Universität Siegen

Embedded Control Lab 2: Discrete Systems (Finite State Machine) Modelling.

by:
Group 8:
Davin Edison
Hatim
Ishita Singh
Oussema Kassebi

Embedded Control Laboratory

Davin Edison, Ishita Singh, Oussema Kasebi, Hatim

December 2023

Contents

List of Figures

1 Introduction

Discrete-time and continuous-time systems represent two fundamental paradigms in the modeling and analysis of dynamic processes. These conceptual frameworks provide distinct ways of understanding and characterizing the behavior of systems with respect to time.

In continuous-time modeling, the passage of time is considered as seamless and uninterrupted flow, described through differential equations capturing the system's smooth evolution. On the other hand, discrete-time modeling quantized time into distinct intervals, offering a snapshot-like view of a system's state at specific time points, often articulated through difference equations.Both approach find applications across various domains, influencing fields such as control systems, signal processing, and simulation.

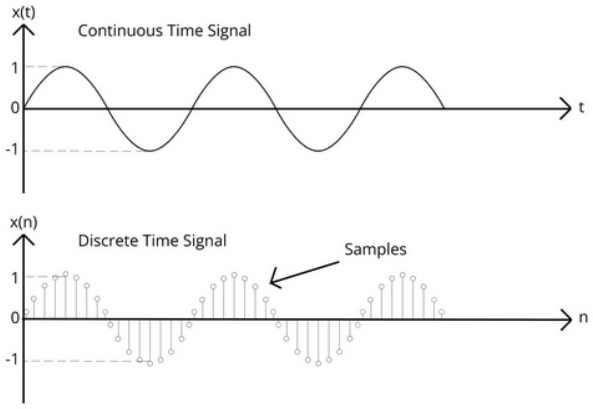

Figure 1.1: Continuous-Time Signal vs Discrete-Time Signal

2 Discrete-Time modeling

Discrete-time modeling is a powerful approach in systems design, allowing the designer to represent and analyze systems that evolve over distinct, separated intervals of time. This method is particularly valuable in scenarios where continuous modeling might be overly complex or not applicable. There are a lot of ways to do discrete-time modeling, but one of the most renown is state machines concept. These state machines, or sometimes called automata, serve as abstract representations of systems that transition between various states, making them instrumental in modeling the behavior of a multitude of applications.

By exploring state machines modeling further, tool such as UPPAAL become indispensable. UPPAAL offers a comprehensive platform for simulating and verifying the behavior of state machines, ensuring their robustness and accuracy.

3 State Machines

3.1 Introduction to State Machines

State Machines or also known as a finite state machine (FSM), and often known as automata, have roots that trace back to the early theories of computation and logic. It serve as indispensable tools in computer science and engineering, offering a structured and intuitive means to conceptualize and articulate the dynamic behavior of systems. At the core, state machines represent a formal methodology for modeling the various states a system can inhabit and the transitions between these states in responses to events or conditions. This powerful abstraction find widespread application in software development, control systems, and protocol design. By breaking down the complexity of a system into distinct states and the transitions between them, state machines provide a clear and organized framework for understanding and designing the behavior of dynamic systems.

State machines are widely used in various applications, including software engineering, control systems, and protocol design. In software development, state machines can be used to model the behavior of a program to design the control flow of an algorithm. They provide a clear and organized way to represent complex systems and their behavior. In fact, the inception of state machines can be linked to mathematicians and logicians like Alan Turing, who conceptualized the Turing machine as a fundamental model of computation and John von Neumann, who pioneered the early architecture of electronic computers.

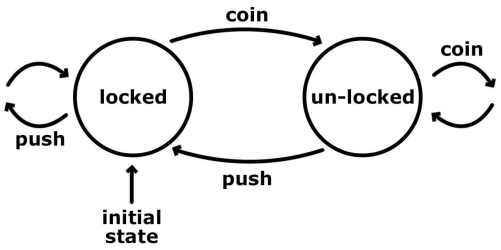

Figure 3.1: Simple State Machine Example

Historically, one of the first applications of state machines was in telephone switching systems, where they were used to manage and direct calls based on user inputs. Similarly, early computer algorithms, especially in text processing and searching, employed finite automata to recognize patterns and sequences.

At its core, a state machine is an abstract representation of a system, capturing its various possible conditions or **states** and the rules that dictate how it moves between these states. Over time, state machines have evolved from theoretical constructs to practical tools, with applications ranging from digital circuit design to software development. They serve as a bridge between the logical design and tangible implementation, allowing for systematic modeling of complex systems in a structured and predictable manner.

3.2 State Machine classification

State machines can be classified into several types based on their characteristics and behavior.

3.2.1 Finite State Machine (FSM)

A Finite State Machine (FSM) is a mathematical model used to design and represent the behavior of a system that can exist in a finite number of states. The systems transitions between these states in response to external inputs or events. FSM can be in exactly one of a finite number of states at any given time.

There are 2 types of Finite State Machine, which are:

1. **Deterministic Finite Automaton (DFA)**
   In Deterministic Finite Automaton (DFA), each state has a specific transition for every possible input. It follows a fixed set of rules and produces a unique output for a given input

2. **Nondeterministic Finite Automaton (NFA)**
   In an Nondeterministic Finite Automaton (NFA), a state can have multiple transitions for the same input. The system is not strictly deterministic, and multiple paths may exist for a given input.

