Lab 2 – Modelling State Machines in UPPAAL

Embedded Control Lab

November 2023

1 Introduction

Discrete-time modeling is a powerful approach in systems design, allowing us 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. Within the realm of discrete-time modeling, state machines emerge as a pivotal concept. These state machines, or automata, serve as abstract representations of systems that transition between various states, making them instrumental in modeling the behavior of a multitude of applications. As we explore this domain further, tools such as UPPAAL become indispensable. UPPAAL offers a comprehensive platform for simulating and verifying the behavior of state machines, ensuring their robustness and accuracy. To provide a tangible understanding, we will delve into a real-world application by modeling a traffic light system at a city cross-section, exemplifying the practical relevance of discrete-time modeling and state machines in everyday scenarios.

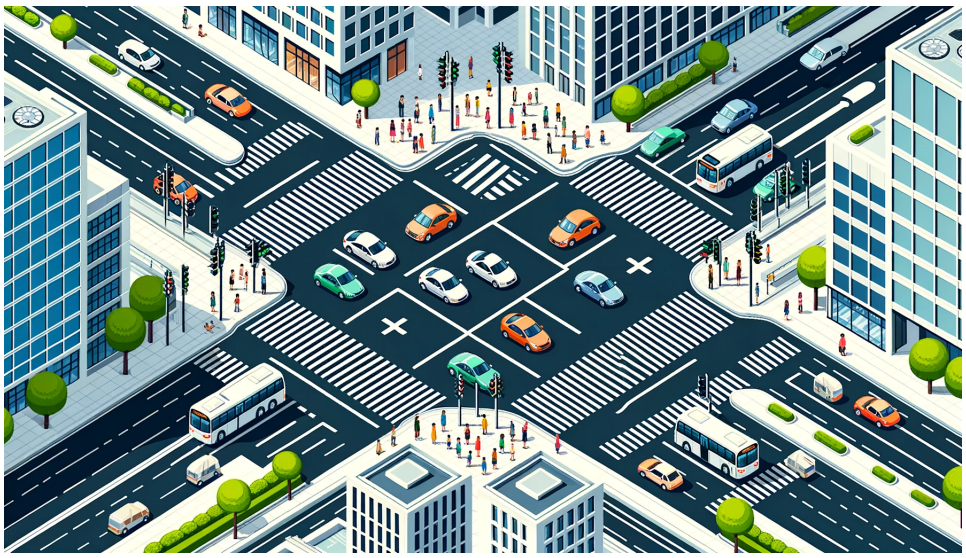

Figure 1: City Cross Section

2 State Machines

State machines, often known as automata, have roots that trace back to the early theories of computation and logic. Their inception 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.

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 transitions 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 the tangible implementation, allowing for systematic modeling of complex systems in a structured and predictable manner.

2.1 Components of State Machine

States: At the heart of every state machine lie its states, which represent the distinct conditions or situations a system can be in. Each state captures a specific configuration or behavior of the system. For instance, in the context of a digital watch, states might include "Displaying Time," "Setting Alarm," or "Timer Countdown." The system can only be in one state at any given time, and as events occur or conditions are met, it transitions from one state to another.

Transitions: Transitions define the pathways or conditions that enable a system to move from its current state to another. Every transition is associated with specific criteria that must be satisfied for the shift to occur. Two primary components characterize transitions:

Guards: A guard is a boolean expression that determines whether a transition is permissible. For the transition to be taken, the guard's condition must evaluate to true. For example, a digital watch might have a guard that checks if the current time equals the alarm time to transition from the "Displaying Time" state to the "Alarm Ringing" state.

Set Actions: Once a transition is taken, certain operations or set actions are executed. Using the digital watch example, a set action upon entering the "Alarm Ringing" state might be to initiate the alarm sound. Initial and Final States: Every state machine starts its operation from a predefined state known as the initial state. This state serves as the entry point, setting the system in motion. On the other hand, some state machines have designated final or terminal states. Upon reaching a final state, the system concludes its operation, indicating the end of its process.

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.

3 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 and Aalborg University, 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.

To download Uppaal 4.0, click – *Download Link*

3.1 Designing State Machines in UPPAAL

3.1.1 Interface Overview

Editor: This is the primary workspace where users design state machines as shown in fig. 2. It provides a canvas for drawing locations (states), transitions, and annotating them with relevant properties.

