Design and Formal Specification of a High-Integrity Temperature Regulation Unit

Sarah Gosch

BSc (Hons) Software Engineering

School of Engineering and Computing

University of Lancashire

Table of Contents

List of Figures

List of Tables

Introduction

The real-time temperature regulation unit for the smart laboratory equipment ensures the stability of temperatures through the monitoring of sensors and the control of hardware devices for heating and cooling. This project, which implements high integrity safety standards, incorporates the formal specification, UML modeling, and concurrency techniques for the reliable software implementation. The following report outlines the requirements, decomposition, and the application of formal contracts for the prevention of faults in a multi-threaded environment.

System Requirements and Decomposition

2.1 Functional and Non-Functional Requirements

The Temperature Regulation Unit is planned for use in a high-integrity laboratory setting and therefore requires a set of requirements to be met in order to satisfy safety and determinable requirements. These Requirements are listed in Tables 2.1 and 2.2.

ID	Functional Requirement
FR1	The system must read temperature data from hardware sensors, with values from -50.0°C to 100.0°C.
FR2	The system must be capable of processing readings from multiple sensors simultaneously using concurrent threads.
FR3	If the temperature falls below the minimum target, the system must activate the heating unit.
FR4	If the temperature exceeds the maximum target, the system must activate the cooling unit.
FR5	If the temperature is within the target range, the system must set the actuator to idle.
FR6	The system must allow users to set a minimum and maximum target temperature, ensuring the minimum is always less than the maximum.
FR7	Every processed reading must be logged, including the sensor ID, current temperature, and the state of the actuator.
FR8	The system must trigger a warning if the temperature is outside the target range but within safe limits (0.0°C to 40.0°C).
FR9	The system must trigger a critical alert if the temperature falls below 0.0°C or rises above 40.0°C.

Table 2.1: Functional Requirements for the Temperature Regulation Unit'

The functional requirements (FR) in Table 2.1 focus on the primary logic of the system. This includes data acquisition within a specific temperature range (FR1), the implementation of a three-stage control loop (FR3/FR4/FR5 - HEATING, COOLING, IDLE) and the interface for administrative configuration (FR6). The system has also been designed to include multi-level alerts (warning and critical) to address temperature deviation issues before hardware failure occurs (FR7/FR8/FR9).

ID	Non-Functional Requirement
NFR1	The system must ensure data integrity when multiple sensors update the actuator or controller state simultaneously using synchronization.
NFR2	The actuator status must only ever be "IDLE", "HEATING", or "COOLING".
NFR3	The target minimum temperature must always be greater than 0.0°C and the target maximum must be less than 50.0°C.

NFR4	Sensor threads must collect data and update the controller at a regular frequency within one second.
NFR5	The system must handle thread interruptions gracefully by updating the running state and terminating the sensor loop.
NFR6	The code must adhere to formal specifications (JML) to allow for formal verification of its behavior and state.

Table 2.2: Non-Functional Requirements for the Temperature Regulation Unit'

The non-functional requirements (NFR) in Table 2.2 focus on fast response times and system reliability. Since the system processes many tasks simultaneously, data protection (NFR1) and control of parallel processes (NFR5) are crucial to prevent errors caused by simultaneous access. The use of Java Modeling Language (NFR6) also ensures that the software can be verified. This helps to ensure complete compliance with the strict security requirements specified in the task description.

2.2 System Decomposition

In order to maintain a clean and modular system, the system is broken down into four main modules:

- The **SensorInput** encapsulates the hardware abstraction layer for temperature sensors. It manages multiple threads to poll data at 500ms intervals (within one second), ensuring that the **TemperatureController** is constantly fed a stream of normalized data.
- The **TemperatureController** is the "brain" of the system. It contains the set-point algorithm and compares the input data against the "Min<Max" values. It is responsible for making the logical decision to change the system state.
- The **ActuatorControl** manages the hardware interface for heating and cooling actuators. It implements the logic that the actuator can only be in one of three discrete states (HEATING, COOLING, or IDLE), preventing conflicting hardware commands.
- The **SystemMonitoring** is a secondary high-integrity module that manages persistent logging and the asynchronous generation of alerts. By separating this concern from the main control loop, the system ensures that logging does not impact real-time temperature control.

2.3 Use Case Analysis

The UML Use Case Diagram provided in 2.1 describes the interaction with the major actors involved in the system: Temperature Sensor (Input), System Administrator (Configuration), and Actuator Hardware (Output).

