Technical Report:
Software System Analysis and Verification

IoT-based Plant Health Monitoring System Case Study

Abstract—**The case study for this paper is the design of a plant health monitoring system that is capable of keeping the plant's environment on an optimal level using sensors and actuators. In order simulate the system's logic and analyze it, we have opted to use the UPPAAL modeling and V&V tool to create an abstract model of the system and verify its requirements.**

In this paper, we will introduce the requirements of our system and their respective formal translations; the UPPAAL templates and their working logic; the modeling process; and finally its simulation and verification results and their interpretation.

We have successfully created an abstraction of our real world system in UPPAAL that satisfies our declared functional and non-functional requirements translated to UPPAAL properties and verified the model through queries. Moreover, we made our model to be uniform and generic in order to be able to scale and generate it automatically from code using the custom Domain Specific Language we have created for configuring the physical system.

We were able to acquire hands-on experience by working with a model-checking system, and we have obtained a deeper understanding of the strengths of it. The modeling process helped us revise the conception of our model at times, oftentimes as a result of limitations imposed by UPPAAL.

I. Introduction

The case study for this project originates from our semester project. The goal is to create a plant health monitoring system through IoT devices using sensors to perceive their environment and actuators to influence on it. Data coming from the sensors is transmitted via a MQTT broker, enabling the system to provide visualization to the user. The system independently measures certain aspects of the environment, such as temperature or soil moisture and activates actuators like heating or water pumps to maintain optimal conditions for the plant. This study depict a prototype of the system, consisting of a single controller board and several sensors and actuators of different types. Furthermore, the system's environment is also modelled to extend the models simulation capabilities. The goal of this paper is to present system requirements and their translation into formal expressions that can be used in UPPAAL to achieve verification and validation of the system behaviour.

II. Requirements

The requirements of the model are based on the requirements for the actual system from the semester project introduced in the previous section. The goal of the requirements should be to constrain the project in such a way that the results of verification and modelling will provide value towards verification of the actual system. It should be noted however that requirements not necessarily included in the verification, will be included in the list of requirements.

The requirements are categorized into performance, predictability, functionality and error detection, allowing an evaluation and discussion of each property.

Performance

- The system should read and react upon sensor data within 1 second.

Predictability

- There should not be deadlocks in the system.
- Only one sensor should be interacted with at a time.
- An activated actuator should not block execution of a different part in the remaining system.
- Actuators should not run within their set cooldown/back off time.

Functionality

- The system should be able to sense moisture, humidity, temperature, Co2 and light.
- Users are notified in case of measurement errors.
- The system should be able to provide water for the plants.
- The system should use collected sensor data to determine whether an actuator needs to be activated.
- The system should be accessible over a network.

- The system should publish sensor readings to a MQTT broker.
- Sensor measurement intervals should be definable for the user.
- Actuators can be individually controlled/invoked by a user, temporarily overriding defined system behaviour.
- The system should be scalable combining the functionality of multiple similar units.

Error Detection

- The system should report errors on invalid measurement read from moisture, humidity, temperature, Co2 and light sensor.
- Actuators should report any error obstructing them from running. Eg. A pumps water reservoir being empty.

III. Modelling

The actual system implementation works with an event system in order to be able to handle the timing in a concise, yet precise manner. A single sensor can be associated with each event, thereby allowing for different sampling periods for each sensor. Events are queued, holding the timestamp of their activation. When the system clock reaches the time of the first event in the FEL (Future Event List), the system proceeds to sample the associated sensor input, process the data if needed, and see if any other event is triggered by it, mainly enabling actuators. Actuator events are prioritized over upcoming sensor events, therefore we can abstract it by putting them right after currently processed sensor-related event before going back to the main loop.

As an addition to our formal and non-formal requirements described in the previous section, we have also formulated a wish to be able to generate the UPPAAL model from the DSL. Looking at the provided course materials, official UPPAAL documentation and previously specified requirements it became obvious that a design facilitating the extensibility required to auto-generate models had to be built into the model from the start. This approach would allow for rebuilding the model based on user requirements.

This requirement manifested itself as generic depictions of certain sub-parts of our system that can be easily modified or multiplied. This resulted in some simple templates (e.g. the templates responsible to simulate our sensors and actuators), and a more complex main template (named General in the example model in the forthcoming sections) that has generatable and uniform sub-sections. This main template is responsible of handling the behaviour and logic of the system based on a system clock, just like the board controller of the real system. However, it should be emphasised that while the templates depicting the behaviour of the actuators and sensors are models of actual physical components, this main template is merely an abstraction of the working logic of the system, and not the physical component of the board controller.

