Research and Development of the Smartie Homie Smart Home System

Younsuk Choi
younsuk.choi@stud.hshl.de

Vu Nguyen
vu-quang-tuong.nguyen@stud.hshl.de

Chen Shih
shih.chen@stud.hshl.de

Üsame Gönül
uesame.goenuel@stud.hshl.de

Thanh Long Phan
thanh-long.phan@stud.hshl.de

Abstract (Nguyen)—Smartie Homie is a project aimed to provide an outline for the future Smart Home Systems, one that focuses more on the general widespread market instead of the enthusiasts' market of today. The project focuses on improving interconnectivity of the systems, their functionality to improve quality of life and a wide range of systems, including security, lighting, device manager, grid system and entertainment, to promote great modularity and choices for the users.

From the prior research conducted, models such as sequence diagrams, use case diagrams, class diagrams and state machine diagrams were formulated to provide a deeper understanding how each system interacts with itself and other systems to be used in the implementation phase of the project.

In the implementation phase, the systems were then simulated within a limited sense due to the resources and time constraints. However, these implementations can display the core functionalities and constraints of the system which then can be analysed. The project shows that, even in limited sense, the smart home systems provide massive benefits in security through smart security systems, efficiency through smart grid and device manager systems and overall improvement in quality of life through smart entertainment and lighting systems while there are many opportunities to improve and expand. Therefore, it is concluded that it should be further explored in the future.

I. Introduction (Nguyen)

Just like computers back in the day, when they were overpriced, oversized and cumbersome, and were only suitable for professional use [1], the smart home systems of today are reserved for technology enthusiasts who can afford it. Despite that, the technology undoubtedly provides their users with an unparalleled quality of life improvement and customizability, bringing many smart systems within the house together and creating an internet of things [2]. All of this is achieved despite the technology is still in its infancy with the first Alexa being released back in 2014 [3].

The smart home system utilizes connection between appliances and other devices. This is done using a central device manager which is used to connect every device into the network and as a messenger between them. The connection protocols to the network can be quite diverse with many possible ways to connect the devices, whether through more established protocols such as Bluetooth and Wi-Fi, more advanced protocols such as Zigbee, or as simple as wired Ethernet [4]. However, the highlight of the Smart Home System is the quality-of-life improvements that these devices bring. Smart lighting systems can be turned on and off remotely simply using voice commands or using their phone as a switch or smart lock system that does not require mechanical mechanisms that are vulnerable [2].

However, with today's technology, the "smart" functions of smart devices are still very limited. Additionally, just like communication devices and computers back in the day, it is highly likely that, with advancement in technology and production techniques, the cost of making a Smart Home System will decrease and lead to the introduction to a wider consumer market. Thus, it is why the Smartie Homie project is started.

II. Motivation (Chen)

The Smartie Homie project was created with the wider market in mind, it aims at outlining the future of smart home systems, improving user experience and quality of life. To achieve this goal, the project focuses on strengthening "interactivity, functionality, and customizability".

High interactivities enable the consumer to manage and monitor the house with ease everywhere. Consumers would also require less time to adapt to the system. Functionality is another core element, things like home automation add convenience to life, so the time that people spend on daily routines can be reduced. Finally, higher customizability indicates a broader user base. Flexible and adjustable functions can fulfil diverse needs, so children, the elderly, and people with disabilities become the beneficiary of the smart home system as well.

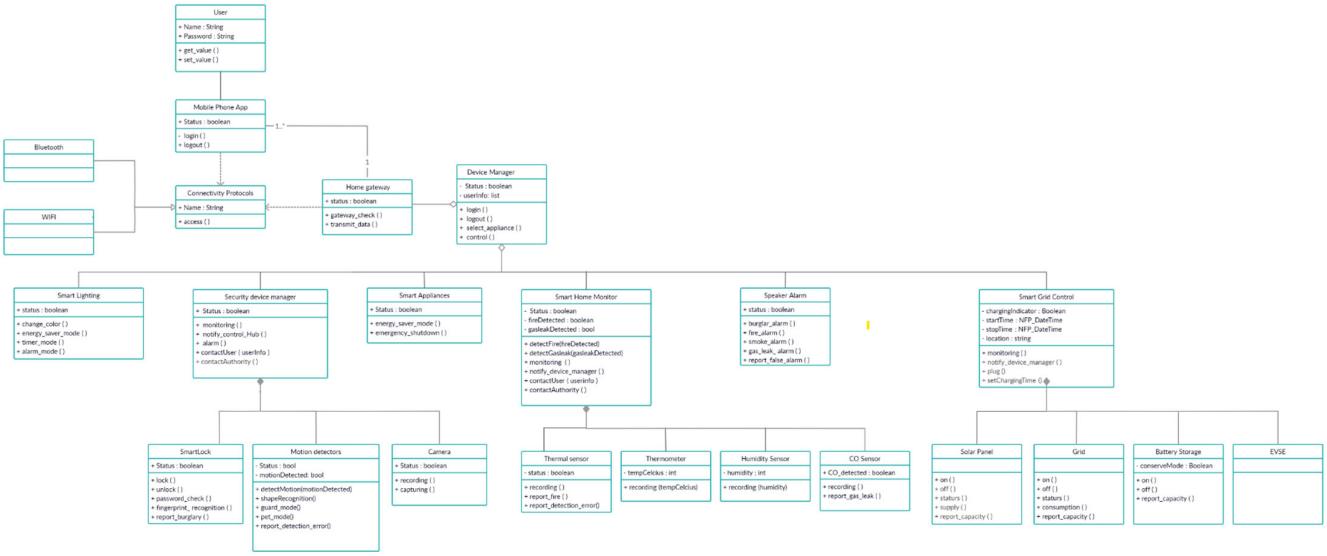

Figure 1. Class diagram of the whole system

III. Methodology (Gonul)

Initially, to gain an insight into the subject, various sources on other smart home studies, existing smart homes and embedded systems (Amazon Echo, August Smart Lock, Ecobee Smart Thermostat) were analysed. This is due to this project's focus being current smart household trends, smart home architecture, and inter-device communication. Furthermore, this research also clarifies many challenges that might be faced during the design stage. These include major challenges like technical limitations and user interaction issues and were tackled or avoided later individually by subsystems. Apart from this, a model home was designed to gain an idea on devices and services that are going to be offered by Smartie Homie.

Following, based on the model home, potential requirements for a basic smart home system were explored. This project's focus point was a system that can cover most of the house: the door, hallway, and the rooms. Interaction of the components within each other, as well as with the user was also a substantial requirement. To further concretize the concept of the house, with the help of specifically written scenarios, UML models were developed. Initially, a sequence diagram was developed for each scenario. The purpose of the sequence diagrams is to enable a better understanding in the interactions and collaborations between users and devices [5]. Use case diagrams were formulated next, to provide insight on the requirements for each subsystem and system-specific use cases [6]. Class diagrams were also used to illustrate relationships between classes and interfaces [7].

For each of the subsystems, individual state machine diagrams were designed. Modelling the behaviour of components and specifying different states of operation, would make the forthcoming implementation stage of the project more convenient [8]. These diagrams also clarified the path Smartie Homie was taking, allowing better structure to be formed around the idea.

Subsequently, initial implementations were developed to visualize the system. Each subsystem was converted into functional code with the help of the UML diagrams. This led the project to complete digitalization, which shifted the development's focus point from versatility to functionality. In other words, the concrete functions were decided on and were now to be reliably implemented.

To validate the functionality, individual code tests were written for every single subsystem. These tests utilized functions and constant values used in the codes. By randomizing the inputs and comparing respective outputs were able to confirm flawless operation.

Several subsystems even had the opportunity to utilize hardware components with the aid of Tinkercad, an online modelling program that allows users to build hardware systems. Despite Tinkercad offering Arduino Uno as the sole controller on the website, simple implementations were mostly unaffected, although this would affect the project's general realization, due to lack of processing power.

IV. Concept

A. General system (Phan)

Power, lighting, security, home appliances are basic aspects of a house, therefore these elements are incorporated into the system.

- For security, there are Smart Lock and Security Camera.
- For lighting, there is Smart Lighting.
- For power, there is Smart Grid.
- For home appliances, the Smart Entertainment module within the Smart Appliances sub-category is developed.

A Device Manager will be needed to control these subsystems, therefore designing a Device Manager is necessary.

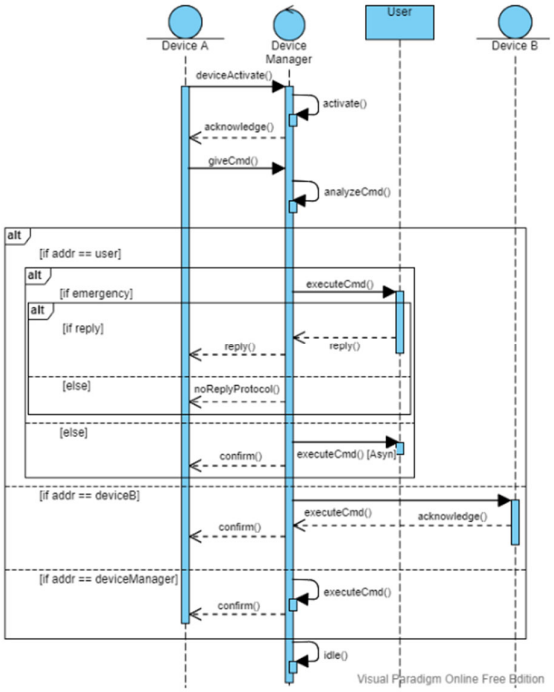

Figure 2. Sequence diagram for devices

Communication: To reduce the cost, Bluetooth is used to connect subsystems to each other [9], and Wi-Fi network is only used to connect the phones to the Device Manager.

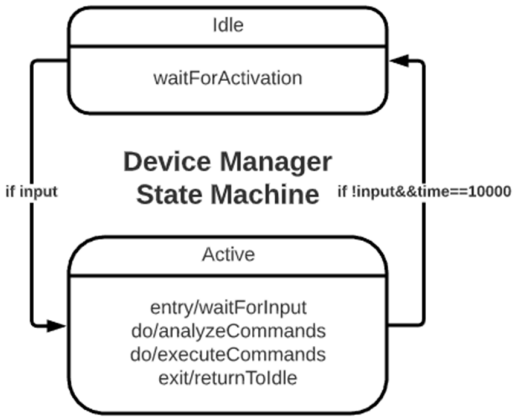

Figure 4. State machine diagram for device manager

From the requirements above, a class diagram for our system is developed, which can be seen in Figure 1.. The class diagram shows how subsystems communicate with each other, and that the user can control things by connecting to the Device Manager.

