A Comparison of LLMs for UML State Diagrams Generation

Martina Basic[1]; Marko Vujasinovic[2]

[1,2]University of Applied Sciences Aspira
Department of Computer Engineering
Split

Publication Date: 2026/03/13

Abstract: Large Language Models (LLMs) are becoming an increasingly important tool in the field of software engineering, particularly in the automation of UML model creation based on requirements described in natural language. This paper examines the effectiveness of applying LLM technologies to generate UML 2.5 state diagrams in PlantUML format from textual system descriptions. The creation of state diagrams traditionally requires a strong understanding of the problem domain and experience in formal modeling, and it is often a time-consuming and error-prone process. The automated generation of UML state diagrams requires models to correctly identify system states, transitions between states, events, guard conditions, as well as entry, exit, and internal activities, while adhering to the syntax and rules of UML 2.5 notation. In the evaluation part of the study, three systems of varying levels of complexity were analyzed in order to assess the models' ability to produce syntactically correct and semantically complete diagrams. Additionally, the impact of different prompt design strategies on the quality of the generated results was examined. The results indicate that LLMs can effectively generate state diagrams for simpler systems and significantly reduce the effort required for initial modeling. However, for more complex systems, issues were observed in the consistent interpretation of requirements, the modeling of hierarchical structures, and the preservation of system semantics. Overall, LLM technologies represent valuable support in the early stages of UML state diagram development; however, the final quality of the models still largely depends on expert validation and additional guidance provided by domain specialists.

Keywords: *LLM; UML State Machine Diagram; Prompt Engineering; Requirement Specification.*

How to Cite: Martina Basic; Marko Vujasinovic (2026) A Comparison of LLMs for UML State Diagrams Generation. *International Journal of Innovative Science and Research Technology*, 11(2), 3134-3158. https://doi.org/10.38124/ijisrt/26feb1435

I. INTRODUCTION

Large Language Models (LLM) are advanced artificial intelligence systems designed to process, understand, and generate text in natural language. The development of LLMs has significantly influenced the way automation tasks are approached in the field of software engineering. Based on descriptions written in natural language, they enable the generation of source code, technical documentation, and various system models. Of particular interest is their application in generating UML diagrams, which represent a fundamental tool for modeling and documenting software systems. Creating UML diagrams is an extremely demanding and time-consuming task that requires labor-intensive processes (Nguyen et al., 2026). LLMs can facilitate the creation of UML diagrams based on user descriptions, potentially resulting in significant time savings (Cámara et al., 2023; Al-Ahmad et al., 2025).

In this paper, the focus is on the usage of LLMs to facilitate creation of Unified Modeling Language (UML) state diagram (Object Management Group, 2017). UML state diagrams have significant application in modeling states and state transitions (processes) in software engineering, as they enable comprehensive understanding of the behavior of system components.

The specific aim of this paper is to present the work done to evaluate the capability of three LLMs, namely, ChatGPT (https://chatgpt.com/), Grok (https://grok.com/) and Qwen (https://qwen.ai/home) in generating UML 2.5 state diagrams in PlantUML format based on differently structured natural language prompts. Emphasis is placed on evaluation of syntactic correctness, semantic completeness, structural complexity, and the impact of prompt design strategy on the quality of the generated models. The structure of the paper is as follows: Chapter 2 presents related work. Then, Chapters 3 and 4 introduce the theoretical background and the research methodology. Chapter 5 presents the case studies and experimental work. Finally, Chapter 6 discusses the results, followed by the conclusion.

II. RELATED WORK

The application of LLMs in software engineering has been intensively researched in recent years, particularly in the context of source code generation, automated testing, and system documentation. One of the significant challenges is translating unstructured information into formal UML models. Natural language processing (NLP) techniques have been used to directly derive UML class diagrams from requirements specifications. However, NLP-based methods face difficulties in accurately interpreting linguistic nuances. Research has shown that existing approaches often generate incomplete or inaccurate UML diagrams from complex requirements texts (Nguyen et al., 2026).

Several empirical studies have already demonstrated that LLMs, such as GPT-based agents, can generate UML models from natural language input with varying levels of correctness and completeness, producing textual UML notations such as PlantUML and transforming these textual representations into visual diagrams (Cámara et al., 2023; Basic & Vujasinovic, 2026). Jahan et al. (2024) experimented with a generative LLM (ChatGPT) to automatically generate UML sequence diagrams from user stories. They found that ChatGPT sometimes produces overly complex sequence diagrams that go beyond the scope of the user stories.

Although previous research has mainly focused on generating UML class diagrams, sequence diagrams, and use case diagrams using LLMs, the generation of UML 2.5 state diagrams, to the best of our knowledge, remains a relatively underexplored area. State diagrams require precise modeling of states, transitions, guards (conditions), entry and exit actions, as well as hierarchical and parallel structures, which poses an additional challenge for LLMs.

Our research empirically evaluates how the capabilities of LLMs can be leveraged to generate UML state diagrams, with particular consideration given to the prompt engineering.

III. THEORETICAL BACKGROUND

A UML state diagram belongs to the group of UML behavioral diagrams. UML state diagrams are most used during the system design process to achieve a more comprehensive understanding of the behavior of system components, through the transitions of system's states. They primarily serve to model the dynamic behavior of system components and illustrate how objects transition between states in response to various events, thereby supporting the design of the system's control logic. They represent the functionality of parts of a software system and often the transitions between states triggered by events, but they do not depict actors or the external interface toward end users.

The elements of a state diagram are the initial state, final state, state, transitions between states, decision or conditional branching, fork and join, and composite state. A state is the fundamental component of a state diagram and is represented by a rounded rectangle. The only mandatory element in a state label is its name, which defines its meaning or purpose and helps in understanding the overall diagram. A state represents a set of specific properties of an entity (e.g., an object, system, or system component), describing how the entity behaves and how it responds to events. A state may include certain activities associated with it, as well as defined conditions or criteria that determine when the entity will transition from one state to another. Within a state, entry and exit actions can be defined, as well as an activity associated with the state that is executed continuously while the entity remains in that state. An entry action is indicated by the keyword entry/, an exit action by the keyword exit/, and a state activity by the keyword do/.

A state diagram may contain one initial state and multiple final states. The initial state is represented by a filled black circle, while the final state is represented by a filled black circle enclosed within another circle. A transition between states is represented by an arrow and may include: the event that triggered the transition, a guard condition or expression that must be satisfied for the transition to occur (written in square brackets), an action that occurs during the transition if the guard condition is satisfied (the action is separated from the event by a slash /), and the type of event, such as call, signal, change, or time event. A decision or conditional branching is represented by a white diamond from which two arrows emerge. When certain activities occur in parallel, fork and join elements are used. A composite state consists of a hierarchy of states and transitions; it is represented by a rounded rectangle containing a nested state diagram within it.

Modeling the behavior of a system or a part of a system using UML state diagrams is a complex process whose details depend on the specific problem. In all but the simplest cases, it is an iterative process in which the model is gradually refined and improved as understanding of the behavior of the system component being modeled increases. When creating a state diagram, it is advisable, if available to use other UML diagrams as a foundation, such as an use case diagram, a sequence diagram, or a class diagram.

Large Language Models (LLMs) are advanced artificial intelligence systems designed to process, understand, and generate human-like text in natural language. Some of the most important and well-known LLMs include the OpenAI GPT series, Claude by Anthropic, LLaMA (and Gemma) by Meta, PaLM / Gemini by Google, BLOOM by BigScience, and Qwen.

LLMs are mathematical models based on neural networks, deep learning techniques, and training on large-scale datasets. LLMs operate on natural languages and are trained on massive corpora typically containing billions of words from various sources such as websites, books, and articles. They do not understand text in the same way humans do; instead, they recognize patterns in language, predict the next word in a sequence, and generate responses based on statistical relationships and context.

LLMs may misinterpret unclear requirements and generate inaccurate information. LLMs do not understand long-term context in the same way humans do and require expert human supervision in professional applications. Users interact with the model through prompts written in natural language in the form of questions, tasks, or descriptions, and the model generates responses in textual form.

IV. METHODOLOGY

For the purposes of this research, three LLMs were selected: GPT-4, Grok, and Qwen, which are widely available and applicable in the field of software engineering and automated diagram generation. Given that UML state diagrams are not as widely represented in academic literature related to usage of LLMs for UML diagrams creation, the selection of tools was based on their performance in generating UML class diagrams and UML sequence diagrams.

The selection criteria were based on tool popularity, natural language understanding capability, proven ability to generate UML diagrams, and availability for repeated experimental testing. The selected LLMs represent both commercial architectures (GPT-4 and Grok) and an open-source architecture (Qwen).

Table 1 LLMs

Tool	Owner	Year of Launch	Commercial / Free
ChatGPT	OpenAI	November 2022	Commercial
Grok	xAI	November 2023	Commercial
Qwen	Alibaba Cloud	April 2023	Free

ChatGPT is a generative artificial intelligence tool developed by OpenAI. It is the most well-known and widely used LLM in the world. For the purposes of this experiment, ChatGPT based on GPT-4.1 was used.

Grok is an AI chatbot developed by xAI. For the purposes of this experiment, the Grok 3 Full / Flagship model was used, which, as of February 2026, represents the most advanced version available in the official chat.

Qwen is an open-source artificial intelligence tool developed by Alibaba Cloud. For the purposes of this experiment, the latest version of the model, Qwen3.5-Plus, was used.

When generating results, Grok specifies the time to first token and the total response time, and it offers the option to export the solution in .pdf format. The other two tools do not provide a display of response time, so their response time was measured using a timer.

The free versions of Grok and ChatGPT were used, but registration was required in order to access them. All the tools used allowed the upload and analysis of textual documents only after registration.

Additionally, all the tools imposed limits on the number of messages and the number of analyzed documents per day. As a result, it was necessary to take breaks of several hours (ranging from 2 to 12 hours) between multiple conversations. The defined breaks were not the same for all prompts, as the prompts varied in size and did not consume the same number of tokens.

All tools used in the experiment received one or more user prompts in the form of questions, instructions, or descriptions written in natural language, and based on them, they predicted and generated the corresponding responses. The prompts were written in English.

Three examples of varying complexity were tested: from the simplest example of a (1) traffic light system, through the more complex operation of an (2) ATM, to the most complex example of a (3) lighting system within a smart home system.

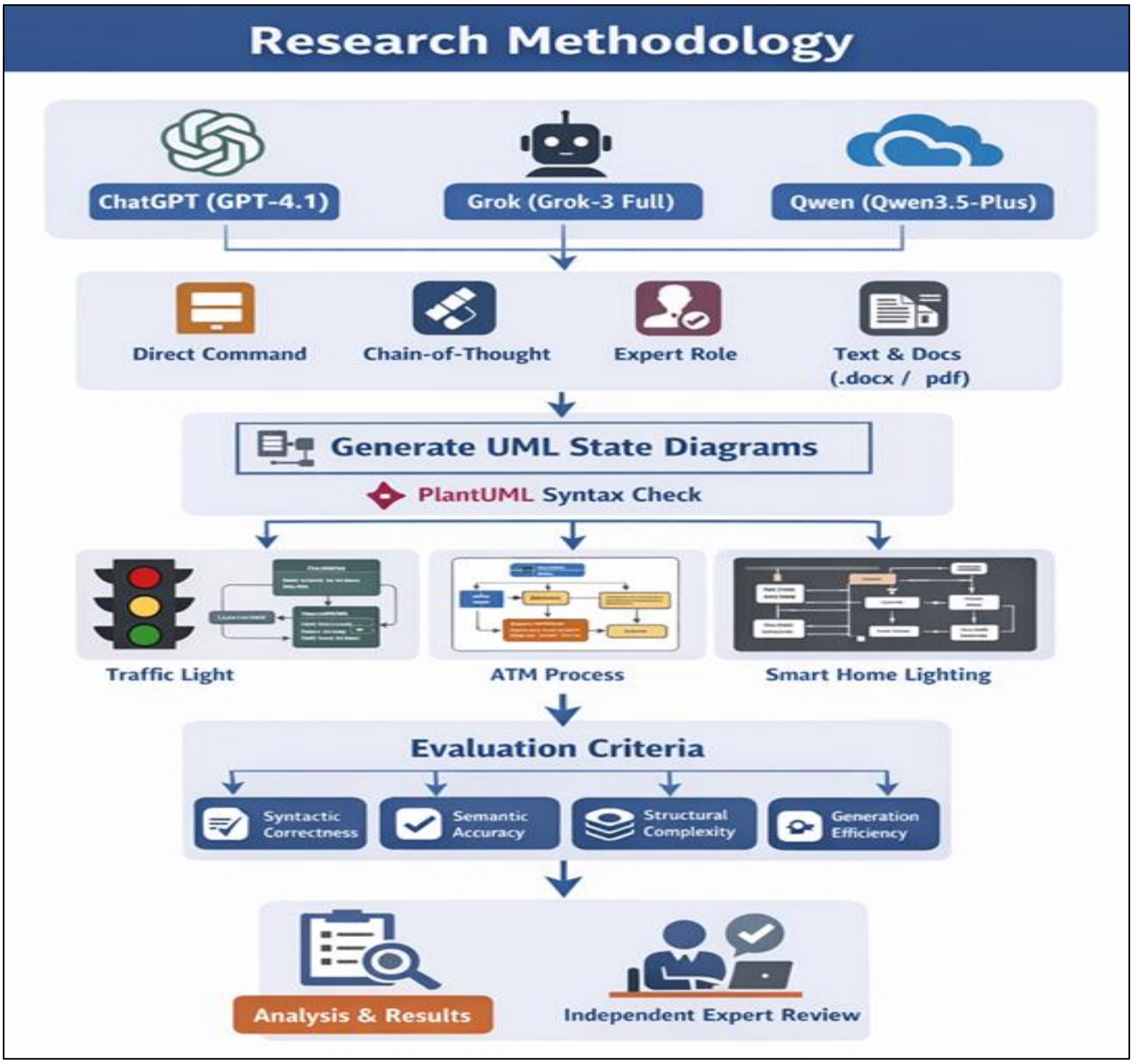


Fig 1 Methodology Overview Created in ChatGPT

For each example, prompts were formulated in four different ways to examine the impact of prompt design on the quality of the results: (1) By giving a direct command to create a UML state diagram for the selected process (e.g., *Create a state diagram for the smart home lighting system. Use only UML 2.5 notation. Generate only PlantUML code without additional text.).* (2) By applying a chain-of-thought approach (e.g., *Think step by step and create a state diagram for the lighting system within the smart home system. Define all states. Identify events and transitions. Add guards, actions on input, output, and during the state. Use only UML 2.5 notation. Generate only PlantUML code*.). (3) By assigning the role of a senior modeler with 20 years of experience in creating UML diagrams (e.g., *You are a senior software modeler with 20 years of experience modeling UML diagrams. Think step by step and create a state diagram for the lighting system within the smart home system. Define all states. Identify events and transitions. Add guards, actions on input, output, and during the state. Use only UML 2.5 notation. Generate only PlantUML code.*). (4) By providing a textual description and textual documents (.docx and .pdf) along with a command to create a UML state diagram.

The generated code was automatically tested using the PlantUML parser available at PlantUML (https://www.plantuml.com/) to verify syntactic correctness. The version of the PlantUML parser used is PlantUML 1.2026.2 beta3. If a syntax error occurred during the generation of the UML state diagram, the resulting error image was imported into the tool, and it was asked to correct the code according to the identified error.

A manual evaluation of the generated UML state diagrams was conducted. The evaluation was performed by an independent expert with experience in UML diagram modeling. The assessment included the following criteria: syntactic correctness (the percentage of outputs that can be executed in PlantUML without errors), semantic correctness of the model (accuracy of states and transitions, correct use of guards, presence of initial and final states), structural complexity (use of entry/exit actions, conditional transitions, and parallelism) and generation efficiency, measured through the average generation time.

All models were tested within the same time period in order to minimize the impact of model version changes. The same prompts were used for all examples. Each prompt was executed five times per model in each tool to assess consistency. A new conversation was used each time. The temperature has balanced towards a lower temperature by giving commands, e.g., be maximally precise, use strictly UML 2.5 concepts, etc. To facilitate evaluation, all queries require a strict PlantUML format as the only output after thinking.

V. CASE STUDY AND EXPERIMENT

Three examples of varying complexity were tested: from the simplest example of a (1) traffic light system, through the more complex operation of an (2) ATM, to the most complex example of a (3) lighting system within a smart home system.

(1) For the purposes of traffic light system experiment, a UML state diagram was generated for a traffic light system according to the European standard. A basic version of the system was used, with the following defined states: red, yellow and green light. When the traffic light shows red, vehicles stop, and when it shows green, vehicles proceed. The yellow light signals the transition to the red light, while the yellow light in combination with the red light signals the switch to green. The pedestrian traffic light system was not included in this model.

(2) The operating principle of an ATM is as follows: after the card is inserted into the ATM, its verification is performed. If the card is accepted, the user enters the PIN. Following successful authentication, the system displays a transaction menu: cash withdrawal, balance inquiry, cash deposit, PIN change, or other options. For cash withdrawal, the ATM first checks the available account balance and daily withdrawal limits. If the transaction is possible, it dispenses cash, updates the account balance in real time via the network, and logs the transaction. After completion, the user can proceed with another transaction or end the session – in which case the card is returned and the ATM returns to the idle state.

(3) In a smart home lighting system, the lighting elements can reside in one of four primary states: off, on, dimmed, and automatic. Transitions between these states are triggered by user interventions (via a mobile application or physical switch), sensor events (presence detection, changes in external illumination), or time-based automation rules. The light intensity can be continuously adjusted between dimmed and full brightness levels, depending on current needs or contextual conditions. The system may also incorporate security or alarm modes (e.g., flashing lights upon detection of fire or unauthorized entry). All state transitions follow conditional "if–then" logic for example, automatically turning off the lights after a specified period of inactivity in the room during nighttime hours.

In the conversation of each tool, a prompt was submitted for each of the three tested examples.

(Prompt 1) *Create a state diagram for a traffic light (according to European standards) / an ATM / a smart home lighting system. Uses UML 2.5 notation exclusively. Generate only PlantUML code without additional text.*

Results obtained after submitting the prompt for a traffic light (according to European standards).

Grok generated PlantUML code, shown in Listing below.

```
@startuml
skinparam backgroundColor transparent
skinparam state {
  BackgroundColor<<red>> #d00
  BorderColor<<red>> #800
  FontColor<<red>> white
  BackgroundColor<<yellow>> #ffcc00
  BorderColor<<yellow>> #cc9900
  FontColor<<yellow>> black
  BackgroundColor<<green>> #080
  BorderColor<<green>> #040
  FontColor<<green>> white
}
[*] --> Red: system on
State Red <<red>> {
  [*] --> SteadyRed
  SteadyRed --> BlinkingYellow: emergency mode / flash
yellow
}
state Yellow <<yellow>> {
  [*] --> SteadyYellow
}
state Green <<green>> {
  [*] --> SteadyGreen
}
Red --> Green : timer expired
Green --> Yellow : timer expired
```

```
Yellow --> Red : timer expired
note right of Yellow
  Yellow is always
  a short transitional
  Phase (3–5 sec)
End note
note bottom of Green
  Green → Red transition
  usually has no direct
  green-to-red change
  (yellow is mandatory)
end note
Red -[hidden]up-> Yellow
Yellow -[hidden]up-> Green
@enduml
```

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 2.

