Bachelorarbeit

Graphical Control Interface for TDDS Measurement Framework

ausgeführt am

Institut für Mikroelektronik

zum Zwecke der Erlangung des akademischen Grades eines

Bachelor of Science (BSc)

unter der Leitung von

A.o. Univ.-Prof. Dipl.-Ing. Dr.techn. Tibor Grasser

und

Dipl.-Ing. Dr.techn. Michael Waltl

ausgeführt durch

Andreas Berger

Matrikelnummer 01228791
Grodnau 102
7433 Mariasdorf

Wien, im November 2017

Andreas Berger

Abstract

The time-dependent defect spectroscopy (TDDS), evolved at the Institute for Microelectronics, is an experimental method to study single defects in metal-oxide-semiconductor field-effect transistors (MOSFETs). These defects are unavoidably introduced during device fabrication and can capture and emit a single charge, thereby causing a drift of the threshold voltage of the transistors. Furthermore, these defects are responsible for aging of p-channel and n-channel MOSFETs. To measure the drift of the threshold voltage a dedicated TDDS setup and sophisticated data analysis algorithms have been developed. Within the TDDS framework the measurements are controlled by a jobserver which is executed on a measurement PC. This jobserver controls the TDDS measurement box, probestations, furnaces and further required periphery. However, the current implementation suffers from complex textual measurement scripts which are required to configure and to control the measurement sequence. To simplify the control, the evaluation and the visualization of the measurement process, a Graphical Control Interface (GCI) is designed. It allows to control the jobserver via SSH connection and can be executed either on PCs or on mobile devices. The GCI is implemented using Python which is quite challenging as it turned out that the support for the required Python modules is not entirely provided for the development of comprehensive GUI applications, especially applications to be compatible with several operating systems like Ubuntu, Windows, Android and iOS. Using the cross-platform Python framework Kivy, a GCI could be developed which runs on personal computers using Linux or Windows and on Android based mobile devices.

Kurzfassung

Die am Institut für Mikroelektronik entwickelte sog. Time-Dependent Defect Spectroscopy (TDDS) ist eine Methode zur experimentellen Untersuchung einzelner Defekte in Metall-Oxid-Halbleiter-Feldeffekttransistoren (MOSFETs). Solche Defekte treten dabei unvermeidbar bei der Herstellung von Transistoren auf. Durch elektrische Lade- und Entladvorgänge einzelner Defekte kommt es unter anderem zur Drift der Schwellspannung von MOSFETs und sind somit eine wesentliche Ursache für die Alterung von p-Kanal und n-Kanal Transistoren. Um die Drift der Schwellspannung zu messen wird ein eigens für diesen Zweck entwickeltes TDDS Messgerät in Kombination mit komplexen Algorithmen zur Datenanalyse verwendet. Innerhalb der TDDS Messumgebung arbeitet ein auf einem Messcomputer laufender Jobserver welche die Messabläufe exakt steuert. Der Jobserver steuert die TDDS Messbox, die Probestations, die Öfen und weiteres benötigtes Equipment. Der Nachteil der derzeitigen Implementierung ist, dass komplexe Messskripte zur Kon guration und Steuerung des Messablaufs notwendig sind. Um die Steuerung sowie die Visualisierung des Messprozesses zu vereinfachen wird ein Graphical Control Interface (GCI) entwickelt. Dieses Interface ermöglicht es den Jobservers über eine Remote Verbindung via SSH verzuwarten und kann sowohl auf PCs als auch auf mobilen Geräten betrieben werden. Das GCI wird in der Skriptsprache Python entwickelt. Die Wahl dieser Programmiersprache stellt eine Herausforderung dar, da es wie im Zuge der Programmierung ersichtlich wurde, viele benötigte Python Module die Entwicklung komplexer GUI Anwendungen derzeit noch nicht unterstützen. Dies ist speziell der Fall, wenn diese Anwendungen auf mehreren Betriebssystemen wie Ubuntu, Windows, Android und iOS stabil laufen sollen. Durch die Verwendung des Python frameworks Kivy wechles mit verschiedensten Plattformen kompatibel ist konnte ein GCI entwickelt werden, welches auf PCs mit Linux oder Windows sowie auf Android basierten mobilen Geräten verwendet werden kann.

Contents

1 Introduction

The lifetime of modern semiconductor transistors is seriously affected by Bias Temperature Instabilities (BTI) and Hot Carrier Degradation (HCD). Both effects are observed in p-channel and n-channel metal-oxide-semiconductor transistors (MOSFETs) and significantly reduce the time-to-failure of these devices.

BTI and HCD can be ascribed to charging and discharging events of single defects which are located at the semiconductor/oxide interface or directly in the oxide. These defects reduce the subthreshold slope, decrease the on-current and cause drifts of the threshold voltage and thus can hamper the stable operation of MOSFETs. Quite remarkable, even at nominal operating conditions a significant drift of the threshold voltage can be observed in mainstream silicon technology. This drift is much higher for SiC MOSFETs and for more exotic transistors based on 2D materials. In order to counteract the device aging mechanism and further enhance the long-term performance of MOSFETs, a detailed knowledge of the physical origin of these mechanism is vital. To deepen the understanding and to study the impact of charge trapping for any transistor related technology, a special custom-made measurement box has been developed at the Institute for Microelectronics. The measurement sequences, which are applied to the devices under test, are controlled by elaborate test scripts which use a special configuration file format. Using such complex configuration files offers on one hand high flexibility for defining the dedicated measurement sequences, however on the other hand their usage is sometimes very inconvenient.

Considering the current state of the TDDS framework, controlling the measurement setup without a detailed knowledge of the powerful software toolset is currently difficult. Therefore, a graphical control interface (GCI) would provide a very convenient way to control the measurement system without a profound knowledge of the overall background processes involved. Within this bachelor thesis a GCI for the TDDS measurement framework, which is available at the Institute for Microelectronics, is developed. The GCI allows to control single measurements and the measurement flow via mobile devices (i.e. smart phone, tablet, etc.) or PC.

The time-dependent defects spectroscopy (TDDS) and the structure of the previously mentioned corresponding TDDS framework is briefly introduced in this chapter. Next, the preparation and research, done to develop a suitable software tool which can be used on mobile devices and PCs, is discussed. Afterwards, the implementation of the GCI, showing the functions provided by it and the challenges which occurred during the implementation, is presented. To illustrate the GCI's development in Python, parts of the code will be described. Finally, a brief outlook for further development and optimization of the GCI is given.

1.1 Impact of Defects on MOSFETs Threshold Voltage

Imperfection of the ideal structure within the gate oxide of a transistor or the lattice mismatch at the interface of different materials, as it is the case for the semiconductor/oxide

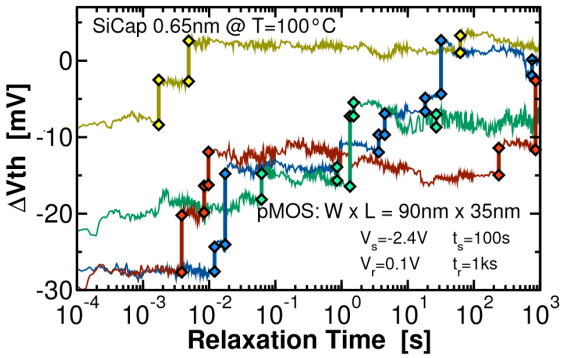

Figure 1: In a 90nm x 35nm small SiGe p-channel MOSFET, fabricated with a high-k gate stack, single charge transitions can be clearly observed. Each discrete step corresponds to a defect emitting its charge. As visible, each defect produces its own step height, that is the impact of a single transition on the $dVth$, at a different emission time.

interface in MOSFETS, can lead to electrically active sites. Such sites can capture or emit a single charge and are considered as defects in MOS transistors. Exactly these defects are the main culprits for drifts of the threshold voltage, reduction of the subthreshold slope and reduction of the on-current in these devices. The total number of defects which are prevalent in a single transistor depends on the gate area A, which is the product between the gate width W and gate length L, $A = W * L$ (in planar devices). Thus, the smaller the device becomes, the smaller the number of defects, which can contribute to the drift of the threshold voltage, gets. Quite interestingly, the impact of a single defect on the total dV_{th} shows the opposite trend. The smaller a MOSFET becomes the larger the impact of a charge transition of a single defect on the total dV_{th} gets. Thus, by using scaled MOSFETs, single charge transitions can be individually studied as the transitions become so large that they can be monitored as discrete steps in the drain current, see **Figure 1**. The only prerequisite for a single defect to be visible in a single recovery is that the defect is charged before. So the device is typically stressed for a certain time at a certain gate stress bias to charge the defects. For the sake of completeness it has to be mentioned that the BTI is further classified into negative BTI and positive BTI. NBTI is considered as BTI after a negative gate bias stress has been applied, which is usually the case for p-channel MOSFETs, and PBTI refers to positive bias stress and is typically studied employing n-channel MOSFETs. In the literature the former is mostly studied, because NBTI is reported to be more pronounced than PBTI. Thus, NBTI is more popular to be investigated, although both effects are of equal importance.

1.2 The Time-Dependent Defect Spectroscopy

The time-dependent defect spectroscopy (TDDS) has been proposed by Tibor Grasser and Hans Reisinger in 2010 and is used to study the charging and discharging behavior of single defects in Si-MOSFETs. This method benefits from the aggressive scaling trend of the semiconductor industry which provides MOSFETs in the sub 100nm regime and enables a unique opportunity to analyze single defects at a great level of detail. To explain the intricate charge capture and emission times, which are the parameters which are typically collected for each defect at various temperatures, stress times and biases, sophisticated charge trapping models in combination with state-of-the art device simulators are used. These simulations enable to estimate the trap position and trap energy levels which are of further interest for device engineers and physicists to enhance the MOSFETs performance

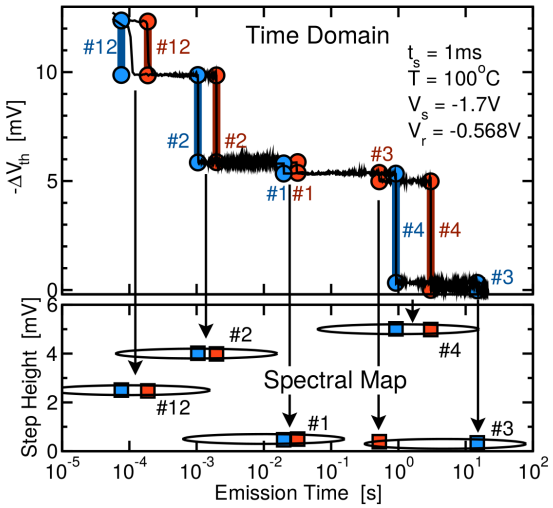

Figure 2: Two recovery traces of a nanoscale p-channel MOSFET are shown, exhibiting the discrete nature of the charge transition events in small devices. Each discrete step corresponds to a single charge transition. When the single steps are plotted in the step height vs. emission time plane, the emission events of the defects form clusters when repeating the same experiment several times. These clusters, which are the fingerprints of the individual defects, are thoroughly analyzed in research studies. It must be noted that the defects, which are visible during such an experiment, are unique for each device. Another transistor contains other defects, producing different step heights at different emission times.

and the time-to-failure.

The measurement principle of the TDDS to characterize the impact of such defects on the threshold voltage is very simple and relies on stress phase, during which the defects can become charged, and recovery phase, during which the defects can emit their charge. These phases are repeatedly applied to the devices under test, see **Figure 1**. As previously mentioned, in case of nanoscale transistors the single charge transition events, occurring during the recovery cycles, become visible as discrete charge transitions. The time point when a discrete transition in the dVth occurs, indicating that a defect emits its charge, is simply called the defects emission time. In general, the defect's charge emission time is a statistical quantity and is calculated as the average emission time considering typically 100 stress and recovery cycles. The single emission times together with the corresponding step height of each measurement cycle is collected in the emission time versus step height plane, which leads to the typical spectral map, see **Figure 2**. The average emission time and the step height of a single defect are important parameters and are extracted at a certain temperature, after a certain stress time. Modifying one of the measurement parameters cause a change of the defects emission time. For instance, with an increasing temperature the emission time gets smaller, indicating that charge trapping is a strong thermally activated process. There are many more intricate dependencies of the emission time so far not explicitly explained. Finally, it has to be mentioned that the spectral map looks different for each transistor. An example for two spectral maps of SiGe p-channel MOSFETs, extracted at two different temperatures, is visible in **Figure 3**.

1.3 Control of the TDDS Measurement Framework

The TDDS measurement framework, developed at the Institute for Microelectronics, is built around the jobserver. The jobserver is a Python application which is executed on a personal computer. It communicates via SQL interface with a database where the measurement configuration of the next measurement tasks is stored. As visible in **Figure 4** there are many more components which are controlled by the jobserver. For instance, the jobserver provides an E-Mail or SMS service to inform the user if a job has been

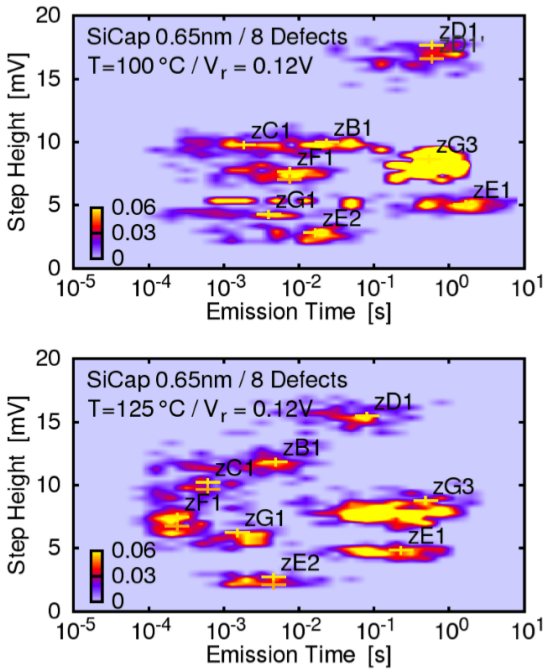

Figure 3: Eight defects have been identified in SiGe p-channel transistors. Each cluster in the spectral maps represents one single defect. As can be seen, with increasing temperature the clusters move towards lower emission time, indicating the temperature activation of charge trapping processes in semiconductors.

finished or an error during a measurement is performed. The jobserver controls an uninterruptible power supply, which is necessary for the system to operate stable even during errors in the laboratories power supply. The UPS is especially important for long term tests which last several months/years. If the system is operated in combination with a semi-automatic probestation, the jobserver triggers the device selection which allows to automatically measure many devices on a single wafer. And finally, the jobserver controls the TDDS measurement instrument (TMI) which performs the electrical measurements.
As already mentioned, the single measurements, simply called jobs, are organized in a queue which is administrated by a jobserver running on a personal computer. Each TMI has its own job queue, containing the measurement configurations which are sequentially processed until the queue is empty. Due to the missing graphical control interface the manipulation of the jobserver's single queues is very inconvenient. The only way to control the measurement was to directly access the SQL database of the jobserver, which requires detailed knowledge of the entire software tool set. A more convenient way to handle the jobqueue is by using a GCI which was developed during this thesis.