The main use case is Process Sensor Reading, which includes all the included use cases like Adjust Actuator State and Log System Status. This indicates that all changes made to the sensor data will initiate a series of actions that will either adjust hardware or record system

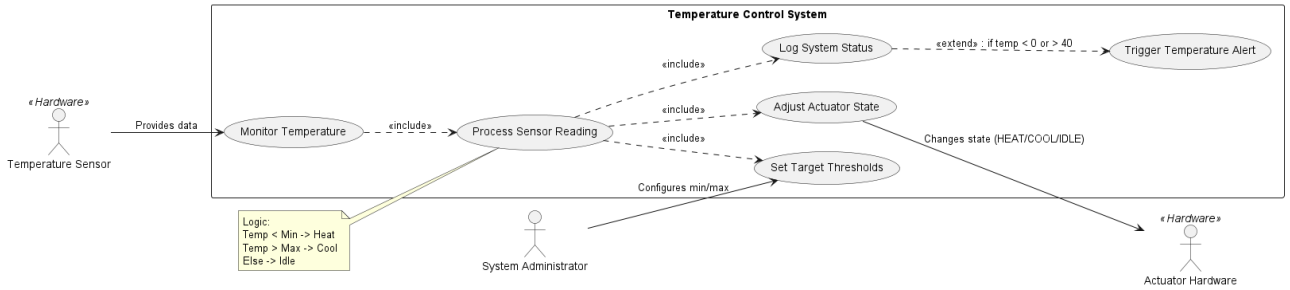

Figure 2.1: Temperature Regulating Unit - Use Case Diagram

status. However, it also includes the extend relationship with Trigger Temperature Alert, which indicates that alerts are only sent when certain safety invariant violations occur.

2.4 Justification for Reliability and Performance

The use of a modular system design directly contributes to system reliability since hardware failures are contained. In case there is a problem with **ActuatorControl**, **SystemMonitoring** can still operate and send out a critical alert.

In terms of system performance, concurrent threads enable the system to remain responsive even if there are many sensors involved. Design by Contract (DbC) principles are applied through NFR3 and NFR6 to ensure that the system fails safely instead of continuing to operate on false data, which is necessary in high-reliability electronic systems.

UML Structural and Behavioural Modelling

3.1 Structural Modelling and Architectural Justification

To ensure structural integrity, the system uses a decoupled architecture (see Fig. 3.1). The strict separation between the hardware level and the logic level fulfills the principle of single responsibility (Meso and Jain (2006)). This modular structure is crucial for the formal verification of the system.