B. Device manager (Nguyen)

As mentioned briefly in the introduction, and the general systems, the device manager is used as a central connection point and as a message transporting network for all devices. This can also be seen in the class diagram, as seen in figure 1. It can also function as a personal assistant for the user using command recognition or as a speaker.

For communication protocols, as the role of a device manager is to both receive and send information, it requires at least a half-duplex communication protocol where it can send and receive data, albeit not simultaneously.

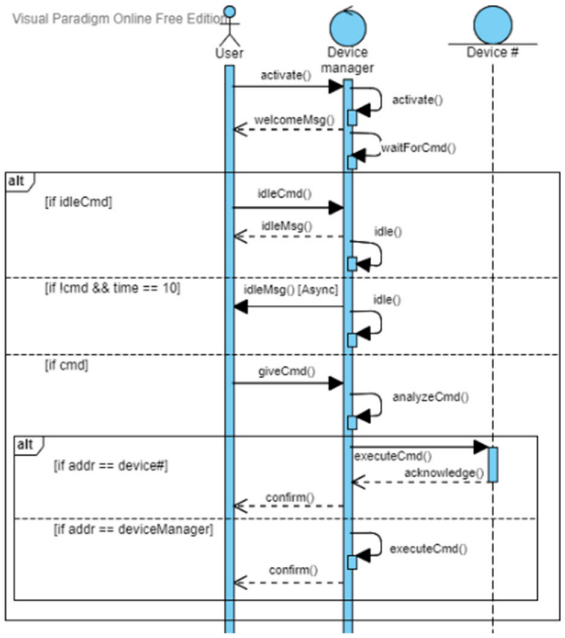

Figure 3. Sequence diagram for user

Additionally, the protocol should be wireless for ease of use. This can be protocols such as Wi-Fi. However, since it is likely that the device manager will have to manage multiple devices, it is best to also implement full-duplex protocols such as Bluetooth [4] so that the device manager can better handle multiple commands in parallel.

Looking at the figure 2 and figure 3, while the device must discern between device communication or user commands, the concept is the same. The device manager first receives a command, analyses the command, decodes the receiving address and executes it accordingly.

However, there are differences between user assistance and device communication. The activation for the device and the user must be different since the two protocols act differently. Additionally, since the user can unintentionally activate the device or change their mind, they can wait 10 seconds or cancel, and the device will return to idle. Whereas, for device communication, for devices such as smart security systems or smoke alarms, there could be emergency scenarios that require the user to respond. If the user does not respond, a no-response emergency protocol will be triggered. Also, with device communication, there will always be a definite endpoint, therefore if the command is finished, the device returns to idle, whereas

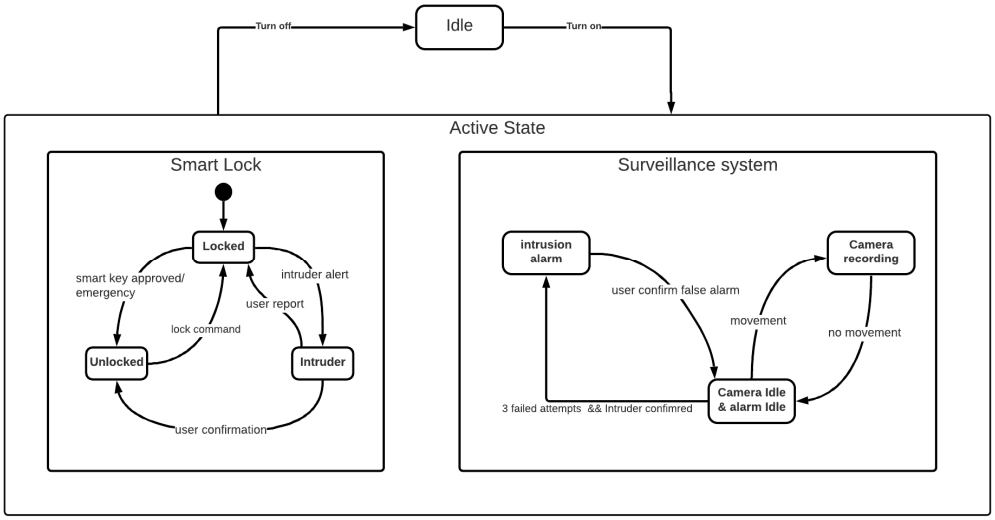

Figure 5. State machine diagram for security system

PPafter the command is finished for the user, the device manager will return to the start of the active state to wait for any other requests from the user.

Using this information, the device manager can be realised using 2 states, namely idle and active, as seen in figure 4. Within the idle state, the device simply waits for activation. In the active state, the device first waits for the command, analyses it and lastly, executes it. Once the counter expires or the device is forced to return to idle, the device then returns to idle.

C. *Smart security (Chen)*

As a member of the smart home, security system serves as a guard to the residence. It monitors activities around the house and secures the homeowner against intrusion. When emergencies occur, it warns users and assists in an evacuation.

Two major elements that constitute the security system, are smart lock and surveillance service respectively. The following sections detail both.

1) *Smart lock*

If the house is a castle, smart locks are the sentries. It is their duty to secure the entrance against intruders and protect the property of the residents. A series of measures are designed to prevent intrusion. Under default circumstances, doors remain locked until entered passwords are verified. Smart lock keeps count failed attempts and notifies the user when the count reaches three. Once the user authenticates the report, an alarm would be raised to deter the intruder.

Under dangerous circumstances such as fire, any door leading to the outside is potentially an emergency exit. In that case, the smart lock would unlock the door so that people in the house can escape easily.

2) *Surveillance system*

The surveillance system keeps an eye on the environment and tracks any suspicious activity in the surrounding area. The cameras are set at entrances to detect

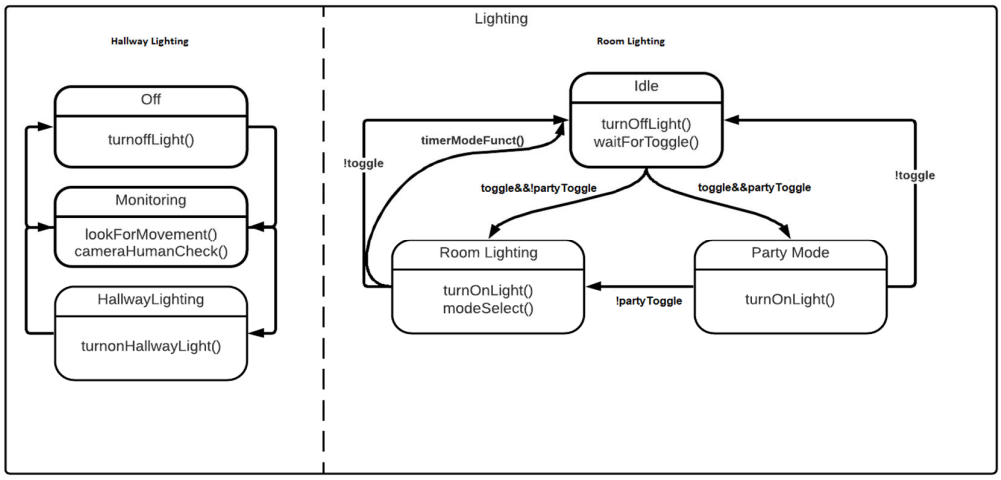

Figure 6. State machine diagram for smart lighting

and identify moving objects. The passive recording is a feature of the surveillance system. Instead of recording all the time, the camera records a video only when there are movements detected. As a result, the surveillance can reduce memory usage and achieve energy efficiency.

D. Smart lighting (Phan)

Smart Lighting system is responsible for the automatic control of the house's lighting. There are two parts of the system that work independently from each other: Hallway Lighting and Room Lighting, which are demonstrated in the State Machine Diagram in Figure 1.

1) Hallway Lighting

Hallway lighting automatically controls the lighting of the hallway without the need of input from the user. To do that, a sensor and a camera are utilized. The sensor constantly looks for movement, and when it is triggered, it sends a request for the camera to do a check-up of the outline size of the object. If the object's outline size is big enough to be of a human, then the state will be changed to HallwayLighting and the light will be turned on. The light will be on for 6 seconds, then the state goes back to Monitoring where the sensor looks for movement. If no movement is found, the state will go to Off state and light will be turned off. If Movement is detected, the camera will check the size of the object again, if it is big enough to be human, the state will go to HallwayLighting and light will be on. If it is not big enough to be human, light will stay off.

2) Room Lighting

Room lighting let the user control the lighting of the room through a smartphone. With a connected smartphone, the user can turn on & off the lights. There are also two additional features, Timer Mode and Party Mode. Timer mode will automatically turn off the light after an X amount of time designated by the user. Party Mode will make the RGB light start blinking similar to in disco floor. To achieve these, 3 switch states are utilized: Idle, RoomLighting and PartyMode. The initial state is Idle, and when Toggle is on and partyToggle is off, the state will be changed to Room Lighting and Light will be on. Then, there are two way to turn off the Light and change the state to Idle: turning off Toggle or initiating Timer Mode Function - which will turn off the light after a certain amount of time chosen by the user. Another feature available is Party Mode. When Toggle and PartyToggle are on, the state will be switched to PartyMode and the RGB light will be on and will keep blinking. When partyToggle is off, the state will go to RoomLighting where the light is back to normal, and if Toggle is off (meaning the light is toggled off), the state will change back to Idle where the light is off.

