

BACHELOR'S THESIS 2022

# Generating user interfaces for ROS-based robots

Hannes Lundh

Elektroteknik  
Datateknik

ISSN 1650-2884

LU-CS-EX: 2022-20

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY





KANDIDATARBETE  
Datavetenskap

LU-CS-EX: 2022-20

**Generating user interfaces for  
ROS-based robots**

**Hannes Lundh**



---

# Generating user interfaces for ROS-based robots

---

Hannes Lundh  
dat15hlu@student.lu.se

May 20, 2022

Bachelor's thesis work carried out at  
the Department of Computer Science, Lund University.

Supervisor: Jacek Malec, jacek.malec@cs.lth.se

Examiner: Maj Stenmark, maj.stenmark@cs.lth.se



## **Abstract**

The aim of this bachelor's thesis is to simplify the process to make a usable graphical user interface (GUI) for different ROS driven robots. The approach is expected to work for many different robots, and on many different platforms, such as PC, Android and IOS. The architecture was based on Kivy and KivyMD and a domain-specific language (DSL) was made for it. KIVY and KIVYMD are both PYTHON libraries containing resources for making portable user interfaces. To demonstrate the approach, a basic GUI was constructed using the DSL and then tested on a number of units.

**Keywords:** DSL, GUI, ROS robots





# Acknowledgements

---

I would like to thank my supervisor Jacek Malec as well as my examiner Maj Stenmark for helping me with this project. Another special thanks go to Faseeh Ahmad and Alexander Dürr for helping me figure out how the simulation of the robots worked.



# Contents

---

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Background . . . . .	7
1.2	State-of-the-art when designing a DSL and GUI . . . . .	8
1.3	Goals . . . . .	10
<b>2</b>	<b>Methodology</b>	<b>11</b>
2.1	Method . . . . .	11
2.1.1	DSL description . . . . .	11
2.1.2	GUI Structure . . . . .	14
2.1.3	Evaluation . . . . .	15
2.2	Tools . . . . .	15
2.2.1	ROS . . . . .	15
2.2.2	Kivy . . . . .	16
<b>3</b>	<b>Results</b>	<b>17</b>
3.1	DSL . . . . .	17
3.2	ROS . . . . .	18
3.3	Example GUI DSL code . . . . .	20
3.4	Portability . . . . .	21
3.5	GIT . . . . .	22
<b>4</b>	<b>Discussion</b>	<b>23</b>
4.1	Summary . . . . .	24
	<b>Bibliography</b>	<b>25</b>
<b>5</b>	<b>Appendix</b>	<b>27</b>
5.1	main.py . . . . .	27
5.2	divide_layout.py . . . . .	30
5.3	robot_program.py . . . . .	31
5.4	command_buttons.py . . . . .	33

5.5	button_callbacks.py . . . . .	35
5.6	initialize_ur5.launch . . . . .	36
5.7	test_ur5_move.py . . . . .	37
5.8	ur5.py . . . . .	39
5.9	test_ur5.py . . . . .	40
5.10	initialize_panda.launch . . . . .	41
5.11	test_panda_move.py . . . . .	42
5.12	panda.py . . . . .	44
5.13	test_panda.py . . . . .	45

# Chapter 1

## Introduction

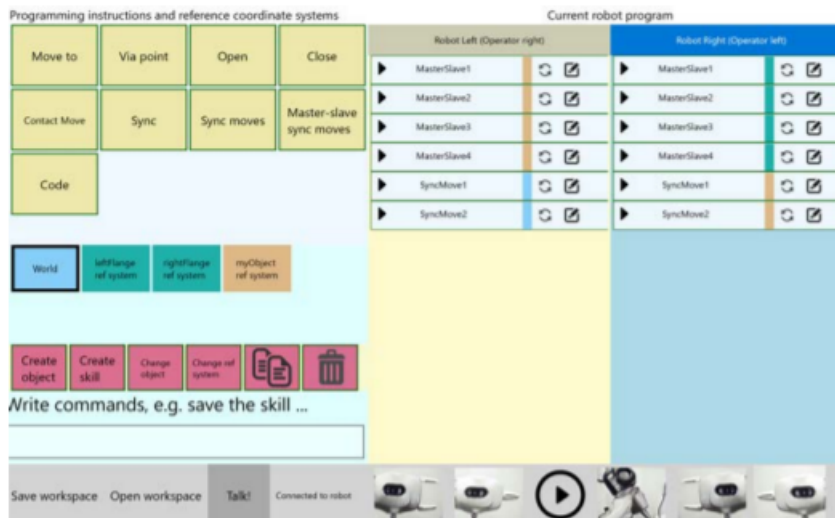
---

It is time consuming to create graphical user interfaces (GUIs). Often, an application should be run on multiple platforms, which requires that the same interface is implemented in different languages. Tools that simplify the GUI specification and generate platform dependent code have the potential to save programming time and effort. The goal of this project is to investigate a method for GUI-creation for ROS-based robots.

### 1.1 Background

The Robot Operating System (ROS, <http://www.ros.org>) is a framework made for writing portable robot software. It is used as a middleware in the communication between the application software provided by the user and the robot. The core components[14] of ROS contain two sub groups, communication infrastructure and tools. Communication infrastructure enables messages being sent between the components of a ROS-based system and includes, among others, standard messages sent to and from a robot. Among the tools one can find, for example, the robot description language. A particularly useful tool is *rviz*. *Rviz* is used to visualize the robot messages and simulate what the commands would make the robot do.

Another level between the robot and the user is a Graphical User Interface (GUI). A GUI is what the human user interacts with e.g., what is shown on a screen. In this project we will use the GUI showed in the article *Supporting Semantic Capture During Kinesthetic Teaching of Collaborative Industrial Robots*[16] as an inspiration to proceed from.



**Figure 1.1:** Screenshot of the GUI from [16]

A DSL is different from general-purpose languages like C, Java and Python, because it is made to be specialized to a particular application domain. An example of a DSL is e.g., Structured Query Language (SQL).

SQL is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data, i.e. data incorporating relations among entities and variables.[15]

## 1.2 State-of-the-art when designing a DSL and GUI

In order to design a DSL for a GUI one needs a domain model of what the GUI should do. A domain model explains how the GUI is supposed to look and function, including buttons, menus etc. Starting with a concrete example GUI and then extracting, simplifying and deriving the domain model is one way to do that. In the article *DSL-driven generation of Graphical User Interface*[1] the authors describe how a GUI can be said to be a definition of a DSL. They then create a tool called DEAL that traverses GUI applications and creates the domain model for that GUI. The authors later continued their research where they went from a domain model and created an iTask application meaning they can now do GUI  $\rightarrow$  domain model  $\rightarrow$  DSL  $\rightarrow$  GUI fully automatically.[2] In this project these steps will not be done automatically, however, the approach stays mostly the same.

In the article *Design Guidelines for Domain Specific Languages*[6] the authors propose several guidelines which they think should be used when creating DSLs. They define 5 categories of different types of guidelines, Language Purpose, Language Realization, Language Content, Concrete Syntax, Abstract Syntax.