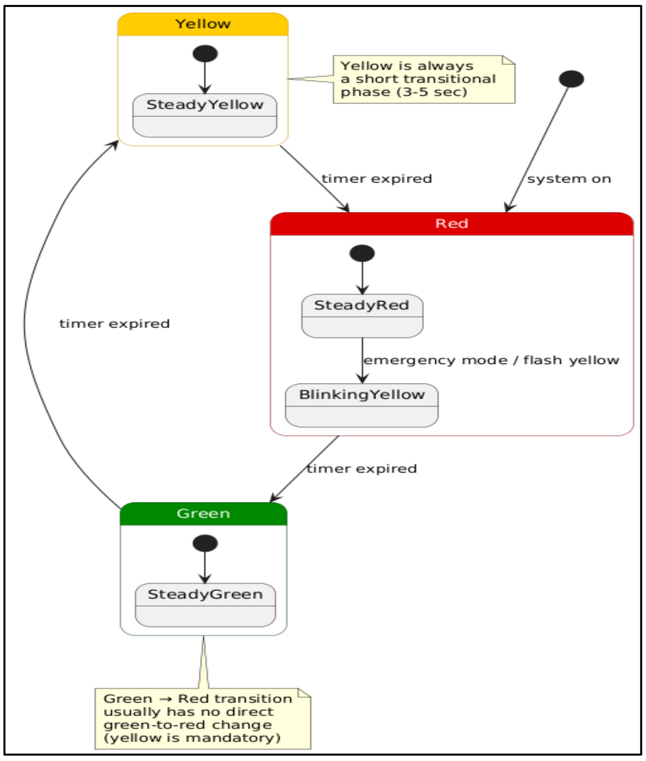


Fig 2 UML State Diagram Created by Grok

ChatGPT generated PlantUML code, shown in Listing below.

```
@startuml
hide empty description
[*] --> Red
Red --> RedYellow : timerExpired
RedYellow --> Green : timerExpired
Green --> Yellow : timerExpired
Yellow --> Red : timerExpired
state Red
state RedYellow
state Green
state Yellow
@enduml
```

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 3.

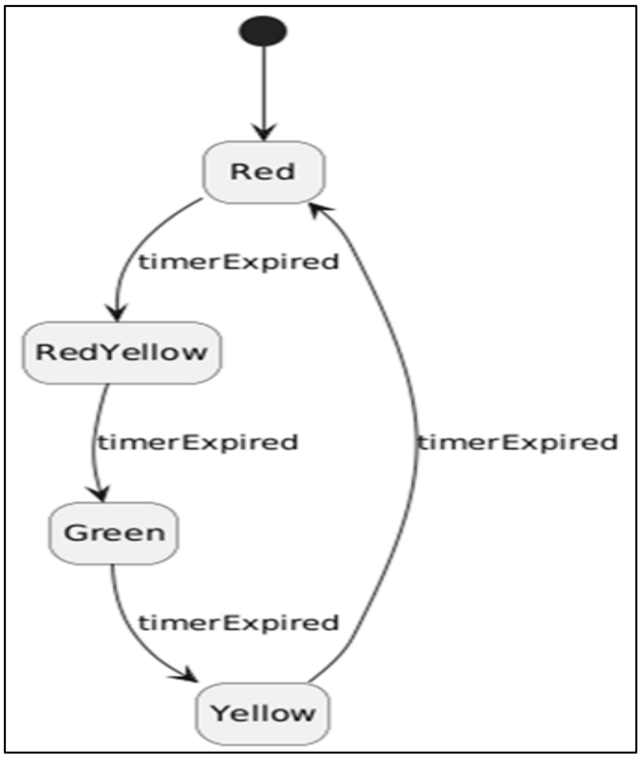


Fig 3 UML State Diagram Created by ChatGPT

Qwen generated PlantUML code, shown in Listing below.

```
@startuml
[*] --> Red
state Red {
  entry : activateRed()
}
state "Red + Amber" as RedAmber {
  entry : activateRedAndAmber()
}
state Green {
  entry : activateGreen()
}
state Amber {
  entry : activateAmber()
}
Red --> RedAmber : TimerExpired
RedAmber --> Green : TimerExpired
Green --> Amber : TimerExpired
Amber --> Red : TimerExpired
@enduml
```

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 4.

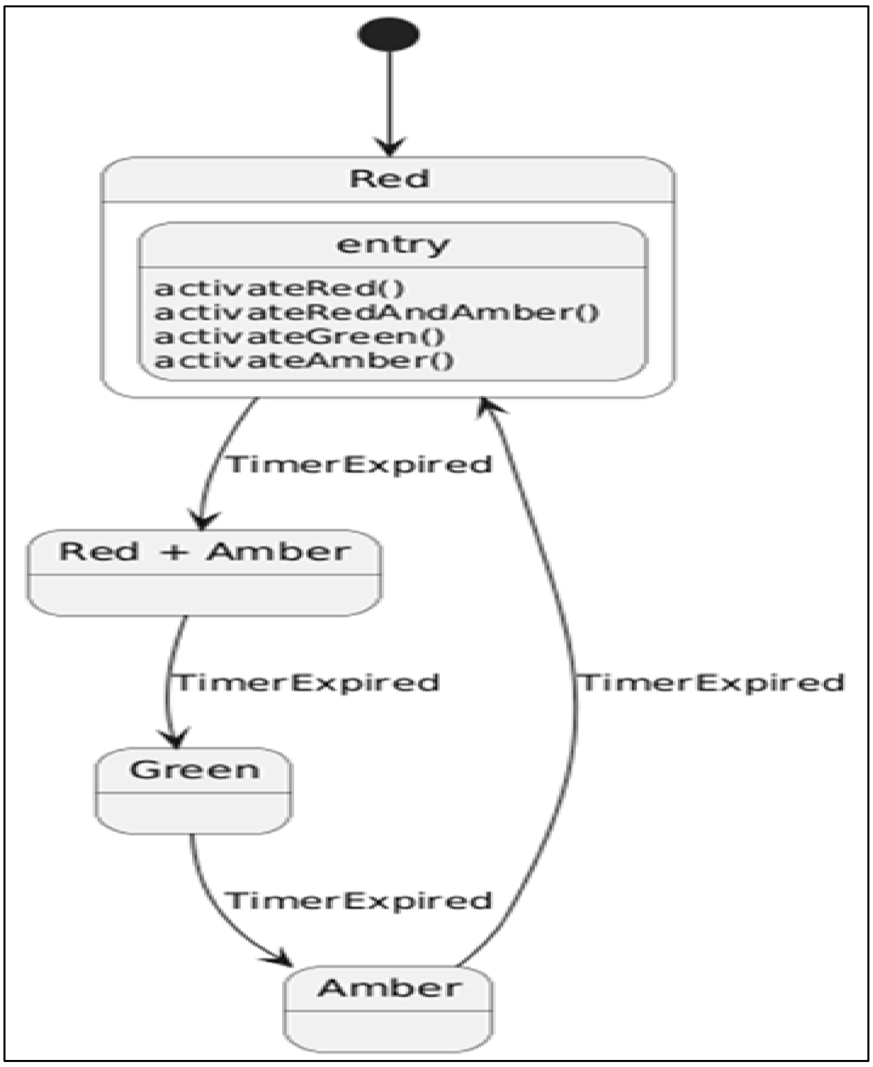


Fig 4 UML State Diagram Created by Qwen

Results obtained after submitting the prompt for an ATM.

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 5.

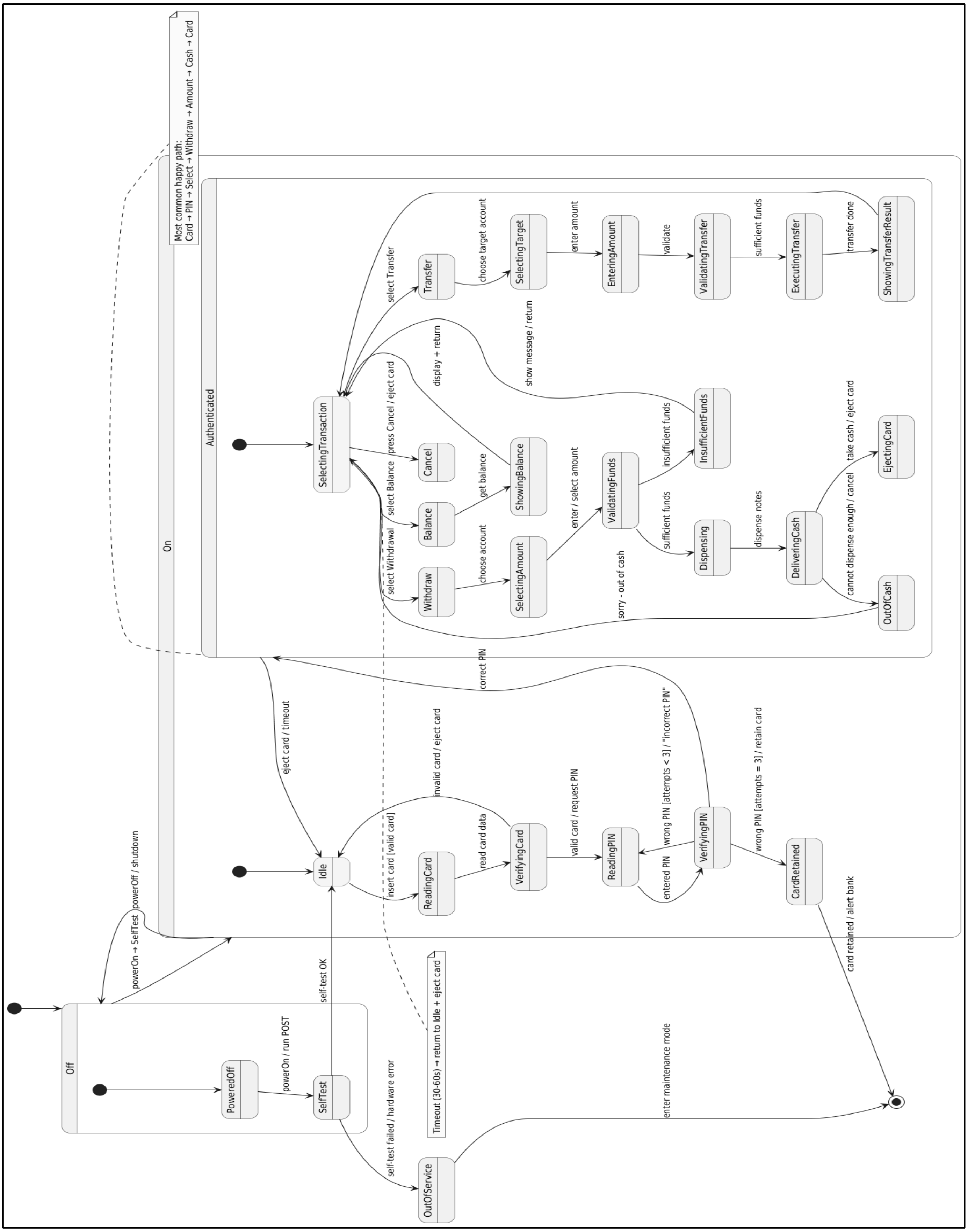


Fig 5 UML State Diagram Created by Grok

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 6.

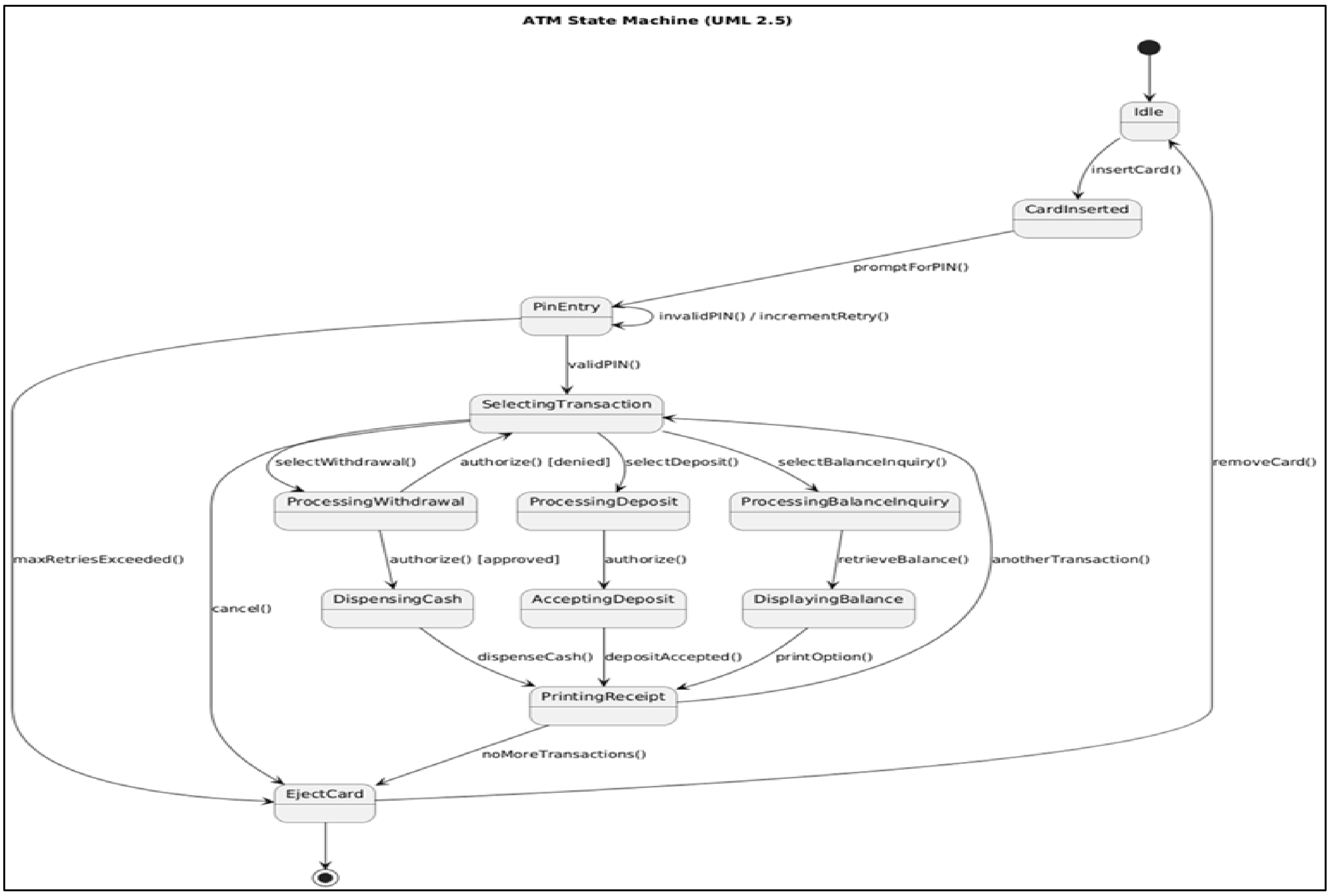


Fig 6 UML State Diagram Created by ChatGPT

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 7.

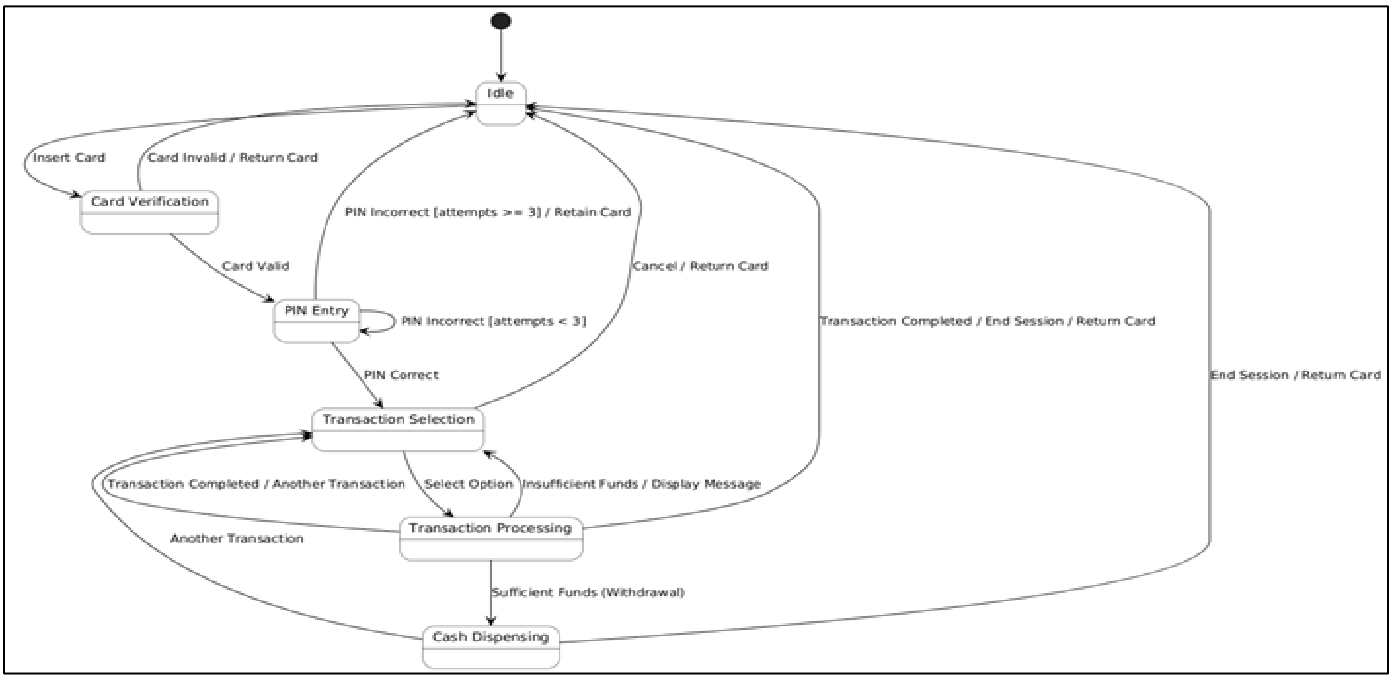

Idle
Insert Card
Card Invalid / Return Card
Card Verification
Card Valid
PIN Incorrect [attempts >= 3] / Retain Card
Cancel / Return Card
PIN Entry
PIN Incorrect [attempts < 3]
Transaction Completed / End Session / Return Card
End Session / Return Card
PIN Correct
Transaction Selection
Transaction Completed / Another Transaction
Select Option
Insufficient Funds / Display Message
Another Transaction
Transaction Processing
Sufficient Funds (Withdrawal)
Cash Dispensing

Fig 7 UML State Diagram Created by Qwen

Results obtained after submitting the prompt for a smart home lighting system.