An exemplary General template, responsible for depicting the working logic of the system can be seen in the Appendix, on Figure 1.

A. *System Architecture & Main Template*

During research and experimentation it was found and agreed upon that the best practice in UPPAAL is to create a central "main" template that handles the behaviour of the system and the environmental events, and activates our sensors and actuators that we modeled to be in their own separate and independent templates. This invocation is modeled in UPPAAL using synchronisation channels. This main template should follow the logic set up in our real system, and simulate its behaviour.

The General template has a generic SystemOn state, that serves as the initial state of the template and is supposed to model an idle or resting behavior of the system. Each and every sensor declared by the user in the DSL has its own sub-section that handles the events and processes associated with the given sensor. These sub-sections are uniform in logic and working, they differ only in names, associated actuators, their events and their user declared values (e.g. sensor reading intervals, actuator working time and back-off timer etc.) which all should be extracted from the DSL during the code-generation. This means that the UPPAAL system in theory could handle an arbitrary number of sensors and actuators declared by the user, achieving the UPPAAL-model independence expressed as a secondary requirement at the beginning of this section. Additionally, every actuator has a shutdown loop originating from the previously mentioned SystemOn state, that forces the actuators to stop executing if they have been active for at least the requested amount of time.

An example of the uniformed sub-section of the main template responsible for handling the one specific sensor (here, the Temperature sensor) can be seen in the Appendix, on Figure 2.

B. *Sensor Sections & Expandability & Extendability*

As mentioned before, the General template has sensor-specific subsections in the template. In this subsection, we will look into the process of requesting, reading, and evaluation of sensory data, as well as checking for activated actuator triggers in the real system, and compare it to the abstract depiction of this behaviour in our UPPAAL model. We will show the certain states and transitions of such a generic subsection that simulate this complex process. This essentially describes the working logic of the code implemented on the board.

The idle system (that is in the SystemOn state) can take a transition into one certain sensory reading section when the timer associated with that sensor has reached or crossed the user defined threshold. After that, we request the current reading of the sensor, and check its validity. The validity intervals for each type of sensor data should be declared by the users in the DSL as well. If the measured data is outside of the validity interval, we assume it is because of random sensory error and request a new reading. If the sensory reading returns invalid data 5 times, each of them being an individual new reading, we assume there is a hardware error (the validity interval set by the user should be realistic, as we do not check it in the DSL), notify the user about the problem and its circumstances and return to SystemOn idle state. Optimally, if the data has been validated successfully, we proceed into the next execution loop, evaluate it and check for triggered values. If the reading is in the "normal" zone and we could not find any actuator event associated with the value, or the associated actuator(s) are not available, due to being on back-off timer, we once again return to our idle SystemOn state. On the other hand, if we do find that the sensory reading triggers an event of an actuator, we advance further by taking a transition that activates said actuator with user defined values (working duration, actuator back-off timer) that we extracted from the DSL. This transition activates the associated actuator by sending a signal on the actuator's synchronisation channel and we immediately return to the idle SystemOn state, while the actuator starts working for the defined time period. Note that every possible outcome of the sensory reading process (such as: invalid measurement; valid, but non-triggering measurement; and actuator-triggering measurement) ends with taking a transition back to our idle SystemOn state, resetting the timer that is associated with the sensor, enabling the system to process the next event in the FEL.

C. Actuators & Communication with User

In the previous section we have described a generic execution of sensory data processing, handled by our main template. In this segment, we will look into the templates depicting our sensors and actuators of the system, their working logic and connections to other parts of the UPPAAL model. Keep in mind, that to show the working flow of our model we will be using generic names for the states found in the templates, a concrete implementation would use extracted actuator and sensor names from the DSL.

An example of an actuator template (here, the Waterpump actuator) can be seen in the Appendix, on Figure 4.

In the figure, we can see an exemplary template of an actuator in our system. As mentioned in the previous parts of this paper, due to the need of being able to automatically generate the .xml file containing the UPPAAL model from the DSL input, we have opted to create simple and uniformed templates for our sensors and actuators, that imitate the different states and actions that the physical parts can be in and execute.