- **Language Purpose** contains for example, *“Make your language consistent.”* where they propose:

DSLs are typically designed for a specific purpose. Therefore, each feature of a language should contribute to this purpose, otherwise it should be omitted.
- **Language Realization** contains for example, *“Decide carefully whether to use graphical or textual realization.”* where they propose that one should reflect early on whether one should use a graphical tools such as Eclipse Modeling Framework[4] or use a text-based one such as MontiCore[10].
- **Language Content** contains for example, *“Reflect only the necessary domain concepts.”*, *“Keep it simple.”*.
  - *“Reflect only the necessary domain concepts.”*:

This guideline proposes that only the tasks that the language is made for gets implemented. This in order to not complicate it.
  - *“Keep it simple.”*:

The importance of keeping it simple cannot be overstated. The sole reason a DSL is created is to simplify in this example the making of a GUI. If the DSL is not simple it defeats the purpose.
- **Concrete Syntax** contains for example *“Use descriptive notations.”* where they propose keeping the semantics of different reused notations or symbols. For example "+" should mean addition and that keywords should be easily identifiable.
- **Abstract Syntax** contains for example *“Enable modularity.”* where they propose that systems nowadays are very complex and having modularity is a way to simplify it.

While not all of these guidelines are relevant to this project a lot of them are and will be followed to some extent.

In the Bachelor’s thesis *Reducing implementation time for the GUI design-to-code process using DSL*[11] the author describes a workflow for designing GUIs. The workflow can be summarized in these steps:

1. Wireframes (low quality static representation)
2. Mockups (high quality static representation)
3. Assets (such as button shapes, images etc.)
4. Specification (specification of the different elements, such as their position, size, colour etc.)
5. The design document itself (alternatively the design document can be given straight away to the software engineer and then the previous steps are unnecessary)

6. Interactive prototypes (how the GUI is supposed to be used so that animations etc can be added)

In this project this workflow will be the basis of how the GUI will be created however it will not be strictly followed.

## 1.3 Goals

The primary goal of this project was to create a tool useful for developing robotic user interfaces.

It was decomposed into the following subgoals:

1. Design of a layered architecture for the tool so that all the necessary interfaces can be created in modular way.
2. Definition of a Domain-Specific Language (DSL) for specifying the structure and functionality of the GUI.
3. Creation of a proof-of-concept Graphical User Interface (GUI) prototype that is able to pass commands to the robot via a Robot Operating System (ROS) connection.
4. Generate code for many devices such as tablets, desktops and mobile phones.
5. Evaluate the design by using it for developing a known user interface[16]. The expectation is that the GUI will be using a tablet as the test hardware.



# Chapter 2

## Methodology

---

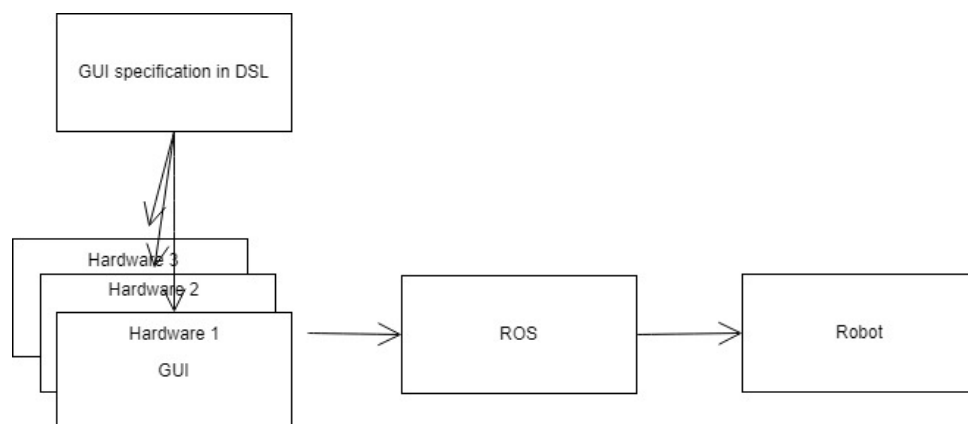
### 2.1 Method

The thesis methodology can be explained in the following steps:

- Designing/creating a DSL using the design science approach.
- Building a GUI using the DSL.
- Evaluating the GUI, its portability and functionality.

#### 2.1.1 DSL description

The system architecture is displayed in Figure 2.1.



**Figure 2.1:** Architecture of the system

---

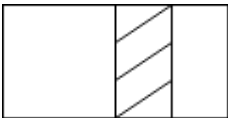
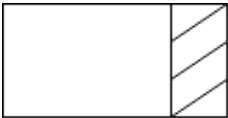
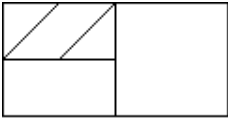
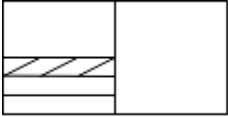
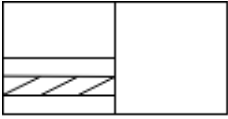
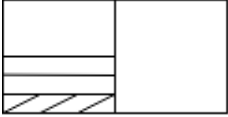

The idea was to be able to use the DSL in order to create multiple GUIs for different hardware.

The Design Science approach could be summarized as create, evaluate and iterate. The GUI from *Supporting Semantic Capture During Kinesthetic Teaching of Collaborative Industrial Robots*[16] was used to specify the requirements of the DSL. The requirements may be summarized as:

1. Specify the areas in a screen

Example what should be achieved:

Right half contains current robot program, left half contains programming tools

Visual area	Contains
	left robot arm program
	right robot arm program
	programming step buttons
	reference frame definitions and switching
	"File menu"
	Text input line
	Buttons for the physical activation of the robot and additional items

## 2. Graphical items on the screen

Example what should be achieved:

Item name	Description
ROBOT PROGRAM	swimlane (ladder), elements correspond to robot instructions, top to bottom order, instructions may be compound (some subroutine graphics, like indentation, should be possible)
PROGRAMMING STEP BUTTON	rectangle with a name
REFERENCE FRAME DEFINITION	rectangle with a name
FILE MENU ITEM	rectangle with a name
TEXT INPUT LINE	at least 40 characters long, one line high
PHYSICAL ACTIVATION BUTTONS	icons or text
MESSAGE AREA	Text

## 3. The visual requirements of a robot program instruction button:

- activation area for running just this instruction in the step-wise execution/debugging
- name
- type
- reference frame used
- edit activation possibility
- deletion possibility

Visualisation of current program pointer position is necessary!

