UnifiedGUI

Benjamin Schiller

October 2021

1 Summary

UnifiedGUI is an abstract implementation of a graphical user interface which can be used for a variety of different purposes in the context of molecular communication.

2 Motivation

The goal of UnifiedGUI is to avoid redundant implementation of features that are required to visualize a transmission/receiving scheme in molecular communication. When trying out a new way of communication, the user should not have to worry about the visualization and data handling, but only about the hardware itself and the communication to it.
To create a GUI that can be used for a variety of different applications, it requires for a very abstract and dynamic implementation – for example the number of receiver and therefore the number of tables/datalines in the plot is only known at runtime. Because of this, UnifiedGUI may look a bit complicated at the first glance. This document offers a detailed description of the software and hopefully provides a comprehensive understanding.

3 Structure

3.1 Model

As mentioned in Section **??**, the implementation is done in an abstract way to allow for high reusability.
There are four classes that can be implemented for the user's purposes: encoders, transmitters, decoders and receivers.

- An encoder consists of n transmitters.
- A transmitter is used to control the input to a communication channel. For instance, this can be a pump.

Level	Description	Representation	Example
Sequence	High-level sequence	Non-primitive data type	”A”
Symbol	Symbol value	Integer	01000001
Physical	Application dependent value	Float	$1.0, 1.1, 1.5, 0.9, ...$

Table 1: Caption

- A receiver is used to generate sensor data from a communication channel. Each receiver can have multiple sensors. For example, a receiver can be a color sensor, where the multiple sensors are the measured colors.
- A decoder consists of m receivers. In the simplest case, the decoder just collects data from the receivers and stores them. But a decoder can provide more functionality, as it allows for further processing to retrieve information from the data generated by the receivers.

3.2 Levels

Note that the user can individually choose what parts they want to implement. For instance, if the user is just interested in visualizing data, implementing a receiver and simple decoder may be sufficient and an encoder does not have to be defined.

3.3 View

3.4 Controller

4 How To Use

4.1 Requirements

1. Download and install the latest version of Python3 (`https://www.python.org/downloads/`).
2. Install the following packages using `python3 -m pip install <package>`.
   - `numpy`
   - `time`
   - `datetime`
   - `PyQt5`
   - `pyqtgraph`
   - `random`

4.2 Execution

UnifiedGUI can be run via `python3 main.py` in the terminal (make sure you are in the correct directory).

Alternatively, you can run the program with the `-O` (the capital letter) flag. This will generate in an 'optimized' mode which might be slightly faster. However, **no debug information** is shown in this mode which can make it hard to spot implementation errors!

Especially for debugging purposes, an IDE like PyCharm is recommended.

4.3 Using UnifiedGUI

4.4 Implement a New Transmitter

Not supported yet.

4.5 Implement a New Encoder

Not supported yet.

4.6 Implement a New Receiver

1. Choose a name for your receiver, e.g. "MyReceiver".
2. In the directory `Models/Implementations/Receivers`, create a new Python file with the **exact** name chosen in step 1, e.g. `MyReceiver.py`.
3. Open the receiver template (`Models/Templates/ReceiverTemplate.py`). Select everything in this file and copy it.
4. Open the file created in step 2.
5. Paste the template copied in step 3.
6. Rename the class. Class name must be the **exact** name chosen in step 1.
7. Set the class attributes and implement the required functions as described below.
8. Once you finished your implementation, test it by running the provided testcases (`python3 TestUnifiedGUI.py TestReceivers`).

Required:

- The member `num_sensors` is an integer that specifies how many sensors the receiver has.

- The function `listen` runs an infinite loop of checking for new values. If new values are available, they are appended as a tuple or a list to the member `buffer` (inherited by the interface) using the function `append_values` (inherited by the interface). Optionally, you can provide a timestamp corresponding to the time the measurement was generated. If you do not provide a timestamp, the current time is used as a timestamp.
  Note: If the receiver only has one sensor, it is still important to convert the measured value to a tuple or a list (of length 1). This can be done like this: `my_tuple = (value,)` or `my_list = [value]`.