3.2.2 Infinite State Machine(ISM)

An Infinite State Machine (ISM) is a theoretical model where the system can potentially exist in an infinite number of states. This type of state machine is often used to represent systems with an unbounded or continuously variable state space.

There are 2 types of Infinite State Machine which are:

1. **Mealy Machine**
   In a Mealy Machine, the output depends on both the current state and the input. Transitions may also produce immediate outputs.

2. **Moore Machine**
   In a Moore Machine, the output depends only on the current state. Outputs are associated with states rather than transitions.

3.2.3 Hierarchical State Machine

A Hierarchical State Machine is an extension of a traditional state machine where states can be organized in a hierarchical structure. This allows for a more modular and organized representation of complex systems. The main point of Hierarchical State Machine is the states can contain sub-states.

3.2.4 Extended State Machine (Harel Statecharts)

An Extended State Machine, specifically Harel Statecharts is an advanced form of state machines that introduces additional features such as concurrent states, history states and orthogonal regions. These features enhance the expressiveness of the model for handling complex systems.

3.2.5 Timed State Machine

A Timed State Machine is a state machine that incorporates the concept of time into its transitions. Transitions may occur after specified time intervals, making this type of state machine suitable for modeling real-time system.

3.2.6 Asynchronous State Machine

An Asynchronous State Machine is a state machine where transitions can be triggered by asynchronous events. In contrast to systems with regular clock-driven updates, asynchronous state machines offer flexibility in handling events that occur at irregular intervals.

3.2.7 Markov Model (Continuous State Machine)

A Markov Model, often considered continuous state machine, is a stochastic model used to describe systems where state changes are continuous rather than discrete. It is characterized by a state space, transition probabilities, and continuous-time dynamics.

3.3 Basic Components of A Finite State Machine

The basic components of a state machine include:

1. **States**
   States is the heart of a State Machines, it represent the different situations or conditions that a system can be in at any given time. Each state captures a specific configuration or behavior of the system. The system can only be in one state at any given time, and as events occurs or conditions are met, it moves from one state to another.

2. **Transition**
   Transitions define the conditions or events that cause a system to move from one state to another. Transitions can be seen as the pathways that enable a system to move from its current state to another state. Every transition is associated with specific criteria that must be satisfied for the shift to occur.

3. **Guard**
   A guard is a subset of possible values in on input ports that determines whether a transition is permissible. For the transition to be taken, the guard's condition must be met.

4. **Actions**
   Once a transition is taken, certain operations or set actions are executed.

5. **Initial State**
   Initial State is the starting point of the state machine, indicating which state the system is in at the beginning.

6. **Final State**
   Final State is the end point of the state machine, indicating the completion of a process or task.

7. **Variables and Clocks**
   Variables play a pivotal role in state machines, representing parameters that can change over time and influence the system;'s behavior. A common variable in many state machines is the clock, which measures the passage of time. Clocks can be used in guards to determine when transitions should occur based on time elapsed.

Which the example can be seen in the figure below.

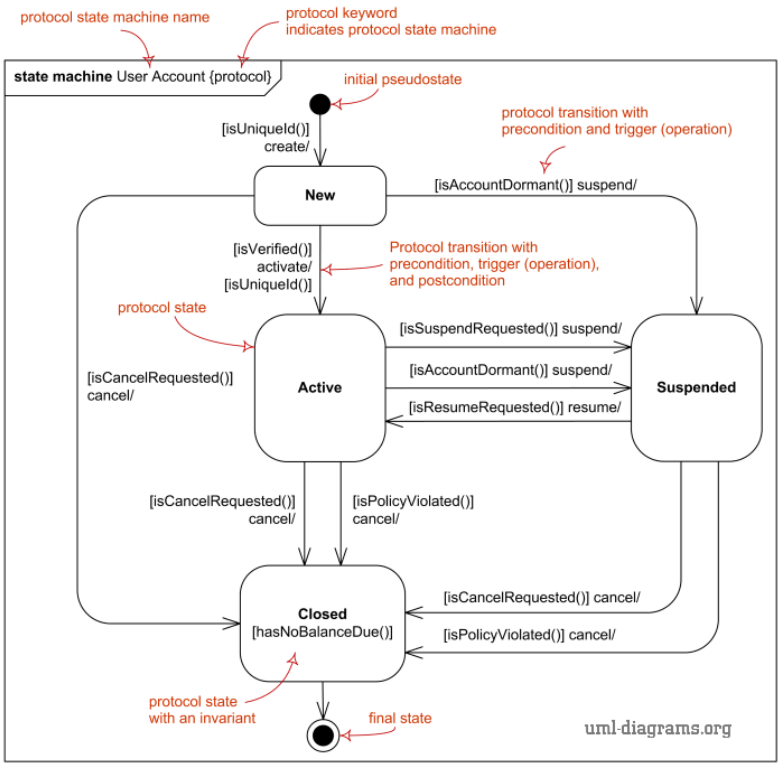

Figure 3.2: Example of State Machine

4 UPPAAL

UPPAAL stands as an integrated tool environment specifically designed for the modeling, simulation, and verification of real-time systems delineated by networks of timed automata. Originating from joint efforts between Uppsala University in Sweeden and Aalborg University in Denmark, UPPAAL incorporates a rich set of features. Its core is built upon formal methods, leveraging timed automata theory to ensure systematic and precise analysis. The environment offers an intricate graphical interface that allows users to construct and visualize state machines. Beyond basic timed automata, UPPAAL supports advanced functionalities such as stochastic hybrid systems and cost-optimal modeling that fits the complexities of real-time system design and verification.

