Documentation wsi-retviz

Markus Hellenbrand

March 2013

Internship

Electronic Vision(s)

Table of contents

1 Introduction

The programme wsi-retviz Visualisation (retviz in the following) documented in the following was developed by Markus Hellenbrand during an internship in the Electronic Vision(s) group. It provides a comprehensive GUI[1] to display voltages, currents and temperatures on the wafer. The data is read using the HALbe interface and the pyhalbe module from the symap2ic project. This documentation gives an overview of the functionality and a detailed explanation of the code structure.

2 Requirements

From the BrainScaleS repository symap2ic has to be installed, which provides the readout access for the relevant data. One way to install it is following these steps:

- git clone git@gitviz.kip.uni-heidelberg.de:symap2ic.git
- cd symap2ic
- . bootstrap.sh.UHEI
- ./waf set_config halbe
- ./waf configure
- ./waf build --test-execnone
- ./waf install --test-execnone

Then place the wsi-retviz folder inside symap2ic (if it is not already included) via

git clone git@gitviz.kip.uni-heidelberg.de:wsi-retviz.git

Beyond symap2ic, using retviz requires certain Python software components on the system executing the programme. Python 2.7 needs to be installed and the modules matplotlib (*http://matplotlib.org/downloads.html*), NumPy (*http://www.scipy.org/Download* – NumPy, not SciPy) and Tkinter need to be added, whereas Tkinter is usually already installed with the Python 2.7 installation and the other two modules should come along with symap2ic.

There is already a Python 3.y version available, but it is not backwards compatible due to partly different syntax and different function implementation. This means, that the programme will most likely not run when executed with a Python 3.y interpreter. This was not tested during development. For more information on Python 3 and its features/none backwards compatibility see *What's new in Python 3*.

[1] GUI – Graphical User Interface

3 GUI

To start the GUI it is necessary to execute

. boostrap.sh.UHEI

in the symap2ic folder first. Then move to the wsi-retviz folder and start the programme via

python retviz.py or via python iretviz.py,

according to which version you want to run. The difference lies in the overview functionality. In the "normal" retviz version an overview over all voltages and all reticles appears in an extra window and does not update continuously, whereas in the iretviz version the overview appears in the main canvas, where the single voltage displaying takes place. It updates continuously, but the overview update runs very slow in this version. For further details about the update speed see *3.1 Delay*.

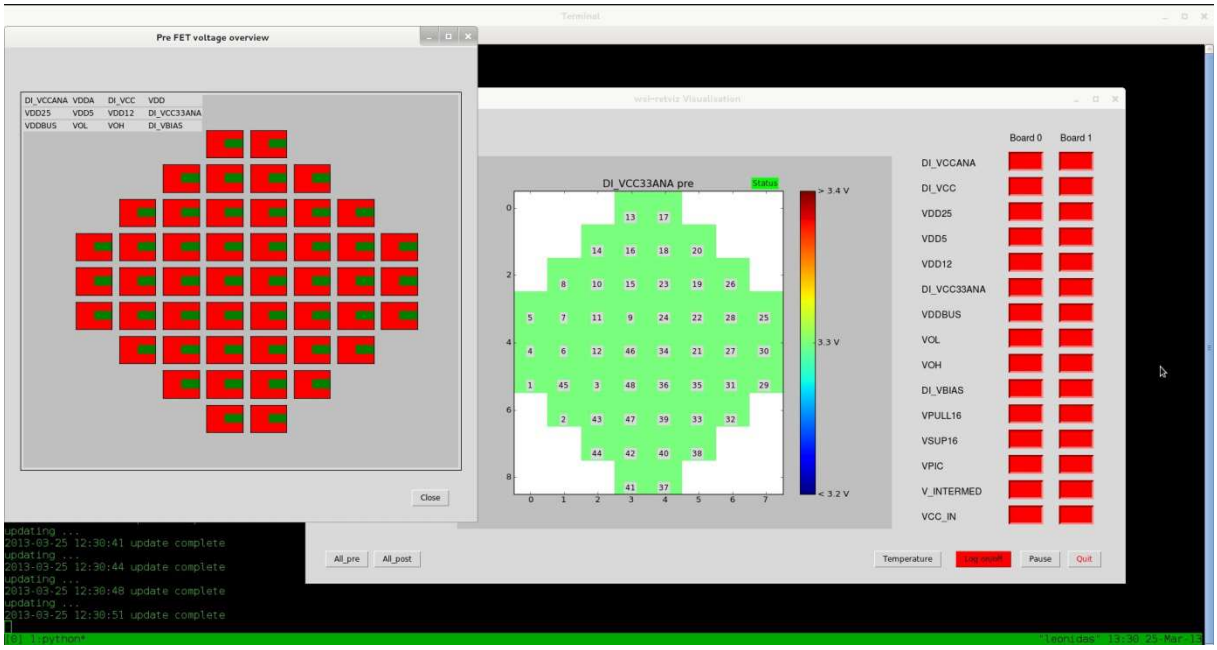

Figure 1: retviz GUI – retviz version, not iretviz

3.1 Threads

As the tasks of the programme differ, they are distributed to different threads. Figure 2 below shows an overview of the running threads. The grey divisions symbolise button clicks, whereas the first one is the start button and the last one the Quit button. The buttons in between are specific for the used functions: temperature button/overview button.