The GCI directly communicates with the jobserver's database which allows a real-time control of the entire measurement sequence, as visible in **Figure 5**. Furthermore, as the jobserver fetches the database in regular intervals (every few seconds) any changes to the database will be immediately noticed. For instance, setting the appropriate keyword within the database to abort the currently running job will be executed after the next update interval. Further possible manipulations, which should be applied to the database, are rearranging the job order, add new jobs to the database, remove jobs from the database and cancel a currently running job.

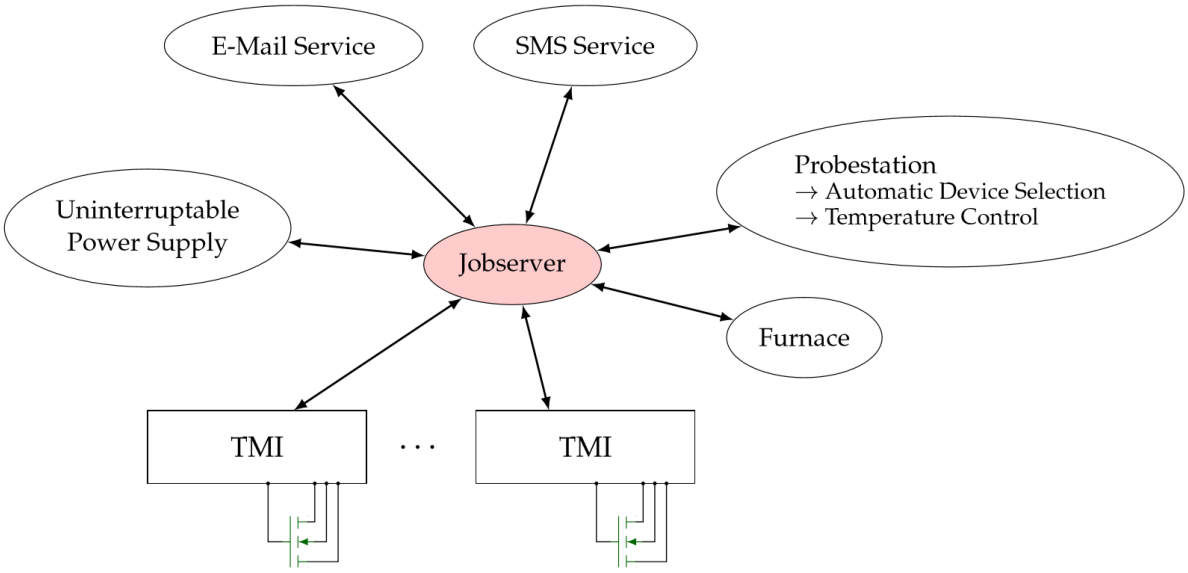

Figure 4: The jobserver provides an notification E-Mail and SMS service. Furthermore, the jobserver controls semi-automatic probe stations or a special furnace setup if connected to the system. Also, the TDDS measurement instrument (TMI) is controlled by the jobserver, taken from [26].

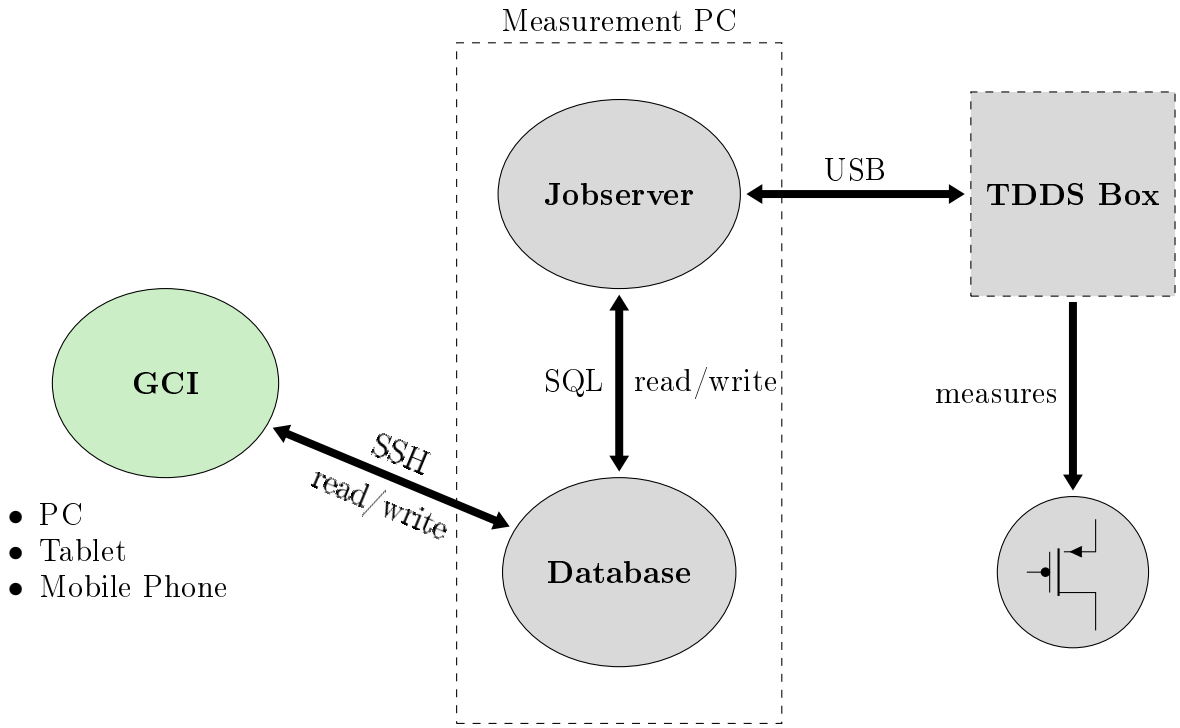

Figure 5: The jobserver, i.e. the central control unit of the TDDS measurement toolset, is executed on the measurement PC and controls the TMI and external periphery such as probestations and furnaces. The executed measurement configurations are stored in a SQL database, which is regularly fetched by the jobserver. To get real-time control over the entire setup the developed GCI directly communicates with the jobserver's database.

2 Prerequisites, Research, System Preparation and Installation

The following section gives an overview of the available software tools which have been tested to be used for the implementation of the graphical control interface.

2.1 Programming Language (Python2 vs Python3)

Although Python 2.7 will not be supported anymore in the nearest future, Python 2.7 is used for the GCI implementation. The existing tool set available at the Institute, which has been grown over more than 10 years, is written using Python 2.7. The main differences between Python 2.7 and Python 3 are changes in the core language, for example print() and exec() being functions instead of statements and integers using divisions. These changes have been made to make it easier to learn Python and be more consistent with the rest of the language. As expected, Python 3 is backwards incompatible to prior Python releases. Nevertheless, the used framework for the GCI development is Python 2.7.

2.2 GUI Framework

The first step before developing and coding the GCI is to decide which graphical framework/toolkit should be used. There are many different packages available, exhibiting limitations making them not all suitable for our GCI. During intensive research, the following frameworks/toolkits are tested:

- Qt(PySide,PyQt)
- Tkinter
- wxPython
- Kivy

2.2.1 Tkinter

Tkinter is the standard GUI package included in the Python standard library. Thus, this toolkit is convenient to use and perfectly compatible with Python. Tkinter can be used on most Unix platforms, also on Windows and Macintosh systems. It is the most commonly used GUI framework for Python. Most unfortunately, there is insufficient support for mobile operating systems like Android and iOS, making Tkinter not suitable for the mobile GCI for TDDS measurement control.

2.2.2 wxPython

A simple and easy way to create programs with stable and feature-rich GUIs is called wxPython. It is a GUI toolkit implemented as a Python extension module which wraps the C++ wxWidget cross-platform GUI library. Again the big handicap is, that there is currently no support for mobile operating systems.

2.2.3 Qt(PySide,PyQt)

Qt is a well known cross-platform application framework currently being developed by the Qt Company and the Qt Project. It is used for developing application software that can be run on a lot of different platforms with minimal changes of the code. Qt is widely used because of its professionality, providing speed and native applications with native capabilities. There are commercial and open source licenses available. PySide and PyQt are both Python bindings of Qt. A way to port your Qt applications to Android/iOS applications is Qt for Android/Qt for iOS. The fact that commercial licenses are required to get access to the full function range, supported our decision not to use Qt. Furthermore, porting Qt applications to mobile devices seems to be a rather sophisticated mechanism which will go beyond the scope of this thesis.

2.2.4 Kivy

Kivy is an open source Python library for rapid development of mobile apps and other multi-touch application software with a natural user interface (NUI). It is free and distributed under the terms of the MIT license, running the same code on Linux, Windows, OS X, Android and iOS. Every Kivy widget supports multi-touch. The framework is also used in commercial products and the toolkit is professionally developed, backed and used by the Kivy community and the Kivy organization. Kivy initially evolved out of the PyMT project. Kivy is based on Python and Cython, uses OpenGL ES 2.0 and has an extensive widget library. Cython includes the Python language and additionally supports calling C functions and declaring C types on variables and class attributes. For more complex applications the widget trees and bindings in Kivy can get confusing. That's where the KV Language, or also kvlang or kivy language, comes into play to counter this disadvantage. It allows you to organize your GUI in a declarative way. Another advantage is, that it also facilitates a good separation between the logic of the application and its user interface. Finally Kivy is selected to be the framework the GCI is developed around.

2.3 SQLite

SQLite is an embedded SQL database engine which runs SQL on local files, using locking to ensure that multiple processes don't open the database at the same time. It reads and writes directly to ordinary disk files. SQLite is not designed for network processes, because it does not have a separate server process. Thats why it's not directly comparable to client/server SQL database engines like MySql, Oracle and so on. SQLite provides local data storage for individual applications and devices. The complete SQL database is saved to a single database file. The format of this file is cross-platform compatible and written in a compressed storage saving format, which makes SQLite a popular database engine choice on memory constrained gadgets such as mobile phones, PDAs and MP3 players.

Most common commands used with SQL databases

The SELECT statement queries the SQLite database. This command does not change anything within the database. The result of this statement is the fetched data in form of

a result table, also called result sets.

The UPDATE statement modifies the existing records stored in the database. A WHERE clause is used to control the modification of the values in the rows. The UPDATE statement only affects these rows where the boolean expression of WHERE is true. If this statement is missing, the modification affects all rows. The list of assignments following the SET keyword in the UPDATE command determines the modification of the selected rows. This assignment must contain a column name and a scalar expression. This scalar expression will replace the original value in the database. Note that only the columns appearing in the list will be modified, the other columns stay unmodified.

The DELETE command is used to delete existing records from tables. Again, the WHERE clause enables the possibility to select single rows. If this clause is missing all records in the table will be deleted.

2.4 Geany

Geany is a fast and lightweight integrated development environment (IDE). It only requires a few other packages to operate independently from different operating systems and desktop environments. In principal, Geany is available for each platform which supports GTK libraries.

2.5 APK Packaging

In order to execute Python scripts on Android systems an Android Package Kit (APK) has to be built. The APK is the package file format used by the Android operating system. It is required for the installation of mobile apps. There are several ways to convert your Python file to an APK which works on Android.

2.5.1 Python-for-Android

Python-for-Android is a tool to generate packages from Python scripts and make these apps to be used on Android systems. The Python distribution, including the modules and dependencies used by the application, together with the code gets bundled in an APK. Creating packages this way has to be done very carefully, because the building process is completely configured by the developer. Using Python-for-Android requires a good knowledge of the Android system and therefore is recommended for Android experienced developers.

2.5.2 Buildozer

Another helpful tool to package mobile applications is called Buildozer. Instead of building every file manually, Buildozer automates the entire building process and downloads prerequisites like Android, SQLite, Kivy etc. itself. The detailed package list has to be specified in the Buildozer specification file. This file controls the build configuration and the parameters passed to python-for-android. Besides building the APK, Buildozer can also

download the APK to the mobile device, read logs from the device and run the application on the device.

The most important commands using Buildozer are:

- `$ buildozer init`

  creates a buildozer.spec file in the current directory.

- `$ buildozer android debug`

  creates an APK in the current directory. Make sure that the Python file, named main.py, and the buildozer.spec file are in the same directory.

- `$ buildozer android deploy`

  deploys the APK package to the device.

- `$ buildozer android run`

  runs the deployed package on the device.

It is also possible to combine several commands to be executed in a single command line, for instance:

```
$ buildozer android debug deploy run
```

To install Buildozer on Linux systems the following command sequence is recommended [2]:

- `$ sudo pip install --upgrade buildozer`
- `$ sudo pip install --upgrade cython==0.21`
- `$ sudo dpkg --add-architecture i386`
- `$ sudo apt-get update`
- `$ sudo apt-get install build-essential ccache git libncurses5:i386 libstdc++6:i386 libgtk2.0-0:i386 libpangox-1.0-0:i386 libpangoxft-1.0-0:i386 libidn11:i386 python2.7 python2.7-dev openjdk-8-jdk unzip zlib1g-dev zlib1g:i386`

2.6 Android Debug Bridge

Android Debug Bridge, often referred to as adb, is a command-line tool to handle the communication with Android devices. It is part of the Android SDK Platform-Tools. It supports device actions like installing and debugging apps, reading and filtering logs, and also provides a Unix shell with access to the device. The abd is a client-server based application and is based on three components [4]:

- **Client:** The Client is running on the development machine (e.g. Linux operating system) and sends commands to the device. The client is started with the terminal command "adb".
- **Daemon(adbd):** The daemon is used to run commands on the mobile device. Therefore, the daemon runs as background process on the device.
- **Server:** The server controls the communication between the client and the daemon. Like the daemon the server runs as a background process, but on the development machine.

