ENGINEERING TRIPOS PART II A

GF2 **SOFTWARE P2**

FINAL REPORT - LOGIC SIMULATOR GRAPHIC USER INTERFACE DEVELOPMENT AND MAINTENANCE

Team 3 - Sherry Fu - jf731

Colleges: Sidney Sussex College

Report due: 6th June 2023 at 4pm

1 Introduction

This report details the development of the GUI module for a logic simulator as part of the Engineering Tripos Part IIA DPO Project GF2. The project involved creating a Python-based application to simulate logic circuits, providing both text-based and graphical user interfaces. My primary contribution was developing the GUI module, ensuring an intuitive and functional user interface that complements the text-based command functionality.

2 Functionality and Software Structure

The logic simulator is a comprehensive software tool designed to simulate the behaviour of digital logic circuits. The system operates in two primary phases: definition and simulation. During the definition phase, the simulator reads a text file that specifies the logic components and their interconnections. In the simulation phase, users can run the circuit, manipulate inputs, and observe outputs through both text-based and graphical user interfaces.

2.1 Software Structure

The software is modular, with each module handling specific aspects of the simulation process. This modular approach facilitates ease of maintenance, testing, and potential future enhancements. The primary modules are:

2.1.1 Names Module (names.py)

- **Purpose**: Manages the mapping between names (strings) and unique integer identifiers. This class deals with storing grammatical keywords and user-defined words, and their corresponding name IDs, which are internal indexing integers. It provides functions for looking up either the name ID or the name string. It also keeps track of the number of error codes defined by other classes, and allocates new, unique error codes on demand.
- **Functions**:
  - `lookup`: Adds a new name to the names table or retrieves the ID of an existing name.
  - `get_name_string`: Retrieves the name string corresponding to a given ID.
  - `unique_error_codes`: Generates unique error codes for various components.

2.1.2 Scanner Module (scanner.py)

- **Purpose**: Translates the definition file into a sequence of symbols for parsing. Once supplied with the path to a valid definition file, the scanner translates the sequence of characters in the definition file into symbols that the parser can use. It also skips over comments and irrelevant formatting characters, such as spaces and line breaks.
- **Functions**:
  - `get_symbol`: Returns the next symbol in the definition file.
  - `skip_spaces`: Skips whitespace in the input.
  - `get_name`, `get_number`: Extracts names and numbers from the input.

2.1.3 Parser Module (parse.py)

- **Purpose**: Processes a stream of symbols from scanning a definition source file and parses instructions using the EBNF grammar. It calls methods from the Devices, Network, and Monitors singletons to build the circuit network. Syntax and semantic errors are checked and caught here. In the event of an error, an instructive error message is generated, and parsing moves to the next convenient symbol to continue (usually a semicolon, depending on the rule being processed).
- **Class**:
  - `Parser`: Main class for parsing.
- **Public Methods**:
  - `__init__(names, devices, network, monitors, scanner)`: Instantiates the parser, passing the necessary singletons.
  - `parse_network`: Analyzes symbols from the provided file, handles errors, and applies grammar rules. Returns True if there are no errors, and False if errors occur, displaying an error count message on the console.
- **Private Methods/Implementation**:
  - Each EBNF grammar rule is defined with a corresponding function. Using an LL(1) grammar, the current rule can be inferred from the current symbol without lookahead, ensuring no ambiguity in rule method calls. Each rule method detects syntax and semantic errors and handles errors from external method calls during network construction.
  - `handle_error`: Selects the appropriate error message based on the error code, passes the message and symbol in question to `self.scanner` to display the error symbol in the context of the source file, and optionally skips to a specified stopping symbol before continuing parsing.

2.1.4 Devices Module (devices.py)

- **Purpose**: Defines the behavior and characteristics of logic devices.
- **Classes**:
  - `Device`: Represents a single device with its inputs, outputs, and state.
  - `Devices`: Manages a collection of Device objects, handles creation, and configuration.

2.1.5 Network Module (network.py)

- **Purpose**: Manages connections between devices and the execution of the logic circuit.
- **Functions**:
  - `make_connection`: Establishes a connection between device outputs and inputs.
  - `execute_network`: Executes the logic network for one simulation cycle.
  - `update_signal`: Updates the state of a signal, handling transitions and edge detection.

2.1.6 Monitors Module (monitors.py)

- **Purpose**: Records and displays signal levels at designated monitor points.
- **Functions**:
  - `make_monitor`, `remove_monitor`: Adds or removes monitor points.
  - `record_signals`: Records the signal levels for each simulation cycle.
  - `display_signals`: Displays the recorded signals in a text-based format.

2.1.7 User Interface Module (userint.py)

- **Purpose**: Handles the text-based user interface for interacting with the simulator.
- **Functions**:
  - `command_interface`: Reads and executes user commands.
  - Command functions (`run_command`, `continue_command`, etc.): Implement specific user commands for running the simulation, setting switches, adding monitors, etc.

2.1.8 Graphical User Interface Module (gui.py)

- **Purpose**: Provides a graphical user interface for the simulator using wxPython and OpenGL. The user can use selection and press buttons to control the logic simulator instead of typing in commands using the text-based user interface.
- **Classes and Functions**:
  - `Gui`: Main GUI window class, handles layout and user interactions. The side-panel button and toolbar handler are designed inside this class.
  - `MyGLCanvas`: OpenGL canvas for rendering signal waveforms.
  - Event handlers (`on_run_button`, `on_set_switch_button`, etc.): Handle various user actions and update the GUI accordingly.
  - `render`: Draws the signal waveforms on the OpenGL canvas.

2.2 Integration and Interaction

- **Main Program** (logsim.py): Initializes the simulator, sets up the GUI or command-line interface, and handles the main event loop.
- **Flow of Execution**:
  1. *Initialization*: The main program initializes the necessary modules and sets up the interface based on user preferences (text-based or GUI).
  2. *Definition Phase*: The parser reads the definition file, using the scanner to translate the input into symbols and build the logic network with the help of the devices and network modules.
  3. *Simulation Phase*: Users interact with the simulator through the user interface. Commands are processed to run the simulation, manipulate inputs, and monitor outputs.
  4. *Output Display*: The monitor module records signal changes, and the GUI or text-based interface displays the results.

3 Teamwork and Contribution

3.1 Detailed Workflow

3.1.1 Week 1: Preliminary Tasks and Initial Development

In the first week, I focused on completing the preliminary tasks and developing the `names.py` module, alongside its associated pytest file. The core functionalities from the preliminary exercises were integrated into `names.py`, with additional refinements. Specifically, the functions for returning a specific name ID and a list of name IDs were separated into two distinct functions: `query` and `lookup`.

Error handling was enhanced by incorporating checks for `TypeError` for non-string inputs and `SyntaxError` for invalid names in query, as well as `TypeError` for non-list inputs in lookup, and `ValueError` and `IndexError` in `get_name_string`. Comprehensive testing was implemented using `pytest.fixture` and
`pytest.mark.parametrize` to streamline the test code and cover all potential errors.

For our group work, I generated the electrical diagrams for all devices as per the client's specifications. This visual representation was crucial for our understanding and planning. Meanwhile, James undertook the responsibility of writing the definition file, leveraging his expertise in EBNF grammar generation.

3.1.2 Week 2: First Interim Report and GUI Design

The second week involved submitting our first interim report and incorporating feedback regarding modifications to the EBNF definition file. Significant changes were made to the grammar to align with the client's requirements. Teamwork allocation remained consistent as following:

- Alex: `scanner.py` and its pytest
- James: `parser.py` and its pytest
- Myself: GUI development

The design phase for the GUI commenced with a review of tutorial demos available on Moodle, followed by sketching the graphical layout on an iPad. The GUI was conceptualized to include a side panel for control functions (running simulations, setting switch states, managing monitors) and a canvas dedicated to displaying signal waveforms. Additionally, the design incorporated a toolbar at the top, replacing the menu in the demo, to facilitate functions such as opening files, accessing the help page, and exiting the application. This design plan was discussed with the team and received unanimous approval, leading to the commencement of implementation in the third week.

3.1.3 Week 3: GUI Implementation and Integration

The initial focus was on identifying the signals required from other Python modules. For the canvas, the essential elements included `trace_names` (monitored signals) and their corresponding values (0/1 for waveform drawing). For the `Gui` class, the necessary data encompassed `switch_ids`, `switch_names`, `switch_values` (0/1 for OFF/ON), `sig_not_mons`, and `sig_mons` (signals not monitored and signals under monitor). Dummy data lists were utilized initially to allow for independent testing of the GUI prior to the completion of the `scanner` and `parser` modules.

The structure of the `Gui` class was meticulously organized, with the initialization method (`__init__`)

setting up all parameters, including toolbar button IDs. The toolbar setup bundle initialized all icon images required for the toolbar, and the sidebar panel's buttons were also initialized in this function. Widget setup followed, with a design choice to add two buttons to each side sizer for horizontal alignment. This method was applied to the run and continue buttons, the switch drop-down and checkbox, the add monitor choice list and button, and the remove monitor choice list and button. Subsequent to widget initialization, event bindings were established.

Event functions were designed with a focus on functionality and user experience. The `toolbarhandler` was the initial focus, where the "Open File" button not only initialized file analysis methods but also displayed the file contents in a new window. The `wx.FileDialog` was configured to restrict viewing to `.txt` files, preventing system crashes from incompatible file types. The "About" button utilized `wx.MessageBox` to display group credits, while the "Help" button generated a user guide window through a custom `HelpWindow` class with specified dimensions for proper display. The "Quit" button closed the application as expected.

3.1.4 GUI Event Handling and Debugging

To ensure the functionality of event handlers, a debugging text line was used on the canvas. For instance, the canvas rendered text "Switch is now ON/OFF" when switch checkbox is ticked to verify correct operation. The function
`run_network_and_get_values` was critical, calling devices and monitors to record signals and reconstruct waveforms using `monitors_dictionary`. This function was integrated into both the run and continue buttons, leading to issues that will be discussed in the <Test Procedure and Debugging> section.

Canvas design involved the development of `render_graph_axes` and `render_trace` functions, which were invoked within the `render` function to display traces if `trace_names` and values were available. In the absence of these lists, the initial state displayed the message "Click Run button to START," ensuring clarity for users.

3.1.5 Additional Details and Implementation

Further details of the GUI implementation included the design of the sidebar panel and toolbar. The sidebar panel was equipped with controls for running the simulation for a specified number of cycles, continuing the simulation, setting switch states, and managing monitors. Each control was meticulously designed for user-friendly interaction.

The toolbar included functionalities such as opening definition files, accessing help, viewing credits, and exiting the application. Each toolbar button was designed with appropriate icons and tooltips to enhance usability. Event handling for toolbar buttons was implemented to ensure responsive and intuitive user interactions.

The integration of GUI components with the underlying logic simulator was carefully planned. Dummy data allowed for initial testing, while the final integration ensured seamless communication between the GUI and other modules. This approach facilitated iterative development and debugging, ensuring a robust and functional GUI.

3.2 Week 4: Maintenance

As the project entered the fourth week, new maintenance requirements were introduced. James joined the GUI development, focusing on expanding the functionality to include 3D visualizations. The team's responsibilities were reallocated as follows:

- Alex: Integrating the RC-circuit module.
- James: Developing the 3D GUI components.
- Myself: Implementing localization, including Chinese translations.

3.2.1 3D GUI Development

To support the new 3D visualization requirements, a toggle button was added to the toolbar, allowing users to switch between 2D and 3D canvases. This involved introducing a `screen_type` variable to manage the canvas state and a `reset_screen` function to ensure the canvas size resets to its default state upon toggling. I did these preparation jobs.

The provided `gui_3D` code was then merged into the original GUI file to assess compatibility. James focused on programming the signal display within the 3D canvas. The initial challenge was adjusting the viewing direction, as the default setup placed signals along the z-axis, whereas the requirement was to draw signals along the x-axis. By thoroughly understanding the parameters of the `draw_cuboid` function, James successfully reoriented the signals. Additionally, he implemented a 3D axis with labels, mirroring the 2D canvas setup, and ensured signal names floated above the signal traces, enhancing readability and user interaction.

3.2.2 Localization Implementation

Localization efforts began by adding functionality to detect the system's language settings using the `os.getenv` function. When the system language is set to Simplified Chinese, or when the command `LANG=zh_CN.utf8` is entered before initialization, the GUI would adapt to display translated content. The translation process involved storing translated strings in a `.po` file within a locale directory, where `msgid` represents the original string IDs, and `msgstr` contains the translated strings.

Manual translation of all strings was performed and incorporated into the `.po` file. The following command was used to compile the `.po` file into a `.mo` file, readable by the application:

```
msgfmt locale/zh_CN/gui.po -o locale/zh_CN/gui.mo
```

All text requiring translation was formatted as `_('text')`. This approach ensured that all GUI text elements were successfully translated. Furthermore, a translated version of the help page was created to appear when a Chinese environment is detected, enhancing usability for Chinese-speaking users.

3.3 Collaboration and online tool

Effective collaboration was facilitated through a combination of in-person sessions at the DPO and the use of several online tools:

- **WhatsApp**: Utilized for daily communication and quick updates, ensuring team members stayed informed of each other's progress and any immediate issues.
- **GitHub**: Served as the primary platform for version control and issue tracking. The ability to highlight changes, make comments on pull requests, and review modifications by team members helped maintain code integrity and streamline the debugging process.
- **Google Drive**: Enabled seamless sharing of documents and collaborative editing, which was particularly useful for preparing reports and documentation.