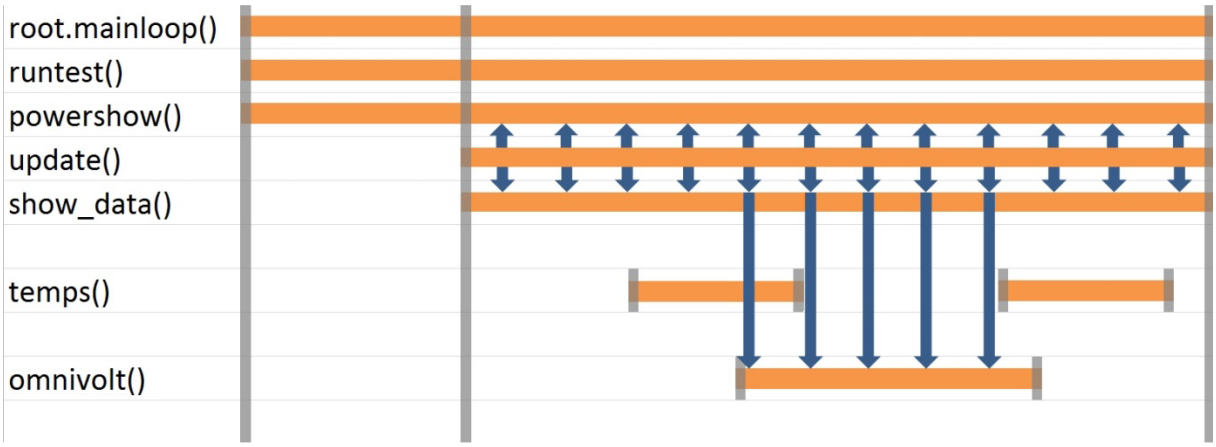

Figure 2: Different threads running during use of the retviz programme.

3.2 Delay

The retviz GUI may seem to run a little stagnant sometimes. The reason for that is Python's Global Interpreter Lock (GIL). As Python is not thread-safe, the GIL is a kind of workaround to prevent confusion in the memory management. The GIL's way to "solve" the missing thread-safety is to block all threads except for one, so only one thread accesses the available memory. Threads request to acquire the GIL to execute their commands. When they are done, the other threads have the chance to acquire the GIL. (For further information see *http://wiki.python.org/moin/GlobalInterpreterLock* or *http://www.dabeaz.com/python/GIL.pdf*)

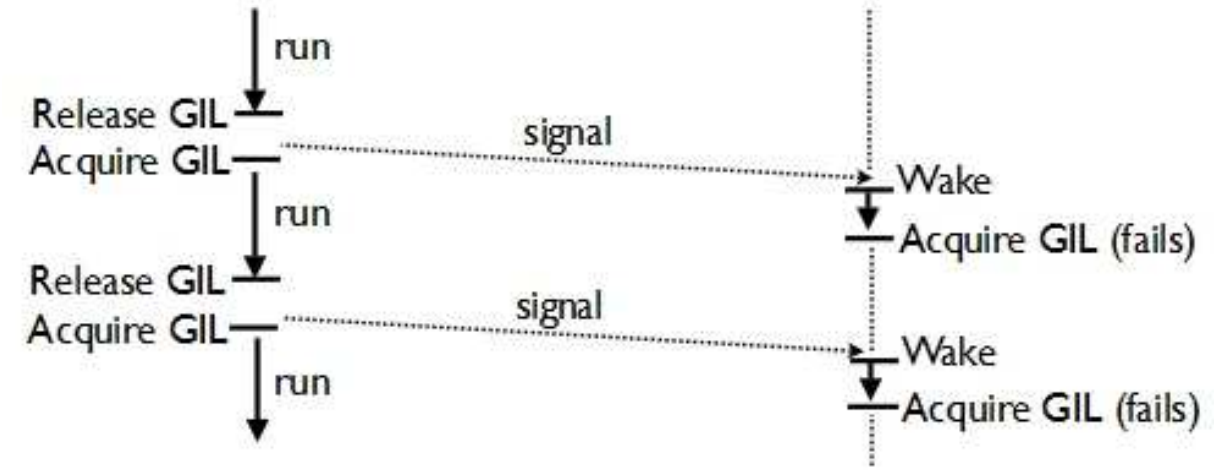

Figure 3: Schematical illustration of GIL problem for two threads, accordingly for more threads. With more than one CPU core one thread may be blocked for very long time. (See second link above, slide 35.)

retviz uses several threads for updating and displaying. While update threads are running, the visualisation threads are blocked and thus buttons or displays sometimes react with some delay. When temperature displaying is enabled, this delay may take up

to a few seconds, because at the date of development (last update on 25th of March, 2013), the temperature readout took this long. To prevent the updating threads from blocking the GUI thread completely, the updating thread's functions contain sleep times.

In the iretviz version the overview for all voltages is integrated in the main canvas, where the single voltages are shown. As well as the single voltage visualisation, it updates continuously. Switching to an overview or from an overview back to single voltages takes up to 30 seconds, so iretviz and especially its overview functionality is only recommended for long term monitoring. The "normal" retviz's overview functionality opens a new window, which takes a few seconds as well, but much fewer than in iretviz.

The reason for the long delay in iretviz is (most probably) again the GIL, which was determined through some tests, shortly described in the following. The readout conducted for the overview takes around 500 ms, tested with timeit in IPython, so the readout is not the reason for the delay. The other functions involved were not tested in IPython due to their complexity. However their tasks are not complex enough to cause a delay like the one given here. The longest time to execute is needed by the overview() function, which runs in omnivolt.py (extra window for overview) in a similar way as it does in the iretviz version. It lasts a few seconds. Examining the switch from the overview back to the single voltages narrowed the problem down on the GIL. The dropdown menu's selection is immediately passed to the show_data() function, which also immediately updates the wafer image according to the chosen value. Examining the switch back to single voltages showed, that the show_data() thread did not do anything for some time, after the overview was active. The only obvious reason for this thread not doing anything is the GIL. In fact, the GIL may even block the show_data() thread completely. (For a descriptive explanation see *www.dabeaz.com/python/UnderstandingGIL.pdf slide 34)*

Another delay of some seconds occurs, when "pause" is enabled. The reason might be, that the update code is "executed" faster, whereas in fact, most of it is skipped, so the show_data() thread has less chance to obtain the GIL.

