CAMBRIDGE UNIVERSITY ENGINEERING DEPARTMENT

Part IIA - GF2 Software
FINAL REPORT

Logic Simulator

Name: Jonty Page
College: Pembroke
Team Number: 16
CRDId: jp730
Date: 04/06/2019

Abstract
A client has contracted a software development company to create a logic simulation program. This program should be able to pass, simulate and display the output of logic circuits and meet a detailed set of requirements given by the client. The program is divided into eight functional modules, four of which are the responsibility of the team to develop, integrate and test. After three weeks of development, some changes and additions to the specifications document was introduced by the client which the team had to integrate. Overall, an out-of-the-box solution was developed which met all of the clients requirements, was easy and intuitive to use and robust to errors in execution and input.

1 Introduction

A software company was contracted by a client to produce a logic simulation program to enable the operation of both combinatorial and clocked logic circuits to be studied by a computer prior to their implementation in hardware. The client provided a detailed specifications document listing the requirements of the software and this can be found in appendix A of the CUED GF2 project main handout [1]. The general function of the software should be:

1. The program reads in a text file (".txt") which fully defines the logic elements, the connections between them, the generators for inputs and the signals to be monitored.
2. The user is then able to control the initial state of switches, add/remove points which are monitored and set the number of clock cycles for which the simulation should be run at will.
3. The network is executed and the output at each user-defined monitoring point is displayed.

There are two methods in which the user can interact with the software, either through a command-line user interface (UI) (which was already fully developed and functional) or through a graphical user interface (GUI) (which was the responsibility of the team to develop). In initial implementation, six logic elements and two generators (a clock and a switch) were to be supported. The circuit should be read into the software through a circuit definition file written in a logic description language created by the developers. The logic description language was to be described formally by EBNF notation and adhere to the specifications laid out in part A3 and A6 of the client requirements document (Our EBNF document can be found in Appendix B). The GUI was to have the same functionality as the UI and full error-checking was to be applied to user command inputs in-order to maintain robustness. Finally, there was to be no limit to the size of the circuit, number of monitoring points or devices except that implied by the memory of the computer.

2 Software Structure

The software is split into eight functional modules, four of which were already developed and four of which were developed by the team. These modules are described in Table 1 below:

Module Name	Description
Names	Maps variable names and string names to unique integers
Scanner	Translates sequence of characters in definition file into sequence of symbols.
Parse	Analyses syntactic and semantic correctness of scanned definition file.
Devices*	Contains routines for creating, configuring and querying devices.
Network*	Manages connections between devices and execution of the logic circuit.
Monitors*	Records signal levels at designated monitor points for each simulation cycle.
Userint*	Handles the text-based user interface.
Gui	Handles the graphical user interface

Table 1: Overview of eight functional modules within the Logic Simulator software. **indicates modules already fully developed.*

These modules are wrapped together using a top level python program *logsim.py* which is the program which should be run to start the software. Also, a ninth module was added by the development team, *errors.py*, in order to manage error reporting through an object which is populated when an error occurs.

The general workflow of the backend of the software is first four objects are created - Names, Devices, Network, Monitors. These objects store the circuit variables and contain all the information required to obtain the output signal for each defined monitoring point required by the GUI. The circuit definition file selected is then ran through the Scanner, which populates the Names object with unique IDs for each variable/string names. This Scanner object is then ran through the parser, alongside the Names, Devices, Network and Monitors objects, the file is checked for syntactic and semantic errors and if none arise, the circuit variables are loaded into each of the four objects as required. When the GUI or UI is asked to execute the circuit, or modify one of the circuit variables such as the state of a switch or the monitor points set, the interfaces make calls to each of the objects and communicate with the variables as needed. The figure below (Figure 1) shows graphically communications between each module required to run the software. More information about the contents of each of the software package files can be found in Appendix D.