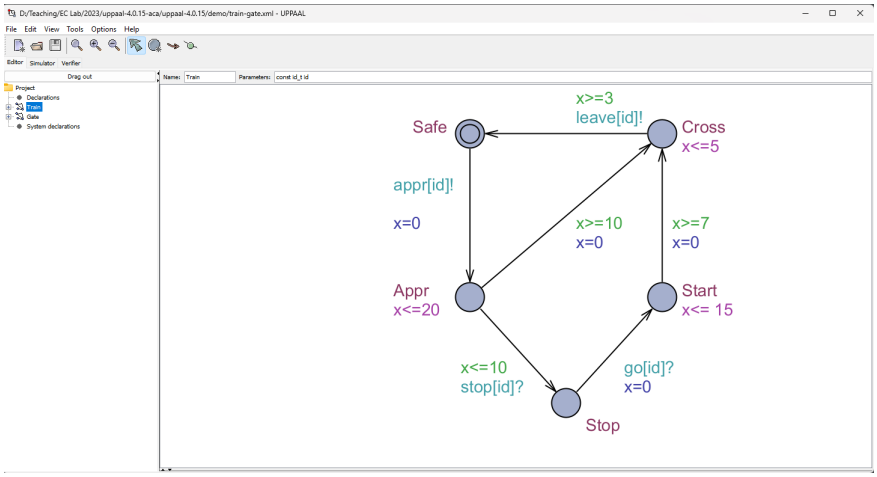

Figure 2: Editor tab in UPPAL

Simulator: Once a model is constructed, the simulator allows users to execute the state machine in a step-by-step manner, observing its behavior in response to different inputs and conditions. Figure 4 shows the simulator view in UPPAAL.

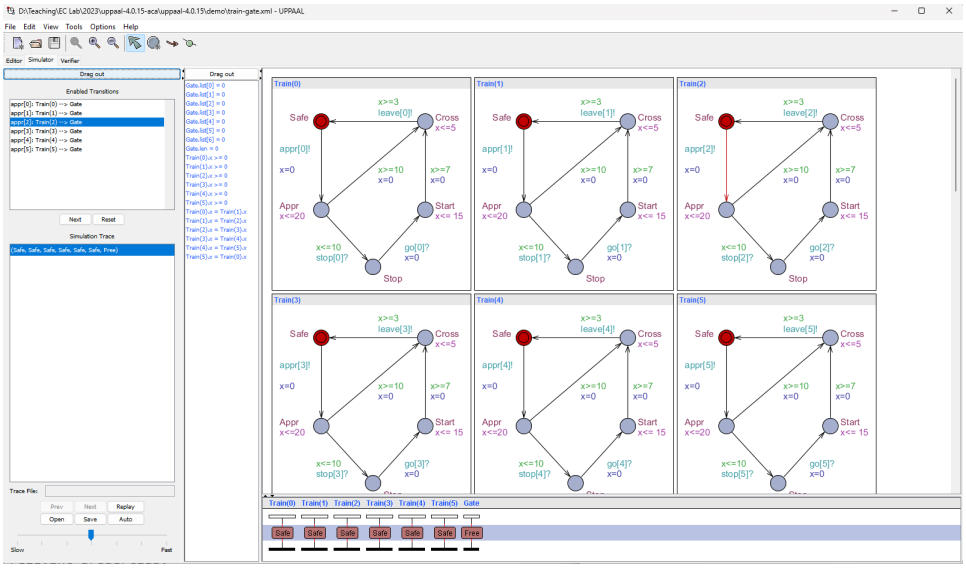

Figure 3: Simulator tab in UPPAAL

Verifier: This component is essential for the model checking process. Users can specify properties they wish to verify against their model, and the verifier evaluates whether the state machine satisfies these properties.

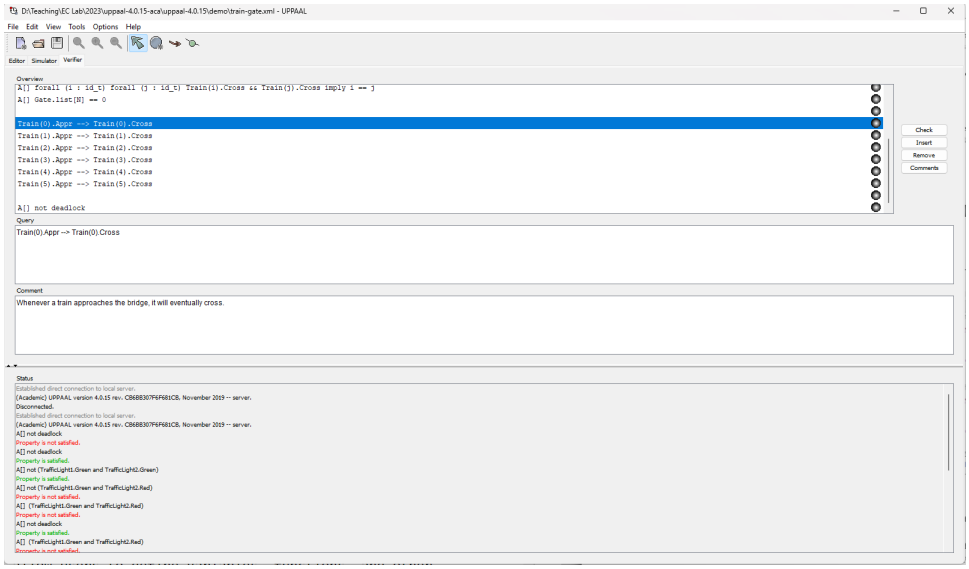

Figure 4: Verifier tab in UPPAAL

Property and Declaration Panes: Located in the Editor tab, these sections of the interface allow users to define variables, functions, and other declarations essential for the model.