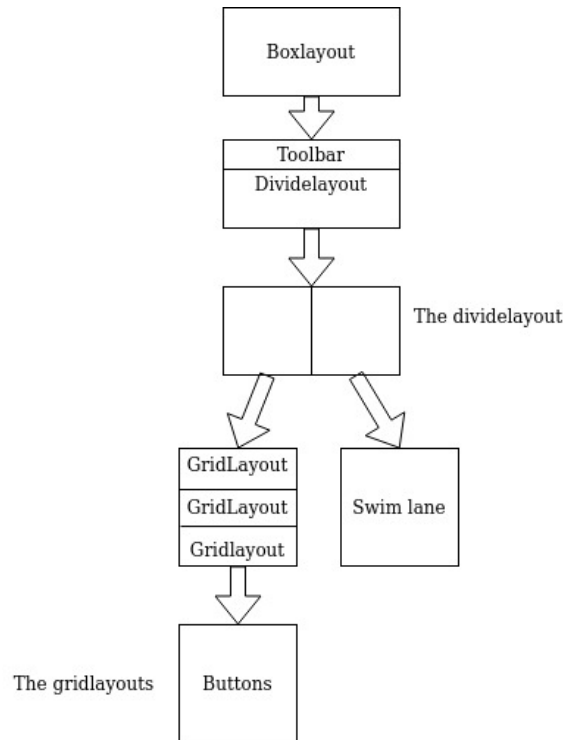
## 4. Actions for some elements (including buttons)

Example of what might be needed:

- activation of the current robot arm
- insertion of a specific command into the active arm's program at the current position
- change of the default frame used
- saving a set of instructions as a skill
- saving the current system state

## 2.1.2 GUI Structure

To test the DSL, a concrete example using it was made according to this structure, mimicking the one from Figure 1.1:



**Figure 2.2:** Structure of the GUI

The structure of the proof-of-concept is shown in Figure 2.2 the first step was to create the basic screen that will be the same for all robots, which was the base layer containing a boxlayout with a toolbar at the top. A boxlayout was used to partition widgets above/below with the orientation vertical or next to each other with the orientation horizontal. There are other arguments one can add like spacing, minimum height and padding etc. The toolbar has a drop down menu containing save and load. Below that is a divide layout which is a subclass to the boxlayout class in `kivy`, with the added function of being able to tell it how many partitions one wants when one creates it and the new functions to support that usage (see chapter 5.2). The way it was used here was to choose an orientation and number of areas one wants. So in the example GUI the orientation "horizontal" and size 2 was chosen. Which divides the screen area into two, from the top to the bottom (see second step in Figure 2.2). Then the function `add_widget_to_layout` was used to add another `dividelayout` class to the left part and a `robot_program` to the right. The `robot_program` is a subclass to the `kivy` class `BoxLayout` and includes a list of `RobotItems` which are a subclass to the `KivyMD` class `MDCardSwipe` (see chapter 5.3). The `dividelayout` on the left part was given the orientation "vertical" and the size 3. Then 3 `Gridlayouts` containing buttons were added to that `dividelayout`. `GridLayout` partitions the area in rows and columns. One can decide either the number of rows, the number of columns or both the number of rows and the number of columns when one creates the `gridlayout`. The buttons were all

created in three different files depending on their functionality (see chapter 5.4). Different colours were used to differentiate between them. All but a few of the buttons use the same placeholder function, since the ROS controller was not added, that function brings up a snackbar with the name of the button showing. The only exceptions were Test Panda and Test UR5, that have test programs that start when pressed (see chapter 5.5).

### 2.1.3 Evaluation

The following points was used when evaluating the GUI:

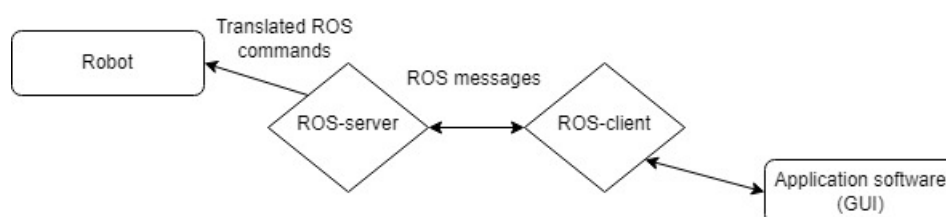
- Portability: could the GUI be ported to different platforms and how easily that can be done.
- Functionality: the GUI was used to control robots.

## 2.2 Tools

Some of the tools used in this thesis project have been named in the introduction. This entire project was made on a Linux virtual machine running Ubuntu 18.04, it was programmed in Python on Visual Studio Code. The GUI was run on a Samsung Galaxy Tab Active Pro SM-T545 as well in order to show portability.

### 2.2.1 ROS

ROS is a middleware in the communication between the application software provided by the user and the robot. It can be seen in the layered architecture of the project in figure 2.3:



**Figure 2.3:** Layered architecture

The ROS-server will publish a topic that the ROS-client will subscribe to. The ROS server in our case is the UR5 driver[18] and the panda driver[5] that were used in order to make the robots ROS compatible. The drivers uses the MoveIt Motion Planning Framework[3] in order to command the robot. This means the project only needs to program the controller/client side of the ROS connection. In order to test the ROS connection between the GUI and a robot a test program was written for two different robots, Franka Emika Panda[13] and UR5[19]. The test programs were run from a button press in the GUI.

## 2.2.2 Kivy

After looking at a couple of options, PyQT5[12], Tkinter[17] and Kivy[8], the option chosen was Kivy with the added collection of material design compliant widgets that KivyMD offers.

Kivy is a free and open source Python framework for developing mobile apps and other multitouch application software with a natural user interface (NUI). It is distributed under the terms of the MIT License, and can run on Android, iOS, Linux, macOS, and Windows.[7]

KivyMD is a version of the Kivy framework, however KivyMD follows Material Design which is a design method created by Google to help developers build high-quality digital experiences for Android, iOS, Flutter, and the web[9].

Kivy as well as KivyMD have very helpful websites where all of the different widgets and their uses are shown.<sup>1</sup>

---

<sup>1</sup>For Kivy the site was <https://kivy.org/doc/stable/> and for KivyMD the site was <https://kivymd.readthedocs.io/en/latest/components/>.

# Chapter 3

## Results

---

### 3.1 DSL

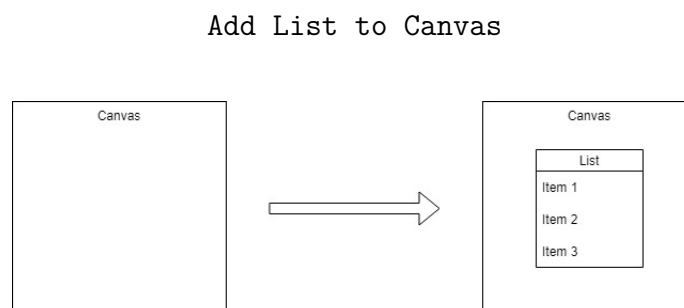
The DSL has 5 commands: base, swimlane, Add and Create. The reason base and swimlane are not capitalized is because they create predefined widgets while the other commands are used as functions that can change depending on what arguments you give them. The base command will create the base layer, which consists of a toolbar on top and a dividelayout below(as seen on the second step in Figure 2.2).

The swimlane command will create the swim lane widget, which can be seen on the right side of the GUI in Figure 3.6.