After starting the adb client from the terminal, the client searches for already running server processes. If there are none, the client starts a new server process using port 5037 for the communication. After the server has been started it sets up a connection to available devices. When the connection is established, adb commands can be used. Note that, in order to use adb with a device connected over USB, USB debugging must be enabled on the mobile device.

The most common abd commands are:

- `$ adb devices`

  List all devices connected to the development machine.

- `$ adb logcat`

  Shows the log output from the connected device. To filter the log file one can use the “-s” option. For example: **adb logcat -s python** shows all logs linked to the Python app.

To install the abd on Linux systems the following command line instructions can be used:

```
$ sudo apt-get install android-tools-adb android-tools-fastboot
```

2.7 Kivy Launcher

A quick and simple way to test your Android app is provided by the Kivy Launcher. This widget is available in the Play Store available on Android devices. To run your apps with the Kivy Launcher, the Python code has to be stored on the mobile device in the corresponding folder which is linked to the Kivy Launcher. Some Python modules and

the appropriate Android permissions, for instance access to the Internet or access to the mobile phone speakers, are already included in the Launcher installation. These Android permissions included in the Kivy Launcher are restricted and therefore not enough for more complex projects. To guarantee full access to Android systems for projects like these, building your own APK is the better solution.

2.8 DB Browser for SQLite

During software development it turns out that a tool allowing a direct manipulation of the database is necessary for debugging purposes. During this work the sqlitebrowser, a lightweight open source graphical tool which is available within the standard Linux packages, is used. With this tool SQL databases can be lightgray!60!60 and single tables can be created, edited and deleted.

3 The Graphical User Interface

The graphical user interface basically consists out of four screens, using the screen manager described in the subsection Modules (4.1). These screens are connected to each other and provide a clear structure for the GCI. In the following **Figure 6** the screens which build the GCI are visualized. The arrows show the connections between them.

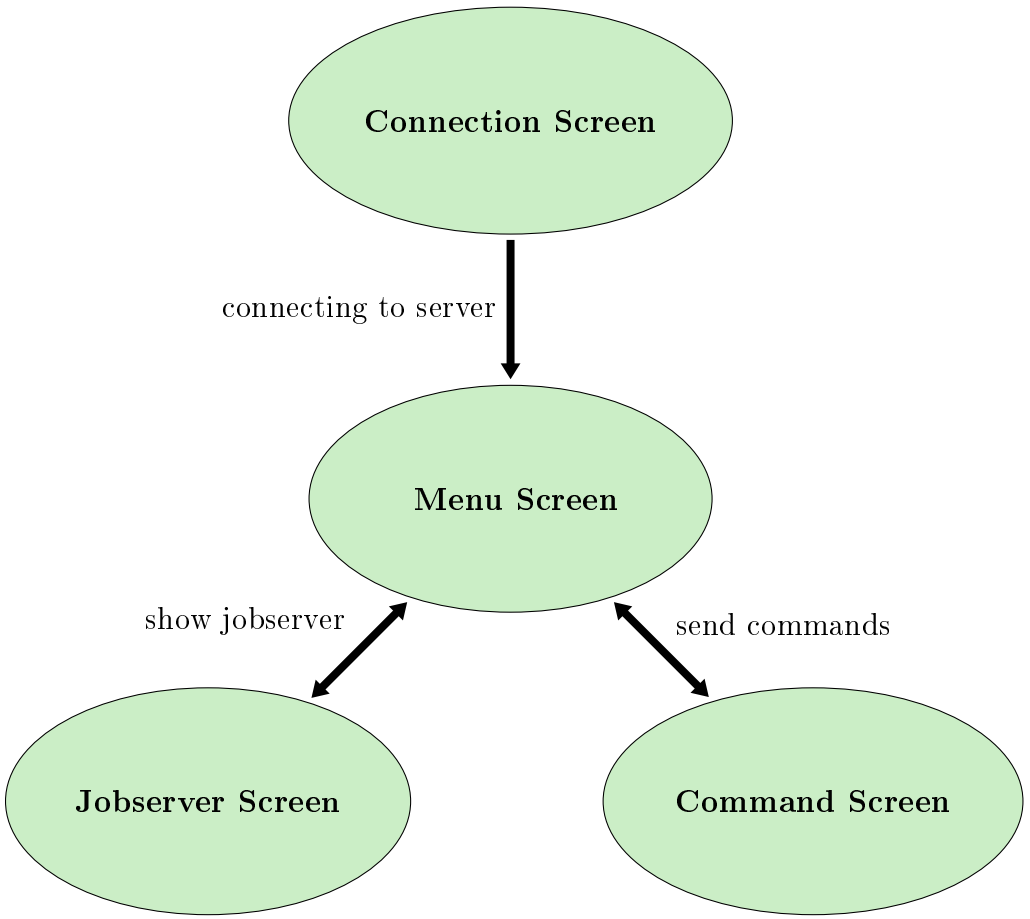

Figure 6: The interaction of the single screens is shown. The startup screen, called connection screen, enables the connection to a jobserver. After successful connection the menu screen will appear. This screen lists the possible functions provided by the GCI. From here the user can navigate to the jobserver and the SSHcommand screen. The jobserver screen provides a clear list of the queued jobs with the appropriate buttons for the manipulation. The SSH command screen lets the user send commands directly to the measurement PC.

The startup screen , see **Figure 7**, is the so called connection screen. It gives the user two options to connect to a jobserver which is running on the corresponding measurement PC. The first option is to select a jobserver using a dropdown list with a preconfigured set of standard jobservers. The second option is to establish a connection to a manually defined jobserver. The cancel button at the bottom is used to close the application.

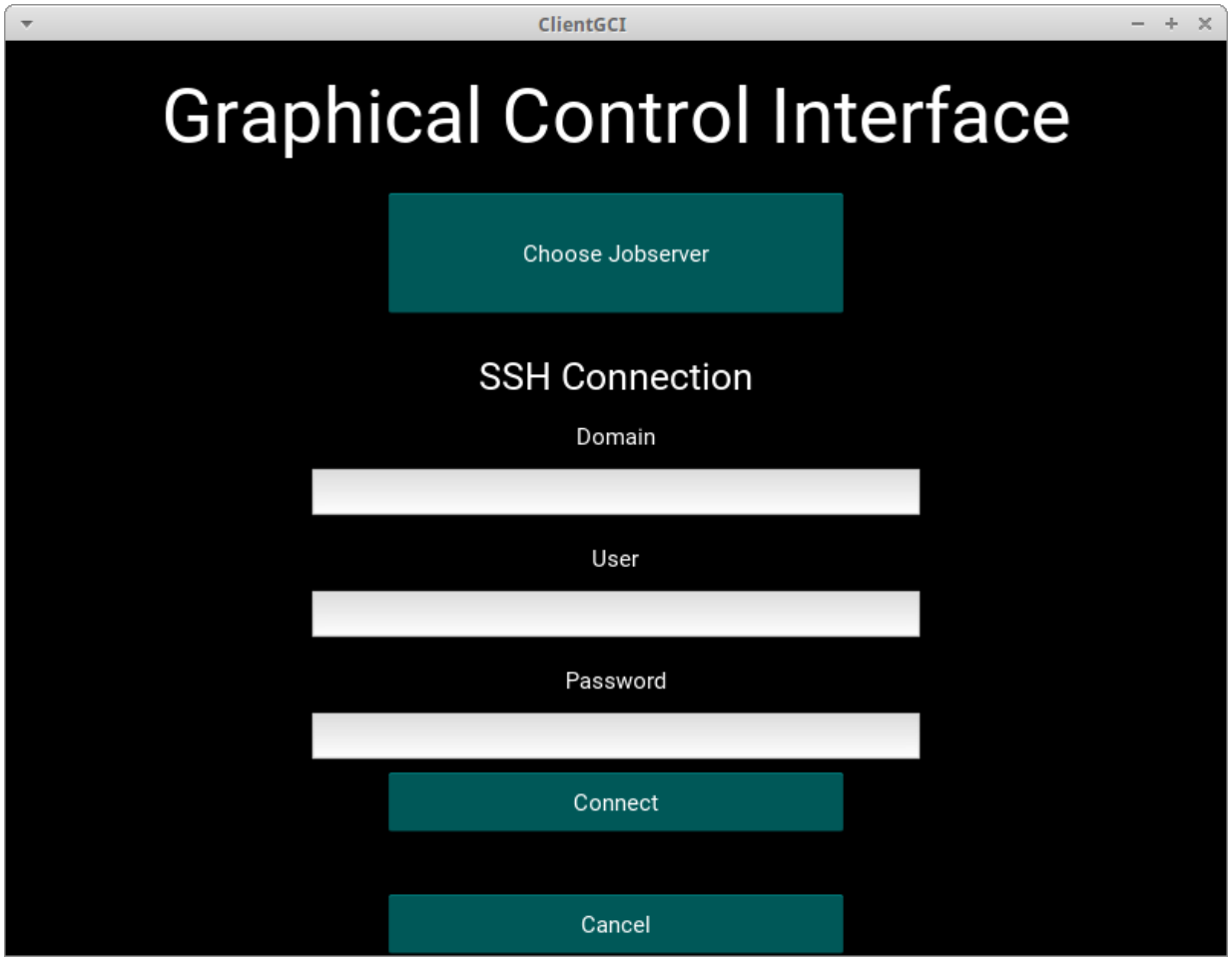

Figure 7: Startup screen for the GCI. The jobserver can either be selected from a dropdown list, or the address can be entered manually.

After successfully connecting to the jobserver, the menu visible in **Figure 8** can now be used to further control the measurement setup. The user can either continue directly to the jobqueue by using the Jobserver button, or can send SSH commands directly to the measurement PC.

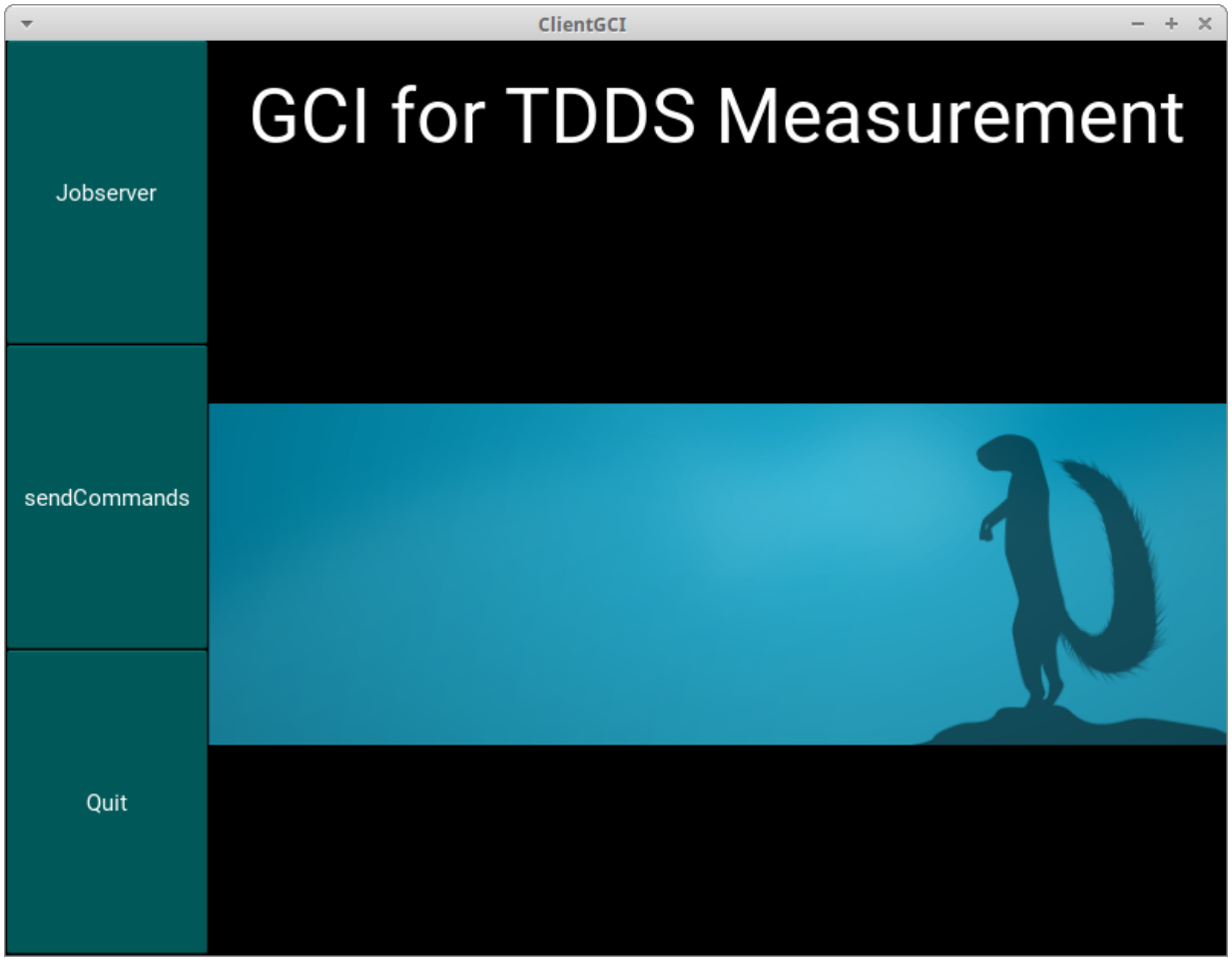

Figure 8: Menu screen for the GCI. From here the user can navigate to the functions provided by the GCI.

Figure 9 shows the GCI when the queue of the jobserver is going to be manipulated. The buttons on the left side of every row enable the control of single jobs. Deleting, editing and moving them around the queue, is supported as well as the cancellation of the currently running job. Using the menu located on the left side of the screen, the measurement queue ,which is currently shown in the GCI, can be selected using a dropdown list. By using the menu button one gets back to the menu screen.