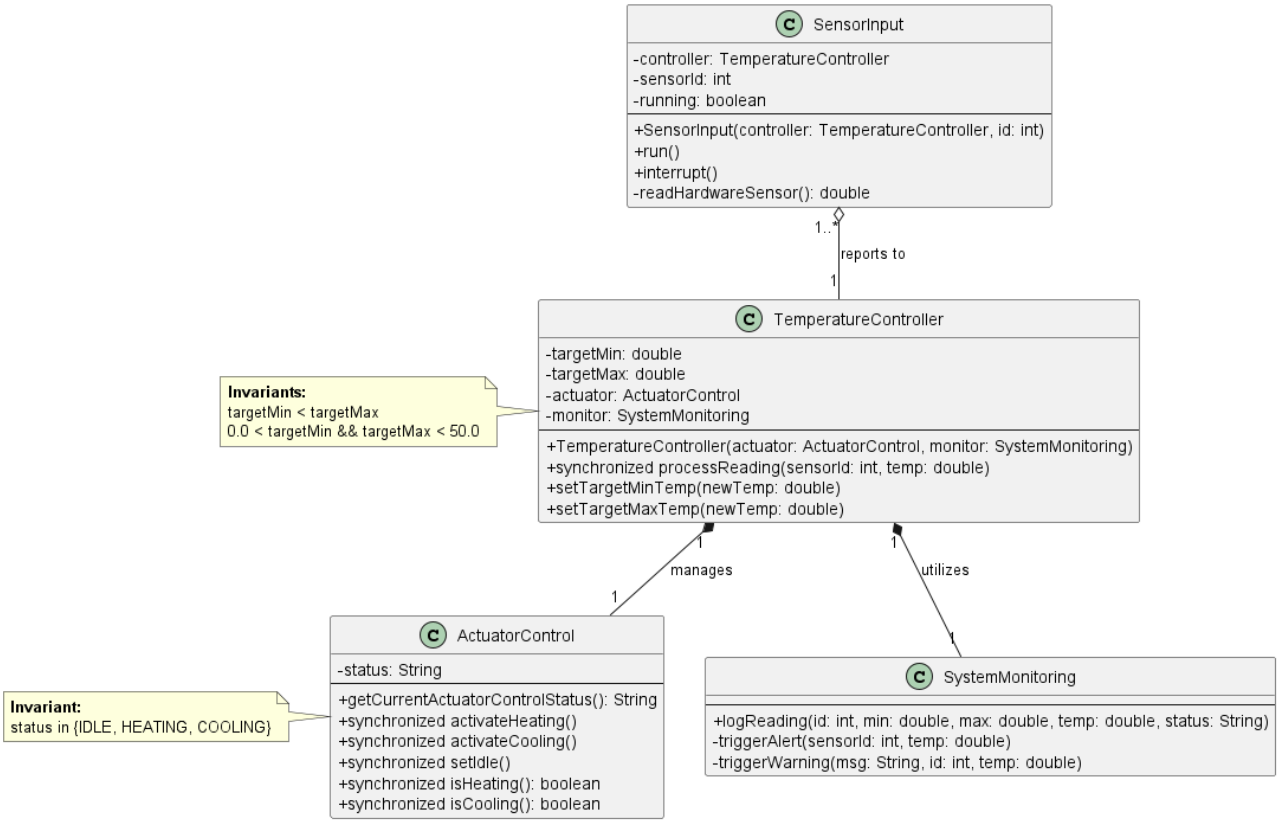

Figure 3.1: Temperature Regulating Unit - Class Diagram

The **SensorInput** class manages the lifecycle of data acquisition from sensors. It has a run-time boolean to enable proper shutdown and a sensor ID to uniquely differentiate multiple simultaneous data streams. The assignment "reports to" to the controller is done via a "1..* to 1" relationship, ensuring that all data is centralised for processing. The **TemperatureController** class (the "Mediator" in this pattern) contains the target set-points (targetMin, targetMax) as private attributes to prevent external tampering. It´s primary method, *processReading*, is the critical section of the software. The module **ActuatorControl** implements the "State" pattern in an strict manner. By defining the state of the system as a String in a strict manner (IDLE, HEATING, COOLING), it avoids getting into an "undefined" state, which would cause damage to the hardware. The state of the system could be moved to a separate ENUM class, but since the code for this report is only for demonstration purposes, the states are used as static strings. The **SystemMonitoring** class acts as an observability layer. The methods in this class, such as *triggerAlert* and *triggerWarning*, are separated from each other in a functional sense

to ensure that in case of a failure in the logging mechanism, it does not affect the temperature regulation process in any way.

3.2 Formal Specification with Object Constraint Language (OCL)

In a high-integrity system, it is not possible to rely on a simple description of a system. OCL is used to ensure mathematical certainty about the state of a system ((Hamie, 2006)).

3.2.1 System Invariants

Invariants define the "Safe State" of the equipment. If these are violated, the system is considered to be in a failed state.

Context: TemperatureController

- inv TargetIntegrity: self.targetMin < self.targetMax
- inv SafeBounds: self.targetMin > 0.0 and self.targetMax < 50.0

Context: ActuatorControl

- inv ValidStatus: Set'IDLE', 'HEATING', 'COOLING'->includes(self.status)

3.2.2 Operational Contracts (Pre/Post-conditions)

These contracts ensure that the processReading method behaves predictably regardless of the input frequency.

Context: TemperatureController::processReading(sensorId: int, temp: double)

- Precondition:

  !Double.isNaN(*temp*) *and*

  temp >= -50.0 and

  temp <= 100.0

  This ensures that the system does not attempt to process invalid data or "out-of-bounds" sensor noise.

- Postcondition:

  (*temp < targetMin implies actuator.status = 'HEATING'*) *and*

  (*temp > targetMax implies actuator.status = 'COOLING'*) *and*

  (*temp >= targetMin and temp <= targetMax implies actuator.status = 'IDLE'*)

  This guarantees that after the method executes, the physical hardware state matches the logical requirements.

3.3 Behavioural Analysis: Control Loop and Concurrency

The system's dynamic behaviour is modelled to highlight thread safety and responsiveness.

The activity diagram in Fig. 3.2 shows a sequence from data acquisition to system monitoring. An important point is the conditional branch for activating the actuator. Instead of sending meaningless 'ON' signals to the hardware, the system checks whether the actuator is already HEATING. This minimizes hardware wear and reduces bus usage for the embedded system. The lower part shows the **SystemMonitoring** block, which performs two checks: one for target thresholds and one for critical safety limits (0°C to 40°C).