E. Smart grid (Gonul)

Smart grid is a subsystem of the smart home that addresses mostly the reliability of the system. In other terms, it assures that the system gets enough power and, in case of any failure, attempts to resolve the issues independently. As a consequence, smart grid can be viewed as an amalgamation of two major sections: Grid Operation and Grid Monitoring, respectively. Following, a demonstration of the idea and design behind these two sections can be found:

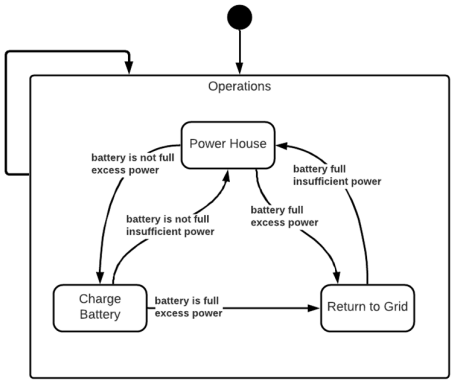

Figure 7. State machine diagram for grid operation

1. Grid Operation

Grid Operation's main concern is to manage and balance the incoming and outgoing power. It takes into consideration three units: solar panels, home battery, and local electrical grid. Depending on the electrical power coming from solar panels and consumed power through the home, this system switches through different power modes. If sufficient power is produced through the solar panels, any discharged battery will be charged and after all supplementary batteries are full, excess produced power will be shared with the local power grid, helping energy scarcity throughout the city. On the other hand, if the panels don't produce any power, first the batteries will be used at utmost efficiency, and later power will be drawn from the city's electrical grid.

As seen in Figure 7. state machine diagram, Operation is a single looping state that consists of three substates, namely: "Power House" state, "Charge Battery" state and lastly "Return to Grid" state. First state, "Power House" state, is intended to be the default state for the grid. In this condition, the system draws power from solar panels, batteries and lastly the city's electrical grid, in that order. If the produced energy is detected to result in excess, the system changes to one of the other two states. Empty batteries would lead to "Charge Battery" state, and full batteries would lead to "Return to Grid" state. "Charge Battery" state is, as the name suggests, where the excess energy is directed to the batteries. Again, full batteries would lead to "Return to Grid" state, meanwhile insufficient power would lead back to initial "Power House" state. Finally, when the system reaches "Return to Grid" state, excess electrical energy is distributed to the city's grid. Once more, insufficient power leads to initial state.

In this way, the house will not only take advantage of renewable energy sources, but also guarantee reliable functionality by having a reserve, meanwhile also aiding the community's energy needs.

2. Grid Monitoring

Grid monitoring, as the name suggests, monitors the system for any possible faults in the grid system. This might be a fault in batteries, or in solar panels or even an

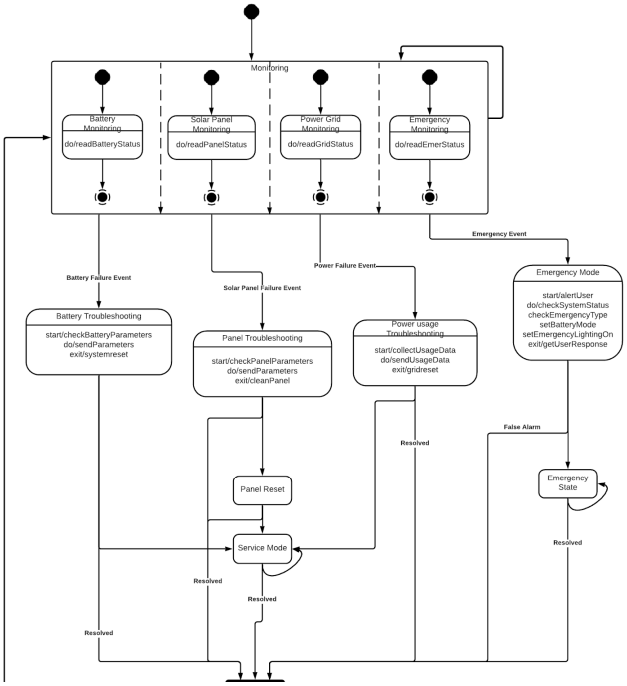

Figure 8. State machine diagram for grid monitoring

emergency situation. Grid monitoring takes action accordingly, while simultaneously alerting users (or authorities in some cases). First step taken by grid monitoring, after the fault is detected, is troubleshooting, which in most cases means collecting data about the error, following certain resolution steps, and resetting the indicated component. If the issue persists, authorized technicians will be contacted, and the system will undergo service maintenance. This ensures that the smart home system does not get compromised due to minor defects.

Additional to general unit monitoring, emergency monitoring is also included in this subsystem. In a basic sense, the central focus of emergency monitoring is to detect and act on various emergency situations, from simple disruptions to life-threatening natural disasters. When such a threat is detected, the first course of action is to analyse the circumstances and alert the users. If it is deemed to be a false alarm, after it is confirmed by the user, the system would return to monitoring state. Alternatively, in a real emergency, the system would undergo an emergency state protocol, which are predetermined steps of action to be followed by both, the system and the user, depending on the situation. In addition to this, any authorities will be contacted and notified of the reported emergency. For instance, in case of a fire, the users will be alerted and guided outside, the gas pipeline will be cut off, and firefighters will be notified, all happening parallelly. After the emergency is considered to be over, the Grid Monitoring system will return to damage diagnosis and resolution automatically.

State machine diagram for this subsystem can be seen on Figure 8. Initial state is the "Monitoring" state. This includes numerous smaller tasks (in this instance there are four) that are done parallelly. These tasks read statuses of specific components to seek out errors and faults. If any of the tasks return a positive event flag, state changes to one of the troubleshooting states (with emergency being the highest priority). Each of these Troubleshooting states contain component-specific resolutions and include final testing to confirm that the system is working as intended. While resolution leads to initial "Monitoring" loop, unresolved component problems lead to "Service Mode". When the system is in "Service Mode", authorized technicians are notified of the faulty part and sent to the address (if the customer gives their confirmation). "Emergency Mode" differs from common device troubleshooting, although its function is similar. System collects as much data as possible to determine the correct emergency protocol and requires user's confirmation. After the emergency is deemed to be true, the system proceeds to "Emergency State".

While the Grid system handles energy consumption, it also assumes the role of a "watchtower" against system faults and emergencies. For Smart Grid, solar panels were chosen as a renewable energy source instead of alternatives (wind turbines, geothermal plants), as it is currently the most efficient means in the consumer market.

F. Smart entertainment (Choi)

As "Home" usually refers to a place where one lives and it often being a place of relaxation and entertainment, the necessity for developing such a system was taken into consideration as well. Smart Entertainment is therefore designed so that users can enjoy entertainment content within the designed smart home system. The main concern for Smart Entertainment is to incorporate several smart entertainment devices such as Smart TV into one domain so that the connected smart entertainment devices will be controlled by a user within the Smart Entertainment system.

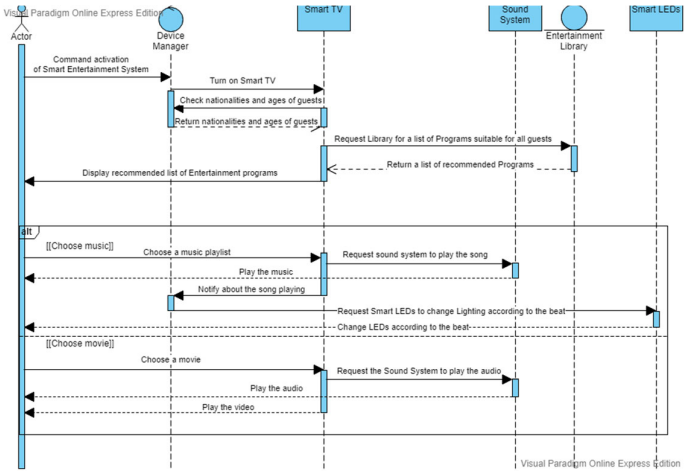

Figure 9. Sequence diagram for smart entertainment

According to the proposed scenario and the sequence diagram (*figure 9*) which were designed first, Smart Entertainment needs to have certain availabilities. First, it has to be able to take users' information and process it for appropriate program recommendation and inappropriate material censor. Also, it has to allow users to turn on and

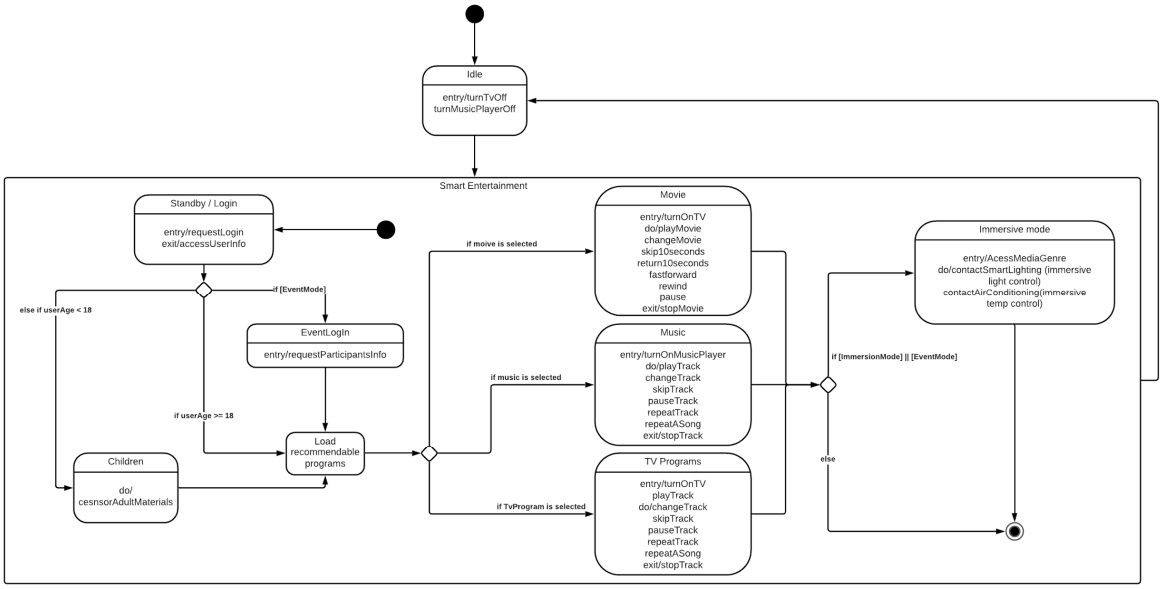

Figure 10. State machine diagram for smart entertainment

off smart devices and play relevant content on the devices. Furthermore, Smart Entertainment needs to enable some additional features such as the event mode, which is developed for multiple users, as well as the immersive mode, which automatically controls the Smart Home environment in accordance with the vibe of the content being played.