3.3 Functionality

When starting retviz, the programme immediately starts to test the connection to the board. This happens via a simple try statement, which tries to read out a single reticle. The result is displayed in a status label (1) in the top right corner of the visualisation canvas. The powerboard control (2) on the GUI's right side also runs straight from the beginning, but is not enabled completely: Its visualisation thread starts right away, but it displays always the same data, as long as the update thread is not started (via start button) to acquire new data. The powerboard control shows, whether the generated

voltages are in a specified range around an optimal value. The optimal values for all data (optimal voltages, optimal currents, allowed deviations, resistances and optimal powerboard voltages) are stored in xml files (optimum_values.xml) and read out from the programme using the valueport module.

To start reading out the wafer voltages, the start button has to be activated. This destroys the button itself and places a pause button (3) in the same place, which allows pausing the update and delivers a snapshot image. (To avoid misunderstanding: This snapshot does not include saving any kind of data.) If no board connection is established, clicking the start button will cause an error popup.

Only before clicking the start button, the possibility to enable logging is given, using the Log on/off button (4). If enabled, all values are written to individual log files after every update. Excepted are the temperature values. As their readout takes several seconds, constant updating would block the whole programme. Temperature logging is only enabled with displaying temperature and is independent of the logging choice, as well as displaying the temperatures is independent of the start button. Clicking the start button disables the possibility to choose logging, because for now, en-/disabled logging starts different threads when clicking the start button. (This is faster than checking for logging on/off in every cycle.)

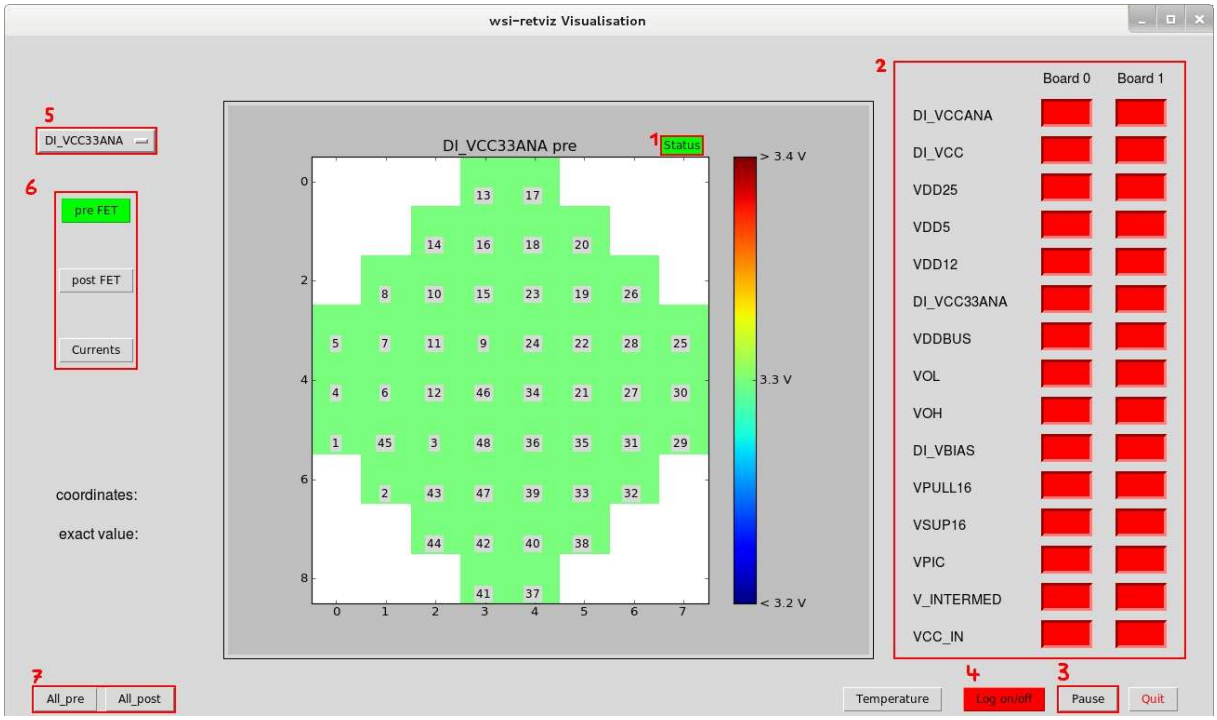

Figure 2: retviz GUI

Enabling and disabling logging while the update threads are running could be implemented by moving the log on/off test from the start() function to the update()

function. update_logs() would not be needed anymore and the update() function had to check after every update, if logging is enabled or disabled.

The choice of the voltages to be displayed is possible via the dropdown menu (5) in the upper left of the GUI. The buttons below (6) allow choosing the values before the FET[2]s, after the FETs or the currents through the FETs.

The buttons in the lower left (7) activate an overview over all reticles and all voltages, if retviz is running. Running iretviz, these buttons do not exist, as overviews are displayed in the same way, as single voltages are.

Clicking on a reticle inside the visualisation canvas displays the canvas coordinate and the exact value for this reticle. This functionality was implemented only rudimentary as a nice-to-have and is not completely debugged. See *4.2.1.8 class event_handler()* for further information.