3 Approach and Team Strategy

As already mentioned, the team was required to implement four of the eight required modules for the software and any other modules which they see fit to add to meet the specifications outlined by the client. The team developing these modules consisted of three members: Jonty Page, Vyas Raina and James Crossley. The team was split into two main groups - Jonty Page was leading the development on the front-end of the software (*gui.py*) and Vyas Raina and James Crossley were developing the backend modules (*names.py, scanner.py, parse.py*). Figure 2 shows a top down view of the software and who was responsible for developing each element of the program.

The development was split into three stages:

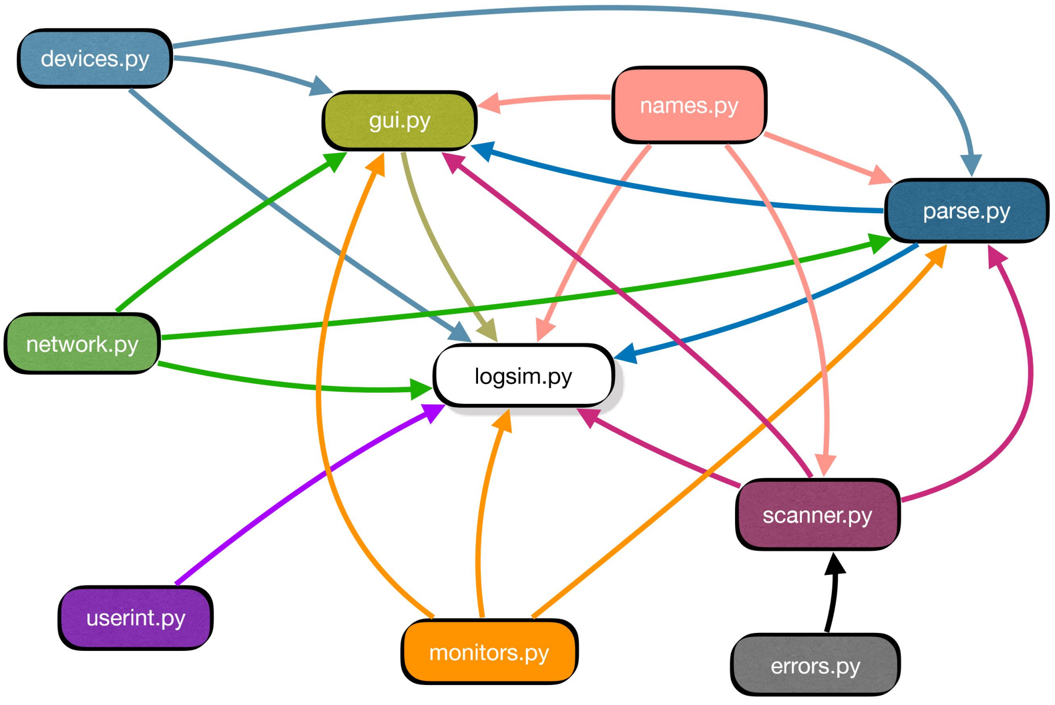

Figure 1: Graph showing the layout of modules in the Logic Simulator.

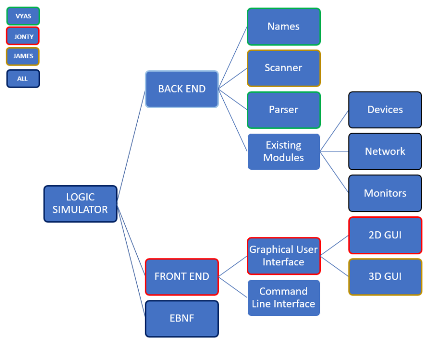

Figure 2: A top down view of the Logic Simulator software and lead developer for each module.

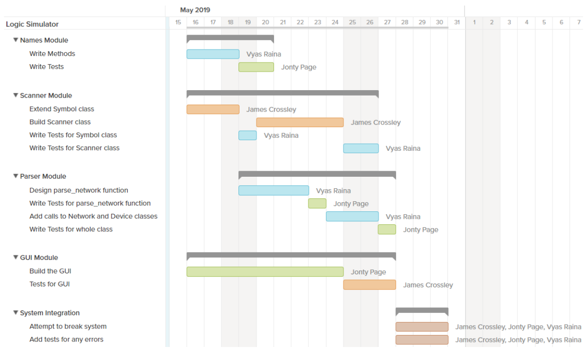

Figure 3: Gantt Chart used by the team for coordination of jobs to be completed by each member.