Figure 4.1: UPPAAL Logo

UPPAAL consists of three main parts, which are a description language, a simulator and a model-checker. The description language is a non-deterministic guard command language with data types. It serves as a modeling or design language to describe system behavior as networks of automata extend with clock and data variables.

The simulator is a validation tool which enables examination of possible dynamic executions of a system during early design stages, thus provides an inexpensive mean of fault detection prior to verification by the model-checker which covers exhaustive dynamic behavior of the system. The model-checker can check invariant and reachability properties by exploring the state-space of a system.

The two main design criteria for UPPAAL have been efficiency and ease of usage. The application of on-the-fly searching technique has been crucial to the efficiency of UPPAAL model-checker. Another important key to efficiency is the application of a symbolic technique that reduces verification problems to that of efficient manipulation and solving constraints.

To facilitate modeling and debugging, the UPPAAL model-checker may automatically generate a diagnostic trace that explains why a property is (or is not) satisfied by a system description. The diagnostic traces generated by the model-checker can be loaded automatically to the simulator, which may be used for visualization and investigation of the trace.

4.1 Model Verification in UPPAAL

Timed CTL (TCTL) extends the traditional Computation Tree Logic (CTL) by introduc- ing timing constraints, making it particularly suitable for real-time systems. The primary operators in TCTL are:

Path Quantifiers :

- **A (For all paths) :**
  Represents universal quantification over paths.
- **E (There exists a path) :**
  Denotes existential quantification over paths. Temporal Modalities:
- **G (Globally/Always) :**
  Asserts that a property holds continuously.
- **F (Future) :**
  Indicates that a property will hold at some point in the future.

UPPAAL leverages TCTL to formulate verification queries, providing a tailored set of operators:

- **A[ ] :**
  Represents "on all paths, always." It asserts that a property holds universally on all paths and at all time points. For example, **A[ ]**φ means φ is always true everywhere.
- **A<> :**
  Denotes "on all paths, eventually." It asserts that a property will hold at some point in the future on all paths. For instance, **A<>** φ indicates that φ will become true at some future time on all execution paths.
- **E[ ] :**
  Represents "there exists a path, always." It signifies that there's at least one path where a property continuously holds. **E[ ]**φ means there's a path where φ is always true.
- **E<> :**
  Denotes "there exists a path, eventually." It states that there's at least one path where a property will hold in the future. **E<>** φ asserts there exists a path where φ will eventually become true.
- **p→q :**
  This is an implication operator. It asserts that whenever property p holds, property q must also hold. It's commonly used to represent system guarantees, such as "if condition **p** is met, then outcome **q** will follow."

5 Task 1 - Traffic Lights State Machine Model

5.1 Problem Statement

To learn more about states space machine, the authors chose to model car and pedestrian traffic lights at a city crossroad, where one road runs along the North-South direction and the other along the East-West. Each road has a pedestrian crossing.

The model will include **Car Traffic Lights** which will consist of two traffic lights state machines for cars, one for each direction. Each traffic light should have the standard three states, which are **Red, Yellow**, and **Green**. Synchronization is used to ensure that at any point, both car traffic lights aren't green simultaneously to avoid accident.

It will also include **Pedestrian Traffic Lights** which will consist of two pedestrian lights one for each crossing. Pedestrian Traffic Lights should have two states, which are **Red**, and **Green**. To ensure the safety of the pedestrian, it should be synchronized with car traffic lights, so that when corresponding **Car Traffic Lights** is green, pedestrian light that is perpendicular to the traffic light should be red.

The key chanllenge is to synchronize the four traffic lights to ensure smooth traffic flow and maximum safety. The configuration of the initial states of four lights is one of the important decision for this scenario.

To test the system, UPPAAL will be used because UPPAAL has built in **Simulation and Verification** feature which will help to observe the behavior of the traffic light system. The tests that will be done to make sure that the system is safe to use are as follows:

- There is never a deadlock.
- Car traffic lights are **never** green at the same time.
- When car lights are green, corresponding pedestrian lights should indicate red.
- When pedestiran lights are green, corresponding car lights should inidcate red.
- Both pedestriand and car traffic lights should eventually turn green
- All lights should never be red at the same time.

5.2 State Machine Variables

Several broadcast synchronization channels and clock variable has to be declared. In UPPAAL variable can be declared globally or locally, for this state machine design the variable will be declared globally because the variables are being used in more than one state machine. Following global variables are declared in Declaration text inside the UPPAAL project folder:

1. **Broadcast Channels :**
    - broadcast chan redStateNS
    - broadcast chan yellowStateNS
    - broadcast chan greenStateNS
    - broadcast chan redStateWE
    - broadcast chan yellowStateWE
    - broadcast chan greenStateWE
2. **Clock Variable :**
    - clock elapsedTime

5.3 State Machine Models

State Machine for this scenario is divided into 4 different State Machines, which are **North-South Car Traffic Light (NS_Car)**, **West-East Car Traffic Light (WE_Car)**, **North-South Pedestrian Traffic Light (NS_Pedestrian)**,**West-East Pedestrian Traffic Light (WE_Pedestrian)**.

UPPAAL has a color code for its transition components which are **Select = Yellow**, **Guard = Green**, **Synchronization = Cyan**, **Update = Blue**, and **State Machine Name = Magenta**.