These tools significantly contributed to maintaining a cohesive workflow and promptly addressing challenges. GitHub, in particular, proved invaluable for its clear visualization of changes and its commenting feature, which facilitated peer reviews and bug fixing.

Overall, team cooperation was highly effective, characterized by the absence of redundant work, efficient workflows, and consistent communication. Each member's contributions were valuable, and the collaborative environment fostered a supportive atmosphere conducive to problem-solving and innovation.

4 Test procedure and Debugging

4.1 Test procedure

For the `name.py` file, we conducted testing using pytest, as detailed in the previous section titled <Teamwork and Contribution>. However, for the graphical user interface (GUI) component, no automated pytest framework is applicable. Therefore, our testing approach relied heavily on manually verifying the UI's appearance on a Linux environment. This included running various edge cases to ensure the system's stability and to identify potential crashes.

Our teammate Alex generated a counter case example with no switches inside the logic circuit. Its definition file is stored in `three-counter.txt` (See Appendix A4). To address the issue of system crashes in the `three-counter.txt` example, we implemented a conditional check to determine if the switch list is empty. If the switch list is not empty, the switch widgets appear in the side panel. This adjustment ensures that when running the `three-counter.txt` example, no switch buttons appear if none are defined, thereby preventing crashes.

Additionally, we improved the dropdown list functionality by changing the `wx.ComboBox` attribute to `style=wx.CB_READONLY`. This modification prevents users from typing into the dropdown list, ensuring it behaves as a selection-only list.

Regarding the transition to a text-based interface, we initially considered using a textbox. However, due to the complexity, we decided to remove the textbox. Instead, when the quit button is pressed, the initial instructions for the text-based interface are printed, automatically returning the user to the Terminal window.

4.2 Debugging

One significant issue encountered was the unnecessary replotting each time a switch was set or a monitor was added. The solution was to avoid calling `run_network_and_get_values` inside each function. This change reduced redundant operations and improved performance.

Another issue was the continue button redrawing previous waveforms instead of continuing from the last point. To address this, we adjusted the execution frequency of `execute_network`. James helped identify that the `render_trace` function's index needed adjustment. Initially, the code always started from index 0. By adding the gap between `time_steps` and the length of values to the index, the waveform now correctly continues from the last point.

4.3 Modifications Based on Interim Suggestions

Based on feedback from the second interim review, we implemented several important modifications to improve the functionality and usability of our system. One of the primary tasks was the removal

of debugging text. This was a straightforward process where we eliminated the `render_text` calls that were printing debugging information within the `render_2D` function. This change helped clean up the output and made the interface more professional and user-friendly.

In addition to removing debugging text, we enhanced the `render_axis` function to dynamically adjust the display of number labels based on the zoom level. This adaptive labeling system improves the user experience by ensuring that the interface remains clear and informative at various zoom levels. Specifically, when the zoom level exceeds 0.8, all number labels are displayed. For zoom levels between 0.5 and 0.8, only every fifth number is shown, and for zoom levels below 0.5, every tenth number is displayed. This graduated display of labels helps to prevent clutter and ensures that the user can always read the axis labels comfortably, regardless of the zoom level.

Furthermore, we undertook a comprehensive review and correction of the `gui.py` file to address several coding style violations that had not been checked by `pycodestyle` in the previous version. This included rectifying issues related to over-indentation and trailing white spaces. By adhering to PEP 8 standards, we not only improved the readability and maintainability of the code but also ensured a higher level of professionalism and quality in our software development practices.

5 Conclusion and potential improvements

5.1 Summary

In conclusion, the development and maintenance of the GUI module for our logic simulator have been comprehensive and multi-faceted, reflecting a detailed approach to both functionality and user experience. Our primary goal was to create an intuitive and efficient graphical interface that complements the existing text-based interface, thereby offering users a versatile tool for simulating digital logic circuits.

Throughout the project, we encountered and addressed several significant issues. By implementing conditional checks and improving the handling of user inputs, we ensured the stability of the system, particularly in edge cases like the `three-counter.txt` example. The enhancements made to the dropdown list functionality and the seamless transition back to the text-based interface upon quitting further streamlined the user experience.

Debugging was a critical aspect of our process. We resolved key issues such as unnecessary replotting and waveform redrawing, leading to improved performance and functionality. The integration of adaptive number labels based on zoom levels enhanced the visual clarity and usability of the interface.

5.2 Improvements

Looking forward, several potential improvements could enhance the application's functionality and user experience. Implementing minimum and maximum constraints for zoom levels would provide better user control and prevent excessive zooming. Additionally, displaying the functionality of each toolbar button when the mouse hovers over the icon would make the interface more intuitive.

Enhancements to the 3D canvas could include adding a floating title that remains static while signals rotate and setting minimum and maximum rotation angles to improve control over the 3D view. Furthermore, translating the canvas content based on detected language preferences could significantly enhance accessibility for non-English users. This, however, would require a custom solution beyond the capabilities of the `locale` and `wx` localization package.

Overall, this project has demonstrated the importance of meticulous design, thorough testing, and responsive debugging in developing a user-friendly and reliable graphical user interface. The collaboration within the team has been highly effective, leading to a well-rounded and functional product that meets the project's objectives.

Appendix A: Definition file

A1. SR flip-flop circuit

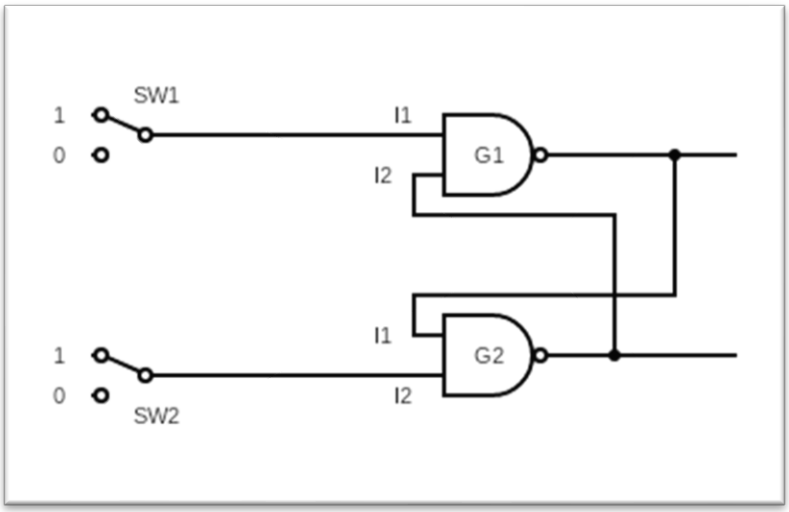


Figure 1: Graph by Sherry Fu, Definition file by James Lecomte

```
###
Example 1: SR flip-flop circuit
###

DEVICES:
SWITCH SW1(0), SW2(0);
NAND G1(2), G2(2);

CONNECTIONS:
SW1 > G1.I1;
G2 > G1.I2;

G1 > G2.I1;
SW2 > G2.I2;

MONITOR G1, G2;
```

A2. Comprehensive circuit

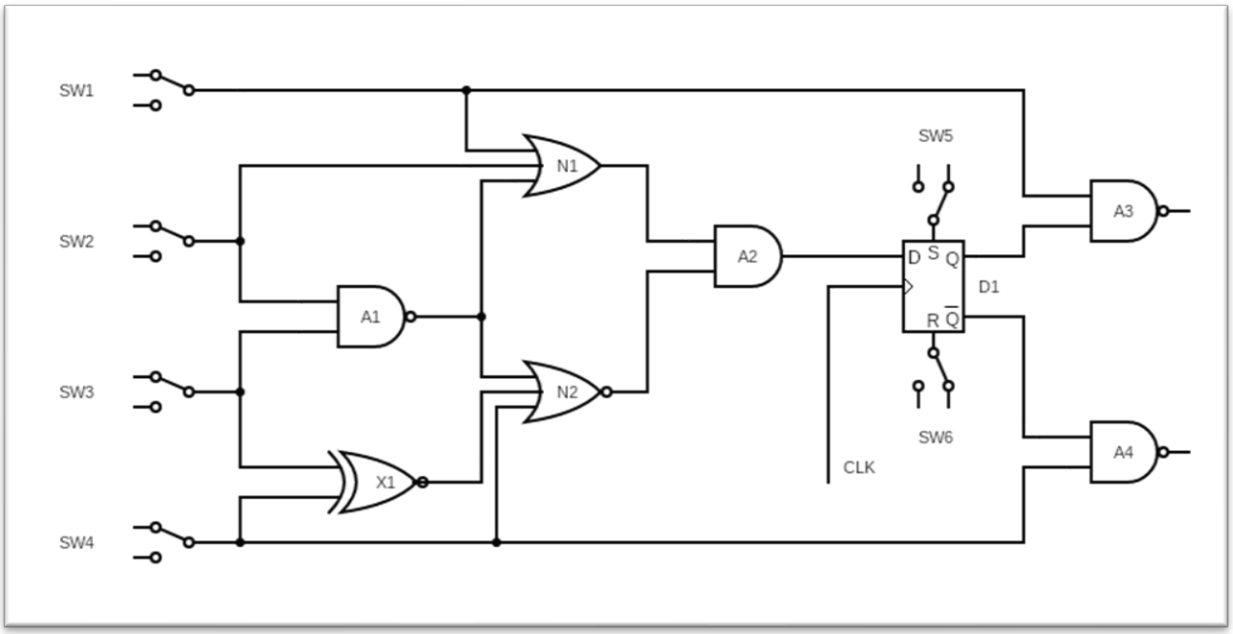


Figure 2: Graph by Sherry Fu, Definition file by James Lecomte

```
###
Example 2: Comprehensive circuit
###

DEVICES:
SWITCH SW1(0), SW2(0), SW3(0),
SW4(0);
NAND A1(2), A3(2), A4(2);
AND A2(2);
XOR X1;
OR N1 (3);
NOR N2 (3);

SWITCH SW5(0), SW6(0);
DTYPE D1;
CLOCK C(2);

CONNECTIONS:
SW2 > A1.I1;
SW3 > A1.I2;
SW3 > X1.I1;
SW4 > X1.I2;
SW1 > N1.I1;
SW2 > N1.I2;
A1 > N1.I3;

A1 > N2.I1;
X1 > N2.I2;
SW4 > N2.I3;

N1 > A2.I1;
N2 > A2.I2;

# D-Type Connections
SW5 > D1.SET;
SW6 > D1.CLEAR;
A2 > D1.DATA;
C > D1.CLK;

# Final NAND Connections
SW1 > A3.I1;
D1.Q > A3.I2;
D1.QBAR > A4.I1;
SW4 > A4.I2;

MONITOR A3, A4;
```

A3. Circuit with 2 D-types

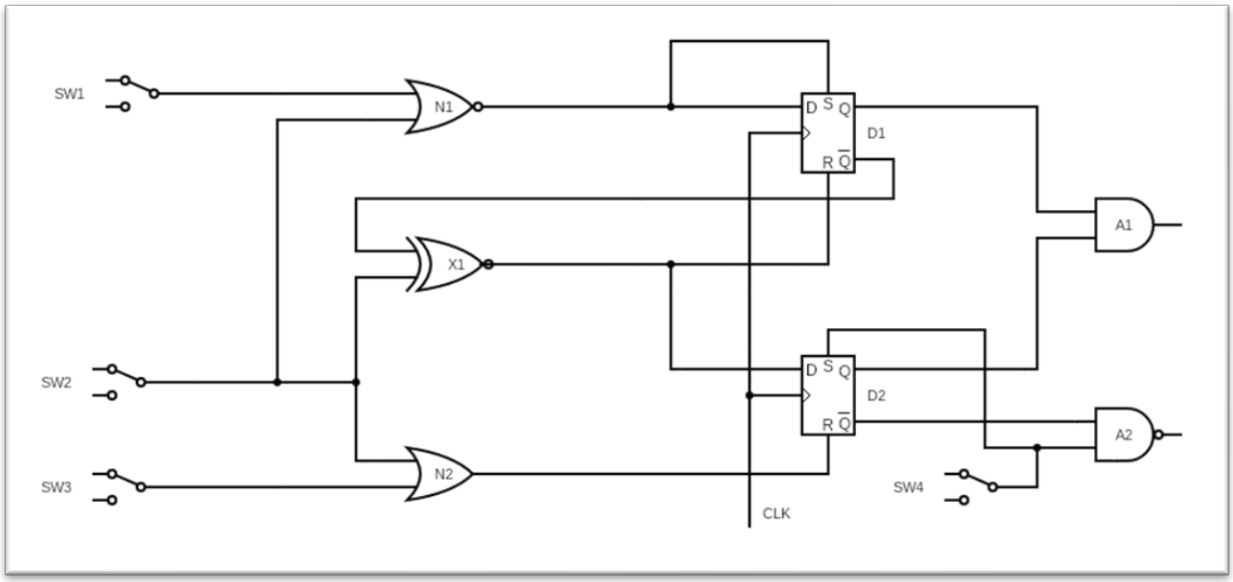


Figure 3: Graph and Definition file by Sherry Fu

```
###
Example 3: Circuit with 2 D-types
###

DEVICES:
SWITCH SW1(0), SW2(1), SW3(0),
SW4(1);
AND A1(2);
NAND A2(2);
XOR X1;
OR N2(2);
NOR N1(2);

DTYPE D1, D2;
CLOCK C(2);

CONNECTIONS:
# N1 NOR gate Connections
SW1 > N1.I1;
SW2 > N1.I2;

# N2 OR gate Connections
SW2 > N2.I1;
SW3 > N2.I2;

# D1 D-Type Connections
N1 > D1.SET;
N1 > D1.DATA;
X1 > D1.CLEAR;
C > D1.CLK;

# D2 D-Type Connections
SW4 > D2.SET;
X1 > D2.DATA;
N2 > D2.CLEAR;
C > D2.CLK;

# X1 XOR gate Connections
D1.QBAR > X1.I1;
SW2 > X1.I2;

# A1 AND gate Connections
D1.Q > A1.I1;
D2.Q > A1.I2;

# A2 NAND gate Connections
D2.QBAR > A2.I1;
SW4 > A2.I2;

MONITOR A1, A2;
```

A4. Corner Case to test GUI

```
### Three Counter Case (no switch)
DEVICES:
CLOCK C(2), C2(7823782);
AND A1(2);
NOR N1(2);
DTYPE D1, D2;

CONNECTIONS:
A1 > D1.DATA;
C2 > D1.SET;
N1 > D2.DATA;
D2.Q > A1.I1;
C2 > D2.CLEAR;
D2.Q > N1.I1;
C2 > D1.CLEAR;
C > D1.CLK;
D1.Q > N1.I2;
D1.QBAR > A1.I2;
C > D2.CLK;
C2>D2.SET;

MONITOR C, D2.Q;
```

A5. Maintenance: New Function Test

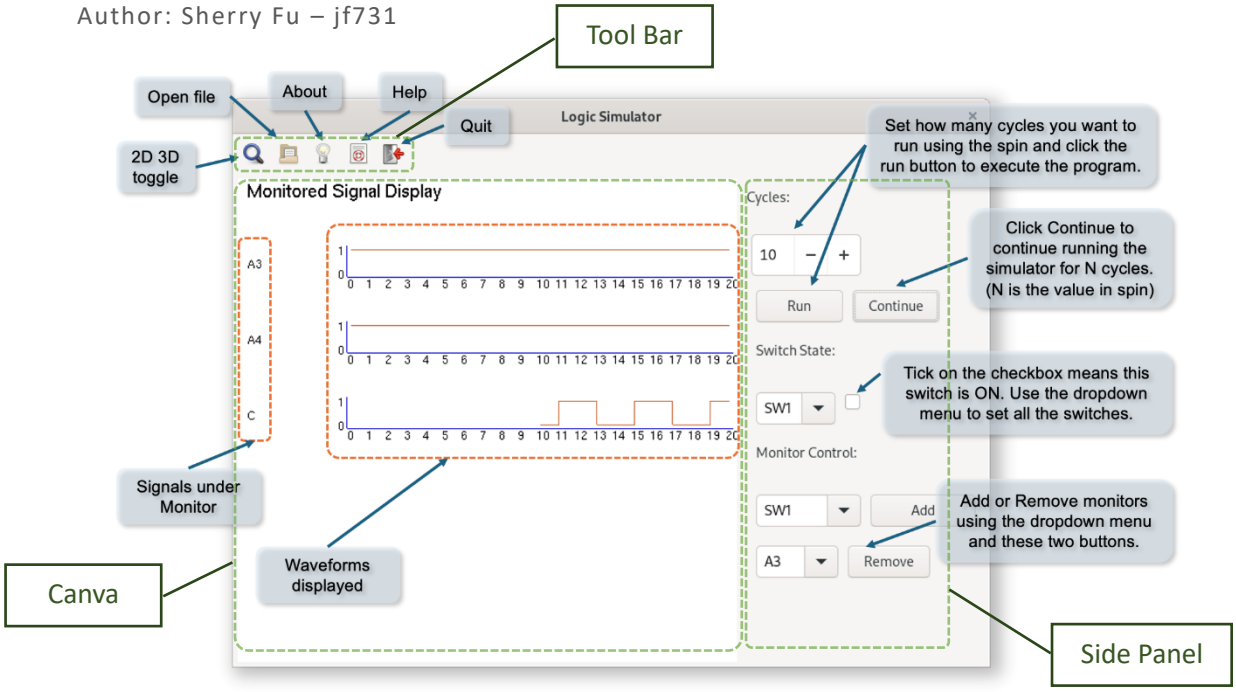
```
###
Testing if the RC device works
###

DEVICES:
RC R1(2), R2(3), R3(4); # The qualifier for RC is the no. cycles before it outputs a low signal
NOR A1(2), A2(2), A3(2), A4(3);

###
If correct, A1 should be high after 2 cycles, A2 after 3 and A3+A4 after 4
###

CONNECTIONS:
R1 > A1.I1;
R2 > A1.I2;
R1 > A2.I1;
R3 > A2.I2;
R1 > A3.I1;
R3 > A3.I2;
R1 > A4.I1;
R2 > A4.I2;
R3 > A4.I3;


MONITOR A1, A2, A3, A4;
```

Appendix B: EBNF Grammar

```
----- EBNF GRAMMAR -----


definition_file = "DEVICES", ":" {device_instantiation}
                    , "CONNECTIONS", ":", {connection}
                    , [monitor];


device_instantiation = device_type, device_name_init, {",", device_name_init} ";" ;
device_name_init = device_identifier, ["(", number ,")"]

device_type = "CLOCK" | "SWITCH" | "AND" | "NAND" | "OR" | "NOR" | "DTYPE" | "XOR" | "RC" ;

monitor = "MONITOR" , output_identifier_group, ";";

connection = output_identifier,  ">", input_identifier, ";";

output_identifier_group = output_identifier , { "," , output_identifier };

output_identifier = device_identifier, [".", ("Q" | "QBAR")];
input_identifier = device_identifier, ".", ("CLK" | "DATA" | "SET" | "CLEAR" | number_input);

device_identifier = letter, {letter | digit} ;
number_input = "I", nonzero_digit, {digit} ;
number = digit, {digit} ;
```

Appendix C: User Guide

Author: Sherry Fu – jf731



To initiate this **Graphic user interface**, you need to download the .zip file or clone the project from GitHub, then type in the following command in the Terminal:

```
python logsim.py eg_def_file_1.txt
```

Remember to check that you are in the folder named **final**. The last part `eg_def_file_1.txt` can be replaced by any other definition files in the folder.

The system supports two languages: English-UK and Chinese-Simplified, if you want to switch to Chinese, please check whether your system setting – language is Chinese-Simplified; You can also type in the following command to settle the language environment in Terminal:

```
LANG=zh_CN.utf8 python logsim.py eg_def_file_1.txt
```

Tool Bar:

2D 3D toggle switches between 2D and 3D canvas. It also reset screen size when pressed.

Open file allows you to select any TXT file in your PC and run this Logic Circuit simulator over it. When you decide the file, the original TXT file would be shown in an additional window.

About button generates an About window showing the information of the development team.

Help button directs to the above Figure, introducing the functionality of each part.

Quit button allows you to quit this simulator, you can carry on with the Text-based user interface.

If you want to initiate with this text-based interface, just type in this command in Terminal:

```
python logsim.py -c eg_def_file_1.txt
```

For other functionalities, just follow the instructions provided above. Hope you have a nice experience with our Logic Simulator! 😊

Appendix D: Brief Description of files

As GitHub lists files in alphabetical order, the description will also follow this sequence.

locale\zh_CN folder: localization files storage, Chinese Simplified translation available

- **gui.po**: A portable object file used in localization that contains human-readable translations
- **gui.mo**: A machine object file used by software to retrieve translations, cannot be viewed directly.

test_parse folder: Test files written specifically for parser.

RC_test.txt: The definition file in Appendix A4, used for new RC function test. (Maintenance phase)

comment_testing.txt: scanner test file

def_file_2_no_comment.txt: Archived test file for GUI.

devices.py: Used in the Logic Simulator project to make devices and ports and store their properties. New **RC** device added in Maintenance phase.

eg_def_file_1.txt, **eg_def_file_2.txt**, **eg_def_file_3.txt**: The definition files in Appendix A1, A2 and A3, the main test files.

eg_errors.txt: scanner and parser test file

gui.py: Implement the graphical user interface for the Logic Simulator.

help.png: Archive of previous help window content (before Maintenance)

help_cn.png: help window content in Chinese (Maintenance phase)

help_en.png: help window content in English (Maintenance phase)

logsim.py: Main program to run. Parse command line options and arguments for the Logic Simulator.

monitors.py: Record and display output signals.

names.py: Map variable names and string names to unique integers.

network.py: Build and execute the network.

parse.py: Parse the definition file and build the logic network.

sample_file.txt: scanner and parser test file

scanner.py: Read the circuit definition file and translate the characters into symbols.

symbol_spam.txt: scanner and parser test file

Test files correponding to all the python files: test_devices.py, test_names.py etc.

three_counter.txt: The definition file in Appendix A5, used for edge case test.

userint.py: Implement the interactive command line user interface.