1. The construction of an EBNF grammar definition file which provided strict rules for the logic description language used by the Logic Simulator.
2. Splitting into the two smaller groups of frontend/backend development and creating the modules required for each.
3. Regrouping as a larger team to integrate the backend modules with the frontend wxPython GUI

Since there was some interdependency between modules in the frontend and backend and there were strict deadlines which had to be adhered to, the team saw it fit to produce a Gantt chart in order to coordinate timings of completion of modules and certain development stages. Figure 3 outlines the Gantt chart created by the team which shows the jobs to be completed by each member of the team and the timeframe for which each job should take. To comply with good software development practice, the team made sure someone other than a module's author to help write the corresponding unit tests.

There were no problems with completing the project before clients deadline of 11.00am Friday 31st May 2019. The only changes to the Gantt chart from the original shown in Figure 3 were that, due to the proportions of work distributed between each team member, it was found that it would be more suitable for James Crossley to write the test file for the Parser module. On the day of the deadline, the client outlined some changes and additions to the specifications document and gave a new deadline for the software to be completed by of 4.00pm Thursday 6th June 2019. The changes to the deadline consisted of three main development tasks and each task was distributed to a single team member in a similar manner to before - more information about the maintenance portion of development can be found in section 4.2.

A private git repository was created to take care of version control and the challenges of collaborative coding. After a developer had completed a piece of working code on their module, the changes were committed to the repository to ensure there are no merge conflicts when working on the same piece of software. The log of git commits can be found in the submitted files .zip folder for each developer. Further to this, all team members agreed to conform to pep8 Python coding standards to ensure the code produced is clean and consistent making it easier to read and update. Code was also agreed to be well documented conforming to PEP257. All code was also to be properly documented with correct docstrings and comments throughout each module.

4 Software Development

4.1 The GUI

The GUI was required to be developed to have at least the same functionality as the command-line interface presented in an intuitive and interactive fashion to the user. This functionality included:

- Running the simulation for N number of cycles.
- Continuing the current simulation for N number of cycles.
- Setting the state of a switch in the circuit at any point in the simulation.
- Setting a monitor point for the circuit whose output is to be displayed.
- Removing a monitor point from the circuit.
- Quitting the simulator.

The GUI was implemented using the wxPython toolkit which is a free and publicaly available set of Python classes which can be used to make sophisticated GUIs in few lines of code. The graphics drawing area in the GUI makes use of the PyOpenGL graphics package which is a portable and easy to use graphics module which is often accelerated by dedicated graphics hardware and supports both 2D and full 3D rendering. There are two classes in the *gui.py* module, the MyGLCanvas class and the Gui class. The MyGLCanvas object is something which is created within the GUI class to make the drawing area in the user interface and handle the displaying of signals returned from the monitors/network classes. The Gui object is the base object which is created and shown in the *logsim.py* top file which draws the GUI onscreen and contains various event handlers such that the GUI can be interacted with by the user.

4.1.1 Layout

The GUIs frame is split into four different panels - The file control panel, the main panel, the monitor points panel and the switches panel. The panels are outlined clearly in Figure 4. The GUI also consists of a number of widgets including buttons, toggle buttons, file picker controls, spin controls, choice boxes and static text. The location of each panel and their constituent widgets are defined using wx.BoxSizer objects and the frame has been made robust to window resizing and maximisation/minimisation using the wx.EXPAND parameter within these Sizer objects and setting a minimum client size for the window.