5.3.1 North-South Car Traffic Light

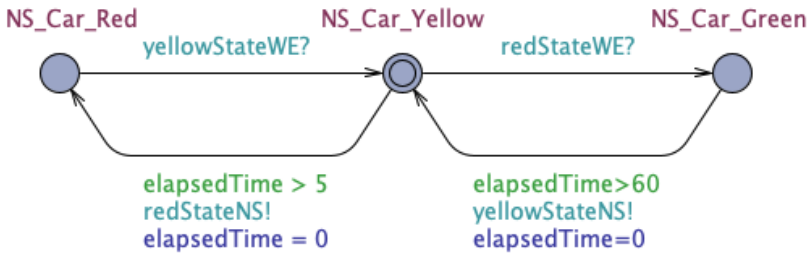

Figure 5.1: North-South Car Traffic Light State Machine Model

For this state machine, the traffic light will turn green at the same time as West-East Car Traffic Light turns red which can be reached by listening to ***redStateWE*** synchronization broadcast channel. Then after in the Green State for 60 seconds, the traffic light will turns Yellow and send a synchronization broadcast to ***yellowStateNS*** channel, then it will turn Red after 5 seconds in Yellow State, in which it will send a synchronization broadcast to ***redStateNS*** channel to indicate that it is turning red. The last transition is from Red State to Yellow State which only possible if it receives an update in synchronization broadcast in ***yellowStateWE*** channel, where it indicates West-East Car Traffic Light is moving to Yellow State.

5.3.2 West-East Car Traffic Light

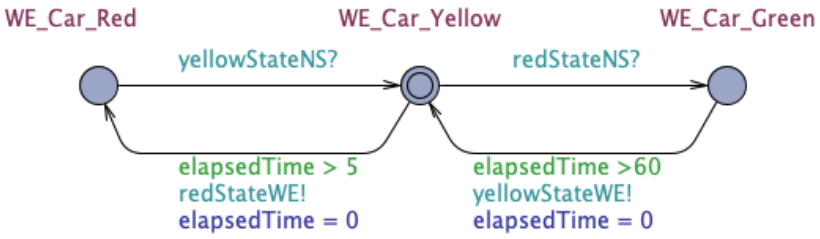

Figure 5.2: West-East Car Traffic Light State Machine Model

For this state machine, the traffic light will turn green at the same time as North-South Car Traffic Light turns red which can be reached by listening to ***redStateNS*** synchronization broadcast channel. Then after in the Green State for 60 seconds, the traffic light will turns yellow and send a synchronization broadcast to ***yellowStateWE*** channel, then it will turn Red after 5 seconds in Yellow State, in which it will send a synchronization broadcast to ***redStateWE*** channel to indicate that it is turning red. The last transition is from Red State to Yellow State which only possible if it receives an update in synchronization broadcast in ***yellowStateNS*** channel, where it indicates West-East Car Traffic Light is moving to Yellow State.

5.3.3 North-South Pedestrian Light

For this state machine, the pedestrian traffic light will turn green at the same time as West-East Car Traffic Light turns red which can be reached by listening to ***redStateWE*** synchronization broadcast channel. Then it will

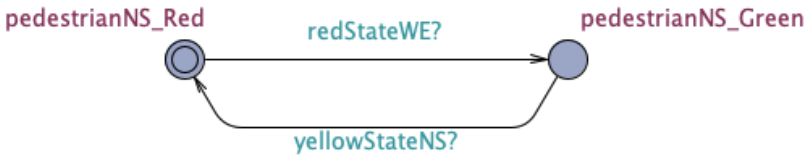

Figure 5.3: North-South Pedestrian Traffic Light State Machine Model

turns red again only if it receives an update in ***yellowStateNS*** synchronization broadcast channel which indicates that North-South Car Traffic Light is in transition to its Yellow State.

5.3.4 North-South Pedestrian Light

Figure 5.4: West-East Pedestrian Traffic Light State Machine Model

For this state machine, the pedestrian traffic light will turn green at the same time as North-South Car Traffic Light turns red which can be reached by listening to ***redStateNS*** synchronization broadcast channel. Then it will turns red again only if it receives an update in ***yellowStateWE*** synchronization broadcast channel which indicates that West-East Car Traffic Light is in transition to its Yello State.

5.4 State Machines Model Verification

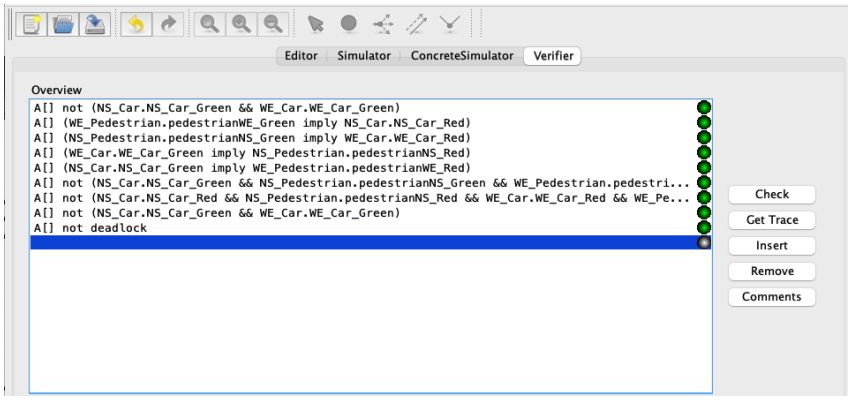

Figure 5.5: Task 1 Verification Passed

To verify all of the criteria has been met, UPPAAL Verifier feature is used. List of all verification syntax is as follows:

- **A[ ] not (NS_Car.NS_Car_Green && WE_Car.WE_Car _Green) :**
  Means that on all path or any state there won't be any North-South Car Traffic Light and West-East Traffic Light turns green at the same time.
- **A[ ] (WE_Pedestrian.pedestrianWE_Green imply NS_Car.NS_Car_Red) :**
  Indicates that when West-East Pedestrian Traffic light is Green, its implying that North-South Car Traffic Light must be Red.
- **A[ ] (NS_Pedestrian.pedestrianNS_Green imply WE_Car.WE_Car_Red) :** Proves that when North-South Pedestrian Traffic light is Green, its implying that West-East Car Traffic Light must be Red.
- **A[ ] (WE_Car.WE_Car_Green imply NS_Pedestrian.pedestrianNS_Red) :**
  Shows that when West-East Car Traffic light is Green, its implying that North-South Pedestrian Traffic Light must be Red.
- **A[ ] (NS_Car.NS_Car_Green imply WE_Pedestrian.pedestrianWE_Red) :**
  Implies that when North-South Car Traffic light is Green, its implying that West-East Pedestrian Traffic Light must be Red.
- **A[ ] not (NS_Car.NS_Car_Green && NS_Pedestrian.pedestrianNS_Green && WE_Pedestrian.pedestrianWE_Green && WE_Car.WE_Car_Green) :**
  Displays that on all path or any state there won't be a condition where all lights is in Green State all together.
- **A[ ] not (NS_Car.NS_Car_Red && NS_Pedestrian.pedestrianNS_Red && WE_Car.WE_Car_Red && WE_Pedestrian.pedestrianWE_Red) :**
  Points out that on all path or any state there won't be a condition where all lights is in Red State all together.
- **A[ ] not deadlock :**
  Signifies that there is no deadlock in any state or in any path of the state machines model.

6 Task 2 - Elevators State Machine Model

6.1 Problem Statement

The challenge of creating a state machines with 2 elevators that will serve 6 floors is to synchronize multiple inputs and map it to be served by 2 separate machines that cannot talk to each other.

The aim for this task is to learn more about synchronization channel of state machines. Previously its only has an input for an output, but for this task, multiple inputs will be received and processed by state machines. Thus will represent the complexity of real-world application of state machines which will exercise the problem solving, and critical thinking of the authors.

To simplify the possibility of 2 elevators that will serves 6 floors system, several constraints has been made, which are:

- Elevator moving up should only accepts up request from floors above its current floor and likewise elevator moving down should only accepts down requests only from the floors below its current floor level.
- Fairness check to ensure that floor requests are handled alternately by the elevators , so that the efficiency and increase the performance of the system.
- To ensure that the Elevators don't stuck in a state, the elevators should not stay for more than 60 seconds in MovingUp or MovingDown states.
- Ground floor should not be able to make a down request and likewise the top floor should not be able to make a up request.
- Once a floor makes a request, it should be served within 60 seconds.

To make sure that the system is working perfectly, Simulation and Verification features of UPPAAL will be used to check the system. Several checks that will be performed to ensure the system is working perfectly are as follows:

- There is never a deadlock
- When a floor in UpRequest/DownRequest state, it eventually returns to Idle state.
- When an elevator is in MovingUp/MovingDown state, it eventually goes to Loading/Unloading state.
- Each elevator eventually services every floor.

6.2 State Machine Templates

6.2.1 Elevator State Machine Template

A template for elevator has been defined to simplify the system even more. Several design parameters that comes with the template are:

1. **States :**
    - **Idle :** The elevator is not in motion and is waiting for a request.
    - **MovingUp :** The elevator is in motion, moving to a floor above.
    - **MovingDown :** The elevator is in motion, moving to a floor below.
    - **Loading/Unloading :** The elevator is stationary, allowing passengers to enter or exit.
2. **Transitions :**
    - From Idle to MovingUp or MovingDown based on the request queue.
    - Transition from MovingUp or MovingDown to Loading/Unloading will happen once the elevator reaches the target floor.
    - Once the Loading/Unloading operation is complete, the state will move back to Idle.
3. **Variables :** Each elevator will have these sets of variables, the variables are as follows:
    - **int currentFloor :** A variable to indicate at which floor the elevetor currently is.
    - **int targetFloor :** A variable to indicate where the floor is heading currently.
    - **int requestQueue :** A variable to store a queue of received floor requests.

6.2.2 Floor State Machine Template

A template for floor has been defined to simplify the system even more. Several design parameters that comes with the template are:

1. **States :**
    - **Idle :** No request has been made from corresponding floor.
    - **UpRequest :** A request to go up has been made.
    - **DownRequest :** A request to go down has been made.
2. **Transitions :**
    - Transition from Idle to UpRequest is made when an upward request is made.
    - Transition from Idle to UpRequest is made when a downward request is made.
    - Floor State Machine state will return to idle after the request has been serviced.
3. **Synchronization Channels :** Each floor will have these sets of Synchronization Channels, the Channels are as follows:
    - **request_up[floorNumber]! :** This channel is used to send an upward request from a floor to elevator 1.
    - **request_down[floorNumber]! :** This channel is used to send an downward request from a floor to elevator 1.
    - **ack[floorNumber]? :** This channel is used to receive acknowledgement from elevator that request has been serviced.