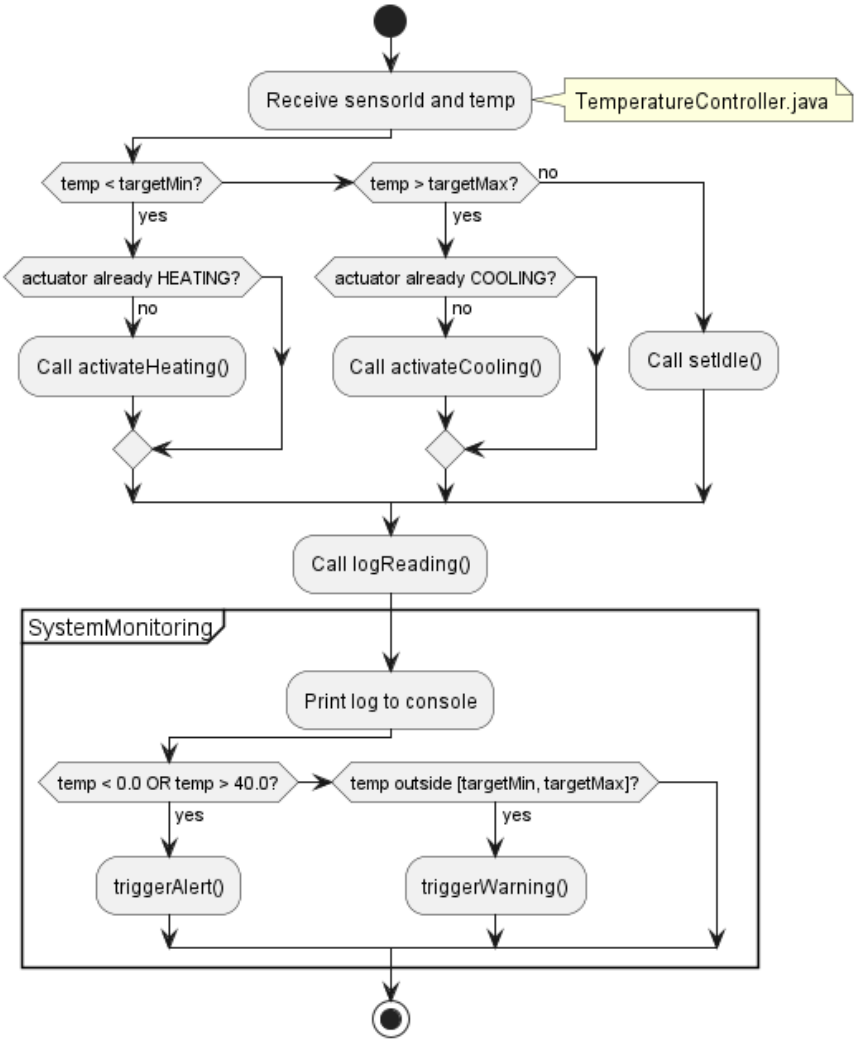

Figure 3.2: Temperature Regulating Unit - Activity Diagram

The Sequence Diagram in Fig. 3.3 represents a time-based view of system execution. As there are several **SensorInput** objects running on separate threads, *processReading* is synchronised. As shown in the figure, when sensor1 enters the controller, it "locks" the method call (the blue activation bar represents this lock state). This guarantees that the state of **ActuatorControl** is checked and updated atomically. Finally, the *status*-String is retrieved and passed to the monitor. This "Read-Process-Act-Log" pattern guarantees that every single pulse from sensors is accounted for and documented, meeting reliability requirements for smart lab equipment.

