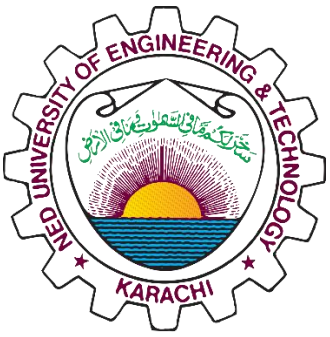
NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY

SOFTWARE ENGINEERING DEPARTMENT



FORMAL METHODS IN SOFTWARE ENGINEERING
(SE-313)

TURBINE SPEED CONTROL SYSTEM

GROUP MEMBERS: ALISHBA MUMTAZ (SE-21054)
MAHEEN SIRAJ (SE-21059)
MARIA MUNAWWAR (SE-21073)
SUBMITTED TO: SIR MUSTAFA LATIF

TURBINE SPEED CONTROL SYSTEM

SCOPE OF THE SYSTEM:

PROJECT OVERVIEW:

The scope of the **Turbine Speed Control System** project is to create a simulation platform focused on **steam turbines**, aiming to facilitate an in-depth understanding and implementation of turbine speed control mechanisms. This comprehensive project involves the development of a control unit capable of dynamically adjusting the valve position based on real-time data from **RPM** and **Valve position** Sensors. The simulation's primary objective is to maintain the rotational speed of the steam turbine at a predefined target RPM, promptly responding to fluctuations and ensuring operational stability through precise and proportional valve adjustments. The project's broader scope extends to providing valuable insights into the intricate interplay between sensors, control mechanisms, and physical components within turbine systems. This simulation contributes to a deeper understanding of efficient and responsive turbine speed control, fostering knowledge and expertise in the field.

OBJECTIVES:

The primary objectives of the Turbine Speed Control System include the development of a robust monitoring and control system capable of dynamically adjusting turbine speeds in real-time. Prioritizing safety, the system aims to prevent turbine speeds from exceeding or underachieving predefined RPM ranges, mitigating the risk of equipment damage or failure. Furthermore, the project targets the integration of sensors to facilitate communication and data exchange, enhancing overall operational efficiency.

KEY FEATURES:

The key features of the **Turbine Speed Control System** lie in its dynamic responsiveness and precision in maintaining optimal turbine performance. The central control unit, Turbine Speed Control System, orchestrates real-time adjustments to the valve position based on data from RPM Sensor and Valve Position Sensor. This ensures the turbine's rotational speed aligns closely with the predefined target RPM, demonstrating the system's efficiency in promptly addressing fluctuations. The integration of smart control mechanisms allows for smooth and proportional valve adjustments, contributing to the stability and reliability of the turbine operation. Additionally, the system's capacity to continuously monitor sensor statuses enhances safety protocols, reinforcing its role as a comprehensive and effective simulation platform for turbine speed control.

4+1 VIEW MODEL

The 4+1 Architectural View Model, when applied within the context of **formal methods** in software engineering, serves as a comprehensive framework for the description and analysis of software architecture. This model offers five distinct views: the **Logical View**, where formal methods can be applied to specify and verify the logical structure of the system; the **Process View**, which involves formal techniques to model and verify dynamic system behavior; the **Physical View**, where formal methods contribute to the modeling and verification of the physical distribution of software components; the **Development View**, where formal methods aid in ensuring the correctness of software modules and their interactions during the development process; and the **Scenarios (Use Case) View**, which employs formal techniques to specify and verify the system's behavior in response to specific scenarios. Together, these views provide a structured and multi-perspective approach, allowing for both formal and informal analysis of different aspects of the software architecture, ultimately enhancing the reliability and correctness of software systems.

LOGICAL VIEW:

In the logical view of the Turbine Speed Control System, the **Class Diagram** encapsulates the essential components and interactions governing the real-time monitoring and control of the steam turbines. The Turbine Speed Control System class diagram comprises three interconnected classes: **TurbineSpeedControlSystem**, responsible for managing the overall system with attributes like current and target RPM, valve position, and sensor statuses; **ValvePositionSensor**, handling the measurement of valve position and sensor status; and **RPMSensor**, overseeing the measurement of RPM and sensor status. The **TurbineSpeedControlSystem** performs crucial operations such as adjusting valve position, comparing RPM, updating sensor statuses, and checking overall system operability. Both sensor classes, **ValvePositionSensor** and **RPMSensor**, provide methods to measure their respective parameters, check sensor statuses, and retrieve the last update time. This structured class diagram captures the essential components and interactions within the Turbine Speed Control System, facilitating a comprehensive understanding of its logical architecture.

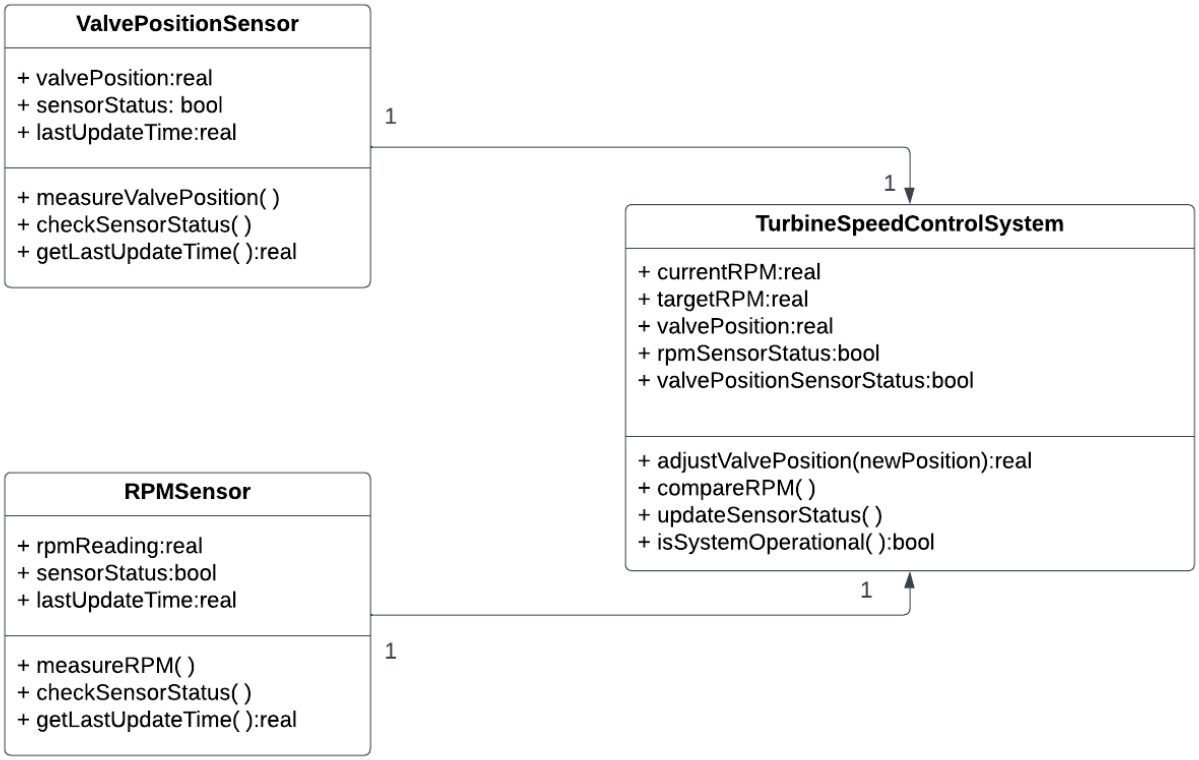


Figure 1: Class Diagram of the Turbine Speed Control System

PROCESS VIEW:

In the **Process View** of the Turbine Speed Control System, the sequence diagram orchestrates the interactions among three key objects: the **Turbine Speed Controller, Valve Position Sensor**, and **RPM Sensor**. The system initiation begins with the Turbine Speed Controller, which triggers the opening of the valve and monitors the system's functionality by querying the RPM Sensor. The Turbine class continuously observes data from both sensors, assessing the current RPM against the target RPM. In the event that the current RPM falls below the target, the Turbine Speed Controller takes corrective action by increasing the valve position, concurrently instructing the RPM Sensor to register the elevated RPM. Conversely, if the current RPM surpasses the target, the Controller reduces the valve position, notifying the RPM Sensor of the decreased RPM. This intricate dance of interactions showcases the real-time decision-making process orchestrated by the Turbine Speed Controller, maintaining optimal turbine speed through dynamic adjustments in valve position, all while ensuring synchronization with RPM sensor readings for precise control and efficient operational management.

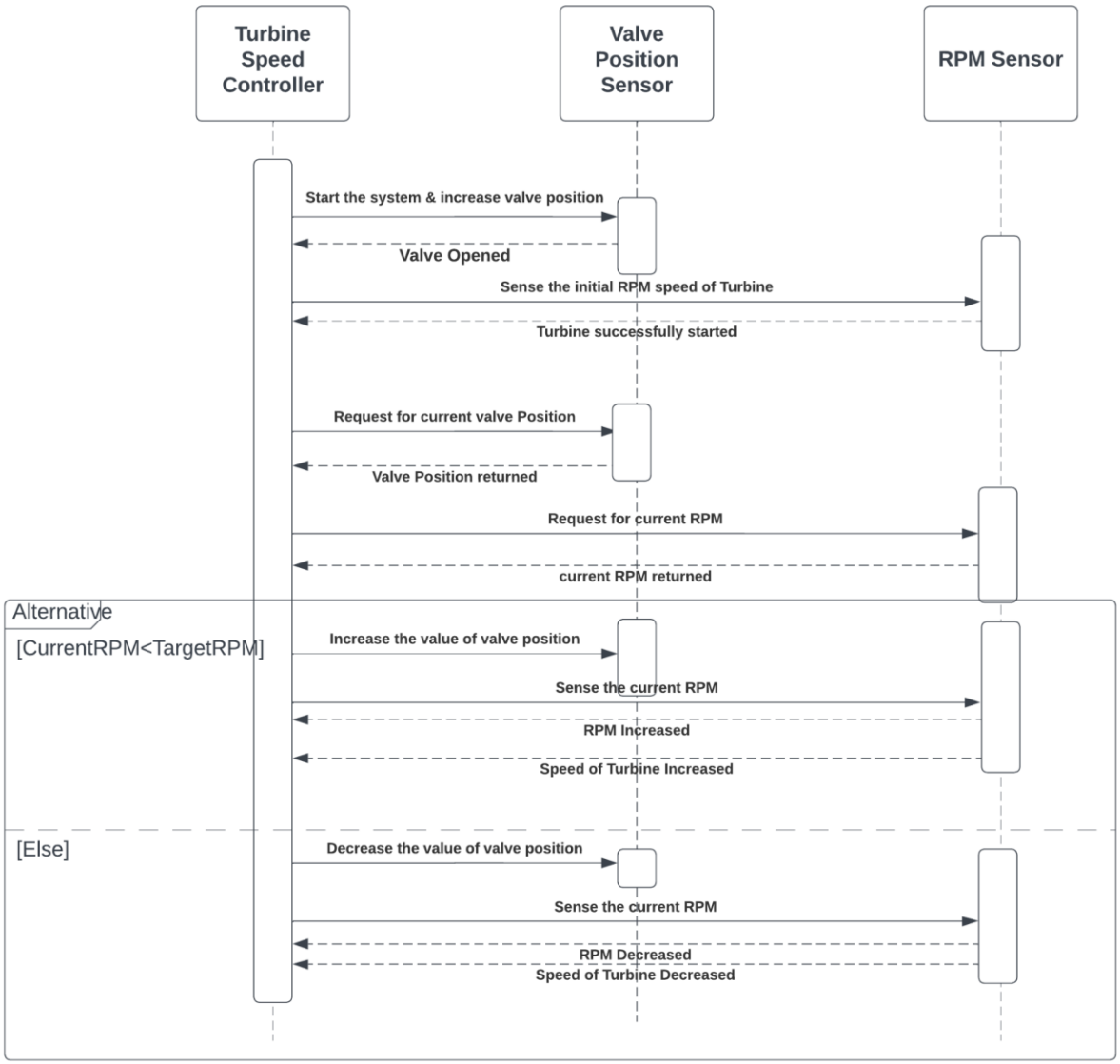


Figure 2: Sequence Diagram of the Turbine Speed Control System

PHYSICAL VIEW:

The **Physical View** of the Turbine Control System, illustrated in the Deployment Diagram, emphasizes the refined hardware architecture and component interactions. The system includes three key components: the **Turbine Speed Controller**, **Valve Position Sensor**, and **RPM Sensor**. The Client Machine node serves as the primary deployment environment, hosting these components, with the foundational software layer represented by the **Operating System (OS**) and the hardware layer by the **Processor** node. The Turbine Speed Controller manages real-time control and regulation of turbine operations, while the Valve Position Sensor and RPM Sensor enable seamless communication with their respective sensors, forming a network for data exchange. The deployment is fortified by a private network connecting the Turbine Control System

to the **RPM Sensor** and **Valve Position Sensor**, ensuring secure and dedicated communication. This refined architecture highlights a sensor-centric approach, enhancing monitoring and control capabilities by interfacing directly with dedicated RPM and Valve Position Sensors. The Physical View provides a comprehensive understanding of the strategic deployment and robust infrastructure supporting the Turbine Control System in its operational environment.

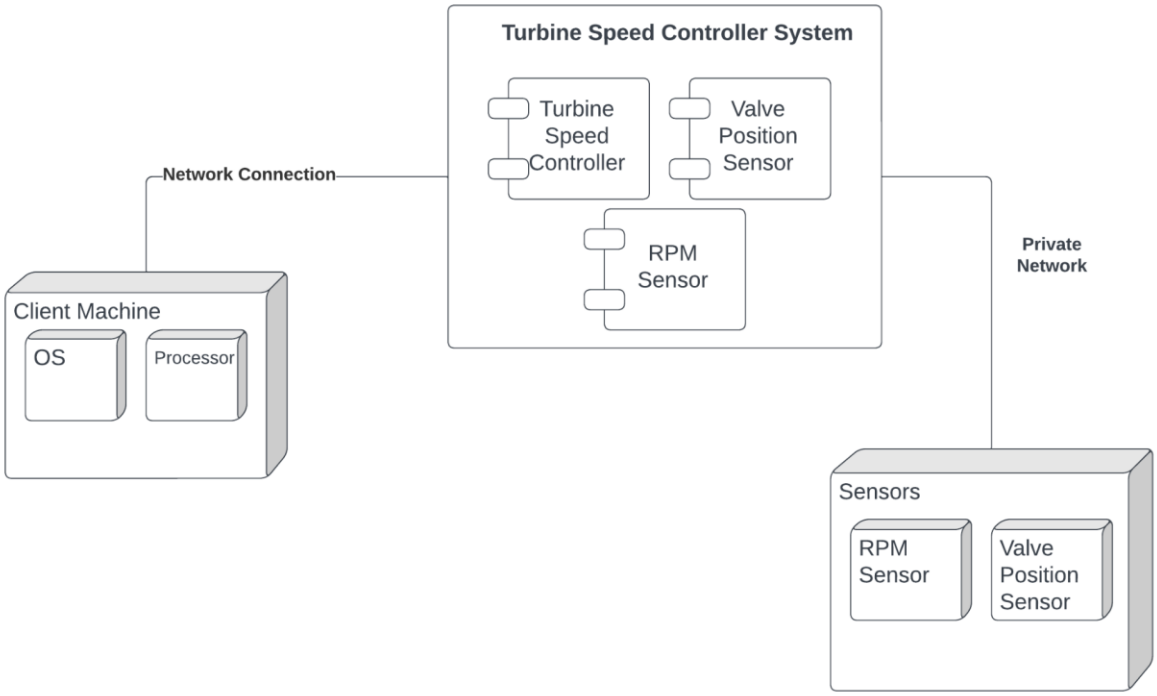


Figure 3: Deployment Diagram of the Turbine Speed Control System

DEVELOPMENT VIEW:

The **Component Diagram** for the enhanced Turbine Control System provides a detailed representation of the **Development View** within the 4+1 Architectural View Model. The Components Box serves as a central entity, encapsulating key elements: the **Turbine Speed Controller, RPM Sensor**, and **Valve Position Sensor**. This structural arrangement emphasizes the interdependencies crucial for the system's operation. The Turbine Speed Controller, positioned at the heart of the system, orchestrates real-time control and regulation, requiring essential data inputs from both the **RPM and Valve Position Sensors**. The Sensors Node acts as a centralized hub, embodying the physical sensors within the system and facilitating seamless communication with the Turbine Speed Controller. This architectural choice simplifies the representation of sensor interactions and enhances the overall modularity of the system. The individual RPM Sensor and Valve Position Sensor components contribute real-time data, empowering the Turbine Speed Controller with accurate information for informed decision-making.

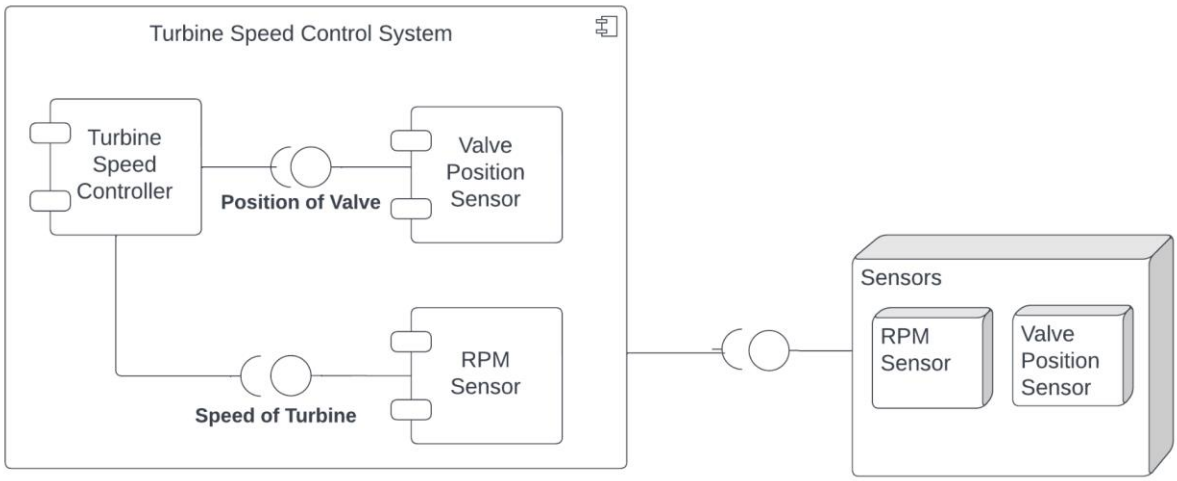


Figure 4: Component Diagram of the Turbine Speed Control System

SCENARIOS:

The **"+1" scenarios**, represented through **Use cases**, depict additional functionalities beyond the core operations of a system, showcasing its adaptability and versatility in addressing a variety of operational situations.

The use case diagram captures the essential interactions between the actors and the system, illustrating the core functionalities and information flow within the Turbine Speed Control System. The **TurbineSpeedControlSystem** dynamically adjusts the valve position based on real-time data from **RPMSenso**r and **ValvePositionSensor** to maintain the target RPM, ensuring smooth turbine operation.

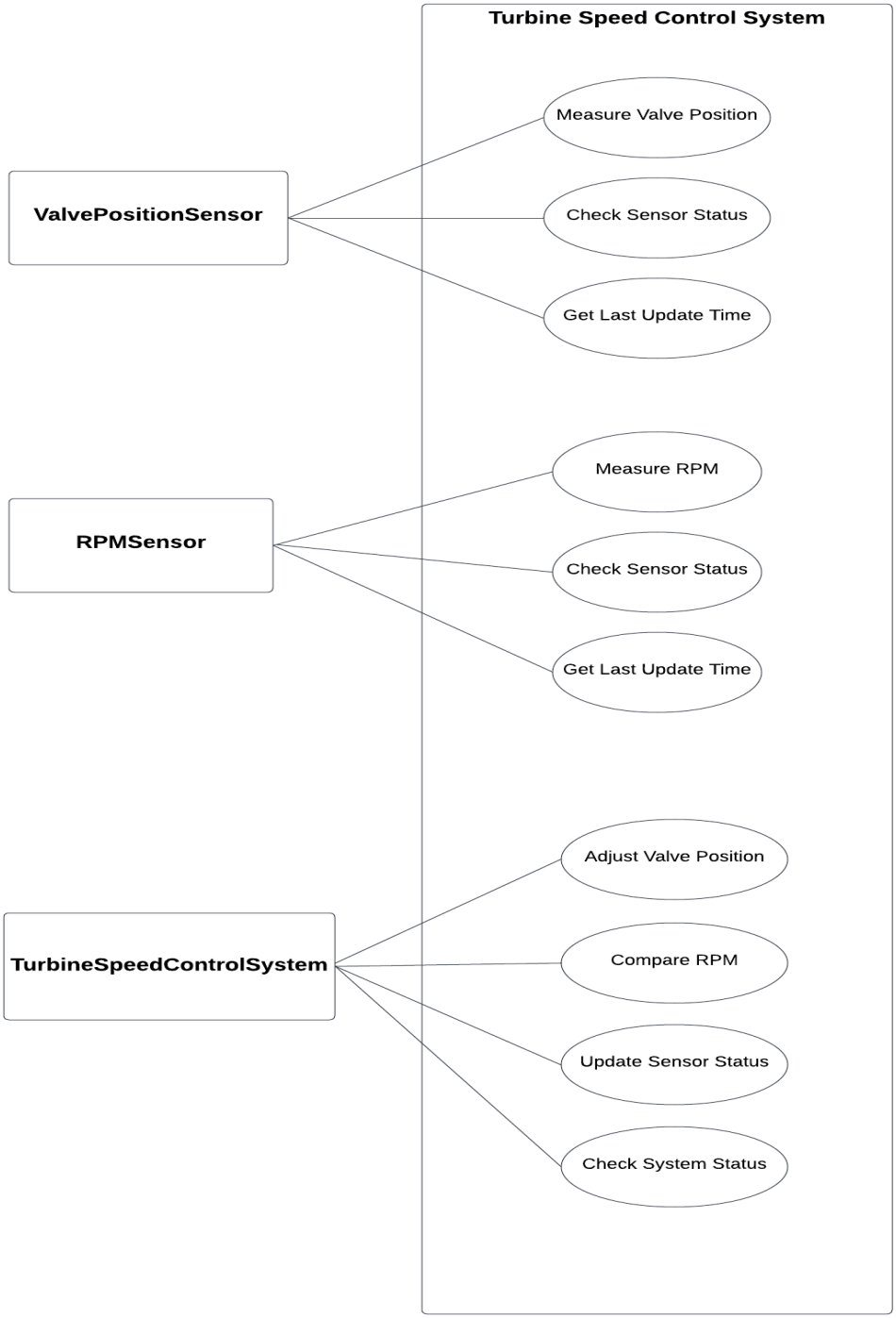
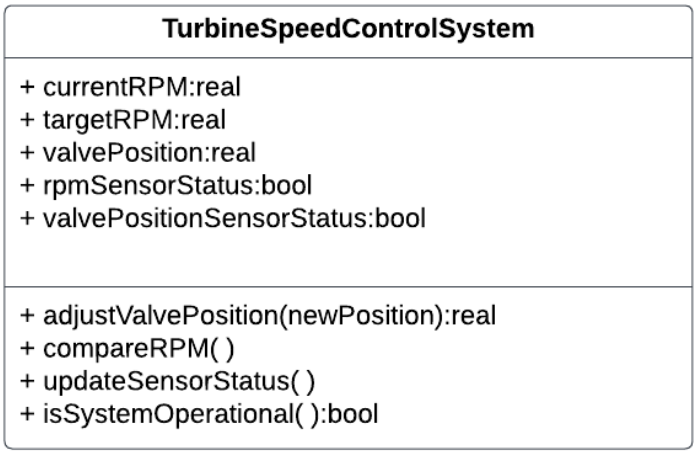


Figure 5: Use case Diagram of the Turbine Speed Control System

UML SPECIFICATION IN VDM-SL

Class TurbineSpeedControlSystem



The following is a formal specification of a **Turbine Speed Control System** using the VDM-SL (Vienna Development Method - Specification Language). This specification delineates the system's behaviour, state, and operations, employing a rigorous and mathematical approach to ensure clarity and precision. The model captures essential aspects of the turbine speed control mechanism, defining state variables, invariants, and operations in adherence to VDM-SL principles. Each component, including types, values, state, functions, and operations, is meticulously defined to provide a comprehensive and unambiguous representation of the Turbine Speed Control System.

values
targetRPM: R= 3000
tolerance:R= 0

In the values section, two key parameters are established: **targetRPM** and **tolerance**. targetRPM represents the desired revolutions per minute for the turbine, set at **3000 RPM**. The tolerance parameter, defined as **0**, sets the acceptable range within which the system adjusts to maintain the turbine speed, ensuring a fine-tuned control mechanism.These values play a fundamental role in shaping the operational characteristics of the Turbine Speed Control System, providing a clear reference point and permissible range for speed adjustments.

```
state TurbineSpeedControlSystem of
currentRPM:[R]
valvePosition:[R]
rpmSensorStatus:[B]
valvePositionSensorStatus:[B]

 - - - both currentRPM and valvePosition must be in range or equal to nil
inv-mk-TurbineSpeedControlSystem(c,v) ≜ (inRange(c) V c= nil) Λ (inRange(v) V v= nil)

 - - - both currentRPM and valvePosition are undefined when the system is initialized
init-mk-TurbineSpeedControlSystem(c,v) ≜ c= nil Λ v=nil

end
```

The **state** details the fundamental state variables that encapsulate the current state of the Turbine Speed Control System. These variables include **currentRPM**, representing the real-time revolutions per minute of the turbine; **valvePosition**, indicating the current position of the steam inlet valve; **rpmSensorStatus**, denoting the status of the RPM sensor; and **valvePositionSensorStatus**, signifying the status of the valve position sensor.

Within the state section of this VDM-SL specification, two critical components are introduced: inv-mk-TurbineSpeedControlSystem and init-mk-TurbineSpeedControlSystem.

- **Invariant (inv-mk-TurbineSpeedControlSystem):**
  The invariant enforces specific conditions that must hold true throughout the system's operation. In this case, it dictates that both currentRPM and valvePosition must either fall within a specified range or be equal to nil. This ensures that these variables maintain a well-defined state, adhering to the operational constraints of the Turbine Speed Control System.
- **Initialization Condition (init-mk-TurbineSpeedControlSystem):**
  The initialization condition specifies the state of the system when it is initialized. It ensures that both currentRPM and valvePosition are set to nil, signifying that these variables are undefined at the start of the system. This condition safeguards against potential issues arising from undefined states during the system's initialization phase.

```
functions
inRange(val : R ) result :B
pre TRUE
post result ⇒tolerance ≤ val ≤ targetRPM
```

The **functions** section of this VDM-SL specification introduces a pivotal utility function, **inRange**, designed to assess whether a given real number falls within a predefined range. This function serves as a foundational tool for subsequent operations, ensuring that critical values, such as the turbine's revolutions per minute (RPM) and valve position, adhere to specified constraints.

The inRange function, with a **precondition** of TRUE, validates that the provided value is within the acceptable range, with a **postcondition** enforcing the established tolerance limits. This function is instrumental in maintaining the consistency and reliability of key parameters within the **Turbine Speed Control System.**

operations
- - - an operation that adjusts the new position of the valve
adjustValvePosition(newPosition: R)
ext wr valvePoistion: [R]
pre inRange(newPosition) ∧valvePosition ≠ **nil**
post valvePosition=newPosition

The **adjustValvePosition** operation manages the adjustment of the steam inlet valve position, governed by a specified new position (newPosition). The operation ensures that the new position is within the predefined range and that the current valve position is well-defined before initiating the adjustment.

- **External Clause:** The operation has an external clause (ext wr valvePosition: [R]), indicating that it modifies the state variable valvePosition during execution.
- **Precondition:** The precondition specifies that the provided new position (newPosition) must be within the specified range.Additionally, the current valve position must not be undefined (nil) for the operation to proceed.
- **Postcondition:** The postcondition asserts that, after the operation, the valve position is updated to the provided new position (newPosition).

- - - an operation that compares the current and targeted RPM values and adjusts it accordingly
compareRPM( )
ext wr valvePosition:[R]
rd targetRPM : [R]
rd currentRPM : [R]
pre (currentRPM < targetRPM ∨ currentRPM > targetRPM) ∧ currentRPM ≠ **nil** ∧ targetRPM ≠ **nil**
post if currentRPM < targetRPM then valvePosition: = valvePosition+ 1 -- Increase valve position if it is below the target

```
else valvePosition: = valvePosition- 1    -- Decrease valve position if it is above the target
```

The **compareRPM** operation is pivotal for maintaining turbine speed. It adjusts the valve position based on the comparison between the current and target RPM values.

- **External Clause:** Modifies the state variable valvePosition: [R].
- **Precondition:** Requires the current RPM and target RPM to be defined and distinct from nil. Ensures the current RPM is either less than or greater than the target RPM, indicating the need for adjustment.
- **Postcondition:** Adjusts the valve position based on the RPM comparison. If the current RPM is less than the target RPM, increases the valve position by 1; if greater, decreases it by 1. Ensures that the valve position aligns with the desired RPM.

```
- - - an operation that retrieves the RPM and Valve sensors status
updateSensorStatus()
ext wr rpmSensorStatus , valvePositionSensorStatus
    rd rpmStatus, valveStatus
pre rpmStatus≠ nil Λ valveStatus≠ nil
post rpmSensorStatus = rpmStatus Λ valvePositionSensorStatus = valveStatus
```

The **updateSensorStatus** operation is instrumental in maintaining system monitoring and safety. It updates the statuses of RPM and Valve sensors based on external input.

- **External Clause:** Modifies the state variables rpmSensorStatus and valvePositionSensorStatus and reads external values rpmStatus and valveStatus.
- **Precondition:** Specifies that the input statuses (rpmStatus and valveStatus) must be defined and distinct from nil.
- **Postcondition:** Specifies that the RPM and Valve sensor statuses are updated based on the provided input. Ensures that the sensor statuses accurately reflect the external input for effective system monitoring.

```
- - - an operation that determines if the system is operational
isSystemOpeational( ) : B
ext rd rpmSensorStatus , valvePositionSensorStatus
pre TRUE
post rpmSensorStatus = true Λ valvePositionSensorStatus = true
```

The **isSystemOperational** operation is crucial for assessing the overall operational status of the Turbine Speed Control System. It determines if both RPM and Valve sensors are operational.

- **External Clause:** Reads the state variables rpmSensorStatus and valvePositionSensorStatus.
- **Precondition:** Specifies that both RPM and Valve sensor statuses must be defined and distinct from nil.
- **Postcondition:** Determines if the system is operational by verifying that both RPM and Valve sensor statuses are true. If either sensor is non-operational, the system is considered non-operational.

values
targetRPM: ℝ= 3000
tolerance:ℝ= 0

state TurbineSpeedControlSystem **of**
currentRPM:[ℝ]
valvePosition:[ℝ]
rpmSensorStatus:[𝔹]
valvePositionSensorStatus:[𝔹]

- - - both currentRPM and valvePosition must be in range or equal to nil
inv-mk-TurbineSpeedControlSystem(c,v) ≜ (inRange(c) ∨ c= **nil**) ∧ (inRange(v) ∨ v= **nil**)

- - - both currentRPM and valvePosition are undefined when the system is initialized
init-mk-TurbineSpeedControlSystem(c,v) ≜ c= **nil** ∧ v=**nil**

end

functions
inRange(*val* : ℝ) *result* :𝔹
pre *TRUE*
post *result* ⇒tolerance ≤ *val* ≤ targetRPM
operations
- - - an operation that adjusts the new position of the valve
adjustValvePosition(newPosition: ℝ)
ext wr valvePoistion: [ℝ]
pre inRange(newPosition) ∧valvePosition ≠ **nil**
post valvePosition=newPosition

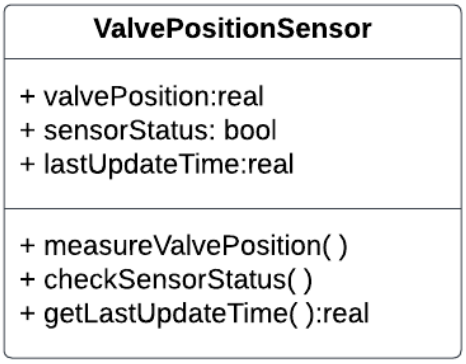
- - - an operation that compares the current and targeted RPM values and adjusts it accordingly
compareRPM( )
ext wr valvePosition:[ℝ]
    rd targetRPM : [ℝ]
    rd currentRPM : [ℝ]
pre (currentRPM < targetRPM ∨ currentRPM > targetRPM) ∧ currentRPM ≠ **nil** ∧ targetRPM ≠ **nil**

post if currentRPM < targetRPM then valvePosition: = valvePosition+ 1 -- Increase valve position if it is below the target
else valvePosition: = valvePosition- 1 -- Decrease valve position if it is above the target

- - - an operation that retrieves the RPM and Valve sensors status
updateSensorStatus()
ext wr rpmSensorStatus , valvePositionSensorStatus
    rd rpmStatus, valveStatus
pre rpmStatus≠ **nil** ∧valveStatus≠ **nil**
post rpmSensorStatus = rpmStatus ∧ valvePositionSensorStatus = valveStatus

- - - an operation that determines if the system is operational
isSystemOpeational( ) : 𝔹
ext rd rpmSensorStatus , valvePositionSensorStatus
pre TRUE
post rpmSensorStatus = true ∧ valvePositionSensorStatus = true

Class ValvePositionSensor



The following is a formal specification of the **ValvePositionSensor** class using VDM-SL. This class is an integral component of the Turbine Speed Control System, dedicated to monitoring the position of the steam inlet valve. The class introduces key state variables, including **valvePosition, sensorStatus, and lastUpdateTime**. The valvePosition is constrained to be within the range of 0 to 100 percent, reflecting operational limits. The class includes invariants and initialization conditions to ensure the consistency and reliability of its state. The specifications focus on capturing the essential characteristics and constraints associated with monitoring the valve position, providing a foundation for precise system behaviour within the broader Turbine Speed Control System.

```
values
-- assuming that the valve position should be between 0 and 100 percent.
MAX: R= 100
MIN: R= 0
```

The **values** section of the ValvePositionSensor class establishes constraints and predefined limits associated with the critical parameters of the component. Here, **MAX** is specified as 100, representing the upper limit of the valve position in percentage, and **MIN** is set to 0, denoting the lower limit. These values reflect the assumed operational range of the steam inlet valve position. The values section provides a clear definition of the permissible range for the valvePosition variable, essential for ensuring that the steam inlet valve operates within expected operational limits during the Turbine Speed Control System's functioning.

```
state ValvePositionSensor of
  valvePosition: [R]
  sensorStatus: [B]
```

lastUpdateTime: [ℝ]

-- valvePosition is in range or equal to nil
inv-mk-ValvePositionSensor(vp) ≜ (inRange(vp) ∨ vp = **nil**)

-- valvePosition is undefined when the sensor is initialized
init-mk-ValvePositionSensor(vp) ≜ vp = **nil**

end

The **state** details essential variables capturing the current state of the ValvePositionSensor within the Turbine Speed Control System. These variables include valvePosition, representing the real-time position of the steam inlet valve; sensorStatus, indicating the operational status of the sensor; and lastUpdateTime, tracking the timestamp of the last sensor update.

Within the state section of this VDM-SL specification, two key components are introduced: inv-mk-ValvePositionSensor and init-mk-ValvePositionSensor.

- **Invariant (inv-mk-ValvePositionSensor):** The invariant imposes specific conditions that must hold true during the sensor's operation. It mandates that the valvePosition must either be within the specified operational range (0 to 100) or be set to nil. This constraint ensures that the valve position remains within the defined limits or is undefined, adhering to the operational constraints of the ValvePositionSensor.
- I**nitialization Condition (init-mk-ValvePositionSensor):** The initialization condition defines the state of the sensor when it is initialised. It ensures that the valvePosition is set to nil, indicating that the valve position is undefined at the start of the sensor's operation. This condition mitigates potential issues arising from undefined states during the sensor's initialization phase.

functions

inRange (*val* : ℝ) *result* :𝔹
pre *TRUE*
post *result* ⇒MIN ≤ *val* ≤ MAX

The **functions** part of the ValvePositionSensor class introduces a single function: inRange(val: R) result: B. This function serves to verify whether a given value (val) falls within the predefined operational range of 0 to 100 percent. The absence of specific preconditions, except for the generic TRUE requirement, ensures the flexibility of the function. The postcondition dictates that the result

is **TRUE** if the provided value is within the operational range and **FALSE** otherwise. This function plays a crucial role in assessing the validity of valve position values within the specified limits, contributing to the overall reliability of the Turbine Speed Control System.

operations

-- an operation to measure the valve position where the value is read through the sensor
measureValvePosition() valvePosition:[ℝ]
ext rd valvePositionValue -- value read from the sensor
wr valvePosition
pre inRange(valvePositionValue) V valvePositionValue = **nil**
post valvePosition = valvePositionValue

The **measureValvePosition** operation in the ValvePositionSensor class specifies the process of determining the current valve position by reading the value from the sensor. This operation, with its clear specifications, defines the precise steps and conditions for obtaining the valve position from the sensor, enhancing the clarity and reliability of the Turbine Speed Control System.

- **External Clause:** Specifies that the operation reads the external value valvePositionValue from the sensor.
- **Precondition:** Specifies that the operation requires the external value (valvePositionValue) to be within the operational range (0 to 100) or set to nil. This specification ensures that the measurement aligns with the expected operational limits or is undefined.
- **Postcondition:** Specifies that the operation sets the internal variable valvePosition to the value read from the sensor (valvePositionValue). This specification ensures that the class's state is updated to reflect the most recent valve position measurement.

-- an operation to check the sensor status
checkSensorStatus() sensorStatus:[𝔹]
ext rd sensorStatusValue -- value read from the sensor
wr sensorStatus
pre sensorStatusValue ≠ **nil**
post sensorStatus = sensorStatusValue

The **checkSensorStatus** operation in the ValvePositionSensor class specifies the process of assessing the operational status of the sensor.

- **External Clause:** Specifies that the operation reads the external value sensorStatusValue from the sensor.

- **Precondition:** Specifies that the operation requires the external value (sensorStatusValue) to be defined and distinct from nil. This specification ensures that the assessment is based on a valid and defined status value.
- **Postcondition**: Specifies that the operation sets the internal variable sensorStatus to the value read from the sensor (sensorStatusValue). This ensures that the class's state is updated to reflect the most recent sensor status assessment.

```
-- an operation to get the last update timestamp
getLastUpdateTime() : R
ext rd lastUpdateTimeValue        -- value read from the sensor
    wr lastUpdateTime
pre lastUpdateTimeValue ≠ nil
post lastUpdateTime = lastUpdateTimeValue
```

The **getLastUpdateTime** operation in the ValvePositionSensor class specifies the process of retrieving the timestamp of the last sensor update.

- **External Clause:** Specifies that the operation reads the external value lastUpdateTimeValue from the sensor.
- **Precondition:** Specifies that the operation requires the external value (lastUpdateTimeValue) to be defined and distinct from nil. This ensures that the timestamp retrieval is based on a valid and defined value.
- **Postcondition:** Specifies that the operation sets the internal variable lastUpdateTime to the value read from the sensor (lastUpdateTimeValue). This ensures that the class's state is updated to reflect the most recent timestamp of the last sensor update.

```
values
-- assuming that the valve position should be between 0 and 100 percent.
MAX: R= 100
MIN: R= 0

state ValvePositionSensor of
  valvePosition: [R]
  sensorStatus: [B]
  lastUpdateTime: [R]

  -- valvePosition is in range or equal to nil
  inv-mk-ValvePositionSensor(vp) ≜ (inRange(vp) V vp = nil)
```

-- valvePosition is undefined when the sensor is initialized
init-mk-ValvePositionSensor(vp) $\triangleq$ vp = **nil**

end

functions

inRange (*val* : ℝ) *result* :𝔹
pre *TRUE*
post *result* ⇒MIN ≤ *val* ≤ MAX

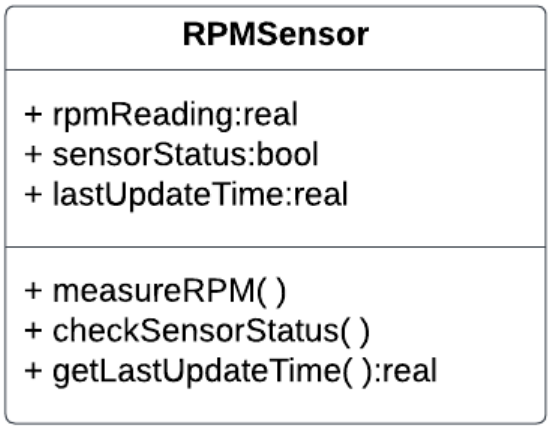
operations

-- an operation to measure the valve position where the value is read through the sensor
measureValvePosition() valvePosition:[ℝ]
ext rd valvePositionValue -- value read from the sensor
wr valvePosition
pre inRange(valvePositionValue) V valvePositionValue = **nil**
post valvePosition = valvePositionValue

-- an operation to check the sensor status
checkSensorStatus() sensorStatus:[𝔹]
ext rd sensorStatusValue -- value read from the sensor
wr sensorStatus
pre sensorStatusValue ≠ **nil**
post sensorStatus = sensorStatusValue

-- an operation to get the last update timestamp
getLastUpdateTime() : ℝ
ext rd lastUpdateTimeValue -- value read from the sensor
wr lastUpdateTime
pre lastUpdateTimeValue ≠ **nil**
post lastUpdateTime = lastUpdateTimeValue

Class RPMSensor



The following is a formal specification of the **RPMSensor** class using VDM-SL. As a critical component of the Turbine Speed Control System, the RPMSensor class is designed to continually measure and provide real-time data on the rotational speed of the turbine's shaft. The class introduces key state variables, including **rpmReading**, representing the current RPM reading; **sensorStatus**, indicating the operational status of the sensor; and **lastUpdateTime**, tracking the timestamp of the last sensor update. Operational limits for **RPM** readings are defined through invariants, ensuring the reliability and consistency of the class state. The specifications encompass functions and operations that facilitate precise control and monitoring of turbine speed within the broader system.

```
values
MAX : R= 3000
MIN :R= 0
```

The **values** of the RPMSensor class in the VDM-SL specification establishes specific values associated with critical parameters. Here, **MAX** is defined as 3000, representing the upper limit for RPM readings, while **MIN** is set to 0, denoting the lower limit. These values characterize the assumed operational range for RPM readings of the turbine's shaft. The values section provides clear constraints on acceptable RPM values, ensuring that the rpmReading variable adheres to operational limits during the Turbine Speed Control System's functioning.

```
state RPMSensor of
  rpmReading: [R]
  sensorStatus: [B]
```

```
lastUpdateTime: [ℝ]

-- rpmReading is in range or equal to nil
inv-mk-RPMSensor(r) ≜ (inRange(r) ∨ r = nil)

-- rpmReading is undefined when the sensor is initialized
init-mk-RPMSensor(r) ≜ r = nil

end
```

The **state** encapsulates crucial variables that capture the current state of the RPMSensor within the Turbine Speed Control System. These variables include rpmReading, signifying the real-time rotational speed of the turbine's shaft; sensorStatus, indicating the operational status of the sensor; and lastUpdateTime, tracking the timestamp of the last sensor update.

Within the state section of this VDM-SL specification, two vital components are introduced: inv-mk-RPMSensor and init-mk-RPMSensor.

- **Invariant (inv-mk-RPMSensor):**The invariant imposes specific conditions that must hold true during the sensor's operation. It mandates that rpmReading must either fall within the specified operational range (0 to 3000) or be set to nil. This constraint ensures that RPM readings remain within defined limits or are undefined, adhering to the operational constraints of the RPMSensor.
- **Initialization Condition (init-mk-RPMSensor):**The initialization condition defines the state of the sensor when it is initialized. It ensures that rpmReading is set to nil, signifying that the RPM readings are undefined at the start of the sensor's operation. This condition mitigates potential issues arising from undefined states during the sensor's initialization phase, enhancing the system's robustness.

```
functions

inRange (val : ℝ ) result :𝔹
pre TRUE
post result ⇒MIN ≤ val ≤ MAX
```

The **functions** of the **RPMSensor** class introduces a single function: inRange(val: R) result: B. This function serves to verify whether a given value (val) falls within the predefined operational range of **0** to **3000 RPM**. The absence of specific preconditions, except for the generic TRUE

requirement, ensures the flexibility of the function. The postcondition dictates that the result is **TRUE** if the provided value is within the operational range and **FALSE** otherwise. This function plays a crucial role in assessing the validity of RPM values, contributing to the overall reliability of the Turbine Speed Control System.

operations

-- an operation to measure the rpm reading where the value is read through the sensor
measureRPM( ) rpmReading: [ℝ]
ext rd rpmReadingValue -- value read from the sensor
wr rpmReading
pre inRange(rpmReadingValue) V rpmReadingValue = **nil**
post rpmReading = rpmReadingValue

The **measureRPM** operation in the RPMSensor class is designed to measure the RPM reading by reading the value from the sensor.

- **External Clause:** Specifies that the operation reads the external value rpmReadingValue from the sensor.
- **Precondition:**Specifies that the operation requires the external value (rpmReadingValue) to be within the operational range (0 to 3000) or set to nil. This specification ensures that the RPM measurement aligns with the expected operational limits or is undefined.
- **Postcondition:** Specifies that the operation sets the internal variable rpmReading to the value read from the sensor (rpmReadingValue). This ensures that the class's state is updated to reflect the most recent RPM reading.

-- an operation to check the sensor status
*checkSensorStatus() s*ensorStatus:[𝔹]
ext rd sensorStatusValue -- value read from the sensor
wr sensorStatus
pre sensorStatusValue ≠ **nil**
post sensorStatus = sensorStatusValue

The **checkSensorStatu**s operation in the RPMSensor class specifies the process of assessing the operational status of the sensor.

- **External Clause:** Specifies that the operation reads the external value sensorStatusValue from the sensor.
- **Precondition:** Specifies that the operation requires the external value (sensorStatusValue) to be defined and distinct from nil. This specification ensures that the assessment is based on a valid and defined sensor status value.

- **Postcondition:**Specifies that the operation sets the internal variable sensorStatus to the value read from the sensor (sensorStatusValue). This ensures that the class's state is updated to reflect the most recent sensor status assessment.

```
-- an operation to get the last update timestamp
 getLastUpdateTime() : R
 ext rd lastUpdateTimeValue        -- value read from the sensor
     wr lastUpdateTime
 pre lastUpdateTimeValue ≠ nil
 post lastUpdateTime = lastUpdateTimeValue
```

The **getLastUpdateTime** operation in the RPMSensor class specifies the process of retrieving the timestamp of the last sensor update.

- **External Clause**:Specifies that the operation reads the external value lastUpdateTimeValue from the sensor.
- **Precondition:** Specifies that the operation requires the external value (lastUpdateTimeValue) to be defined and distinct from nil. This ensures that the timestamp retrieval is based on a valid and defined value.
- **Postcondition:** Specifies that the operation sets the internal variable lastUpdateTime to the value read from the sensor (lastUpdateTimeValue). This ensures that the class's state is updated to reflect the most recent timestamp of the last sensor update.

```
values
MAX : R= 3000
MIN :R= 0


state RPMSensor of
  rpmReading: [R]
  sensorStatus: [B]
  lastUpdateTime: [R]

  -- rpmReading is in range or equal to nil
  inv-mk-RPMSensor(r) ≜ (inRange(r) V r = nil)

  -- rpmReading is undefined when the sensor is initialized
  init-mk-RPMSensor(r) ≜ r = nil
```

end

functions

inRange (*val* : ℝ) *result* :𝔹
pre *TRUE*
post *result* ⇒MIN ≤ *val* ≤ MAX

operations

-- an operation to measure the rpm reading where the value is read through the sensor
measureRPM( ) rpmReading: [ℝ]
ext rd rpmReadingValue -- value read from the sensor
wr rpmReading
pre inRange(rpmReadingValue) V rpmReadingValue = **nil**
post rpmReading = rpmReadingValue

-- an operation to check the sensor status
checkSensorStatus() sensorStatus:[𝔹]
ext rd sensorStatusValue -- value read from the sensor
wr sensorStatus
pre sensorStatusValue ≠ **nil**
post sensorStatus = sensorStatusValue

-- an operation to get the last update timestamp
getLastUpdateTime() : ℝ
ext rd lastUpdateTimeValue -- value read from the sensor
wr lastUpdateTime
pre lastUpdateTimeValue ≠ **nil**
post lastUpdateTime = lastUpdateTimeValue

VDM SL TO C++ IMPLEMENTATION

Class TurbineSpeedControlSystem:

VDM to C++ conversion is a crucial step in the development of any software, ensuring a seamless transition from formal specifications to executable code. Constants defined in the `values` clause of the VDM specification, such as `targetRPM` and `tolerance`, are implemented in C++ as constant attributes of the TurbineSpeedControlSystem class. Adopting a public declaration for these constants allows convenient access both within and outside the class, promoting a clear and accessible representation of fundamental system parameters throughout the C++ implementation.

VDM-SL	**C++ CODE**
values targetRPM: ℝ= 3000 tolerance:ℝ= 0	Const double targetRPM=3000; Const double tolerance=0;

The `invariant` in VDM-SL ensures the consistency of the TurbineSpeedControlSystem class by enforcing conditions on its attributes, such as `currentRPM` and `valvePosition`. In the C++ implementation, an analogous `public` boolean `inv` method is provided within the TurbineSpeedControlSystem class. This method evaluates whether the specified conditions, such as being within range or being nil, hold true for both `currentRPM` and `valvePosition`.

VDM-SL	**C++ CODE**

inv-mk-TurbineSpeedControlSystem(c,v) ≜ (inRange(c) V c= **nil**) Λ (inRange(v) V v= **nil**)	**Public** boolean inv {
	
	Return (inRange(currentRPM) || currentRPM =NIL) && (inRange(valvePosition) || valvePosition=NIL)
	
	}

The initialization function in VDM-SL, represented by `init-mk-TurbineSpeedControlSystem(c,v) ≜ c= nil Λ v=nil`, specifies the conditions under which the TurbineSpeedControlSystem class is initialized. The equivalent C++ code provides a public constructor for the TurbineSpeedControlSystem class. In this constructor, `currentRPM` and `valvePosition` are both set to NIL, aligning with the initialization conditions specified in VDM-SL.

VDM-SL	C++ CODE

init-mk-TurbineSpeedControlSystem(c,v) ≜
c= **nil** Λ v=**nil**
	Public TurbineSpeedControlSystem {
	currenRPM=NIL;
	valvePosition=NIL;
	}

The VDM-SL function `inRange(val : R)` is translated to a C++ function with a boolean return type. It checks if the provided value is within the specified range defined by `tolerance` and `targetRPM`, returning `true` if the condition is met and `false` otherwise.

VDM-SL	C++ CODE

inRange(*val* : R) *result* :B
pre *TRUE*
post *result* ⇒tolerance ≤ *val* ≤ targetRPM
	bool inRange(double val) {
	return (tolerance<= val && val<= targetRPM);
	}

The VDM-SL operation `adjustValvePosition(tempIn: R)` is translated into C++ as a function that adjusts the `valvePosition` based on the provided `tempIn`. The function includes assertions to verify that the input is within the specified range and that the `valvePosition` is not nil, ensuring the integrity of the operation. The C++ implementation maintains the intended behavior of the VDM-SL operation by enforcing these pre and post conditions.

VDM-SL	C++ CODE

adjustValvePosition(newPosition: R)
ext wr valvePoistion: [R]
pre inRange(newPosition) ΛvalvePosition ≠ **nil**
post valvePosition=newPosition
	void adjustValvePosition(double newPosition) {
	assert(inRange(newPosition) && valvePosition != 0);
	valvePosition = newPosition;
	assert(valvePosition == newPosition);
	}

The VDM-SL operation `compareRPM( )` is translated into C++ as a function named `compareRPM`. The function includes assertions to ensure that the preconditions, such as non-nil values for `currentRPM` and `targetRPM` and their comparison, are met. The C++ implementation then adjusts the `valvePosition` based on the comparison results, followed by assertions confirming the correct adjustment in alignment with the specified postconditions.

VDM-SL	**C++ CODE**

compareRPM( )
ext wr valvePosition:[R]
rd targetRPM : [R]
rd currentRPM : [R]
pre (currentRPM < targetRPM V currentRPM > targetRPM) Λ currentRPM ≠ **nil** Λ targetRPM ≠ **nil**
post if currentRPM < targetRPM then valvePosition: = valvePosition+ 1
else valvePosition: = valvePosition- 1
	void compareRPM() {
	assert((currentRPM < targetRPM || currentRPM > targetRPM) && currentRPM != 0 && targetRPM != 0);
	if (currentRPM < targetRPM) {
	valvePosition = valvePosition + 1;
	} else {
	valvePosition = valvePosition - 1;
	}
	assert((currentRPM < targetRPM && valvePosition == valvePosition + 1) ||
	(currentRPM > targetRPM && valvePosition == valvePosition - 1));
	}

The VDM-SL operation `updateSensorStatus()` is implemented in C++ as the `updateSensorStatus` function, incorporating calls to check the sensor statuses and updating `rpmSensorStatus` and `valvePositionSensorStatus` accordingly.

VDM-SL	**C++ CODE**

updateSensorStatus()
ext wr rpmSensorStatus , valvePositionSensorStatus
rd rpmStatus, valveStatus
pre rpmStatus≠ **nil** ΛvalveStatus≠ **nil**
	void updateSensorStatus() {
	rpmSensor.checkSensorStatus();
	valveSensor.checkSensorStatus();
	rpmSensorStatus = rpmSensor.sensorStatus;
	valvePositionSensorStatus =

post rpmSensorStatus = rpmStatus Λ valvePositionSensorStatus = valveStatus	valveSensor.sensorStatus; }

The VDM-SL operation `isSystemOperational( ) : B` is translated to a C++ function named `isSystemOperational`. The function returns the operational status by checking the preconditions that both `rpmSensorStatus` and `valvePositionSensorStatus` are initially false and asserting the postconditions that, upon execution, both statuses are true. This C++ implementation accurately represents the VDM-SL specifications for determining the system's operational status.

VDM-SL	**C++ CODE**

isSystemOperational( ) : B
ext rd rpmSensorStatus ,
valvePositionSensorStatus
pre TRUE
post rpmSensorStatus = true Λ
valvePositionSensorStatus = true
	bool isSystemOperational() {
	
	// Return the operational status
	
	return (rpmSensorStatus == true &&
	valvePositionSensorStatus == true);
	
	}

Class ValvePositionSensor:

The VDM-SL values `MAX` and `MIN` representing the upper and lower limits for the Valve Position Sensor are translated to C++ as constant attributes `MAX` and `MIN` with assigned values of 100 and 0, respectively. This ensures that the Valve Position Sensor class in C++ maintains the specified maximum and minimum limits for valve position readings throughout its implementation.

VDM-SL	**C++ CODE**
values MAX: ℝ= 100 MIN: ℝ= 0	Const double MAX=100; Const double MIN=0;

The VDM-SL invariant `inv-mk-ValvePositionSensor(vp) ≜ (inRange(vp) V vp = nil)` for the Valve Position Sensor is translated to a C++ public boolean invariant named `inv`. This invariant checks whether the valve position is either within the specified range or is nil, ensuring the consistency of the Valve Position Sensor class in adhering to the defined constraints.

VDM-SL	**C++ CODE**

inv-mk-ValvePositionSensor(vp) ≜ (inRange(vp) V vp = **nil**)	**Public** boolean inv {
	
	Return (inRange(valvePosition) || valvePosition=NIL)
	
	}

The VDM-SL initialization function `init-mk-ValvePositionSensor(vp) ≜ vp = nil` for the Valve Position Sensor class is implemented in C++ as a public constructor named `ValvePositionSensor`. In this constructor, the `valvePosition` attribute is set to `NIL`, ensuring that instances of the Valve Position Sensor are initialized with a nil valve position, consistent with the specified VDM-SL initialization conditions.

VDM-SL	**C++ CODE**

init-mk-ValvePositionSensor(vp) ≜ vp = **nil**	Public ValvePositionSensor{
	valvePosition=NIL;
	}

The VDM-SL function `inRange(val : R)` is succinctly translated to a C++ boolean function named `inRange`, ensuring that the provided value falls within the specified range defined by `MIN` and `MAX`.

VDM-SL	**C++ CODE**

inRange(*val* : ℝ) *result* :𝔹
pre *TRUE*
post *result* ⇒MIN ≤ *val* ≤ MAX
	bool inRange(double val) {
	return (MIN<= val && val<= MAX);
	}

The VDM-SL operation `measureValvePosition()` is converted to a C++ function with assertions ensuring the provided `valvePositionValue` adheres to the specified conditions. The function updates the `valvePosition` attribute and confirms its synchronization with the provided value, maintaining consistency with the VDM-SL specification.