4 Code

retviz is written in Python 2.7. Modules used are Tkinter (GUI), Matplotlib (visualising data), NumPy (handling arrays and converting lists to arrays), time, logging and thread (not the threading module, but the older thread module – was easier to use and sufficient for the given purposes).

4.1 Structure, moduling

The main code is retviz.py (or iretviz.py). It contains the most parts with reference to the GUI module Tkinter. The programme is divided in six modules.

The omnivolt module contains functions to display an overview over all reticles and all voltages. There are two versions of this module, one for the normal retviz and one (called iomnivolt) for the iretviz version. The latter does not contain positioning of the overview reticles, as this requires addressing the main figure from the iretviz module. In the normal retviz version, a new window (=> new root widget, details on root widgets see *4.2.1.3 root elements*) appears, so the positioning can take place in the omnivolt module as well.

supporting_functions.py contains some of the functions used by the programme, so there is one file bundling *supporting* functions, taking the code away from the other modules. valueport.py converts the xml files into Python understandable elements. logmodule.py initialises all loggers, if logging is enabled and also does the logging itself. Excepted are, as mentioned above, temperature values.

[2] field-effect transistor

A schematic description of the connections and data flows between the modules is given in figure 4 below.

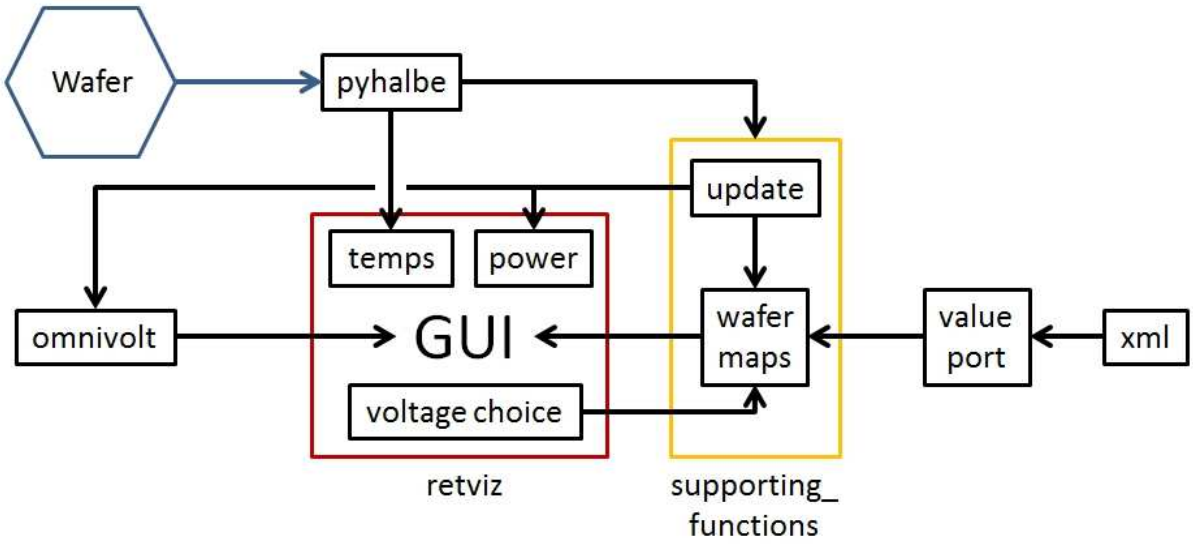

Figure 4: Data flow in retviz.

4.2 Functions

The following chapter describes the most important connections between the functions in the programme. It is divided the same way as the programme is, there are further chapters for every module. At every chapter's beginning there is a short summary about the module as a whole.

4.2.1 retviz.py

The retviz module as well as the iretviz module contain elementary functions, such as start(), which starts the update thread, show_data() and powershow(), which perform the actual visualisation and the dropdown menu to choose, which data is to be shown.

The retviz/iretviz module is also the one, which has to be executed in the terminal to start the whole application. This is, because it contains the root.mainloop() statement, which initialises the root widget containing all GUI elements.

4.2.1.1 Imports

Especially in the retviz module, several imports take place. Some of them are existing Python modules (matplotlib with matplotlib.pyplot and matplotlib.figure, numpy, thread, time, logging), the others were developed for this programme (omnivolt, logmodule, supporting_functions, cords). For the existing Python modules the internet delivers plenty of information, the newly developed modules are explained in detail below. A very short overview, what the already existing modules do, is taken from the code comments:

- matplotlib - python module, but not in python standard library, basis for matplotlib.pyplot
- matplotlib.use('TkAgg') - specifies backend to be used
- import omnivolt as omni - newly developed module for overview over all reticles and voltages
- import logmodule as lm - newly developed module for logging
- import supporting_functions as sup - newly developed module - contains some functions to make other modules contain less code
- import cords as c - newly developed module for setting coordinate labels
- import numpy as np - python module, but not in python standard library - numerical operations
- import matplotlib.pyplot as plt - python module, but not in python standard library - matlab-like plotting framework
- from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg - python module, but not in python standard library - special canvas for visualising numpy arrays
- from matplotlib.figure import Figure - python module, but not in python standard library - special figure needed for Tkinter
- import thread - standard python module - older threading module of the two available for python
- import time - standard python module; shell always shows system time - 1 hour (maybe no summer/winter time?) - time operations
- import logging - standard python module - logging
- from pyhalbe import * - necessary for data readout – comes along with symap2ic
- import sys - standard python module - grants advanced variable access and functions for interpreter interaction
- import Tkinter as Tk - GUI

4.2.1.2 pyhalbe

The pyhalbe module provides the functions able to read data from the board. The functions used are:

- Support.get_reticle_status(Coordinate.DNC(geometry.EnumGlobal(i))), where i is a DNC number.
  This reads the reticle voltages and returns an object containing two vectors with the voltages before and after the FETs. In case of VDD5, VDD12 and VDD25 the second vector contains 0.00 values and the data in the first vector delivers differential voltages amplified by a factor of 50. The vectors are called v_mon (before FET) and v_fet (after FET).
- Support.get_psb_voltages()

This reads the powerboard voltages. As well as the function above it contains two vectors. They are called board0 and board1 and deliver the respective data.