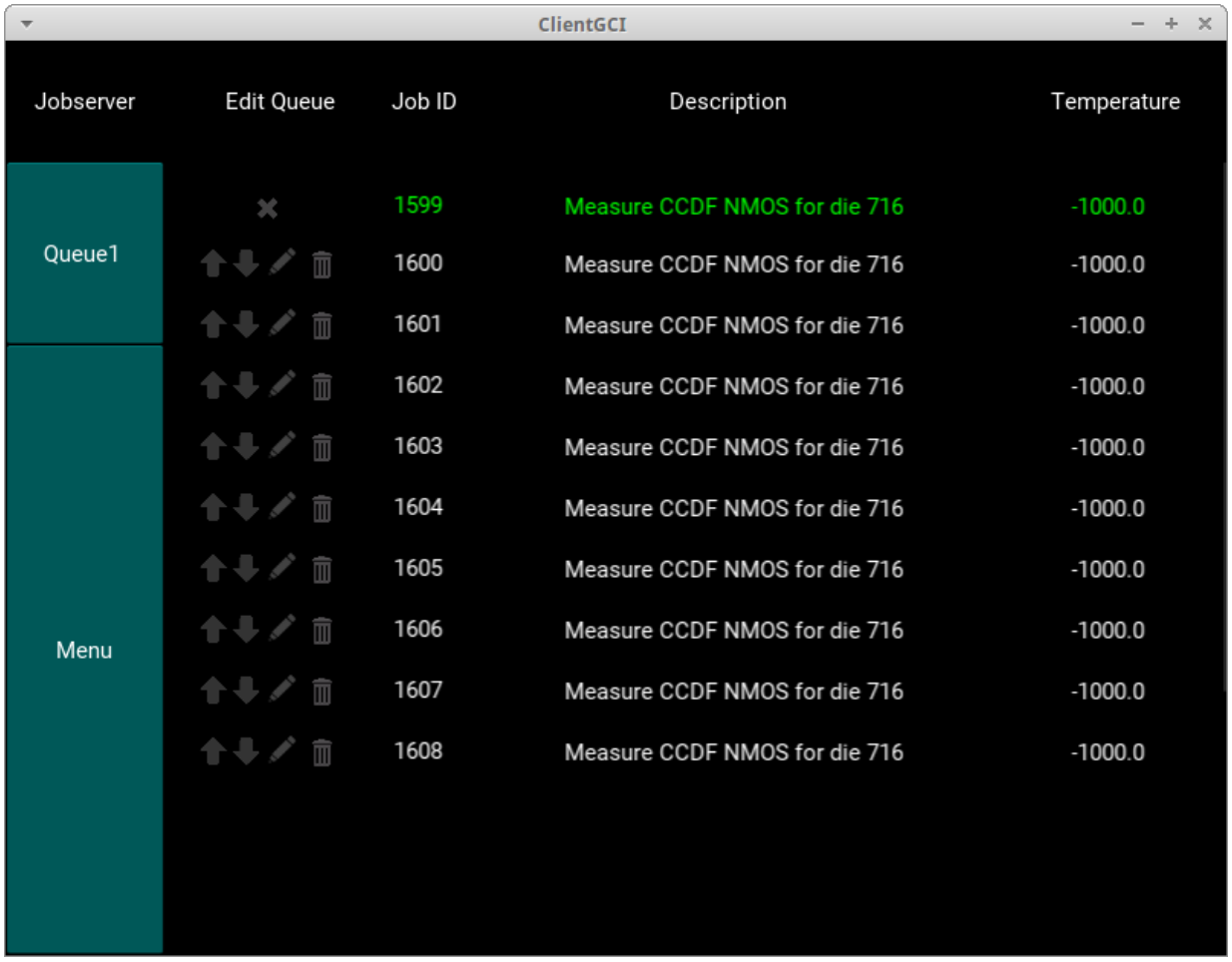

Figure 9: Jobserver screen for the GCI. The joblist of the currently selected queue is shown and enables the database manipulation using the buttons in the configuration lines. The temperature of -1000°C indicates that the job will be started ignoring the temperature of the device.

The SSH command screen visible in **Figure 10** enables the possibility to directly send commands using the existing SSH connection to the measurement PC. These commands are executed in the command line and can be used to call measurement scripts to fill the queue of the jobserver or to start the measurement. The commands can be directly written in the appropriate blank white field which is used for the textinput. Pressing the send button will transmit the command and execute it on the measurement host. With the menu button the user can go back to the jobserver menu screen from **Figure 8**.

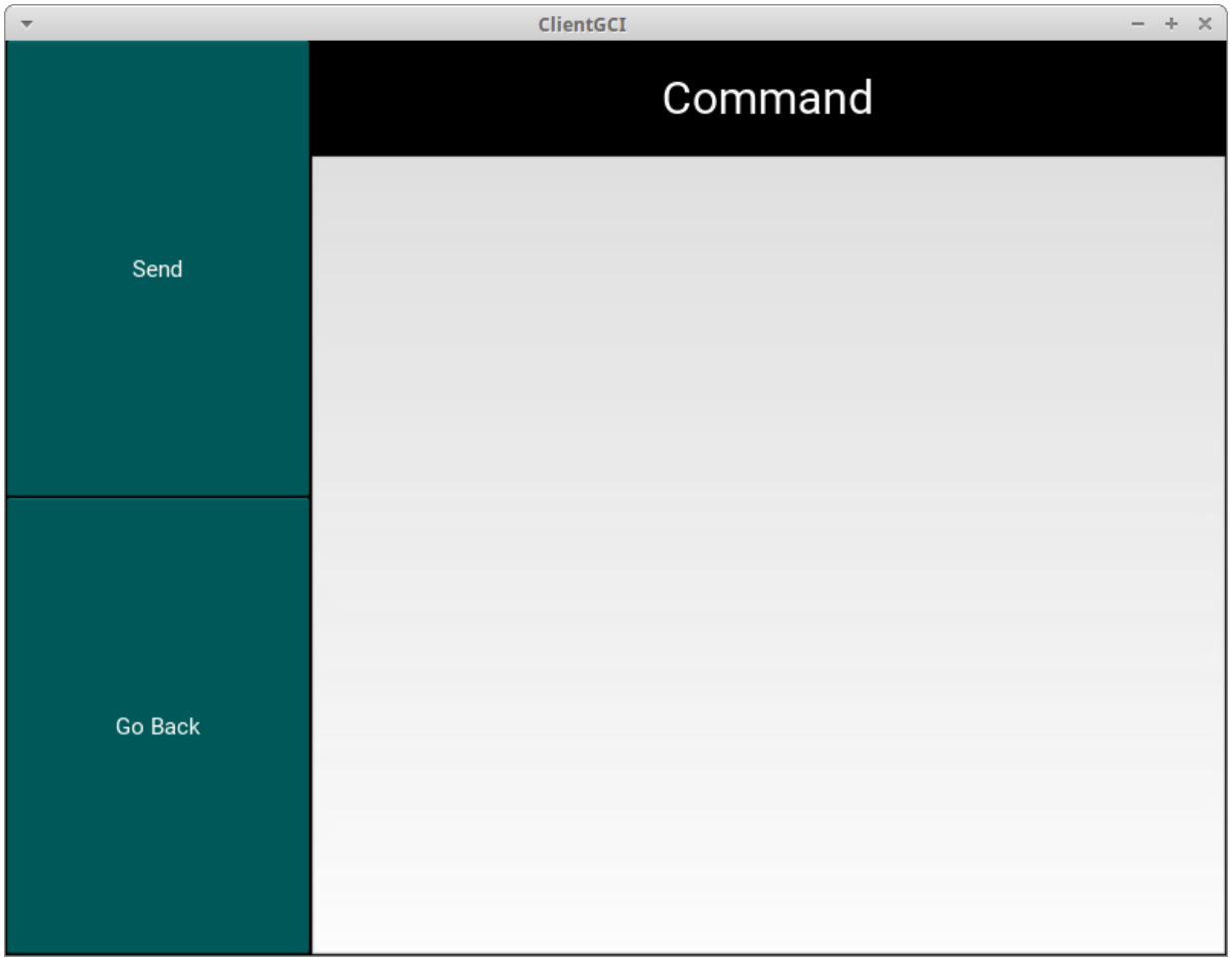

Figure 10: Command screen for the GCI. From here the user can send commands using the existing SSH connection directly to the measurement PC.

4 Development and Code Explanation

The following chapter discusses the implementation of the GCI. First the implemented modules are described, followed by the explanation of the most important classes. This chapter also deals with compatibility problems and difficulties which arose during the code development. Finally the installation guide provides detailed instructions how to install the application on a desktop PC and on mobile devices.

4.1 Modules

The following overview of imported modules introduces the module description in this subsection.

```
#Importing kivy framework
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.screenmanager import ScreenManager,Screen
from kivy.uix.screenmanager import NoTransition
from kivy.uix.label import Label
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.button import Button
from kivy.uix.dropdown import DropDown
from kivy.uix.scrollview import ScrollView
from kivy.uix.anchorlayout import AnchorLayout
from kivy.uix.popup import Popup
from kivy.uix.textinput import TextInput
from kivy.properties import Property
from kivy.uix.image import Image
from kivy.uix.behaviors import ButtonBehavior
from kivy.clock import Clock

#Importing ssh connection modul (paramiko)
from paramiko import client, transport

#Importing Timer
from threading import Timer

#Importing os to send terminal commands
import os

#Importing pickle for aborting
import pickle

#Importing sqlite3 to manipulate SQL Database
import sqlite3
```

```
from kivy.app import App
```

The App class in Kivy is the base class for any Kivy based application. Most commonly developed applications are derived from this class. After an instance of the App class is created, the application can be started.

```
from kivy.lang import Builder
```

The Builder is required to use KV language in your code . As described in 2.2.4, the KV language is an effective graphic design language. There are two ways to do this. Using the code line:

```
Builder.load_file('path/to/file.kv')
```

gets the KV code stored in a file, or

```
Builder.load_string(kv_string)
```

loads the KV code from a string.

```
from kivy.uix.screenmanager import ScreenManager,Screen
from kivy.uix.screenmanager import NoTransition
```

With the screen manager widget one can manage multiple screens used in the application. There are several options available to modify the appearance, like Transition Base, which defines the kind of transition between the screens.

```
from kivy.uix.label import Label
```

Labels in Kivy are used to render text, supporting ASCII and UNICODE strings.

```
from kivy.uix.boxlayout import BoxLayout
```

The BoxLayout in Kivy is used to structure the widgets in the application. It arranges the children-widgets in a horizontal or vertical box.

```
from kivy.uix.button import Button
```

The Button is considered a text label which reacts to associated actions. These actions are triggered when the button is either pressed or released. The configuration is very similar to the configuration of the label.

```
from kivy.uix.dropdown import DropDown
```

The Dropdown widget provides a versatile dropdown list that can implement custom widgets. The special advantage using this toolkit is, that it can contain different kinds of

widgets like buttons, images or label. To open the dropdown list, one needs to press the mainbutton. Once the list is opened, the positioning is fully automatic fitted in the GUI and uses the space in a beneficial way.

```
from kivy.uix.scrollview import ScrollView
```

ScrollView is the widget which provides scrollable views in Kivy. It accepts one widget as a child, for instance a Label, and applies a scrollable window to it, according to the scroll_x and scroll_y properties. In order to distinguish between scrolling and touch events ScrollView uses the properties scroll distance and scroll timeout.
Scroll distance is the minimum distance to travel (default: 20 pixels).
Scroll timeout is the maximum time period (default: 55 milliseconds).
Touch events traveling the scroll distance faster than the scroll timeout will be recognized as scrolling gestures. If the touch event lasts longer than the scroll timeout, it will be handled as a touch down and the event will be dispatched to the child.

```
from kivy.uix.anchorlayout import AnchorLayout
```

The AnchorLayout is very similar to the BoxLayout. In contrast to the vertical and horizontal alignment of the BoxLayout, the Anchor Layout aligns it children-widgets either to the border or to the center.

```
from kivy.uix.popup import Popup
```

The Popup module enables its children-widgets to show up when associated events, like pressing a button, take place. The minimal configuration for a pop up is to set a title and a content. By default, the size is (1,1), like for every other widget in Kivy. This means the pop up window will cover the whole parent window, which isn't a desired behavior in common cases. Pop ups can include other widgets like shown in the code below:

```
popup = Popup(title='Info', content=Label(text='Deleted Row %d'
%delID),size_hint=(None, None), size=(200, 200))
popup.open()
```

```
from kivy.uix.textinput import TextInput
```

The TextInput widget provides a text input box with editable plain text for the user. There are some configuration options like "Unicode","multiline" and so on.

```
from kivy.properties import Property
```

Properties in Kivy provide class variables. Every function of this class can access this variable.

```
from kivy.uix.image import Image
```

The Image widget provides image support for Kivy.

```
from kivy.uix.behaviors import ButtonBehavior
```

The ButtonBehavior class is a mixin class in Kivy. A mixin class is a concept in object-oriented programming languages that offers methods to be used by other classes without being parent of those other classes. Therefore the ButtonBehavior class provides button behavior for other widgets such as images.

```
from kivy.clock import Clock
```

In some cases it is very useful to call functions time delayed. The Clock object provides this function in Kivy. By using the Clock module it is also possible to schedule function calls in the future. Such calls can be configured to occur only once or repeated at specified intervals.

```
from paramiko import client
from paramiko import transport
```

A preferable way to implement the SSHv2 protocol in Python 2.6+ and Python 3.3+ is Paramiko. It provides client and server functionality. Paramiko is a combination of the Esperanto words for "paranoid" and "friend" and is released under the GNU Lesser General Public License (LGPL). It is a pure Python interface around SSH networking concepts, leverages a Python C extension for low-level cryptography and therefore provides secure (encrypted and authenticated) connections. Therefore the Cryptography library is it's only direct hard dependency and needs to be installed. A low-level, C-based encryption to implement the SSH protocol is provided by this module. To get Paramiko working on Linux one needs a C build tool chain , additional development headers for Python, OpenSSL and lib installed on the system. Paramiko mainly supports POSIX (Portable Operating System Interface) platforms with standard OpenSSH implementations. It is well tested on Linux, but most unfortunately it is not working on Android. This compatibility issue will be discussed later in 4.3.