The Add command will be structured like this:

```
Add widget_to_be_added to widget_that_will_be_added_to
```

And as implied it will add a widget to another widget. Example:



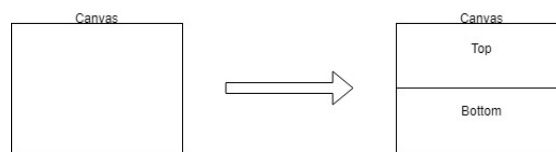
**Figure 3.1:** Example of Add

The Create command has the following structure for divide layout:

```
create dividelayout layout_name file_containing_information
  (orientation, number of parts, name of part(as many as there are parts with commas in between them))
```

It divides a layout in the orientation, either vertical or horizontal, in x number of parts. The parts are DivideLayouts with the names given after the comma. Example:

```
Create dividelayout Canvas file_containing_this (vertical, 2, Top, Bottom)
```



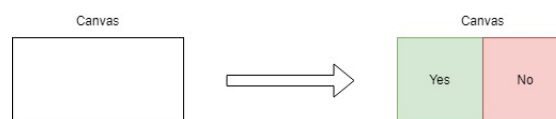
**Figure 3.2:** Example of Divide

The Create command has the following structure for button layout:

```
Create buttonlayout layout_name file_containing_buttons
```

This creates a buttonlayout with the buttons made in the file. Example:

```
Create buttonlayout answer_buttons file_containing_buttons_yes_and_no
```

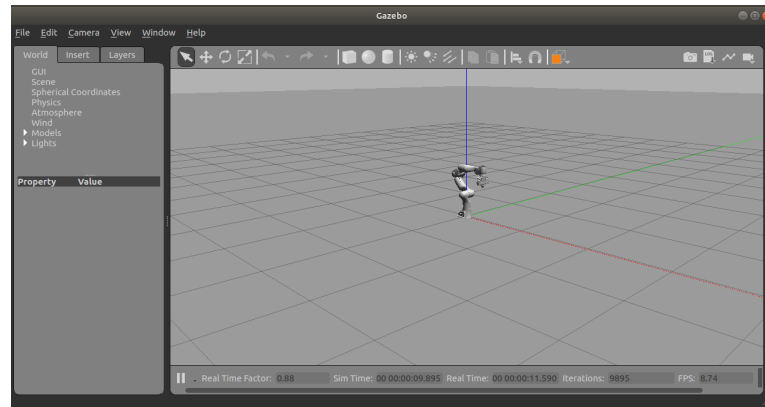


**Figure 3.3:** Example of Divide

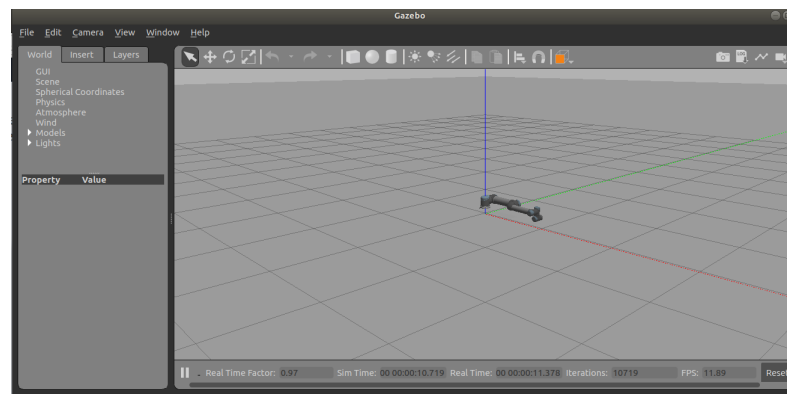
## 3.2 ROS

The way the ROS communication is handled in this project is by first initializing the robot simulations using their ROS drivers (see chapters 5.6 and 5.10) then a test program was written that sends the commands to the robot simulation (see chapters 5.7-5.9 and 5.11-5.13). It is therefore completely independent from the GUI, which means the GUI could be used for many robots as well as the same robot could be used for many GUIs. Here are some images of the two different robot simulations:



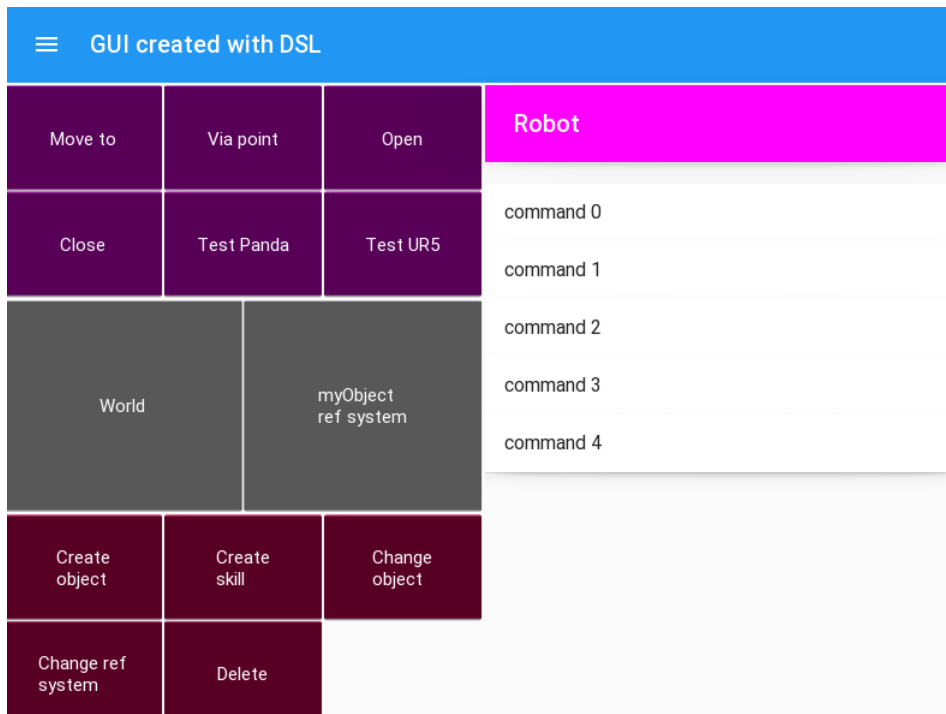


**Figure 3.4:** Picture of the panda simulation



**Figure 3.5:** Picture of the UR5 simulation

### 3.3 Example GUI DSL code



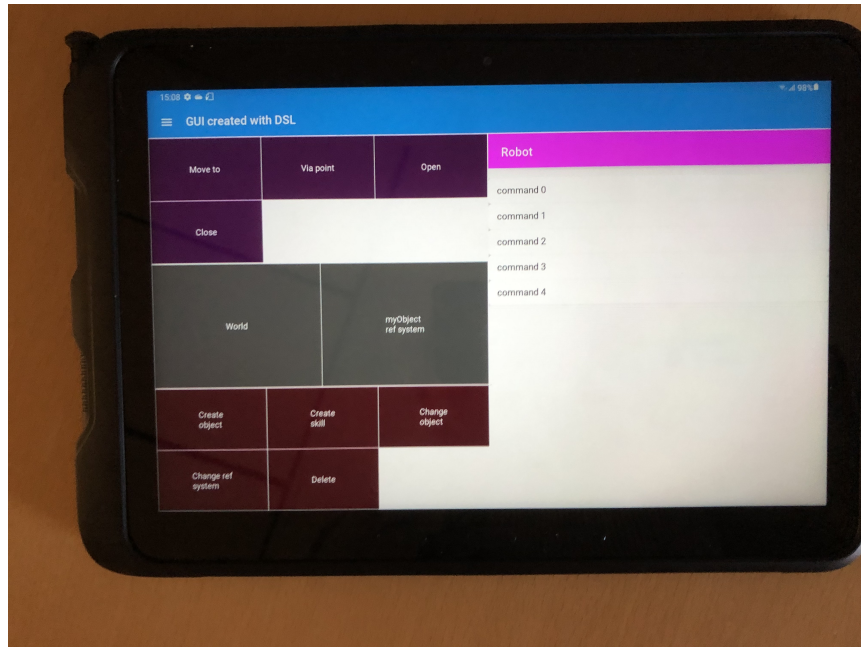
**Figure 3.6:** Screenshot of the GUI

The commands to build this GUI with the DSL are the following (see Figure 2.2 and the abstraction of the DSL in chapter 3.1 for reference):

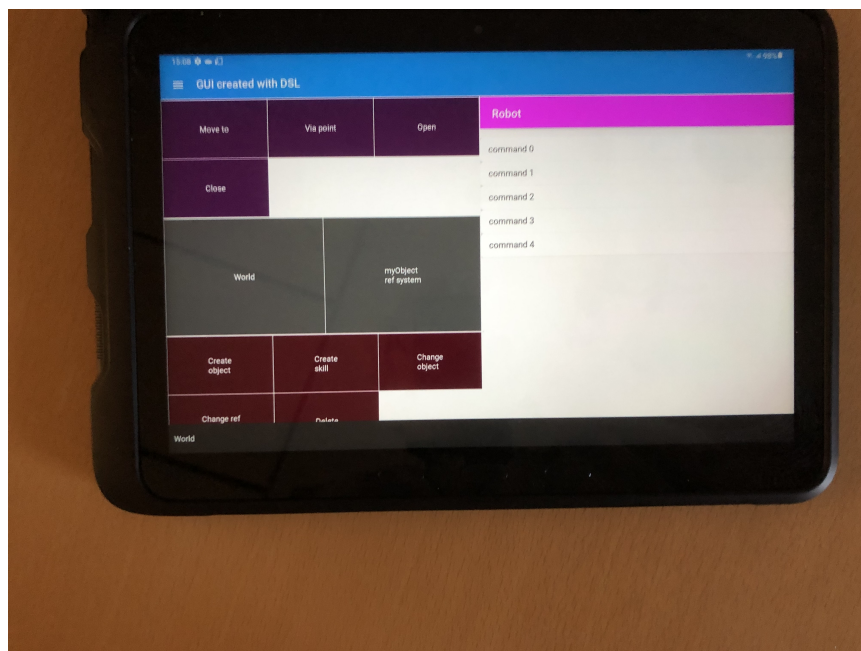
```
Add base to screen (which is the boxlayout with the toolbar)
Create dividelayout baselayout (vertical, 2, buttons, swim_lane)
(creates the dividelayout)
Add baselayout to base
Create dividelayout buttonslayout (horizontal, 3, command, reference,
programming) (creates the dividelayout for the gridlayouts)
Add buttonslayout to buttons
Add swimlane to swim_lane
Create buttonlayout command_layout command_buttons.py (creates
a gridlayout with all the buttons in command_buttons.py)
Create buttonlayout reference_layout reference_buttons.py (cre-
ates a gridlayout with all the buttons in reference_buttons.py)
Create buttonlayout programming_layout programming_buttons.py
(creates a gridlayout with all the buttons in programming_buttons.py)
Add command_layout to command
Add reference_layout to reference
Add programming_layout to programming
```

## 3.4 Portability

The implementation was tested on a Samsung tablet, see Figure 3.7 and 3.8



**Figure 3.7:** Picture of the app running



**Figure 3.8:** Picture of app after the World button was pressed



**Figure 3.9:** Picture of the Demo GUI app

## 3.5 GIT

The entire code base as well as a DSL sketch and a quick guide on how to run the GUI can be found here:

<https://git.cs.lth.se/jacek/agenericgui>.

# Chapter 4

## Discussion

---

In this thesis project it has been shown that a DSL could be used to generate a GUI. The GUI could be used to send commands to ROS based robots as shown with the test programs for the Franka Emika Panda and UR5 robots. It could also be ported to different platforms in this project it was used in Ubuntu as well as Android. Looking into the future of this project the next step would be to create the translator for the DSL so that it can be used to create GUIs. Another thing would be to make it so that the GUI creates the robot programs as well and not just runs them. Since the connection between ROS and the GUI is already made that should not be too difficult. The only thing that needs to be done is to create a ROS controller that later ties into the GUI through buttons.

Below are some pros and cons to using this approach in order to simplify the creation of the GUIs for ROS-based robots.

pros:

- It is fast to create a GUI with a DSL since it is designed for that purpose.
- By having the same GUI be portable to all the different platforms one saves time by not having to program the entire GUI from scratch, in a different programming language depending on the platform.

cons:

- One loses a lot of functionality of the original libraries, in this case Kivy and KivyMD, since the DSL, on purpose, limits the options in order to be more efficient at what it is made for.

## 4.1 Summary

In summary this project completed all the subgoals set at the beginning of the project.

1. Design of a layered architecture for the tool so that all the necessary interfaces can be created in modular way.

This was achieved and is shown in Figure 2.1. The structure can be modular for each robot as the GUI can change independently from the robot.

2. Definition of a Domain-Specific Language (DSL) for specifying the structure and functionality of the GUI.

The basic structure of the DSL is shown in the chapter 3.3.

3. Creation of a proof-of-concept Graphical User Interface (GUI) prototype that is able to pass commands to the robot via a Robot Operating System (ROS) connection.

This was achieved by the Test Panda and Test UR5 buttons that sets up a simulation for the robots and makes them move, as previously mentioned in chapter 3.2. The simulations can be seen in figures 3.4 and 3.5.

4. Generate code for many devices such as tablets, desktops and mobile phones.

Just by using Kivy/KivyMD for the GUI this goal was achieved automatically. As shown in Chapter 3.4 and in Figures 3.7, 3.8 and 3.9.

5. Evaluate the design by using it for developing a known user interface[16]. The expectation is that the GUI will be using a tablet as the test hardware.

This was achieved since the user interface will work but the expected functionality i.e., all the buttons should work as they did on the original GUI, does not. In order for the buttons to work a ROS controller needs to be programmed and then the functionality could be done as well.