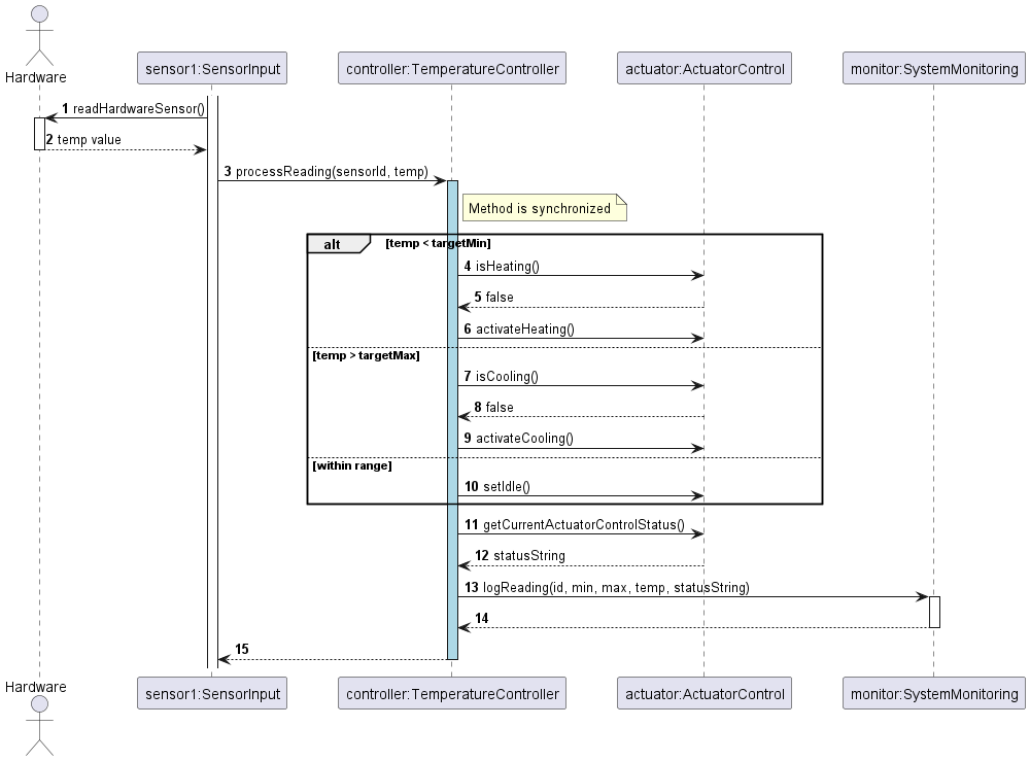

Figure 3.3: Temperature Regulating Unit - Sequence Diagram

Formal Specification and Design by Contract (DbC)

4.1 Principles of Design by Contract (DbC) in High-Integrity Systems

"Design by Contract" (DbC), as a methodology, is an essential technique in the field of embedded software engineering ((Algarni and Magel, 2018) & (Tang et al., 2024)), especially when implementing high-integrity electronic devices, such as temperature control systems. The idea is to develop software components that interact with one another based on a contract. The contract is embedded, in the form of a comment, in the source code by the use of "Java Modeling Language" (JML) ((Hamie, 2006)). The reason for doing this is to ensure correctness, which is a "fail-fast" approach. Therefore, if the function is not satisfied with the preconditions, the function will not execute, and the program will not go into an inconsistent state. The postcondition, on the other hand, is a warranty given by the supplier, which is a "promise" of a correct state. In the case of the **TemperatureController**, DbC prevents logic errors (such as activating both heating and cooling at the same time) that could result in hardware failure or safety risks in a laboratory environment.

4.2 Formal Specification and Implementation

The following implementation of the **TemperatureController** and **ActuatorControl** classes demonstrate the application of JML to enforce system safety and operational integrity.

4.2.1 Core Invariants and Safety Bounds

The system maintains strict invariants to define the "Safe State." These must hold true at every stage of the object's lifecycle.

```
1    public class TemperatureController {
2        /*@ public invariant targetMin < targetMax; @*/
3        /*@ public invariant targetMin > 0.0 && targetMax < 50.0; @*/
4        private double targetMin = 18.0;
5        private double targetMax = 24.0;
6        // ...
7    }
```

Listing 4.1: Invariants and Safety Bounds in the TemperatureController Class

The first invariant in Listing 4.1 ensures that the thermal range is logically correct, while the second ensures that the laboratory equipment is operated within the physical safety limits (0.0 °C to 50.0 °C). Furthermore, the **ActuatorControl** class (Listing 4.2) restricts the hardware status to a defined set of valid strings:

```
1    /*@ public invariant status.equals("IDLE") ||
2    @                   status.equals("HEATING") ||
3    @                   status.equals("COOLING");
4    @*/
```

Listing 4.2: Invariants and Safety Bounds in the ActuatorControl Class

4.2.2 Contract-Based Method Specifications

The *processReading* method (Listing 4.3) is the key control point for the software. The contract for this method guarantees proper input data and correct actuator state according to logic requirements upon completion.

```
1    /*@ public normal_behavior
2    @   requires !Double.isNaN(temp) && temp > -50.0 && temp < 100.0;
3    @   assignable \nothing;
4    @   ensures (temp < targetMin) ==> actuator.isHeating();
5    @   ensures (temp > targetMax) ==> actuator.isCooling();
6    @   ensures (temp >= targetMin && temp <= targetMax) ==>
7    @           (!actuator.isHeating() && !actuator.isCooling());
8    @*/
9    public synchronized void processReading(int sensorId, double temp) {
10       if (temp < targetMin) {
11           if(!actuator.isHeating()) {
12               actuator.activateHeating();
13           }
14       } else if (temp > targetMax) {
15           if(!actuator.isCooling()) {
16               actuator.activateCooling();
17           }
18       } else {
19           actuator.setIdle();
20       }
21       monitor.logReading(sensorId, targetMin, targetMax, temp, actuator.
             getCurrentActuatorControlStatus());
22   }
```