4.1.2 File Selector Control

The file selector control allows the user to select a circuit definition file to be loaded into the Logic Simulator software. In the event of a file being chosen which contains a syntactic/semantic error, an error message giving the details of the error is displayed to the user in the form of a wx.MessageDialog. Following this, the file is not loaded into the software but the background colour of the top panel is changed to red indicating to the user that the file chosen cannot be loaded by the software. There are two methods in the GUI class related to the file selector control and these are *checkFile, loadNetwork, clearNetwork* and *displaySyntaxErrors*. *checkFile* is the event handler for the event that the user changes the file selection of the wx.FilePickerCtrl widget. The general structure for this event handler is:

```python
def checkFile(self, event):
        # ... creates names, network, monitors, devices, names and scanner objects
        # Check network definition file is correctly configured
        if self.parser.parse_network():
            # Clear old network, load new and reset file picker colour
            self.top_panel.SetBackgroundColour(wx.NullColour)
            self.clearNetwork()
            self.loadNetwork()
        else:
            # Clear old network, display error message and change colour
            self.clearNetwork()
            self.displaySyntaxErrors()
            self.top_panel.SetBackgroundColour(wx.Colour(255, 130, 130))
```

If the file is correctly parsed without error then the old network is cleared from the GUI using the *clearNetwork* method and the new one loaded into the GUI using the *loadNetwork* method. The general structure of each very similar and follows the same form of loading/clearing the monitor points, the switches and canvas. The general structure for clearNetwork is:

```
def clearNetwork(self):
      # Clear monitored points from GUI
      for i in range(len(self.all_mp_names)-1, -1, -1):
          name = self.all_mp_names[i]
          self.mp_sizer.Hide(i)
          self.mp_sizer.Remove(i)
          self.number_of_mps -= 1
          self.Layout()
          self.all_mp_names.remove(name)

      self.mp_names.Clear()
      self.mp_names.Append(_('SELECT'))
      self.mp_names.SetSelection(0)

      # Clear switches from GUI
      if self.loaded_switches is True:
          self.side_sizer.Hide(3)
          self.side_sizer.Remove(3)
          self.Layout()
          self.loaded_switches = False

      # Clears canvas of previous signals
      self.canvas.clear()
```

Finally, if the file is not successfully parsed, then the Errors object is loaded with the syntax/semantic errors to be displayed to the user and the method *displaySyntaxErrors* is called. This method handles the creation of a message dialog displaying the errors. Since wxPython's default font is not one in which each character is of the same width (unlike the terminal in linux/windows/macOS) the caret placement was found to be quite tricky to get correct when highlighting the location of errors. In the terminal, the correct location is found just by having the correct number of spaces equal to the number of characters before the caret should be shown. However, in the GUI, the caret, the characters before which the caret is shown must first be measured in pixels using the GUI's font, the number of spaces then calculated from this number of pixels and then the caret displayed after this many spaces. The code achieving this method is:

```
# ... displaySyntaxErrors(self)
font = error_message.GetFont()
dc = wx.ScreenDC()
dc.SetFont(font)
caret_string = list(error.caret_pos)
sub_string = error.line[:len(caret_string)-1]
w, h = dc.GetTextExtent(sub_string)
space_w, space_h = dc.GetTextExtent(' ')
error_string += ' '*round(w/space_w) + '^'
```

Figure 5 shows an example file dialog and the subsequent events which occur if the user selects an invalid fie.

