Software System Analysis and Verification

Simon Platz Beck-Nielsen
Technical Faculty, Software Engineering
University of Southern Denmark
Odense, Denmark
sibec19@student.sdu.dk

Kasim Emre Sahin
Technical Faculty, Software Engineering
University of Southern Denmark
Odense, Denmark
kasah19@student.sdu.dk

Daniel Bahrami
Technical Faculty, Software Engineering
University of Southern Denmark
Odense, Denmark
dabah20@student.sdu.dk

Marc Bertelsen
Technical Faculty, Software Engineering
University of Southern Denmark
Odense, Denmark
berte20@student.sdu.dk

***Abstract*—The report detail the analysis, design, and verification of a smart-lock IoT device. The designed system features an smart-lock which is the actual locking mechanism, an ESP32 responsible for handling the wireless communication to the smart-lock, a server handing the communication between users and the ESP32, and users who interact with the system by sending commands. In the report, various requirements for the system are formulated and later verified using timed automata in UPPAAL.**

I. Introduction

This report presents a case study of a smart lock system, illustrating the interaction between the user and the smart-lock through a smart-home system. The user communicates with a server through a smart-home application, which publishes messages to an MQTT[1] topic. These messages are then subscribed to by an ESP32[2] device that collects them. The received messages serve as commands for the smart lock, controlling it's motor to lock or unlock based on the user's instructions. To display this model, the model and verification tool UPPAAL[3] is used. This model is created to verify the functional and non-functional requirements of the smart lock system, to create a trustworthy and reliable smart-lock system. This specific case is based on the Semester Project in Trustworthy Systems, where the different components has been modelled.

II. System Description

The elements within the system consist of the following entities and devices:
The ESP32 microcontroller which uses Wi-Fi radio technology and the MQTT communication protocol. A server with an integrated MQTT Broker which handles the communication between the ESP32 and the user, through the smart-home application interface, which allows the user to log in and control the state of the lock. The overall system functionalities, included controlling the lock, and requiring authentication to do so. Furthermore, the smart-lock has to unlock the door within 10 seconds, otherwise it should cancel it's operation.
The embedded systems utilized in the project, requires a proper system modeling in analyzing and verifying different parts of the system.

III. Requirements

The system requirements are derived from both mandatory requirements stated by the project description, and properties identified by analysing the case.

Functional Requirements:

- #F1 The user must be able to open the lock from a smartphone
- #F2 The user must be able to close the lock from a smartphone
- #F3 The user must be able to see the state of the lock on a smartphone
- #F4 The user must be authenticated before they can control the lock
- #F5 If the smart lock loses connection with the internet, it must enter into a recoverable state until the internet is active again

Non-Functional Requirements:

- #NF1 Opening the door should open the door within 10s or not open at all
- #NF2 There must be a minimalist interface for the user to interact to open the lock with little obfuscation of the button
- #NF3 There must be multiple authorization levels, a administrator and normal user. Where the administrator can add locks to the system and the normal user can only use them.

These requirements are created to secure the right functionality is created in the system, this behaviour should be tested to uphold the requirements. Here UPPAAL is used to test if the requirements are fulfilled and the system functions as expected. The Verification tool in UPPAAL is optimal to analyse to verify if there exist a path to a specific functionality or avoiding creating any form of flaw, which could be a deadlock which can block some of the required functionality in the system.

IV. Modelling with UPPAAL

For modelling the different entities and devices, the system was split into different parts within UPPAAL. Some elements remained as a whole while others were made into multiple smaller process making the modeling more manageable when running the simulation and for verification. Within this section, are the ESP32, SmartLock, Server, User and Authenticator.

A. ESP32

The ESP32 device will be listening for commands on the MQTT topic. It will only listen for ”lock” or ”unlock” commands, and other commands will be ignored by the system. When it receives a command to lock or unlock the smart-lock, it will start by checking the lock's current state to see if the command that was sent is valid in the context of the smart-locks current state. For example, if the user sends a command to lock the smart-lock while it already is locked, or if they send a command to unlock the smart-lock while it already is unlocked, then the command will be ignored. These checks are present for preventing contradicting logic.