The sensors and actuators both have an idle SensorWorking or WaitingTrigger state as their initial locations respectively. In the case of actuators, we transition into the a StartActivity committed state when we receive a trigger signal via a synchronisation channel from the main template (if a sensory data has triggered an associated actuator by going over its "normal" threshold), which acts as a symbolic starting state for the current job of the actuator, and resetting the actuator timer to zero. Being a committed state, we immediately take the next transition into the ActuatorWorking state, in which the template stays for at least the amount of time that the main template has requested the actuator to work, guaranteed by an invariant. To take the transition to the StopActivity location the actuator must receive a signal from the main template on the StopActuator channel. Note that the main template can send signals on such channels from solely the SystemOn idle location only if the time required for the actuator to work for has passed. This means that in this model we can only guarantee that the actuator will work for at least the required amount of time, as if the main model is in a different location (such as checking a different sensory data), and not in the idle SystemOn state, where the looping transitions can be taken to shut down the working actuators, the actuator will continue working. This is a characteristic of the design, that we accepted on the premise that execution of processes in the system only takes negligible amount of time, and as the actuators are guaranteed to eventually shut down, this behaviour should not be harmful for the plant. The discovery of this behaviour (due to not being able to force the UPPAAL simulator to take one exact transition when multiple transitions are enabled, as we are not allowed to have an urgent channel and a guard checking a clock type variable on one transition) led us to experiment with our other mentioned model, one that simulates the environment with the help of ODEs. Despite this realisation, we still deemed our model to be a useful representation of our system's working logic, that potentially could be improved with the addition of timing constraints. Unfortunately, the working times of the actuators are the only time properties we can, to some level, guarantee, but due to not being able to extract timing statistics from our hardware components, we could not formulate timing properties for our other processes. Additionally, due to the continuous nature of the UPPAAL simulation we were not able to verify that system is deadlock free, even though there aren't states in the model where it can't advance further. These properties are checked and guaranteed in our other, modified UPPAAL model, but it is important to mention here this slight difference between our UPPAAL model and our real system.

Among the templates of sensors and actuators we can find a very similar Notify template which we created with the intent of being able to simulate the interface that the user can use to communicate with our system. Initially, we made it to be able to receive and send signals as well, to satisfy the requirement of allowing the user to override the system.

This however costed us the ability to validate that there is no deadlock in the system as the user is always allowed to start up actuators, making infinite loops in the state space that the verifier can not check through. In order to verify this property we had to exclude the ability to simulate the user's manual override capabilities from the scope of the project.

The template used to simulate the user's ability to communicate with our system can be seen in the Appendix, on Figure 5.

As we mentioned before, we have chosen to model our sensors in two different ways. The one introduced previously relies upon generation of random numbers, which was useful to check for certain scenarios in our system (such as sensor data validation), the upcoming one is using ODEs to simulate more realistically the changes in the environment. One concrete example where the difference between the two models would be significant is to choose a random number as the result of one sensory reading that triggers an associated actuator. This leads to the actuator working for a defined amount time, and when stopped, the actuator will be on back off-timer. Because of the randomness of the generated sensory data, it is possible to register greatly different, unrealistic novel sensory readings of the same environmental variable, as if the working of the actuator had no effect on it. In this case, a new activation of the actuator will not happen until its back-off timer expires, even if a novel sensory reading would normally require it. This however only causes the simulation to be not lifelike, the working of the system follows the same logic. In this manner, the simulation of environment with the help of ODEs results in a more realistic simulation of the environment for our system to sense and act upon.

D. Environment Modelling & Sensor Logic

Ensuring that a model is grounded in reality is a notoriously difficult problem, that requires a lot of parameters or historic data of the system to be known. As an example Huang et. al. has attempted to model OU44 at SDU for the purposes of minimizing the price of temperature and CO2 regulation. However even with a more informed grey-box model they could not accurately predict temperature i.e. model it correctly by taking into account all relevant parameters [1]. One such important tie to reality for modelling a plant maintenance system is the environment and specifically the local surroundings of the plant to be maintained. Attempting to capture the environmental factors of relevance, such as temperature, moisture and lighting in the model would allow an extension of guarantees for the immediate surrounding of the system, and as such allow statements for optimal growth parameters to be issued. However this type of modelling is difficult at best because of wildly differing setups that can not be generalized with precision.

As such two different approaches with differing goals have been explored for this case. One picks random numbers to get more exploration of exception cases, and the other attempts to model a small example of showing how extending simulation to the surroundings allow for more domain-specific queries and guarantees.