Based on these, the structure of the Smart Entertainment system was specified. (*figure 10*) When a user triggers the system to operate by giving any input to the system, the system will request the user's information for the upcoming censoring function as well as the event function. Then the system will show an appropriate list of programs that could be recommended based on the provided user information. Then, the user can choose a program to play as well as the suitable device to play it on. Finally, the user can decide whether to initiate immersive function so that the surrounding environment can be adjusted in accordance with the program being played. For constructing this system, Smart Entertainment is intended to be realised in four states, namely "Idle" state, "Login" state, "ChooseEntertainment" state, and "Active" State. (*figure 11*)

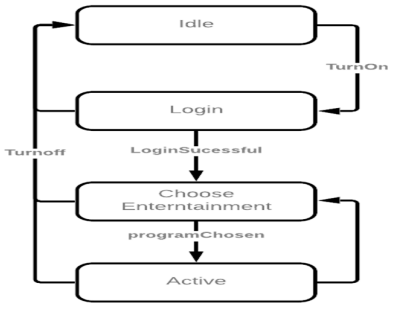

Figure 11. Revised state machine diagram for smart entertainment

```
34          switch (state)
35          {
36          case idle:
37              waitForActivation(activation, stat
38              break;
39          case active:
40              loginUser(loginStatus, userList, l
41              chooseCommand(loginStatus, state,
42              break;
43          case controlDevice:
44              chooseDevice(deviceList, deviceCou
45              deviceControl(deviceList, deviceCc
46              break;
47          case management:
48              chooseManagementFunction(mode, sta
49              managementMode(mode, userList, use
50              break;
51          }
```

Figure 12. State of the device manager

V. EVALUATION

A. Device manager (Nguyen)

1) Computer implementation

With the computer implementation of the device manager, the system is designed as a user interface and focuses on the simulation of the management and control capabilities. Due to the complexity and limitation of resources, the program is developed closer to an application control for the device manager than the usual application for the device manager, command using voice. The user must choose from a list of defined functionalities instead of commanding with their own wording. While this

does not fully simulate the user interaction with the device, it is possible to simulate the fundamental control and management aspects of the device manager.

The code is first mapped to the state machine diagram using switch cases. Looking at figure 12, the code inherits the two fundamental states idle and active. As seen in the code, the idle state is where the device manager simply

```
19   class user
20   {
164
165  class device
166  {
```

Figure 13. Classes of the device manager

```
206  void addNewUser(std::vector<user> &userList, int &
207  {
208      userList.push_back(user());
209      usleep(2000);
210      ClearScreen();
211      userCount = userList.size();
212      int i = userList.size() - 1;
213      userList[i].getUserName();
214      userList[i].getGender();
215      userList[i].getCurrentDate();
216      userList[i].getUserDoB();
217      userList[i].convertDoB2Age();
218      userList[i].assignUserID();
219      usleep(1000);
220      std::cout << "Account created successfully!" <
221      usleep(2000);
222      ClearScreen();
223  }
224
225  void addNewDevice(std::vector<device> &deviceList,
226  {
227      deviceList.push_back(device());
228      usleep(2000);
229      ClearScreen();
230      deviceCount = deviceList.size();
231      int i = deviceList.size() - 1;
232      deviceList[i].getDeviceName();
233      deviceList[i].assignDeviceID();
234      usleep(1000);
235      std::cout << "Device added successfully!" << s
236      usleep(2000);
237      ClearScreen();
238  }
239
```

Figure 14. Vectors of the device manager

waits for activation. Once the correct activation code is recognised, the device manager goes into the active state. Within the active state, the user can choose from a list of functions, including control devices and manage devices and users. However, problems arise when only using these two fundamental states when trying to develop a UI system with defined functionalities, the code will get cluttered and become too complicated. Therefore, for this implementation, the active state is divided into two extra states for each of the currently implemented functionalities, namely control and management. In control state, the user can control all the devices that they have added. Presently, the control function is limited to switching the devices remotely. In the management state, the user can connect their smart devices and add other users for privacy and separate preferences profiles when interacting with connected devices.

Looking at the figure 13, the class diagram shows that the device manager interacts with 3 parent classes: devices, user and internet. Apart from the internet class, these classes were mapped to the code using classes and vectors, with vectors acting as a list of elements for each class, as

```
66  void deviceClassTest(int &testResult, int i)
67  {
68      device newDevice;
69      newDevice.deviceName = deviceNameList[i]; // Assign it a name
70      string a = newDevice.returnDeviceName();
71      if (a != deviceNameList[i])
72      {
73          cout << "ERR_3: Return device name is faulty." << endl;
74          testResult = testFailed;
75      }
76      int b = rand() % 2; // Randomise a variable
77      newDevice.status = b;
78      int c = newDevice.returnDeviceStatus(); // Test return device status function
79      if (b != c)
80      {
81          cout << "ERR_4: Return device status is faulty." << endl;
82          testResult = testFailed;
83      }
84      newDevice.deviceSwitch(); // Test device switch
85      c = newDevice.returnDeviceStatus();
86      if (b == c)
87      {
88          cout << "ERR_5: Device switch is faulty." << endl;
89          testResult = testFailed;
90      }
91  }
```

Figure 15. Test code for the device manager

seen in figure 14.

The code was tested using a self-developed test framework. The implementation code was edited so that the part that requires input from the user was removed. Within the test code, each variable is assigned a value, or the value is randomly generated and is tested accordingly, as seen in the example shown in figure 15. Using this test code, it is possible to test whether the code functions as intended, albeit not being able to test user inputs. Using these tests, it can be concluded that the code works without error as the code runs through the tests 10 times without issues.

2) Hardware implementation

Hardware implementation of the device manager aims to simulate the interaction and communication between the users and devices and devices and devices. This implementation simulates two scenarios. The first scenario is where there are two connected devices and the user wanted to switch them on and off. The second scenario is where the smoke detector detects smoke and contacts the device manager. The device manager then tells the user that there are smokes and sounds the alarm.

Looking at the circuit in figure 16, since Tinkercad does not support any of the mentioned communication protocols within the concept part, the device manager is connected to the other devices using 2IC protocol. 2IC is deemed suitable because, comparing between 2IC, UART and SPI, 2IC allows the maximum number of connected devices, namely 128 individual devices with SPI only allowing 4 devices and UART was designed for the communication between 2 devices. 2IC is also a half-duplex communication protocol. While it is not as powerful as SPI, as mentioned in the concept part, it is sufficient. An

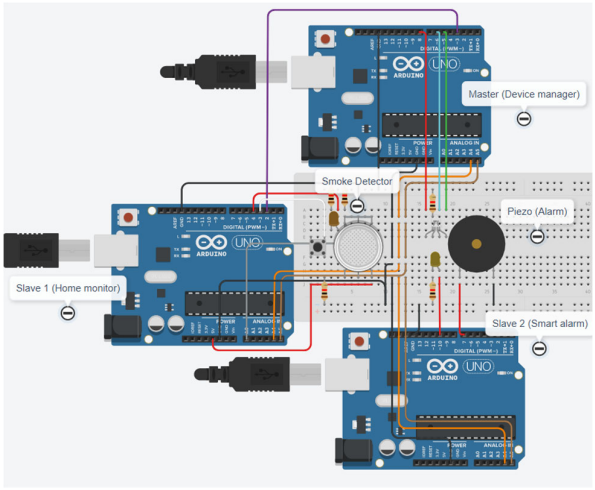

Figure 16. Circuit of the device manager

```
267  void loop()
268  {
269    switch (state)
270    {
271    case idle:
272      turnOnIdleLight();
273      printIdleMessage();
274      waitForActivation();
275      break;
276
277    case active:
278      turnOnActiveLight();
279      printActiveMessage();
280      waitForCommand();
281      analyseCommand();
282      executeCommand();
283      returnToIdle();
284      break;
285    }
286  }
```

Figure 17. Arduino states for the device manager

additional downside of 2IC when compared to UART is that the 'slave' devices can only communicate with the 'master' devices when the 'master' is explicitly asking information from it whereas, for UART, the connected devices can communicate whenever they must. However, a way to bypass this downside is to connect a pin of the slave's to the master's interrupt pin and set it as output. Whenever the slave wants to talk to the master, the slave activates the master's interrupt pin which triggers a function where the master opens a communication with them. Another compromise due to limitation of the tools is that there is no mic module so instead serial communication using UART is used [10].

Looking at the code for the device manager in figure 17, in the loop function, it is possible to see that the state machine diagram is more authentically mapped. The idle and the active state functions as mentioned in the concept, with an addition of it printing messages to notify the user.

Looking at the analyseCommand() function, which can be seen in figure 18, the device manager simply compares the input with a library of known commands and sets the flags accordingly. For example, if the user type in the serial

```
146  void analyseCommand()
147  {
148    if (commandReceived)
149    {
150      digitalWrite(GREEN, LOW);
151      digitalWrite(BLUE, HIGH);
152      delay(500);
153      if (receiveBuffer == turnOn1)
154      {
155        chooseDevice = device1;
156        turnOn = 1;
157        delay(100);
158      }
159      else if (receiveBuffer == turnOn2)
160      {
161        chooseDevice = device2;
162        turnOn = 1;
163        delay(100);
164      }
```

Figure 18. Analyse command for the device manager

```
222      if (smokeAlarm)
223      {
224        Wire.beginTransmission(device2);
225        Wire.write(SmkAlrm);
226        Wire.endTransmission();
227        Serial.print("SMOKE! SMOKE! SMOKE!\n");
228        deviceCall = 0;
229        delay(100);
```

Figure 19. Execute command for the device manager

monitor 'turn on 1', the device manager will recognise the command and switch the turnOn flag, which is the analysis command part mentioned in the concept, and chooseDevice will be set to device1, which is the address decoding. This can then be used for the executeCommand() function. If the user types a random command, the device simply returns an unknown command message.

Looking at the executeCommand() function, as seen in figure 19, the device manager simply acts as according to the task flag that was flipped. Looking at the example, if the smoke detector tells the device manager that there is smoke, it opens a communication with the alarm system and tells it to sound. Then it prints a message to the user to notify that there is smoke detected.

B. Smart security (Chen)

Security system guarantees the safety of residents and security of property. Therefore, its implementation was delivered with a mission. To offer the users best protection, the security system fulfils all the criteria of a dependable system, which are availability, reliability, safety, and security respectively. The software implementation exhibits these four qualities, and there are further descriptions in the next few sectors. For some detailed information please refer to the diagram as well.

Hardware implementation is also carried out in the project. Although the implementation was constrained due

to limited resources, it virtually fulfills all the requirements based on its initial design.