6.3 Elevator State Machine

The authors designed an Elevator State Machine as follows:

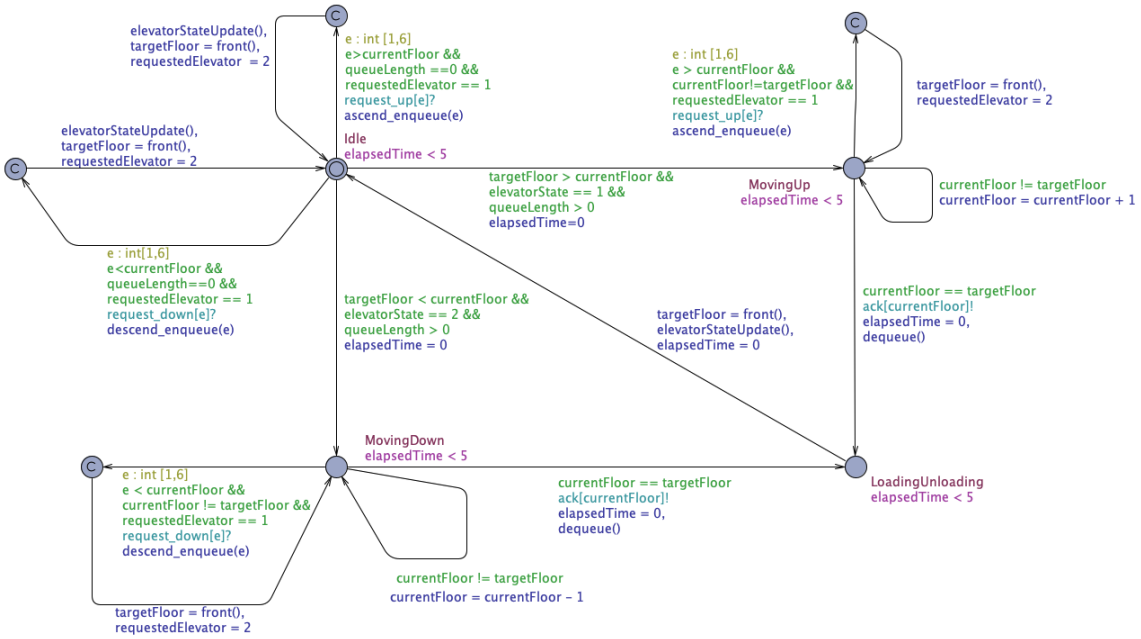

Figure 6.1: Elevator 1 State Machine

The differences between Elevator 1 and Elevator 2 are the guard that being used to assign the new request. In Elevator 1 it will assign the new request to its queue if requestedElevator value is 1 and as for Elevator 2 it will assign the new request to its queue if requestedElevator value is 2 which the differences can be compared between Figure 6.1 and Figure 6.2.

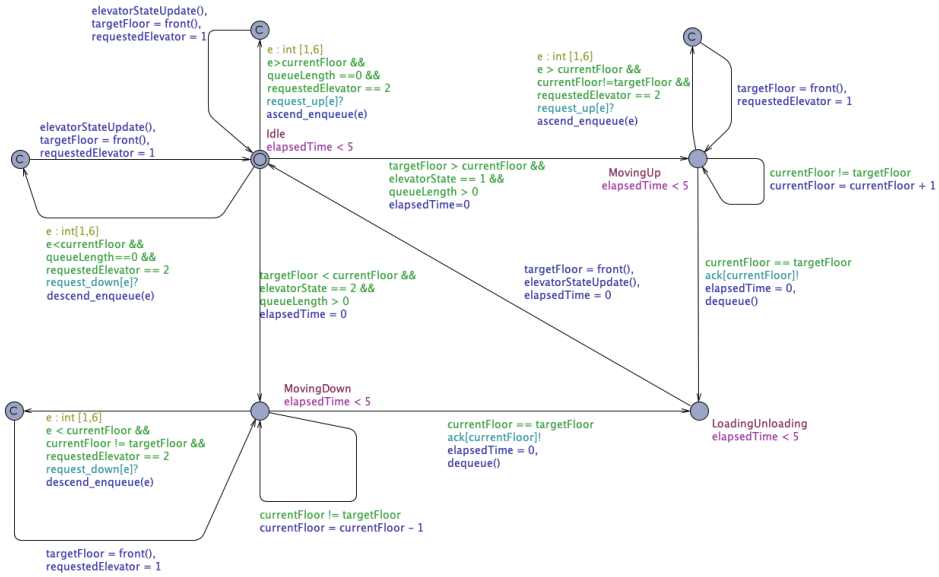

Figure 6.2: Elevator 2 State Machine

As it is mentioned above, the Elevator State Machine has 4 main states, which are:

1. Idle
2. MovingUp
3. MovingDown
4. LoadingUnloading

The transition and the condition can be summarized as follows:

- **Idle –> MovingUp** This transition is made if there is an up request from one of the floor state machines.
- **Idle –> MovingDown** This transition is made if there is an down request from one of the floor state machines.
- **MovingUp/MovingDown –> LoadingUnloading** This transition is made if the elevator has already reached its target floor.
- **LoadingUnloading –> Idle** This transition will always be made after setting up the new target floor if elevator queue is not empty, if its empty, it will set the target floor to 0 (Elevator being Idle) then go to Idle state.