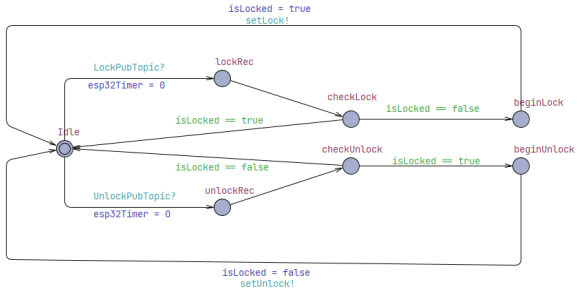


Fig. 1. ESP32 Timed Automata

B. SmartLock

The smart-lock awaits the ESP32 to send commands to either lock or unlock. It will then execute the action by moving the lock mechanism. As mentioned above, if the command attempts reapply its current state (locked/unlocked), it will be ignored.

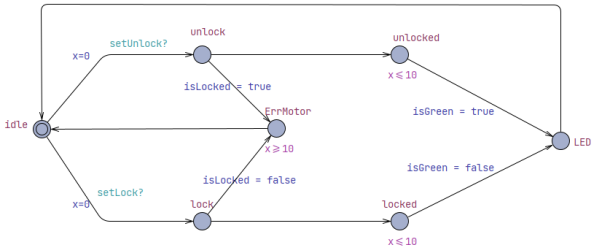


Fig. 2. Smart Lock Timed Automata

C. Authenticator

This automata awaits authentication request from the server, where currently it randomly succeeds or fails, from this it notifies the server that the process is complete by signaling that the process has concluded.

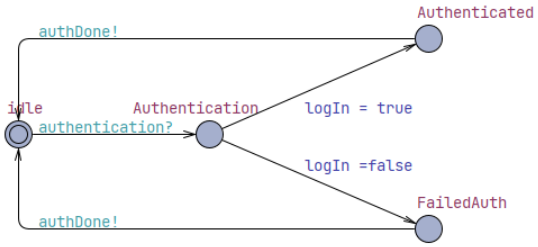


Fig. 3. Authenticator Timed Automata

D. Server

The server requires that the user is authenticated in order to send commands to the ESP32. So firstly they will have to login, which is handled by the authenticator, if successful it allow for sending commands to either lock or unlock the smart lock device.

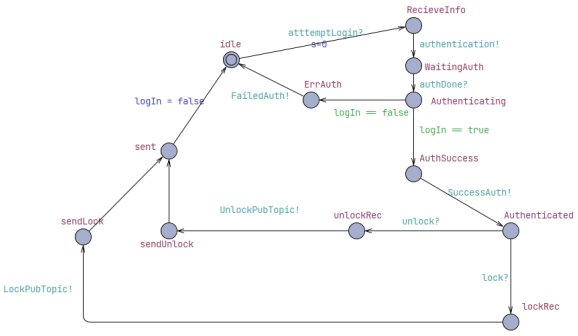


Fig. 4. Server Timed Automata

E. User

The user will have to first log in before they can press any button a button for unlocking or locking the smart lock. First it will send out a signal of a attempted login, if failed, the user will be sent back to the idle and attempt to log in again. On a successful attempt, the user will moved into the pressButton location. They will only be able to unlock or lock according to the smartlock's current state. When the user moves into either location toUnlock or toLock, they will publish payload to the server's MQTT broker.

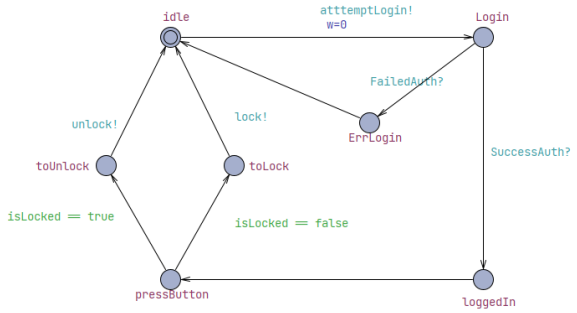


Fig. 5. User Timed Automata

V. Verification With UPPAAL

For verifying the requirements listed in III, multiple queries were made and created in UPPAAL which will check whether the queries are valid or false.

