

# **Greenhouse control system**

Modeling of Mission Critical Systems

György Ihász

12.12.2020

## I. Introduction



*Figure 1.1 - Expected physical design of the greenhouse [3]*

The aim of this document is to provide a detailed design description for a controlling system of a greenhouse especially where it may involve mission-critical aspects. The contents include a requirements specification, UML class diagram, a VDM Model structure and descriptions, results and conclusions as well.

The purpose of the VDM++ model is to clarify the rules of controlling actuators depending on the environment parameters.

## II. System requirements

The table below describes the control logic that the system shall follow when it's running.

| Subject of regulation | Control logic  |
|-----------------------|--|
| Temperature           | <ol style="list-style-type: none"> <li>1. If the current temperature is higher by 5 degrees than the target temperature, then the windows shall be opened, and the Heating device shall be turned off.</li> <li>2. If the temperature is lower by one 1 degree than the target temperature, windows shall be closed, and the heating shall be turned to partial heating (&lt;HALF&gt;)</li> <li>3. If the temperature drops below the target temperature by 3 or more degrees, then the windows shall be closed, and the heating shall turned on full power (&lt;INCREMENT&gt;)</li> </ol> |
| Fire Suppression      | <ol style="list-style-type: none"> <li>1. If smoke is detected <ol style="list-style-type: none"> <li>a. Raise alarm</li> <li>b. Close windows</li> <li>c. Stop heating</li> </ol> </li> </ol>   |

|          |  |
|----------|--|
|          | d. Stop watering<br>e. Open carbon dioxide extinguisher valve  |
| Watering | 1. If the soil moisture is lower then the target soil moisture by 5% or more then the watering valve should open.<br>2. If the soil moisture is higher then the target soil moisture by 5% or more then the watering valve should close. |
| Humidity | 1. If the humidity is lower then the target humidity by 10% or more then the watering valve should open.<br>2. If the humidity is higher then the target humidity by 5% or more then the watering valve should close.                    |

### III. Dictionary

**Actuator:** A mechanical component that can be controlled by digital signals and has an impact on the environment.

- **Watering Valve:** Responsible for the watering system. Has three states:
  - <OPEN>
  - <CLOSE>
  - <HALF>
- **Window Node:** Responsible for opening the windows. Has two states:
  - <OPEN>
  - <CLOSE>
- **Fire Suppression Valve:** Responsible for opening the carbon dioxide extinguisher. Has two states:
  - <OPEN>
  - <CLOSE>
- **Heating Node:** Responsible for the temperature control of the greenhouse. Has three states:
  - <INCREMENT>
  - <DECREMENT>
  - <HALF>

**Sensor:** A component in the system which is responsible for the measurement of a specific environmental property.

- **Temperature Sensor:** Measures the environment's temperature periodically
- **Humidity Sensor:** Measures the environment's humidity periodically
- **Smoke Sensor:** Measures the environment's smoke level periodically
- **Soil Moisture Sensor:** Measures the plant's soil moisture periodically

**Controller:** The main component of the system. It is a deployed unit, which accesses the environmental properties and sends controlling messages to the actuators.

