuito, e em seguida mostra a tabela verdade do circuito funcionando. Para o circuito indicado, são gerados 10 defeitos com apenas uma falha, 10 defeitos com 2 falhas e 10 defeitos com 3 falhas. A “Iteração 1”mostrada na figura 14 indica que o primeiro defeito de uma falha foi gerado.

Depois “Cout preso em 0”indica que a entrada Cout está presa em 0 e a tabela verdade do circuito defeituoso é mostrada. Esse sistema considera a possibilidade de defeitos também nas entradas e saídas do circuito como mostrado no exemplo. Por fim, todo o algoritmo explicado nesse trabalho é executado para descobrir as combinações de falhas que geraram a mesma tabela verdade defeituosa.

Na figura 14 pode-se notar que foram encontrados 2 defeitos de uma falha, 50 defeitos de 2 falhas e 360 defeitos de 3 falhas. A última linha da figura 14 mostra que dentre os 2 diagnósticos de uma falha encontrados nessa iteração, um deles possui a falha sorteada, ou seja, Cout preso em 0.

A figura 14 é utilizada como exemplo de como se interpretar os arquivos de resultados. Abaixo de cada linha contendo o número de defeitos encontrados, há uma listagem com todos os nomes das falhas que compõem os defeitos.

Para facilitar o entendimento, segue-se o padrão do tipo pXeY, onde pX é o nome da porta e eY o número da ordem de cima para baixo da entrada. “p3e1”, por exemplo, significa entrada 1 da porta 3. Esse padrão é escolhido na criação do JSON referente ao circuito. Todos os arquivos de resultados dos circuitos estão disponíveis para consulta junto com esse trabalho.

	Circuito 1 (5 portas)	Circuito 2 (7 portas)	Circuito 3 (6 portas)	Circuito 4 (6 portas)	Circuito 5 (24 portas)
Diagnósticos de 1 defeito	0.049 segs	0.127 segs	0.136 segs	0.125 segs	-
Diagnósticos de 2 defeitos	0.904 segs	3.161 segs	3.119 segs	2.707 segs	-
Diagnóstico de 3 defeitos	10.463 segs	46.613 segs	45.114 segs	38.062 segs	-

Figura 15. Tempo de CPU utilizado para se gerar a lista de diagnósticos de 1, 2 e 3 falhas simultâneas

Um teste de tempo computacional foi gerado a fim de se verificar a eficiência do trabalho. Os testes foram realizados em uma máquina com o processador 2,6 GHz Intel Core i5 e memória RAM 8 GB 1600 MHz DDR3. A figura 15 mostra uma tabela contendo o tempo de CPU em segundos utilizado para se gerar a lista de diagnósticos de 1, 2 e 3 falhas simultâneas. Cada número mostrado nas células representam uma média das 30 iterações do teste principal.

Pode-se notar que quanto mais combinações de defeitos simultâneos são testadas, mas tempo é requerido da CPU já que mais possibilidades são geradas. Pode-se notar pela figura 15, por exemplo que no circuito 1 foi necessário em média 10.463 segundos para que todas as possibilidades envolvendo 3 falhas simultâneas seja testadas e assim os diagnósticos sejam encontrados.

Como último cometário é relevante dizer que mesmo com aproximadamente 30 horas de processamento o sistema não foi capaz de gerar diagnósticos para o circuito 5 considerando que esse possui 24 portas e o número de combinações de falhas se torna demasiadamente grande.

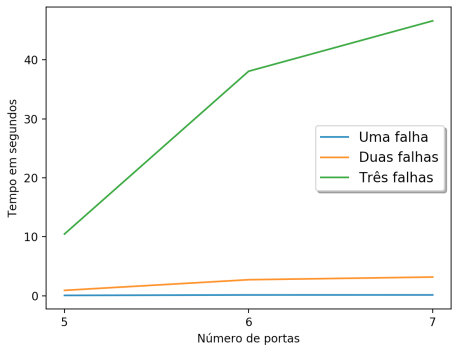

Figura 16. Gráfico do tempo em segundos utilizado para gerar os diagnósticos em função do número de portas

Todos os testes e resultados apontam que quanto maior o número de portas mais tempo será necessário para se obter os diagnósticos através do sistema contido neste trabalho. Um gráfico comprovando isso é mostrado na figura 16.