TABLE I
VERIFICATION OVER THE TIMED AUTOMATAS

Purpose	Query
1. There exists a path where lock time is less than 10	E<> Smartlock.x $\leq$ 10
2. Log in will enter failled authenticaiton	E<> logIn imply Server.ErrAuth
3. ESP32 initiates unlock, the smart lock will be unlocked	E<> ESP32.beginUnlock imply Smartlock.unlocked
4. ESP32 initiates lock, the smart lock will be locked	E<> ESP32.beginUnlock imply Smartlock.unlocked
5. There should be at least one path user can press button	E<> User.pressButton
6. When the user press button they are authenticated	A[] User.pressButton imply Server.Authenticated
7. There should be no deadlocks in the system	A[] not deadlock
8. There should at least be one path to smart lock unlocked	E<> Smartlock.unlocked
9. There should at least be one path to smart lock locked	E<> Smarlock.locked
10. The user can not press button while logged out	E<> not (User.pressButton && logIn == false)

The first query will simply check if there was will ever be path where the smart lock's will be less or equal with 10.
Second query will check whether there exists a path where login will fail.
Third query will check if the ESP32 starts unlock it will reach the Smart Lock unlocked location.
Fourth query will check if the ESP32 starts lock it will reach the Smart Lock locked location.
Fifth query will check whether actually exists any path where the user can press the button.
Sixth query for all paths, the user must be authenticated before they can press a button.
Seventh query, the system should have any deadlock within its mechanisms as all entities and devices within the system must be able to recover from any errors or faults. Ensuring the system's reliability and trustworthiness.
Eighth query checks whether there is actually a point where the smart lock is unlocked.
Ninth query checks whether there is actually a point where the smart lock is locked.
The tenth query checks whether if the user can still press a button while they are logged out.

VI. Discussion

The use of the UPPAAL model checking software was a learning process to understand the different aspects and possibilities of developing software that could be verified, hence be deemed reliable. This tool turned out to be powerful to use for simulation and verification of different system. Using this tool for the semester project, made it possible to see different pitfalls in the architecture of the code, with the two simulators and the verification tool, which led to saved time in the developing process of the smart lock system. This was also used to visualise the different components, and how they were interacting with each other.

In terms of modelling the system, some concessions were made in order to capture the most essential logic of the system. Given this, the focus was on modelling the ESP32 and smart-lock, with a lesser importance on modelling the server, as most requirements involved the ESP32 and smart-lock. Even with the actual back-end/server part of the implemented solution having multiple other parts (MQTT, Home Assistant) it was modelled as a single sub-system, as it provided enough of an abstraction to simulate the interaction of users. This was however just the way our system was modelled given it's abstraction. Normally, the real-world implementation of the server would contain more logic than just for authenticating users and handle their lock/unlock requests. Most likely would most of the server logic be separated, or at least run on different threads. So containing all server logic in a single automata would also be an unrealistic abstraction. Therefore it might have been a better idea to name our server automata "Lock Server" or something similar. This would then allow for future additions of other server automatas such as "MQTT Server", "Home Assistant Server" and so on.
As of now, the current model only verifies one single timing constraint within the 'Smart Lock', but checking for more timings under adverse or normal conditions (e.g, network delays or failure) would be for the better.
Furthermore, additional comprehensive scenario testing could

be added into the models for testing beyond normal operations such as security breaches or hardware failures. These aspects could further validate the system's reliability and discover insight into different aspects of the model mirroring the semester project.

For working with UPPAAL and timed automatas, there were certain elements which sowed confusion in terms of validation. Most of all, was the difference of 'imply' within the context of 'Software Analysis and Verifcation' and within 'Discrete Mathematics'. Where in timed automata's, the imply is concerned with state changes over time or sequences of events while in discrete mathematics its a static relationship between truth values of propositions with no consideration for the temporal aspect. For example, the given query.

```
'E<> User.pressButton imply logIn == false'
```

The query would be satisfied within UPPAAL which went against the group's expectations as it was assumed it would be unsatisfied as the user should never be in 'pressButton' location while not being logged in. Therefore, for gaining further understanding, it was necessary to break the query into its basic forms.

```
'E<> User.pressButton': There exists
a path where the User reaches the
pressButton location eventually.
logIn == false: The logIn variable
is false.
```

The reason for the validation being satisfied is due to there being cases where the User does not reach the 'pressButton' location, making the first statement false and in turn will make the implication true regardless of the 'logIn' value. Therefore, for remedying the issue. The query was modified by utilizing a && and NOT operator for ensuring the 'pressButton' is never reached if the user is not logged in.

```
E<> not (User.pressButton &&
logIn == false)
```

The opportunity gave the group members further understanding of the nuances within verification in timed automatas.