Listing 4.3: Contract-Based Method Specifications in the TemperatureController Class

The use of "==>" (logical implication) in the postconditions offers a mathematical guarantee for correctness. If the temperature is below the minimum, the actuator must be in the heating state after the execution of the method. The use of the assignable *nothing* clause is also important for high-integrity systems, as it states that there are no side effects on the class's fields after executing the method.

4.3 Supporting Verification and Testing

Formal specifications add greatly to the quality assurance process by changing the focus from "testing for bugs" to "proving correctness."

OpenJML or KeY can use these formal JML specifications for static verification (Brizhinev and Goré (2018)). These tools can verify mathematically that the postconditions will be fulfilled for all preconditions, without actually executing the code. This is critical for the reliability of the laboratory unit because it points out the edge conditions, such as floating-point precision, that could be overlooked in the course of functional testing.

In the development stage, JML contracts can be compiled to check for errors at runtime. If the **SensorInput** thread is given a NaN value or a temperature above 100°C, a JML exception is thrown immediately. This ensures that such an error is not forwarded to the **ActuatorControl** hardware interface, which could result in physical damage to the device.

The contracts provide a clear outline for unit testing. Each postcondition explicitly states what output is expected for a given input. This makes it easier to generate tests automatically. For example, in the *setTargetMinTemp* method, there is a precondition "requires newTemp < targetMax". Testing can be performed on specific input values, such as attempting to set targetMin equal to targetMax, to ensure that such an invalid condition is rejected by the system and therefore maintaining system integrity.

4.4 Prevention of Software Faults

By defining the state space and transitions, JML systematically minimises potential software errors:

- **Race conditions**: Synchronised blocks in combination with JML invariants ensure that the condition "targetMin < targetMax" is always maintained, even during parallel sensor accesses.
- **Logic errors**: Complete postconditions for *processReading* cover all possible scenarios. This prevents undefined states of the actuator and increases reliability.

The software therefore meets the strict safety requirements for use in laboratory environments.

Concurrency and Real-Time Behaviour

5.1 Modelling Concurrent Components and Resource Competition

The architecture of the **TemperatureControl** system has been designed as a multi-threaded real-time application, which is capable of processing simultaneous data inputs from various hardware. As indicated in the concurrency diagram in Fig. 5.1, the architecture uses a "shared controller lock" approach in order to manage concurrent access to the system's resources. The major entities involved in concurrent processing include **SensorInput** threads and the **TemperatureController** class.

