Smart Home Simulator

(C++ Object Oriented Programming Project)

Kishan N Prasad* Shresta P Nayaka† Mehul Krishna‡
Kapil Kumar§
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY VADODARA,
INTERNATIONAL CAMPUS DIU

November 18, 2025

Abstract

This report documents the design, implementation and testing of a **Smart Home Simulator** implemented in C++ using Object Oriented Programming (OOP) principles. The simulator models virtual devices (Lights, Fans, ACs, Doors, Sensors, Vehicles, Media) and provides features including device groups/zones, automation rules (if-then), scheduling, energy tracking, simulated sensors, operator overloading, and persistence (save/load). The project demonstrates inheritance, encapsulation, polymorphism, dynamic memory management, file I/O, exception handling and operator overloading.

Contents

*Roll No.: 202411054
†Roll No.: 202411089
‡Roll No.: 202411061
§Roll No.: 202411050

1 Introduction

Modern smart homes require flexible, testable, and maintainable software that maps real-world devices to code. This project implements a command-line Smart Home Simulator focused exclusively on OOPs. The simulator is built in modular header/source files in C++ and uses a vector-based runtime store and a single persistence file to save the system state on exit.

2 Objectives

- Implement core OOP concepts (inheritance, polymorphism, encapsulation, abstraction).
- Provide real-life features: device control, groups/zones, scheduling, automation, simulated sensors.
- Track energy usage per device and persist the configuration between runs.
- Demonstrate advanced concepts: operator overloading, friend classes, exception handling.

3 System Architecture and UML

The system is organized into modular headers and a main file. Core classes include:

```
SmartController
  |-- appliances
  |     |-- Light
  |     |-- Fan
  |     |-- AirConditioner
  |     |-- WashingMachine <-> friend Dishwasher
  |     |-- Refrigerator
  |     |-- SmartTV
  |     |-- Speaker
  |-- sensors
  |     |-- TemperatureSensor
  |     |-- MotionSensor
  |     |-- HumiditySensor
  |     |-- rainSensor
  |     |-- Thermostat
  |-- doors
  |     |-- SmartDoor
  |     |-- GarageDoor
  |-- vehicles
  |     |-- Car
  |     |-- Bicycle
  |-- media (subset of Appliance)
  |     |-- SmartTV
  |     |-- Speaker
  |-- (CustomDevice slot, if needed)
```