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 8.

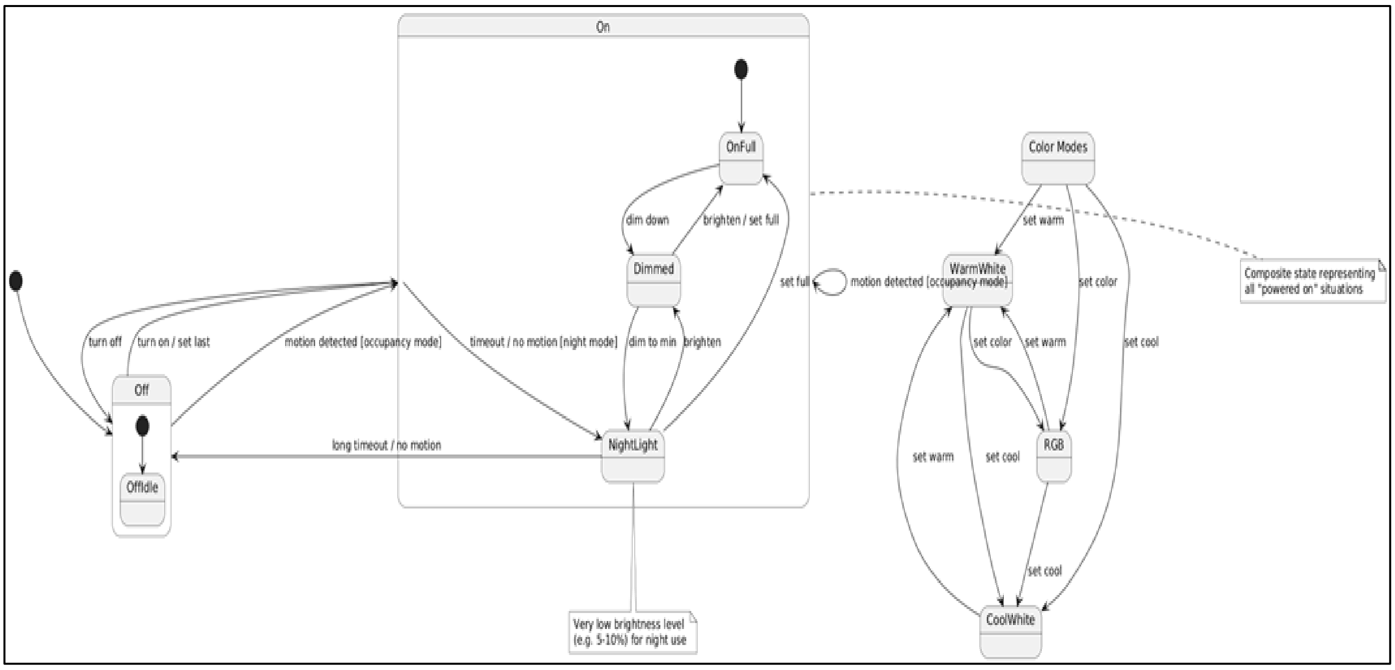


Fig 8 UML State Diagram Created by Grok

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 9.

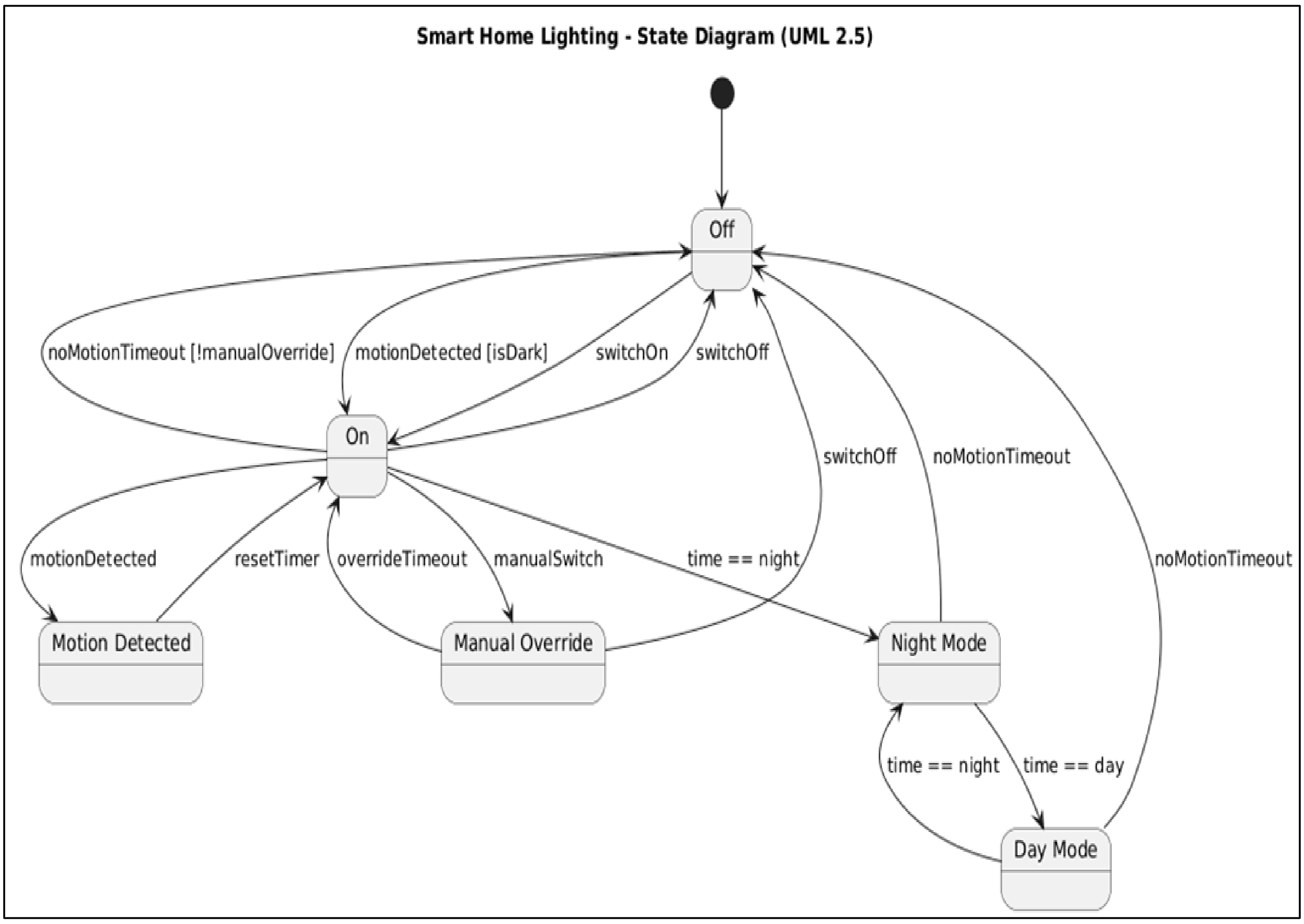


Fig 9 UML State Diagram Created by ChatGPT

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 10.

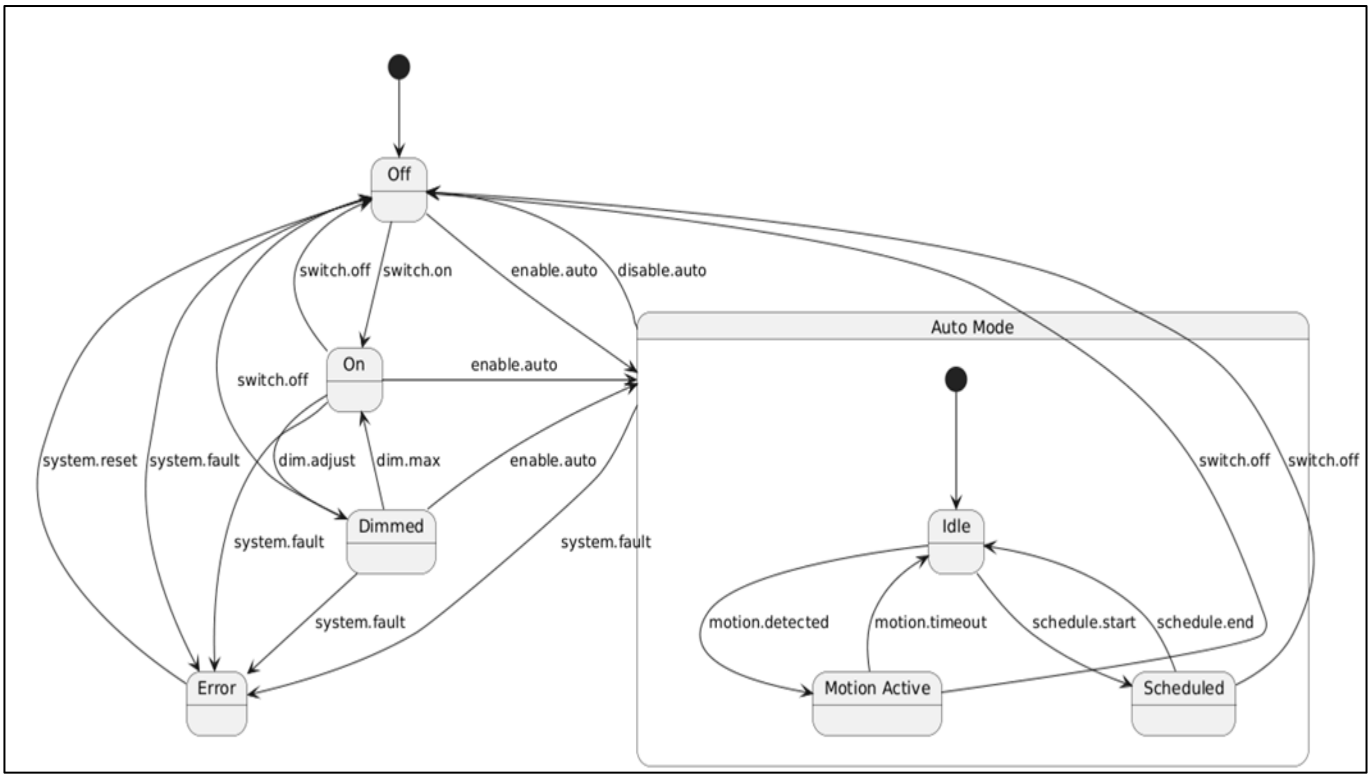


Fig 10 UML State Diagram Created by Qwen

(Prompt 2) *Think step by step and create a state diagram for a traffic light (according to the European standard) / an ATM / a smart home lighting system. Define all states. Identify events and transitions. Add guards, actions at entry, exit and during states. Use only UML 2.5 notation. Generate only PlantUML code.*

Results obtained after submitting the prompt for a traffic light (according to European standards).
UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 11.

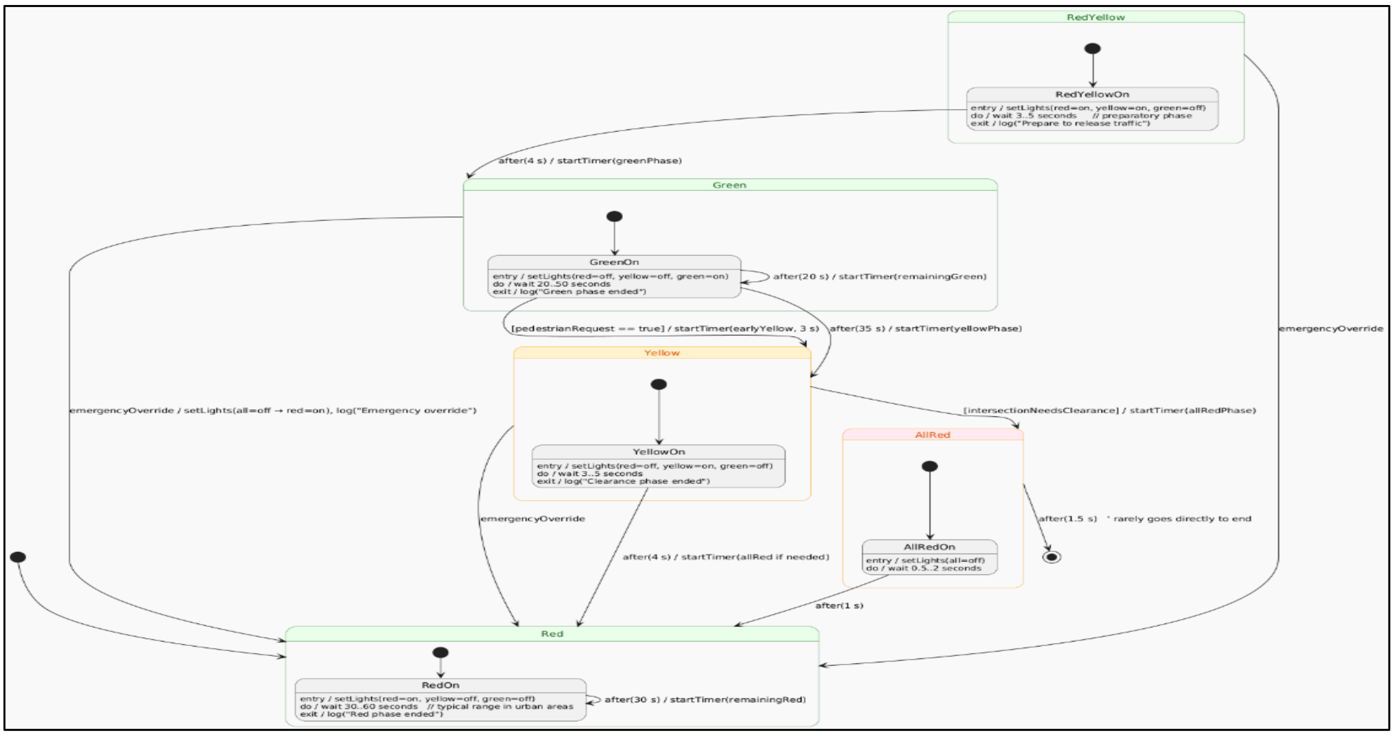

RedYellow
RedYellowOn
entry / setLights(red=on, yellow=on, green=off)
do / wait 3..5 seconds // preparatory phase
exit / log("Prepare to release traffic")
after(4 s) / startTimer(greenPhase)
Green
GreenOn
entry / setLights(red=off, yellow=off, green=on)
do / wait 20..50 seconds
exit / log("Green phase ended")
after(20 s) / startTimer(remainingGreen)
[pedestrianRequest == true] / startTimer(earlyYellow, 3 s)
after(35 s) / startTimer(yellowPhase)
emergencyOverride
Yellow
YellowOn
entry / setLights(red=off, yellow=on, green=off)
do / wait 3..5 seconds
exit / log("Clearance phase ended")
emergencyOverride / setLights(all=off → red=on), log("Emergency override")
[intersectionNeedsClearance] / startTimer(allRedPhase)
AllRed
AllRedOn
entry / setLights(all=off)
do / wait 0.5..2 seconds
after(1.5 s) ' rarely goes directly to end
emergencyOverride
after(4 s) / startTimer(allRed if needed)
after(1 s)
Red
RedOn
entry / setLights(red=on, yellow=off, green=off)
do / wait 30..60 seconds // typical range in urban areas
exit / log("Red phase ended")
after(30 s) / startTimer(remainingRed)

Fig 11 UML State Diagram Created by Grok

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 12.

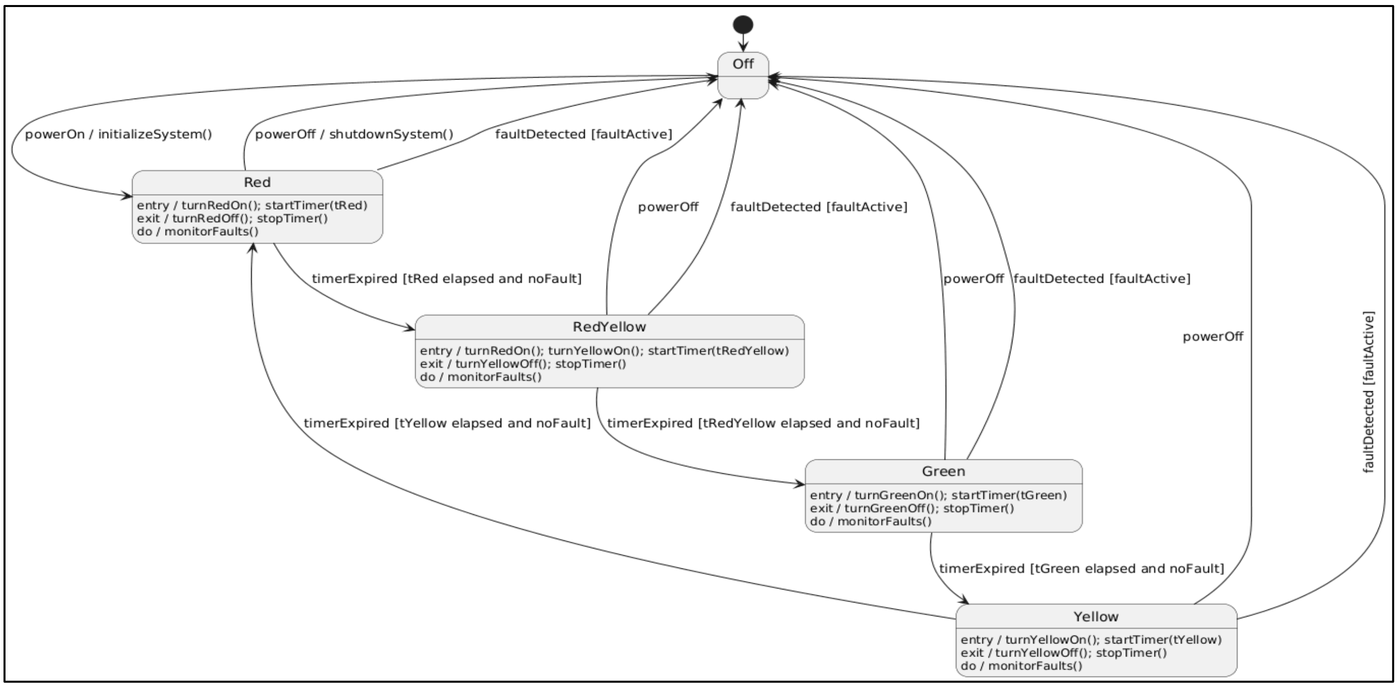

Off
powerOn / initializeSystem()
powerOff / shutdownSystem()
faultDetected [faultActive]
Red
entry / turnRedOn(); startTimer(tRed)
exit / turnRedOff(); stopTimer()
do / monitorFaults()
powerOff
faultDetected [faultActive]
timerExpired [tRed elapsed and noFault]
powerOff
faultDetected [faultActive]
powerOff
faultDetected [faultActive]
RedYellow
entry / turnRedOn(); turnYellowOn(); startTimer(tRedYellow)
exit / turnYellowOff(); stopTimer()
do / monitorFaults()
timerExpired [tYellow elapsed and noFault]
timerExpired [tRedYellow elapsed and noFault]
Green
entry / turnGreenOn(); startTimer(tGreen)
exit / turnGreenOff(); stopTimer()
do / monitorFaults()
timerExpired [tGreen elapsed and noFault]
Yellow
entry / turnYellowOn(); startTimer(tYellow)
exit / turnYellowOff(); stopTimer()
do / monitorFaults()

Fig 12 UML State Diagram Created by ChatGPT

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 13.