1) *Software implementation*

This part outlines the structure of codes and explains how the program works in general. In addition, descriptions of functions in the program are provided as supplementary information.

In order to ensure the availability and reliability of the system, the possible scenarios and corresponding reactions are taken into account in the design phase. The state machine diagram *(see fig.5)* describes the relations among states, including what triggers the transfer of the states. To ensure that the implementation is consistent with the design, the program is created according to the state machine diagram, which prevents mistakes and confusions.

The states can be considered as the skeleton of the code, and functions are the flesh *(see fig.20)*.The program is divided into different states, and each state represents a situation that may happen to the house. The switch case statement controls the flow of states, specifying different codes that should be executed in various conditions. Underneath the states are functions, they are the executors who carry out the program. In order to reduce the overall complexity of the code, all the functions are defined outside of the main function *(see fig.21)*.

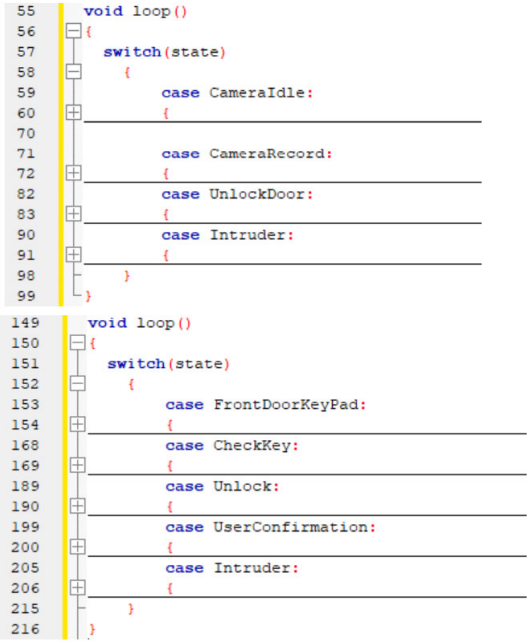

Figure 20. State of security system

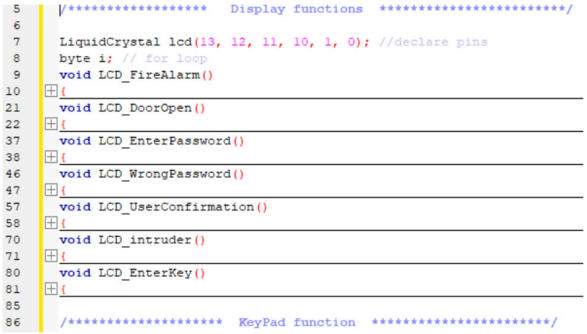

Figure 21. Functions of the security system

As mentioned above, the project is benchmarked against the Dependable system, therefore, rigorous examinations were conducted during the test phase. The program passes one hundred rounds of tests, the results exhibit the availability and reliability of the Security system.

The libraries provide rich collections of built-in functions *(see fig.22)*, and some of them serve as device drivers, for this reason, these functions play an important role in the embedding programs into the hardware. Further details regarding how embedded software interacts with the device can be found in the next part.

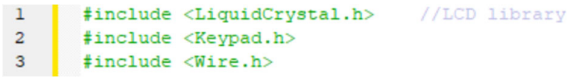

Figure 22. Built-in libraries

2) *Hardware implementation*

This part covers the selection of hardware, the overall design, the peripherals, and the embedded software.

Out of the consideration of limited resources and ease of collaboration, Tinkercad is chosen as the development tool for hardware implementation. Since Tinkercad only supports Arduino Uno at present, it has naturally become the final choice.

In terms of hardware, the security system is composed of two motherboards, one bread board, and peripherals. The number of ports on Arduino is inadequate for connecting all the external devices, thus an additional motherboard is configured.*(see fig.23)* The one on the top (hereafter referred to as board 1) is responsible for user interface such as LCD display and the keypad. The one at the bottom (hereafter referred to as board 2) manages peripherals like smoke detector, PIR motion sensor, piezo buzzer, servo, and LED.

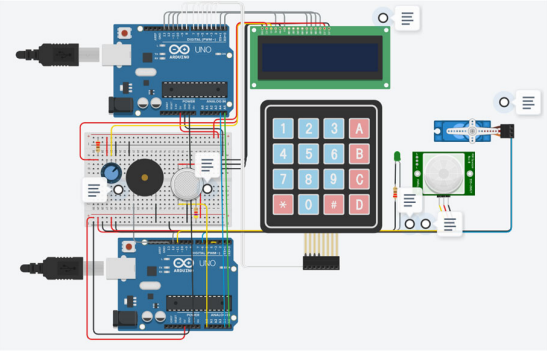

Figure 23. Prototype of the security system

When the board 1 receives the correct passwords from the keypad, the LCD screen would pop up a message that says ”Door is open”. In the meantime, it transmits a signal to board 2 and subsequently triggers the servo to rotate. In this case, two boards are programmed to communicate with each other in a Master Writer/Slave Receiver configuration via the I2C synchronous serial protocol *(see fig.24)*. As said in the concept part, the smart lock would inform the user if 3 failed attempts occur. After the user authenticates the report, the system raises an intruder alarm. As for hardware implementation, once the intrusion happens, piezo buzzer on board 2 would sound the alarm.This function is accomplished through I2C communication as well.

```
192         Wire.beginTransmission(4);    // transmit to device #4
193         Wire.write(signal);              // sends one byte
194         Wire.endTransmission();       // stop transmitting
195         LCD_DoorOpen();
196         state = FrontDoorKeyPad;
```

Figure 24. I2C communication

In emergencies, the security system must provide protection immediately to minimize damages. Hence, quick reaction is of great importance. In response to time critical events such as fire, gas leak, and intrusion, the interrupts are configured *(see fig.25)*. Interrupts are a mechanism that forces the processor to stop to react to external stimuli that are being fed to the system. Once that event has wrapped up, the processor would continue the previous task. In the program, external stimulus from the smoke detector triggers an interrupt, and the alarm and servo on board 2 are activated immediately.

```
50    attachInterrupt(digitalPinToInterrupt(SmokeISR_input), Smoke_ISR, HIGH);    //set ISR
51    attachInterrupt(digitalPinToInterrupt(PassISR_input), Pass_ISR, HIGH);

100   void Smoke_ISR()
101   {
102     tone(Piezo,700);
103     interrupt();
104     servo1.write(90);
105     state = CameraIdle;
106     //Serial.println(state);
107   }
108
109   void Pass_ISR()
110   {
111     state = UnlockDoor;
112   }
```

Figure 25. Interrupts

C. *Smart lighting (Phan)*

Implementation:

For the implementation, two separated programs are written, one for Hallway Lighting and one for Room Lighting. The implementation programs are mapped from the state machine diagram to the code, as can be seen in Figure 26 and 27.

```
int main()                                          //Hallway Lighting
{
    for (int i = 0; i < 10; i++)
    {
    switch (state)
    {
    case off:
        turnoffLight(hallwayLightOutput, state);
        break;

    case monitoring:
        lookForMovement(movementDetected, height, state);
        cameraHumanCheck(humanDetected, height, state);
        break;

    case hallwayLighting:
        turnonHallwayLight(hallwayLightOutput, height, state);
        break;
    }
    }
    return 0;
}
```

Figure 26. Main program code for hallway lighting

1. Hallway Lighting

lookForMovement() is responsible for reacting to the event triggered by the movement sensor when movement is detected. When *movementDetected* = 1 (meaning motion is detected by the sensor), it initiates another function *cameraHumanCheck()*. This function will look for the outline size of the object, if it is to be about the

```
int main()
{
    for (int i = 0; i < 10; i++)
    {
    switch (state)
    {
    case idle:                                                    //
        turnOffLight(lightOutput);
        waitForToggle(toggle, partyToggle);
        if(toggle&&partyToggle)
        {
        state = partyMode;

        }
        else if(toggle && !partyToggle)
        {
        state = roomLighting;
        }
        break;

    case roomLighting:                                            //
        turnOnLight(lightOutput);
        modeSelect(timerMode, desiredTime);
        if(timerMode)
        {
            timerModeFunct(desiredTime, state, timerMode, sleeptime);
        }
        else
        {
            toggleModeFunct(state, toggle);
        }
        break;

    case partyMode:                                               //
        turnOnLight(lightOutput);
        cout << "Party Mode on" << endl;
        cout << "Light is blinking in disco mode!" << endl;
        while(partyToggle)
        {
            checkForInput(toggle, partyToggle, state);
        }
        break;
    }
    }
    return 0;
}
```

Figure 27. Main program code for room lighting

size of a human (determined by the height value - for now we choose the value of between 100cm and 250cm, but these values can be finetuned further), it will change the state to *hallwayLighting*. In this state, a function is run, *turnonHallwayLight()*, and the light will be on. After a duration of 6 seconds, the *height* value will be reset, and the state will go back to *monitoring* state, where *lookForMovement()* is run again to detect movement. If no more movement is detected, the state will change to *off* state where *turnoffLight()* is run to turn off the light, and if movement is detected, it goes back to the previous loop.

2. Room Lighting

The initial state will be *Idle* state where *turnOffLight()* is run to make sure the light is off. Then, function *waitForToggle()* is run, it is responsible for reading the input of *toggle* (toggle the light) and *partyToggle* (toggle the Party Mode). If toggle and partyToggle are on, then the state is *partyMode*. If *toggle* is on and *partyToggle* is off, the state will be *roomLighting*.

- In the state of *roomLighting*, *turnOnLight()* is initiated to turn on the Light. Then *modeSelect()* will be engaged for the user to decide to use Timer Mode or not. If user chooses to use Timer Mode and enters a desired time, *timerModeFunct()* is run to shut off the light after the amount of time decided by the user. If the user chooses to not use Timer Mode, *toggleModeFunct()* is initiated, it waits for the user's input of turning off the light (when *toggle* is off). When the user decides to turn off the light, the state goes back to *idle*.
- In the state of *partyMode*, *turnOnLight()* is initiated to turn on the Light. Then, as long as *partyToggle* is on, *checkForInput()* is run to wait for the user's command to turn off the light or turn off Party Mode. When the user decides to turn off the light (*toggle* is off), partyToggle is toggled off as well as the state goes back to *idle* where *turnOffLight() is run to turn off the light.* When the user decides to turn off Party Mode (*partyToggle* is off), the state goes back to *roomLighting* and the program continues to function as described in the *roomLighting* section above.