## IV. VDM model structures

In this section, the class structure and a few significant parts of the model are discussed. This project is intended to show how an actual greenhouse control system should work. In this VDM project I used the Real-Time concurrent concepts to simulate the parallelism of the sensors, actuators and the controller itself. Each Sensor and Actuator is deployed on a CPU instance and the CPUs are interconnected with a Bus instance. Each Sensor and Actuator is instantiated and deployed on a CPU in the GreenHouseAutomation.vdmrt file. The main aspect is to use periodic threads in both the actuators and the sensors. Each actuator will make an impact on the environment (e.g.: the opened window decreases humidity) and each sensor will read the environment's properties (temperature, humidity, level of smoke detected and soil moisture). Both the sensor- and actuator nodes are accessible by the controller. It will periodically read the environment properties from the sensor nodes and depending on the target values it will set a new state for the actuator nodes.

```

system GHA

instance variables

  cpu1 : CPU := new CPU(<FCFS>, 1E6); -- cpu for host controller

  cpu2 : CPU := new CPU(<FCFS>, 1E6); -- cpu for sensors
  cpu3 : CPU := new CPU(<FCFS>, 1E6); -- cpu for sensors
  cpu4 : CPU := new CPU(<FCFS>, 1E6); -- cpu for sensors
  cpu5 : CPU := new CPU(<FCFS>, 1E6); -- cpu for sensors

  cpu6 : CPU := new CPU(<FCFS>, 1E6); -- cpu for actuators
  cpu7 : CPU := new CPU(<FCFS>, 1E6); -- cpu for actuators
  cpu8 : CPU := new CPU(<FCFS>, 1E6); -- cpu for actuators
  cpu9 : CPU := new CPU(<FCFS>, 1E6); -- cpu for actuators

  -- bus connecting host controller and sensors
  bus1 : BUS := new BUS(<FCFS>, 1E3, {cpu1, cpu2, cpu3, cpu4, cpu5, cpu6, cpu7, cpu8, cpu9});

  -- Target: Temp, Humid, Smoke, Soil Moist
  public static Host: HostController := new HostController(25, 75, 0, 50);
  ----- SENSORS

  public static TemperatureNode: TemperatureSensor := new TemperatureSensor(1, <TEMPSENSOR>, 20);
  public static HumidityNode : HumiditySensor := new HumiditySensor(2, <HUMIDSENSOR>, 75);
  public static SmokeNode : SmokeSensor := new SmokeSensor(3, <SMOKESENSOR>, 75);
  public static SoilMoistureNode : SoilMoisture := new SoilMoisture(4, <MOISTSENSOR>, 75);

  ----- ACTUATORS
  public static HeatingNode : Thermostat := new Thermostat(5, <THERMOSTAT>);
  public static WaterValve : WateringValve := new WateringValve(6, <WATERVALVE>);
  public static FireSuppressorValve : FireSuppressorValve := new FireSuppressorValve(7, <FIRESUPPRESSORVALVE>);
  public static WindowNode : Window := new Window(8, <WINDOW>);

operations

public GHA: () ==> GHA
GHA() == (
  cpu1.deploy(Host);

  cpu2.deploy(TemperatureNode);
  cpu3.deploy(HumidityNode);
  cpu4.deploy(SmokeNode);
  cpu5.deploy(SoilMoistureNode);

  cpu6.deploy(WaterValve);
  cpu7.deploy(FireSuppressorValve);
  cpu8.deploy(WindowNode);
  cpu9.deploy(HeatingNode);
);

end GHA

```

Figure 4.0.1 - GreenHouseAutomation.vdmrt file

The periodicity is declared in the `Sensor` superclass, `Actuator` superclass and the `HostController` class. After the `thread` keyword, the `periodic` keyword determines the scheduling parameters (`period`, `jitter`, `delay`, `offset`). Though the superclasses specify the concurrent concepts, the actual behaviour is subclass responsibility.

## 1. Actuator

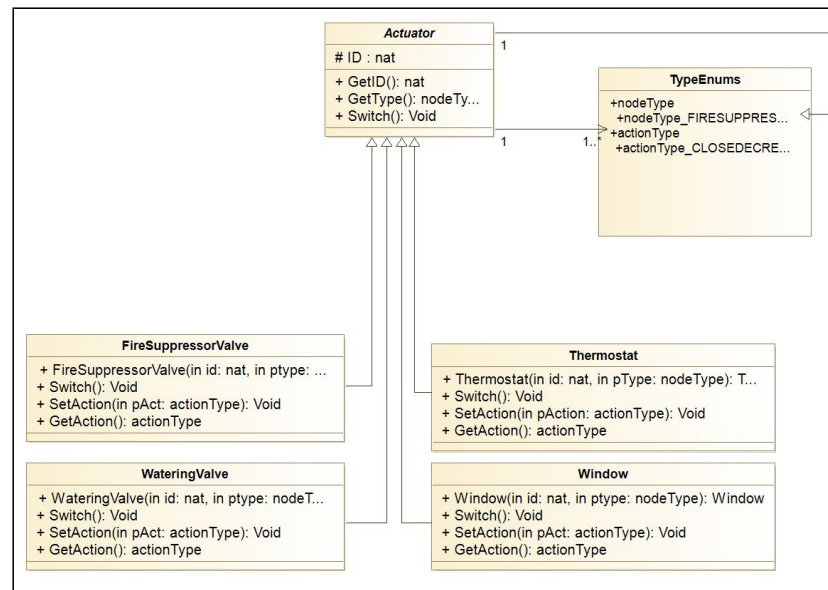


Figure 4.1.1 - Structure of the Actuator super- and the concrete classes.

As described in Figure 4.1.1, there is an `Actuator` superclass which holds the basic properties. Because of the inheritance, code redundancy is decreased in the subclasses.

|        |  |
|--------|--|
| ID     | nat → It can't be less than zero and it must be whole numbers.   |
| type   | TypeEnums`nodeType : It is an enum type, that has the following possible values: <ul style="list-style-type: none"> <li>- &lt;WINDOW&gt;</li> <li>- &lt;WATERVALVE&gt;</li> <li>- &lt;FIRESUPPRESSORVALVE&gt;</li> <li>- &lt;THERMOSTAT&gt;</li> <li>- &lt;HOSTCONTROL&gt;</li> <li>- &lt;TEMPSENSOR&gt;</li> <li>- &lt;HUMIDSENSOR&gt;</li> <li>- &lt;SMOKESENSOR&gt;</li> <li>- &lt;MOISTSENSOR&gt;</li> <li>- &lt;NONE&gt;</li> </ul> |
| action | TypeEnums`actionType : It is an enum type, that has the following possible values: <ul style="list-style-type: none"> <li>- &lt;INCREMENT&gt;</li> <li>- &lt;DECREMENT&gt;</li> </ul>  |

|  |   |
|--|---|
|  | <ul style="list-style-type: none"> <li>- &lt;OPEN&gt;</li> <li>- &lt;CLOSE&gt;</li> <li>- &lt;HALF&gt;</li> <li>- &lt;NONE&gt;</li> </ul> |
|--|---|

The type parameters above are passed in as parameters when the instantiation of an Actuator happens in the GreenHouseAutomation.vdmrt file. In each of the subclasses there is a `SetAction()` method which has a precondition. The precondition validates that only those actions(/states) can be set to the actuator that it can actually handle. For example, the concrete Window actuator has only two valid states: <OPEN> and <CLOSE>. If other `actionType` is passed in, a runtime error will occur.

## 2.Sensor

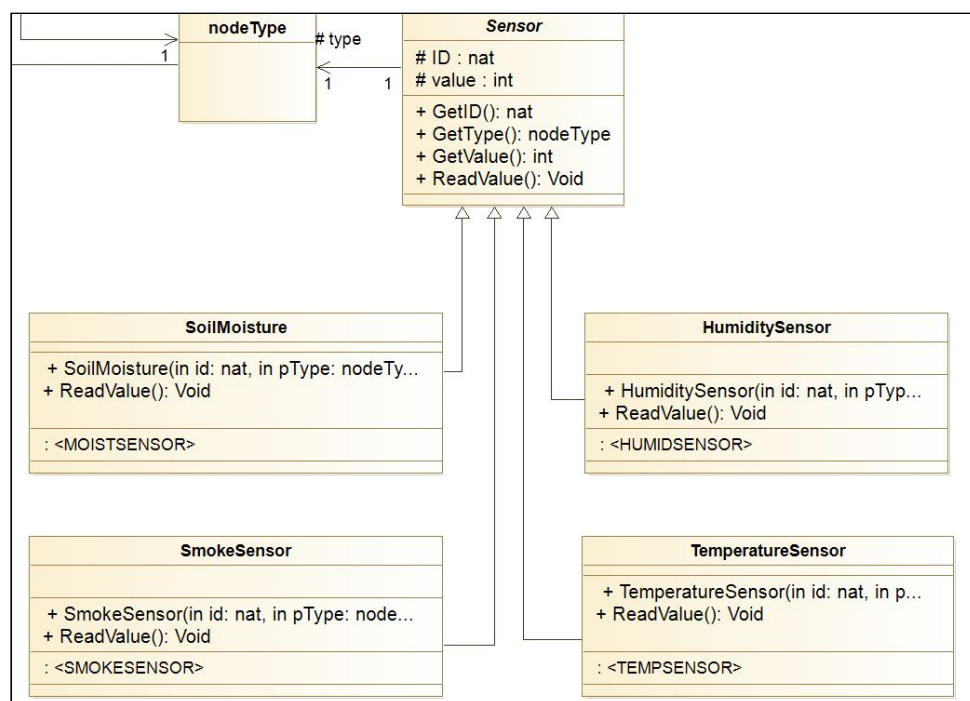


Figure 4.2.1 - Structure of the Sensor super- and the concrete classes.

As it is shown on Figure 4.2.1, there is a Sensor superclass which holds the basic properties. The Sensor superclass is similar to the Actuator Superclass. It holds an `ID` of `nat` type, and `type` of `TypeEnums`nodeType`. The `value` property is type of `int`. In the sensor superclass, `ReadValue()` is declared as periodic. This means, that each sensornode will update its `value` property periodically, so the controller can read this updated value, whenever the read is necessary for the controller.

```

class Sensor
instance variables
    protected ID      : nat;
    protected type    : TypeEnums.nodeType;
    protected value    : int;

operations

public GetID: () ==> nat
GetID() == return ID;

public GetType: () ==> TypeEnums.nodeType
GetType() ==
    return type;

public GetValue: () ==> int
GetValue() == return value;

public ReadValue: () ==> ()
ReadValue() == is subclass responsibility

thread
    -- period of thread (period, jitter, delay, offset)
    periodic(1000E6,0,0,0) (ReadValue)
end Sensor

```

Figure 4.2.2 - The code of the Sensor superclass

### 3. The Controller

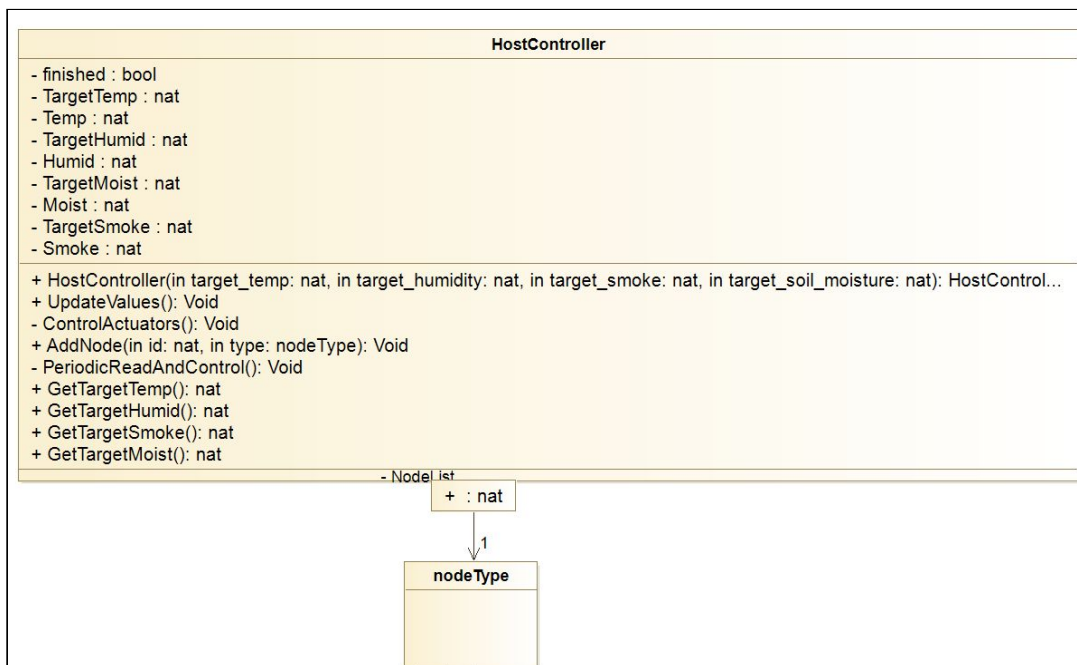


Figure 4.3.1 - Class diagram of the Host controller.