For a simulation to adhere to / predict natural phenomena it requires a good set of sample data from the environment sought to be replicated in the model. Additionally having real-world metrics for how long the system takes to perform certain actions is necessary to state something qualified about how these systems interact outside simulations. Due to time-constraints and deadlines these aspects have been left out of the scope here.

One of the models of the environment is a simple tick-based system used to allow parameters to be updated at a regular frequency, that allows for use of ODEs and updates of variables regardless of system behaviour. This way of modelling was inspired by David et. al. and their UPPAAL SMC tutorial [2]. Their implementation based on clocks and invariants did cause some issues so instead, as previously mentioned, ticks and integers were used to model these phenomena. The following will go over the model, refer to Figure 6 through 8 and the submitted environmental sim UPPAAL files.

To implement this a simple clock mechanism was made. The mechanism uses the invariant:

```
envTimer <= 3
```

And the following edge guard:

```
envTimer >= 3
```

The clock is then reset and a broadcast channel is used to simultaneously run a step in all environment automata. These environmental automata can use more elaborate equations to model phenomena as needed. For example the temperature is modelled by the following update statement in UPPAAL when the heater is off:

```
mT = mT + -(mT-15000)/100
```

An exemplary implementation of using ODEs to simulate the changing nature of the environment (here, the temperature) can be seen in the Appendix, on Figure 8.

Alternatively the following when the heater is off:

```
mT = mT + 250 -(mT-15000)/100
```

Likewise other phenomena can be modelled like for example sunlight, which is approximated by a sinusoid:

```
t = (t < 360) ? (t + 1) : 0,
_sL = fint((sin((t/360.0)*2.0*3.1415)+1)
    *50),
sL = _sL
```

Potentially any number of parameters could be pulled into the expressions and as such, any system of ODEs should feasible to be modelled in UPPAAL similarly.

The previously introduced randomised approach uses a simple automaton. When prompted for a sample a random number in a set interval is generated using a SELECT statement like this:

```
e : tempRand
```

A template depicting a sensor (here a Temperature measuring sensor) utilising the random-number generation to produce sensory reading data can be seen in the Appendix, on Figure 3.

The sensor automaton reads out the value for the system to access when a sample is performed / requested. This is done like the following:

```
tempData = e
OR
tempData:= fint(floor(mT / 1000))
```

IV. Verification

In order to establish the validity and correctness of the system, and to verify whether the system satisfies all key requirements necessary for its success as an agricultural product, the UPPAAL model and its properties was tested for reachability, liveness and safety as defined by Behrmann et. al in "A Tutorial on Uppaal" [3] . Focusing on these three elements, we can check and ensure that the system behaves as intended in regards to the requirement specification and our expectations. All the queries presented in this report are written using the UPPAAL query language [4]. For this section refer the submitted UPPAAL files.

As mentioned in the previous sections, some of the most crucial and key properties of the system is its ability to observe and gather data from its surrounding environment and being able to automatically and correctly act upon the gathered sensor data. To demonstrate whether reachability, liveness and safety are satisfied (or not) in our system, based on our requirements, the interactions between the temperature sensor and the water pump actuator will be used as an example. Even though the system is composed of multiple different sensors and actuators, using a single sensor-actuator relationship is sufficient as an example of how the system was verified, due to the similarities of the rest of the system.

A. Reachability

Reachability is the simplest property to check out of the three mentioned above, and is often used as an sanity check for a model. The goal of reachability is to verify that some point in the model's/system's execution, a specific state can be reached. If the model is valid, all states are eventually reachable at some point in time given the necessary conditions are being met. All of the states of our model has been verified using the query structure below.

```
E<> "Template Name"."state"
```

B. Liveness

We are testing for liveness to ensure that some of the model's states are eventually reached and are satisfied. It is important that the pump eventually reaches an active state. The example implementation in the DSL has a requirement for the pump to turn on for 15 seconds when the moisture level is below 20 units. This can be specified as the following:

```
Trigger: Pumpe for 15s @ 50% when Fugt <
    20 back-off 3min
```

To check this property the following query is used in the verifier:

```
(moistData < 50 && SysTimer < 5000) --> (
    PumpAct.PumpActive && SysTimer <
    10000)
```

See Figure 9 for running of this query. Which for the system in question is satisfied.

C. Safety

Since inaccurate sensor readings can cause unwanted behaviours in our system, it is important to check and ensure that this property is handled appropriately by the model. A safety requirement such as "Invalid measurement read from moisture, humidity, temperature, Co2 and light sensor (should cause an error)" needs to be detected and handled properly, ensuring that the actuators cannot be triggered falsely due to error in the sensors. Using the following query we can check if this property is satisfied:

```
E<> Main.EvalTempData imply
    tempMeasureError <= 5 && tempData <=
    tempMaxValid && tempData >=
    tempMinValid
```

Furthermore, if more than X amount of readings fall outside the acceptable range for a specific sensor, the system should stop reading from that said sensor and switch to an invalid data state. The following query checks whether this system property is satisfied and that the system switches to the correct state:

```
A<> ((tempMeasureError >= 5) imply Main.
    InvalidTempData)
```

Lastly, it is also important to ensure that the actuators are not running within their set back-off time or whenever some predetermined condition is not satisfied. The query for checking this property:

```
A<> Main.TempPumpEvent1 imply (pumpTimer
    >= pumpBackOff && tempData < 10)
```

In the case of our system, all of the above mentioned properties are satisfied.

D. Difficulties & Shortcoming

The simulation potentially never terminates if there is no ending deadlock or no end time specified in the query, and as such any query for the system using the approximated environment automata have this qualifier for the query. This is because of the continuous nature of the simulation and as such UPPAAL can not find a place in the state space to terminate properly. Running the query below in the UPPAAL verifier, will result in a "out of index system error".

```
A[] not deadlock
```

However we can limit the execution time, by only taking the first 10000 steps into account as a potential workaround to the above stated problem. By doing so, we can guarantee that the system is free of deadlocks, up until the specified time. The extended query for checking deadlocks bounded by time, can be seen below:

```
A[] not deadlock and SysTimer < 10000
```

This effectively limits the state-space and ensures a reasonable amount of states have been traversed.

If a guarantee about a environmental factor is of interest, it could be checked with the following structure of query:

```
A[] mT > 18000 && SysTimer < 10000
```

See Figure 9 for running of this query. For the system in question this property is not satisfied, see Figure 9, because the back-off disallows the heater turning on again after the first cycle. Setting the back-off to 0, i.e. allowing the heater to run exactly when needed results in the query being "satisfied". The query still fails but only because of the SysTimer exceeding 10000, and to make sure the property actually holds UPPAAL is set to return the shortest trace that does not satisfy the property. This trace that is returned is at timestamp 10002 and a temperature of 28.56 °C. Keep in mind that the values in the model here are chosen arbitrarily, and that to guarantee any behaviour in the actual system some data about this would have to be collected and analysed.

E. Requirements mapping

Table II illustrates a mapping between all of our requirements and their respective UPPAAL queries. Requirements that have been omitted or where a query cant be created with the current state of the model are marked as "No query".

V. Discussion

A. Result Interpretation

Through the modelling workflow and the verification of the model, we have received results that can be interpreted in the context of the project and the requirements. The most important factor in order to verify the validity of the project is the results gathered through the verification tests. Especially the tests based on the initial requirements of the model.

As seen on Table II and explained in Section IV, the verification process has shown that we, to a high degree, have managed to design and implement a model fulfilling the requirements for the system. This proved to us that the system's major components worked on paper and it should be possible to implement as a real physical system.

Most of the initially stated requirements have been included and satisfied in the model, but certain requirements have been omitted mostly due to time constraints and relevance to the results we wish for. In this section we will go through some of these requirements and the reasoning for their omission.

One requirement that was omitted from the modelling is the functionality of publishing sensor readings after they have been measured. As with many of the other omitted requirements, this is left out not because of feasibility issues, but rather because of it not being beneficial to what we are trying to achieve with the act of modelling the system.

Another requirement that was not necessary to implement in the model was the ability for the user to override the current flow of events and activate an actuator directly. This would be very useful in the actual system but would provide very little towards verifying the concept of the system through the model.

The ability to scale systems by combining multiple units along with their sensors and actuators would provide required functionality for larger agricultural projects. While this would enable us to reach a previously uninterested section of users, this requirement was deemed out of scope and was therefore omitted from the model.

B. Future Work

To further enhance the model's capabilities, several improvements could be made in the future. First, adding stochastic time scheduling for representing the networking, sampling and computing times would allow us to remove a layer of abstraction to get closer to the real system's behavior. However, this would necessitate extensive measurements to work up random distributions for these points. This would allow for greater adaptability from the model, especially regarding user input and if combined with the code generator of our Semester Project.