Test code:

Defect testing is done via two separate testing programs, one for Hallway Lighting and one for Room Lighting. Each function is tested to make sure all functions will work correctly.

The result of defect testing is that both programs passed, which can be seen in figure 28.

```
PS C:\Users\Cortana> cd "d:\Programming\GitHub\ELE-03-Microcontroller-Group\Smartie Homie\3. Design\Codes\SmartLighting\Test"
PS D:\Programming\GitHub\ELE-03-Microcontroller-Group\Smartie Homie\3. Design\Codes\SmartLighting\Test> & .\"Smart Lighting-roomLight
ingTest.exe"
Congrats, test passed!
PS D:\Programming\GitHub\ELE-03-Microcontroller-Group\Smartie Homie\3. Design\Codes\SmartLighting\Test> cd "d:\Programming\GitHub\ELE
-03-Microcontroller-Group\Smartie Homie\3. Design\Codes\SmartLighting\Test"
PS D:\Programming\GitHub\ELE-03-Microcontroller-Group\Smartie Homie\3. Design\Codes\SmartLighting\Test> & .\"Smart Lighting-hallwayLi
ghtingTest.exe"
Congrats, test passed!
PS D:\Programming\GitHub\ELE-03-Microcontroller-Group\Smartie Homie\3. Design\Codes\SmartLighting\Test>
```

Figure 28. Result of defect testing for both programs

Simulation:

To show that the program would work in reality, a simulation for Hallway Lighting scenario is done through Tinkercad.

The program code is slightly modified to work with Tinkercad's environment. Here, a sensor is used to detect movement, when the movement is detected, the *height* value is input as a simulation for the real value in reality, and when it is within the range of a human, the light is on for 6 seconds. After that, it loops back to the previous loop of looking for movement, if there's no movement, the light will be off, and if there's movement, the program starts checking for the *height* value again.

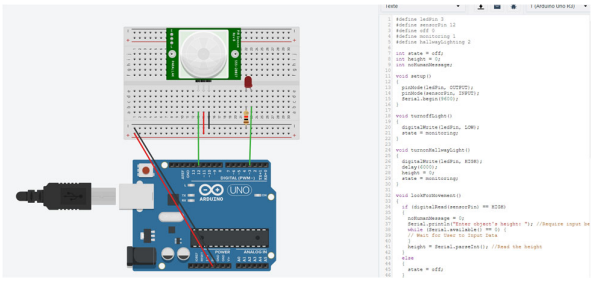

Figure 29. Simulation for the motion sensor feature

This simulation is included in GitHub repository at the address of "Smartie Homie\3. Design\Codes\SmartLighting\Simulation".

D. Smart grid (Gonul)

First line of action while designing the Smart Grid, was emphasizing its power management functionality. Subsequently, reliability of the system was also found to be as important. This led to the division of the grid into two equally important sections with different structures. The implementation had to be designed similarly. Diagrams in Figure 7. And Figure 8. show the basic organization, and later were used as the basis for the implementation code.

The most apparent difference between these two subsystems is that Grid Monitoring is a solely internal operation and is limited to a negligible number of outputs. On the other hand, Grid Operation can be somewhat presentable with visual cues. This led to different implementation approaches during the development stage. For the sake of better demonstration Grid Operation is designed to utilize hardware model while Monitoring remains on software.

Below are both implementation paths of the subsystems explained in detail:

1. Grid Operation

In the first stages of development, Grid Operation was planned to be software-implemented only. In reference to the state machine diagram (Figure 7.) designed in the research phase, a switch case equivalent was written on C++ language. To focus further into the subject, Figure 7. And Figure 30. are demonstrated. Figure 30 is the direct representation of Figure 7., as: lines 27-43 represents "Power House" state, lines 44-60 is for "Charge Battery" state, and finally lines 61-71 stand for "Return to Grid". In this early stage of implementation, code was limited to placeholder functions for a mere illustration. Essentially, the program started in "PowerHouse" state and shifted into other two states depending on "bstat" (battery state) which represents battery percentage with a number between 0 and 100, where the latter is represented by "full". Of course, it would happen only if "pgain" (Power Gain or Produced

```
        switch (state)
        {
        case PowerHouse:
            use();
            if (bstat < full && pgain > pstat)
            {
                state = ChargeBattery;
                return state;
                break;
            }
            else if (bstat >= full && pgain > pstat)
            {
                state = Return2Grid;
                return state;
                break;
            }
            else
                return state;
            break;
        case ChargeBattery:
            chargeBattery();
            if (bstat >= full && pgain > pstat)
            {
                state = Return2Grid;
                return state;
                break;
            }
            else if (pgain <= pstat)
            {
                state = PowerHouse;
                return state;
                break;
            }
            else
                return state;
            break;
        case Return2Grid:
            sharepower();
            if (pgain <= pstat)
            {
                state = PowerHouse;
                return state;
                break;
            }
            else
                return state;
            break;
        }
```

Figure 30. First code for grid operation

Power) is bigger than “pstat” (Power State or Power Usage). Full battery led the system to “Return2Grid” state and less than 100 would mean that the state would change into “ChargeBattery”. On the other hand, “pgain” less than “pstat” would return the grid back to “PowerHouse” state.

Later, it was improved into the code in Figure 32. to add actual functions. This also led to a change in states, thus also a change in codes structure. New code now consists of four states:

- “NoPow” meaning no power state, as the system relies on city’s electrical grid (no energy from the panels or the batteries).
- “Charge” state means that enough power is produced to run the house and excess is being used to fill the batteries.
- “Share” state delivers the excess energy to local electricity network.
- “Discharge” state relies on battery supply to power the house.

Instead of a single “PowerHouse” state like in the previous plan, this design utilizes “NoPow” and ”Discharge”.

Transmission between these states is similar to the previous code. That being, the initial state is “NoPow” state and later changes to “Charge” (for insufficient battery) and “Share” (for full battery) depending on the battery percentage (“battery”) and of course incoming power (“inc”). If incoming power is lower than used power (“use”), no change happens. Similarly, “Charge” state changes into “Share” if the battery is full and there is excess energy, and into “Discharge” if battery is not yet full but energy is insufficient. “Share” returns to “Discharge” if power is insufficient and “Discharge” can lead to “Charge” or “NoPow” depending on the difference between incoming and used power.

```
void checkstates()
{
    if (readBatteryStatus())
        count[0] = 1;
    else
        count[0] = 0;
    if (readPanelStatus())
        count[1] = 1;
    else
        count[1] = 0;
    if (readGridStatus())
        count[2] = 1;
    else
        count[2] = 0;
    if (readEmerStatus())
        count[3] = 1;
    else
        count[3] = 0;
}

void movetostate()
{
    printf("Analyzing issues");
    loading();
    if (count[3] == 1)
    {
        printf("Emergency!!!\n");
        Sleep(600);
        state = EmergencyMode;
        count[3] = 0;
    }
    else if (count[0] == 1)
    {
        printf("Battery is faulty\n");
        Sleep(600);
        state = BatteryT;
        count[0] = 0;
    }
    else if (count[1] == 1)
    {
        printf("Solar Panel is faulty\n");
        Sleep(600);
        state = PanelT;
        count[1] = 0;
    }
    else if (count[2] == 1)
    {
        printf("Power Grid is faulty\n");
        Sleep(600);
        state = PowerT;
        count[2] = 0;
    }

    else if (std::equal(std::begin(count), std::end(count), std::begin(empty)))
    {
        printf("All issues resolved!\n");
        Sleep(1000);
        state = Monitoring;
        printf("Returning to Monitoring mode\n");
        Sleep(1000);
    }
}

int getUserResponse()
{
    char response;
    do
    {
        printf("[y/n]\n");
        cin >> ("%c", &response);
        if (!cin.fail() && response != 'y' && response != 'n')
            printf("Please enter a valid response\n");
    } while (!cin.fail() && response != 'y' && response != 'n');
    if (response == 'y')
        return 1;
    else
        return 0;
}

bool isitresolved()
{
    printf("Is the issue resolved?\n");
    return getUserResponse();
}

bool isitfalse()
{
    printf("Is it a false alarm?\n");
    return getUserResponse();
}

void loading()
{
    Sleep(400);
    printf(".");
    Sleep(400);
    printf(".");
    Sleep(400);
    printf(".\n");
}
void printemergency()
{
    printf("Emergency State Activated\n");
    printf("Contacting Authorities");
    loading();
}
void printdeepem()
{
    printf("Please follow Emergency Procedures");
    loading();
    Sleep(1000);
    printf("Emergency over. Analyzing damages");
    loading();
    movetostate();
}
void printservice()
{
    printf("System in Service Mode\n");
    printf("Contacting Authorized Service");
    loading();
}
void printdeepserv()
{
    printf("System in deep Service Mode\n");
    Sleep(1000);
    printf("Sending Maintenance Personnel");
    loading();
    printf("Repair in progress");
    loading();
    movetostate();
}

void monitor()
{
    checkstates();
    printf("Monitoring");
```

Figure 31. Grid monitoring functions

```
switch (state)
{
case NoPow: //Power House State
  lcd.setCursor(0, 1);
  lcd.write("No Power        ");
  if (inc > use)
  {
    if (battery >= 100)
    {
      state = Share;
    }
    else
    {
      state = Charge;
    }
  }
  break;

case Charge: //Charge Battery State
  battery = charging(battery);
  lcd.setCursor(0, 1);
  lcd.write("Battery Full    ");
  delay(500);

  if (inc > use)
  {
    state = Share;
  }
  else if (inc <= use)
  {
    state = Discharge;
  }
  break;
case Share: //Return to Grid State
  lcd.setCursor(0, 1);
  lcd.write("Sharing         ");
  if (inc <= use)
  {
    state = Discharge;
  }
  break;
case Discharge: //Power House State
  battery = decharging(battery);
  if (inc > use)
    state = Charge;
  if (inc <= use)
    state = NoPow;
  break;
}
```

```
int charging(int bat)
{
  lcd.setCursor(0, 1);
  lcd.write("Charging        ");
  while (bat < 105)
  {
    printlevel(bat);
    delay(10);
    bat++;
  }
  return bat;
}

int decharging(int bat)
{
  lcd.setCursor(0, 1);
  lcd.write("Discharging    ");
  while (bat > 0)
  {
    printlevel(bat);
    delay(20);
    bat--;
  }
  return bat;
}

int blevel(int bat)
{
  if (bat >= 100)
    return 7;
  else if (bat >= 80)
    return 6;
  else if (bat >= 60)
    return 5;
  else if (bat >= 40)
    return 4;
  else if (bat >= 20)
    return 3;
  else if (bat >= 0)
    return 2;
  else
    return 1;
}

void printlevel(int lvl)
{
  lcd.setCursor(15, 0);
  lcd.write(blevel(lvl));
}
```

Figure 32. Grid Operation switch case and functions

Later improvements showed that designing a hardware model was also possible. The circuit in Figure 33. was designed for this purpose. Two potentiometers were used to simulate the changes in incoming and used electrical power and LCD display presented the resulting state including an image representing battery percentage.