The following command lines are recommended to successfully install the Paramiko tool:

Like mentioned before, cryptography needs to be installed first.

```
$ pip install cryptography
```

The preferable method to install Paramiko is to use pip:

```
$ pip install paramiko
```

Paramiko currently supports Python 2.6, 2.7, 3.3+ and PyPy.

```
import threading
from threading import Timer
```

A thread is a separate part of the program which can be run independently of other parts. Threading is a kind of an extension of the function concept of a programming language. You could understand a thread like a function call. The advantage of threads compared to processes is, that several threads share one storage for global variables. If one thread modifies a variable, the other threads will immediately recognize this. In this way threads are able to communicate. Another important advantage of threading is that the program can be executed way faster on multi core systems, because the threads can be executed simultaneously.
The module threading implements threads in Python. Every thread is represented by an object. Timer is a part of the threading module and provides the execution of a thread, after a specified time interval has passed.

```
import os
```

The os module provides the possibility to use operating system dependent functionality. In this application, the os module is used to perform bash commands on a linux operating system.

```
import pickle
```

The pickle module provides the function to convert a Python object hierarchy into a byte stream and the inverse operation. This fundamental, but powerful algorithm offers a lot of useful applications like writing the byte stream to files, sending it over networks or store them in a database.
Pickle supports different data formats for this serialization. This could be a printable, human readable, ASCII representation or a binary representation which takes less space.

```
import sqlite3
```

In order to implement SQLite functionality in this application, the sqlite3 module is used.

4.2 Classes

In this subsection the classes, which are the main components of the developed code, will be discussed.

4.2.1 Overview

In the following overview all classes and functions which build the app are mentioned.

```
class ImageButton(ButtonBehavior, Image)
class CustomDropDown(DropDown)
```

```
class sendPythonCommand(Screen)
  def sendOverSSH(self,text)
class Menu(Screen)
class JobserverConnection(Screen)
  def on_enter(self):
  def connectToServer(self,url,user,password):
  def updateLocalConnection(self,url):
  def checkConnection(self,cursor):
  def setCurrent(self,String):
  def closeConnection(self):
class Jobserver(Screen):
  def createQueueData(self,table,cursor):
  def setQueue(self, string):
  def clearQueue(self):
  def getTableEntry(self, table, value, row,cursor):
  def getRunning(self,table,box,cursor):
  def getRows(self,table,cursor):
  def deleteRow(self,table,cursor, runjobid, delID):
  def moveRow(self,table,cursor,orID, UpDown):
  def moveRowRelativeUp(self, text, popup, table,
        cursor, orID):
  def moveRowRelativeDown(self, text, popup, table,
        cursor, orID):
  def moveRowAbsolute(self, text, popup, table, cursor, orID):
  def editRow(self,table,cursor, runjobid):
  def updateQueue(self,table):
  def abortCurrentJob(self,cursor,table):
  def on_enter(self):
  def on_leave(self):
class ssh:
  def __init__(self, address, username, password):
  def sendCommand(self, command):
class ClientGUI(App):
  def build(self):
```

The implementation of the following classes will be discussed more detailed in this subsection:

- class sendPythonCommand(Screen)
- class Menu(Screen)
- class JobserverConnection(Screen)
- class Jobserver(Screen)
- class ClientGUI(App)

These classes have been developed within the scope of this thesis and represent the screens in the GUI and the ClientGUI class, which this is the main class and starts the application.

4.2.2 sendPythonCommand

```
class sendPythonCommand(Screen)
        def sendOverSSH(self,text):
```

The class sendPythonCommand is one of the four main functionalities available in a single screen in the GCI. This class provides only one function sendOverSSH which handles the SSH communication.

sendOverSSH(self,text)

```
def sendOverSSH(self,text):
        sshCon.sendCommand(text)
```

This function calls the sendCommand() function of the SSH module, which is described in the SSH class. In this function the global SSH object is used. The commands passed to the function will be sent using the existing SSH connection.

4.2.3 Menu

```
class Menu(Screen)
```

The main task of the menu screen is to enable the navigation withing the GCI. The menu is implemented using the KV language which will be briefly discussed at this point.

KV Code - Menu Screen

```
<Menu>
  name: 'Menu'
  BoxLayout:
    orientation: 'horizontal'
        BoxLayout:
          id: Menu
          size_hint:    root.widthNavColumn, 1
          orientation: 'vertical'
          Button:
            text: 'Jobserver'
            on_release:
            root.manager.current = 'Jobserver'
          Button:
            text: 'sendCommands'
```

```
            on_release:
            root.manager.current = 'sendPythonCommand'
      Button:
        text: 'Quit'
        on_release: app.stop()
    BoxLayout:
      orientation: 'vertical'
      Label:
        text: 'Client GUI'
            font_size: 50
            size_hint_y: 0.2
      Image:
        source:
            "source of image"
            orientation: 'top'
```

The KV language makes designing of user interfaces in Kivy easier . The <Menu> expression identifies the class of which the file is about. The name and the id entries are for the access to the single parts of the KV file. Using these identifiers, one can also manipulate the design programmed in KV from outside, using a Python script. For example one can add a button from Python to the KV graphic, using the following command:

```
self.ids.JobserverButton.add_widget(button)
```

One can also use functions, insert pictures and adjust the graphical surface. Doing all this in a clear way makes the KV language a useful part of Kivy.

4.2.4 JobserverConnection

```
class JobserverConnection(Screen)
    def on_enter(self):
    def connectToServer(self,url,user,password):
    def updateLocalConnection(self,url):
    def checkConnection(self,cursor):
    def setCurrent(self,String):
    def closeConnection(self):
```

The class JobserverConnection provides all the functions needed to connect to the jobserver using SSH. In the following only the most important parts will be explained.

on_enter(self)

The first function called on_enter() is an event which is fired when the screen is displayed. It is a standard part of the screen manager. There are also other standard functions like on_leave(), on_pre_enter() and on_pre_leave() which allow to react to the leaving, the

imminent entering and the imminent leaving of a screen. In on_enter() the graphical surface of the JobserverConnection screen is designed.

First the dropdown list is created:

```
dropdown = DropDown()
```

Creating the dropdown object.

```
btn = Button(text= "JobserverName", size_hint_y=None, height=44)
```

Creating the button object. Every button in the dropdown list is a standard jobserver. In this case it is "JobserverName".

```
btn.bind(on_release=lambda btn: dropdown.select("btn.text"))
btn.bind(on_release=lambda btn: self.setCurrent("Menu"))
btn.bind(on_press=lambda btn:
  self.connectToServer("Server.Address","user","password"))
```

The btn.bind() expression binds functions to the button. Lambda is a placeholder for the function. The functions select(), setCurrent() and connectToServer() will be called once the button is released. Select() is a standard function in the dropdown class and it is obligatory to attach a callback that calls this method. Passing the text of the button makes sure that the selection picks this button. SetCurrent() sets the screen to the string given as a parameter. In this case when the button is pressed, the screen switches to menu. ConnectToServer() will be discussed in the following.

```
dropdown.add_widget(btn)
```

Now add this button to the dropdown list.

```
mainbutton = Button(text='Choose Jobserver',
  size_hint=(None, 0.8),width=300)
```

The mainbutton will appear as the first button before the dropdown list is pressed.

```
mainbutton.bind(on_release=dropdown.open)
```

This expression binds the open function of the dropdown object to the mainbutton.

```
dropdown.bind(on_select=lambda instance,
  x: setattr(mainbutton, 'text', x))
```

The last step completing the dropdown list is listening for the selection in the list, using the on_ select statement and assign the data to the mainbuttons text.

```
self.ids.JobserverConnection.add_widget(mainbutton)
```

Finally adding the mainbutton to the screen described in the KV file.
The rest of the class JobserverConnection adds the textinputs to the screen. This is pretty similar to adding the dropdown list to the screen and will not be mentioned here.

connectToServer(self,url,user,password)

```
sshCon = ssh(url,user,password)
```

This line opens the SSH connection using the ssh class with the parameters url, user and password. The global variable sshCon is used all over the code to communicate through SSH.

```
os.system("sshpass -p password scp user@Server.Address:
path/to/database path/to/storage location")
```

The os.system() function is a standard C function and executes the string given to it in a sub shell. The sshpass command, executed in the shell, copies the database from the measurement PC to the app device.

```
connection = sqlite3.connect("/path/to/database",
check_same_thread=False)
```

After copying the database to a local directory, the connection to it is opened with the sqlite3.connect() function.

```
cursor = connection.cursor()
```

Now the cursor to manipulate the database can be opened.

```
if(self.checkConnection(cursor)):
        print ("Connection to database established")
        CURSOR=cursor
else:
        App.get_running_app().stop()
        print ("Unable to connect")
```

Finally the connection to the database is checked with the checkConnection() function. If it worked, the local cursor is given to the global one. If not, the app stops and prints an error message.

updateLocalConnection(self,url)

```
global CURSOR

os.system('sshpass -p password scp user@Server.Address:'
'/path/to/database /path/to/storage location')

connection = sqlite3.connect("/path/to/database"
,check_same_thread=False)

CURSOR = connection.cursor()
```

updateLocalConnection() repeats the steps from connectToServer() to make sure that the connection is up to date. It is called after making changes to the database and in regular time intervals.

checkConnection(self,cursor)

```
try:
        cursor.execute("SELECT * FROM queue1")
        data=cursor.fetchall()
except:
        return 0

return 1
```

Using a try except statement, the checkConnection() function consists out of a basic SQL statement to test the connection. If it is working the value 1 will be returned, if not 0.

setCurrent(self,String)

```
self.parent.current=String
```

setCurrent() sets the screen to the given string and is used when parts of the graphical surface(e.g. buttons) are created dynamically. It provides the same functionality like the root.manager.current call in the KV file.

closeConnection(self):

```
self.connection.commit()
CURSOR.close()
```

Once the connection to the database isn't needed anymore, closeConnection() will be called. The commit() function saves the changes to the database, the close() function finally closes the connection.

4.2.5 Jobserver

```
class Jobserver(Screen):
  def createQueueData(self,table,cursor):
  def setQueue(self, string):
  def clearQueue(self):
  def getTableEntry(self, table, value, row,cursor):
  def getRunning(self,table,box,cursor):
  def getRows(self,table,cursor):
  def deleteRow(self,table,cursor, runjobid, delID):
  def moveRow(self,table,cursor,orID, UpDown):
  def moveRowRelativeUp(self, text, popup, table,
        cursor, orID):
  def moveRowRelativeDown(self, text, popup, table,
        cursor, orID):
  def moveRowAbsolute(self, text, popup, table, cursor, orID):
  def editRow(self,table,cursor, runjobid):
  def updateQueue(self,table):
  def abortCurrentJob(self,cursor,table):
  def on_enter(self):
  def on_leave(self):
```

The Jobserver class is the most important class in the application. It provides the functionalities to view and control the jobs listed in the jobserver.

getTableEntry(self, table, value, row,cursor)

```
cursor.execute("SELECT %s FROM %s" %(value,table))

for index in range(row):
  data=cursor.fetchone()

return data[0]
```

The function getTableEntry() returns values from the database. One can select the preferred value by passing the parameters table, value and row. An example for the use of getTableEntry() is creating the joblist in the createQueueData() function.

createQueueData(self,table,cursor)

createQueueData() is a very important function, it visualizes the data saved on the jobserver.

```
DbCon=JobserverConnection()
DbCon.updateLocalConnection("path/to/database")
```

To guarantee the topicality of the shown data, the connection to the database is always updated before showing any data. This happens with the help of an JobserverConnection instance called DbCon and the function updateLocalConnection().

```
self.clearQueue()
```

Before showing any new data, the screen is cleared by using the clearQueue() function which will be described later.

```
saveRun = int(table[5]+table[6])
```

Out of the given parameter table you can extract the queue number.

```
QueueBox=BoxLayout(orientation='vertical')
```

Creating the BoxLayout for the data stored on the jobserver dynamically.

```
for index in range(0,self.getRows(table,cursor)):

  Layout=BoxLayout(orientation='horizontal',size=(1,40),
    size_hint=(1,None))

  if self.getRunning('running',saveRun+1,cursor) !=
   self.getTableEntry(table,"runjobid",index+1, cursor):

    text=Label(size_hint_x=0.3)
    Layout.add_widget(text)

    btn = ImageButton(source='Path to Up png',
    size_hint_x=0.3,size_hint_y=0.7,center_y=
      self.parent.center_y)
    btn.index=index+1
    btn.indexRow=1
    btn.bind(on_release=lambda btn: self.moveRow(table,cursor,
     btn.index, btn.indexRow))
    Layout.add_widget(btn)

    btn = ImageButton(source='Path to Down png',
     size_hint_x=0.3,size_hint_y=0.7,center_y=
     self.parent.center_y)
    btn.index=index+1
    btn.indexRow=2

    btn.bind(on_release=lambda btn: self.moveRow(table,cursor,
     btn.index,btn.indexRow))
```

```
        Layout.add_widget(btn)


        btn = ImageButton(source='Path to Edit png',
         size_hint_x=0.4,size_hint_y=0.8,center_y=
         self.parent.center_y)
        Layout.add_widget(btn)
        btn.index=self.getTableEntry(table,"runjobid",index+1,
         cursor)
        btn.bind(on_release=lambda btn:
         self.editRow(table,cursor,btn.index))
        Layout.add_widget(btn)

        btn = ImageButton(source='Path to Delete png',
         size_hint_x=0.3,size_hint_y=0.65,center_y=
         self.parent.center_y)
        btn.index=self.getTableEntry(table,"runjobid",index+1,
         cursor)
        btn.deleteIndex=index+1
        btn.bind(on_release=lambda btn:
         self.deleteRow(table,cursor,btn.index,btn.deleteIndex))
        Layout.add_widget(btn)

        text=Label(text=str(self.getTableEntry(table,"runjobid",
         (index+1),cursor)),size_hint_x=1.5,size_hint_y=0.7,
         center_y= self.parent.center_y)
        Layout.add_widget(text)

        text=Label(text=str(self.getTableEntry(table,"name",(index+1),
         cursor)), text_size=(270,20), shorten='true',size_hint_x=5,
         size_hint_y=0.7)
        Layout.add_widget(text)

        text=Label(text=str(self.getTableEntry(table,"temperature",
         (index+1),cursor)), text_size=(100,20), halign='right',
         size_hint_x=0.5,size_hint_y=0.7,center_y=
         self.parent.center_y)
        Layout.add_widget(text)

        text=Label(size_hint_x=1.2)
        Layout.add_widget(text)


    else:
```

```
text=Label(size_hint_x=0.3)
Layout.add_widget(text)

btn = ImageButton(source='Path to Edit png',
 size_hint_x=1.3,size_hint_y=0.5,center_y=
 self.parent.center_y)
btn.bind(on_release=lambda btn:
 self.abortCurrentJob(cursor,table))
Layout.add_widget(btn)

text=Label(text=str(self.getTableEntry(table,"runjobid",
 (index+1),cursor)), color=(0, 1, 0, 1), size_hint_x=1.5,
 size_hint_y=0.6)
Layout.add_widget(text)

text=Label(text=str(self.getTableEntry(table,"name",
 (index+1),cursor)), color=(0, 1, 0, 1),text_size=(270,20),
 shorten='true',size_hint_x=5,size_hint_y=0.6)
Layout.add_widget(text)

text=Label(text=str(self.getTableEntry(table,"temperature",
 (index+1),cursor)),color=(0, 1, 0, 1), text_size=(100,20),
 halign='right',size_hint_x=0.5,size_hint_y=0.6)
Layout.add_widget(text)

text=Label(size_hint_x=1.2)
Layout.add_widget(text)
```