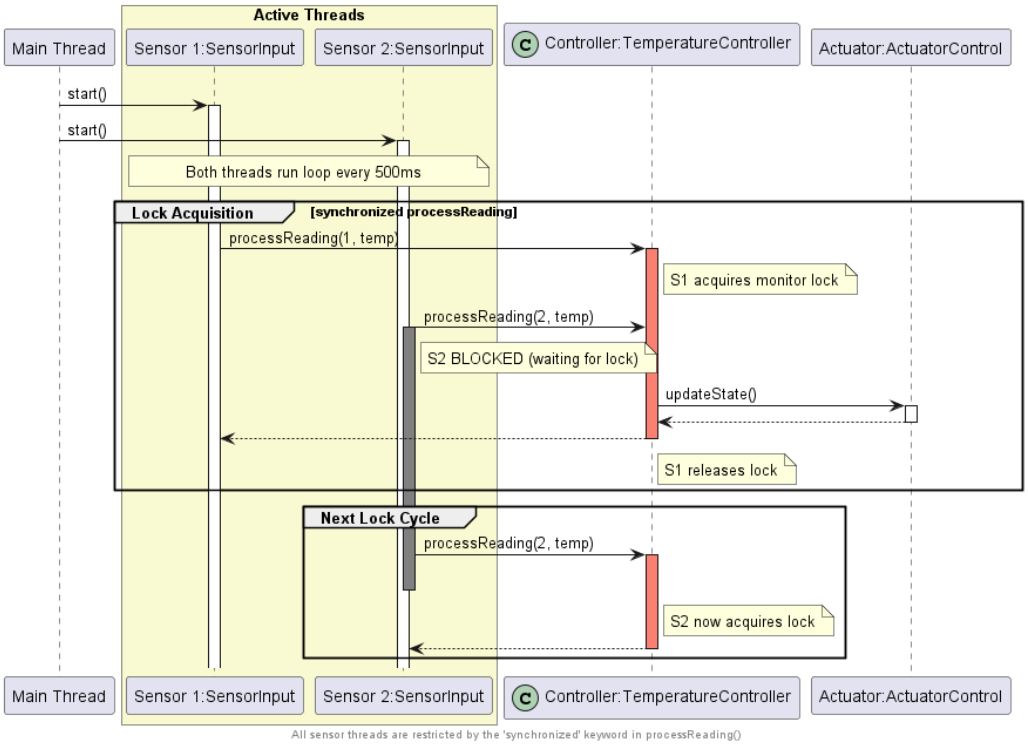

Figure 5.1: Temperature Regulating Unit - Concurrency Diagram

Each **SensorInput** class extends the Thread class and operates in independent execution cycles. This allows for scalability, as additional threads for sensors can be added without affecting the main application thread. These threads represent the active entities involved in parallel processing and accessing shared resources within the **TemperatureController** and **ActuatorControl** modules. As multiple sensors are involved in sending asynchronous data in a multi-hardware environment, it is important for the system to handle concurrent execution paths, where multiple threads may attempt to execute the processing logic at a given millisecond.

5.2 Synchronisation and State Management

For the high-integrity requirements of laboratory equipment to be met, the system needs to have deterministic state transition management. The State Machine Diagram (Fig. 5.2) for

the actuator has three mutually exclusive states: IDLE, HEATING, and COOLING. The transitions between the states are protected by temperature thresholds, specifically where "temp < targetMin" or "temp > targetMax".

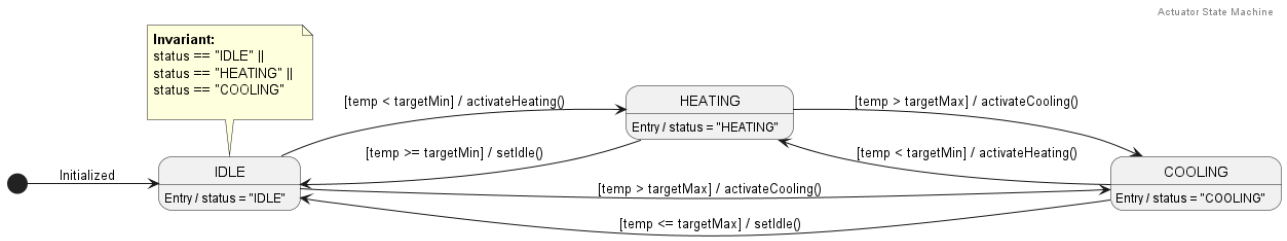

Figure 5.2: Temperature Regulating Unit - State Diagram

Synchronization is implemented by the Monitor Pattern. In Java, this is done by the synchronized keyword, which is a mutual exclusion (mutex) lock ((Welc et al., 2006)). As illustrated in the Concurrency Diagram (Fig. 5.1), when a **SensorInput** thread calls *processReading*(), it needs to lock the monitor for the **TemperatureController** object. If another thread (Sensor 2, for instance) tries to call the same method while Sensor 1 has the monitor locked, the other thread is placed in a BLOCKED state. This guarantees that the system state of the controller, including the decision logic for temperatures and the actuator is atomic and consistent.

5.3 Identification of Concurrency Issues and Technical Solutions

In a multi-threaded embedded system, several critical concurrency issues must be addressed to ensure software correctness and safety.

5.3.1 Race Conditions and Atomicity

A race condition occurs when multiple threads attempt to modify shared data simultaneously, leading to unpredictable results. In this system, if two sensors measure different temperatures at the exact same time (one suggesting a need for heating and the other suggesting cooling) they might attempt to set the actuator to conflicting states. The solution is that the system prevents race conditions by marking the *processReading*() method as synchronized. This ensures that the evaluation of the temperature logic and the subsequent hardware state update are performed as a single, atomic operation.