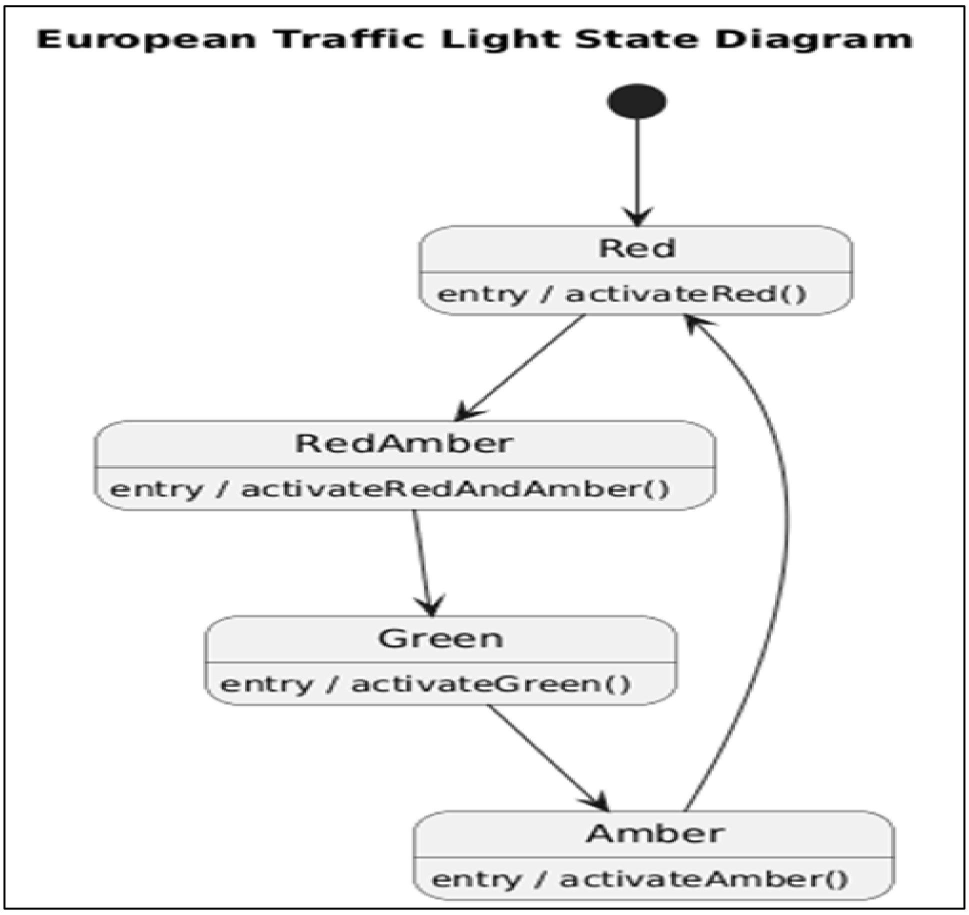


Fig 13 UML State Diagram Created by Qwen

Results obtained after submitting the prompt for an ATM.

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 14.

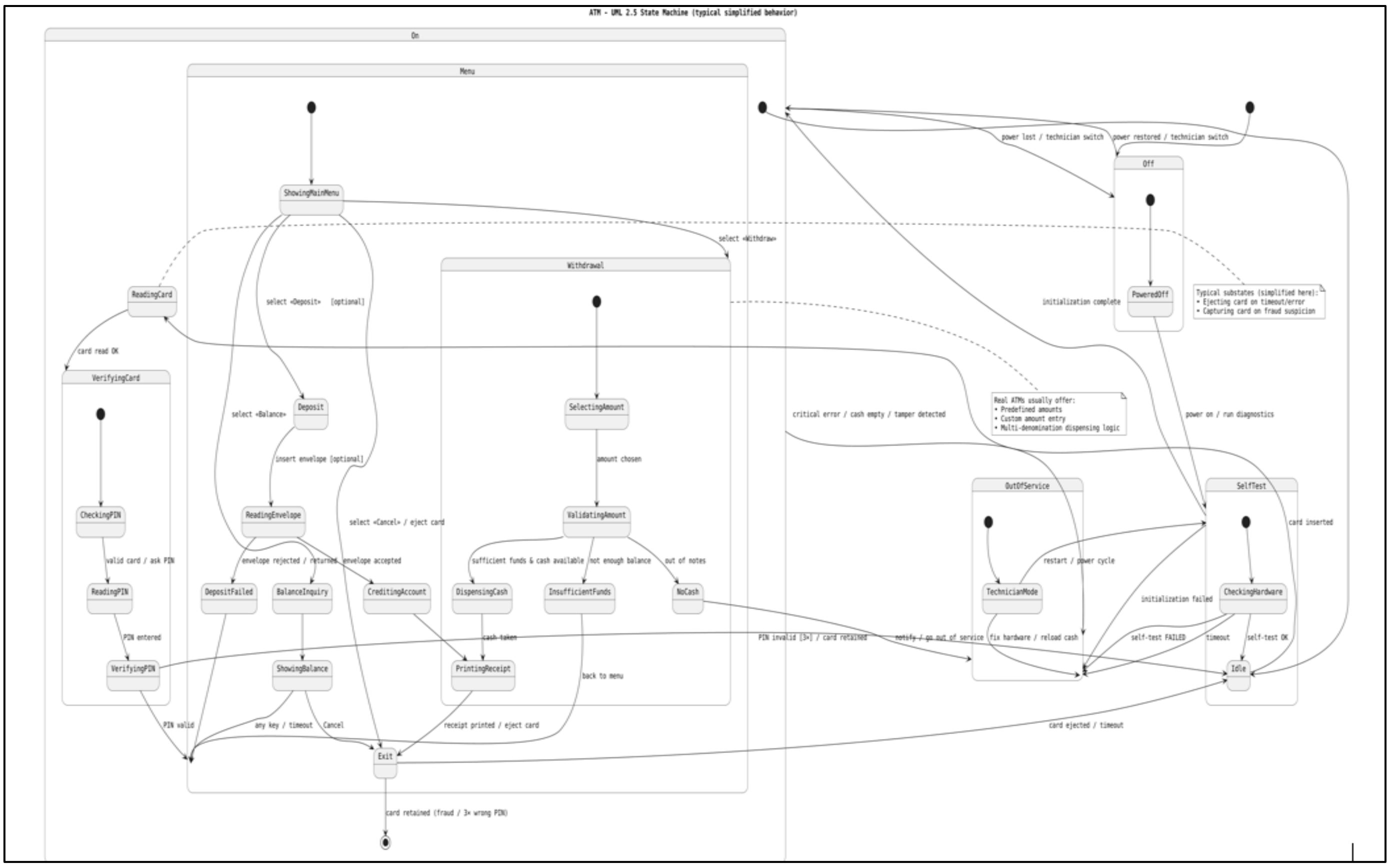


Fig 14 UML State Diagram Created by Grok

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 15.

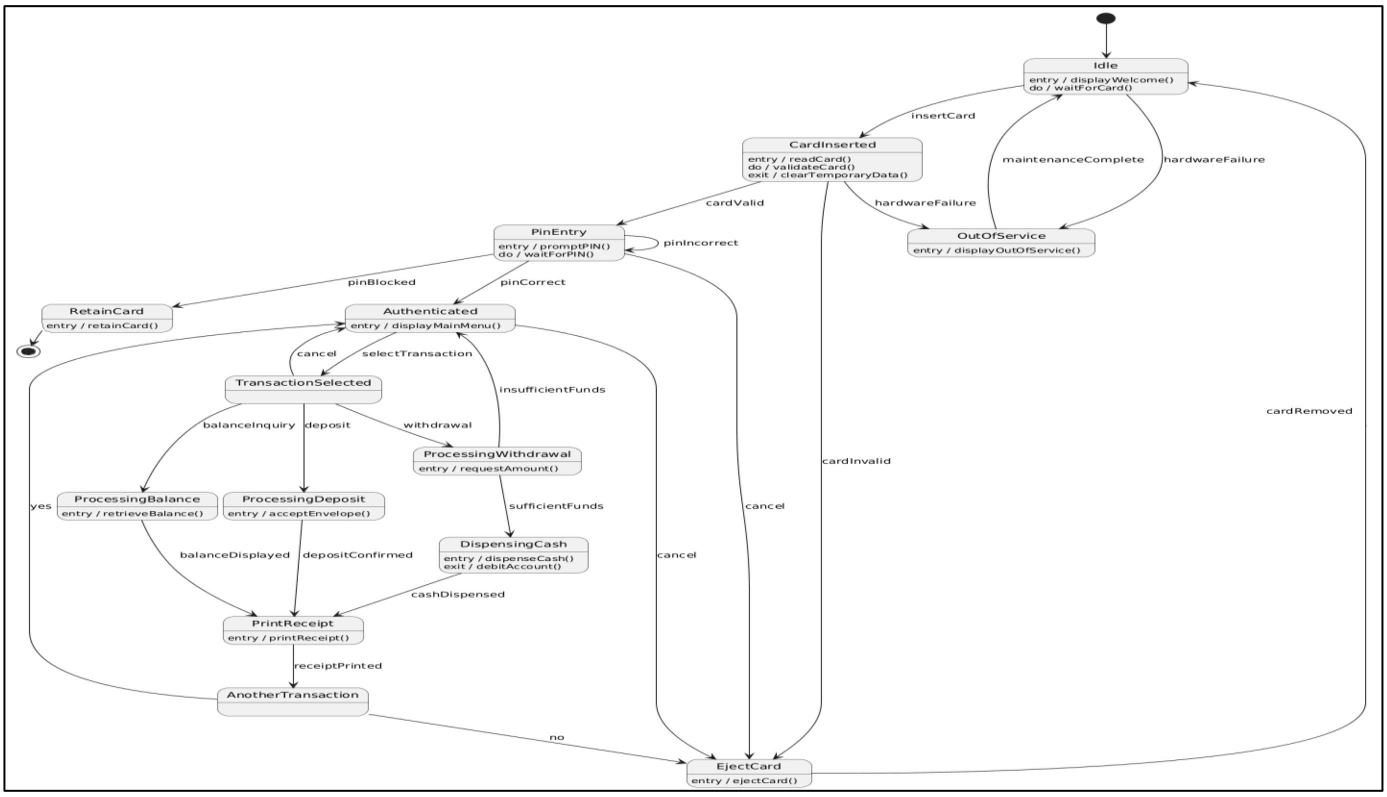


Fig 15 UML State Diagram Created by ChatGPT

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 16.

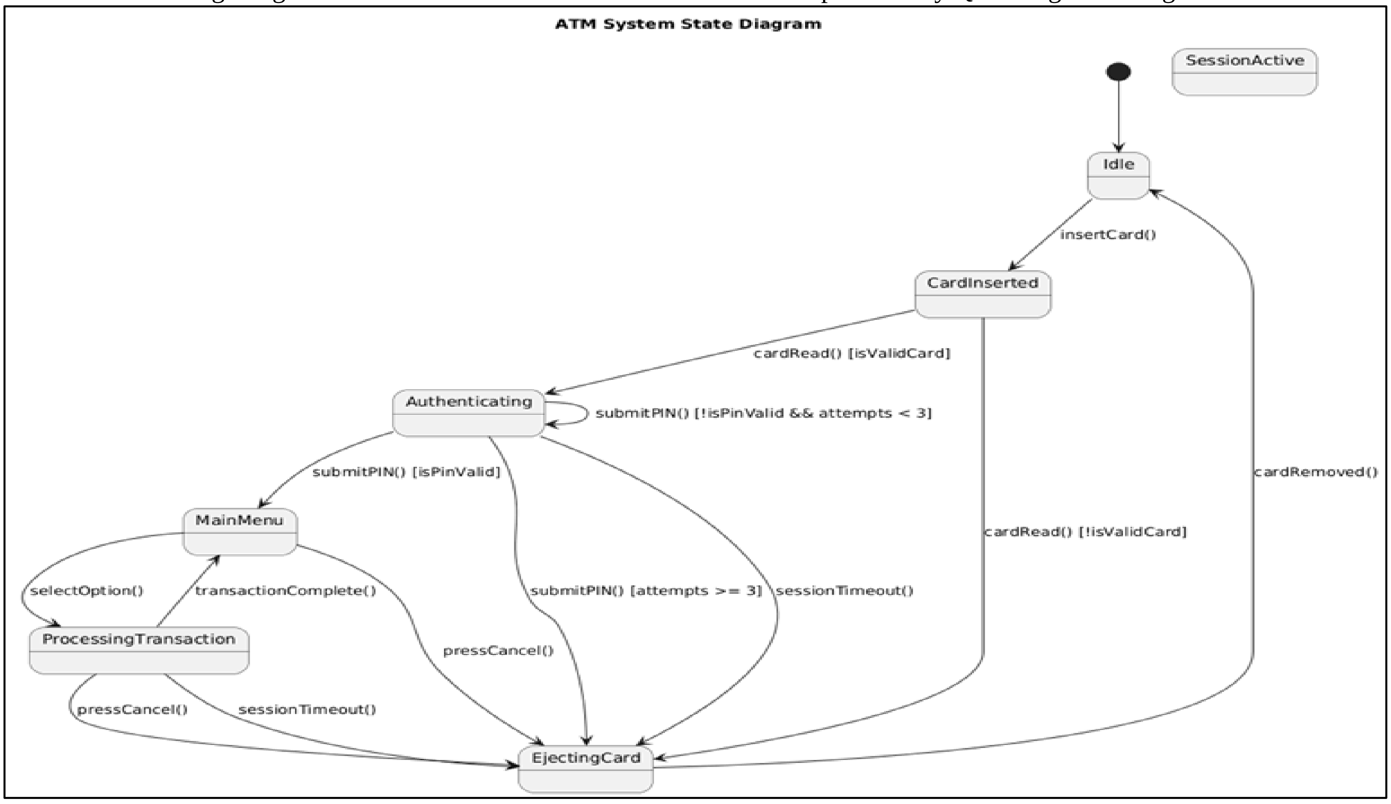


Fig 16 UML State Diagram Created by Qwen

Results obtained after submitting the prompt for a smart home lighting system.

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 17.

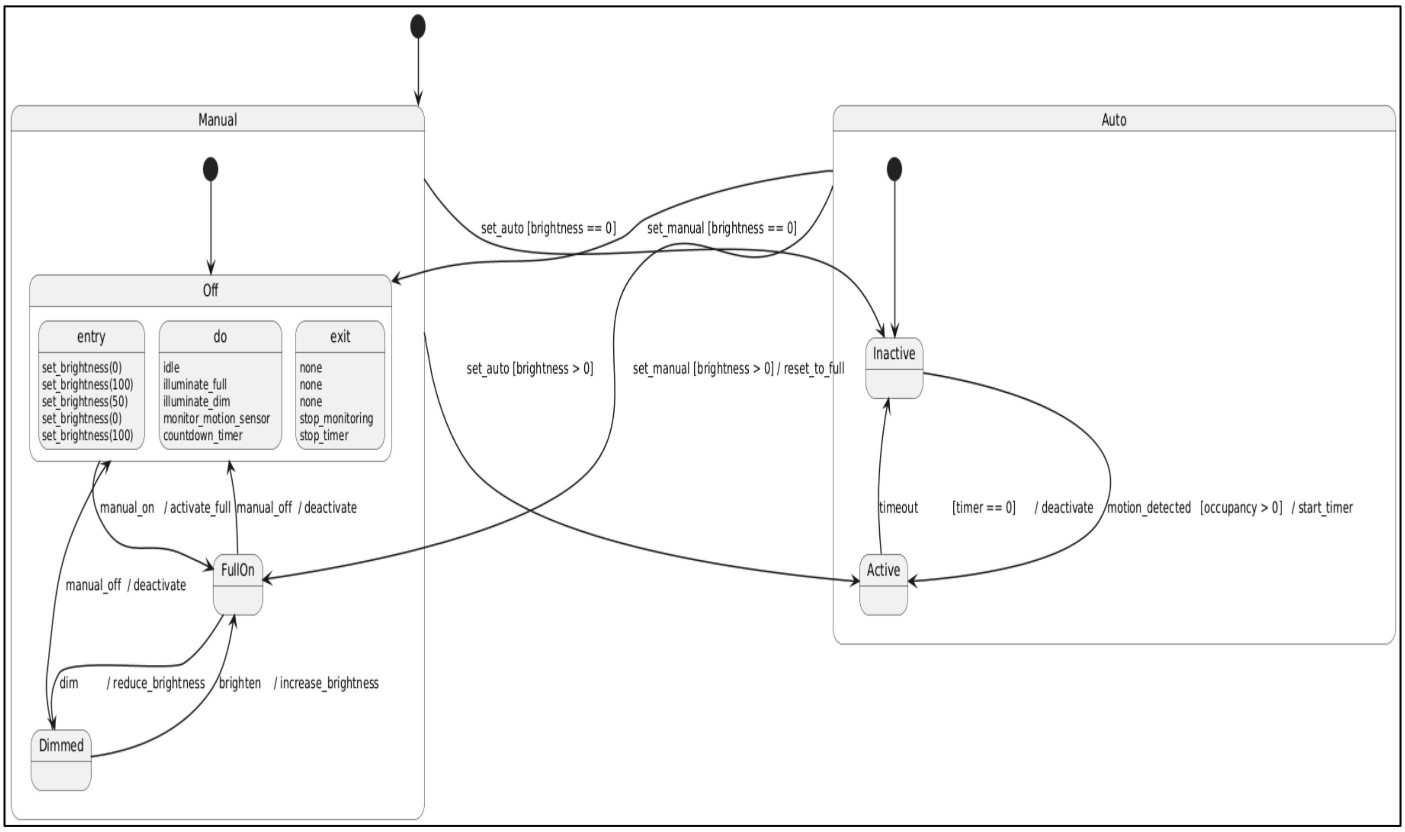


Fig 17 UML State Diagram Created by Grok

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 18.