4.1.3 Monitor Point Controls

The monitor point control panel consists of two different sections - the control bar which allows the user to add a new control point from a wx.Choice selection and a click of a wx.Button, and the ScrolledPanel which displays the currently monitored points in the network alongside a remove button allowing the user to remove a control point. Figure 6 shows how the monitor point control area appears graphically. At first, the monitor points from the file are loaded into the monitor point control area through *loadNetwork* and can be removed through *clearNetwork* however all interaction performed by the user is performed through two event handlers - *onAddMP* and *onRemoveMP*. When the network is loaded, all possible points in the network which are not currently being monitored are loaded as choices in the wx.Choice widget. When the user selects a new monitoring point and clicks add, the event handler *onAddMP* is triggered:

```
def onAddMP(self, event):
      # Finds selected monitor points and adds to monitor object
      index = self.mp_names.GetSelection()
      mp_name = self.mp_names.GetString(index)

      # ... Performs some checks on the selection

      self.monitors.make_monitor(device, port, self.cycles_completed)

      # Removes monitor point from drop-down list
```

```
        reset_index = self.mp_names.FindString(_('SELECT'))
        self.mp_names.SetSelection(reset_index)

        # Adds monitor point and remove button to GUI
        self.number_of_mps += 1
        self.all_mp_names.append(mp_name)
        new_button = wx.Button(
            self.mp_panel, label=_('Remove'), name=mp_name)
        new_sizer = wx.BoxSizer(wx.HORIZONTAL)
        new_sizer.Add(wx.StaticText(self.mp_panel, wx.ID_ANY, mp_name),
                      1, wx.ALIGN_CENTRE)
        new_sizer.Add(new_button, 1, wx.LEFT | wx.RIGHT | wx.TOP, 5)
        new_button.Bind(wx.EVT_BUTTON, self.onRemoveMP)
        self.mp_sizer.Add(new_sizer, 0, wx.RIGHT, 5)
        self.Layout()
```

When the user clicks the remove button next to the monitoring point name in the list of currently monitored points, the event handler *onRemoveMP* performs a similar set of actions in reverse to remove the monitoring points: Remove the wx.StaticText and wx.Button widgets from the sizer, add the option to the wx.Choice widget and remove the monitoring points from the monitors object.

4.1.4 Switch Value Controls

The user can change the values of each switch in the circuit at will at any point during a simulation using the switch control panel. Each switch has a wx.StaticText label and a wx.ToggleButton. The switches are loaded into this panel during the *loadNetwork* call. There is one event handler associated with the switch control panel - *onToggleButton*. This handles the event that a user clicks on the toggle button next to a switch name to change the switch value.

```
def onToggleButton(self, event):
      button = event.GetEventObject()
      switch_id = self.names.query(button.GetName())
      if button.GetValue():
          # Switch is off, so turn button on and green
          self.devices.set_switch(switch_id, 1)
          button.SetBackgroundColour(wx.Colour(100, 255, 100))
          button.SetLabel(_('On'))
          text = _("%s turned on.") % button.GetName()
          self.canvas.render(text)
      else:
          # ... Same as above but turn off
```

The wx.ToggleButton is coloured to give a visual indication to the current state of the switch as well as the text label. When the user clicks the toggle button, the switch state is also changed in the devices object such that it has an actual impact on the results of the simulated circuit. Both this panel and the monitor point control panel are ScrolledPanels, such that if the number of monitor points/switches are so great that the buttons would disappear of the screen, a scroll bar appears (as shown in Figures of the GUI) which allows the user to scroll across all of the buttons contained in the panel. This means that there is no limit placed on the number of monitor points/switches in the circuit except that defined by the memory capacity of the computer.

4.1.5 Control Buttons

The control buttons consist of three buttons - ”Run”, ”Continue” and ”Exit” buttons, each with a different event handler *on_run_button*, *on_continue_button* and *on_exit_button* respectively. There is also a wx.SpinCtrl widget which allows the user to select the number of clock cycles for which the circuit should be executed upon clicking of the ”Run”/”Continue” buttons. This has an associated event handler *on_spin* which displays a notification of the change of spin value. The general structure of the event handler for the both the ”Run” and ”Continue” buttons is largely the same with the ”Run” button resetting the locally global variable storing the number of cycles completed in the current simulation to zero.

```
  def on_run_button(self, event):
        # Reset canvas and render notification
        self.canvas.reset()
        text = _("Run button pressed.")
        self.canvas.render(text)

        # Reset number of cycles and run network as desired
        if self.loaded_network:
            self.cycles_completed = 0
            cycles = self.spin.GetValue()

            if cycles is not None:
                self.monitors.reset_monitors()
                self.devices.cold_startup()
                if self.run_network(cycles):
                    self.cycles_completed += cycles
```