A figura 17 mostra um grande resumo dos resultados bastante específicos por iteração, circuito e número de falhas reais sorteadas.

Circuito 1												
Iteração	Uma falha defeituosa gerada				Duas falhas defeituosas geradas				Três falhas defeituosas geradas			
	Número de diagnósticos encontrados			Tempo computacional em segundos	Número de diagnósticos encontrados			Tempo computacional em segundos	Número de diagnósticos encontrados			Tempo computacional em segundos
	Uma falha	Duas falhas	Três falhas		Uma falha	Duas falhas	Três falhas		Uma falha	Duas falhas	Três falhas	
1	1	5	8	11.212	1	10	35	12.359	6	84	516	12.647
2	1	10	35	12.902	0	13	187	13.416	0	12	450	13.613
3	1	0	0	12.350	0	12	181	13.483	0	4	254	13.725
4	4	13	16	13.417	0	4	13	13.791	0	13	361	14.049
5	2	50	360	12.737	2	50	360	13.606	0	4	108	13.322
6	1	0	0	13.924	0	8	125	12.755	0	12	250	13.407
7	1	0	0	14.685	2	12	16	13.325	1	16	91	11.132
8	14	13	16	12.753	6	84	516	12.830	0	13	345	13.422
9	1	5	8	14.044	0	9	35	12.171	0	4	193	12.815
10	6	84	516	13.462	0	15	242	13.927	0	1	36	13.841
Circuito 2												
Iteração	Uma falha defeituosa gerada				Duas falhas defeituosas geradas				Três falhas defeituosas geradas			
	Número de diagnósticos encontrados			Tempo computacional em segundos	Número de diagnósticos encontrados			Tempo computacional em segundos	Número de diagnósticos encontrados			Tempo computacional em segundos
	Uma falha	Duas falhas	Três falhas		Uma falha	Duas falhas	Três falhas		Uma falha	Duas falhas	Três falhas	
1	3	0	0	55.121	0	8	170	55.091	0	7	39	52.175
2	3	0	0	54.403	0	7	59	64.327	0	47	1042	52.471
3	3	6	4	53.076	0	47	1042	57.431	0	0	104	51.019
4	4	81	465	51.370	0	19	994	51.225	0	32	1626	51.101
5	3	6	4	53.679	4	47	216	52.513	0	28	416	50.387
6	1	18	95	52.419	0	18	917	52.008	0	19	994	55.007
7	1	3	0	50.766	0	28	659	50.867	0	52	620	49.392
8	7	39	36	49.174	0	2	2	50.199	0	9	178	49.225
9	5	22	42	48.853	0	7	6	51.995	0	19	994	48.922
10	4	81	465	49.891	0	18	917	50.281	0	32	1626	48.800
Circuito 3												
Iteração	Uma falha defeituosa gerada				Duas falhas defeituosas geradas				Três falhas defeituosas geradas			
	Número de diagnósticos encontrados			Tempo computacional em segundos	Número de diagnósticos encontrados			Tempo computacional em segundos	Número de diagnósticos encontrados			Tempo computacional em segundos
	Uma falha	Duas falhas	Três falhas		Uma falha	Duas falhas	Três falhas		Uma falha	Duas falhas	Três falhas	
1	8	37	90	49.155	1	38	612	46.268	0	0	30	54.285
2	5	59	365	46.855	1	32	460	46.627	10	219	2393	55.613
3	1	0	0	48.172	2	156	3782	59.842	1	8	43	53.154
4	10	219	2393	47.860	5	226	4889	49.654	5	226	4889	50.296
5	1	32	460	46.441	10	219	2393	47.800	1	10	31	50.249
6	5	226	4889	47.220	5	226	4889	49.062	1	38	612	53.344
7	1	8	43	48.213	2	156	3782	47.750	0	2	20	53.967
8	1	8	37	50.260	2	156	3782	47.336	5	226	4889	55.632
9	1	8	37	46.213	4	47	277	46.669	10	219	2393	52.402
10	10	219	2393	45.066	1	32	460	50.273	0	20	466	52.448
Circuito 4												
Iteração	Uma falha defeituosa gerada				Duas falhas defeituosas geradas				Três falhas defeituosas geradas			
	Número de diagnósticos encontrados			Tempo computacional em segundos	Número de diagnósticos encontrados			Tempo computacional em segundos	Número de diagnósticos encontrados			Tempo computacional em segundos
	Uma falha	Duas falhas	Três falhas		Uma falha	Duas falhas	Três falhas		Uma falha	Duas falhas	Três falhas	
1	4	18	32	42.055	0	35	444	48.809	0	0	180	43.918
2	2	2	0	44.591	0	21	147	47.953	0	10	104	48.148
3	1	50	781	46.532	0	25	423	45.031	0	11	222	50.432
4	4	28	76	40.853	4	28	76	40.070	0	0	76	45.829
5	2	31	166	42.031	0	2	40	46.977	0	32	444	44.738
6	2	2	0	41.076	4	18	32	45.872	0	14	130	49.339
7	5	54	238	39.266	2	14	46	45.174	0	10	216	45.616
8	4	28	76	40.326	0	43	901	46.703	0	0	20	43.884
9	5	54	238	41.783	0	7	79	42.266	0	0	70	44.306
10	7	58	196	46.305	5	54	238	41.695	0	12	644	44.020