- Support.get_system_temperatures()
  This reads the temperature data. It returns an object containing a single list called systemps.

The long function names and thus also function calls are necessary because of the different namespaces in pyhalbe.

4.2.1.3 Root elements

Tkinter provides widgets to use in a GUI. All these widgets have to be placed inside a root widget, which has to be initialised at the very beginning of the code. The root widget instance runs inside the mainloop, from which all visualisation tasks have to be done, if carried out inside a thread. This is not only a security issue in the programmer's interest, but the Python interpreter throws an error if tried in a different way.

4.2.1.4 runtest()

This function tries to read out a single reticle via

Support.get_reticle_status(Coordinate.DNC(geometry.EnumGlobal(i))), where i is a DNC number.

If it fails, the status label turns red.

4.2.1.5 start()

start() is executed by clicking the start button. It initialises loggers, if logging is enabled and starts a thread to continuously update data (update() without or lm.update_log() with logging enabled) and one to visualise it (show_data()). Clicking the start button destroys the button itself and places a pause button instead.

If the programme fails to establish a connection to the board, clicking the start button causes an error popup.

4.2.1.6 show_data()

This function is started as a thread. It shows a constantly updated image of the retrieved data. To do so, it clears the figure widget (instead of creating a new one) and accesses the global dictionaries optimum_U and dev_U from the supporting functions module to get optimum and deviation values for the voltages. Then it creates correspondent colour maps using this data and displays the current voltages. The information, which data is to be displayed, is retrieved by calling get() on the dropdown menu called var in the retviz/iretviz module.

The data read by the board is stored in the global variable ret, which is a list of lists containing vectors and represents a wafer map. Every reticle is a list containing two vectors with the data before and after the FET. (Or in case of differential voltage measurement one vector with data and one null vector.) To fill the ret wafer map, v_mon() (before FETs), v_fet() (after FETs) or in case of currents, current() is called. (For details on wafer maps see *4.2.2.3 wafer maps*)

4.2.1.7 class event_handler()

This is an only rudimentary nice-to-have functionality implemented as one of the last steps during development and not completely debugged. Its code was taken from a similar example from the internet and reads the canvas variable and its exact value. Bugs are that this does not work with every click and it does never work for the bottom line and for the lower left edge. The click issue probably has to do something with the GIL and the thread state. The problem with the bottom line seems to be with the Python source code and the size of the array, as the error retrieved said something about an index being out of range (i <= 7) and the bottom line has index 8. Maybe it could be fixed by accessing the Python source code, but this was not done as it would only affect the retviz development environment and not every running environment.

The bug with the lower left edge is not understood at all.

4.2.1.5 set_position()

This function sets the global variable position to the value chosen by the dropdown menu and indicates the choice by setting the respective button's colour to green and the others to the GUI background's colour. To do so with the help of a loop, there is a list containing the three buttons. The position integer is an index for this list, so it allows to address the buttons inside. set_position() is not placed in the supporting_functions module, because it contains GUI configuration.

4.2.1.6 log_button_click()

Similar to set_position() it sets the global variable, which indicates, if logging is enabled or not. Before the start() function starts an update() thread, it checks, if logging is enabled and according to this check's outcome it starts udate() from the supporting_functions module or it starts update_log() from the logmodule module. It is not placed in the supporting_functions module, because it contains GUI configuration.

4.2.1.7 set_pause()

Similar to the two functions above it sets the global variable pause, which indicates, if the update thread pauses. If so, the button turns yellow and an also yellow pause label appears inside the visualisation canvas. The global variable pause is read by the

update() function (or update_log(), if logging is enabled). While the update pauses, changing the displayed data via the dropdown menu is still possible, as the whole wafer data is stored within the ret variable. It is not placed in the supporting_functions module, because it contains GUI configuration. Although the updating thread does not do anything, when paused, changing between voltages takes longer than with a running update thread. An explanation for this phenomenon could be, that it takes less time to "execute" the updating thread's code, when it skips most of it, so it is even more difficult for other threads to obtain the GIL.

4.2.1.10 powershow()

This function is started as a thread. It initialises the labels and frames on the right side of the GUI which show the powerboard values and updates the colours afterwards, depending on the values retrieved. To show new data, an update thread has to be running. The framcolour is set by calling framecolor('<voltage name>', <board index>), which evaluates the powerboard voltage <voltage name> to set the colour to green, if inside the allowed range and to red otherwise.

4.2.1.11 set_temps()

Clicking the temperature button calls set_temps(). Similar to the other set_x() functions it sets a global variable t, which indicates, if the temperatures are updated and shown or if they are not. Depending on t, it places or removes the correspondent labels and reads the temperature data. After setting the label's positions, set_temps() starts a thread with argument temps() to continuously update the temperature. With the configure statement the labels are updated as well.

4.2.1.12 temps()

This function is started as a thread by set_temps() to update the temperature data. It reads the data via

Support.get_system_temperatures().systemps

and displays the values shortened to three digits after point. The function contains a time.sleep(5) statement to prevent the GUI from being blocked completely by the temperature readout. Nevertheless it is blocked for the time it takes to read the temperature data (about 5 – 7s) every five seconds.