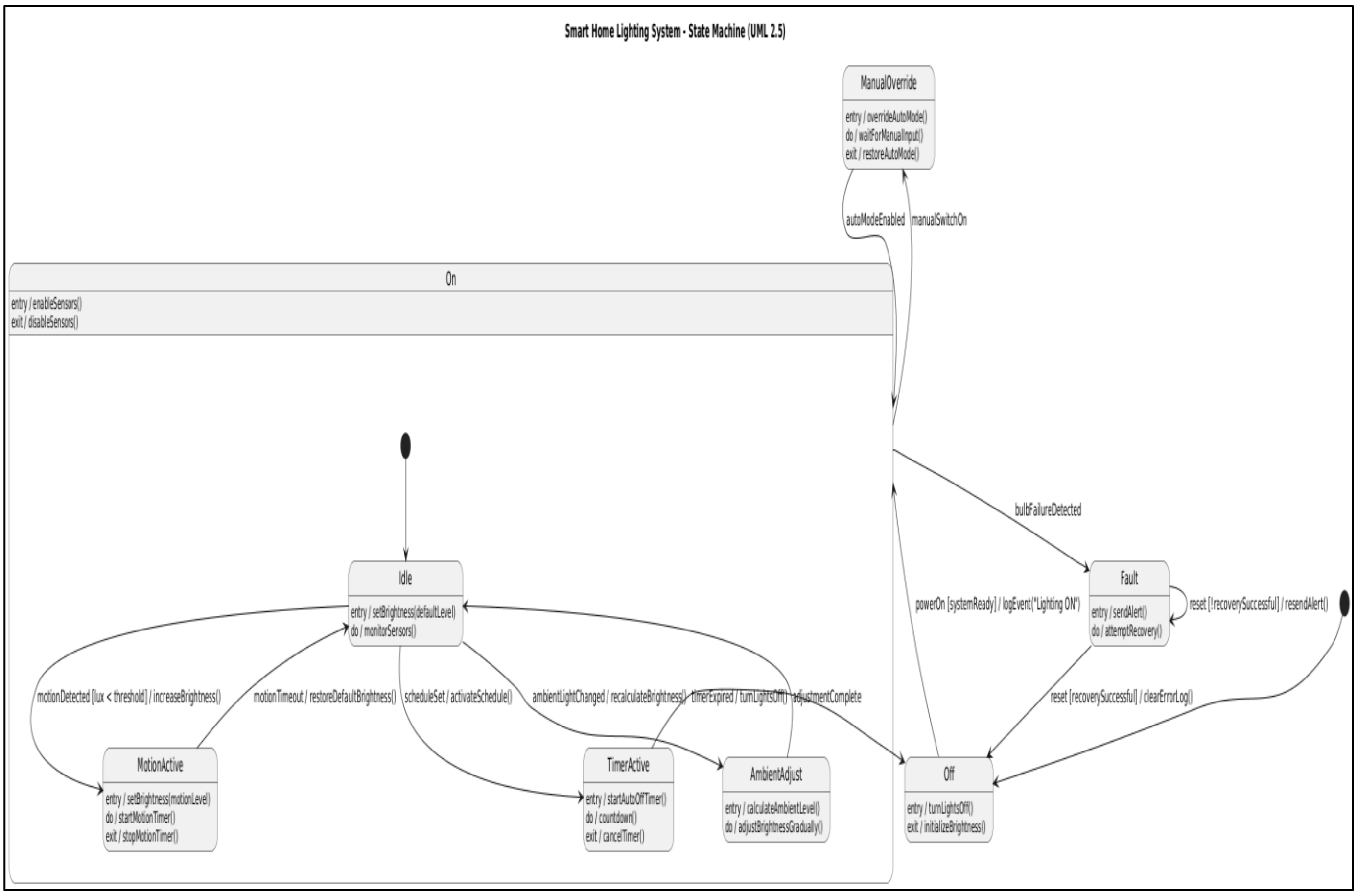

Smart Home Lighting System - State Machine (UML 2.5)
ManualOverride
entry / overrideAutoMode()
do / waitForManualInput()
exit / restoreAutoMode()
autoModeEnabled
manualSwitchOn
On
entry / enableSensors()
exit / disableSensors()
Idle
entry / setBrightness(defaultLevel)
do / monitorSensors()
motionDetected [lux < threshold] / increaseBrightness()
motionTimeout / restoreDefaultBrightness()
scheduleSet / activateSchedule()
ambientLightChanged / recalculateBrightness()
timerExpired / turnLightsOff()
adjustmentComplete
MotionActive
entry / setBrightness(motionLevel)
do / startMotionTimer()
exit / stopMotionTimer()
TimerActive
entry / startAutoOffTimer()
do / countdown()
exit / cancelTimer()
AmbientAdjust
entry / calculateAmbientLevel()
do / adjustBrightnessGradually()
bulbFailureDetected
powerOn [systemReady] / logEvent("Lighting ON")
Fault
entry / sendAlert()
do / attemptRecovery()
reset [!recoverySuccessful] / resendAlert()
reset [recoverySuccessful] / clearErrorLog()
Off
entry / turnLightsOff()
exit / initializeBrightness()

Fig 18 UML State Diagram Created by ChatGPT

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 19.

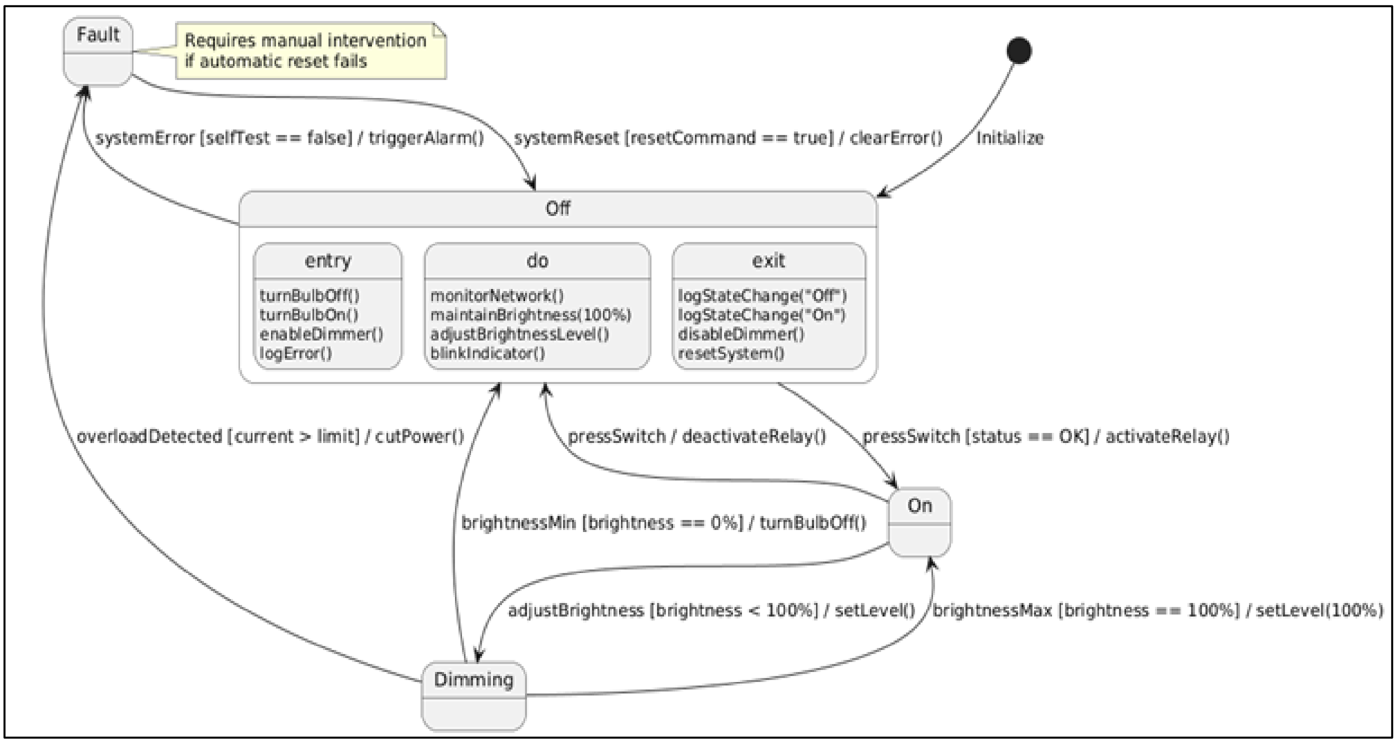

Fault
Requires manual intervention
if automatic reset fails
systemError [selfTest == false] / triggerAlarm()
systemReset [resetCommand == true] / clearError()
Initialize
Off
entry
turnBulbOff()
turnBulbOn()
enableDimmer()
logError()
do
monitorNetwork()
maintainBrightness(100%)
adjustBrightnessLevel()
blinkIndicator()
exit
logStateChange("Off")
logStateChange("On")
disableDimmer()
resetSystem()
overloadDetected [current > limit] / cutPower()
pressSwitch / deactivateRelay()
pressSwitch [status == OK] / activateRelay()
On
brightnessMin [brightness == 0%] / turnBulbOff()
adjustBrightness [brightness < 100%] / setLevel()
brightnessMax [brightness == 100%] / setLevel(100%)
Dimming

Fig 19UML State Diagram Created by Qwen

(Prompt 3) *You are a senior software modeler with 20 years of experience modeling UML diagrams. Think step by step and create a state diagram for a traffic light (according to the European standard) / an ATM / a smart home lighting system. Define all states. Identify events and transitions. Add guards, actions on input, output and during the state. Use only UML 2.5 notation. Generate only PlantUML code.*

Results obtained after submitting the prompt for a traffic light (according to European standards).
UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 20.

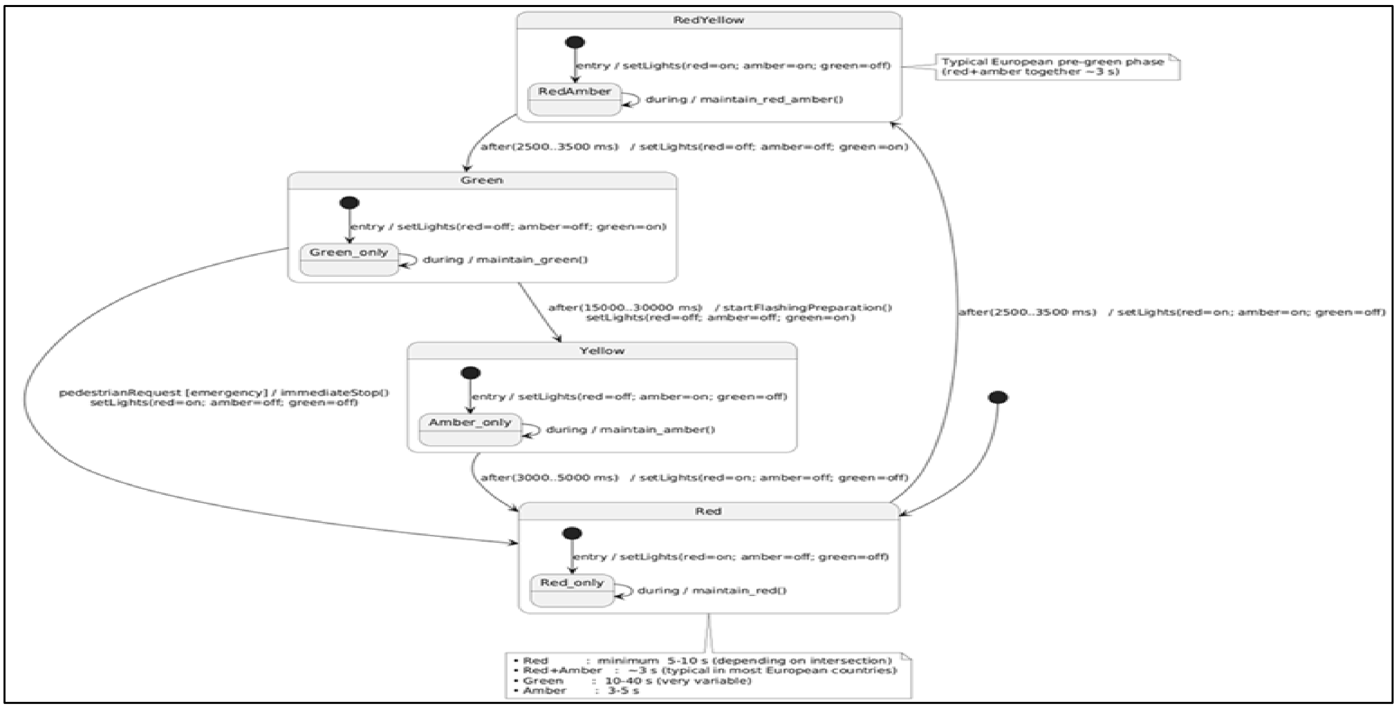

RedYellow
entry / setLights(red=on; amber=on; green=off)
RedAmber
during / maintain_red_amber()
Typical European pre-green phase (red+amber together ~3 s)
after(2500..3500 ms) / setLights(red=off; amber=off; green=on)
Green
entry / setLights(red=off; amber=off; green=on)
Green_only
during / maintain_green()
after(15000..30000 ms) / startFlashingPreparation() setLights(red=off; amber=off; green=on)
after(2500..3500 ms) / setLights(red=on; amber=on; green=off)
Yellow
pedestrianRequest [emergency] / immediateStop() setLights(red=on; amber=off; green=off)
entry / setLights(red=off; amber=on; green=off)
Amber_only
during / maintain_amber()
after(3000..5000 ms) / setLights(red=on; amber=off; green=off)
Red
entry / setLights(red=on; amber=off; green=off)
Red_only
during / maintain_red()
• Red : minimum 5-10 s (depending on intersection)
• Red+Amber : ~3 s (typical in most European countries)
• Green : 10-40 s (very variable)
• Amber : 3-5 s

Fig 20 UML State Diagram Created by Grok

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 21.

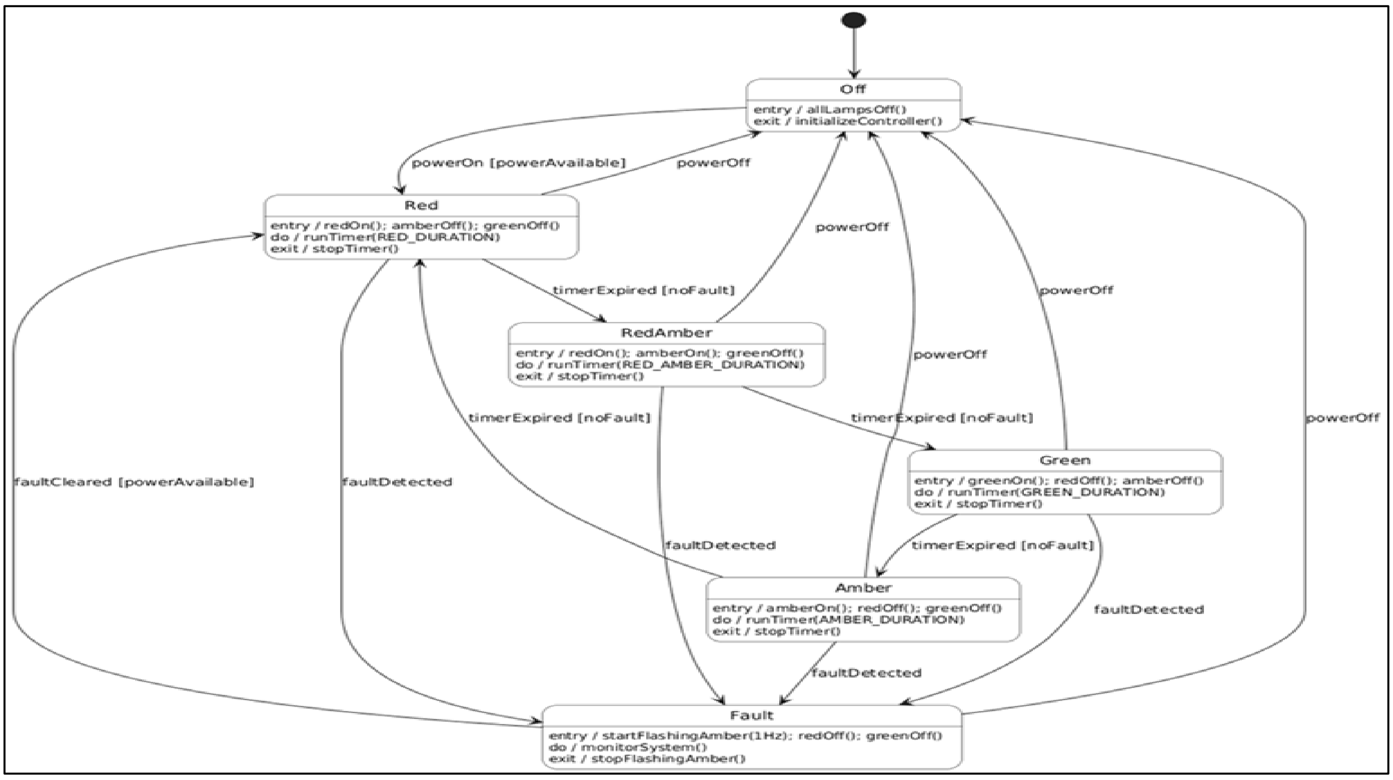

Off
entry / allLampsOff()
exit / initializeController()
powerOn [powerAvailable]
powerOff
Red
entry / redOn(); amberOff(); greenOff()
do / runTimer(RED_DURATION)
exit / stopTimer()
powerOff
timerExpired [noFault]
powerOff
RedAmber
entry / redOn(); amberOn(); greenOff()
do / runTimer(RED_AMBER_DURATION)
exit / stopTimer()
powerOff
timerExpired [noFault]
timerExpired [noFault]
powerOff
faultCleared [powerAvailable]
faultDetected
Green
entry / greenOn(); redOff(); amberOff()
do / runTimer(GREEN_DURATION)
exit / stopTimer()
faultDetected
timerExpired [noFault]
Amber
entry / amberOn(); redOff(); greenOff()
do / runTimer(AMBER_DURATION)
exit / stopTimer()
faultDetected
faultDetected
Fault
entry / startFlashingAmber(1Hz); redOff(); greenOff()
do / monitorSystem()
exit / stopFlashingAmber()

Fig 21 UML State Diagram Created by ChatGPT

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 22.