There are several functions that has been written in order for the system is functioning as the constrains. Those functions are:

- **ascend_enqueue()**
  This function is used to record every upRequest from Floor State Machines if the floor number is greater than currentFloor value.
- **descend_enqueue()**
  This function is used to record every downRequest from Floor State Machines if the floor number is less than currentFloor value.
- **dequeue()**
  This function is used delete a floor request once its has been served.
- **front()**
  This function returns the value of the first index in the floor queue. The purpose of this function is to update the target floor value with the value of the first item in queue array.
- **elevatorStateUpdate()**
  The purpose of this function is to determine the state of the elevator, whether it is going up or going down. The elevator has 3 states, which are:

  1. 0 = Idle
  2. 1 = Going Up
  3. 2 = Going Down

  Elevator can only change its state from going up or going down to the other way from idle state. To be able to change direction from idle state, there must be no floor in respective elevator's queue which can be determine by queueLength variable of respective elevator value is equal to 0.

All of the function syntax will be written in the following sub section.

6.3.1 ascend_enqueue() Function Syntax

The function syntax of ascend_enqueue is written in C++ language because UPPAAL uses C++ language, and as follows:

```
void ascend_enqueue(id_t newFloorNumberRequest){
    if (queueLength == 0){
        requestQueue [0] = newFloorNumberRequest;
         queueLength = queueLength + 1;
    }

    else if (queueLength == 6){
        queueLength = 6;
    }

    else{
        inputIndexLocation = 100;
        //Find where the new request should be put inside the array
        findLocationIndex = 0;
        sortIndex = queueLength - 1;
        while (findLocationIndex < queueLength){
            if (inputIndexLocation == 100){
                if (requestQueue[findLocationIndex] > newFloorNumberRequest){
                    inputIndexLocation = findLocationIndex;
                }

                else if (findLocationIndex + 1 == queueLength){
                    inputIndexLocation = findLocationIndex + 1;
                }
            }
            findLocationIndex = findLocationIndex + 1;
        }

        //Sorting previous queue to make space for the new request floor
        while (sortIndex >= inputIndexLocation){
            requestQueue[sortIndex+1] = requestQueue[sortIndex];
            sortIndex = sortIndex - 1;
        }

        if (inputIndexLocation != 100){
            requestQueue[inputIndexLocation] = newFloorNumberRequest;
            queueLength = queueLength + 1;
        }
    }
}
```

6.3.2 descend_enqueue() Function Syntax

The function syntax of descend_enqueue is also written in C++ language which is as follows:

```
void descend_enqueue(id_t newFloorNumberRequest){
    if (queueLength == 0){
        requestQueue[0] = newFloorNumberRequest;
        queueLength = queueLength + 1;
    }

    else if (queueLength == 6){
        queueLength = 6;
    }

    else{
        inputIndexLocation = 100;
        //Find where the new request should be put inside the array
        findLocationIndex = 0;
        sortIndex = queueLength-1;
        while (findLocationIndex < queueLength){
            if (inputIndexLocation == 100){
                if (requestQueue[findLocationIndex] < newFloorNumberRequest){
                    inputIndexLocation = findLocationIndex;
                }
                else if (findLocationIndex + 1 == queueLength){
                    inputIndexLocation = findLocationIndex + 1;
                }
            }
            findLocationIndex = findLocationIndex + 1;
        }

        //Sorting previous queue to make space for the new request floor
        while (sortIndex >= inputIndexLocation){
            requestQueue[sortIndex+1] = requestQueue[sortIndex];
            sortIndex = sortIndex - 1;
        }

        if (inputIndexLocation != 100){
            requestQueue[inputIndexLocation] = newFloorNumberRequest;
            queueLength = queueLength + 1;
        }
    }
}
```

6.3.3 dequeue() Function Syntax

The function syntax of dequeue() is also written in C++ language which is as follows:

```
void dequeue(){
    int dequeueIndex = 0;
    currentFloor = requestQueue[0];
    while (dequeueIndex < queueLength-1){
        requestQueue[dequeueIndex] = requestQueue[dequeueIndex + 1];
        dequeueIndex++;
    }

    if (queueLength > 0){
        requestQueue[queueLength-1] = 0;
        queueLength = queueLength-1;
    }

    else{
        requestQueue[0]=0;
        queueLength = 0;
     }
}
```

6.3.4 front() Function Syntax

The function syntax of front() is also written in C++ language which is as follows:

```
int front(){
   return requestQueue[0];
}
```

6.3.5 elevatorStateUpdate() Function Syntax

The function syntax of elevatorStateUpdate() is also written in C++ language which is as follows:

```
void elevatorStateUpdate(){
    if (queueLength == 0){
        elevatorState = 0;
    }

    else if (queueLength != 0){
        if (requestQueue[0] > currentFloor){
            elevatorState = 1;
        }

        else if (requestQueue[0] < currentFloor){
            elevatorState = 2;
        }
    }
}
```

As it can be seen in Figure 6.1 and Figure 6.2, there are several committed states that have been used. The authors put these committed states so that the transition will be immediately taken if the guard condition is satisfied, in addition to that, those committed states also used to ensure the robustness of the code by ensuring that variables that will be used in the next function has already been set before the next function is executed.