4.2.1.9 labels, frames, alltime threads

Many lines of code only initialise or configure labels or frames for the GUI. There is no other way to do this than doing so for every single label or frame individually, as they all have different names. Creating variables dynamically is "dangerous" (not only in

Python) and it would be quite complicated to access the variables other than explicitly in this case.

As mentioned above, Tkinter runs its root widgets inside mainloop(), which executes everything GUI related. It starts right after executing 'python retviz.py' (or 'iretviz,py').

Two other threads starting right away are the runtest() thread, which tests, if the board connection is established, necessary to read data, and the powershow() thread, which shows, if the powerboard voltages surpass the allowed deviations.

4.2.2 supporting_functions.py

As mentioned in the import comments above, this module exists mainly to take away code from the other modules. It contains some global storage variables, such as the wafer maps, the update() function without logging, a function to calculate the currents and the powershow() function mentioned above.

4.2.2.1 dictionaries, lists, global variables

At the beginning of the supporting_functions module some dictionaries and lists are defined. They are needed to switch between voltage names and indexes, as the passing of voltage names in the code is easier to follow, but the values read by the pyhalbe functions return lists. To get certain values from the lists, indexes are needed, hence the dictionaries. Some of the lists defined at the beginning of the code allow accomplishing tasks (e.g. logging) with the help of loops. To test, if the voltages are in their allowed ranges or to set the colourmaps in general, the “optimum”-lists contain reference values, read from xml files by the function listfill(‘<filename>.xml’) from the valueport module. It is possible to set the allowed deviation for each voltage individually. The resistances for every reticle and every voltage are stored in a list and are read from an xml file as well.

4.2.2.2 update()

This function runs as a thread and updates the voltage data for the whole wafer via

Support.get_reticle_status(Coordinate.DNC(geometry.EnumGlobal(<DNC number>)))

in a loop, which calls the function 48 times. The data is stored in the ret variable The powerboard voltages are updated via

Support.get_psb_voltages().board0 or Support.get_psb_voltages().board<boardnumber>,

which directly accesses the correspondent part of the element returned by the function.

4.2.2.3 wafer maps: ret, v_mon(U), v_fet(U), current(U_name)

Storing the acquired data in convenient variables was one of the central tasks to be accomplished during the development of this programme. Calling

Support.get_reticle_status(Coordinate.DNC(geometry.EnumGlobal(<DNC number>)))

returns an object, which contains two vectors. One of them contains the voltages before the FETs (v_mon), the other contains the voltages after the FETs (v_fet). The update() function reads these double vector objects for every reticle on the wafer and writes them into the first wafer map called “ret”. With the dropdown menu and the three buttons below it one can choose, which one of the values (pre/post FETs or currents) is to be displayed. By doing so, the global variable “position” is set to the chosen value and the show_data() thread updates the wafer image accordingly. This happens using one of the functions v_mon(U), v_fet(U) or current(U_name) from the supporting_functions module, which extract the chosen values from the “ret” wafer map to the next wafer map, which contains only the chosen values (e.g. VOL before the FETs). The process is shown below with the help of three cuttings from the wafer maps.