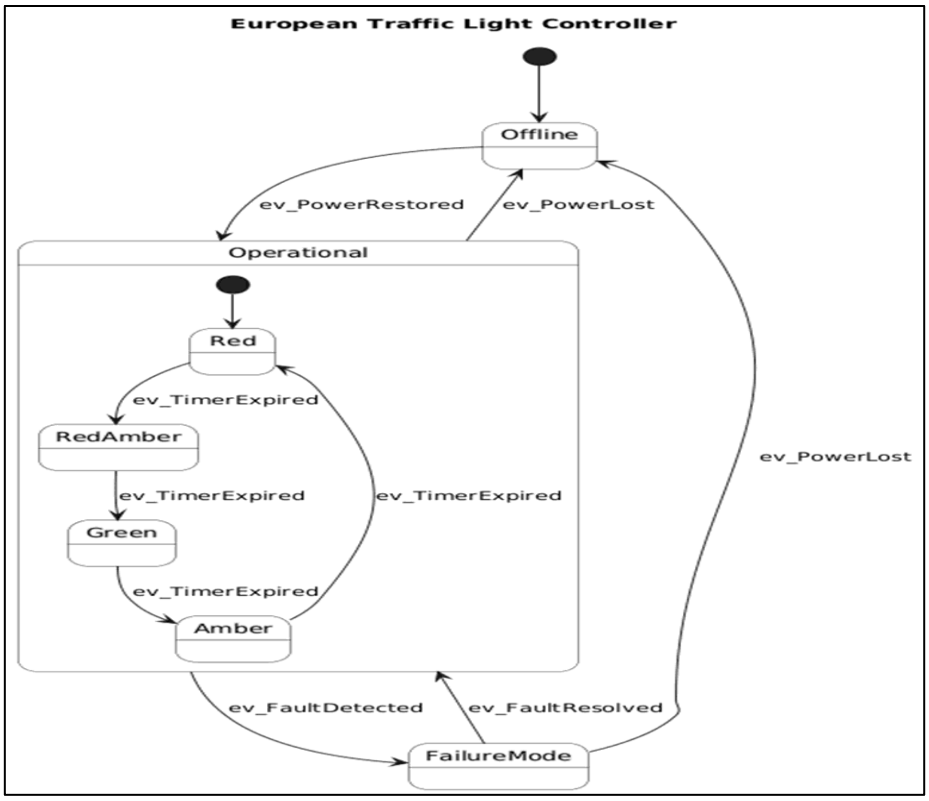


Fig 22 UML State Diagram Created by Qwen

Results obtained after submitting the prompt for an ATM.

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 23.

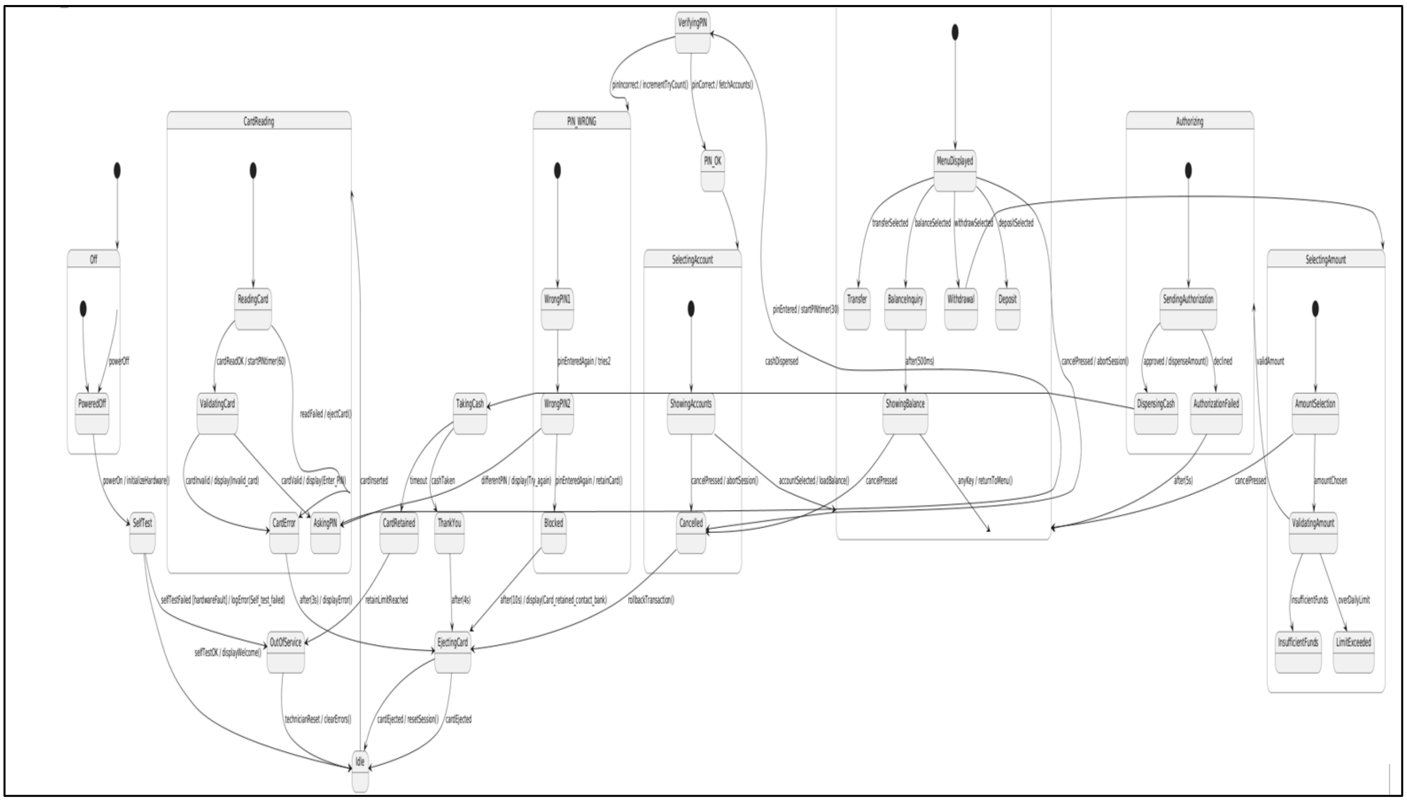


Fig 23 UML State Diagram Created by Grok

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 24.

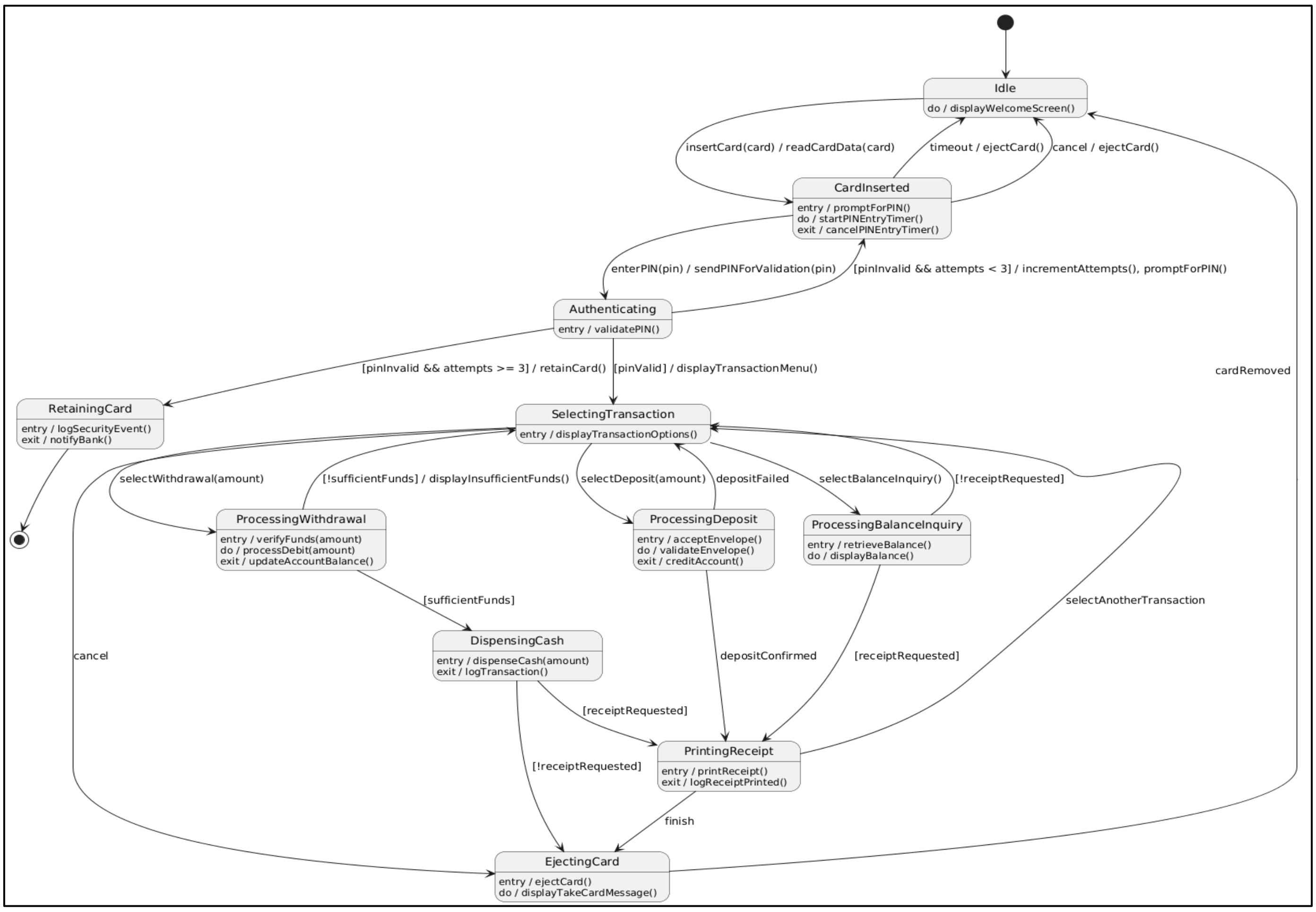


Fig 24 UML State Diagram Created by ChatGPT

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 25.

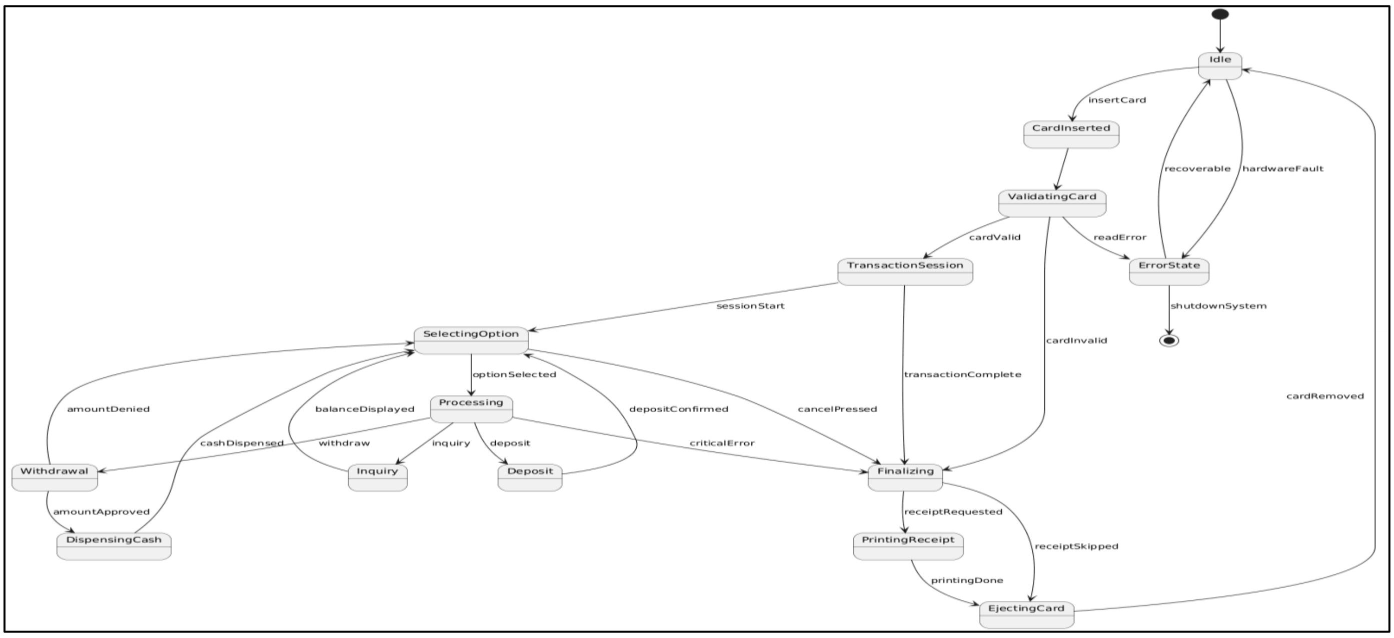


Fig 25 UML state diagram created by Qwen

Results obtained after submitting the prompt for a smart home lighting system.

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 26.

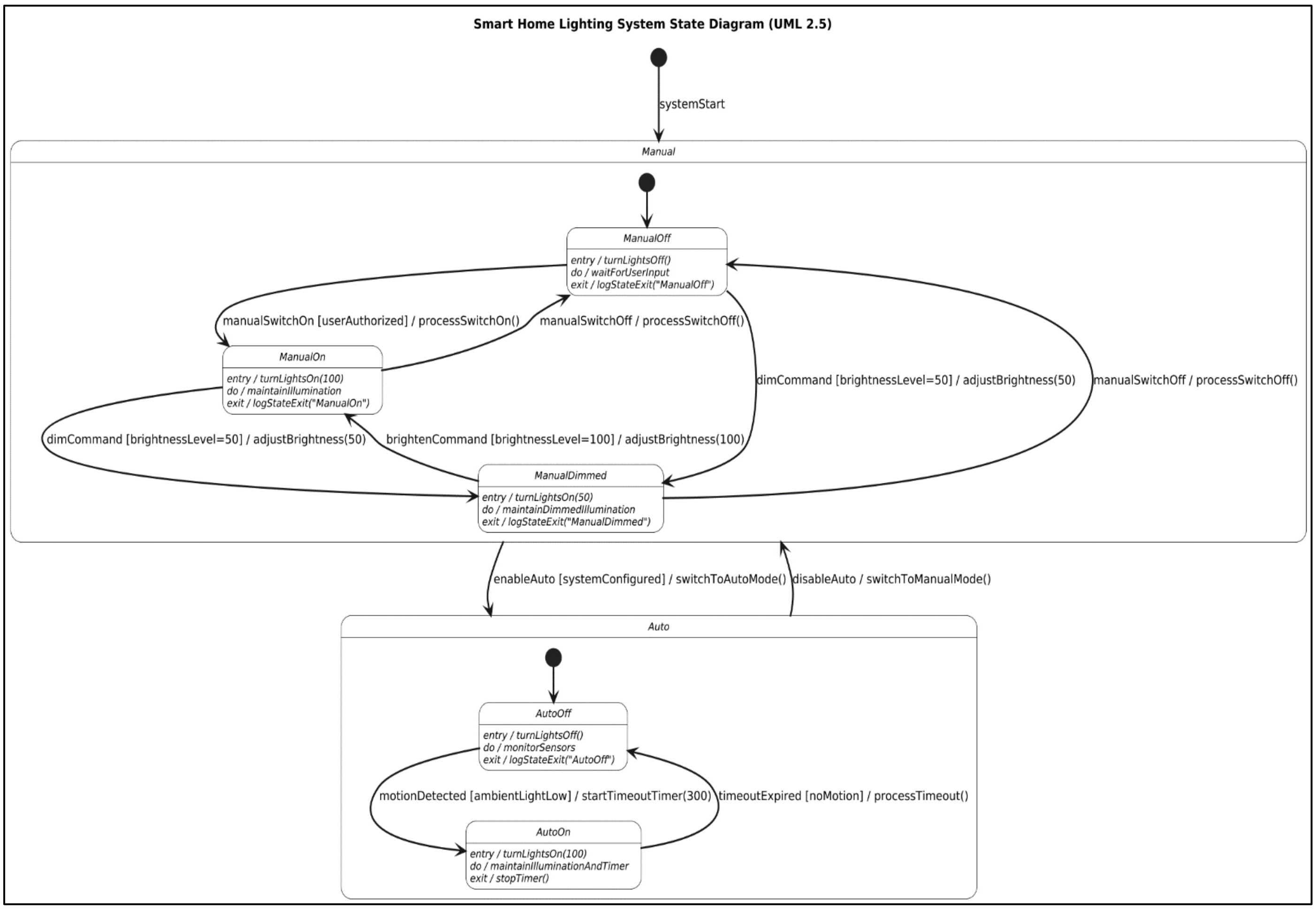

Smart Home Lighting System State Diagram (UML 2.5)
systemStart
Manual
ManualOff
entry / turnLightsOff()
do / waitForUserInput
exit / logStateExit("ManualOff")
manualSwitchOn [userAuthorized] / processSwitchOn()
manualSwitchOff / processSwitchOff()
ManualOn
entry / turnLightsOn(100)
do / maintainIllumination
exit / logStateExit("ManualOn")
dimCommand [brightnessLevel=50] / adjustBrightness(50)
manualSwitchOff / processSwitchOff()
dimCommand [brightnessLevel=50] / adjustBrightness(50)
brightenCommand [brightnessLevel=100] / adjustBrightness(100)
ManualDimmed
entry / turnLightsOn(50)
do / maintainDimmedIllumination
exit / logStateExit("ManualDimmed")
enableAuto [systemConfigured] / switchToAutoMode()
disableAuto / switchToManualMode()
Auto
AutoOff
entry / turnLightsOff()
do / monitorSensors
exit / logStateExit("AutoOff")
motionDetected [ambientLightLow] / startTimeoutTimer(300)
timeoutExpired [noMotion] / processTimeout()
AutoOn
entry / turnLightsOn(100)
do / maintainIlluminationAndTimer
exit / stopTimer()

Fig 26 UML State Diagram Created by Grok

UML state diagram generated from code in the PlantUML tool that was provided by ChatGPT is given in Figure 27.