Both event handlers call a function *run_network(cycles)* which executes the network for the given number of cycles, retrieves the output signals and renders them to the drawing area.

```
  def run_network(self, cycles):
        for i in range(cycles):
            if self.network.execute_network():
                self.monitors.record_signals()
                self.canvas.render(_('Drawing signal'), self.monitors)
            else:
                text = _("Error! Network oscillating.")
                print(text)
                self.displayError(text)
                return False
        return True
```

In order to get the signals in the formal required by the *canvas.render* method, the *monitors.py* module was edited slightly from the module developed prior to the team taking over this software development task. The method *monitors.record_signals* was added to record the traces at the monitored points in the circuit and store them as an attribute of the monitors object. The monitors object is then passed (hence passing the stored signals) to the *canvas.render* method which handles all the drawing operations. The "Exit" button's event handler simply closes the window and exits the application.

4.1.6 OpenGL Canvas

The MyGLCanvas class handles all canvas drawing operations and makes use of the PyOpenGL package. It consists of seven methods which allow the drawing of objects and traces on the canvas, as well as rending text, clearing the canvas and resetting the viewport. The *render* method iterates through the output signals, and draws each signal in a different colour as lines on the 2D canvas. It also renders a time axis showing the number of clock cycles which have passed and renders any text to be shown as notification of actions being performed by the user. The *render* method is called every time paint operation is performed on the canvas. For this reason, the current signals are stored as locally global variables which are only edited when a monitors object is passed to the *render* method. The drawing operations are then performed as shown in the code below:

```
def render(self, text, monitors=None):
      # ... Load signals if monitors object given and render text

      # Draw signals
      if self.current_signal != []:
          # Draw each signal
          for j in range(len(self.current_signal)):
              GL.glColor3f(self.signal_colours[j][0], (
                  self.signal_colours[j][1]), self.signal_colours[j][2])
              GL.glBegin(GL.GL_LINE_STRIP)
              for i in range(len(self.current_signal[j])):
                  if self.current_signal[j][i] != 4:
                      x = (i * 20) + 40
                      x_next = (i * 20) + 60
                      if self.current_signal[j][i] == 0:
                          y = 75*(j+1)+30
                      else:
                          y = 75*(j+1)+55
                      GL.glVertex2f(x, y)
                      GL.glVertex2f(x_next, y)
              GL.glEnd()
              self.render_text('0', 10, 75*(j+1)+30)
              self.render_text('1', 10, 75*(j+1)+55)
              self.render_text(self.current_monitor_points[j], 10, (
                  75*(j+1)+10))
              self.render_text(self.current_monitor_points[j], 10, (
                  75*(j+1)+10))

          # ... Draw and label time axis in similar fashion

      # ... Flushing pipeline and swapping buffers etc.
```

The user is able to pan through clicking and dragging on the canvas and zoom using the scroll wheel on their mouse. These mouse events are handled in the method *on_mouse*. Finally the *reset* and *clear* methods are used by the packages to clear the canvas of signals and reset the viewport to the original location.

4.1.7 logsim.py

logsim.py is the top level python file which is run to start the logic simulator program. This file had already been developed prior to the team picking up the project, however some edits were made to this file to allow the user to not specify a file path to a circuit definition file from the command line or rather in the terminal. If the no file path is specified then the GUI will open without a file loaded and the user can select the file using the file picker control. If a file path is specified then the file will be pre-loaded when the GUI opens. This is handled in the code by:

```
def main(arg_list):
  # ... Run command line interface if -c flag is given
  if not options:  # no option given, use the graphical user interface

        if len(arguments) > 1:  # wrong number of arguments
            print("Error: one file path required\n")
            print(usage_message)
            sys.exit()

        if len(arguments) == 1:
            [path] = arguments
            scanner = Scanner(path, names)
            parser = Parser(names, devices, network, monitors, scanner)
            if parser.parse_network():
                app = wx.App()
                gui = Gui("Logic Simulator", names, devices, network,
                          monitors, os.path.abspath(path), scanner, parser)
                gui.Show(True)
                app.MainLoop()

        if len(arguments) == 0:
            app = wx.App()
            gui = Gui("Logic Simulator", names, devices, network, monitors)
            gui.Show(True)
            app.MainLoop()
```

4.2 Maintenance

4.2.1 Overview

At 11.00am on Friday 31st of May, the client introduced three additions and changes to the specifications document. These three changes included:

1. A new device called a SIGGEN signal generator was to be added to the list of supported devices.
2. The software should be internationalized to support English and at least one other language which contains some non-Latin characters.
3. The user should have the option to draw the signals in 3D and be able to switch between 2D and 3D drawing at will.