3.1.2 Modeling Basics

Defining Locations, Transitions, and Initial States: In the Editor pane, you can easily define locations by placing circles and labeling them. Transitions are represented by arrows connecting these locations. Every state machine must have an initial state, distinguished by a double circle, indicating the starting point of the system.

Incorporating Clock Variables and Constraints: Clock variables are essential for timed automata. In the Declarations pane, you can define clock variables, and subsequently, in the Editor, you can annotate transitions and locations with clock constraints, specifying the timing behavior of the system.

Synchronization Channels and Urgent Actions: UPPAAL allows for modeling of concurrent systems using synchronization channels. These channels enable communication between different automata, ensuring coordinated behavior. Channels are declared in the Declarations pane and are used in transitions as synchronization labels. Urgent actions are special constructs in UPPAAL that force the system to take a particular transition as soon as its guard is enabled, without any delay.

System Declarations: While individual state machines capture the behavior of single components, real-world systems often consist of multiple interacting components. UPPAAL allows construction of such multi-component systems using the system declaration.

In the Declarations pane, after defining the individual automata, you can specify a system declaration that initializes and runs the automata concurrently. This declaration looks like:

```
system Automaton1, Automaton2;
```

It indicates that Automaton1 and Automaton2 should run concurrently and may interact through synchronization channels if defined.

3.1.3 Simulation Basics

Navigate to the Simulator pane and click on the "Next" or "Auto" button. "Next" allows you to manually step through simulation one transition at a time. While the simulation is paused at any point, you can inspect the current state, variable values, and clock valuations.

As the simulation progresses, UPPAAL provides a real-time visualization in the right pane. Locations change color based on their active/inactive status, and transitions animate as they're taken, offering a dynamic view of the system's behavior.

3.2 Model Verification in UPPAAL

Timed CTL (TCTL) extends the traditional Computation Tree Logic (CTL) by introducing 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, $A[]\phi$ 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, $A <> \phi$ 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. $E[]\phi$ 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. $E <> \phi$ 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."

4 Tasks

4.1 Task 1

Model car and pedestrian traffic lights at a city crossroad, where one road runs along the North-South direction and other along the East-West. Each road has a pedestrian crossing.

Car Traffic Lights: Model two traffic lights for cars– one for each direction. Each traffic light should have the standard three states: Red, Yellow, and Green. Ensure synchronization such that at any point, both car traffic lights aren't green simultaneously to avoid accidents.

Pedestrian Traffic Lights: Model two pedestrian lights, one for each crossing. Lights should have two states: Green and Red. Synchronize it with car traffic lights to ensure pedestrian safety. For instance, when cars have a green light, the corresponding pedestrian light should be red .

The key challenge is to synchronize the four lights to ensure smooth traffic flow and maximum safety. Make sure to properly configure the intial states of four traffic lights.

Simulation and Verification: Once the model is set up, use UPPAAL simulator to observe the behavior of your traffic light system.

Formulate verification queries in UPPAAL to ensure the following:

- 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 pedestrian lights are green, corresponding car lights should indicate red.
- Both pedestrian and car lights should eventually turn green
- All lights should never be red at the same time.

4.2 Task 2

Model two elevator system where the elevators services requests from six floors. The elevators maintains a queue of requests and efficiently travel between floors to service these requests.

a. Elevator Template

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.

Transitions:

- From Idle to MovingUp or MovingDown based on the request queue.
- From MovingUp or MovingDown to Loading/Unloading once the elevator reaches the target floor.

- From Loading/Unloading back to Idle once the operation is complete.

Variables:

- int currentFloor: To keep track of the elevator's current floor.
- int targetFloor: The next floor the elevator is heading to.
- int requestQueue: A queue to maintain the floor requests.

Constraints:

- 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 , for example, if first elevator receives the a request from any of the floors, then next request should go to the second elevator.
- Elevators should not stay for more than 60 seconds in MovingUp or MovingDown states.

b. Floor Template

Since there are nine floors, model the floors as a template in UPPAAL, parameterized by a floor number.

States:

- Idle: No request has been made from this floor.
- UpRequest: A request to go up has been made.
- DownRequest: A request to go down has been made.

Transitions:

- Idle to UpRequest when an upward request is made.
- Idle to DownRequest when a downward request is made.
- Return to Idle after the request has been serviced.

Synchronization Channels:

- request_up[floorNumber]! : To send an upward request from a floor.
- request_down[floorNumber]! : To send a downward request from a floor.
- ack[floorNumber]? : To receive acknowledgement that request has been serviced

Constraints:

- 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.

Simulation and Verification: Once the model is set up, use UPPAAL simulator to observe the behavior of your elevator system and Formulate verification queries in UPPAAL to ensure the following:

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

4.3 Deliverables

1. UPPAAL files for each task.
2. Report describing your approach, understanding of state machines modelling, simulation results from UPPAL, the verification queries, and outcomes.

4.4 Evaluation Criteria

1. Model Complexity and Accuracy
2. Simulation Behavior
3. Verification Outcomes: All your verification queries should return "Satisfied."
4. Quality of report.