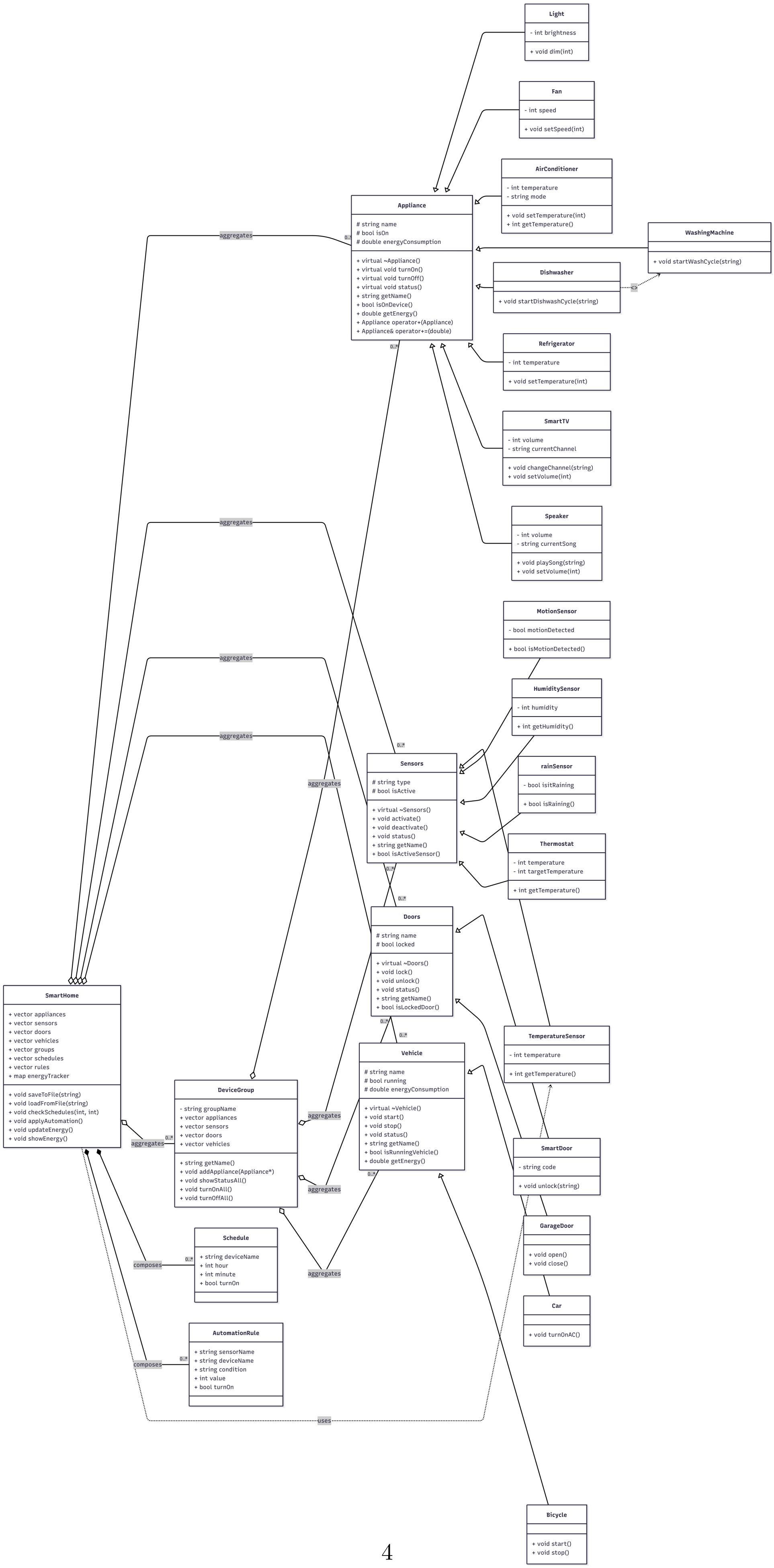

Figure 1: System UML diagram (Class Diagram)

4 Use Case Diagram

Actors: **Admin/User** (single CLI user). Main use-cases implemented:

- Add / Remove devices
- Control device (on/off, set properties)
- Create / control Device Groups
- Add Automation Rules (if sensor cond → device action)
- Add Scheduling entries
- Show status, show energy usage
- Save/load persistent state