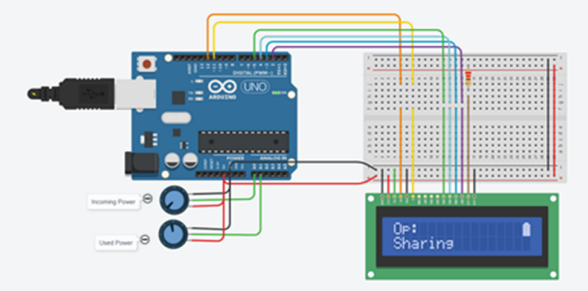

Figure 33. Grid model in Tinkercad

```
#include <LiquidCrystal.h>
#define INC 0
#define USE 1
#define MinUsage 200

int blevel(int bat);
void printlevel(int lvl);
int charging(int bat);

int const NoPow = 0;
int const Charge = 1;
int const Share = 2;
int const Discharge = 3;

int inc;
int use;
int state;

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
//set battery icon
byte BatterylCD1[8] = {0x0E, 0x11, 0x11, 0x11, 0x11, 0x11, 0x1F};
byte BatterylCD2[8] = {0x0E, 0x11, 0x11, 0x11, 0x11, 0x1F, 0x1F};
byte BatterylCD3[8] = {0x0E, 0x11, 0x11, 0x11, 0x1F, 0x1F, 0x1F};
byte BatterylCD4[8] = {0x0E, 0x11, 0x11, 0x1F, 0x1F, 0x1F, 0x1F};
byte BatterylCD5[8] = {0x0E, 0x11, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F};
byte BatterylCD6[8] = {0x0E, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F};
byte BatterylCD7[8] = {0x0E, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F};
```

Figure 34. Grid operation statements and variables

Figure shows the necessary LCD library and functions included in this code. “MinUsage” in line 5 assigns an arbitrary minimum usage value of a home. Lines between 21 and 29 sets up the LCD and creates an alterable battery image on it. Finally, “battery” value is assigned a random

```
switch (state)
{
case Monitoring:
    monitor();
    break;

case EmergencyMode:
    ement();
    if (isitfalse())
    {
        movetostate();
        break;
    }
    else
        state = EmergencyS;
    break;

case BatteryT:
    batt();
    if (isitresolved())
    {
        movetostate();
        break;
    }
    else
        state = ServiceMode;
    break;
case PanelT:
    pant();
    if (isitresolved())
    {
        movetostate();
        break;
    }
    else
        state = ServiceMode;

    break;
case PowerT:
    powt();
    if (isitresolved())
```

```
case PowerT:
    powt();
    if (isitresolved())
    {
        movetostate();
        break;
    }
    else
        state = ServiceMode;
    break;

case Panelreset:
    panelres();
    if (isitresolved())
    {
        movetostate();
        break;
    }
    else
        state = ServiceMode;
    break;

case ServiceMode:
    printservice();
    if (isitresolved())
    {
        movetostate();
        break;
    }
    else
        printdeepserv();
    break;
case EmergencyS:
    printemergency();
    if (isitresolved())
    {
        movetostate();
        break;
    }
    else
        printdeepem();
```

Figure 35. Monitoring code

starting number since the current prototype does not have a real battery. As seen in Figure 32. various functions were used to simulate real-life mechanics of battery and grid (*charging()* increases battery percentage over time, *discharging()* decreases it).

2. *Grid Monitoring*

Although the initial stages of Monitoring were quite similar to Operation, in many ways, it came to differ in the late development. Similar to the Operation, the code was written based on the state machine diagram Figure 8, with the illustrative help of placeholder functions. Later, when actual functions came into play, a better organized code was developed. This can be seen in Figure 35. Every single state in the diagram was transformed into a different case and each of them led consecutively according to the diagram. While main code consists only of this switch case in loop, there also exists a header file with all functions and constants Figure 31.

As in the diagram, the initial case is “Monitoring”. Here, random events will be generated and stored in “count” array as flags. The array consists of as many elements as there are monitored devices. Next step in monitoring is to decide which state to transmit first, and by the predetermined priority of the events the transmission happens. In this case, emergency even has the highest priority and other events will be postponed until this is resolved. “EmergencyMode” state determines the validity of the event by inquiring about it from the user. If it is a false alarm, the system proceeds to the next event. A real event leads to the “EmergencyS” state (emergency state) where the steps to be taken are simulated. Figure 36. shows the output of the system during an unresolved emergency. Due to technical and time limitations it is only simulated but can be expanded to real-life responses when possible.

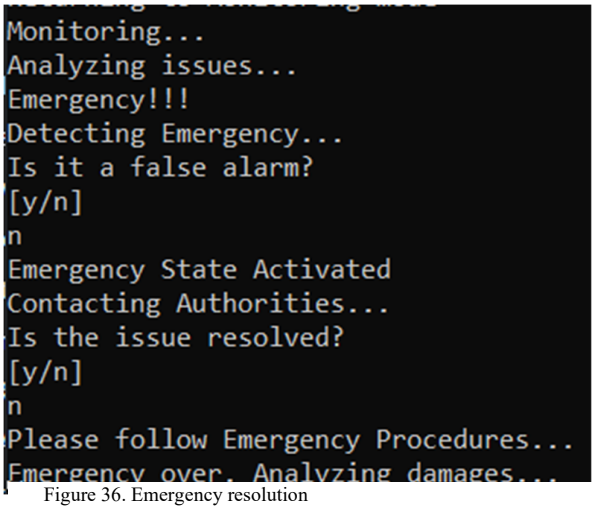

Figure 36. Emergency resolution

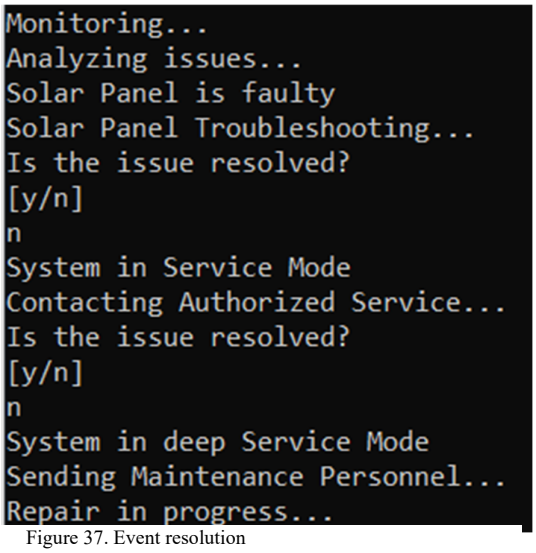

Figure 37. Event resolution

Three other Troubleshoot states "BatteryT", "PanelT", "PowerT" have the same structure. They simulate an analysis sequence and inquire for a response about the resolution. If the subject is resolved, they lead to the next event resolution, otherwise they initiate the service mode. "ServiceMode" state simulates another resolution sequence, as seen in Figure 37, with two layers of action. First service sequence only notifies the technicians and assumes the user or the system to solve the issue by technicians' help and second one directly sends the technicians to deal with the faulty hardware. After the issues are resolved, systems return to the initial "Monitoring" state.

A short explanation of the core functions can be found below:

- "*checkstate*" formats the flags randomly generated by "*read...status*" function to an array for better utility.
- "*movetostate*" is the main engine of this program. Proceeds to the states that are 1, depending on the flags, and sets them to 0 after they are resolved.
- "*getUserResponse*" requests user's y/n response and returns it in 0/1 format.

Remaining functions act as visual responses to inner operations and utilize *Sleep()* to simulate time delay during processes.

```
121 int main()
122 {
123     while (n==0){
124         switch(state){
125             case IdleState:
126             {
127                 wakeUpFunction();
128                 break;
129             }
130             case LoginState:
131             {
132                 censorFunction();
133                 eventFunction();
134                 break;
135             }
136             case ChooseEntertainmentState:
137             {
138                 chooseDeviceFunction();
139                 chooseContentsTypeFunction();
140                 break;
141             }
142             case ActiveState:
143             {
144                 chooseProgramFunction();
145                 chooseImmersiveFunction();
146                 completeMessage();
147                 break;
148             }
149         }
150     }
151     return 0;
152 }
```
Figure 38. Main program for smart entertainment

After the implementation was done for both sections, testing was required, to validate the programs functionality. Grid Operation test code was deemed to be difficult to write, as it was designed for and written on Tinkercad, and used different libraries and functions that don't work on other environments. On the other hand, Monitoring has a test code, since it was written in compatible form. This test code also refers to the main code's header file and has access to all its functions and variables. Grid Monitoring test consists of three functions that test three main functions of Monitoring implementation. These are: *checkstateTest, movetostateTest, userresponseTest*. Each of the functions use a random number generator and test the main function by checking its output. Any unexpected response will result in the program ending and printing out the specific error code on the screen. Although they don't directly test the said functions, test functions are modified copies of the originals, which work the same way.

E. Smart entertainment (Choi)

Based on the state machine diagram (*Figure 11*), the implementation program is written using switch cases in a loop. The four main states are Idle, Login, ChooseEntertainment and Active while each state has its designated functions. (*figure 38*)

When no input is taken by the system, in Idle state, the system remains dormant and waits for user input. Any input from a user will be taken by *"wakeUpFunction"(line 127, figure 38)* and it will switch to Login state. In Login state, by *"censorFunction"(line 132, figure 38),* the system will request the user age to censor inappropriate materials from the upcoming recommended list of programs. *"eventFunction"(line 133, figure 38),* designed for multiple people in a group, will assist *"censorFunction"* in its effect, checking whether there is need for censoring by asking the user if there is any minor member in the group.