VII. Conclusion

The analysis and verification of the smart lock IoT system demonstrated a robust and efficient approach towards smart-home trustworthiness. By leveraging UPPAAL, the project successfully verified both functional and non-functional requirements, ensuring the system's reliability and trustworthiness. The ESP32 microcontroller and server based MQTT communication protocol and authentication were modeled cohesively for responding correctly to the user's commands. Wherein, UPPAAL facilitated identification of potential pitfalls within the system during development, consequently optimizing the system architecture. Furthermore, the verification process highlights the importance of modeling and simulation for developing secure and dependable systems.

Appendix

Contributions

- Abstract
  - Marc Bertelsen
- Introduction
  - Simon Platz Beck-Nielsen
  - Daniel Bahrami
- System Description
  - Daniel Bahrami
  - Kasim Emre Sahin
- Requirements
  - Kasim Emre Sahin
  - Simon Platz Beck-Nielsen
  - Daniel Bahrami
  - Marc Bertelsen
- Modelling with UPPAAL
  - Kasim Emre Sahin
  - Simon Platz Beck-Nielsen
  - Daniel Bahrami
  - Marc Bertelsen
- Verification with UPPAAL
  - Kasim Emre Sahin
  - Simon Platz Beck-Nielsen
  - Marc Bertelsen
- Discussion
  - Simon Platz Beck-Nielsen
  - Marc Bertelsen
  - Daniel Bahrami
  - Kasim Emre Sahin
- Conclusion
  - Kasim Emre Sahin

References

[1] "Eclipse Mosquitto," Jan. 2018, [Online; accessed 31. May 2024]. [Online]. Available: https://mosquitto.org

[2] E. Systems. [Online]. Available: https://www.espressif.com/en/products/socs/esp32

[3] "Home — uppaal," https://uppaal.org/, 2024, accessed: 2024-06-03.

Software System Analysis and Verification Portfolio Report

Marc Bertelsen
Technical Faculty, Software Engineering
University of Southern Denmark
Odense, Denmark
berte20@student.sdu.dk

***Abstract*—This portfolio report works on adding new functionality to the system modelling the semester project from the group report. This includes formulating new requirements, extending and updating the UPPAAL model, and creating verification based on the requirements.**

I. Introduction

One notably overlooked part of the semester project smart-lock system, was the ability for the device to reconnect to the wireless connection, if lost during operation. This was discussed in the later stages of the project but was not implemented due to time constraints. So, this individual part will attempt to introduce this by extending the modelled system by modelling the Wi-Fi connection for the ESP32.

II. Description

In order to receive commands, the device first needs to establish a connection to the Wi-Fi, which the server is communicating through. This connection procedure will also be needed if at any time the connection is lost. Since the communicating is done via an MQTT broker, it follows that the device must also be able to connect to this.

III. Requirements

The requirement is already formulated as **F5** from the group report, but additional requirement are also added.

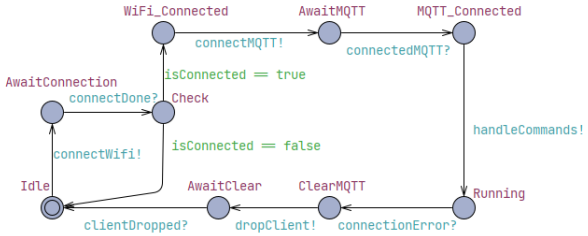
- **F5**: If the smart lock loses connection with the internet, it must enter into a recoverable state until the internet is active again.
- **F6**: The device must ba able to connect to WiFi.
- **F7**: The lock must be able to automatically connect to the MQTT broker.
- **F8**: The lock must be able to reconnect to the MQTT broker, if connection is lost during operation.

IV. System Modelling

A. ESP32

The ESP32 device, which was modelled in a single automata, has been split into different self-contained parts. This first part is the background logic responsible for initializing the device and reestablishing connections if they are lost. This automata will firstly signal the ESP32Wifi on the **connectWifi** channel, here it await a signal back on **connectDone**. If the *isConnected* flag is true, it will proceed to connect to the MQTT broker, else it will retry the connection. Connecting to the MQTT broker, it sends a signal over **connectMQTT** and awaits on **connectedMQTT**. From here it signals the ESP32Hanlder on the **handleCommnds** channel, giving it the green light to run. This automata will only resume if it receives a signal on **connectionError**, which will make it drop the MQTT client and attempt to reconnect to both Wi-Fi and the MQTT broker.