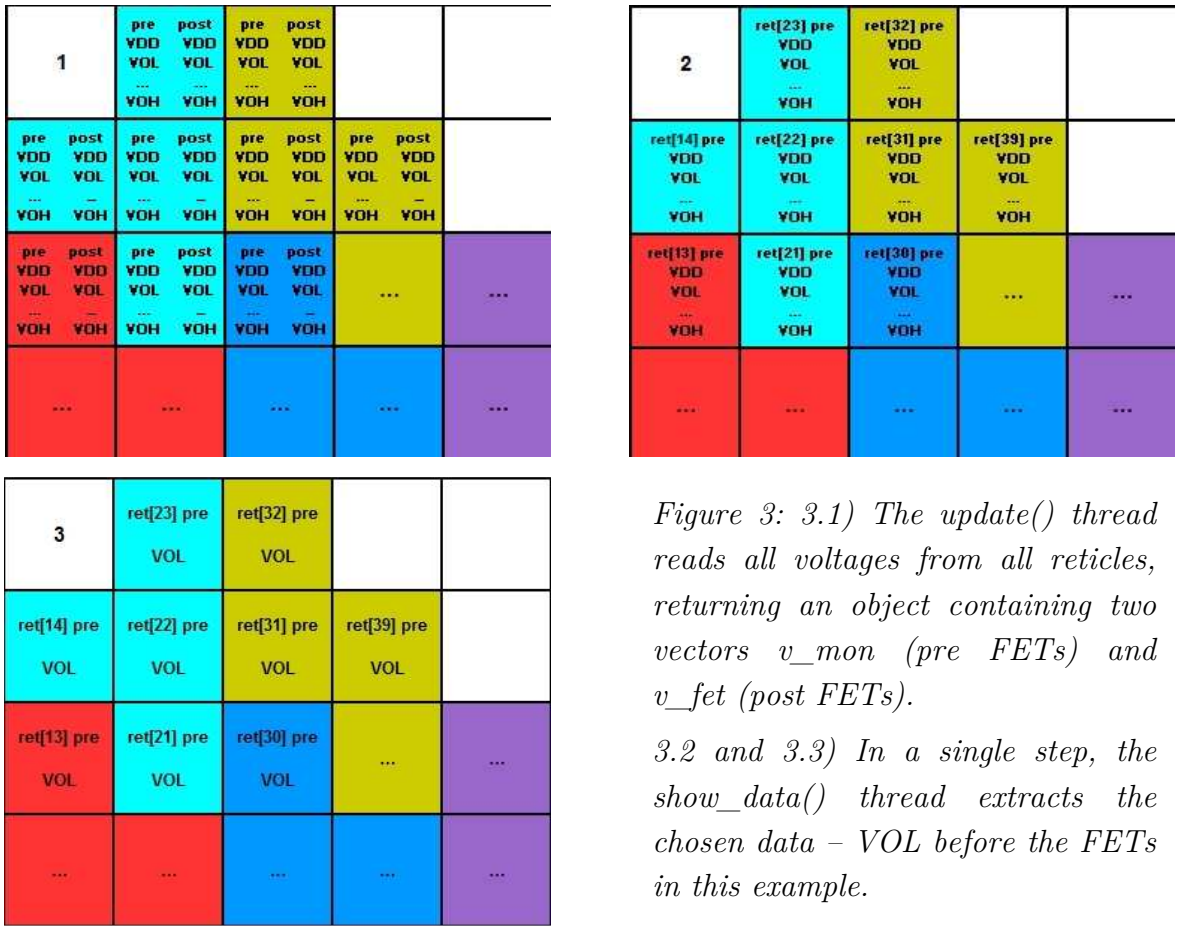

Figure 3: 3.1) The update() thread reads all voltages from all reticles, returning an object containing two vectors v_mon (pre FETs) and v_fet (post FETs).

3.2 and 3.3) In a single step, the show_data() thread extracts the chosen data – VOL before the FETs in this example.

In fact, the wafer maps are lists consisting out of lists. The NumPy module is able to create an array out of these wafer map objects, which can be visualised with the help of matplotlib. The white spaces in the corners derive from nan values, which are NumPy's Nonetype.

4.2.2.4 cur(i, U_name)

This function calculates the current for the reticle with number i for the voltage U. The resistances needed for the calculation are read from an xml file by the function listfill('<filename>.xml') from the valueport module. The if clause takes into account, that some of the voltage data is measured differentially with a voltage amplifier with a factor 50.

4.2.2.5 framecolor('<voltage name>', <board index>)

This function checks, if the powerboard voltage value is in the allowed range or if the deviation is too big. Depending on the check's outcome it returns green or red, which sets the indication frame's colour on the right side of the GUI.

4.2.3 omnivolt.py

The omnivolt module creates the overview image, which shows all voltages of all reticles. In case of retviz.py the overview appears in a new window, in case of iretviz.py it appears inside the main window, where the single voltages are showed.

4.2.3.1 arrays

In both cases two nested loops (one for v_mon, one for v_fet) fill 48 3x4 arrays with the data. mon as well as fet are lists of these arrays, so both of them contain 48 of these 3x4 arrays, if omnivolt(c) is executed. (c takes 'mon' or 'fet'.) As it is only possible to set one colourmap for one visualisation, every value has to be checked individually, if it is inside or outside the allowed range.

4.2.3.2 subplots and visualisation

The easiest way to arrange the reticle arrays is using subplots, because they are positioned automatically. In case of retviz.py the distribution to subplots is done in omnivolt.py itself, which is no problem in terms of possible mainloop() conflicts with visualisation updates, as a new mainloop() instance is created, which displays the overview. In case of iomnivolt.py the subplot distribution has to be done in iretviz.py, because the overview appears inside the main window and adding subplots needs to address the figure element, which is defined in iretviz.py.

The iretviz version updates the overview continuously by calling iomnivolt.py and overview(arr) in every cycle of the show_data() thread. overview(arr) is only defined in

iretviz.py, not in retviz.py. This function takes the overview array returned by iomnivolt(c) and distributes the arrays to the subplots.

4.2.3.3 explanation and closure

In both cases retviz.py and iretviz.py an explanation appears in the upper left corner of the overview. It explains, which field of the 3x4 array shows which voltage.

In case of retviz.py and omnivolt.py an additional function close() is implemented, which closes the overview window by destroying the root widget.

4.2.4 cords.py

To show, which reticle is which, coordinate number labels are placed in all visualisations. The reticle numbers as well as the DNC numbers are shown in the two images below. (They can be found online following gitviz – projects – wsi-retviz – repository – reticle_numbers.ods/reticle_numbers_DNC.ods) Inside the programme a "conversion" between these numbers has to take place, as the readout via Support.get_reticle_status(...) delivers the reticles with DNC coordinates, which differ from the reticle numbers as can be seen in the two documents above as well.

Actually, there is no real conversion. The Support.get_reticle_status(...) uses the DNC coordinate system defined in pyhalbe. (Left picture below – actual wafer rotated by 45° to the right.) The numbers shown on the right side are placed on top of the visualisation as labels to deliver the numbers used whilst hardware development.