Optional:

- The member `sensor_descriptions` is a list of strings that provide a meaningful description for each sensor.

For reference, you can have a look at the already implemented example receiver (`Models/Implementations/Receivers/ExampleReceiver.py`).

4.7 Implement a New Decoder

1. Choose a name for your receiver, e.g., "MyDecoder".
2. In the directory `Models/Implementations/Decoders`, create a new Python file with the **exact** name chosen in step 1, e.g., `MyDecoder.py`.
3. Open the decoder template (`Models/Templates/DecoderTemplate.py`). Select everything in this file and copy it.
4. Open the file created in step 2.
5. Paste the template copied in step 3.
6. Rename the class. Class name must be the **exact** name chosen in step 1.
7. Set the class attributes and implement the required functions as described below.
8. Once you finished your implementation, test it by running the provided testcases (`python3 TestUnifiedGUI.py TestDecoders`).

Required:

- The member `receiver_types` is a list containing the the types of receivers belonging to the decoder.

Optional:

- The constant `PARAMETERS` can be used to specify additional parameters which the user has to provide values for upon adding the decoder and can later edit. See Section 4.8 for more information.

- The function `calculate_landmarks` can be used to generate landmarks for edges, peaks etc. The landmarks can then be shown in the plot. If you want to use this, set the member `landmarks` accordingly. Landmarks are in the form of a dictionary with keys `x` for the position of the landmark on the x-axis (timestamp) and `y` for the position on the y-axis respectively. Since multiple landmarks sets are supported, `landmarks` is a list of dictionaries.
  Note: If you only want to display one set of landmarks, make sure `landmarks` is a list of dictionaries (of length 1). This can be done like this: `my_list = [my_landmark]`.

- The function `calculate_symbol_intervals` can be used to divide the signal into different interval where each intervals corresponds to one symbol. The symbol intervals can then be shown in the plot as vertical infinite lines. If you want to use this, set the member `symbol_intervals` to a list of timestamps. For instance, symbol intervals can either be constructed from fixed time intervals (for example every t milliseconds), from a fixed amount of samples (for example every n samples) or from landmarks (edges, peaks, etc.).
  Note: The first symbol intervals begins at the first timestamp provided and the last symbol intervals ends at the last timestamp provided, resulting in a total of $n-1$ intervals where n is the length of `symbol_intervals`.

- The function `calculate_symbol_values` can be used to assign a symbol to each symbol intervals. If you want to use this, first make sure to implement `calculate_symbol_intervals` correctly and set the member `symbol_values` accordingly (expects a list of strings). Also pay close attention where the first intervals starts and the last intervals ends. If implemented correctly, the length of `symbol_values` should be 1 smaller than the length of `symbol_intervals`.

- The function `calculate_sequence` can be used to decode the symbol values to a sequence.

4.8 Parameters

Parameters can be used can be used to specify additional parameters which the user has to provide values for upon adding the decoder and can later edit.

Parameters are optional, if you do not want to use any parameters, set `PARAMETERS = None` in your implementation or simply do not define it in the first place.

Parameters are defined as a list of dictionaries where each dictionary represents one parameter. Every parameter dictionary must contain the following keys:

- `description`: Description of the parameter.
- `default`: Default value of the parameter.

- `type`: Datatype of the parameters = {`bool`, `int`, `float`, `string`, `item`}

Depending on the datatype, additional keys may be required:

- `bool`:
  - No additional keys required.
- `int`:
  - `min` (`int`): Minimum value.
  - `max` (`int`): Maximum value.
- `float`:
  - `min` (`float`): Minimum value.
  - `max` (`float`): Maximum value.
  - `decimals` (`int`): Number of decimals.
- `string`:
  - `max_length` (`int`): Maximum length.
- `item`:
  - `items` (`list`): A list of selectable `string` items.

For reference, have a look at the parameters defined in the example decoder (`Models/Implementations/Decoders/ExampleDecoder.py`).