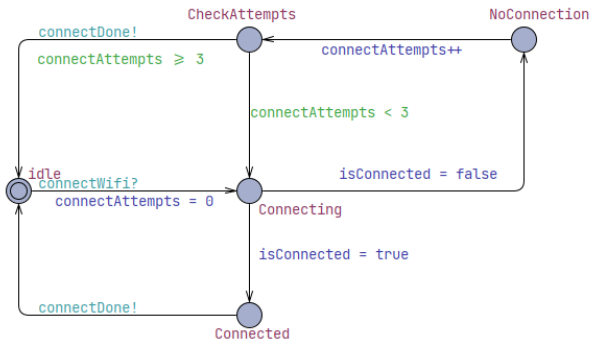
Fig. 1. ESP32 Timed Automata



B. ESP32 WiFi

Here the automata handle the connection to the Wi-Fi. It waits for the ESP32 to signal a connection attempt on the **connectWifi** channel. If the connection fails it will retry up to 3 times. After the 3 attempts or a successful attempt, it will signal back to ESP32 on **connectDone**.

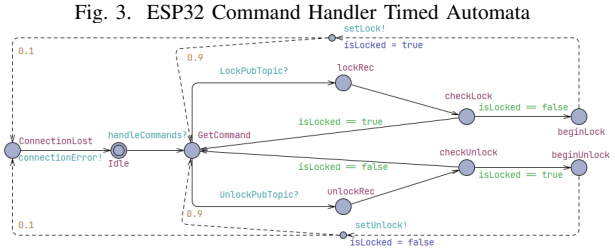
Fig. 2. ESP32 WiFi Connector Timed Automata



C. ESP32 Handler

This is most of the logic from the group version of the ESP32. Here is was been modified to await the Wi-Fi and MQTT connections before being able to get and execute commands. In the automata is also modelled the chance of losing connection, which will trigger the reconnection process.

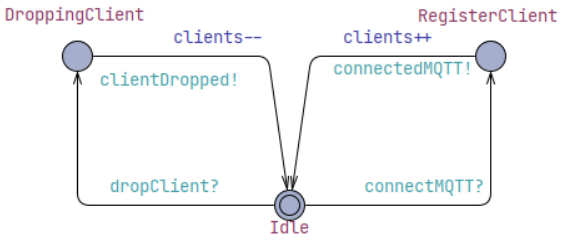
Fig. 3. ESP32 Command Handler Timed Automata



D. MQTT

A very simple representation of the MQTT broker is added with only the connection process modelled. When receiving a signal on **connectMQTT**, it will increment its clients count and signal back on **connectedMQTT**. Similarly, when receiving on **dropClient**, it will decrement the clients count and signal back on **clientDropped**.

Fig. 4. MQTT Broker Timed Automata



V. Verification

Verifying the requirements, from section III, within the model, was done in UPPAAL. Also, to verify that the changes did not break other parts of the system, the deadlock verification is also included. Verifying **F5** requires checking that it is possible for the system to enter the connection procedure is connection is lost. The same is true for **F8**, as it requires the same procedure. For **F6**, there needs to be checked that it is possible to establish connection, which has been accomplished if the *isConnected* flag is set true. Lastly, verifying **F7** will require that the amount of clients registered with the broker is not 0, when listening for commands. Formulating the requirements into UPPAAL verification:

Purpose	**Query**
No deadlock	A[] not deadlock
F6	E<> isConnected
F5, **F8**	E<> !isConnected imply ESP32Wifi.Connecting
F7	A[] ESP32Handler.GetCommand imply MQTT.clients > 0
All	A[] ESP32Handler.GetCommand imply isConnected

VI. Discussion

Working with extending the system to model the Wi-Fi and MQTT connection, required modelling the ESP32 as separate parts, which could have been done earlier. This would allow for separating different logic into their own self-contained automata. During the process of making the model a few validations failed, this helped in making sure the correctness of the extended work, as it meant the modelling was either incorrect or incomplete. An example of this was using the **connectionError?** for both the MQTT broker to decrement its client count, and in the ESP32 to reestablish connection. This was solved by adding an intermediate step after **connectionError**, where the MQTT client is dropped. Some concessions were made when modelling the MQTT broker and connection to it. Realistically it would handle the loss of a client completely by itself, however, the model must signal since it need to be aware that the connection is lost. Furthermore, many of the functionalities of the broker (subscribe, publish, etc.) are not modelled, as it provided little to no benefit in satisfying the requirements.