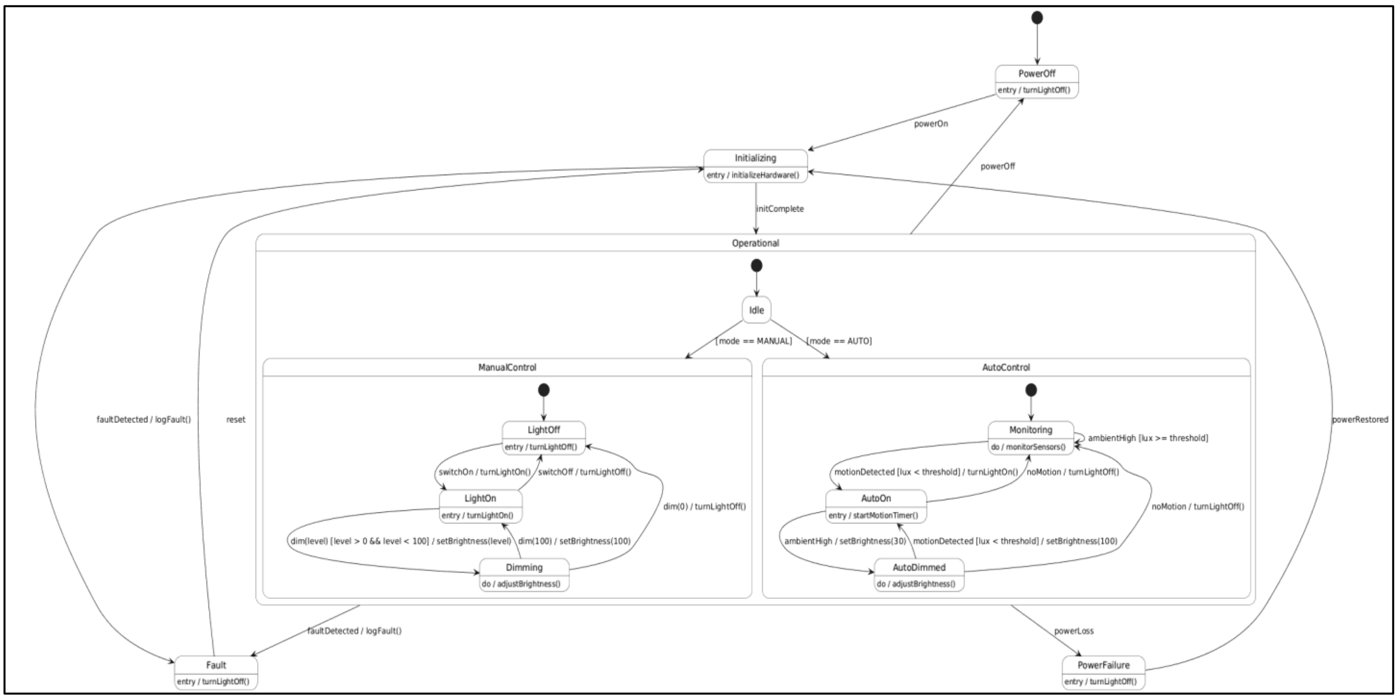

PowerOff
entry / turnLightOff()
powerOn
powerOff
Initializing
entry / initializeHardware()
initComplete
Operational
Idle
[mode == MANUAL]
[mode == AUTO]
ManualControl
LightOff
entry / turnLightOff()
switchOn / turnLightOn()
switchOff / turnLightOff()
LightOn
entry / turnLightOn()
dim(0) / turnLightOff()
dim(level) [level > 0 && level < 100] / setBrightness(level)
dim(100) / setBrightness(100)
Dimming
do / adjustBrightness()
AutoControl
Monitoring
do / monitorSensors()
ambientHigh [lux >= threshold]
motionDetected [lux < threshold] / turnLightOn()
noMotion / turnLightOff()
AutoOn
entry / startMotionTimer()
noMotion / turnLightOff()
ambientHigh / setBrightness(30)
motionDetected [lux < threshold] / setBrightness(100)
AutoDimmed
do / adjustBrightness()
faultDetected / logFault()
reset
faultDetected / logFault()
Fault
entry / turnLightOff()
powerLoss
PowerFailure
entry / turnLightOff()
powerRestored

Fig 27 UML State Diagram Created by ChatGPT

UML state diagram generated from code in the PlantUML tool that was provided by Qwen is given in Figure 28.

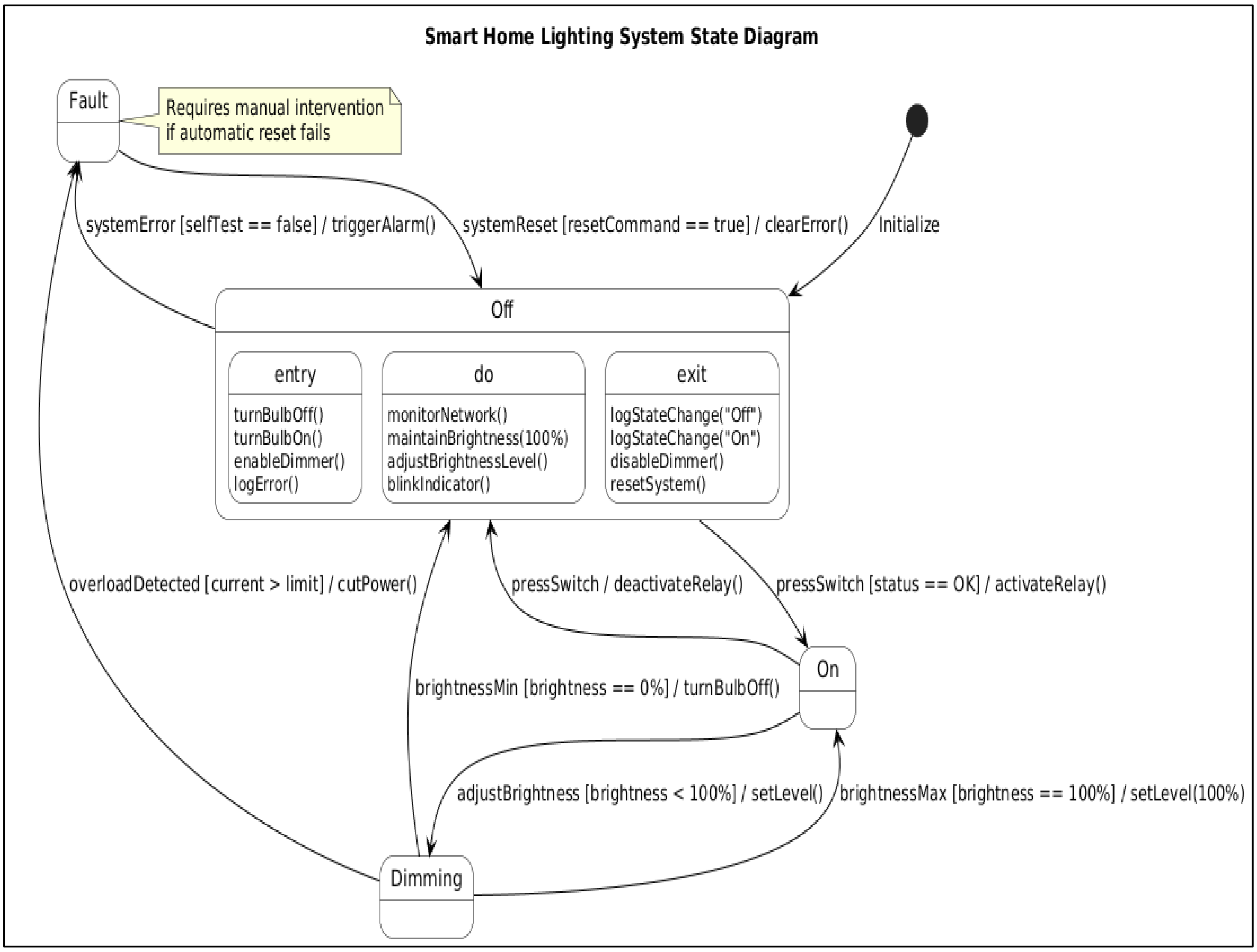
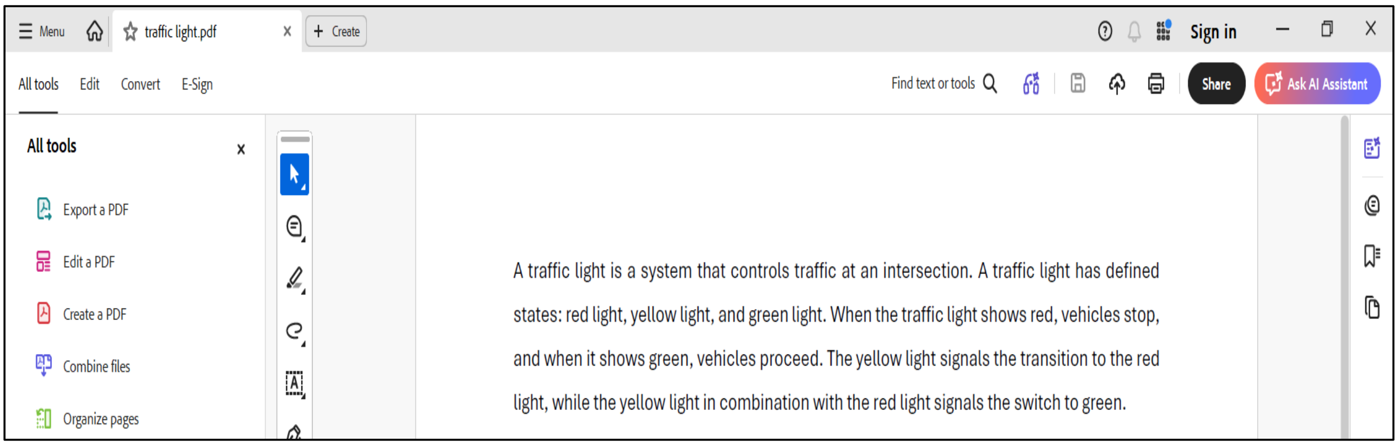

Smart Home Lighting System State Diagram
Fault
Requires manual intervention if automatic reset fails
systemError [selfTest == false] / triggerAlarm()
systemReset [resetCommand == true] / clearError()
Initialize
Off
entry
turnBulbOff()
turnBulbOn()
enableDimmer()
logError()
do
monitorNetwork()
maintainBrightness(100%)
adjustBrightnessLevel()
blinkIndicator()
exit
logStateChange("Off")
logStateChange("On")
disableDimmer()
resetSystem()
overloadDetected [current > limit] / cutPower()
pressSwitch / deactivateRelay()
pressSwitch [status == OK] / activateRelay()
On
brightnessMin [brightness == 0%] / turnBulbOff()
adjustBrightness [brightness < 100%] / setLevel()
brightnessMax [brightness == 100%] / setLevel(100%)
Dimming

Fig 28 UML State Diagram Created by Qwen

(Prompt 4) *Create an UML state diagram according to the description from the text document. Uses UML 2.5 notation exclusively. Generate only PlantUML code without additional text.*

The uploaded PDF document is shown in the figure 29.



A traffic light is a system that controls traffic at an intersection. A traffic light has defined states: red light, yellow light, and green light. When the traffic light shows red, vehicles stop, and when it shows green, vehicles proceed. The yellow light signals the transition to the red light, while the yellow light in combination with the red light signals the switch to green.

Fig 29 PDF Document with a Description of a Traffic Light (According to European Standards), Created by an Expert.

Results obtained after uploading the PDF document and submitting the prompt.

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 30.

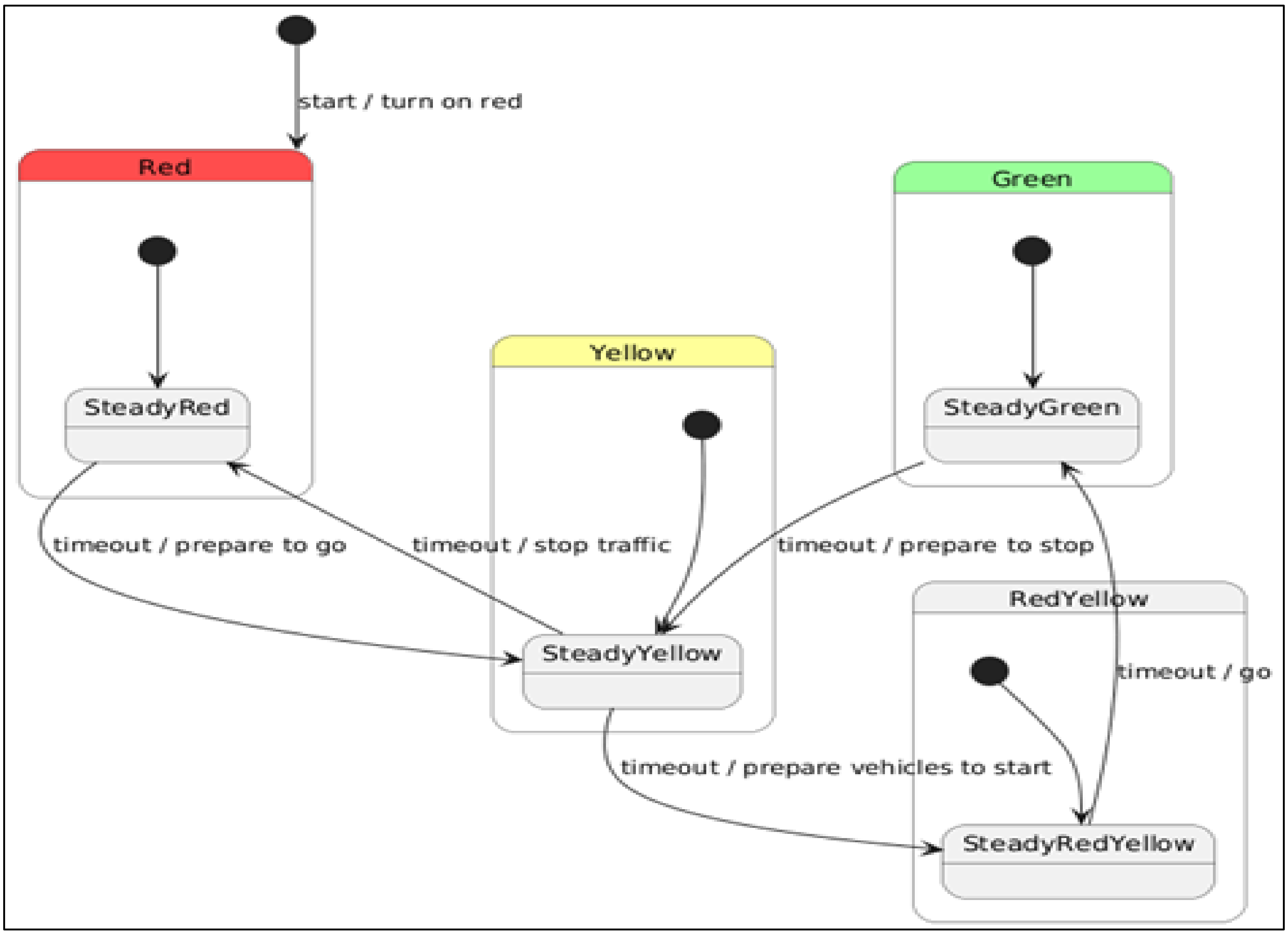
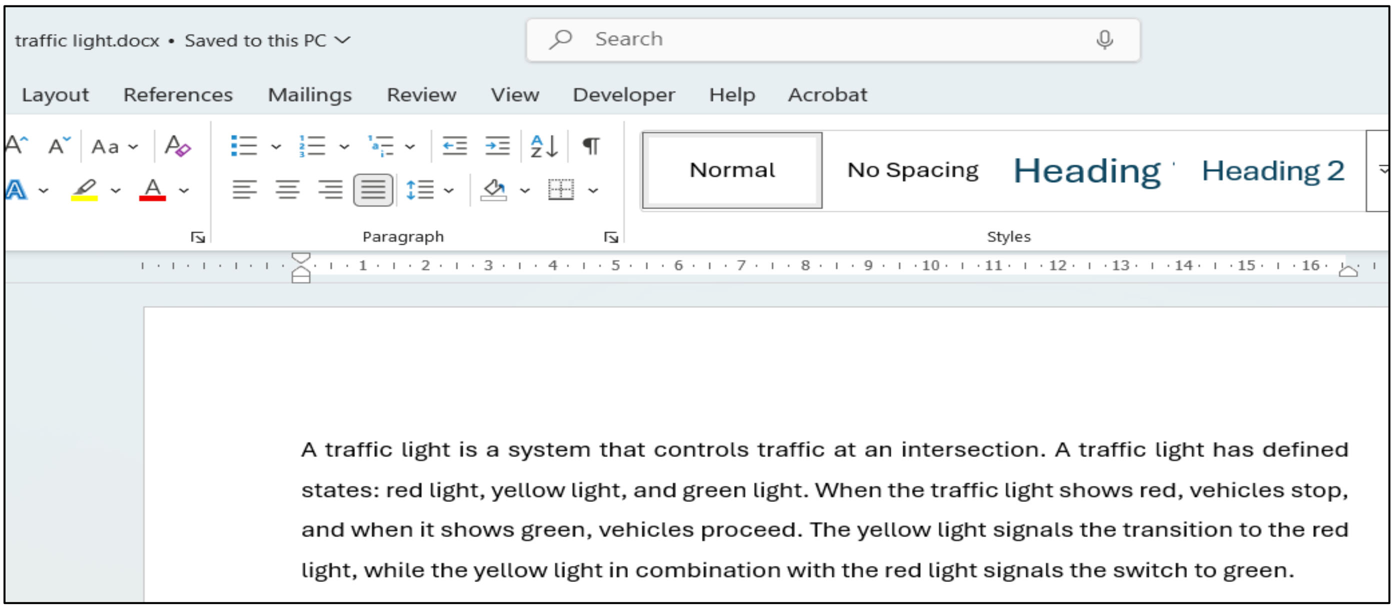


Fig 30 UML state diagram created by Grok

The uploaded Word document (.docx) is shown in the figure 31.



A traffic light is a system that controls traffic at an intersection. A traffic light has defined states: red light, yellow light, and green light. When the traffic light shows red, vehicles stop, and when it shows green, vehicles proceed. The yellow light signals the transition to the red light, while the yellow light in combination with the red light signals the switch to green.

Fig 31 Word Document with a Description of a Traffic Light (According to European Standards), Created by an Expert

Results obtained after uploading the Word document and submitting the prompt.

UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 32.