The team decided that the best way to approach these modifications was to assign a team member to each of the three changes and since most of the maintenance was to be done on the frontend of the software, Jonty Page would oversee the integration of all parts of the changes to ensure a coherent solution and smooth integration with the original GUI. Vyas Raina was assigned to adding the new device, James Crossley to constructing the 3D traces and Jonty Page to the internationalization.

4.2.2 Internationalization

Internationalization allows the software to detect the locale of the machine on which the software is being run and if possible, translate all strings into the language specified by these parameters. This was achieves through creating a ".pot" file for the GUI using the python module *pygettext.py*. This ".pot" file contains a list of all of the possible string which might be shown to the user such that they can be translated using a program such as *Poedit* to create ".po" and ".mo" files for the software's supported languages. In order to do this, each string in the *gui.py* must first be wrapped with a call to the function "_()" which indicated to the *pygettext.py* module to pick up this string and add it to the list. Poedit is a piece of software which allows the generation of ".po" and ".mo" files manually from a base ".pot" file. The languages of French and Hindi were chosen by the team to be supported by the software since members of the team were familiar with these languages and they provided an example of non-Latin character support. The translations were generated manually and ".po" and ".mo" files saved for each supported language under the file path "locale/(language)/LC_MESSAGE/(logicsimapp.po/.mo)". In order to configure the software to pick up these translations, the wxPython internationalization support was implemented in the *logsim.py* file just after the wx.App as created. The code added was:

```
# ... wx.App created
# Internationalisation
builtins._ = wx.GetTranslation
locale = wx.Locale()
locale.Init(wx.LANGUAGE_DEFAULT)
locale.AddCatalogLookupPathPrefix('./locale')
locale.AddCatalog('logicsimapp')
# ... GUI created and shown
```

This code configures the wx.Locale object to look for translations under the file path at which the ".po" files are saved. The object will automatically detect the language environment variable set either when running the program from the terminal (through "LANG=fr_FR.utf8"), or in the computer's settings.

4.2.3 Feedback after Initial Testing Round

On Friday 31st May there was an initial testing round performed by the client. This was the first time which the client had seen the software and although the client was happy with the out-of-the-box style solution which felt robust and simple, one bug was noted and a suggestion to improve the program was made. The bug noted by the client was that the notification bar on the OpenGL canvas was still displaying the debug messages left by the previous developers of the software as a means to check that the canvas was operating correctly in development. This was an easy bug fix and the code which generated this text was quickly removed after the feedback session. The suggestion for an improvement was that, although it was possible for the user to reset the position of the viewport to the original, a nice feature would be to prevent the user from panning such that the signal traces are not visible in the first place. This was achieved by coming up with a set of rules which set limits on the number of pixels over which the user could pan. Two new locally global variables were introduced measuring the dimensions of the currently stored signals in pixels and these were then manipulated in the *on_mouse* method of the MyGLCanvas class limit the panning variables. The limits were successfully added and the code to do so is shown below:

```
if event.Dragging():
        size = self.GetClientSize()
        intended_move_x = (self.pan_x + event.GetX() - self.last_mouse_x)
        intended_move_y = (self.pan_y - event.GetY() + self.last_mouse_y)
        if intended_move_x < 0 and intended_move_x > min(
                0, size.width-self.max_x*self.zoom):
            self.pan_x += event.GetX() - self.last_mouse_x
            self.last_mouse_x = event.GetX()
        if intended_move_y < 0 and intended_move_y > min(
                0, size.height-self.max_y*self.zoom):
            self.pan_y -= event.GetY() - self.last_mouse_y
            self.last_mouse_y = event.GetY()
        if intended_move_x > 0:
            self.pan_x = 0
        if intended_move_y > 0:
            self.pan_y = 0
        if intended_move_x < min(0, size.width-self.max_x*self.zoom):
            self.pan_x = min(0, size.width-self.max_x*self.zoom)
        if intended_move_y < min(0, size.height-self.max_y*self.zoom):
            self.pan_y = min(0, size.height-self.max_y*self.zoom)
```