VII. Conclusion

Using UPPAAL to model a missing requirement, and a few new ones, from the semester project, allowed for verifying the model in terms of capturing the necessary logic required. This was done by changing some parts of the existing model to fit the new logic and extending it with new automata. To summarize, the additions to the UPPAAL model were able to fulfill the missing and the new requirements.

Assignment 1 for Software System Analysis and Verification - Group 10

Simon Platz Beck-Nielsen*, Kasim Emre Sahin*, Daniel Bahrami*, Marc Bertelsen*,
University of Southern Denmark, SDU Software Engineering, Odense, Denmark
Email: * {sibec19,kasah19,dabah20,berte20}@student.sdu.dk

***Index Terms*—Software, Analysis, Verificaiton, UPPAAL, State Modelling**

I. Questions

1) Do you find UPPAAL to be a useful tool? Yes or no, then why?
2) Do the case study and the user experience help in understanding the use/the strength of the model checker?
3) In addition to the specified requirements given in the article, what other properties do/can you identify and verify?
4) Are there any failed properties you found during verification, if yes, what are they? What are the causes for such failures? How can they be fixed? e.g. Do you have to refine or modify your system models and/or requirements? Can you provide any counter-examples of the failed cases?
5) In addition to the suggested points above, you can, of course, also explore and identify something else or something additional during the experiment and share that with your classmates on your presentation day. Indeed, this is strongly encouraged.

II. Answers

A. UPPAAL and Gear Controller

For modelling, validation and verification of real time systems, UPPAAL serves as a valuable utility in grasping the bounds between and interaction of several systems with potential pitfalls which may occur during simulation. For example, the group unexpectedly discovered an error occurred with gear controller system when running the concrete simulator, a ”Deadlock” appeared halting the simulator from moving further. Correspondingly, each group member could see the points of failure within the system, which at first caused confusion whether there was a mistake in the group's implementation of the system within UPPAAL. These points being the errors within the Gear Controller computational tree logic which has has the following locations with no edges leading to other locations: COpenError, GneuError, GSetError and CCloseTimer. Yet, during the presentation on April 2nd between the differing groups, we gained a better understanding of the system, as the ”Deadlock” was intentionally designed.

The Formal Design and Analysis of a Gear Controller report also describes the system using the timer GCTimer as a response time from the components to detect errors. When failures are not signaled and depending on the error, the system attempts to recover or terminate in a pre-specified location that points out the unrecoverable error. Meanwhile, recoverable errors are detected within the CheckTorque and CheckSyncSpeed.

Therefore the following verification within the system will not pass:
A[] not deadlock
A[] Engine.Torque imply Clutch.Closed

However, the requirement specifications within the report has been verified and does indeed work.

B. Experiences and Discussion

While UPPAAL is a useful tool for modeling, simulating and verifying real-time systems, it has a steep learning curve. It is not easy to get started using the tool, due to the fact that the tool consist of multiple parts. The first thing to understand is the description language. This is the modeling or design language used for describing the system behavior. It is then also necessary to comprehend the simulator and model-checker. Once these parts of UPPAAL have been inferred, it is a very powerful tool for modeling, simulating and verifying systems. The difficulty is also dependent on the size of the systems, as bigger and more complex systems always will be more challenging to model and simulate.

There were some issues in transcribing the computational tree logic detailed in the report and onto UPPAAL. One of which was the different syntax utilized within the figures such as ”:=” and ”c:(LocationName)” which caused some difficulties. Only later did the group learn that the meaning of the ”c” meant committed and needed to be checked within the locations, otherwise, the group would encounter concurrency problems with the system. On the former, learning the ”:=” was a older expression of the ”=” operator which still worked within UPPAAL.

Furthermore, the group and the others on the presentation day conversed over the difficulties of within UPPAAL, where the faults occurred within the system, there was a insufficient stack trace of possible errors. More insidious were the semantically errors, where certain locations were named differently and each member had to type manually each time. Therefore, one had to ensure the same exact letters were used for each variable. This was problematic within a group

setting, as different members had chosen different names of the locations, requests and receivers for their individual system.