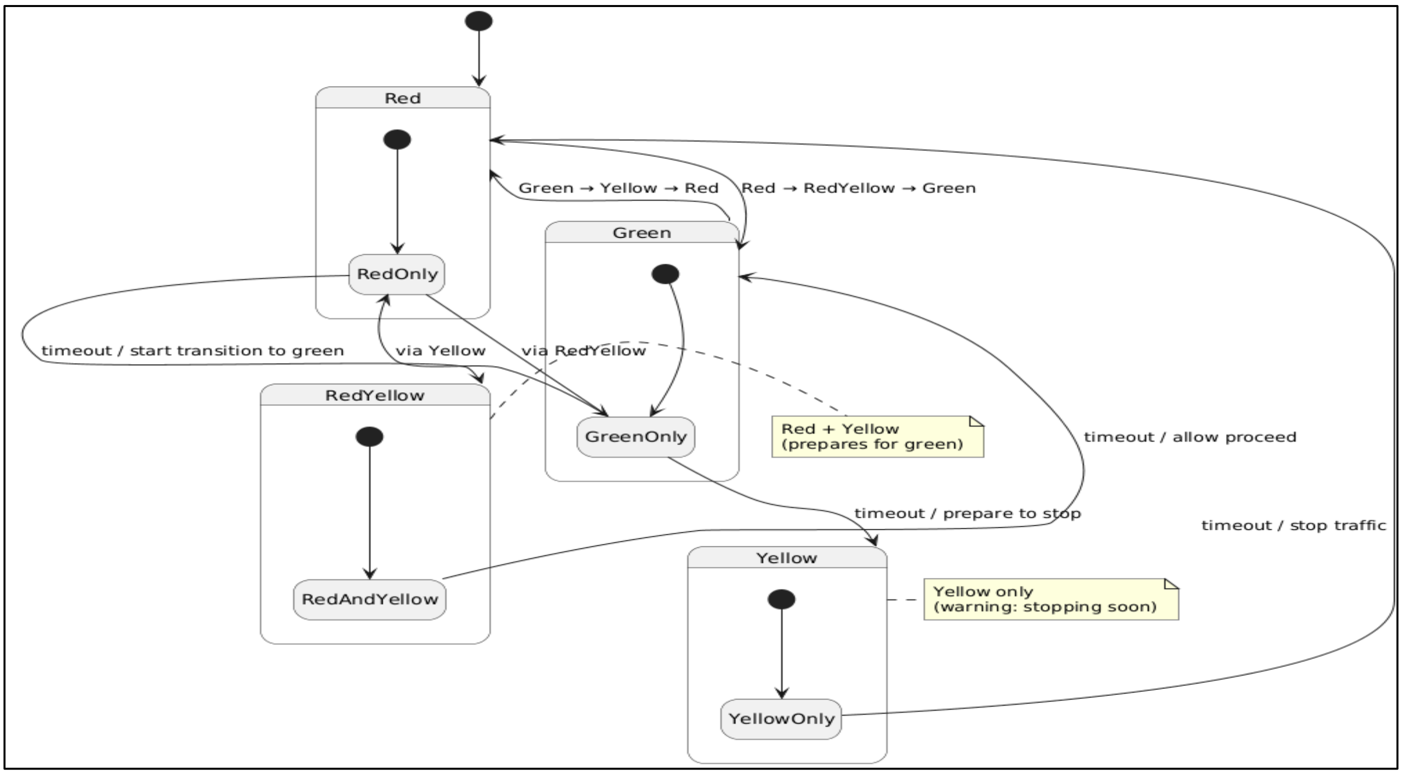


Fig 32 UML State Diagram Created by Grok

(Prompt 4.1) *Create an UML state diagram according to the text description: A traffic light is a system that controls traffic at an intersection. A traffic light has defined states: red light, yellow light, and green light. When the traffic light shows red, vehicles stop, and when it shows green, vehicles proceed. The yellow light signals the transition to the red light, while the yellow light in combination with the red light signals the switch to green. Uses UML 2.5 notation exclusively. Generate only PlantUML code without additional text.*

Result obtained from the textual description and submitting the prompt.
UML state diagram generated from code in the PlantUML tool that was provided by Grok is given in Figure 33.

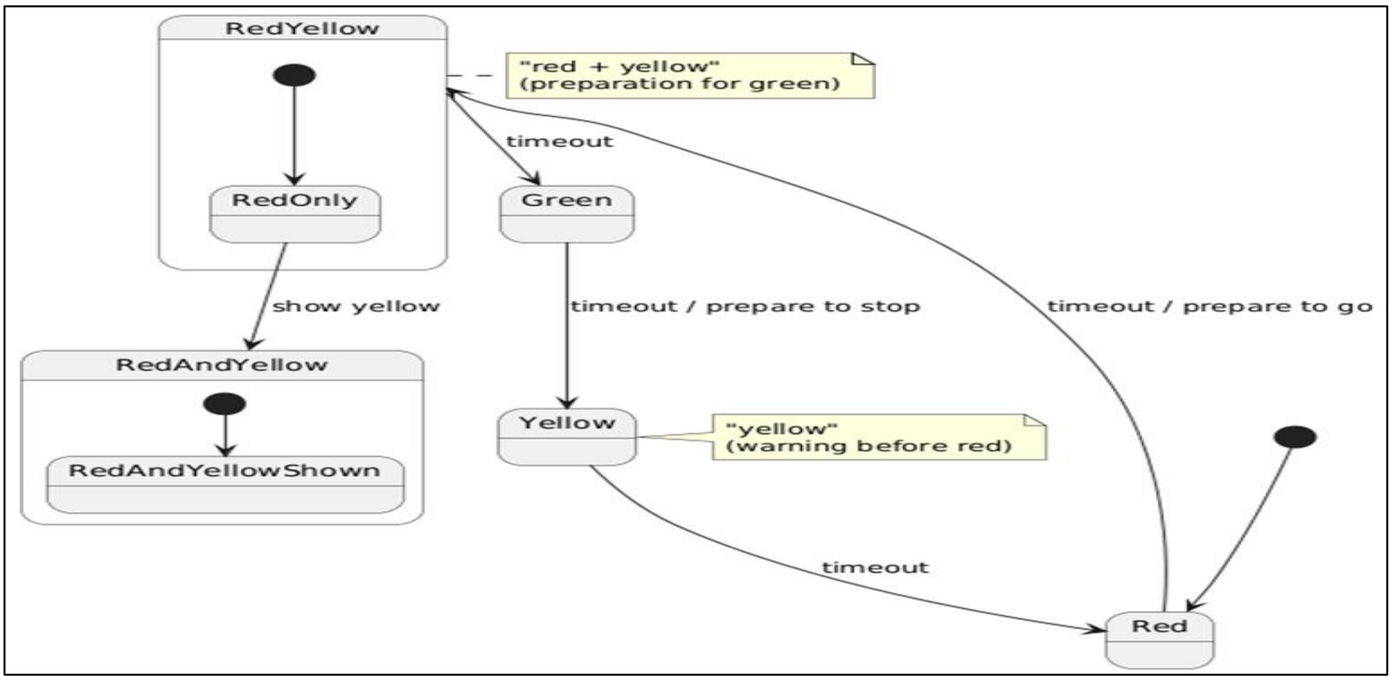


Fig 33 UML State Diagram Created by Grok

VI. RESULTS AND DISCUSSION

This study evaluated the capability of three LLMs - GPT-4 (ChatGPT 4.1), Grok (Grok 3 Full/Flagship), and Qwen (Qwen3.5-Plus) to generate UML 2.5 state machine diagrams in PlantUML format based on prompts written in natural language with varying structure and complexity. The evaluation focused on three systems of different complexity: (1) the European standard traffic light system, (2) the ATM transaction process, and (3) a smart home lighting system. The assessment covered syntactic correctness, semantic accuracy, completeness, structural complexity and code generation efficiency. Four prompt engineering strategies were applied: (1) direct instruction, (2) chain-of-thought (CoT), (3) role assignment as a senior modeler with 20 years of experience and (4) document-based prompting (textual description or uploaded .docx/.pdf documents).

For the quantitative evaluation, the following metrics were used: syntactic correctness (%) – percentage of executions without PlantUML errors, semantic completeness (1–5) – level of coverage of all functional requirements, structural complexity (1–5) – level of hierarchy and use of advanced UML constructs. Scale 1–5: 1 = very low, 2 = low, 3 = moderate, 4 = high, 5 = very high.

Table 2 Traffic Light System (Low Complexity)

Model	Syntactic Correctness (%)	Semantic Completeness	Structural Complexity
Grok3	100%	very high	low
ChatGPT 4.1	100%	very high	low
Qwen3.5	90%	moderate	moderate

Table 3 ATM System (Medium Complexity)

Model	Syntactic Correctness (%)	Semantic Completeness	Structural Complexity
Grok3	85%	high	high
ChatGPT 4.1	90%	high	high
Qwen3.5	75%	moderate	moderate

Table 4 Smart Lighting System within a Smart Home System (Highest Complexity)

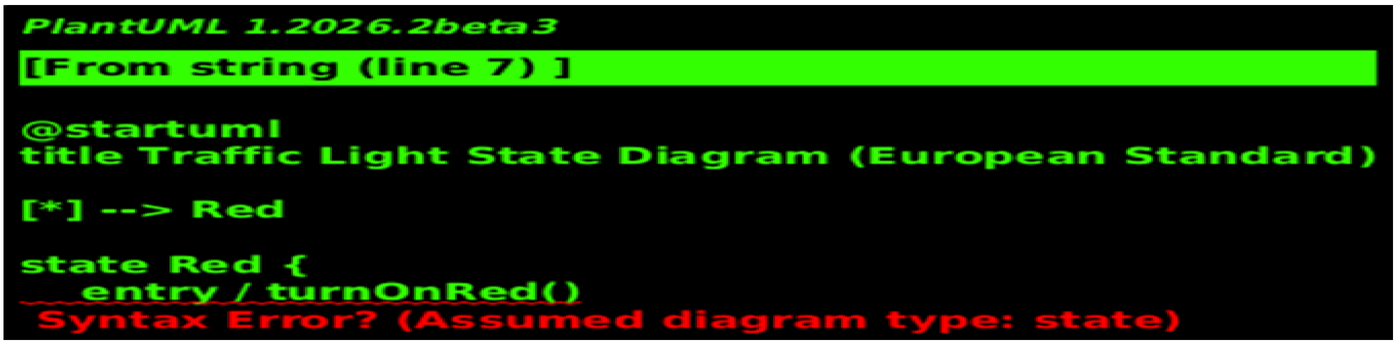
Model	Syntactic Correctness (%)	Semantic Completeness	Structural Complexity
Grok3	80%	moderate	high
ChatGPT 4.1	85%	very high	very high
Qwen3.5	70%	low	low

Table 5 Impact of Prompting Strategy (Average Across All Models)

Model	Syntactic Correctness (%)	Semantic Completeness	Structural Complexity
Direct Command	95%	moderate	low
Chain-of-Thought	90%	high	high
Role-Based Prompt	85%	very high	very high
Document-Based Prompt	95%	very high	very high

All three tested models were generally capable of producing syntactically correct PlantUML code. For the simplest example (traffic light system), syntactic correctness was achieved in most executions across all prompting strategies.

For more complex systems (ATM and smart home lighting system), syntactic errors occurred more frequently, particularly when prompts required strict adherence to UML 2.5 notation and hierarchical modeling. Composite states with substates and advanced constructs (e.g., conditional transitions), the use of Unicode operators (e.g., ≤), multiple guards per transition, pseudo-comments (e.g., [*comment*]), and multiple actions within entry/do/exit sections caused parser errors. Example of an error is given in Figure 34.


```
PlantUML 1.2026.2beta3
[From string (line 7) ]

@startuml
title Traffic Light State Diagram (European Standard)

[*] --> Red

state Red {
   entry / turnOnRed()
 Syntax Error? (Assumed diagram type: state)
```
Fig 34 The Error Generated by the PlantUML Parser from the Code Produced by Qwen

When a syntactic error occurred, iterative correction based on feedback from the PlantUML parser significantly improved the quality of the result. This demonstrates that the models are responsive to structured corrective instructions and can effectively refine previously generated code.

Semantic accuracy varied significantly depending on the model, prompting strategy, and system complexity.

In the traffic light example, all models correctly identified the fundamental states (Red, Yellow, Green) and their cyclic sequence. Differences emerged in: the representation of the transitional Red+Yellow state, the modeling of time-triggered events, and the representation of flashing behavior. Grok more frequently included domain-specific details (e.g., time intervals and explanatory notes), while ChatGPT generated structurally cleaner but sometimes simplified models. Qwen occasionally omitted transitional states.

In the ATM example, semantic variability was more pronounced. All models recognized key phases such as card insertion, authentication, transaction selection, processing, and termination. However, model completeness varied: some solutions did not include error-handling states, conditions related to the number of PIN entry attempts were not always properly modeled, and the logic for real-time account balance updates was sometimes simplified.

The smart lighting system proved to be the most demanding example. Although all models correctly identified the basic states (Off, On, Dimmed, Automatic), only some solutions fully captured: sensor-triggered transitions, time-based automation rules, and continuous light intensity adjustment.

Completeness was strongly dependent on the prompting strategy. A direct prompt (e.g., *“Create a state diagram…”*) resulted in concise but often minimal models. Basic states were included, but extended logic (guards, entry/exit actions, automation rules) was frequently omitted. In most Grok outputs, initial and final states were present, whereas in ChatGPT and Qwen outputs they were often omitted. The chain-of-thought approach significantly improved completeness. When explicitly instructed to define all states, events, transitions, and actions, the models generated richer and more detailed diagrams. Prompts that assigned a role (e.g., an experienced modeler with 20 years of experience) produced the most structurally elaborate diagrams. Such results typically included: entry and exit actions, conditional transitions, and a more formal UML structure. Role assignment (“senior software modeler with 20 years of experience”) and chain-of-thought prompting significantly improved semantic accuracy and completeness across all models, particularly Grok and ChatGPT. However, increased structural complexity was also associated with a higher likelihood of syntactic errors. Simple diagrams without hierarchical structures were almost always executable, whereas more complex hierarchical models often generated code that was not fully compatible with the tested version of PlantUML. Document-based prompts and detailed textual descriptions produced the most faithful models.

Grok provided explicit performance metrics (time to first token and total response time) and consistently delivered complete responses. The response times of ChatGPT and Qwen were measured manually and were comparable to Grok’s in simpler cases. However, response time in Qwen outputs increased significantly for the smart lighting system scenario. The average time required by the tested LLMs to generate the UML state diagram is shown in the figure 35.

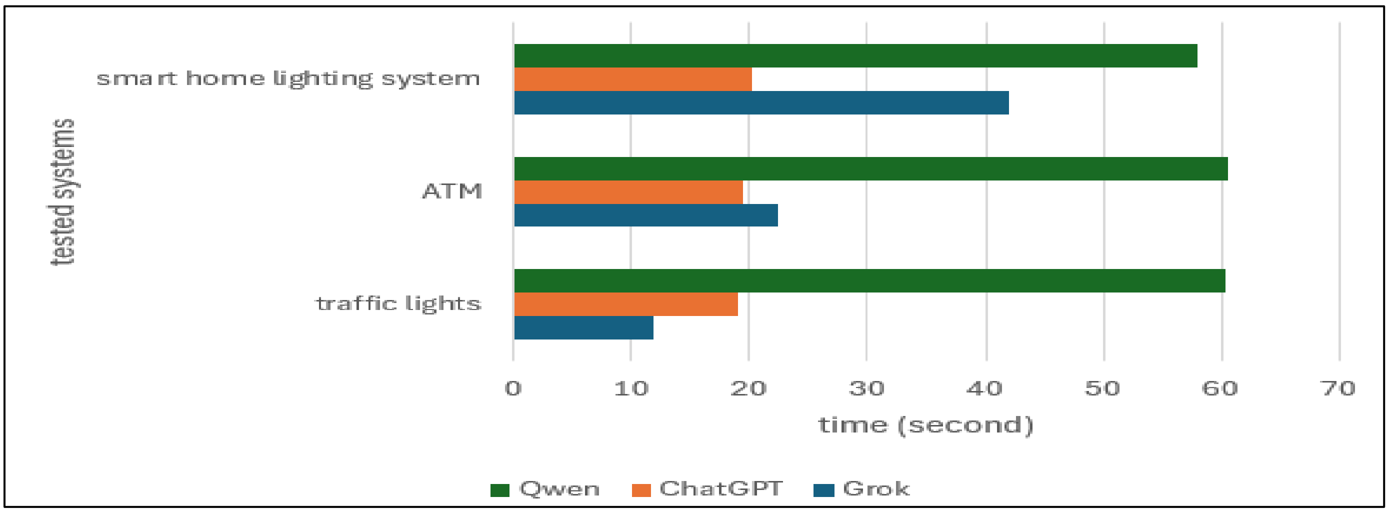


Fig 35 The Average Time Required to Generate a UML State Diagram for the Tested LLMs (Grok, ChatGPT, Qwen)

The results indicate that the tested LLMs are capable of effectively generating UML 2.5 state machine diagrams in PlantUML format, particularly for systems of low and medium complexity.

The advantages of LLMs include rapid model generation based on textual descriptions, the ability to incorporate guards and actions when explicitly instructed and adaptability based on feedback.

However, several limitations were observed, including reduced semantic precision in complex systems, occasional syntactic instability in hierarchical models and variability across repeated executions.

The results of this evaluation (conducted in February 2026) show that GPT-4 outperforms Grok3 and Qwen3.5-Plus in generating accurate, complete, and syntactically robust UML state machine diagrams from natural language descriptions.

Overall, the findings suggest that LLMs are suitable as supportive tools in modeling rather than fully autonomous tools for UML design. Human validation remains essential, especially for complex systems.

VII. CONCLUSION

The results of the conducted research indicate that the analyzed LLMs (ChatGPT, Grok and Qwen) are capable of generating UML state diagrams in PlantUML format with a high level of syntactic correctness, particularly for systems of low and medium complexity. For simple systems, such as a traffic light, all models in most cases generated a syntactically correct and semantically accurate model.

As system complexity increased, which is the case with ATM and smart home lighting system, higher number of syntactic errors and greater variability in the semantic completeness of the models were observed. Hierarchical states, multiple guards, complex actions within entry/exit/do sections, and specific UML 2.5 constructs posed challenges, especially when combined with the strict requirement to generate exclusively PlantUML code without additional text.

The impact of prompt design strategy proved to be highly significant. Direct commands resulted in shorter and simpler models, while chain-of-thought and role-based approaches led to semantically richer and structurally more complex diagrams. Document-based prompts produced the most faithful models in relation to the original description, but they also increased the risk of syntactic deviations.

Comparatively, the GPT-4.1-based model achieved the most balanced results in terms of syntactic stability, semantic completeness, and structural elaboration of the models. Nevertheless, no model can be considered a fully autonomous tool for formal UML modeling without human verification.

In conclusion, LLMs represent a powerful supportive tool in the conceptual modeling phase and in the rapid creation of initial UML diagram versions; however, expert human validation remains necessary, particularly for complex and safety-critical systems.

REFERENCES

[1]. Nguyen, V.-V., Nguyen, H.-K., Nguyen, K.-S., Loung, T.M.-H., Vu, D.-Q., Phung, T.-N., and Nguyen, T.-V, 2026. A Novel Unified Framework for Automated Generation and Multimodal Validation of UML Diagrams. *Computer Modeling in Engineering and Science*, *146*(1), https://doi.org/10.32604/cmes.2025.075442

[2]. Cámara, J., Troya, J., Burgueño, L. and Vallecillo, A., 2023. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling*, *22*(3), pp.781-793.

[3]. Al-Ahmad, B., Alsobeh, A., Meqdadi, O. and Shaikh, N., 2025. A Student-Centric Evaluation Survey to Explore the Impact of LLMs on UML Modeling. *Information*, *16*(7), p.565.

[4]. Object Management Group, 2017. *Unified Modeling Language (UML), version 2.5.1.* https://www.omg.org/spec/UML/2.5.1/About-UML

[5]. Basic, M., & Vujasinovic, M., 2026. Usage of LLM for Generation of UML Class Diagrams from UML Use-Case Diagrams. *International Journal of Innovative Science and Research Technology*, *11(1)*, https://doi.org/10.38124/ijisrt/26jan1576

[6]. Jahan, M., Hasan, MM., Golpayegani, R., Roy, C. and Roy, B., 2024. Automated derivatio of UML sequence diagrams from user stories: unleashing the power of generative AI vs. rule-based approach. *In Proceedings of the ACM/IEEE 27th International Coference on Model Driven Engineering Languages and Systems (MODELS '24)*, p.p.138-48. https://doi.org/10.1145/3640310.3674081