VDM-SL	**C++ CODE**

measureValvePosition() valvePosition:[ℝ]
ext rd valvePositionValue -- value read from the sensor
wr valvePosition
pre inRange(valvePositionValue) V valvePositionValue = **nil**
post valvePosition = valvePositionValue
	void measureValvePosition(double valvePositionValue) {
	assert((inRange(valvePositionValue) || valvePositionValue == 0) && valvePositionValue != nil);
	
	valvePosition = valvePositionValue;
	
	assert(valvePosition == valvePositionValue);
	}

The VDM-SL operation `checkSensorStatus()` is translated into a C++ function with assertions ensuring that the provided `sensorStatusValue` is not nil. The function updates the `sensorStatus` attribute and confirms its synchronization with the provided value, aligning with the VDM-SL specifications for sensor status checks.

VDM-SL	**C++ CODE**
checkSensorStatus() sensorStatus:[𝔹] **ext rd** sensorStatusValue	void checkSensorStatus(bool sensorStatusValue) { assert(sensorStatusValue != nil);

wr sensorStatus
pre sensorStatusValue ≠ **nil**
post sensorStatus = sensorStatusValue
	sensorStatus = sensorStatusValue;
	
	assert(sensorStatus == sensorStatusValue);
	}

The VDM-SL operation `getLastUpdateTime() : R` is converted to a C++ method `getLastUpdateTime` within the `Sensor` class. The method asserts that `lastUpdateTimeValue` is not nil and confirms synchronization with `lastUpdateTime` before returning the last update time, aligning with the VDM-SL specifications for retrieving the sensor's last update time.

VDM-SL	**C++ CODE**

getLastUpdateTime() : R
ext rd lastUpdateTimeValue
wr lastUpdateTime
pre lastUpdateTimeValue ≠ **nil**
post lastUpdateTime = lastUpdateTimeValue
	double Sensor::getLastUpdateTime() const {
	assert(lastUpdateTimeValue != nil);
	
	assert(lastUpdateTime == lastUpdateTimeValue);
	
	return lastUpdateTime;
	}

Class RPMSensor:

The VDM-SL values `MAX` and `MIN` for the RPM Sensor are translated to C++ as constant attributes `MAX` and `MIN` with assigned values of 3000 and 0, respectively. This ensures that the RPM Sensor class in C++ maintains the specified maximum and minimum limits for RPM readings throughout its implementation.

VDM-SL	**C++ CODE**
values MAX: R= 3000 MIN: R= 0	Const double R=3000; Const double R=0;

The VDM-SL invariant `inv-mk-RPMSensor(r) ≜ (inRange(r) V r = nil)` for the RPM Sensor is translated to a C++ public boolean invariant named `inv`. This invariant checks whether the RPM reading is either within the specified range or is nil, ensuring the consistency of the RPM Sensor class in adhering to the defined constraints.

VDM-SL	**C++ CODE**

inv-mk-RPMSensor(r) ≜ (inRange(r) V r = **nil**)	**Public** boolean inv {
	
	Return (inRange(rpmReading) || rpmReading=NIL)
	
	}

The VDM-SL initialization function `init-mk-RPMSensor(r) ≜ r = nil` is implemented in C++ as a public constructor named `RPMSensor`. In this constructor, the `rpmReading` attribute is set to `NIL`, ensuring that instances of the RPMSensor class are initialized with a nil RPM reading, consistent with the specified VDM-SL initialization conditions.

VDM-SL	**C++ CODE**

init-mk-RPMSensor(r) ≜ r = **nil**	Public RPMSensor{
	rpmReading=NIL;
	}

The VDM-SL function `inRange(val : R)` is succinctly translated to a C++ boolean function named `inRange`, ensuring that the provided value falls within the specified range defined by `MIN` and `MAX`.

VDM-SL	C++ CODE

inRange(*val* : ℝ) *result* :𝔹
pre *TRUE*
post *result* ⇒MIN ≤ *val* ≤ MAX
	bool inRange(double val) {
	return (MIN<= val && val<= MAX);
	}

The VDM-SL operation `measureRPM( )` is translated to a C++ function named `measureRPM`. This function includes assertions to ensure that the provided `rpmReadingValue` adheres to the specified conditions. The function updates the `rpmReading` attribute and confirms its synchronization with the provided value, maintaining consistency with the VDM-SL specification for measuring RPM.

VDM-SL	C++ CODE

measureRPM( ) rpmReading: [ℝ]
ext rd rpmReadingValue
wr rpmReading
pre inRange(rpmReadingValue) V rpmReadingValue = **nil**
post rpmReading = rpmReadingValue
	void measureRPM(double rpmReadingValue)
	{
	assert((inRange(rpmReadingValue) || rpmReadingValue== 0) && rpmReadingValue!= nil);
	
	rpmReading= rpmReadingValue;
	
	assert(rpmReading == rpmReadingValue);
	}

The VDM-SL operation `checkSensorStatus()` is translated into a C++ function with assertions ensuring that the provided `sensorStatusValue` is not nil. The function updates the `sensorStatus` attribute and confirms its synchronization with the provided value, aligning with the VDM-SL specifications for sensor status checks.

VDM-SL	C++ CODE

checkSensorStatus() sensorStatus:[𝔹]
ext rd sensorStatusValue
wr sensorStatus
pre sensorStatusValue ≠ **nil**
	void checkSensorStatus(bool sensorStatusValue) {
	assert(sensorStatusValue != nil);
	
	sensorStatus = sensorStatusValue;

post sensorStatus = sensorStatusValue	assert(sensorStatus == sensorStatusValue); }

The VDM-SL operation `getLastUpdateTime() : R` is converted to a C++ method `getLastUpdateTime` within the `Sensor` class. The method asserts that `lastUpdateTimeValue` is not nil and confirms synchronization with `lastUpdateTime` before returning the last update time, aligning with the VDM-SL specifications for retrieving the sensor's last update time.

VDM-SL	**C++ CODE**

getLastUpdateTime() : R
ext rd lastUpdateTimeValue
wr lastUpdateTime
pre lastUpdateTimeValue ≠ **nil**
post lastUpdateTime = lastUpdateTimeValue
	double Sensor::getLastUpdateTime() const {
	assert(lastUpdateTimeValue != nil);
	
	assert(lastUpdateTime == lastUpdateTimeValue);
	
	return lastUpdateTime;
	}

SOURCE CODE:

```
#include <iostream>
#include <ctime>
#include <algorithm>
#include <cassert>

// RPMSensor class definition
class RPMSensor {
public:
   static const float MINRPM;
   static const float MAXRPM;
   float rpmReading;
   bool sensorStatus;
   time_t lastUpdateTime;

   RPMSensor() : rpmReading(0), sensorStatus(true) {}

   bool inRange(double val) {
      return (MINRPM<= val && val<= MAXRPM);
      }
    bool invariant() {
      // Check the invariant condition
      return inRange(rpmReading);
   }
   void measureRPM(float newRPM) {
      rpmReading = newRPM;
      lastUpdateTime = time(nullptr);
       //pre condition
      assert((inRange(rpmReading) ));
   }
   void checkSensorStatus() {
      // Simulate sensor status check (always operational in this simulation)
      sensorStatus = true;
       //pre condition
      assert(sensorStatus != false );
   }
};
const float RPMSensor::MINRPM = 0;
const float RPMSensor::MAXRPM = 3000;
```

```
// ValvePositionSensor class definition
class ValvePositionSensor {
public:
   static const float MIN;
   static const float MAX;
   float valvePosition;
   bool sensorStatus;
   time_t lastUpdateTime;

   ValvePositionSensor() : valvePosition(0.15f), sensorStatus(true) {} // Start with valve position
15%

     bool inRange(double val) {
       return (MIN<= val && val<= MAX);
       }
    bool invariant() {
       // Check the invariant condition
       return inRange(valvePosition);
   }
   void measureValvePosition(float newPosition) {
       valvePosition = newPosition;
       lastUpdateTime = time(nullptr);
       //pre condition
       assert((inRange(valvePosition) ));
   }
   void checkSensorStatus() {
       // Simulate sensor status check (always operational in this simulation)
       sensorStatus = true;
       //pre condition
       assert(sensorStatus != false );

   }

};
const float ValvePositionSensor::MIN = 0.0f;
const float ValvePositionSensor::MAX = 100.0f;
```

```
// TurbineSpeedControlSystem class definition
class TurbineSpeedControlSystem {
public:
   static const float targetRPM;      // declaring static const Target RPM means they are shared across all instances of this class and their values are constant
   static const float tolerance;     // static const Tolerance
   static const float minValvePosition;  // Minimum valve position
   static const float maxValvePosition;  // Maximum valve position

   float currentRPM;
   float valvePosition;
   bool rpmSensorStatus;
   bool valvePositionSensorStatus;

   RPMSensor rpmSensor;
   ValvePositionSensor valveSensor;

   TurbineSpeedControlSystem(): currentRPM(0), valvePosition(0.15f) {}

   bool inRangeRPM(float val) {
      // Check if RPM is within the range
      return (tolerance <= val && val <= targetRPM);
   }
   bool inRangeValve(float val) {
      // Check if valve position is within the range
      return (minValvePosition <= val && val <= maxValvePosition);
   }
   bool invariant() {
      // Check the invariant condition for currentRPM and valvePosition
      return inRangeRPM(currentRPM) && inRangeValve(valvePosition);
   }
   void adjustValvePosition(float newPosition) {
      // Precondition check: newPosition should be in range and valvePosition should not be zero
      assert(inRangeValve(newPosition) && valvePosition != 0.0f);
      // Constrain valve position between 10% and 100%
      valvePosition = std::max(0.1f, std::min(newPosition, 1.0f));
      valveSensor.measureValvePosition(valvePosition);
   }
```

```
    void compareRPM() {
        currentRPM = rpmSensor.rpmReading;
        float adjustment = (targetRPM - currentRPM) / targetRPM * 0.01; //smaller fluctuation
        adjustValvePosition(valvePosition + adjustment);
        //Pre condition check
        assert((currentRPM < targetRPM || currentRPM > targetRPM) && currentRPM != 0 &&
targetRPM != 0);
    }
    void updateSensorStatus() {
        rpmSensor.checkSensorStatus();
        valveSensor.checkSensorStatus();
        rpmSensorStatus = rpmSensor.sensorStatus;
        valvePositionSensorStatus = valveSensor.sensorStatus;

        //Pre condition
        assert(rpmSensor.sensorStatus == true && valveSensor.sensorStatus == true); //(This is
showing error)
    }

    bool isSystemOperational() {
        return rpmSensorStatus && valvePositionSensorStatus;
    }

};

// Initialize static constants outside the class
const float TurbineSpeedControlSystem::targetRPM = 3000.0f;
const float TurbineSpeedControlSystem::tolerance = 0.0f;
const float TurbineSpeedControlSystem::minValvePosition = 0.1f;
const float TurbineSpeedControlSystem::maxValvePosition = 1.0f;

// Main logic
int main() {
      TurbineSpeedControlSystem controlSystem;

    for (int i = 0; i < 180; i++) {
        float rpmIncrease = std::min(50.0f, (TurbineSpeedControlSystem::targetRPM -
controlSystem.rpmSensor.rpmReading) * 0.05f);
        float simulatedRPM = controlSystem.rpmSensor.rpmReading + rpmIncrease;
        controlSystem.rpmSensor.measureRPM(simulatedRPM);
```

```
        controlSystem.updateSensorStatus();

        if (controlSystem.isSystemOperational()) {
            controlSystem.compareRPM();

            std::cout << "System is operational. Time: " << i << "s, "
                      << "Current RPM: " << controlSystem.currentRPM << ", "
                      << "Valve Position: " << controlSystem.valvePosition * 100 << "%"
                      << std::endl;
        } else {
            std::cout << "Sensor failure detected. System is not operational." << std::endl;
        }
    }
    return 0;

}
```