For our presentation on the 2nd April, we were advised by one group in having a better transition from the report and into the systems within UPPAAL. Explaining the different systems, variable declarations, simulators and verification, yet forgoing a explanation of the Gear Controller System. Therefore, next time involving parts of the intended design goals of the report would be a priority for a possible next presentation.

Keeping track of any large and complex system, can become increasingly difficult to keep an overview of. As the list of parameters, states, and their interactions grow, so does the difficulty in checking and debugging such a system.

One of the talking points, was UPPAAL's lack of scalability when there is a relative high number of simulations to run, it would be hard to keep track of the different variables and states in the system, and navigate to the right problems in the system in order to understand what is happening.

Even with UPPAAL's success in being utilized within the development of industrial real time systems, there still is a persisting State-Explosion problem which prevents the tool from providing fully automatic verification of arbitrarily large and complex systems. [1]

Originally, when the group discovered the deadlock within the system, we wondered whether there was a mistake within the implementation and therefore removed the corresponding locations within the computational tree logic. Then on the day of presentation, the system with deadlock was introduced for the group where the deadlock points of the system were described, before going onwards to the fixed version. Yet, at the end of the presentation some of the other groups commentated the deadlock itself was possibly a intended feature and not a fault with the system itself.

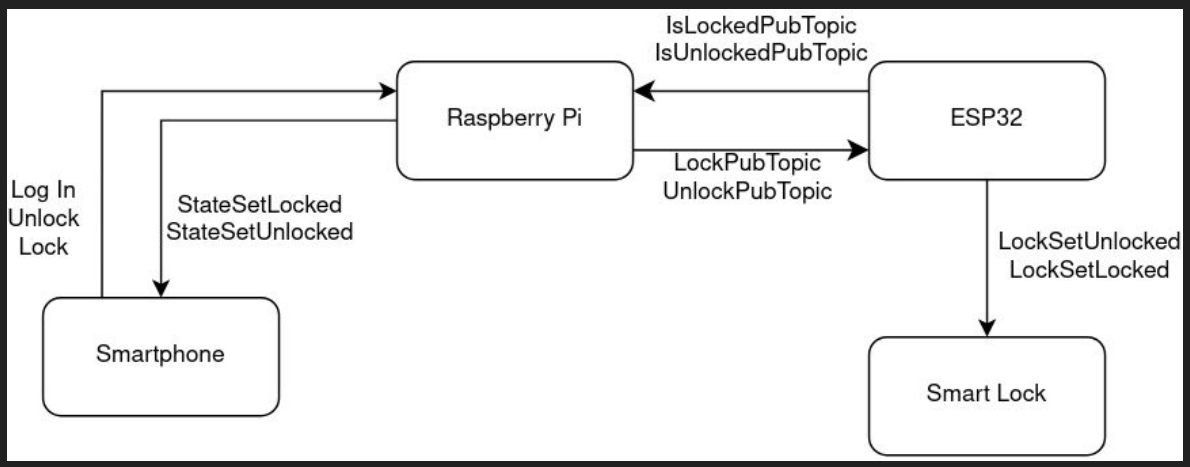
REFERENCES

[1] H. Jensen, K. Larsen, and A. Skou, "Scaling up uppaal," 01 2000, pp. 641–678.

Software System Analysis and Verification, Student Activity 2

Kasim Sahin, Daniel Bahrami, Marc Bertelsen, Simon Platz Beck-Nielsen

Architecture



Requirements

Functional:

- Users must be able to access the lock wirelessly (from a smartphone)
  - Must be able to login
  - Must be able to see the state of the lock
  - Must be able to engage/disengage the lock (open/close lock)

Non-Functional:

- Performance
  - The locking/unlocking process should maximum take 10 seconds
- Reliability
  - Faults in the lock should be detected
- Security
  - The system should verify the authenticity of users
- Availability
  - The system should have no downtime

Automata

- ESP32
- User
- Server
- Smartlock

Verifications

- E<> Smartlock.x <= 10
- E<> Smartlock.x => 10
- E<> logIn imply Server.ErrAuth
- E<> ESP32.beginUnlock imply Smartlock.unlocked
- A[] User.pressButton imply Server.Authenticated
- A[] not deadlock

Expected deliverables - Future Work

- Expand on error handling
- Cora and energy efficiency
- More model verification
- Include timeout for too long requests
- Heartbeat also needs to be developed

Contribution

Everybody has contributed equally to the student activity 2