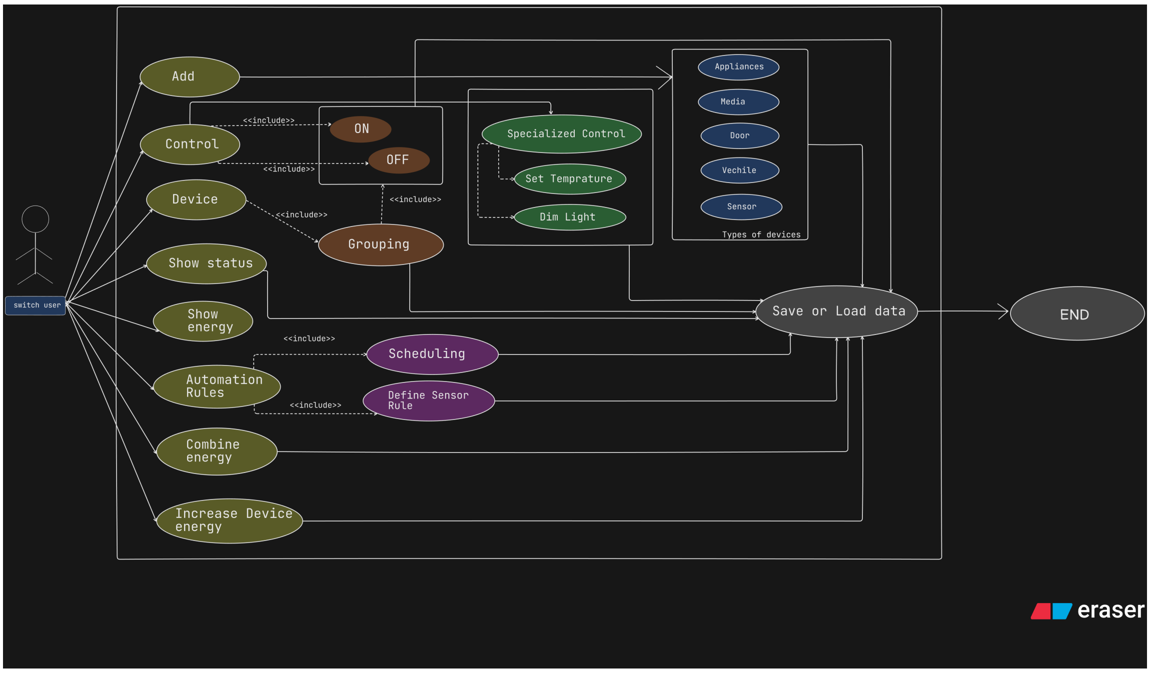

Figure 2: Use case diagram

5 Class Design and Functionality

Below is a brief on major classes and their responsibilities.

5.1 Appliance (Base class)

File: `classappliances.h`
Attributes: `name`, `isOn`, `energyConsumption`.
Methods: `turnOn()`, `turnOff()`, `status()`, getters/setters.
Operator overloads: `operator+` (combine energies), `operator+=` (increase energy).

5.2 Derived Appliances

- **Light**: brightness control (`dim(int)`).
- **Fan**: speed control (`setSpeed(int)`).

- **AirConditioner**: temperature and mode (`setTemperature()`, `setMode()`).
- **WashingMachine / Dishwasher / Refrigerator**: appliance-specific operations.

5.3 Sensors

File: `classsensors.h`
Types: TemperatureSensor, MotionSensor, HumiditySensor, RainSensor, Thermostat.
Each has `activate()`, `deactivate()`, `readData()` (or getters). Temperature and humidity sensors support simulated/random readings.

5.4 Doors

File: `classdoors.h`
Types: Doors (base), SmartDoor (code-protected unlock), GarageDoor (open/close). Methods for lock/unlock, status.

5.5 Vehicles

File: `classvehicles.h`
Basic start/stop/status plus energy consumption storage.

5.6 DeviceGroup

Groups multiple devices (appliances, sensors, doors, vehicles) and supports `turnOnAll()`, `turnOffAll()`, `showStatusAll()`.

5.7 SmartHome (Controller)

Holds vectors of pointers to all devices and manages:

- Add / Remove devices at runtime (vectors).
- **Persistence:** `saveToFile()` and `loadFromFile()` interact with a single text file `smarthome.txt`.
- **Scheduling:** vector of `Schedule` entries checked each main loop iteration.
- **Automation Rules:** vector of `AutomationRule` entries evaluated periodically.
- **Energy Tracking:** map accumulating energy per device when active.

6 Sequence and State Diagrams

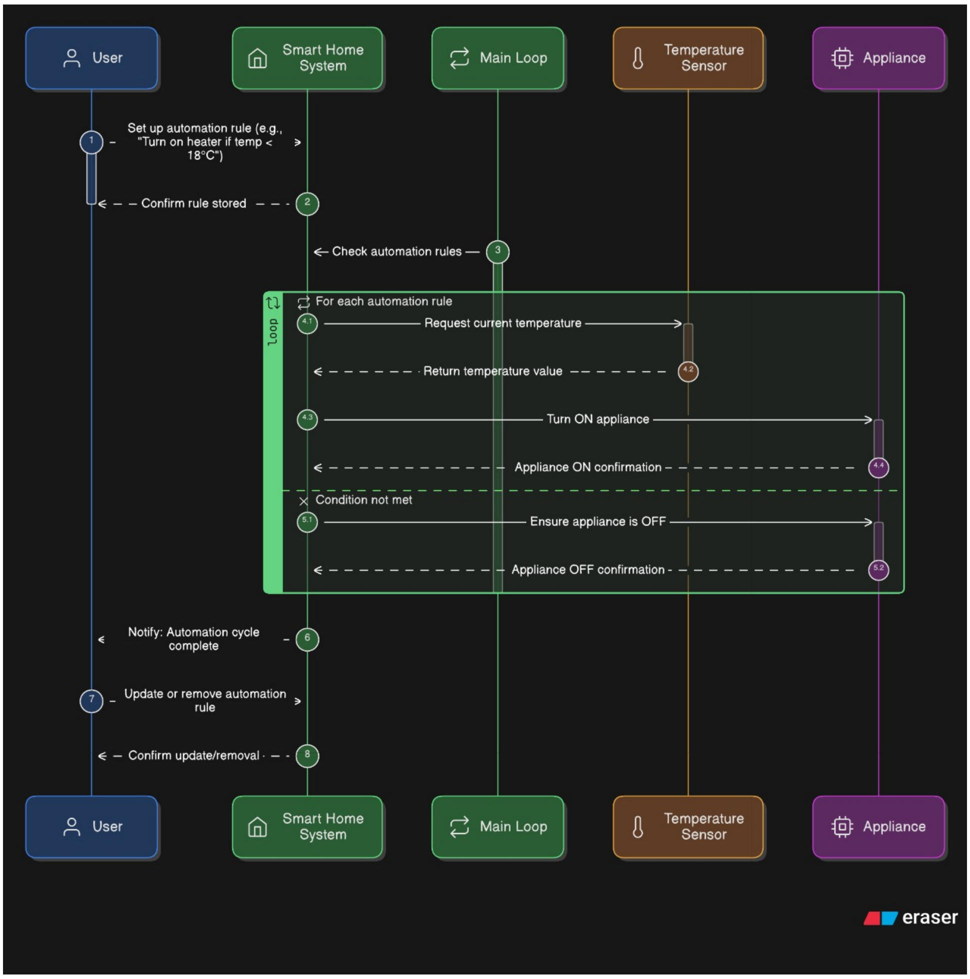

Figure 3: Sequence diagram 1

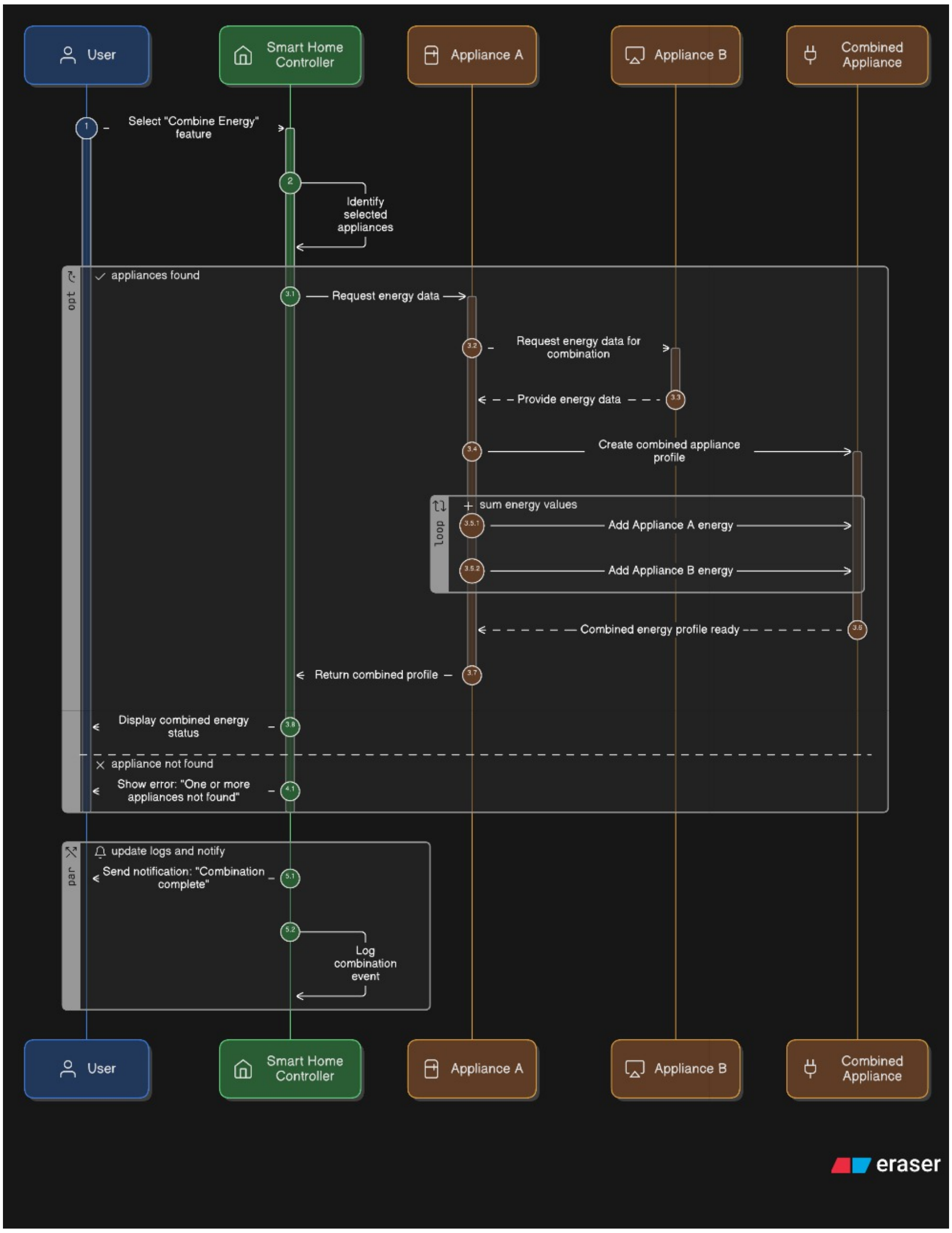

Figure 4: Sequence diagram 2

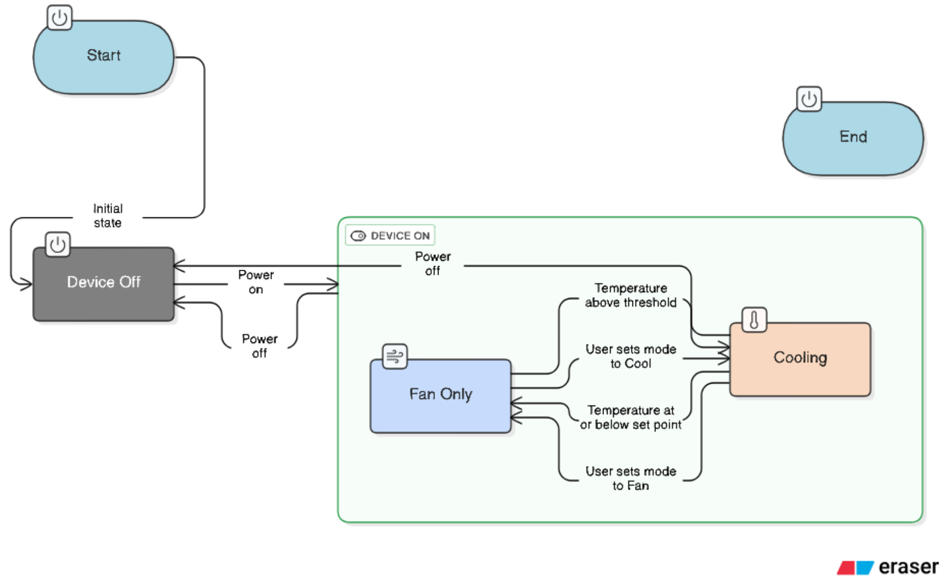

Figure 5: State diagram 1 (appliance)

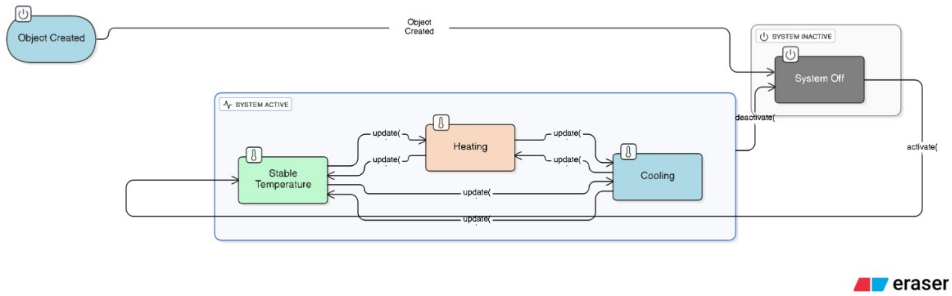

Figure 6: State diagram 2 (Sensor)

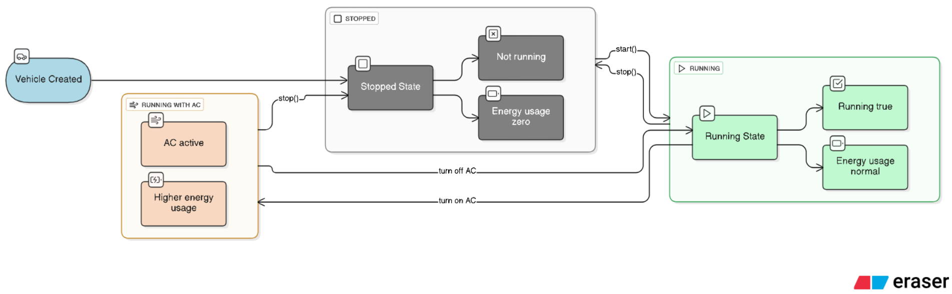

Figure 7: State diagram 3 (Vehicle)

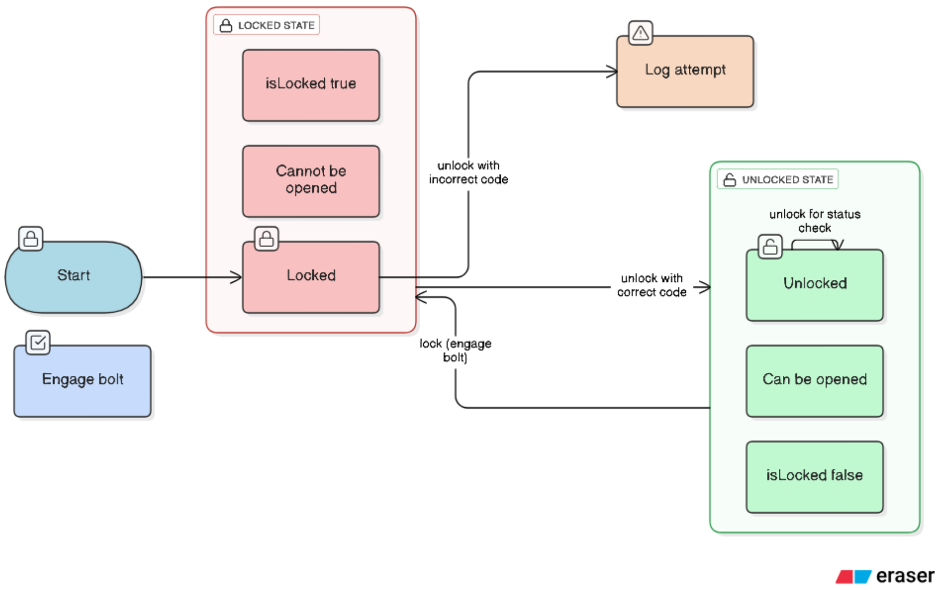

Figure 8: State diagram 4 (Door)

7 OOP Concepts Used

This project demonstrates the following concepts (mapped to code locations):

Concept	Where/How implemented
Inheritance	`Appliance` → `Light`, `Fan`, `AirConditioner`; `Sensors` base class. (see `classappliances.h`, `classsensors.h`)
Polymorphism	Virtual methods: `turnOn()`, `turnOff()`, `status()` in base classes and overriding in derived classes.
Encapsulation	Member variables private/protected; access through public getters/setters.
Abstraction	High-level `SmartHome` API hides device internals from main menu.
Dynamic allocation	Devices created using `new` and stored in `vector<...*>`.
Operator overloading	`Appliance::operator+`, `Appliance::operator+=` (see `classappliances.h`).
Friend classes	`Dishwasher` declared friend of `WashingMachine` (example).
Exception handling	Input validation and memory allocation wrapped with try/-catch blocks in `main.cpp`.
File I/O (Persistence)	`saveToFile()` / `loadFromFile()` in `SmartHome` (main storage: `smarthome.txt`).
Scheduling	`Schedule` struct and `SmartHome::checkSchedules()`.
Automation	`AutomationRule` struct and `SmartHome::applyAutomation()`.
Energy tracking (Operator overloading)	`map<string,double> energyTracker` updated by `updateEnergy()`.

8 Special Features (Detailed)

8.1 Device Groups/Zones

Users can create named groups and add devices from the existing lists. Group operations let you turn on/off all devices in a zone or control individual devices in a group. Use `manageDeviceGroups(home)` from the menu.