TESTER CLASS:

```
class TurbineSpeedControlSystemTester {
public:
    static void runTests() {
        TurbineSpeedControlSystem controlSystem;
        char choice;
        try {
            do {
                displayMenu();
                std::cin >> choice;

                switch (choice) {
                    case '1':
                        simulateSystemOperation(controlSystem);
                        break;
                    case '2':
                        displaySensorStatus(controlSystem);
                        break;
                    case '3':
                        adjustValvePosition(controlSystem);
                        break;
                    case '4':
                        displayCurrentRPM(controlSystem);
                        break;
                    case '5':
                        displayValvePosition(controlSystem);
                        break;
                    case '6':
                        displaySystemStatus(controlSystem);
                        break;
                    case '7':
                        break;
                    default:
                        std::cout << "Invalid choice. Please enter a number between 1 and 7.\n";
                }
            } while (choice != '7');
        } catch (const std::exception& e) {
            std::cerr << "Exception caught: " << e.what() << std::endl;
        }
    }
```

```
private:
    static void displayMenu() {
        std::cout <<"            ***************";
        std::cout << "\n\t\tTurbineSpeedControlSystem Tester\n"
                  << "1. Simulate System Operation\n"
                  << "2. Display Sensor Status\n"
                  << "3. Adjust Valve Position\n"
                  << "4. Display Current RPM\n"
                  << "5. Display Valve Position\n"
                  << "6. Display System Operational Status\n"
                  << "7. Quit\n"
                  << "Enter choice (1-7): ";
    }

    static void simulateSystemOperation(TurbineSpeedControlSystem& controlSystem) {
    try {
        for (int i = 0; i < 180; i++) {
            float rpmIncrease = std::min(50.0f, (TurbineSpeedControlSystem::targetRPM -
controlSystem.rpmSensor.rpmReading) * 0.05f);
            float simulatedRPM = controlSystem.rpmSensor.rpmReading + rpmIncrease;
            controlSystem.rpmSensor.measureRPM(simulatedRPM);
            controlSystem.updateSensorStatus();

            if (controlSystem.isSystemOperational()) {
                controlSystem.compareRPM();

                std::cout << "System is operational. Time: " << i << "s, "
                          << "Current RPM: " << controlSystem.getCurrentRPM() << ", "
                          << "Valve Position: " << controlSystem.getValvePosition() * 100 << "%"
                          << std::endl;
            } else {
                std::cout << "Sensor failure detected. System is not operational." << std::endl;
                break;  // Exit simulation if a sensor failure occurs
            }
        }
    } catch (const std::exception& e) {
        std::cerr << "Exception caught during system operation: " << e.what() << std::endl;
    }
}
```

```
    static void displaySensorStatus(const TurbineSpeedControlSystem& controlSystem) {
        std::cout << "RPM Sensor Status: " << (controlSystem.getRPMSensorStatus() ? "Operational" : "Not Operational") << std::endl;
        std::cout << "Valve Position Sensor Status: " << (controlSystem.getValvePositionSensorStatus() ? "Operational" : "Not Operational") << std::endl;
    }
    static void adjustValvePosition(TurbineSpeedControlSystem& controlSystem) {
        try {
            float newPosition;
            std::cout << "Enter new Valve Position (0.0 to 1.0): ";
            std::cin >> newPosition;

            if (newPosition >= 0.0 && newPosition <= 1.0) {
                controlSystem.adjustValvePosition(newPosition);
                std::cout << "Valve Position adjusted successfully.\n";
            } else {
                std::cout << "Invalid Valve Position. Please enter a value between 0.0 and 1.0.\n";
            }
        } catch (const std::exception& e) {
            std::cerr << "Exception caught during valve adjustment: " << e.what() << std::endl;
        }
    }
    static void displayCurrentRPM(const TurbineSpeedControlSystem& controlSystem) {
        std::cout << "Current RPM: " << controlSystem.getCurrentRPM() << std::endl;
    }
    static void displayValvePosition(const TurbineSpeedControlSystem& controlSystem) {
        std::cout << "Valve Position: " << controlSystem.getValvePosition() * 100 << "%\n";
    }
    static void displaySystemStatus(const TurbineSpeedControlSystem& controlSystem) {
        std::cout << "System Operational Status: " << (controlSystem.isSystemOperational() ? "Operational" : "Not Operational") << std::endl;
    }
};
 int main() {
TurbineSpeedControlSystemTester::runTests();
return 0;

}
```

OUTPUTS OF TESTER CLASS:

SIMULATING SYSTEM OPERATION:

```
            ************************************
               TurbineSpeedControlSystem Tester
1. Simulate System Operation
2. Display Sensor Status
3. Adjust Valve Position
4. Display Current RPM
5. Display Valve Position
6. Display System Operational Status
7. Quit
Enter choice (1-7): 1
System is operational. Time: 0s, Current RPM: 50, Valve Position: 15.9833%
System is operational. Time: 1s, Current RPM: 100, Valve Position: 16.95%
System is operational. Time: 2s, Current RPM: 150, Valve Position: 17.9%
System is operational. Time: 3s, Current RPM: 200, Valve Position: 18.8333%
System is operational. Time: 4s, Current RPM: 250, Valve Position: 19.75%
System is operational. Time: 5s, Current RPM: 300, Valve Position: 20.65%
```

DISPLAYING SENSOR STATUS:

```
            ************************************
               TurbineSpeedControlSystem Tester
1. Simulate System Operation
2. Display Sensor Status
3. Adjust Valve Position
4. Display Current RPM
5. Display Valve Position
6. Display System Operational Status
7. Quit
Enter choice (1-7): 2
RPM Sensor Status: Operational
Valve Position Sensor Status: Operational
            ------------------------------------
```

ADJUSTING VALVE POSITION:

```
            ************************************
               TurbineSpeedControlSystem Tester
1. Simulate System Operation
2. Display Sensor Status
3. Adjust Valve Position
4. Display Current RPM
5. Display Valve Position
6. Display System Operational Status
7. Quit
Enter choice (1-7): 3
Enter new Valve Position (0.0 to 1.0): 0.30
Valve Position adjusted successfully.
            ------------------------------------
```

DISPLAYING CURRENT RPM:

```
            ************************************
               TurbineSpeedControlSystem Tester
1. Simulate System Operation
2. Display Sensor Status
3. Adjust Valve Position
4. Display Current RPM
5. Display Valve Position
6. Display System Operational Status
7. Quit
Enter choice (1-7): 4
Current RPM: 2999.24
            ....................................
```

DISPLAYING VALVE POSITION:

```
            ************************************
               TurbineSpeedControlSystem Tester
1. Simulate System Operation
2. Display Sensor Status
3. Adjust Valve Position
4. Display Current RPM
5. Display Valve Position
6. Display System Operational Status
7. Quit
Enter choice (1-7): 5
Valve Position: 30%
            ....................................
```

DISPLAYING OPERATIONAL STATUS OF SYSTEM:

```
            ************************************
               TurbineSpeedControlSystem Tester
1. Simulate System Operation
2. Display Sensor Status
3. Adjust Valve Position
4. Display Current RPM
5. Display Valve Position
6. Display System Operational Status
7. Quit
Enter choice (1-7): 6
System Operational Status: Operational
            ....................................
```

QUITING THE TESTING:

```
            ************************************
               TurbineSpeedControlSystem Tester
 1. Simulate System Operation
 2. Display Sensor Status
 3. Adjust Valve Position
 4. Display Current RPM
 5. Display Valve Position
 6. Display System Operational Status
 7. Quit
 Enter choice (1-7): 7
 mariakhan@Muhammads-MacBook-Pro output %
```