The controller can access the sensor and actuator instances through `static` variables declared in the `GreenHouseAutomation.vdmrt`. Also it holds a list of nodes which can be loaded by the `AddNode(nat * nodeType)` operation. This method is validated via a precondition, which specifies, that the node in the parameter can only be added if its ID is not already in the list. It also has a postcondition, which ensures that the node is added to the list by checking the size of the map's domain before and after the operation.

```

public AddNode: nat * TypeEnums `nodeType ==> ()
  AddNode(id, type) == {
    NodeList := NodeList ++ {id |-> type};
  }
pre id not in set dom NodeList
post card(dom NodeList) = card(dom NodeList~) + 1;

```

*Figure 4.3.2 - The AddNode operation in the controller.*

The controller has a periodic operation called `PeriodicReadAndControl()`, which contains the code for reading the environment properties, and the control logic.

```

dcl tempHigh: bool := TargetTemp + 3 <= Temp;
dcl tempExtremeLow: bool := TargetTemp - 3 >= Temp;
dcl tempLow: bool := TargetTemp - 1 >= Temp;

dcl humidHigh: bool := TargetHumid + 5 <= Humid;
dcl humidLow: bool := TargetHumid - 10 >= Humid;

dcl smokeHigh: bool := TargetSmoke < Smoke;

dcl noSmoke: bool := TargetSmoke = Smoke;

dcl moistHigh: bool := TargetMoist + 5 <= Moist;
dcl moistLow: bool := TargetMoist - 5 >= Moist;

if (tempHigh) then (
  GHA `HeatingNode.SetAction(<DECREMENT>);
  GHA `WindowNode.SetAction(<OPEN>);
) else if (tempExtremeLow) then (
  GHA `HeatingNode.SetAction(<INCREMENT>);
  GHA `WindowNode.SetAction(<CLOSE>);
) else if (tempLow) then (
  GHA `HeatingNode.SetAction(<HALF>);
  GHA `WindowNode.SetAction(<CLOSE>);
);

```

*Figure 4.3.3 - Part of the control logic declared in the controller.*

In the `HostController.vdmrt` file, a basic unit Test is declared which can be executed by setting the `Execute` method of the `Tests` class as an entry point in the run configuration.



```

protected TestConstructor : () ==> ()
TestConstructor() == (
  dcl targetTemp : int := 10;
  dcl targetHumid : int := 20;
  dcl targetMoist : int := 30;
  dcl targetSmoke : int := 40;
  dcl controller : HostController := new HostController(
    targetTemp, targetHumid, targetSmoke, targetMoist);

  dcl setTempOk : bool := targetTemp = controller.GetTargetTemp();
  dcl setHumidOk : bool := targetHumid = controller.GetTargetHumid();
  dcl setSmokeOk : bool := targetSmoke = controller.GetTargetSmoke();
  dcl setMoistOk : bool := targetMoist = controller.GetTargetMoist();

  dcl constructorWorks : bool := setTempOk and setHumidOk and setSmokeOk and setMoistOk;

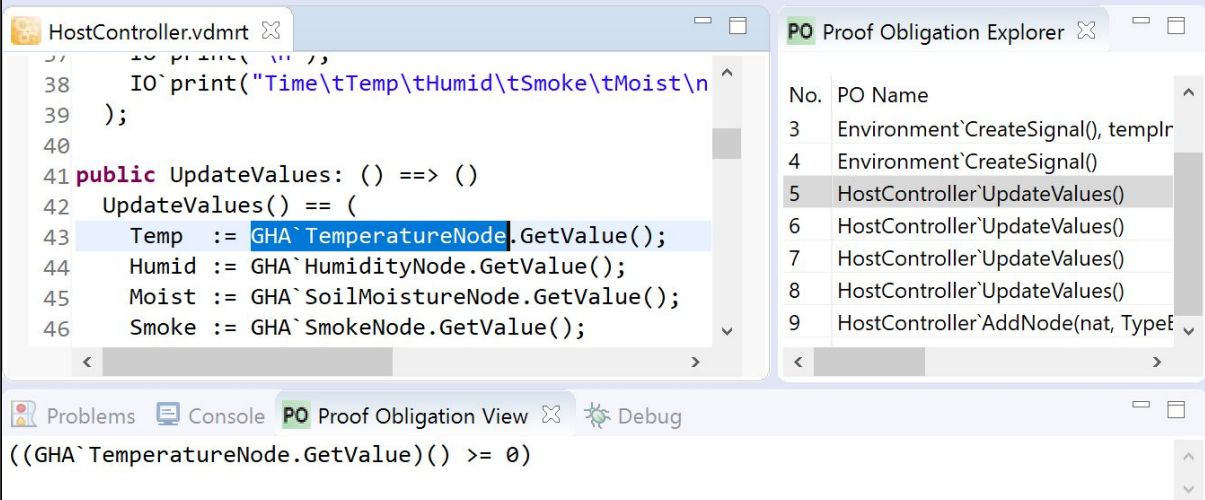
  assertTrue(constructorWorks);
);