The for loop iterates over all jobs of the queue and is used to create the visualization objects for each job. The number of jobs is fetched using the function getRows() which will be explained later. While iterating over each job the function getRunning() is used to check whether the currently displayed job is being executed or not. If this is not the case, the ImageButtons to manipulate the current row will be added to the screen. These ImageButtons are bound to the functions moveRow(), editRow() and deleteRow(). Next, the labels which show the temperature, the name of the job as well as the configuration are added to the configuration line. If the current row is the running job, only one ImageButton, namely the button providing the functionality to abort the current job, is visible in the configuration line. The other entries of the configuration line remain as mentioned before, except the entire line is highlighted green to emphasize that this job is currently executed.

```
self.ids.QueueData.add_widget(Layout)
```

Finally the layout, including the data from the jobserver database, is added to the screen.

setQueue(self, string)

```
self.queue = str(string)
```

The function setQueue() sets the value appropriate private variable of the class representing the selected queue which is passed to the function as string. This function is called when the selection from dropdown list, where one can select the currently displayed job queue, is changed.

clearQueue(self)

```
self.ids.QueueData.clear_widgets()
```

Before a new dataset can be displayed the screen has to be cleared . Therefore Kivy provides the function clear_widgets().

getRunning(self,table,box,cursor)

```
cursor.execute("SELECT jobid FROM %s" %table)

for index in range(box):
 running=cursor.fetchone()

return running[0]
```

If a measurement job is currently running, the database contains an entry with the job identification number of the running job. The above function checks the table called running and returns the table entry in order to check if there is currently a job running on the selected device. If a job is running, the return value is its job identification number, if not -1 will be returned. getRunning() is pretty similar to the function getTableEntry() and was introduced to improve the readability of the code.

getRows(self,table,cursor)

```
cursor.execute("SELECT * FROM %s" %table)
data = cursor.fetchall()
return len(data)
```

getRows() returns the number of entries available in a defined jobserver queue, i.e. the number of submitted jobs. The function len() provides the length of the data after the whole table is fetched.

deleteRow(self,table,cursor, runjobid, delID)

Intuitively, the function deleteRow() removes a selected jobs from the queue.

```
sshCon.sendCommand("sqlite3 \path\to\database \"SELECT *
 FROM %s\"" %table)
cursor.execute("SELECT * FROM %s" %table)
```

All commands which are used to manipulate the jobserver database are sent via SSH connection. At the same time, each command is executed at a local copy of the database. By using the latter the application gets way faster, because the database has not to be transferred between the mobile device and the measurement PC over the SSH connection every time a command is executed.

```
result = cursor.fetchall()
```

First of all, the whole table is fetched using the command fetchall().

```
runjobidDel = result[delID-1][1]
nameDel = result[delID-1][2]
temperatureDel = result[delID-1][3]
configDel = result[delID-1][4]
```

The variable delID identifies the row which should be removed from the queue. Using this index, the data of this row is saved in the variables runjobidDel, nameDel, temperatureDel and configDel. Storing the data is necessary, because it will be copied to the last entry before deletion. The row indices has to be decremented by one, because of the ID counter starts at one but the programming index starts at zero.

```
sshCon.sendCommand("sqlite3 \path\to\database \"UPDATE %s SET
 runjobid=%d,name='%s' ,temperature=%d  WHERE ID=%d\""
 %(table,int(runjobidDel),nameDel,int(temperatureDel),
 self.getRows(table,cursor)))
cursor.execute("UPDATE %s SET  runjobid=%d,name='%s' ,
 temperature=%d  WHERE ID=%d" %(table,int(runjobidDel),nameDel,
 int(temperatureDel), self.getRows(table,cursor)))
```

The data stored in the above mentioned variables is copied to the last entry in the joblist, this entry will be deleted. By using this mechanism, the order of the IDs stays consistent and complicated algorithms for sorting aren't necessary.

```
i=self.getRows(table,cursor)
```

The number of rows is written to the variable i.

```
while(i>delID):
 runjobidI = result[i-1][1]
 nameI = result[i-1][2]
 temperatureI = result[i-1][3]
 configI = result[i-1][4]

 sshCon.sendCommand("sqlite3 \path\to\database \"UPDATE %s SET
  runjobid=%d,name='%s',temperature=%d  WHERE ID=%d\""
  %(table,int(runjobidI),nameI,int(temperatureI),i-1))
 cursor.execute("UPDATE %s SET  runjobid=%d,name='%s',
  temperature=%d WHERE ID=%d" %(table,int(runjobidI),nameI,
  int(temperatureI),i-1))

  i=i-1
```

Using a while loop to iterate over the entries of the queue, the jobs with ID greater than the delete ID are copied to its predecessor ID.

```
sshCon.sendCommand("sqlite3 \path\to\database \"DELETE FROM %s
WHERE runjobid=%d\"" %(table, runjobid))
cursor.execute("DELETE FROM %s WHERE runjobid=%d" %(table,
 runjobid))
```

Next, the last entry in the list is deleted.

```
self.createQueueData(table,cursor)
```

The function createQueueData() is called to display the current joblist.

```
popup = Popup(title='Info', content=Label(text='Deleted Row %d'
 %delID), size_hint=(None, None), size=(200, 200))
popup.open()
```

Finally, a pop up informs the user about the deleted row. The deletion process for job 3 is visualized in **Figure 11**.

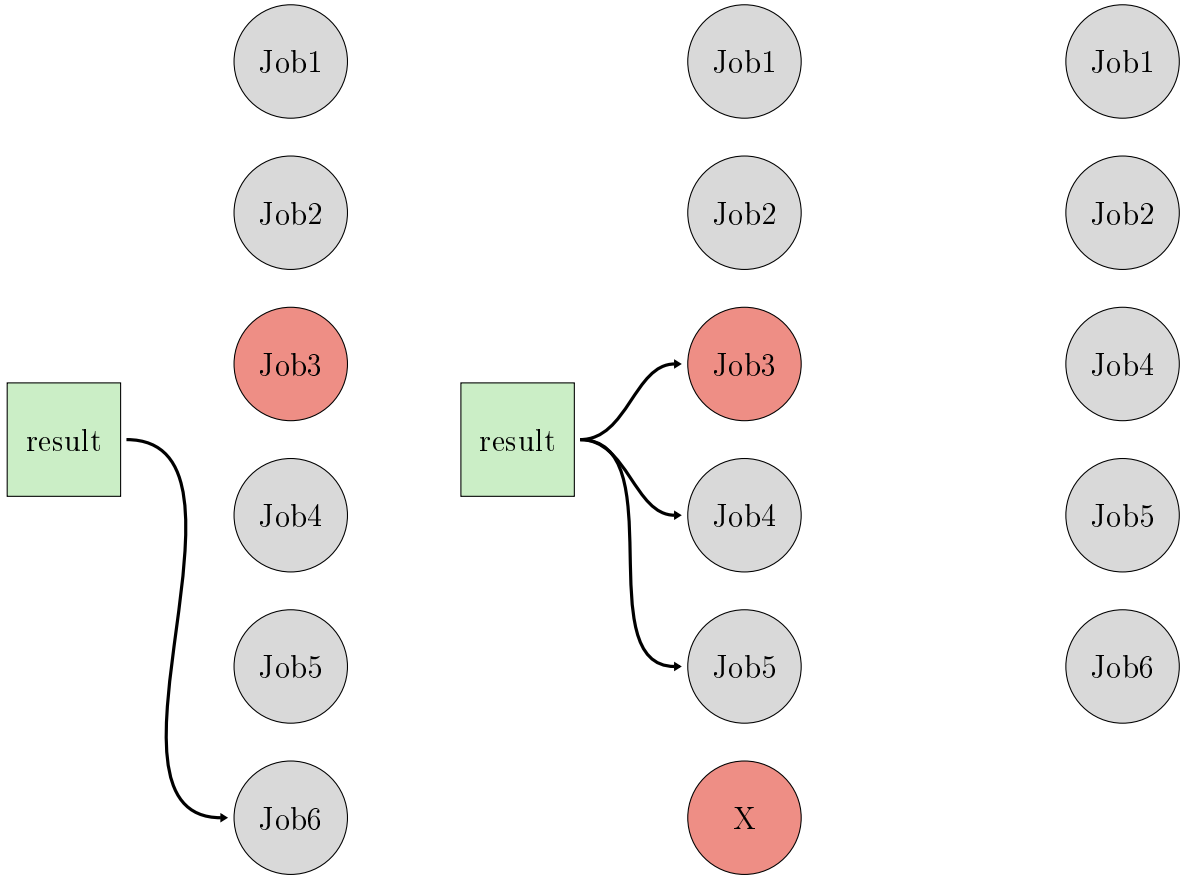

Figure 11: The deletion process for Job3 is visualized. In order to maintain the right ID order in the joblist, Job3 is copied to the end of the list and removed from there using the result set. This result set contains the entire joblist before the deletion and provides the data for the next step where Job3, Job4 and Job5 are replaced with their successor row. Finally, Job3 is removed and the queue is in the right ID order.

moveRow(self,table,cursor,orID, UpDown)

Overview

moveRow() is the base function from which the functionalities of moveRowRelativeUp(), moveRowRelativeDown() and moveRowAbsolute() are derived. Depending on the passed arguments, the function displays the options to relatively move rows up/down and to move rows absolutely. Before discussing the implementation, the working principle of these functions is described closer, see **Figure 12** and **Figure 13**.

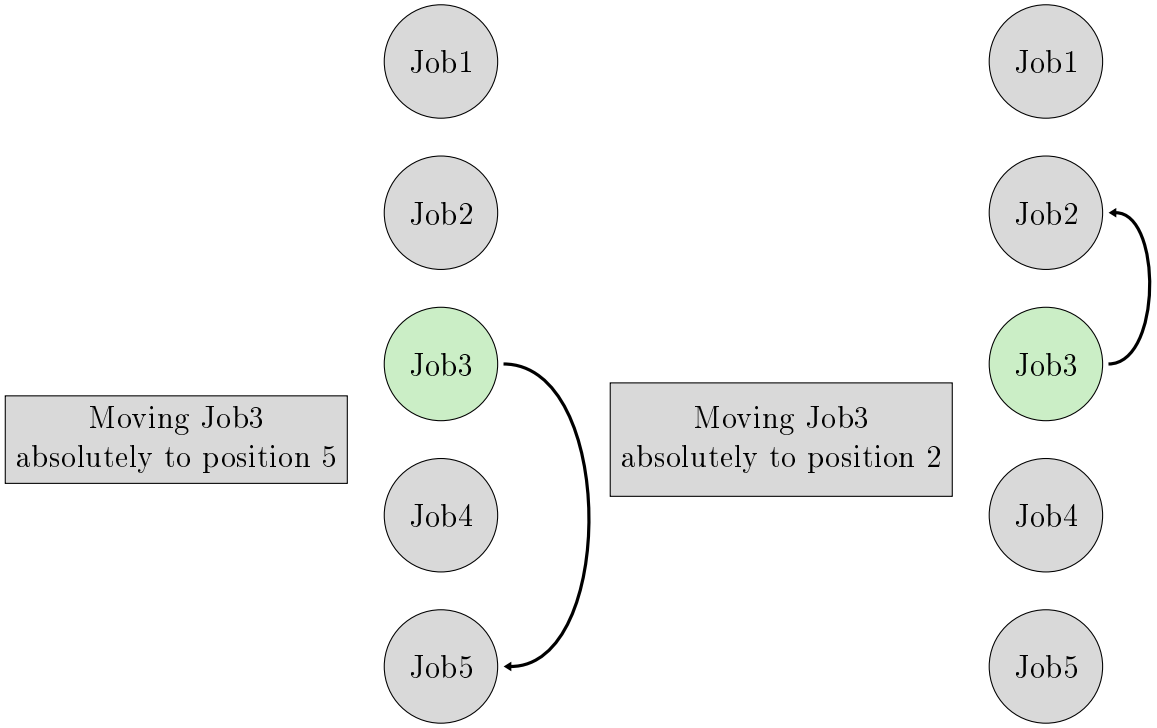

Figure 12: Absolute movement of Job3. The destination ID, specifying the position where the job should get moved to, needs to be declared as an absolute value.