# Bibliography

---

- [1] Michaela Baciková, Jaroslav Porubán, and Dominik Lakatos. “Defining Domain Language of Graphical User Interfaces”. In: *2nd Symposium on Languages, Applications and Technologies*. Ed. by José Paulo Leal, Ricardo Rocha, and Alberto Simões. Vol. 29. OpenAccess Series in Informatics (OASICS). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 187–202. ISBN: 978-3-939897-52-1. DOI: 10.4230/OASICS.SLATE.2013.187. URL: <http://drops.dagstuhl.de/opus/volltexte/2013/4038>.
- [2] Michaela Bačíková and Jaroslav Porubán. “DSL-driven generation of Graphical User Interfaces”. In: *Open Computer Science* 4.4 (2014), pp. 204–221. DOI: doi:10.2478/s13537-014-0210-9.
- [3] Chitta Coleman Şucan and Correll. “Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study”. In: *Journal of Software Engineering for Robotics*. Vol. 5. 2014, pp. 3–16. DOI: 10.6092/JOSER\_2014\_05\_01\_p3.
- [4] *Eclipse Modeling Project | The Eclipse Foundation*. <https://www.eclipse.org/modeling/emf/>. Accessed: 2022-04-12.
- [5] *franka\_ros*. [https://github.com/frankaemika/franka\\_ros](https://github.com/frankaemika/franka_ros). Accessed: 2022-04-08.
- [6] Gabor Karsai et al. “Design Guidelines for Domain Specific Languages”. In: *CoRR* abs/1409.2378 (2014). URL: <http://arxiv.org/abs/1409.2378>.
- [7] *Kivy (framework) - Wikipedia*. [https://en.wikipedia.org/wiki/Kivy\\_\(framework\)](https://en.wikipedia.org/wiki/Kivy_(framework)). Accessed: 2022-02-07.
- [8] *Kivy: Cross-platform Python Framework for NUI Development*. <http://kivy.org>. Accessed: 2021-11-24 - 2021-12-22.
- [9] *Material Design*. <https://material.io/design/introduction>. Accessed: 2022-01-21.
- [10] *MontiCore*. <https://monticore.github.io/monticore/>. Accessed: 2022-04-12.

- [11] Charilaos Mulder. “Reducing implementation time for the GUI design-to-code process using DSL”. Bachelor’s thesis. Universitet van Amsterdam, Sept. 2016.
- [12] *PyQt5 \* PyPI*. <https://pypi.org/project/PyQt5/>. Accessed: 2022-02-07.
- [13] *Robot System*. <https://www.franka.de/robot-system>. Accessed: 2022-05-01.
- [14] *ROS.org | Core Components*. <https://web.archive.org/web/20211024082252/https://www.ros.org/core-components/>. Accessed: 2021-09-18.
- [15] *SQL - Wikipedia*. <https://en.wikipedia.org/wiki/SQL>. Accessed: 2021-09-18.
- [16] Maj Stenmark et al. “Supporting Semantic Capture during Kinesthetic Teaching of Collaborative Industrial Robots”. In: *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*. 2017, pp. 366–371. DOI: 10.1109/ICSC.2017.40.
- [17] *tkinter - Python interface to Tcl/Tk*. <https://docs.python.org/3/library/tkinter.html>. Accessed: 2022-02-07.
- [18] *Universal Robots ROS drivers*. [https://github.com/ros-industrial/universal\\_robot](https://github.com/ros-industrial/universal_robot). Accessed: 2021-09-19.
- [19] *UR5e => Flexibel automationsplattform | Universal robots*. <https://www.universal-robots.com/se/produkter/ur5-robot/>. Accessed: 2022-05-01.



# Chapter 5

## Appendix

---

### 5.1 main.py

```
#!/usr/bin/env python3

import roslaunch
from kivy.uix.boxlayout import BoxLayout
from kivymd.app import MDApp
from kivymd.uix.snackbar.snackbar import Snackbar
from kivymd.uix.toolbar import MDToolbar
from kivymd.uix.menu import MDDropdownMenu
from kivy.uix.gridlayout import GridLayout
from kivy.metrics import dp
from items.divide_layout import DivideLayout
from items.command_buttons import *
from items.reference_buttons import *
from items.programming_buttons import *
from items.robot_program import RobotItem, RobotProgram

class MyApp(MDApp):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def open_menu(self, button):
        self.menu.caller = button
```

```
self.menu.open()

def open_robot_menu(self, button):
    self.robot_menu.caller = button
    self.robot_menu.open()

def menu_callback(self, text_item):
    self.menu.dismiss()
    Snackbar(text=text_item).open()

def build(self):
    root = BoxLayout(orientation='vertical', spacing=2)

    layout = DivideLayout(2, orientation='horizontal')

    menu_items = [
        {
            "viewclass": "OneLineListItem",
            "text": "Open",
            "height": dp(56),
            "on_release": lambda x="Open": self.menu_callback(x)
        },
        {
            "viewclass": "OneLineListItem",
            "text": "Save",
            "height": dp(56),
            "on_release": lambda x="Save": self.menu_callback(x)
        }
    ]
    self.menu= MDDropdownMenu(
        items=menu_items,
        width_mult=4
    )

    toolbar = MDToolbar(title="GUI_created_with_DSL")
    toolbar.left_action_items=[["menu", lambda x: self.open_menu(x)]]

    layout.add_widget(toolbar)

    button_layout1 = GridLayout(cols=3)
    button_layout1.add_widget(MoveTo())
    button_layout1.add_widget(ViaPoint())
    button_layout1.add_widget(Open())
    button_layout1.add_widget(Close())
    button_layout1.add_widget(Test_panda())
```

```
button_layout1.add_widget(Test_ur5())

button_layout2 = GridLayout(cols=3)
button_layout2.add_widget(World())
button_layout2.add_widget(MyObjRef())

button_layout3 = GridLayout(cols=3)
button_layout3.add_widget(CreateObject())
button_layout3.add_widget(CreateSkill())
button_layout3.add_widget(ChangeObject())
button_layout3.add_widget(ChangeRefSystem())
button_layout3.add_widget(RemoveButton())

layout.add_widget_to_layout(
    DivideLayout(3, orientation='vertical'), 0)
left = layout.get_widget(0)
left.add_widget_to_layout(button_layout1, 0)
left.add_widget_to_layout(button_layout2, 1)
left.add_widget_to_layout(button_layout3, 2)

robot_program = RobotProgram("Robot")
for i in range(5):
    robot_program.add_item(RobotItem(
        robot_program, text= f"command_{i}"))

layout.add_widget_to_layout(robot_program, 1)

root.add_widget(toolbar)
root.add_widget(layout)
return root

MyApp().run()
```