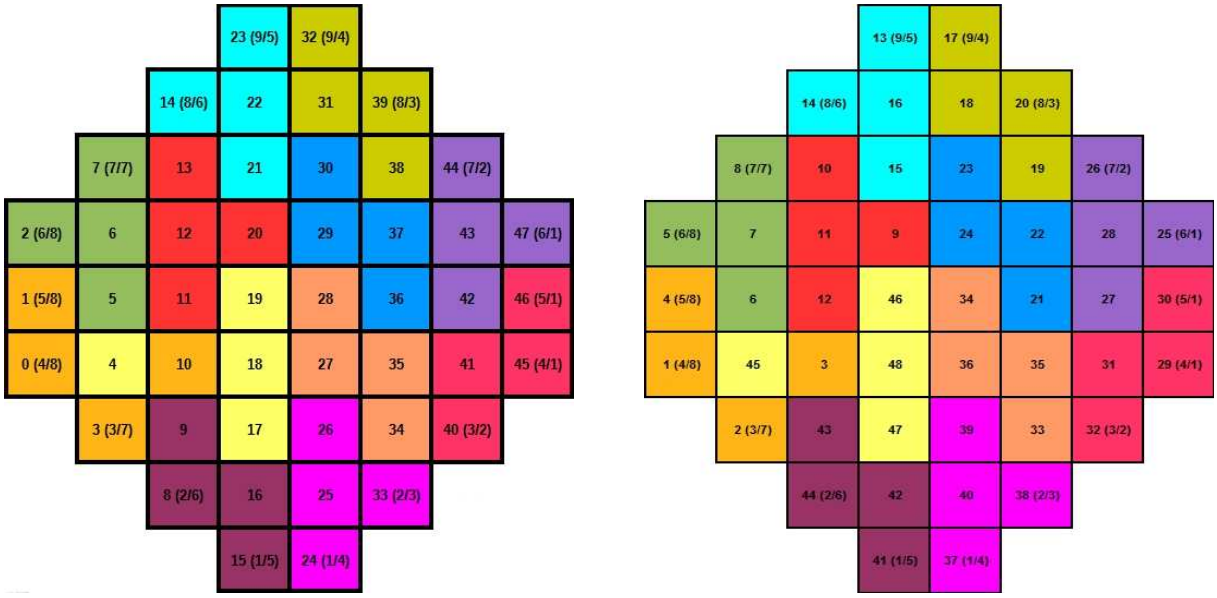

Figure 3: DNC numbers on the left and reticle numbers on the right. The (x/y) at the borders are array coordinates. The wafer is turned around 45 degrees below the bracket.

4.2.4.1 global variables

At the beginning of the code, some global positioning variables are defined. This allows an easier positioning of the labels and also an easier change of the positions, in case the size of the programme window is changed lateron. In case of cords.py there are global variables for the single voltage's reticle visualisation and for the powerboard labels and frames. In case of icords.py there are further global variables for the overview labels.

4.2.4.2 build_cords(), set_temperaturelabels()

This functions only initialise the labels. The positioning happens with the help of another function, as with icords.py, the positions have to change from time to time. In icords.py the explanation for the overview is initialised here as well.

4.2.4.3 set_cords(), set_powerlabels(), set_powerframes(), set_overcords()

set_cords() positions the reticle number labels. set_powerlabels() and set_powerframes() initialise as well as position the labels and frames for the powerboard voltages, as they do not change in position.

set_overcords() is an additional function in icords.py, which moves the reticle number labels, so their position fits the overview reticle layout and positions the overview explanation. set_cords() in icords.py removes the explanation and moves the reticle number labels from iverview position to “normal” position, if the programme switches back to single voltage mode.

4.2.5 valueport.py

Optimum values for voltages and resistances are delivered in xml files. The valueport module reads these files and creates list/dictionary objects, which can be handled by Python. The only thing important about the functions is the structure and the correspondent indents. The valueport functions are called in the supporting_functions module. Calling listfill(filname) is used for the resistances xml file and reads the whole file at once. dictfill(filename, value) is used for the optimum values file and takes the 'value' specification to read only certain parts (voltages, powerboards, deviations, …) of the optimum values xml file.

4.2.6 logmodule.py

As all values are logged to individual files, if logging is enabled, all loggers have to be initialised individually as well. This is implemented in log_init(). update_log() is called instead of the update() function without logging, if logging is enabled and logs() is the function, which writes into the logfiles after every update.

4.2.4.1 update_log()

This function does the same as update(), except it calls logs() after every update. It was faster and simpler for the programme's execution to write two functions to distinguish between updating with logging and updating without logging than writing a single function which checks in every cycle, if logging is enabled or not, via an if clause. (Details on possible changes in 3.2 Functionality – fourth paragraph.)

4.2.4.2 logs()

As mentioned above this function is called after every update by update_log() and makes all loggers write to their respective logfiles.

4.2.4.3 format_temp()

This puts the temperature data in a format conveniently to read in the logfiles. There are two lines between every entry because temp_format() inserts one and appending a new log insers another.

4.2.4.4 format_power(i)

The format_power(i) function makes the powerboard logfiles easier to read. All values for one powerboard i = 0 or i = 1 are stored in one logfile for this powerboard. format_power(i) writes the values one below the other instead of in a single row and it writes the voltages name in front of the value to guarantee further clarification.

4.2.4.5 log_init()

After initialising the loggers, they are written in lists, so logs() can access all loggers via 'for' statements and does not need to address every logger individually. Every logger needs to be initialised and as a format different from the standard settings was chosen for the logfiles, the settings have to be specified.

5 Conclusion

It was possible to create this comprehensive GUI with the rather simple means of the Tkinter module and a few more Python modules. This is an advantage, as none of the modules are difficult to obtain or complicated to install, which guarantees easy usage and availability on many systems and does not use many system resources.

A small disadvantage of the Tkinter module is the fact, that all buttons, labels etc. have to be instantiated and accessed individually, which brings along a certain amount of work for a programmer.

The most severe problem of the retviz programme is Python's global interpreter lock (GIL). To solve this problem completely, a different language than Python has to be used to re-implement the whole programme. That the GIL would be a problem, was observed not before the work on the programme was almost finished and it became most obvious, when the temperature display and the overview-included version were implemented. At that time it would have caused too much work to switch to another language.

Using only the retviz version (as opposed to the iretviz version) reduces the GIL problem to the temperature display. One way to reduce this problem even further and to allow a pleasant use of all functionalities would be to change the implementation of the temperature readout on the ARM board.

6 Sources

- *http://wiki.python.org*
- *http://www.dabeaz.com/python/GIL.pdf*