```

*Figure 4.3.4 - One unit test for the HostController. It checks whether the constructor works properly or not. Setting the target values is very important, because the control logic is dependent on the correctly set target values.*

During the model development Proof Obligations can be generated. “In all VDM dialects, Overture can identify places where run-time errors could potentially occur if the model was to be executed. The analysis of these areas can be considered as a complement to the static type checking that is performed automatically. Type checking accepts specifications that are possibly correct, but we also want to know the places where the specification could possibly fail.”[4]

In this project, the generated POs helped in realizing mistakes in the code. Consider the following example:



The screenshot shows the Overture IDE with a VDM file named `HostController.vdmrt` open. The code in the file includes a `print` statement and a `public UpdateValues` function. The `UpdateValues` function calls `GHA`TemperatureNode.GetValue()` to get the temperature value. To the right, the `PO Proof Obligation Explorer` window displays a list of generated proof obligations. The list includes obligations for `Environment`CreateSignal()` and `HostController`UpdateValues()`. The bottom status bar shows the current proof obligation being worked on: `((GHA`TemperatureNode.GetValue()) >= 0)`.

| No. | PO Name                            |
|-----|------------------------------------|
| 3   | Environment`CreateSignal(), templr |
| 4   | Environment`CreateSignal()         |
| 5   | HostController`UpdateValues()      |
| 6   | HostController`UpdateValues()      |
| 7   | HostController`UpdateValues()      |
| 8   | HostController`UpdateValues()      |
| 9   | HostController`AddNode(nat, Typef  |

*Figure 4.3.5 - Using the generated Proof obligations to enhance a stable code.*

Here something interesting happens. `Temp` is a `nat`, while `The TemperatureNode.GetValue()` returns an `int`. There is no problem until `TemperatureNode.GetValue() > 0`. But a runtime error will occur when `TemperatureNode.GetValue() < 0` because that is not a natural number so it cannot be converted to `nat`.