## 5.2 divide\_layout.py

```
#!/usr/bin/env python3

from kivy.uix.boxlayout import BoxLayout

class DivideLayout(BoxLayout):

    def __init__(self, number_of_parts, **kwargs):
        super().__init__(**kwargs)
        self.widgets=[None]*number_of_parts
        self.number_of_parts=number_of_parts
        self.spacing = 2

    def add_widget_to_layout(self, widget, index):
        if (self.number_of_parts>index):
            self.widgets[index]=widget
            self.__update_widgets()
        else:
            raise IndexError('outside_of_range')

    def remove_widget_from_layout(self, index):
        self.widgets[index]=None
        self.__update_widgets()

    def __update_widgets(self):
        self.clear_widgets()
        for i in range(0, self.number_of_parts):
            if (self.widgets[i] != None):
                self.add_widget(self.widgets[i])
            else:
                self.add_widget(BoxLayout())

    def get_widget(self, index):
        return self.widgets[index]
```

## 5.3 robot\_program.py

```
#!/usr/bin/env python3

from kivy.uix.boxlayout import BoxLayout
from kivy.uix.scrollview import ScrollView

from kivymd.uix.button.button import MDIconButton
from kivymd.uix.card import MDCardSwipe
from kivymd.uix.card.card import MDCardSwipeFrontBox, MDCardSwipeLayerBox
from kivymd.uix.list.list import OneLineListItem
from kivymd.uix.toolbar import MDToolbar
from kivymd.uix.list import MDList

from items.button_callbacks import callback

class RobotProgram(BoxLayout):

    def __init__(self, title, **kwargs):
        super().__init__(**kwargs)
        self.orientation='vertical'
        self.spacing="10dp"

        self.toolbar = MDToolbar(
            elevation=10,
            title=title,
            md_bg_color = (1,0,1,1)
        )

        self.scrollview = ScrollView(scroll_timeout=100)
        self.md_list = MDList(padding=0)
        self.scrollview.add_widget(self.md_list)

        self.add_widget(self.toolbar)
        self.add_widget(self.scrollview)

    def add_item(self, item):
        self.md_list.add_widget(item)

    def remove_item(self, item):
        self.md_list.remove_widget(item)
```

```
class RobotItem(MDCardSwipe):

    def __init__(self, robot_program, text, **kw):
        super ().__init__ (**kw)

        self.robot_program = robot_program

        self.layer_box = MDCardSwipeLayerBox(padding="8dp")
        self.delete_icon = MDIconButton(
            icon="trash-can",
            pos_hint={'center_y': .5},
            on_release=callback
        )

        self.layer_box.add_widget(self.delete_icon)

        self.front_box = MDCardSwipeFrontBox()
        self.content=OneLineListItem(
            text=text,
            _no_ripple_effect = True
        )
        self.front_box.add_widget(self.content)

        self.size_hint_y=None
        self.height=self.content.height

        self.add_widget(self.layer_box)
        self.add_widget(self.front_box)
```

## 5.4 command\_buttons.py

```
#!/usr/bin/env python3

from kivy.uix.button import Button
from items.button_callbacks import callback, test_panda, test_ur5

class MoveTo(Button):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.background_color = (1, 0, 1, 1)
        self.text = "Move_to"
        self.bind(on_press=callback)

class ViaPoint(Button):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.background_color = (1, 0, 1, 1)
        self.text = "Via_point"
        self.bind(on_press=callback)

class Open(Button):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.background_color = (1, 0, 1, 1)
        self.text = "Open"
        self.bind(on_press=callback)

class Close(Button):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.background_color = (1, 0, 1, 1)
        self.text = "Close"
        self.bind(on_press=callback)

class Test_panda(Button):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.background_color = (1, 0, 1, 1)
        self.text = "Test_Panda"
```

```
        self.bind(on_press=test_panda)

class Test_ur5(Button):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.background_color = (1, 0, 1, 1)
        self.text = "Test_UR5"
        self.bind(on_press=test_ur5)
```



## 5.5 button\_callbacks.py

```
#!/usr/bin/env python3
import roslaunch
from kivy.md.uix.snackbar import Snackbar

def callback(button):
    text = button.text if button.text else "delete"
    Snackbar(text= text ).open()

def test_panda(self):
    uuid = roslaunch.rlutil.get_or_generate_uuid(None, False)
    roslaunch.configure_logging(uuid)
    launch = roslaunch.parent.ROSLaunchParent(uuid,
        ["/home/hannes/catkin_ws/src/motion_scripts/launch/test_panda.launch"])
    launch.start()

def test_ur5(self):
    uuid = roslaunch.rlutil.get_or_generate_uuid(None, False)
    roslaunch.configure_logging(uuid)
    launch = roslaunch.parent.ROSLaunchParent(uuid,
        ["/home/hannes/catkin_ws/src/motion_scripts/launch/test_ur5.launch"])
    launch.start()
```

## 5.6 initialize\_ur5.launch

```
<launch>
  <include file="$(find_ur_gazebo)/launch/ur5.launch" />
  <include
    file="$(find_ur5_moveit_config)/launch/ur5_moveit_planning_execution.launch" />
</launch>
```

## 5.7 test\_ur5\_move.py

```
#!/usr/bin/env python

import rospy, sys
import moveit_commander
from geometry_msgs.msg import Pose
from copy import deepcopy
from std_msgs.msg import Header

from trajectory_msgs.msg import JointTrajectory

from trajectory_msgs.msg import JointTrajectoryPoint

endpoints = [[0.0, -1, 1.5, 1.0, 0, -1]]

def main():

    rospy.init_node('test_ur5_move')
    pub = rospy.Publisher('/arm_controller/command',
                          JointTrajectory,
                          queue_size=10)

    arm = moveit_commander.MoveGroupCommander('manipulator')
    endpoints.append(arm.get_current_joint_values())

    # Create the topic message
    traj = JointTrajectory()
    traj.header = Header()
    # Joint names for UR5
    traj.joint_names = ['shoulder_pan_joint', 'shoulder_lift_joint',
                       'elbow_joint', 'wrist_1_joint', 'wrist_2_joint',
                       'wrist_3_joint']

    rate = rospy.Rate(1)
    move_away_from_all_zeros = True
    pts = JointTrajectoryPoint()
    traj.header.stamp = rospy.Time.now()

    while not rospy.is_shutdown():

        if move_away_from_all_zeros:
            pts.positions = endpoints[0]
```

```
        move_away_from_all_zeros = False
    else:
        pts.positions = endpoints[1]
        move_away_from_all_zeros = True

    pts.time_from_start = rospy.Duration(1.0)

    # Set the points to the trajectory
    traj.points = []
    traj.points.append(pts)
    # Publish the message
    pub.publish(traj)
    rate.sleep()

if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        print ("Program_interrupted_before_completion")
```

## 5.8 ur5.py

```
#!/usr/bin/env python
import roslaunch

class Ur5():
    def __init__(self):
        uuid = roslaunch.rlutil.get_or_generate_uuid(None, False)
        roslaunch.configure_logging(uuid)
        self.launch = roslaunch.parent.ROSLaunchParent(uuid, [
            "/home/hannes/catkin_ws/src/motion_scripts/launch/initialize_ur5.launch"])
        self.test_launch = roslaunch.parent.ROSLaunchParent(uuid, [
            "/home/hannes/catkin_ws/src/motion_scripts/launch/test_ur5_move.launch"])
        self.ur5_running = False
        self.test_running = False

    def connect(self):
        if not self.ur5_running:
            self.launch.start()
            self.ur5_running = True

    def disconnect(self):
        if self.ur5_running:
            self.launch.shutdown()
            self.ur5_running = False

    def start_test_move(self):
        if self.ur5_running and not self.test_running:
            self.test_launch.start()
            self.test_running = True

    def end_test_move(self):
        if self.ur5_running and self.test_running:
            self.test_launch.shutdown()
            self.test_running = False
```