5 Testing Procedures

5.1 Pytests

Unit testing is an extremely important practice in software development especially when it comes to integrating many modules together. The pytest framework was used to perform unit tests on each of the backend modules in the software. This package was chosen for unit testing as it allows the use of parameterization, using decorator functions, which allow many tests to be run together at once. The team felt it important to have someone other that a modules author write the pytests for each module to ensure a high quality of solution was produced with no bias in the testing phase. There are in total 125 pytests for all of the backend modules testing the functionality and robustness of each method of each module to user input and runtime errors. One example of a pytest created is for the *names.py* module to test that the function *unique_error_codes* raises the correct exceptions:

```
@pytest.fixture
def new_names():
    """Return a new names instance."""
    return Names()
def test_unique_error_codes_raises_exceptions(new_names):
    """Test that unique_error_codes raises expected exceptions"""
    with pytest.raises(TypeError):
        new_names.unique_error_codes('Hello')
    with pytest.raises(TypeError):
        new_names.unique_error_codes(1.5)
    with pytest.raises(ValueError):
        new_names.unique_error_codes(-1)
```

All other unit tests follow a similar pattern and all tests can be performing using the comment ”pytest -v” from the terminal in the main software directory. The approach of White Box testing was used to write the pytest files since, although the author of the pytest files was deliberately different to that of the module, the pytests were written with the structure of the underlying code in mind such that every possible error was raised and correctly identified.

5.2 Testing the GUI

Since formal unit testing of the GUI is not feasible, to test robustness, the team came up with a number of black box tests in which it is tested that a specific input gives to correct expected output. These tests are detailed in the Table 2.

Test	Expected result
Valid File is loaded	GUI loads the file and displays the stated monitoring points traces (default in 3D with option to change to 2D).
Invalid File loaded	Dialogue Box appears reporting the specific error.
Toggling between 3D and 2D views	Signal traces flips between the two views.
Monitoring points added and removed	Monitoring points and signal traces update appropriately. If many monitoring points are added a scrollbar appears.
User enters a large number of cycles	Range is limited to 0-100 and the control buttons RUN and CONTINUE display the cycles for the selected number of cycles.
User selects the EXIT button	The program closes without any errors in the terminal.
User sets the language to Hindi or French	The GUI loads with the correct translations of all buttons
User attempts panning of the signal traces into white space	GUI prevents panning into complete white space.

Table 2: Example Black Box tests for GUI.

6 Conclusions and Improvements

Overall, the team successfully developed a robust, clean and simple solution which achieved all of the requirements set out by the client in the specifications document. Good software engineering practices were adhered to (such as pep8 pycodestyle, PEP257 conforming documentaton, unit testing) and all deadlines were met. If there was an opportunity for further development on this project some new features which could be implemented may include:

- A canvas which renders the circuit diagram for the circuit being simulated.
- The ability for the user to make amendments to the circuit from the GUI which in turn will amend the circuit definition file accordingly.

- Expanding the list of supported devices even further to support things such as an RC-time delay unit and other types of flip-flops.
- The ability to save and store simulation outputs which could then be loaded and continued at a later date.

Appendix A Example Circuit Definition Files, Diagrams and Results

Appendix B EBNF Description of the Logic Description Language

Appendix C Software User Guide

Appendix D Brief Description of File Contents

References

[1] Andrew Gee and Mojisola Agboola. *CUED GF2 Software: Main Handout.* 2018. URL: `https://www.vle.cam.ac.uk/pluginfile.php/13279532/mod_resource/content/18/handout_19.pdf`.