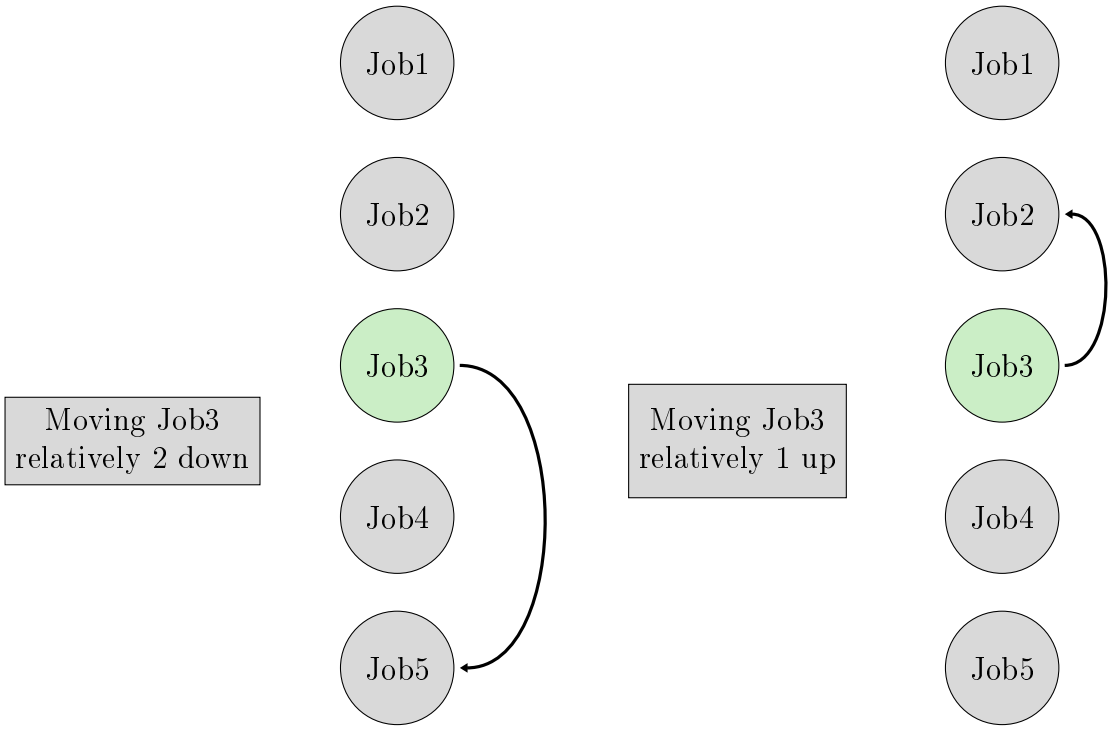

Figure 13: Relative movement of Job3. The destination ID, specifying the position where the job should get moved to, needs to be declared as a relative value starting from the current position of the row.

Description

```
box = BoxLayout(orientation ='vertical')
box.add_widget(Label(text='relative'))

textinput=TextInput(multiline=False)

if UpDown == 1:
  textinput.bind(on_text_validate=lambda textinput:
   self.moveRowRelativeUp(textinput.text,popup, table,
   cursor, orID))
elif UpDown == 2:
  textinput.bind(on_text_validate=lambda textinput:
   self.moveRowRelativeDown(textinput.text,popup, table,
   cursor, orID))

box.add_widget(textinput)
```

These lines describe the textinput for the relative movement. The parameter UpDown decides whether the function moveRowRelativeUp() or moveRowRelativeDown() is called.

```
box.add_widget(Label(text='absolute'))

textinput=TextInput(multiline=False)
textinput.bind(on_text_validate=lambda textinput:
self.moveRowAbsolute(textinput.text,popup, table,
 cursor, orID))

box.add_widget(textinput)
```

Afterwards the textinput for the absolute movement is described.

```
popup = Popup(title='Move Row', content=box, size_hint=
(None, None), size=(200, 200))
popup.open()
```

After configuring the pop up, it's opened using the function popup.open().

moveRowRelativeUp(self,text,popup,table,cursor,orID)
and
moveRowRelativeDown(self, text, popup, table, cursor, orID)

Considering that the implementation of the two functions moveRowRelativeUp() and moveRowRelativeDown() are relatively similar, in the following only the up movement

will be described. moveRowRelativeUp() provides the functionality to move a row upside from its original position. The ID of the configuration line where the job should get moved to, needs to be declared relative starting from the current position of the row.

```
if((orID-desID)==1):
 popupError=Popup(title='ERROR',content=
  Label(text='Not allowed!'), size_hint=(None, None),
  size=(100, 100))
 popupError.open()
else:
 sshCon.sendCommand("sqlite3 Path to database \"SELECT *
 FROM %s\"" %table)
 cursor.execute("SELECT * FROM %s" %table)
 result = cursor.fetchall()
```

This if statement makes sure that no row can be moved to the top of the queue. This is not possible, because the first entry is always the currently running job. If this isn't the case, the function continues with fetching the whole list.

```
if ((orID-desID) >=1):

 runjobidOr = result[orID][1]
 nameOr = result[orID][2]
 temperatureOr = result[orID][3]
 configOr = result[orID][4]

 sshCon.sendCommand("sqlite3 Path to database \"UPDATE %s SET
  runjobid=%d,name='%s' ,temperature=%d  WHERE ID=%d\""
  %(table,int(runjobidOr),nameOr,int(temperatureOr),(orID-desID))
 cursor.execute("UPDATE %s SET  runjobid=%d,name='%s' ,
  temperature=%d WHERE ID=%d" %(table,int(runjobidOr),nameOr,
  int(temperatureOr),orID-desID))
 i=orID
   .
   .
else:
 popupError=Popup(title='ERROR',content=
  Label(text='Wrong Input'),size_hint=(None, None),
  size=(100, 100))
 popupError.open()
```

The following if statement checks if the users input is valid. If this is the case, the configuration line data of the row which should be moved is stored in the variables runjobidOr, nameOr, temperatureOr and configOr. Afterwards this stored data is copied to the destination row and the origin ID is saved in the variable i. If the if statement is FALSE an

error message will pop up and inform the user that the input is invalid.

```
        .
        .
while(i>=(orID-desID)):
 runjobidI = result[i-1][1]
 nameI = result[i-1][2]
 temperatureI = result[i-1][3]
 configI = result[i-1][4]

 sshCon.sendCommand("sqlite3 Path to database \"UPDATE %s
  SET  runjobid=%d,name='%s' ,temperature=%d  WHERE ID=%d\""
  %(table,int(runjobidI),nameI,int(temperatureI),i+1))
 cursor.execute("UPDATE %s SET  runjobid=%d,name='%s' ,
  temperature=%d WHERE ID=%d" %(table,int(runjobidI),nameI,
  int(temperatureI),i+1))

 i=i-1
```

Continuing in the if branch, the while loop is used to iterate over all elements and to replace all rows between the origin ID and the destination ID with their predecessor row. Afterwards the joblist is in the right order again.

```
connection.commit()
self.createQueueData(table,cursor)
```

Using the function commit() one can update the database. Now the new list of jobs of the selected queue is displayed. The whole procedure is visualized in **Figure 14**.

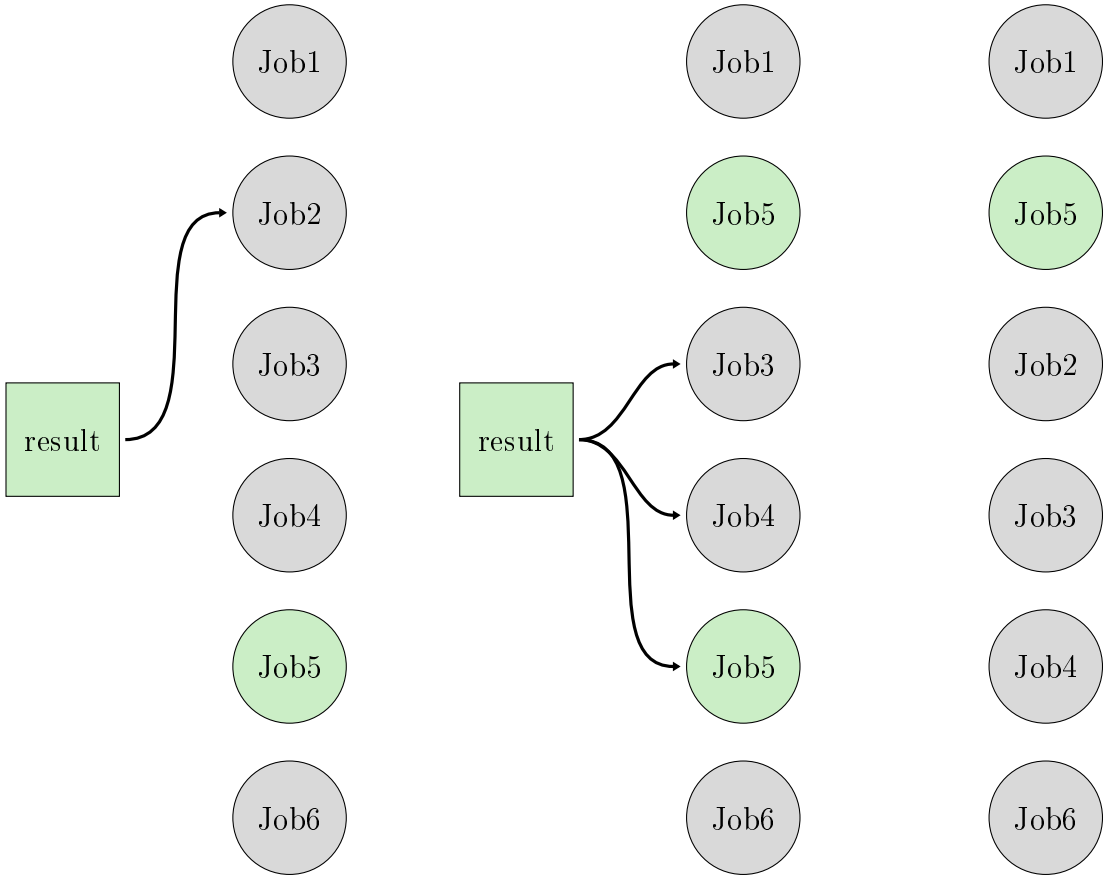

Figure 14: The up movement of Job5 is visualized. As a first step, Job5 is copied to its destination ID using the result set and replaces Job2. Following Job3, Job4 and the initial Job5 are replaced with their predecessors in the joblist. Finally Job5 is at the desired position and the ID order of the joblist is right.

moveRowAbsolute(self, text, popup, table, cursor, orID)

Besides moving a row relatively, the application also provides the function to move the row absolutely. In the following this function will be described.

```
desID=int(text)-1
orID=orID-1
```

Saving the passed parameter text to the local variable desID.

```
if((desID+1)==1):

  popupError=Popup(title='ERROR',content=
  Label(text='Not allowed!'),size_hint=(None, None),
  size=(100, 100))
  popupError.open()
```

```
else:
            .
            .
```

The first if statement checks if the user tries to move a row to the first entry. Like mentioned before, this is not allowed and a pop up will appear if this happens.

```
else:
  sshCon.sendCommand("sqlite3 path/to/database \"SELECT *
   FROM %s\"" %table)
  cursor.execute("SELECT * FROM %s" %table)
  result = cursor.fetchall()
```

Continuing with the else branch, the first step is to fetch the whole table.

```
if (desID+1) >=1 and (desID+1) <= self.getRows(table,cursor):
          .
          .
          .
          .
else:
  popupError=Popup(title='ERROR',content=
   Label(text='Wrong Input'), size_hint=(None, None),
   size=(100, 100))
  popupError.open()
```

Now the correct input with the help of an if statement is checked.

```
          .
          .

if desID > orID:
          .
          .

if desID < orID:
          .
          .
```

If the input is correct, the process flow continues in the if branch. The next step there is to distinguish between two cases, desID greater or smaller than orID.

```
if desID > orID:

  runjobidOr = result[orID][1]
  nameOr = result[orID][2]
  temperatureOr = result[orID][3]
  configOr = result[orID][4]

  sshCon.sendCommand("sqlite3 path/to/database \"UPDATE %s SET
   runjobid=%d,name='%s' ,temperature=%d  WHERE ID=%d\""
   %(table,int(runjobidOr),nameOr,int(temperatureOr),desID+1))
  cursor.execute("UPDATE %s SET  runjobid=%d,name='%s' ,
   temperature=%d WHERE ID=%d" %(table,int(runjobidOr),nameOr,
   int(temperatureOr), desID+1))

  i=orID

  while(i<desID):
    runjobidI = result[i+1][1]
    nameI = result[i+1][2]
    temperatureI = result[i+1][3]
    configI = result[i+1][4]

    sshCon.sendCommand("sqlite3 path/to/database \"UPDATE %s SET
     runjobid=%d,name='%s' ,temperature=%d  WHERE ID=%d\""
     %(table,int(runjobidI),nameI,int(temperatureI),i+1))
    cursor.execute("UPDATE %s SET  runjobid=%d,name='%s' ,
     temperature=%d WHERE ID=%d" %(table,int(runjobidI),nameI,
     int(temperatureI),i+1))

    i=i+1

self.createQueueData(table,cursor)
```

In the case that the destination ID is greater than the origin ID, the values of the original row are stored in the variables runjobidOr, nameOr, temperatureOr and configOr. Next, these values are copied to the destination ID. The variable i contains the origin ID. Now the while loop runs as long as the counter i is smaller than the destination ID. The loop copies all rows between the origin and the destination to their successor row. After the while loop has ordered the joblist again, the function createQueueData() displays the new queue. The case that the destination ID is smaller than the origin ID is very similar. The only difference is that the outer while loop iterates over the entries in a reversed order.

editRow(self,table,cursor, runjobid)

The function editRow() is not developed yet.

updateQueue(self,table)

```
global do
global checkUpdate
global CURSOR


if checkUpdate:

        do=threading.Timer(120, self.updateQueue,[table])
        do.start()
        checkUpdate = 0

else:

        do.cancel()
        do=threading.Timer(120, self.updateQueue,[table])
        do.start()
        self.createQueueData(table,CURSOR)
```

The function updateQueue() refreshes the joblist which is visible in the application. Using an if statement with a check variable makes sure that only one thread is running simultaneously. The thread runs in the background and calls the function recursively after 120 seconds. This updates the joblist repeatedly and makes sure that it's up to date.

abortCurrentJob(self,cursor)

```
cursor.execute("SELECT control FROM devices WHERE connected=1")
stream=cursor.fetchone()

if len(stream):

  control=pickle.loads(stream)
  control["abortCurrentJob"]=1

  cursor.execute("UPDATE devices SET control=\"%s\" WHERE
   connected=1" %(pickle.dumps(control)))

  connection.commit()
```

```
  sshCon.sendCommand("sqlite3 path/to/database \"UPDATE devices
   SET control=\\\"%s\\\" WHERE connected=1\""
   %(pickle.dumps(control)))

  popup=Popup(title='Info',content=
   Label(text='Aborting current job...'),
   size_hint=(None, None), size=(200, 200))
  popup.open()
  Clock.schedule_once(popup.dismiss, 45)
  t = threading.Timer(40, self.createQueueData,[table,cursor])
  t.start()

else:
  print("Error when loading control")
```