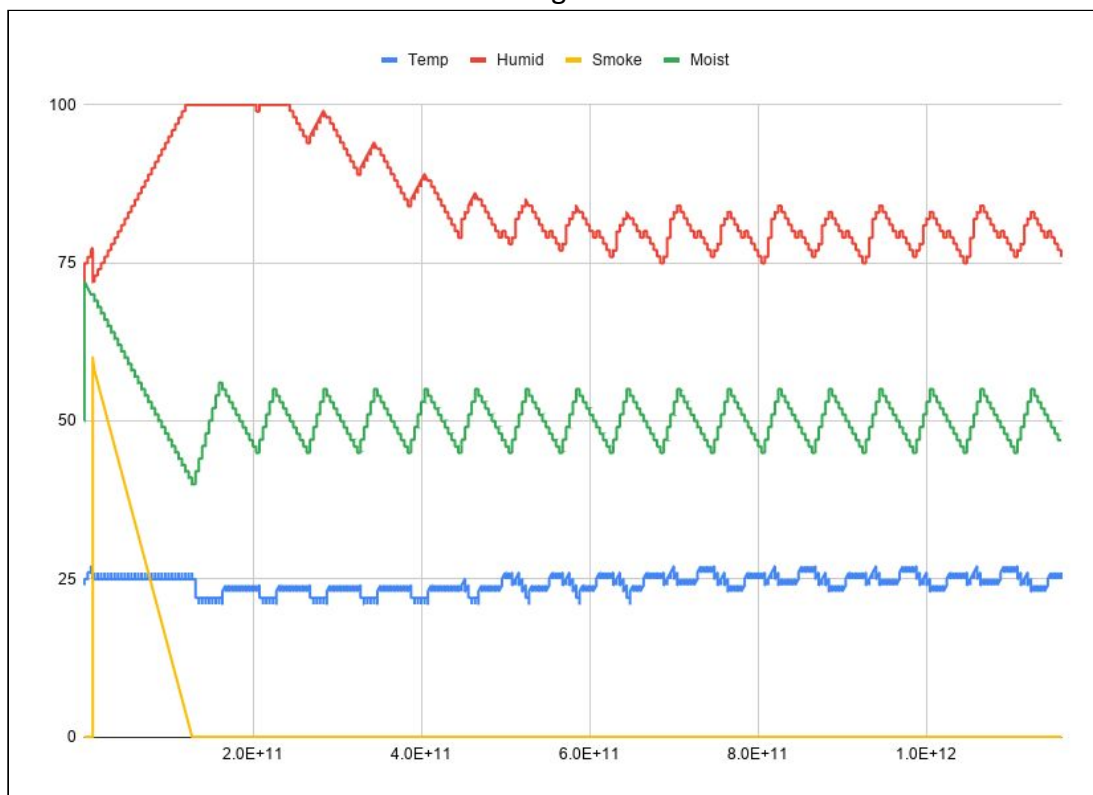
The same behaviour applies to the Humid, Moist and Smoke variables and the solution will be the same for all. We are not considering negative values in this system. Though temperatures could be negative numbers, this greenhouse is not intended for use in negative temperatures. The solution for the four similar violations is to change the value type in `Sensor` class from `int` to `nat`.

## V. Results

When the VDM project is running, it works as expected in the functional requirements. With a start scenario of the following inputs:

- Target Temperature: 25 (°C)
- Target Humidity: 75 (%)
- Target Smoke: 0 (%)
- Target Soil Moisture: 50 (%)
- Start Temperature: 25 (°C)
- Start Humidity: 72 (%)
- Start Smoke: 60 (%)
- Start Soil Moisture: 70 (%)

it produces the visualized environmental changes below.



*Figure 5.1 - The measured Temperature, Humidity, Smoke and Soil Moisture levels, visualized. The simulated fire in the greenhouse at the beginning made a huge impact on the humidity and the soil moisture. After the successful suppression, the system was able to drive the extreme humidity back to normal levels without changing the watering cycles.*

## VI. Conclusion

Though the VDM++ model provided the expected output, it still misses a few external properties which can not be controlled, like the sun, rain, and outside temperature. They could be modeled in the system as autonomous periodic threads, which have different impacts on the environmental properties. This way, a more detailed and realistic system could be modeled in the future.

## VII. Bibliography

- [1] D. F. Systems, "Delta Fire Systems," 2020. [Online]. Available: <https://www.deltafiresystems.com/suppression-systems.php>. [Accessed 19 October 2020].
- [2] A. S. Technology, "Analox Sensor Technology," 25 August 2015. [Online]. Available: <https://www.analoxsensortechnology.com/blog/2015/08/25/carbon-dioxide-sensors-fire-suppression/>. [Accessed 19 October 2020].
- [3] kcardani, "TurboSquid," 5 October 2012. [Online]. Available: <https://www.turbosquid.com/3d-models/3d-model-of-greenhouse-wood-glass/697981>. [Accessed 19 October 2020].
- [4] Peter Gorm Larsen, Kenneth Lausdahl, Peter Tran-Jørgensen, Joey Coleman, Sune Wolff, Luís Diogo Couto and Victor Bandur. Overture VDM-10 Tool Support: User Guide Version 2.7.0. Overture Technical Report Series No. TR-002, May 2019
- [5] Peter Gorm Larsen, John Fitzgerald and Sune Wolff. Methods for the Development of Distributed Real-Time Embedded Systems using VDM. In Technical report Series No. CS-TR-1163, August 2009