Figura 17. Tabela fornecendo resultados específicos de cada iteração, circuito e número de falhas geradas

8. Modo usuário

Caso se deseje utilizar o sistema para diagnosticar circuitos reais, basta utilizar o modo usuário. Para utilizá-lo deve-se criar um arquivo JSON descrevendo o circuito a ser diagnosticado. Com circuito descrito deve-se chamar a função “createTableToConfigure”no arquivo ”main.py”passando o nome do arquivo JSON com a descrição do circuito. Essa função irá criar um arquivo chamado “TabelaDefeituosa”. Esse arquivo contém a tabela verdade funcionando sem defeitos referente ao circuito informado. A tabela deve ser alterada pra ficar de acordo com os resultados defeituosos do circuito real. Com a tabela modificada deve-se executar a função “userMode”passando nome do circuito. Ao fim desse processo será gerado um novo arquivo chamado “TabelaDefeituosaResultado”com todos os diagnósticos encontrados. A figura 18 mostra um exemplo de arquivo de resultado do modo usuário.

```
S = XOR(Cin,XOR(A,B))
Cout = OR(AND(Cin,XOR(A,B)),AND(B,A))
A  B  Cin  S  Cout
0  0    0  0     0
0  0    1  1     0
0  1    0  1     0
0  1    1  0     1
1  0    0  1     0
1  0    1  0     1
1  1    0  0     1
1  1    1  1     1
Tabela Defeituosa
A  B  Cin  S  Cout
0  0    0  0     1
0  0    1  1     1
0  1    0  1     1
0  1    1  0     1
1  0    0  1     1
1  0    1  0     1
1  1    0  0     1
1  1    1  1     1
Número de defeitos únicos: 6
[Cout preso em 1][p3s preso em 1][p4e
Número de dupla de defeitos: 84
[Cout preso em 1, p3e1 preso em 0][Co
Número de trio de defeitos: 516
[Cout preso em 1, p3e1 preso em 0, p3
```

Figura 18. Arquivo de resultados do modo usuário

9. Conclusão

Com o desenvolvimento desse sistema foi possível entender como funciona a implementação de um sistema especialista. Como conclusão revelante também pode-se notar que visitar todas as possibilidades de defeitos não é uma meta razoável para circuitos muito grandes.

Referências

Hill, F. J., Peterson, G. R., and Hill, F. J. (1981). *Introduction to switching theory and logical design*. Wiley New York.

JSON (2013). The JSON Data Interchange Format. Technical Report Standard ECMA-404 1st Edition / October 2013, ECMA.

Oliphant, T. E. (2015). *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition.

Rossum, G. (1995). Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands.

Taub, H. (1984). *Circuitos digitais e microprocessadores*. McGraw-Hill.