8.2 Automation Rules

Rules are simple if-then constructs:

```
If <SensorName> <condition> <value> then <DeviceName> ON/OFF
```

Examples:

- If `LivingTempSensor` ¿ 30 then `LivingRoomAC` ON.
- If `HallMotionSensor` == 1 then `HallLight` ON.

Rules are evaluated every main loop iteration.

8.3 Scheduling

Schedules use hour/minute-based triggers (24-hour format). On each loop iteration the program checks the system clock and executes scheduled actions at exact times.

8.4 Energy Tracking

Each appliance/vehicle stores a per-loop energy value (kWh). When active, the system accumulates this into a map<string,double> and the user can view cumulative consumption from the menu.

8.5 Operator Overloading

- `operator+`: Returns a temporary `Appliance` with combined energy values (useful for quick comparisons).
- `operator+=`: Increment a device's stored energy by a specified amount (used by menu or internal updates).

Example usage:

```
Appliance combined = *appl1 + *appl2;
*appl1 += 0.5; // add 0.5 kWh
```

9 Files

- `main.cpp` — main menu, controllers, persistence calls.
- `classappliances.h` — base and derived appliances (operator overloads included).
- `classsensors.h` — sensor classes (temperature, humidity, motion, rain).
- `classdoors.h` — doors and smart door.
- `classvehicles.h` — vehicle classes.
- `classmedia.h` — (SmartTV, Speaker) optional media appliances.

- `smarthome.txt` — persistence file (output).

10 Representative Code Snippets

10.1 Operator Overload (Appliance)

```
Appliance operator+(const Appliance& other) const {
    Appliance temp("Combined");
    temp.energyConsumption = this->energyConsumption + other.energyConsumption;
    // Optionally set isOn if either isOn
    // temp.isOn = this->isOn || other.isOn;
    return temp;
}

Appliance& operator+=(double energy) {
    this->energyConsumption += energy;
    return *this;
}
```

Listing 1: Appliance::operator+ and operator+= (excerpt)

11 Testing and Sample Run

- **Manual tests:** Add appliances, add sensors, create rules, set schedules, and verify expected behavior (AC turns on when temperature sensor reading crosses threshold; scheduled ON/OFF at specified time).
- **Persistence test:** Add several devices, exit program, restart — ensure devices are reloaded with previous state.
- **Energy test:** Turn a device ON for multiple iterations and check cumulative energy in `Show Energy`.

12 Limitations and Future Work

- Current scheduling checks at minute granularity — future improvement: simulated clock or background thread for second-level precision.
- Persistence is a simple text format. Future: JSON or binary save for richer typed storage.
- Add unit tests to validate automation logic.
- Extend operator overloading to other meaningful operators (e.g., comparison by energy or device merging).
- Replace raw pointers with `std::unique_ptr` / `shared_ptr` for safer memory.

13 Conclusion

The Smart Home Simulator demonstrates practical application of OOP using C++ and covers essential and advanced concepts required for the OOPS lab. It provides a realistic command-line environment to simulate smart devices, automation, scheduling and persistence — all implemented in a modular and extendable way.

Member Contributions

- **Kapil Kumar** (`Roll No.: 202411050`)
  Developed core modules for `Doors` and `Vehicles` (`doors.h`, `vehicles.h`), performed integration testing on these modules, and contributed to collaborative documentation.
- **Mehul Krishna** (`Roll No.: 202411061`)
  Implemented `Appliances` and `Sensors` (`appliances.h`, `sensors.h`), designed device grouping strategies, and authored documentation for these classes.
- **Shresta P Nayaka** (`Roll No.: 202411089`)
  Created `Media` module (`media.h`), developed operator overloading for energy calculations, and implemented scheduling and automation features.
- **Kishan N Prasad** (`Roll No.: 202411054`)
  Developed the main application logic and CLI (`main function`, `smarthomesim.cpp`), handled integration of all modules, and led reporting and final documentation.
- All members contributed equally to:
  - OOP architecture and interface design
  - Debugging, code review, and test case generation
  - Preparation of UML diagrams (class, use case, sequence, state charts)
  - Final integration, results analysis, and manuscript editing

Acknowledgments

Thanks to the lab tutors and the course material that provided the OOP concepts used in this project.

References

- Bjarne Stroustrup, *The C++ Programming Language.*
- Course lecture notes for OOPS (C++), IIITV DIU.
- Online references for UML and design patterns.