To ensure that the request will be accepted alternately between Elevator 1 and Elevator 2, a guard for each elevator state machine is added which are requestedElevator==1 in Elevator 1 state machine and requestedElevator==2 in Elevator 2 state machine. requestedElevator value will only be updated after a request has been successfully added to the queue by ascend_enqueue or descend_enqueue in Elevator 1 and Elevator 2 state machines, the value of requestedElevator will be set to 2 if Elevator 1 has successfully added a new request to its queue and the value of requestedElevator will be set to 1 if Elevator 2 has successfully added a new request to its queue

6.4 Floor State Machine

The authors designed a Floor State Machine as follows:

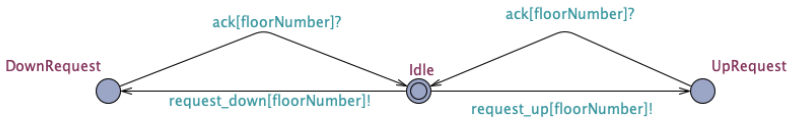

Figure 6.3: Floor State Machine

This state machine can either transition from **Idle** state to **UpRequest** or **DownRequest** state. By transitioning into either **UpRequest** or **DownRequest** it will send a synchronization broadcast through **request_up[floorNumber]** channel and **request_down[floorNumber]** channel.

It will return to **Idle** state from **UpRequest** or **DownRequest** if it has received an update in **ack[floorNumber]** broadcast channel from either elevator 1 and elevator 2.

6.5 Verification of State Machine Designs

In order for a system works properly, there should be not deadlock, and works as it should be. In this task, to ensure that the state machine is working as it should be, the state machine has to ensure several points which are:

1. There is never a deadlock.
2. When a floor is in UpRequest/DownRequest state, it eventually returns to Idle state.
3. When an elevator in MovingUp/MovingDown state, it eventually goes to Loading/Unloading state.
4. Each elevator eventually services every floor.

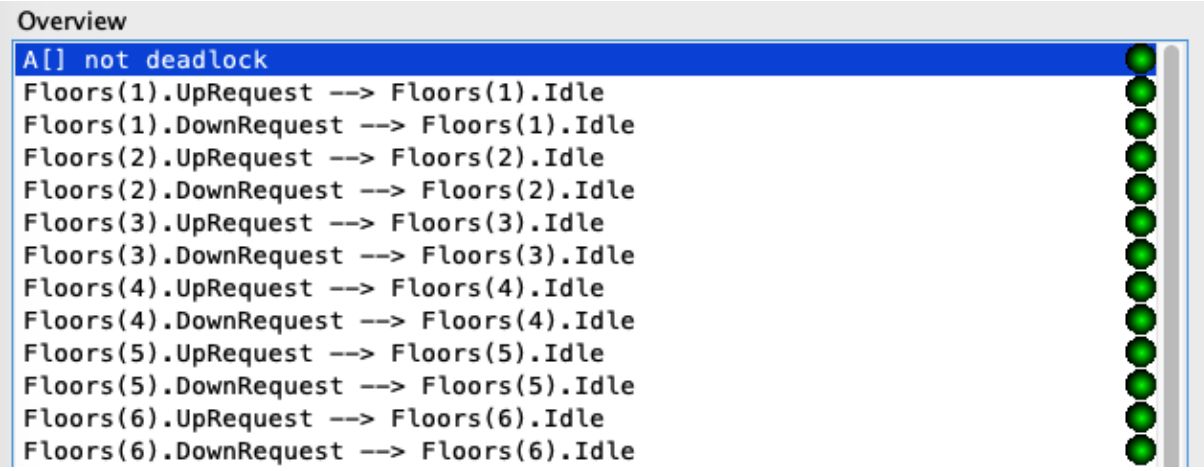

Figure 6.4: Floor State Machines Queries Verification

Figure 6.5: Elevator State Machines Queries Verification

To ensure that the behavior of author's state machine is working properly and satisfied all of the test, several queries have been made and tested using UPPAAL verifier feature. Those queries are:

1. **A[] not deadlock**
   To ensure that there is no deadlock in the designed state machine.
2. **Floors[floorNumber].UpRequest –> Floors[floorNumber].Idle**
   To ensure that each floor state machine eventually will come back to Idle from UpRequest state.

3. **Floors[floorNumber].DownRequest –> Floors[floorNumber].Idle**
   To ensure that each floor state machine eventually will come back to Idle from DownRequest state.

4. **Elevator1.MovingUp –> Elevator1.LoadingUnloading**
   To ensure that Elevator 1 will eventually goes to LoadingUnloading state from MovingUp state.

5. **Elevator1.MovingDown –> Elevator1.LoadingUnloading**
   To ensure that Elevator 1 will eventually goes to LoadingUnloading state from MovingDown state.

6. **Elevator2.MovingUp –> Elevator2.LoadingUnloading**
   To ensure that Elevator 2 will eventually goes to LoadingUnloading state from MovingUp state.

7. **Elevator2.MovingDown –> Elevator2.LoadingUnloading**
   To ensure that Elevator 2 will eventually goes to LoadingUnloading state from MovingDown state.

8. **E<> Elevator1.LoadingUnloading && Elevator1.currentFloor == each possible floor**
   To ensure that Elevator 1 will eventually serves each possible floor, in this task the possible floor is 1 to 6.

9. **E<> Elevator2.LoadingUnloading && Elevator2.currentFloor == each possible floor**
   To ensure that Elevator 2 will eventually serves each possible floor, in this task the possible floor is 1 to 6.

All of the queries have been tested using UPPAAL verifier feature and it satisfied each of the queries, as can be seen in Figure 6.4 and Figure 6.5.

7 Summarize

To summarize, designing Discrete Time systems is not a simple task. The relation between one state to the other could be intertwined with another states especially if the system has lots of state machines and it is interconnected to one another. But with tools like UPPAAL, it helps system designer to run a simulation of their system to make sure that the system works as intended and the most important one is there will be no deadlock in any given situation.