Subsequently, the system will be switched to ChooseEntertainment state which has two functions. With *"chooseDeviceFunction"(line 138, figure 38),* the user decides the device to play the program on and the system will turn it on. With *"chooseContentsTypeFunction"(line 139, figure 38),* the user decides the types of the content which needs to be considered for browsing the recommended list. Lastly, in Active state, the user can choose a program from the recommended list by *"chooseProgramFunction"(line 144, figure 38)* and play it on the pre-selected device. Also, here, the user can turn on the immersive setting by *"chooseImmersiveFunction"(line 145, figure 38)*, which will automatically adjust the surrounding temperature as well as lighting in consonance with the program being played. The system will eventually return to the Idle state.

```
113  void loop()
114  {
115      switch(state){
116
117      case IdleState:
118        printIdleMessage();
119        waitForInput();
120        break;
121
122      case ActiveState:
123        printChooseDeviceMessage();
124        waitForChoice();
125        printImmersiveModeMessage();
126        waitForImmersiveModeChoice();
127        executeInput();
128
129      }
130  }
131
```

Figure 39. Main program for Arduino

```
94   void setup(){
95       pinMode(Phone, OUTPUT);
96       digitalWrite(Phone, LOW);
97
98       pinMode(TV, OUTPUT);
99       digitalWrite(TV, LOW);
100
101      pinMode(MusicPlayer, OUTPUT);
102      digitalWrite(MusicPlayer, LOW);
103
104      pinMode(ImmersiveLightSwitch, OUTPUT);
105      digitalWrite(ImmersiveLightSwitch, LOW);
106
107      lcd.begin(16,2);
108      pinMode(Button, INPUT);
109      digitalWrite(Button, LOW);
110
111  }
```

Figure 40. Main setup for Arduino

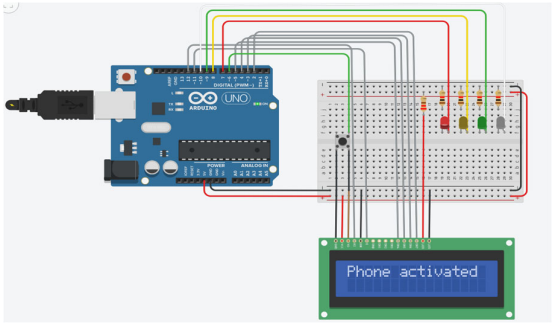

Figure 41. Arduino for smart entertainment

To simulate that the system could be realized in the real world, the Arduino model on Tinkercad was designed. The main concern in this simulation was to visualize human-and-computer interaction and to have all the procedures in a loop so that a user can automatically start again after a loop. In the course of this hardware implementation stage, the system was reduced down to a simpler version, which is reflected on the arduino code. (*figure 39*). Note that there are only two states (Idle and Active) in a loop in contrast to the state machine diagram (*figure 10*) or the implementation code (*figure 11*) where there are four states.

In this model, LCD screen and LED bulbs are used for different purposes. (*figure 41*) LCD screen will display commands to be executed while LED bulbs will work as indicators which show whether one function works. For example, on the Active state, for the *printChooseMessage* (*line 123, figure 39*), it will print out "Choose device, 1 for Phone, 2 for TV, 3 for Player" on the LCD screen. If a user inputs 1, LCD will display "Phone activated" and the red LED will turn on to show that TV is turned on in real time. This loop of codes can be interrupted when an invalid input is taken. Then it will return to the initial state.

VI. SUMMARY AND OUTLOOK

(Choi) In the attempt to improve users' quality of life and the interactivity of the smart home and the users, the general systems of the smart home were explored. With the help of design models such as sequence diagrams and state machine diagrams, a better understanding of how the system intended to work was achieved. Based on this understanding, the simulation and the implementation was developed to represent main functionalities and practicality that the system withholds. In the end, the fundamentals of the smart home system were developed successfully. To recapitulate, the project reflects that the designed smart home model offers greater efficiency and convenience to the users as well as the potential for future modification.

Presently, the hardware prototype remains in the simulator rather than the physical world because the time and resources for implementation are under limits, and the financial constraint is the major hurdle. However, these hurdles mean that there is room for growth, and more functionalities are waiting to be discovered. If hardware simulation was the milestone of the system development, the next destination is going to be a real prototype.

(Nguyen) For device manager, the systems that are currently available in the market only support voice recognition, otherwise, the users must use the phone application. This is very disadvantageous for the elderly,

people with strong accents or people with disabilities. In the future, it is possible that gesture sensing could be incorporated to give commands instead. This could be done using the camera systems already present within the house or possibly external devices such as smart wristbands, gloves or watches.

(Chen) In order to provide residents with better protection, it is necessary to monitor the house thoroughly and find out potential dangers. Therefore, the security system would be equipped with more sensors and detectors in the future. Importing AI visual recognition techniques to the system is also part of the optimization, it would increase accuracy and efficiency of surveillance service, and subsequently improve overall security.

(Phan) For Smart Lighting, the method for human detection is still at the early concept phase, and it can be reworked to improve accuracy using techniques such as artificial intelligence machine vision.

(Choi) For Smart Entertainment, the goal to be reached is for users to have complete and convenient control over all the connected devices and media through voice, remote control or app. The current model can surely be improved further that it has a more fundamentally unified interface to control smart devices as well as environmental factors such as ambient lighting. This could be reached by considering a smart hub or smart controller as an option for a central point of contact for wireless devices communicating through several wireless protocols. Smart Entertainment can be improved even further, that it embraces some degree of home automation where the system automatically could prepare environmental factors such as lighting, and audio environment based on the user usage pattern [11, 12].

(Gonul) For Smart Grid other energy source alternatives may be considered if they are deemed to be more accessible. Currently the project only takes advantage of solar panels for their ease of use and abundance, although it can be upgraded to multi-source plants for higher utilization. Another point that can be taken into consideration would be the number of devices being monitored, as this system is designed to include only the most basic elements of the grid. An ideal monitoring service would also include components of other subsystems, which would then require better architecture and less technical limitations. This can be achieved with better networking and powerful controllers. As it was mentioned numerous times before, Smart Grid can only be considered a simulation at this point of time, in comparison to its potential.

VII. ANNEX

A. *Tinkercad simulation links:*

Device manager:

https://www.tinkercad.com/things/lbuU14TwMKz-device-manager-implementation/editel?sharecode=aOk95dTqB1sae-c_pNja5m7MvUST997nJFVqztTPfdA

Security system:

https://www.tinkercad.com/things/iDdVn9ZmYVo

Lighting:
https://www.tinkercad.com/things/7yJEaHMjb9V-dazzling-amur-albar/editel?sharecode=eg3UwrttMnvwhteQ7Hekixg1-q8DbBAOBJuQZ8ZaDvs

Electrical grid:

https://www.tinkercad.com/things/3hgVqPC0hgf-power-manager/editel?sharecode=dKLS4oWn_yTaSd0yUZhXytcfcE1CbfAe-mNVzQJ96F8

Entertainment:

https://www.tinkercad.com/things/egMFsRJBUad-entertainment-system-implementation/editel?sharecode=FSRjZYW3oivuofhcxngLC8oKfjxvkwGojAnHgBT-cbs

B. *Contribution tally*

1) *Lines count:*

- Vu Nguyen: 1729 lines of code
- Thanh Long Phan: ~765 lines of code
- Üsame Gönül: 627 lines
- Chen Shih: 593 lines
- Younsuk Choi: 603 lines

2) *Percentage contribution:*

- Vu Nguyen: 20%
- Thanh Long Phan: 20%
- Üsame Gönül: 20%
- Chen Shih: 20%
- Younsuk Choi: 20%

C. *Folder hierarchy*

- Sequence Diagrams & Class Diagrams: Smartie Homie\2. Analysis
- Code: Smartie Homie\3. Design\Codes
- State Machine Diagrams: Smartie Homie\3. Design\Diagrams

VIII. REFERENCES

[1] C. Trueman, "The Personal Computer - History Learning Site", History Learning Site, 2015. [Online]. Available: https://www.historylearningsite.co.uk/inventions-and-discoveries-of-the-twentieth-century/the-personal-computer/.

[2] "Top 10 Benefits of Smart Home Technology", Smarteon, 2020. [Online]. Available: https://www.smarteon.ae/post/top-10-benefits-of-smart-home-technology.

[3] B. Vigliarolo, "Amazon Alexa: Cheat sheet", TechRepublic, 2020. [Online]. Available: https://www.techrepublic.com/article/amazon-alexa-the-smart-persons-guide/.

[4] DERKA, "Smart Home Protocols Explained", Medium, 2017. [Online]. Available: https://medium.com/iotforall/smart-home-protocols-thread-zigbee-z-wave-knx-and-more-71efa4b410e1.

[5] Sparx Systems. n.d. UML 2 Tutorial - Sequence Diagram. [online] Available at: <http://sparxsystems.com/resources/uml2_tutorial/uml2_sequence diagram.html>.

[6] Sparxsystems.com. n.d. Use Case Diagram - UML 2 Tutorial | Sparx Systems. [online] Available at: <https://sparxsystems.com/resources/tutorials/uml2/use-case-diagram.html>.

[7] Sparxsystems.com. n.d. Class Diagram - UML 2 Tutorial | Sparx Systems. [online] Available at: <https://sparxsystems.com/resources/tutorials/uml2/class-diagram.html>

[8] Sparxsystems.com. n.d. State Machine Diagram - UML 2 Tutorial | Sparx Systems. [online] Available at: <https://sparxsystems.com/resources/tutorials/uml2/state-diagram.html>

[9] Obaidat, M. and Nicopolitidis, P., 2016. Smart Cities And Homes. ch.2.

[10] "Arduino Communication Peripherals: UART, I2C and SPI", Seeed Studio, 2020. [Online]. Available: https://www.seeedstudio.com/blog/2019/11/07/arduino-communication-peripherals-uart-i2c-and-spi/.

[11] Triton. n.d. Smart Home Integration. [online] Available at: <https://tnt.co.bw/smart-home-integration/>

[12] Arsham Hatambeiki, Universal Electronics Inc., 2017. Smart Entertainment in the Smart Home.

IX. AFFIDAVIT

We (Chen Shih, Thanh-Long Phan, Üsame Gönül, Vu Nguyen, Younsuk Choi) herewith declare that we have composed the present paper and work ourselves and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance.