# APÉNDICE C: Código del programa.

A continuación se presentaran las principales secciones de codigo del programa:

INICIO DEL PROGRAMA PRINCIPAL

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

#

# simanaly.py

#

# Copyright 2010 Facundo-std <Facundo-std@FACUNDO-XPS>

#

# This program is free software; you can redistribute it and/or modify

# it under the terms of the GNU General Public License as published by

# the Free Software Foundation; either version 2 of the License, or

# (at your option) any later version.

#

# This program is distributed in the hope that it will be useful,

# but WITHOUT ANY WARRANTY; without even the implied warranty of

# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

# GNU General Public License for more details.

#

# You should have received a copy of the GNU General Public License

# along with this program; if not, write to the Free Software

# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,

# MA 02110-1301, USA.

""" This module is the main program """

#Mine imports

import fail\_inject

#Python imports

import sys

import decimal as dc

import time

import os

import subprocess

from mineparser import ParserMain

import re

#PyQt imports

from PyQt4 import QtCore, QtGui

#UI imports

from ui.about import Ui\_AboutDialog

from ui.info import Ui\_InfoDialog

from ui.simulation\_analyzer import Ui\_MainWindow

class AboutDialog(QtGui.QDialog, Ui\_AboutDialog):

def \_\_init\_\_(self, parent=None):

QtGui.QWidget.\_\_init\_\_(self, parent)

self.setupUi(self)

class InfoDialog(QtGui.QDialog, Ui\_InfoDialog):

def \_\_init\_\_(self, texto, parent=None):

QtGui.QWidget.\_\_init\_\_(self, parent)

self.setupUi(self)

class MainWindow(QtGui.QMainWindow, Ui\_MainWindow):

def \_\_init\_\_(self, parent=None):

QtGui.QMainWindow.\_\_init\_\_(self, parent=None)

self.setupUi(self)

self.dirmodel = QtGui.QFileSystemModel()

self.dirmodel.setRootPath(QtCore.QDir.currentPath())

self.treeView.setModel(self.dirmodel)

self.treeView.setColumnHidden(1,True)

self.treeView.setColumnHidden(2,True)

self.treeView.setColumnHidden(3,True)

self.selectmodel = QtGui.QItemSelectionModel(self.dirmodel)

# This var holds the treeView Indexes added into the list widget

self.global\_indexes = {}

# Output voltage constants

self.vth\_l = 1.0

self.vth\_h = 2.3

# Global comparators for each digital voltage

self.comps = []

# Set default output CIR source file and output dir to generated files

self.lineEdit\_5.setText("D:\\Documents\\TESIS\\fiocs\\Testing\\Flash\\flash.cir")

self.lineEdit\_6.setText("D:\\Documents\\TESIS\\fiocs\\Testing\\Flash\\Simulation Flash\\")

# This dict holds the vars and contents of the CIR file

self.cirFileTab = {'cirsrc':'','outdir':'','injfail':[],'injpins':[],

'injtypes':[]}

# RE that matchs each one of the folders for CSD files

self.field\_dir = re.compile("^.\*/(.\*)/(.\*)/(.\*)/(.\*)$")

#This list contains the failing bits

self.failed\_bits = [0,0,0,0,0,0]

# This vars stores the gui voltages

self.gui\_voltages = []

# First and End times for fails

self.firsttime = []

self.endtime = []

self.first\_t = []

def update\_status(self):

indices = self.treeView.currentIndex()

self.statusbar.showMessage(str(indices),1500)

def show\_about\_dialog(self):

dialog = AboutDialog(self)

dialog.show()

def add\_one(self,multiple=None):

if multiple:

model = multiple

else:

model = self.treeView.currentIndex()

csdstr = QtCore.QString("csd File")

namefile = self.dirmodel.fileName(model)

matchflag = QtCore.Qt.MatchFixedString

if csdstr == self.dirmodel.type(model):

if self.listWidget.findItems(namefile,matchflag) == []:

self.listWidget.addItem(namefile)

self.global\_indexes[namefile] = model

self.statusbar.showMessage("CSD file added!",500)

else:

self.statusbar.showMessage("Already added!",700)

else:

self.statusbar.showMessage("Not a CSD file!",1000)

def add\_all(self):

for singleindex in self.treeView.selectedIndexes():

print "INDEX: %s" % singleindex

self.add\_one(singleindex)

def remove\_one(self):

#self.listWidget.removeItemWidget(self.listWidget.selectedItems())

self.listWidget.takeItem(self.listWidget.currentRow())

self.statusbar.showMessage("Removed!",1000)

def remove\_all(self):

self.listWidget.clear()

def \_get\_voltage\_param\_from\_gui(self):

if self.digitalRadioButton.isChecked():

self.comps = [self.spinBox\_1.value(),

self.spinBox\_2.value(),

self.spinBox\_3.value(),

self.spinBox\_4.value(),

self.spinBox\_5.value(),

self.spinBox\_6.value()]

voltages = [str(self.d2a(self.spinBox\_1.value())),

str(self.d2a(self.spinBox\_2.value())),

str(self.d2a(self.spinBox\_3.value())),

str(self.d2a(self.spinBox\_4.value())),

str(self.d2a(self.spinBox\_5.value())),

str(self.d2a(self.spinBox\_6.value()))]

print "INFO voltages: %s" % str(voltages)

return voltages

elif self.analogRadioButton.isChecked():

# This should be readed from the gui

#self.comps = []

voltages = [str(self.doubleSpinBox\_1.value()),

str(self.doubleSpinBox\_2.value()),

str(self.doubleSpinBox\_3.value()),

str(self.doubleSpinBox\_4.value()),

str(self.doubleSpinBox\_5.value()),

str(self.doubleSpinBox\_6.value())]

print "INFO voltages analog: %s" % str(voltages)

return voltages

else:

self.statusbar.showMessage("SELECT ANALOG OR DIGITAL VOLTAGE",1500)

def analyze(self):

dialog = InfoDialog(self)

dialog.show()

self.gui\_voltages = self.int2dc(self.\_get\_voltage\_param\_from\_gui())

items\_ = self.listWidget.selectedItems()

print "Total items added to analize: %s" % str(items\_)

items\_qty = items\_.\_\_len\_\_()

if items\_qty == 0:

step = 0

else:

step = (100 / items\_qty).\_\_trunc\_\_()

valor = 0

file\_row = ''

file\_name = ''

# This loop iterates over all list widget items

for item\_ in items\_:

parser = ParserMain()

csdparsed = None

valor = valor + step

dialog.progressBar.setValue(valor)

model\_ = None

# This loop check the model index from the tree view for the item

for key\_ in self.global\_indexes:

if key\_ == item\_.text():

model\_ = self.global\_indexes[key\_]

item\_ = self.dirmodel.filePath(model\_)

# e.g. of expressions that field\_dir has to match:

# D:\test\exp\n\2sd

fields = self.field\_dir.match(str(item\_))

inode = fields.group(4).split(".")[0]

mostyp = fields.group(3)

vin = fields.group(2)

failtyp = fields.group(1)

file\_name = 'D:\\atest\\' + mostyp + '\_' + vin + '.csv'

csdparsed = parser.run(str(item\_))

self.gui\_voltages = self.autodetectlevels(csdparsed)

file\_row = file\_row + failtyp + ";" + vin + ";" + mostyp + ";" + inode + ";"

#This list contains the failing bits

self.failed\_bits = [0,0,0,0,0,0]

v\_ranges = [0,0]

self.firsttime = [0,0,0,0,0,0]

self.endtime = [0,0,0,0,0,0]

global\_detect = 0

self.first\_t = [True,True,True,True,True,True]

time\_points = csdparsed["body"]["ntime"].\_\_len\_\_()

# This loop iterates over all values

for time\_ in xrange(0,time\_points-1):

sum\_bit = 0

self.cmp\_out(csdparsed,"'V(C\_F\_D\_LSB)'",time\_,0)

self.cmp\_out(csdparsed,"'V(C\_F\_D\_2SB)'",time\_,1)

self.cmp\_out(csdparsed,"'V(C\_F\_D\_3SB)'",time\_,2)

self.cmp\_out(csdparsed,"'V(C\_F\_D\_4SB)'",time\_,3)

self.cmp\_out(csdparsed,"'V(C\_F\_D\_5SB)'",time\_,4)

self.cmp\_out(csdparsed,"'V(C\_F\_C32\_MSB)'",time\_,5)

#Check if one or more of the bits failed and save the file

for bit in self.failed\_bits:

sum\_bit += bit

if sum\_bit > 0:

global\_detect = 1

# Add fail condition

file\_row += str(global\_detect) + ";"

# Added failed\_bits to the row. e.g. [0,0,1,0,0,1]

file\_row += str(self.failed\_bits) + ";"

# Calculate the v\_ranges

v\_ranges = self.calc\_v\_ranges(csdparsed)

# Add ranges for time and voltages

file\_row += str(self.firsttime) + ";"

file\_row += str(self.endtime) + ";"

file\_row += str(v\_ranges) +"\n"

# File in wicht results will be saved

csv\_file = open(file\_name,"w")

csv\_file.write("Fail Type;Vin;MOS Type;Node;Fail?;Failed bits;Time\_start;Time\_end;Voltage\n")

# Write the values

csv\_file.write(file\_row)

csv\_file.close()

dialog.progressBar.setValue(100)

dialog.progressBar.close()

self.listWidget.clear()

def cmp\_out(self,parsed,bit\_,tim\_,n):

l\_l = dc.Decimal('-0.001')

l\_h = dc.Decimal('1.001')

h\_l = dc.Decimal('2.299')

h\_h = dc.Decimal('3.301')

\_time\_ = parsed["body"]["ntime"][tim\_].split(" ")[0]

m\_val = parsed["body"]["voltages"][bit\_][tim\_]

if self.gui\_voltages[n] == dc.Decimal("1.0"):

if not (m\_val >= l\_l and m\_val <= l\_h):

self.failed\_bits[n] = 1

self.endtime[n] = \_time\_

if self.first\_t[n]:

self.firsttime[n] = \_time\_

self.first\_t[n] = False

else:

if not (m\_val >= h\_l and m\_val <= h\_h):

self.failed\_bits[n] = 1

self.endtime[n] = \_time\_

if self.first\_t[n]:

self.firsttime[n] = \_time\_

self.first\_t[n] = False

def autodetectlevels(self,parsed):

l\_l = dc.Decimal('0.0')

l\_h = dc.Decimal('1.0')

h\_l = dc.Decimal('2.3')

h\_h = dc.Decimal('3.3')

voltage = [-1,-1,-1,-1,-1,-1]

m\_val = parsed["body"]["voltages"]["'V(C\_F\_D\_LSB)'"][0]

if (m\_val >= l\_l and m\_val <= l\_h):

voltage[0] = dc.Decimal("1.0")

if (m\_val >= h\_l and m\_val <= h\_h):

voltage[0] = dc.Decimal("2.3")

m\_val = parsed["body"]["voltages"]["'V(C\_F\_D\_2SB)'"][1]

if (m\_val >= l\_l and m\_val <= l\_h):

voltage[1] = dc.Decimal("1.0")

if (m\_val >= h\_l and m\_val <= h\_h):

voltage[1] = dc.Decimal("2.3")

m\_val = parsed["body"]["voltages"]["'V(C\_F\_D\_3SB)'"][2]

if (m\_val >= l\_l and m\_val <= l\_h):

voltage[2] = dc.Decimal("1.0")

if (m\_val >= h\_l and m\_val <= h\_h):

voltage[2] = dc.Decimal("2.3")

m\_val = parsed["body"]["voltages"]["'V(C\_F\_D\_4SB)'"][3]

if (m\_val >= l\_l and m\_val <= l\_h):

voltage[3] = dc.Decimal("1.0")

if (m\_val >= h\_l and m\_val <= h\_h):

voltage[3] = dc.Decimal("2.3")

m\_val = parsed["body"]["voltages"]["'V(C\_F\_D\_5SB)'"][4]

if (m\_val >= l\_l and m\_val <= l\_h):

voltage[4] = dc.Decimal("1.0")

if (m\_val >= h\_l and m\_val <= h\_h):

voltage[4] = dc.Decimal("2.3")

m\_val = parsed["body"]["voltages"]["'V(C\_F\_C32\_MSB)'"][5]

if (m\_val >= l\_l and m\_val <= l\_h):

voltage[5] = dc.Decimal("1.0")

if (m\_val >= h\_l and m\_val <= h\_h):

voltage[5] = dc.Decimal("2.3")

return voltage

def calc\_v\_ranges(self,parsed):

ranges = [0,0,0,0,0,0]

volts = self.gui\_voltages

nodes = [parsed["body"]["voltages"]["'V(C\_F\_D\_LSB)'"],

parsed["body"]["voltages"]["'V(C\_F\_D\_2SB)'"],

parsed["body"]["voltages"]["'V(C\_F\_D\_3SB)'"],

parsed["body"]["voltages"]["'V(C\_F\_D\_4SB)'"],

parsed["body"]["voltages"]["'V(C\_F\_D\_5SB)'"],

parsed["body"]["voltages"]["'V(C\_F\_C32\_MSB)'"]]

c = 0

for enode in nodes:

a = min(enode)

b = max(enode)

if volts[c] == 1:

if a < dc.Decimal(0):

ranges[c] = abs(float(dc.Decimal('1.0') - a))

if a > dc.Decimal(1):

ranges[c] = abs(float(dc.Decimal('1.0') - a))

else:

if b < dc.Decimal('2.3'):

ranges[c] = abs(float(dc.Decimal('2.3') - b))

if b > dc.Decimal('3.3'):

ranges[c] = abs(float(dc.Decimal('2.3') - b))

c += 1

return ranges

def auto\_adjust(self):

self.treeView.resizeColumnToContents(0)

def d2a(self,d):

if d == 0:

return self.vth\_l

else:

return self.vth\_h

def int2dc(self,i):

i = map(dc.Decimal,i)

return i

def giveId(self,l):

out = []

for i in xrange(len(l)-1,-1,-1):

if "LSB" in l[i]:

b1 = i

if "2SB" in l[i]:

b2 = i

if "3SB" in l[i]:

b3 = i

if "4SB" in l[i]:

b4 = i

if "5SB" in l[i]:

b5 = i

if "MSB" in l[i]:

b6 = i

return [b1,b2,b3,b4,b5,b6]

def compare\_(self,op,gvolt,pvolt):

local\_count = 0

#This checks that the conditions for LSB is accomplished

if op == 1:

if gvolt.compare(pvolt) == dc.Decimal('1'):

local\_count = 1

#print "Valor binario: %s" % op

#print "Threshold Value: %s" % gvolt

#print "Simulado: %s" % pvolt

#print "COMP(1 gui mayor que parsed): %s" % gvolt.compare(pvolt)

else:

if gvolt.compare(pvolt) == dc.Decimal('-1'):

local\_count = 1

#print "Valor binario: %s" % op

#print "Threshold Value: %s" % gvolt

#print "Simulado: %s" % pvolt

#print "COMP(1 gui mayor que parsed): %s" % gvolt.compare(pvolt)

#print "=========================================================="

return local\_count

def \_get\_cirfiletab\_vars(self):

""" This method returns a dict with the vars from the CIRFILES tab."""

sourceFile = ''

outputDir = ''

failToInject = []

pinsToInject = []

typesToInject = []

# Gets sourcefile and output dir fields from GUI

sourceFile = self.lineEdit\_5.text()

outputDir = str(self.lineEdit\_6.text())

# Check all checkboxes to set the fails to inject

if self.checkBox.isChecked():

failToInject.append(self.lineEdit.text())

if self.checkBox\_2.isChecked():

failToInject.append(self.lineEdit\_2.text())

if self.checkBox\_3.isChecked():

failToInject.append(self.textEdit.toPlainText())

if self.checkBox\_4.isChecked():

failToInject.append(self.textEdit\_2.toPlainText())

# Check pin checkboxes to set the nodes to inject

if self.checkBox\_5.isChecked():

aux5 = self.checkBox\_5.text()

pinsToInject.append(aux5.toLower())

if self.checkBox\_6.isChecked():

aux6 = self.checkBox\_6.text()

pinsToInject.append(aux6.toLower())

if self.checkBox\_7.isChecked():

aux7 = self.checkBox\_7.text()

pinsToInject.append(aux7.toLower())

# Check MOSTYPE checkboxes to set the nodes to inject

if self.radioButton\_3.isChecked():

typesToInject.append(self.radioButton\_3.text())

if self.radioButton\_2.isChecked():

typesToInject.append(self.radioButton\_2.text())

if self.radioButton.isChecked():

typesToInject.append(self.radioButton\_3.text())

typesToInject.append(self.radioButton\_2.text())

# Add custom point - NOT WORKING. NOW EXCLUDING THE textEdit POINTS!

#if self.checkBox\_8.isChecked():

# self.textEdit\_3.toPlainText()

# Check if all required vars were set

if not sourceFile:

self.statusbar.showMessage("Please set a CIR source file!",

1200)

if not outputDir:

self.statusbar.showMessage("Please set the output dir for generated files",

1200)

if not failToInject:

self.statusbar.showMessage("Please set at least one fail to inject",

1200)

if not pinsToInject:

self.statusbar.showMessage("Please set at least one pin to inject",

1200)

if not typesToInject:

self.statusbar.showMessage("Please set at least one type to inject",

1200)

#self.statusbar.showMessage(sourceFile,1200)

#time.sleep(1)

#self.statusbar.showMessage(outputDir,1200)

#time.sleep(1)

#self.statusbar.showMessage(str(failToInject),1200)

#time.sleep(1)

#self.statusbar.showMessage(str(nodesToInject),1200)

self.cirFileTab = {'cirsrc':sourceFile, 'outdir':outputDir,

'injfail':failToInject, 'injpins':pinsToInject,

'injtypes':typesToInject}

return self.cirFileTab

def generate\_files(self):

""" This method is in charge of retrieve all params to call

fail\_inject.py and generate the simulations files.

"""

self.\_get\_cirfiletab\_vars()

injectFail = fail\_inject.FailInject(self)

injectFail.run(self.cirFileTab['cirsrc'],self.cirFileTab['outdir'],

self.cirFileTab['injfail'],self.cirFileTab['injpins'],

self.cirFileTab['injtypes'])

def open\_cir\_files(self):

""" This method open the selected cir file."""

selectedItem = self.treeWidget\_2.currentItem()

cirFile = []

cirPath = ''

while selectedItem != None:

cirFile.append(str(selectedItem.text(0)))

selectedItem = selectedItem.parent()

for fp in cirFile:

cirPath = fp + '\\' + cirPath

cirPath = cirPath.rstrip('\\')

args = 'notepad++ %s.cir' % cirPath

proc = subprocess.Popen(args)

if \_\_name\_\_ == '\_\_main\_\_':

app = QtGui.QApplication(sys.argv)

app.allWidgets()

win = MainWindow()

win.show()

status = app.exec\_()

app.closeAllWindows()

sys.exit(status)

**FIN DEL PROGRAMA PRINCIPAL**

**INICIO DE LA LIBRERÍA DE INYECCION**

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

#

# fail\_inject.py

#

# Copyright 2010 Facundo-std <Facundo-std@FACUNDO-XPS>

#

# This program is free software; you can redistribute it and/or modify

# it under the terms of the GNU General Public License as published by

# the Free Software Foundation; either version 2 of the License, or

# (at your option) any later version.

#

# This program is distributed in the hope that it will be useful,

# but WITHOUT ANY WARRANTY; without even the implied warranty of

# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

# GNU General Public License for more details.

#

# You should have received a copy of the GNU General Public License

# along with this program; if not, write to the Free Software

# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,

# MA 02110-1301, USA.

""" This module injects fails into CIR files """

#Python imports

import sys

import re

import os

import time

import datetime

#PyQt imports

from PyQt4 import QtCore, QtGui

#UI imports

#from ui.simulation\_analyzer import Ui\_MainWindow

class FailInjectError(Exception):

""" Module main exception """

pass

class FailInject():

""" This class is to manage fail injections in CIR files. """

def \_\_init\_\_ (self,gui=None):

""" Initializer """

# Dict containing all transistors in the cir and all nodes information

self.\_trans = {}

# Dict with node information. This var is stored for each key in \_trans

self.\_nodes = {}

# This var stores the source circuit file

self.\_cirfile = ''

# This var stores the content of the source circuit file

self.\_cirfilecont = ''

# This var has all nodes involved in a fail injection point

self.\_related\_transistors = []

# This var contains the string expresion of the fail

self.\_fail = ''

# This var holds the output dir

self.\_outdir = ''

# This var holds the base of the output dir

self.\_basedir = ''

# This var holds the fail dir

self.\_faildir = ''

# This var holds the injection point

self.\_inject\_point = ''

# This var holds the type of transistor in wich the fail'll be injected

self.\_trantype = []

# This var holds the pin in which the fail will be injected

self.\_pintype = []

# This var holds the address of the MainWindow GUI

self.\_gui = gui

# This var holds the parent TreeWidgetItem for all othres

self.\_topdir = None

def \_transistor\_count (self,file):

""" This method count the transistors on the given file. """

f = open(file)

counter = 0

for line in f:

if re.match("M.\*",line):

counter = counter + 1

return counter

def \_get\_nodes (self,file):

""" This method returns a dict with all transistor names as keys,

and another dict with nodename as key and node as value as value.

e.g.

self.\_trans = {'transistor1':{

'drain':nodo1,

'gate':nodo2,

'source':nodo3,

'bulk':nodo4},

'transistor2':{...}}"""

self.\_trans = {}

f = open(file)

for line in f:

if re.match("M.\*",line):

aux = re.split('\W+',line)

self.\_nodes = {'drain': aux[1],

'gate': aux[2],

'source': aux[3],

'bulk': aux[4],

'type': aux[5]}

self.\_trans[aux[0]] = self.\_nodes

def \_inject\_fail (self,ffail,nnode,ooutdir,ppoint,ttype,ttran):

""" This method injects the code into the CIR file """

# Opens a new cirfile

os.chdir(ooutdir)

file\_ = ppoint + '.cir'

file\_\_ = ppoint + '.cmd'

# Generate a command file to automate the graph of the curves in PSPICE

try:

f2 = open(file\_\_,'w')

except:

print "Could not create the file %s" % file\_\_

createdate = datetime.datetime.today()

f2.write("""\*Command file created by Fail Injector for \

PSpice version 16.0.0\n""")

f2.write("\*Creation date: %s\n" % createdate)

self.\_generate\_cmd(f2,ppoint,ooutdir)

f2.close()

# Generate the cir file

try:

f = open(file\_,'w')

except:

print "Could not create the file %s" % file\_

# Add the source file

for line in self.\_cirfilecont:

f.write(line)

# Injection information

f.write("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")

f.write("\* Transistor: %s\n" % ttran)

f.write("\* Transistor type: %s\n" % ttype)

f.write("\* Punto de inyeccion: %s\n" % ppoint)

#f.write("\* Related nodes: %s\n" % nnode)

f.write("\* Falla: %s\n" % ffail)

f.write("\* Directorio: %s\n" % ooutdir)

f.write("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n")

# Add ffail code at bottom

if ttype == 'CMOSP':

f.write("\* P CMOS type injection\n")

f.write("I\_INY1 0 %s DC 0Adc AC 0Aac\n" % ppoint)

elif ttype == 'CMOSN':

f.write("\* N CMOS type injection\n")

f.write("I\_INY1 %s 0 DC 0Adc AC 0Aac\n" % ppoint)

else:

f.write("\* CMOS type could not be identified. Setting default config:\n")

f.write("I\_INY1 0 %s DC 0Adc AC 0Aac\n" % ppoint)

f.write("%s\n\n" % ffail)

# Add probe cmd for ffail. Add \n for multiple lines

f.write("\* Out voltages\n")

f.write(".PROBE/CSDF V([C\_F\_D\_LSB])\n")

f.write(".PROBE/CSDF V([C\_F\_D\_2SB])\n")

f.write(".PROBE/CSDF V([C\_F\_D\_3SB])\n")

f.write(".PROBE/CSDF V([C\_F\_D\_4SB])\n")

f.write(".PROBE/CSDF V([C\_F\_D\_5SB])\n")

f.write(".PROBE/CSDF V([C\_F\_C32\_MSB])\n\n")

# Add probe cmd for ppoint.

f.write("\* Voltage and current at the injection point\n")

f.write(".PROBE/CSDF V([%s])\n" % ppoint)

f.write(".PROBE/CSDF I(I\_INY1)\n\n")

# Add probe cmd for pin nnodes involved

f.write("\* Related nodes values\n")

for \_tran in nnode:

f.write(".PROBE/CSDF ID(%s) IB(%s) IS(%s) IG(%s)\n" % \

(\_tran,\_tran,\_tran,\_tran))

# Add .END cmd.

f.write("\n.END\n")

# Close file

f.close()

def \_get\_related\_transistors(self,drainmos=None):

""" This method returns a list with all transistors related to a node"""

# Clean the list

self.\_related\_transistors = []

nodes = ''

if drainmos == None:

return None

for mos in self.\_trans:

nodes = self.\_trans[mos]["drain"] + self.\_trans[mos]["gate"] + \

self.\_trans[mos]["source"] + self.\_trans[mos]["bulk"]

if drainmos in nodes:

self.\_related\_transistors.append(str(mos))

def \_generate\_cmd (self,fg,point,outputdir):

""" Genereta the cmd file for PSPICE 16.0 """

fg.write("File Open\n")

fg.write("""%s\\%s.csd\n""" % (outputdir,point))

fg.write("OK\n")

fg.write("Plot Add\_Plot\n")

fg.write("Plot Add\_Plot\n")

fg.write("Plot Add\_Plot\n")

fg.write("Plot Add\_Plot\n")

fg.write("Plot Add\_Plot\n")

fg.write("Plot Add\_Plot\n")

fg.write("Trace Add\n")

fg.write("V(C\_F\_C32\_MSB)\n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("6\n")

fg.write("Trace Add\n")

fg.write("V(C\_F\_D\_5SB)\n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("5\n")

fg.write("Trace Add\n")

fg.write("V(C\_F\_D\_4SB)\n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("4\n")

fg.write("Trace Add\n")

fg.write("V(C\_F\_D\_3SB)\n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("3\n")

fg.write("Trace Add\n")

fg.write("V(C\_F\_D\_2SB)\n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("2\n")

fg.write("Trace Add\n")

fg.write("V(C\_F\_D\_LSB)\n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("1\n")

fg.write("Trace Add\n")

fg.write("V(%s)\n" % point)

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("2\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Y Axis\n")

fg.write("Set Range\n")

fg.write("0V 4V\n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("3\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Save As Defaults\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Y Axis\n")

fg.write("Set Range\n")

fg.write("0 4 \n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("4\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Save As Defaults\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Y Axis\n")

fg.write("Set Range\n")

fg.write("0 4 \n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("5\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Save As Defaults\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Y Axis\n")

fg.write("Set Range\n")

fg.write("0 4 \n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("6\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Save As Defaults\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Y Axis\n")

fg.write("Set Range\n")

fg.write("0 4 \n")

fg.write("OK\n")

fg.write("Plot Select\n")

fg.write("7\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Save As Defaults\n")

fg.write("Plot Axis\_Settings\n")

fg.write("Y Axis\n")

fg.write("Set Range\n")

fg.write("0 4 \n")

fg.write("OK\n")

fg.write("\n")

def run (self,file\_=None,dir\_=None,fail\_=None,pins\_=None,mostype\_=None):

""" Main method """

# Check if a file was received in the calling of the function

if file\_ == None:

ex = "Please insert a CIR file!"

sys.exit(ex)

raise FailInjectError(ex)

else:

try:

f = open(file\_)

self.\_file = file\_

f = None

except IOError, ex:

print "Could not open file: %s" % file\_

raise FailInjectError(ex)

# Checks if an output dir was passed in the calling

if dir\_ == None:

ex = "Please enter an output directory"

sys.exit(ex)

raise FailInjectError(ex)

else:

try:

tmp = os.getcwd()

#print "DIR: %s" % dir\_

os.chdir(dir\_) # NameError exception occur if dir\_ doesn't exist

self.\_outdir = dir\_

self.\_basedir = self.\_outdir

os.chdir(tmp)

except WindowsError, ex:

print "Could not change to dir %s" % dir\_

raise FailInjectError(ex)

# Checks if a fail was entered

if fail\_ == None:

ex = "Please enter a fail to inject"

sys.exit(ex)

raise FailInjectError(ex)

else:

self.\_fail = fail\_

# Checks if a node/s was entered

if pins\_ == None:

ex = "Please specify the nodes to inject the fail"

sys.exit(ex)

raise FailInjectError(ex)

else:

self.\_pintype = pins\_

# Read the transistors to parse

print "############################################"

print "Transistors: %s" % self.\_transistor\_count(self.\_file)

print "Source file: %s" % self.\_file

print "Fail: %s" % self.\_fail

print "Outdir: %s" % self.\_outdir

print "Nodes to inject: %s" % pins\_

print "############################################"

# Parse file to allow fail injection and sets: self.\_trans

self.\_get\_nodes(self.\_file)

ptrans = 0

ntrans = 0

for mos in self.\_trans:

if self.\_trans[mos]["type"] == 'CMOSP':

ptrans = ptrans + 1

elif self.\_trans[mos]["type"] == 'CMOSN':

ntrans = ntrans + 1

print "CMOS P type transistors: %s" % ptrans

print "CMOS N type transistors: %s" % ntrans

try:

f = open(self.\_file,'r')

except:

print "Could not open the file %s" % self.\_file

self.\_cirfilecont = f.readlines()

f.close()

# This block calculates the total files to generate

if mostype\_.\_\_len\_\_() == 2:

totalPercentage = self.\_fail.\_\_len\_\_() \* self.\_transistor\_count(self.\_file)

if (mostype\_.\_\_len\_\_() == 1):

totalPercentage = self.\_fail.\_\_len\_\_() \* self.\_transistor\_count(self.\_file)

#if (mostype\_.\_\_len\_\_() == 1) and ('CMOSP' in mostype\_):

# totalPercentage = self.\_fail.\_\_len\_\_() \* ptrans

#if (mostype\_.\_\_len\_\_() == 1) and ('CMOSN' in mostype\_):

# totalPercentage = self.\_fail.\_\_len\_\_() \* ntrans

items\_qty = totalPercentage

if items\_qty == 0:

step = 0

else:

step = (100.00 / items\_qty)

currentProgress = 0

####################################################################

# Set the TreeView

self.\_gui.treeWidget\_2.clear()

self.\_topdir = QtGui.QTreeWidgetItem(QtGui.QTreeWidgetItem.Type)

self.\_topdir.setText(0,self.\_basedir)

self.\_gui.treeWidget\_2.addTopLevelItem(self.\_topdir)

counter = 0

for cfail in self.\_fail:

# Create a dir for each new fail

counter = counter + 1

failname = "fail\_%s" % counter

self.\_faildir = "%s\\%s" % (self.\_basedir,failname)

try:

os.mkdir(self.\_faildir)

except WindowsError, ex:

if 183 == ex.winerror:

print "Directory %s already created. Skipping..." % \

self.\_faildir

else:

raise FailInjectError(ex)

faildir = QtGui.QTreeWidgetItem(self.\_topdir)

faildir.setText(0,failname)

self.\_gui.treeWidget\_2.addTopLevelItem(faildir)

for eachmostype in mostype\_:

self.\_outdir = "%s\\%s" % (self.\_faildir,eachmostype)

try:

os.mkdir(self.\_outdir)

except WindowsError, ex:

if 183 == ex.winerror:

print "Directory %s already created. Skipping..." % \

self.\_outdir

else:

raise FailInjectError(ex)

mostypedir = QtGui.QTreeWidgetItem(faildir)

mostypedir.setText(0,eachmostype)

self.\_gui.treeWidget\_2.addTopLevelItem(mostypedir)

# This avoid the app to crash due a lot of disk writing

time.sleep(1)

# Inject the fail into the new file

for mos in self.\_trans:

# Detect where to inject the fail

excluded = self.\_gui.checkBox\_8.isChecked() and \

(str(self.\_gui.textEdit\_3.toPlainText()) in \

mos)

if not excluded:

# Get the mos type

self.\_trantype = self.\_trans[mos]["type"]

if self.\_trantype == eachmostype:

for eachpin in self.\_pintype:

# Select the pin to inject the fail

self.\_inject\_point = self.\_trans[mos][str(eachpin)]

# Parse file and sets: self.\_related\_transistors

self.\_get\_related\_transistors(self.\_inject\_point)

cirfileitm = QtGui.QTreeWidgetItem(mostypedir)

cirfileitm.setText(0,self.\_inject\_point)

self.\_gui.treeWidget\_2.addTopLevelItem(cirfileitm)

# Generate all the files with the fail inside

self.\_inject\_fail(cfail,self.\_related\_transistors,

self.\_outdir,self.\_inject\_point,

self.\_trantype,mos)

currentProgress = currentProgress + step

self.\_gui.progressBar2.setValue(currentProgress.\_\_trunc\_\_())

# Avoid the progress bar be setted to 0 until 1 second pass

self.\_gui.progressBar2.setValue(100)

time.sleep(1)

self.\_gui.progressBar2.setValue(0)

self.\_gui.statusbar.showMessage("DONE!",1200)

# Print a specific transistor information

#print self.\_trans

if \_\_name\_\_ == '\_\_main\_\_':

A = FailInject()

CIRFILE = sys.argv[1]

OUTDIR = sys.argv[2]

FALLAS = [sys.argv[3],sys.argv[4]]

NODOS = [sys.argv[5],sys.argv[6]]

MOSTYPE = ['CMOSP','CMOSN']

A.run(CIRFILE,OUTDIR,FALLAS,NODOS,MOSTYPE)

**FIN DE LA LIBRERÍA DE INYECCION**

**INICIO DE LA LIBRERÍA DE ANALISIS**

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

#

# parser.py

#

# Copyright 2010 Facundo-std <Facundo-std@FACUNDO-XPS>

#

# This program is free software; you can redistribute it and/or modify

# it under the terms of the GNU General Public License as published by

# the Free Software Foundation; either version 2 of the License, or

# (at your option) any later version.

#

# This program is distributed in the hope that it will be useful,

# but WITHOUT ANY WARRANTY; without even the implied warranty of

# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

# GNU General Public License for more details.

#

# You should have received a copy of the GNU General Public License

# along with this program; if not, write to the Free Software

# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,

# MA 02110-1301, USA.

""" This module parses CSDF file format """

#python imports

import os, sys, string, re, time, mmap

from stat import \*

#from common import \_isfile

import common

import decimal as dc

class ParserMain():

""" Main class of the parser"""

def \_\_init\_\_ (self):

""" Initializer """

#try:

self.dir = ''

#self.dir = sys.argv[1]

#except IndexError, ex:

# sys.exit("Please insert a directory!")

self.dir\_dict = {}

self.dir\_list = []

self.header = {}

self.nodes = {}

self.nodes\_values = {}

# Data structure pf a CSD file

\_header = {"SOURCE":None,

"VERSION":None,

"TITLE":None,

"SUBTITLE":None,

"TIME":None,

"DATE":None,

"TEMPERATURE":None,

"ANALYSIS":None,

"SERIALNO":None,

"ALLVALUES":None,

"COMPLEXVALUES":None,

"NODES":None,

"SWEEPVAR":None,

"SWEEPMODE":None,

"XBEGIN":None,

"XEND":None,

"FORMAT":None,

"DGTLDATA":None}

\_ntime = []

\_nnodes = []

\_voltages = {}

\_body = {"ntime":\_ntime,

"nnodes":\_nnodes,

"voltages":\_voltages}

self.\_csdfile = {"header":\_header,

"body":\_body}

def \_filetype (self,file):

""" Return file for a regular file and dir for a directory. """

if os.path.isdir(file):

return "dir"

elif os.path.isfile(file):

return "file"

else:

return "unknow"

def \_isfilecsdf (self,file):

""" Function to check if file is a CSDF file format.

For now the function only checks the extension. """

if os.path.isfile(file) and os.access(file,os.R\_OK):

[filename,extension] = os.path.splitext(file)

if extension != ".csd":

return False

else:

return True

else:

print "WARNING: The file %s doesn't exists or you haven't read permissions" \

% file

return 254

def \_readdir (self,name\_dir):

""" Read dir (a full path)and return a dict with dir contents.

The dict var has the format {filename,filetype}

e.g. {'design':directory, 'readme.txt':file}"""

print "INFO: Root dir: %s " % name\_dir

if os.path.exists(name\_dir):

try:

os.chdir(name\_dir)

except Exception, ex:

print "WARNING: Could not change to %s " % name\_dir

print "WARNING: Please check the directory permissions"

return 254

self.dir\_list = []

self.dir\_dict = {}

self.dir\_list = os.listdir(name\_dir)

else:

print "INFO: Directory not found!"

self.dir\_dict = {}

return 255

for i in self.dir\_list:

self.dir\_dict[i] = self.\_filetype(i)

return self.dir\_dict

def \_parsecsdf (self,file):

""" Function to parse the CSDF file type. """

header\_str = ''

if not common.\_isfile(self,file,".csd"):

return 254

f = open(file)

try:

line = f.readline()

line = f.readline()

line = re.findall(r"SOURCE='(.\*)' VERSION='(.\*)'",line)

(self.header['SOURCE'], self.header['VERSION']) = line.pop()

line = f.readline()

line = re.findall(r"TITLE='(.\*)'",line)

(self.header['TITLE']) = line.pop()

line = f.readline()

line = re.findall(r"SUBTITLE='(.\*)'",line)

(self.header['SUBTITLE']) = line.pop()

line = f.readline()

line = re.findall(r"TIME='(.\*)' DATE='(.\*)' TEMPERATURE='(.\*)'",line)

(self.header['TIME'], self.header['DATE'], self.header['TEMPERATURE']) = line.pop()

line = f.readline()

line = re.findall(r"ANALYSIS='(.\*)' SERIALNO='(.\*)'",line)

(self.header['ANALYSIS'], self.header['SERIALNO']) = line.pop()

line = f.readline()

line = re.findall(r"ALLVALUES='(.\*)' COMPLEXVALUES='(.\*)' NODES='(.\*)'",line)

(self.header['ALLVALUES'], self.header['COMPLEXVALUES'], self.header['NODES']) = line.pop()

line = f.readline()

line = re.findall(r"SWEEPVAR='(.\*)' SWEEPMODE='(.\*)'",line)

(self.header['SWEEPVAR'], self.header['SWEEPMODE']) = line.pop()

line = f.readline()

line = re.findall(r"XBEGIN='(.\*)' XEND='(.\*)'",line)

(self.header['XBEGIN'], self.header['XEND']) = line.pop()

line = f.readline()

line = re.findall(r"FORMAT='(.\*)'",line)

(self.header['FORMAT']) = line.pop()

line = f.readline()

line = re.findall(r"DGTLDATA='(.\*)'",line)

(self.header['DGTLDATA']) = line.pop()

line = f.readline()

line = f.readline()

self.nodes = line.split(' ')

if self.nodes[len(self.nodes)-1] == '\n':

self.nodes.pop()

except Exception, ex:

print "WARNING: The file %s seems to be empty. Please check it." % file

print "CRITICAL: %s" % ex

#Only for debug

#for key in self.header:

# print "%s -> %s " % (key,self.header[key])

lines = f.readlines()

timeseek = []

for box in lines:

if "#C" in box:

timeseek = box.split(' ')

self.nodes\_values[timeseek[1]] = ''

elif "#;" in box:

pass

else:

nodesvalues = box.split(' ')

count = 0

aux = []

for i in nodesvalues:

i = i.split(':') #i0

aux.append(i[0])

aux.pop()

nodesvalues = aux

self.nodes\_values[timeseek[1]] = nodesvalues

return self.nodes\_values,self.nodes

#debug muestra la info del dict con los datos

#for key in self.nodes\_values:

# print "%s -> %s" % (key, self.nodes\_values[key])

def \_get\_headers(self,map,\_first,\_second):

""" extract the headers values from the mapped file """

\_value = map[map.find(\_first):map.find(\_second)]

\_value = \_value.rstrip()

\_value = re.findall(r"(.\*)'(.\*)'",\_value).pop()

\_value = \_value[0] + " " + \_value[1]

return \_value

def \_parsecsdf2(self,file):

""" This is an improved way to parse CSD files """

f = open(file, "r+b")

map = mmap.mmap(f.fileno(),0)

f.close()

# PARSE HEADER OF THE FILE

self.\_csdfile["header"]["SOURCE"] = \

self.\_get\_headers(map,"SOURCE","VERSION")

self.\_csdfile["header"]["VERSION"] = \

self.\_get\_headers(map,"VERSION","TITLE")

self.\_csdfile["header"]["TITLE"] = \

self.\_get\_headers(map,"TITLE","SUBTITLE")

self.\_csdfile["header"]["SUBTITLE"] = \

self.\_get\_headers(map,"SUBTITLE","TIME")

self.\_csdfile["header"]["TIME"] = \

self.\_get\_headers(map,"TIME","DATE")

self.\_csdfile["header"]["DATE"] = \

self.\_get\_headers(map,"DATE","TEMPERATURE")

self.\_csdfile["header"]["TEMPERATURE"] = \

self.\_get\_headers(map,"TEMPERATURE","ANALYSIS")

self.\_csdfile["header"]["ANALYSIS"] = \

self.\_get\_headers(map,"ANALYSIS","SERIALNO")

self.\_csdfile["header"]["SERIALNO"] = \

self.\_get\_headers(map,"SERIALNO","ALLVALUES")

self.\_csdfile["header"]["ALLVALUES"] = \

self.\_get\_headers(map,"ALLVALUES","COMPLEXVALUES")

self.\_csdfile["header"]["COMPLEXVALUES"] = \

self.\_get\_headers(map,"COMPLEXVALUES","NODES")

self.\_csdfile["header"]["NODES"] = \

self.\_get\_headers(map,"NODES","SWEEPVAR")

self.\_csdfile["header"]["SWEEPVAR"] = \

self.\_get\_headers(map,"SWEEPVAR","SWEEPMODE")

self.\_csdfile["header"]["SWEEPMODE"] = \

self.\_get\_headers(map,"SWEEPMODE","XBEGIN")

self.\_csdfile["header"]["XBEGIN"] = \

self.\_get\_headers(map,"XBEGIN","XEND")

self.\_csdfile["header"]["XEND"] = \

self.\_get\_headers(map,"XEND","FORMAT")

self.\_csdfile["header"]["FORMAT"] = \

self.\_get\_headers(map,"FORMAT","DGTLDATA")

self.\_csdfile["header"]["DGTLDATA"] = \

self.\_get\_headers(map,"DGTLDATA","#N")

# PARSE THE BODY OF THE FILE

\_body = map[map.find("#N")+2:map.find("#C")]

\_body = \_body.split(' ')

for \_each in \_body:

\_each = \_each.strip()

if \_each != '':

self.\_csdfile["body"]["nnodes"].append(\_each)

self.\_csdfile["body"]["voltages"][\_each] = []

if self.\_csdfile["header"]["NODES"].split(' ')[1] == \

str(self.\_csdfile["body"]["nnodes"].\_\_len\_\_()):

print "Var nodes match nodes readed!!"

else:

print "Var nodes NOT match nodes readed!!"

\_rawdata = map[map.find("#C"):]

\_rawdata = \_rawdata.split("#C ")

\_rawdata.\_\_delitem\_\_(0)

#print "el split tiene %s" % \_rawdata.\_\_len\_\_()

for \_tslot in \_rawdata:

#print "%s : %s" % (counter,times.\_\_getslice\_\_(0,times.find("\r\n")))

(\_time, \_sep, \_vals) = \_tslot.partition("\r\n")

self.\_csdfile["body"]["ntime"].append(\_time)

\_vals = \_vals.split(' ')

\_vals.pop()

for \_each in \_vals:

\_each = \_each.strip()

(\_volt,\_node) = \_each.split(':')

\_node = int(\_node,16) - 1

\_node = self.\_csdfile["body"]["nnodes"][\_node]

if \_each != '':

self.\_csdfile["body"]["voltages"][\_node].append(dc.Decimal(\_volt))

return self.\_csdfile

def run (self,csdfile):

""" Function doc """

#print self.\_readdir(self.dir)

if self.\_isfilecsdf(csdfile):

return self.\_parsecsdf2(csdfile)

else:

print "ERROR: %s" % self.\_isfilecsdf(csdfile)

#print "HOLA"

if \_\_name\_\_ == '\_\_main\_\_':

A = ParserMain()

FILE = "D:\\test\\test-LSB.csd"

pepo = A.run(FILE)

#print "DICTIONARY:\n"

#print A.\_csdfile.keys()

#print A.\_csdfile["body"]

nodo = A.\_csdfile["body"]["nnodes"][0]

valor = A.\_csdfile["body"]["voltages"][nodo]

print "Values en el nodo %s = %s" % (nodo,valor)

nodo = pepo["body"]["nnodes"][0]

valor = pepo["body"]["voltages"][nodo]

print "Values en el nodo %s = %s" % (nodo,valor)

**FIN DE LA LIBRERÍA DE ANALISIS**