5.3.2 Non-Determinism and CPU Overload

For real-time systems, it is important that the system is deterministic, i.e., the system should respond to a set of input stimuli in a predictable time frame. Otherwise, there is a possibility that all the Sensor Input threads might consume a large number of CPU cycles, resulting in "starvation" of other critical system processes. To avoid this possibility, each Sensor Input thread contains a sleep cycle of 500ms in its *run*() loop, thereby providing a sampling frequency for the system while maintaining low power consumption for embedded systems.

5.3.3 Data Visibility and Volatile Memory

The shared variables, like the running flag in **SensorInput**, need to be accessible to all threads for a clean shutdown of the system. In most JVMs, variables are cached by threads, which results in a situation where a thread never "notices" that the running flag has been turned off by another thread. To prevent this the running variable is declared volatile. This ensures that any thread accessing the variable sees the latest value written by the *interrupt*() method, which causes the thread to terminate immediately.

5.4 Real-Time Scheduling and Thread Management

The following Java code snippets demonstrate the practical implementation of these concurrency and real-time principles.

5.4.1 Thread Lifecycle and Periodic Execution

The **SensorInput** class (Listing 5.1) illustrates how independent execution units are handled, as well as how periodic scheduling is implemented for predictable data acquisition.

```
1   public void run() {
2       while (running) {
3           // Acquisition of hardware data
4           double temp = readHardwareSensor();
5
6           // Concurrent access to the shared controller
7           controller.processReading(sensorId, temp);
8
9           try {
10              // Real-time scheduling: periodic sampling every 500ms
11              Thread.sleep(500);
12          } catch (InterruptedException e) {
13              this.interrupt(); // Graceful handling of thread
                    interruption
14          }
15      }
16  }
```

Listing 5.1: Real-Time Scheduling and Thread Management in the SensorInput Class

5.4.2 Thread Safety Implementation

The **ActuatorControl** and **TemperatureController** classes (Listing 5.2) use method-level synchronization to maintain the integrity of the actuator status.

```
    public class ActuatorControl {
        private String status = "IDLE";

        // Synchronized method acts as a mutex lock on the 'status' variable
        public synchronized void activateHeating() {
            status = "HEATING";
        }

        public synchronized boolean isHeating() {
            return status.equals("HEATING");
        }
    }
```

Listing 5.2: Thread Safety Implementation in the ActuatorControl Class

This ensures that the Actuator State Machine is in a valid state regardless of how many concurrent sensor inputs are processed. By using these synchronization primitives, it is ensured that transitions between HEATING, COOLING, and IDLE are never interrupted and therefore preventing invalid hardware configurations. The use of volatile flags for control, synchronized code for mutual exclusion, and Thread.sleep for periodic scheduling provides a robust, high-integrity environment for real-time temperature control.

Conclusion and Evaluation

The combination of formal models and concurrent design meets the reliability and performance requirements for laboratory electronics. The modular decomposition guarantees that hardware failures are isolated from each other, and **SystemMonitoring** offers autonomous safety notifications. UML and OCL ensure mathematical certainty for safe operation, and JML contracts ensure that "fail-fast" logic is used to prevent false information from damaging hardware. Finally, Java's synchronization primitives and volatile variables ensure that multiple sensor threads are managed properly, ensuring that the system remains deterministic and thread-safe. All of these methodologies ensure that the software meets the integrity requirements.

References

Algarni, A. and Magel, K. (2018), 'Toward design-by-contract based generative tool for object-oriented system', *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS* **2018-November**, 168–173.
URL: *https://ieeexplore.ieee.org/document/8663719*

Brizhinev, D. and Goré, R. (2018), 'A case study in formal verification of a java program'.
URL: *http://arxiv.org/abs/1809.03162*

Hamie, A. (2006), 'On the relationship between the object constraint language (ocl) and the java modeling language (jml)', *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings* pp. 411–414.
URL: *https://ieeexplore.ieee.org/document/4032217*

Meso, P. and Jain, R. (2006), 'Agile software development: Adaptive systems principles and best practices.', *Information systems management* **23**(3), Chapter 8.

Tang, Q., Zhou, Z., Bu, Z., Ma, L. and Zhong, Y. (2024), 'Formal specification and verification of api security based on design by contract', *2024 IEEE 6th International Conference on Power, Intelligent Computing and Systems, ICPICS 2024* pp. 929–933.
URL: *https://ieeexplore.ieee.org/document/10796383*

Welc, A., Hosking, A. L. and Jagannathan, S. (2006), 'Transparently reconciling transactions with locking for java synchronization', *Lecture Notes in Computer Science* (*including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*) **4067 LNCS**, 148–173.
URL: *https://link.springer.com/chapter/10.1007/11785477-8*