## 5.9 test\_ur5.py

```
#!/usr/bin/env python
```

```
import rospy
import sys
sys.path.append("/home/hannes/catkin_ws/src/motion_scripts/gui")
from items.ur5 import Ur5
```

```
def main():
    rospy.init_node("test_ur5")
    ur5 = Ur5()
    ur5.connect()
    rospy.sleep(20)
    ur5.start_test_move()
    rospy.sleep(20)
    ur5.end_test_move()
    ur5.disconnect()
```

```
if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        print ("Program_interrupted_before_completion")
```

## 5.10 initialize\_panda.launch

```
<launch>  
  <include file="$(find_panda_gazebo)/launch/panda_world.launch" />  
  <include file="$(find_panda_sim_moveit)/launch/sim_move_group.launch" />  
</launch>
```

## 5.11 test\_panda\_move.py

```
#!/usr/bin/python

import rospy
import numpy as np
from franka_interface import ArmInterface
from copy import deepcopy

endpoints = [[0,0,0,0,0,0], [0.0, -1, 1.5, 1.0, 0, -1]]

def main():
    rospy.init_node("test_panda_move")
    r = ArmInterface()

    rate = rospy.Rate(400)

    elapsed_time_ = rospy.Duration(0.0)
    period = rospy.Duration(0.005)

    r.move_to_neutral() # move to neutral pose before beginning

    initial_pose = deepcopy(r.joint_ordered_angles())

    vals = deepcopy(initial_pose)
    count = 0

    while not rospy.is_shutdown():

        elapsed_time_ += period

        delta = 3.14 / 16.0 * (1 - np.cos(3.14 / 5.0 * elapsed_time_.to_sec())) \
            * 0.2

        for j, _ in enumerate(vals):
            if j == 4:
                vals[j] = initial_pose[j] - delta
            else:
                vals[j] = initial_pose[j] + delta

        # r.set_joint_positions_velocities
        # (vals, [0.0 for _ in range(7)]) # for impedance control
        r.set_joint_positions(dict(zip(r.joint_names(), vals)))
```



```
count += 1
rate.sleep()

if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        print ("Program_interrupted_before_completion")
```

## 5.12 panda.py

```
#!/usr/bin/env python
```

```
import roslaunch
```

```
class Panda():
```

```
    def __init__(self):
```

```
        uuid = roslaunch.rlutil.get_or_generate_uuid(None, False)
```

```
        roslaunch.configure_logging(uuid)
```

```
        self.launch = roslaunch.parent.ROSLaunchParent(uuid, [
```

```
            "/home/hannes/catkin_ws/src/motion_scripts/launch/initialize_panda.launch"]
```

```
        self.test_launch = roslaunch.parent.ROSLaunchParent(uuid, [
```

```
            "/home/hannes/catkin_ws/src/motion_scripts/launch/test_panda_move.launch"])
```

```
        self.panda_running = False
```

```
        self.test_running = False
```

```
    def connect(self):
```

```
        if not self.panda_running:
```

```
            self.launch.start()
```

```
            self.panda_running = True
```

```
    def disconnect(self):
```

```
        if self.panda_running:
```

```
            self.launch.shutdown()
```

```
            self.panda_running = False
```

```
    def start_test_move(self):
```

```
        if self.panda_running and not self.test_running:
```

```
            self.test_launch.start()
```

```
            self.test_running = True
```

```
    def end_test_move(self):
```

```
        if self.panda_running and self.test_running:
```

```
            self.test_launch.shutdown()
```

```
            self.test_running = False
```

## 5.13 test\_panda.py

```
#!/usr/bin/env python

import rospy
import sys
sys.path.append("/home/hannes/catkin_ws/src/motion_scripts/gui")
from items.panda import Panda

def main():
    rospy.init_node("test_panda")
    panda = Panda()
    panda.connect()
    rospy.sleep(20)
    panda.start_test_move()
    rospy.sleep(20)
    panda.end_test_move()
    panda.disconnect()

if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        print ("Program_interrupted_before_completion")
```

**BACHELOR'S THESIS** Generating user interfaces for ROS-based robots**STUDENT** Hannes Lundh**SUPERVISOR** Jacek Malek (LTH)**EXAMINER** Maj Stenmark (LTH)

# Is it possible to simplify the creation of Graphical User Interfaces (GUI) for ROS-based robots?

POPULAR SCIENCE ARTICLE **Hannes Lundh**

Creating GUIs has always been a hassle, where you have to choose between many different languages, packages etc. And when you finally have chosen one you have to learn how they work even if almost everything is not relevant for you. This project aimed to simplify this for specifically GUIs for ROS-based robots.

A quick explanation of what a GUI is would be to say everything a user sees and interacts with on their devices screen is a GUI. In order to simplify the process of creating a GUI, I first started by looking at a GUI that was already in use at the Robot Lab at LTH. The purpose of looking at the GUI was to see what basic building blocks were necessary for a GUI that is used for controlling robots. From those basic building blocks a Domain-Specific Language (DSL) could be defined. A DSL could be explained as a language created for one purpose only, in this case creating GUIs.

A second part of the project was also controlling a robot from the GUI. In order to test if it was possible to control a robot from the GUI a test GUI was created. While the GUI created was not made from the DSL, as the translation between the DSL and the chosen language and packages is not yet made, it was made such that the DSL would be able to create it. Since I had no access to a real robot, a simulation was used instead. The test GUI showed in Figure 1 has a Test UR5 button, UR5 being the robot. When pressed a simulation of the robot starts and runs through a program

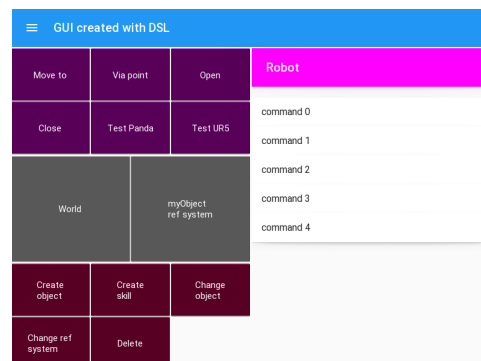


Figure 1: Screenshot of the GUI

after which it closes the simulation. The project has shown that the defined DSL can be used to create a GUI capable of controlling a robot. However, for it to be used some more work needs to be done. The translation between the DSL and the chosen language and packages is still not completed, and for every new robot a ROS controller for that specific robot needs to be created. Once that is complete, this project will help everyone who wants to create a personalized GUI for controlling their ROS-based robots.