In order to cancel running jobs the abortCurentJob() function has been implemented. The green highlighted row in the joblist is canceled after the corresponding button in this row is pressed. First, the control column from the table devices is selected. Using the WHERE statement one can select a single row, in this case where the parameter connected is 1. The next step fetches this entry and saves it to the variable stream. The if statement checks if fetching of the data is successful. Continuing in the TRUE branch, pickle is used to convert the stream in an object using pickle.load(). Now the property abortCurrentJob can be set to logical one. Using pickle.dump() the object is converted back into a string and can be copied back to the table. This will abort the current job the next time the jobserver fetches the database. A pop up informs the user that the abortion is in progress. This pop up will be closed after 45 seconds using the Clock module. At the same time, the createQueueData() function is called 5 seconds before the pop up closes to display the actual joblist.

on_enter(self)

The on_enter() function is called when the screen Jobserver is opened. The dropdown list, viewed on the Jobserver screen, needs to be created dynamically to check which boxes are currently active.

```
self.ids.JobserverColumnBoxes.clear_widgets()
```

Every time the Jobserver screen is entered, the widgets are cleared to ensure that they are not inserted multiple times.

```
for index in range(1,16):#16 number of boxes
  if self.getRunning('running',index,CURSOR)!= -1:
    #testing which box is currently running a job
    btn = Button(text='Queue%d ' % (index-1),
       size_hint_y=None, height = 50)
```

```
    btn.bind(on_release=lambda btn: dropdown.select(btn.text))
    btn.bind(on_release=lambda btn: self.setQueue(btn.text))
    btn.bind(on_release=lambda btn:
           self.createQueueData(btn.text,CURSOR))
    btn.bind(on_release=lambda btn: self.updateQueue(btn.text))

    dropdown.add_widget(btn)
```

In the current implementation a total number of 16 measurement boxes is supported. Only these boxes which run a job are added to the list using the getRunning() function. The functions setQueue(), createQueueData() and updateQueue() are bind to the buttons so that they are called once the button is pressed.

```
mainbutton = Button(text='Select Box', size_hint=(1,1))
mainbutton.bind(on_release=dropdown.open)
dropdown.bind(on_select=lambda instance,
 x: setattr(mainbutton, 'text', x))

self.ids.JobserverColumnBoxes.add_widget(mainbutton)
```

Finally, the dropdown list is completed and added to the Jobserver screen.

on_leave

```
global do
do.cancel()
```

on_eave() is similar to on_enter(). The only difference is that it is called when the screen is closed. Here, the global thread do is canceled when leaving the Jobserver screen to stop the timer which updates the queue every 120 seconds.

4.2.6 ClientGUI

Finally, the class ClientGUI which is an App class is described.

```
class ClientGUI(App):
        def build(self):
```

A App class is the base for Kivy applications and the main entry into the Kivy run loop. To create your own app you need to subclass the App class and create an instance of it. With the help of the run() method you can start the applications life cycle.

build(self)

```
return sm #returning the root widget
```

The only function in the ClientGUI class is the build() function. The job of this function is to return the screen manager object. In the following lines the implementation of the screen manager is discussed:

```
# Create the screen manager
sm = ScreenManager(transition=NoTransition())
sm.add_widget(JobserverConnection())
sm.add_widget(Menu())
sm.add_widget(Jobserver())
sm.add_widget(sendPythonCommand())

if __name__ == '__main__':
        ClientGUI().run()
```

Every instance of the screen class is added to the screenmanager.
The if statement prevents the automatic execution of imported code, because it checks if the script is the main running script. It could also just be referenced by another script.

4.3 Challenges and Compatibility Issues

Throughout the development and implementation of every software project several challenges and compatibility issues must be tackled. In the following, selected challenges which occurred during creation of the GCI, are discussed.

4.3.1 Which Framework should be used for the GCI?

The most crucial question at the beginning was to select the framework which provides the best support for the requirements of the GCI. Basically, the GCI should be executable on any personal computer using Linux operating systems, which is not really a constraint for the graphical tools available. Next, the GCI should also be applicable on mobile devices, initially the ones using Android. After carefully studying many different toolkits like Tkinter, PyQt, wxPython and so on, it turned out that Kivy was the best compromise for an application which should run on PC and on a mobile device. The reason to select a common system for PCs and mobile devices is also to avoid redevelopment of the GCI using individual languages for each system. As the TDDS measurement framework is still under heavy development, each separate tool would have to update every time. Thus, to avoid inconsistencies between the different tools, the development of one common application is preferred.

4.3.2 Which SQL implementation would be the most suitable?

Another question arises when considering the different SQL versions. For the GCI the decision had to be made between sqlite3 and MySQL. The available implementation of the Institute for Microelectronics is based on the sqlite3 interface. For the GCI it would have been advantageous to use MySQL, because it supports network manipulation of databases, which is not possible with sqlite3. Using a MySQL database would have required an extensive update of the existing databases and the manipulation functions used by the

jobserver. The current implementation of the GCI allows to manipulate sqlite3 databases. This can be achieved by sending the commands to the PC connected to the measurement box and executing them there locally. For future releases an update of the jobserver to use MySQL databases should be considered.

4.3.3 How to implement the SSH Communication?

Python offers several possibilities to use the SSH communication protocol. Nevertheless, the SSH communication also needs to work on mobile devices, where only very few tools providing SSH support are available. After carefully studying server-client-socket applications, port forwarding, etc. Paramiko turned out to be the best toolset for SSH. On PCs the SSH communication works as expected. However, there are still many difficulties with the support for Android OS using this toolset.

4.3.4 Building the Application for Android

To compile the GCI for Android using the python file, the buildozer toolkit is used. The configuration of the buildozer tool is done in the buildozer.spec file. This way of building the apk is pretty elaborated, although it is still the easiest way of building the package. The installation and configuration is pretty simple and is described in the installation subsection(4.4) more detailed. To test the application the Kivy launcher is recommended, as it provides a quick way to run the Python code on an Android device.

4.3.5 Dynamically Created Buttons

During development of the application, the need for dynamical created widgets arose. The main part of the screens is programmed in the KV language, which is a part of Kivy. When the design of a screen dynamically depends on the input data set, the screen has to be adjusted dynamically and the data has to be fetched regularly from the corresponding sources. This is tackled using the ids lookup object in the KV file, as visible in the following code:

```
<JobserverConnection>
  name: 'JobserverConnection'
  BoxLayout:
    orientation: 'vertical'
    AnchorLayout:
      id:JobserverButton
```

Kivy collects the widgets tagged with the id expression and stores them in the self.ids dictionary type property. Thus, it is possible to access the object with the following line:

```
self.ids.JobserverButton.add_widget(button)
```

Using these IDs it is possible to add widgets from Python to the screens designed in the KV file.

4.3.6 Background Android Application and Pop-Ups

During planning of the GCI the idea that the GCI, when operated on mobile devices, should also regularly update the connection to the jobserver when operated in the background. If the communication connection brakes or an error occurs, a pop-up message should appear at the home screen of the mobile device and inform the user. As it turned out later during the implementation, Kivy doesn't support raising general pop-up messages outside of the application.

4.3.7 Terminal Embedded in the GCI

To execute custom commands on the measurement PC through the SSH connection a SSH terminal should be implemented in the GCI. Several existing toolkits like python shell, kivyconsole, RemoteShellKivy and paramikoShell were studied for this purpose. Most unfortunately none of these applications fulfilled the requirements for the GCI. Therefore, a simpler version, which sends the commands without respond, is embedded in the initial release of the GCI.

4.3.8 Database Manipulation

As already mentioned, the used SQL language to manipulate the database is sqlite3, which is designed for local database manipulation only. To use sqlite3 in combination with the GCI, the application first copies the whole database to a local directory on the mobile device or PC. Afterwards, the SQL commands are executed locally and sent to the host PC simultaneously. This is required to update the jobserver's database and the locally stored database simultaneously. The advantage of this implementation is that this makes the database manipulation way faster than fetching all the data over the SSH connection every time a manipulation command is performed.

4.4 Installation

4.4.1 Installation of Kivy

```
$ sudo add-apt-repository ppa:kivy-team/kivy
$ sudo apt-get update
$ sudo apt-get install python-kivy
$ sudo apt-get install python3-kivy
```

4.4.2 Installation of Paramiko

```
$ pip install paramiko
```

4.4.3 Installation of sshpass

```
$ sudo apt-get install sshpass
```

4.4.4 Installation of sqlite3

```
$ sudo apt-get install sqlite3
```

4.4.5 Installation of buildozer (Ubuntu 16.04 64bit)

```
$ pip install --upgrade buildozer
$ sudo pip install --upgrade cython==0.21
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
$ sudo apt-get install build-essential ccache git libncurses5:
   i386 libstdc++6:i38 libgtk2.0-0:i386 libpangox-1.0-0:i386
   libpangoxft-1.0-0:i386 libidn11:i386 python2.7 python2.7-dev
   openjdk-8-jdk unzip zlib1g-dev zlib1g:i386
```

4.4.6 Configuration of buildozer

```
$ buildozer init
$ sudo apt-get install autoconf
$ sudo apt-get install libtool
$ sudo apt-get install android- tools-adb
$ sudo apt-get install libsqlite 3-dev
$ sudo pip install pysqlite
```

Additionally these requirements have to be added to the buildozer.spec file: python2, kivy, sqlite3

5 Outlook

The functionality provided by the GCI already simplifies the control and manipulation of the measurement queue. During the development of the GCI many ideas arose which would further improve the usability of the GCI. In the following an outlook, discussing improvements which are planned to be further implemented, is given.

5.1 Implementation of the SSH Connection

As already mentioned, the SSH connection using mobile devices does not work properly yet. The main show-stopper is the Paramiko module which does not support ARM processors. However, such processors are commonly used in a large range of mobile devices. Therefore, alternative methods to provide the SSH connection must be used. For instance, scripts written in alternative programming languages (e.g. Java) can be used to handle the SSH connection and and will be called directly from the Kivy application. Another possibility is that there might be a working tool within Kivy which provides SSH functionality.

5.2 Graphical Representation of the Measurement Data

Another highly demanded feature is the visualization of the measurement data on mobile devices. This would also open the possibility for live monitoring of ongoing measurements and would be of interest as subsequent measurement sequences often depend on current results. Also, the user can survey if the measured device still works or got irreversible damaged during the experiment. There are several ways to realize the visualization. For instance, the Kivy garden module looks very promising for this application. Garden centralizes add-ons for Kivy which are maintained by users. There is a garden.graph module available which can be used for displaying graphs, otherwise also matplotlib interfaces are thinkable. Alternatively, the measurement PC can create png graphic files showing the measurement data which can be downloaded to the mobile device and displayed on it.

5.3 Editing the Measurement Configurations

In future releases the measurement configuration data which is displayed in the jobserver screen should become editable. This enables changing the measurement configurations without the need of deleting a single job, resubmitting the modified job to the job queue and adjust the job order. As a consequence thereof, graphical configuration utility has to be implemented to simplify creating the job configuration.

5.4 Improving the SendCommand Screen

Using the current release of the GCI the SSH commands are sent to the host PC without any response if the execution is successful or failed. The idea to be implemented in future versions is that there will be a specific folder structure, containing several Python scripts, which will be visualized in the GCI. Then there will be the possibility to select the script, define the parameter list and send commands to the measurement PC. Additionally, a help

button will be available to show the first few lines of the selected Python script which provide a description of the command line parameters.

5.5 GCI for iOS

Finally, the GCI should also be available for mobile devices using iOS. Buildozer also provides application packaging for iOS via the kivy-ios project. This project is still under investigation, thus the iOS version of the GCI will be implemented as soon as the kivy-ios project is completed.

References

[1] Ask Ubuntu. https://askubuntu.com. Accessed: 2017-11-09.

[2] Buildozer. https://buildozer.readthedocs.io/en/latest/. Accessed: 2017-11-09.

[3] Cryptography. https://cryptography.io. Accessed: 2017-11-09.

[4] Developer Android. https://developer.android.com. Accessed: 2017-11-09.

[5] Docs Python. https://docs.python.org. Accessed: 2017-11-09.

[6] Geany. https://www.geany.org/. Accessed: 2017-11-09.

[7] Github. https://github.com/. Accessed: 2017-11-09.

[8] Kivy. https://kivy.org/. Accessed: 2017-11-09.

[9] Paramiko. http://www.paramiko.org/. Accessed: 2017-11-09.

[10] Python for Engineers. http://pythonforengineers.com. Accessed: 2017-11-09.

[11] Python General. http://pythoncentral.io. Accessed: 2017-11-09.

[12] Python Kurs. http://www.python-kurs.eu. Accessed: 2017-11-09.

[13] Python Modules. https://pymotw.com. Accessed: 2017-11-09.

[14] Python Programming. https://pythonprogramming.net. Accessed: 2017-11-09.

[15] SQLite. https://www.sqlite.org/. Accessed: 2017-11-09.

[16] SQLite Browser. http://sqlitebrowser.org/. Accessed: 2017-11-09.

[17] Stackoverflow. https://stackoverflow.com/. Accessed: 2017-11-09.

[18] Twisted Matrix. https://twistedmatrix.com/. Accessed: 2017-11-09.

[19] Ubuntu Forums. https://ubuntuforums.org. Accessed: 2017-11-09.

[20] Wiki Python. https://wiki.python.org. Accessed: 2017-11-09.

[21] Wikipedia. https://en.wikipedia.org. Accessed: 2017-11-09.

[22] *Twisted Network Programming Essentials*. O'Reilly Media, 2013.

[23] *Creating Apps in Kivy*. O'Reilly Media, 2014.

[24] Marco Huymajer. Cluster Detection Algorithm to Study Single Charge Trapping Events in TDDS. Diploma thesis, TU Wien, 2016.

[25] Michael Waltl. Change Point Detection in Time Dependent Defect Spectroscopy Data. Diploma thesis, TU Wien, 2011.

[26] Michael Waltl. Characterization of Bias Temperature Instabilities in Modern Transistor Technologies. Phd thesis, TU Wien, 2016.