Our system as we aim to implement it would use code generated from our own Domain Specific Language, enabling the user to configure the system to fit his needs. This is where UPPAAL's full power comes into play. Using the same DSL code written by the user, we expect to generate UPPAAL XML code as well in order to produce a matching model to verify its configuration viability and/or usability. This could be also be extended to queries, that would be generated using this language as well to ensure that the system remains consistent with the relevant constraints, both in terms of sanity and usability regarding user requirements.

These requirements could also take new forms that were not part of the scope of this prototype, especially in terms of resources, namely processing power, memory usage, power consumption or broadband usage. Enabling the model to verify resource usage would prove to be very relevant to our project as it is intended to be scalable and adaptable to fit the user's needs, despite the small scale of this first prototype that is meant to be plugged in at all times. It would however require heavy testing and measurements to produce usable results for resource usage verification. Calculation alone would probably not be a good fit to estimate the consequences of the wide array of outcomes enabled by our system's adaptability through its DSL. This functionality would also integrate well with the stochastic time scheduling as described above, since resource consumption is directly linked to the amount of time spent on activities such as network communication, processing or actuation.

C. Hardware

The model has been made in order to approximate the real system as closely as possible. There are a few minor details that have not been adapted between the model and the system. First and foremost the model reads sensors without any delay, while the real system has some delay as data needs to be transferred over wires. We have decided that the delay present in the real system is too minuscule to have a real influence on any results as sensor readings happens so fast that they can be perceived as happening in real time. A potential improvement on the current model could be to implement this time constraint, but this would vary between sensors and require further work even though the quality of the model would not be increased significantly.

Another difference is the sensor reading temperature and humidity. In the real system a single sensor, a DHT11, detects both temperature and humidity while the model distributes this into two separate loops. We could have represented the real system by modelling the DHT11 sensor accurately, but this would restrain the model as we want a general representation of what the system should achieve instead of a 1:1 representation of the hardware we test the implementation on.

VI. Feedback

As part of the project, groups were required to provide feedback on another predetermined groups work. This has the purpose of bettering the students work and ability to put into context the models without working on or seeing all aspects of the project in question. The feedback given about this project will be briefly outlined to provide context for how the work has progressed and what points provided the most interesting discussions. The provided feedback commented on environmental modelling being a nice thought and attempt to provide some more tangible results. Additionally bringing hardware into considerations and efforts to make system and model similar in function by using code-generation with some abstractions is a nice touch. Missing at this point in time was the table relating queries to their respective requirements as specified in the project, which has since been rectified. A concern with getting real-world values for timing of processes and environmental modelling was appropriately raised and addressed as a desire. This however is difficult when basing the models on a generic DSL that is meant to support differing hardware, configurations and settings.

VII. Conclusion

While the project succeeds in proving most points it is aimed at, it also revealed a bunch of issues, potential extensions and possible improvements. The model provides a nice reflection of the most important aspects of the function of the actual system, and allows for liveness and reachability testing. These aspects are however made somewhat trivial because of the design chosen, and this was trivialized by building safety into the system from the ground up. It did however provide some considerations upon how the actual system was to be implemented, based on how the model looked and behaved initially. This corrected some erroneous assumptions in the design early in the implementation and designing phases of the project it is based on. It also proved that the most interesting points of the modelling and therefore guarantees lay in the timing, energy / resource and environmental sides of the project. It did however prove to be well outside scope to actually take these parameters and modelling practices into consideration in this iteration of the project. The two modelling approaches attempted here did however help in specifying transition requirements and system design for the implementation, and as such it has helped in the fulfilment of the project it is based on. Finally the project has served its purpose by allowing us to have first-hand experience using a modeling and verification tool by working with UPPAAL.

References

[1] S. Huang, K. Filonenko, Y. Zhao, T. Yang, T. Xiong, and C. Veje, "Indoor Climate Modelling and Optimal Planning With Respect To Electricity Prices," in *2021 IEEE Power Energy Society General Meeting (PESGM)*, pp. 1–5, July 2021. ISSN: 1944-9933.

[2] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal SMC tutorial," Aug. 2015.

[3] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems* (D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Bernardo, and F. Corradini, eds.), vol. 3185, pp. 200–236, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. Series Title: Lecture Notes in Computer Science.

[4] "Requirements Specification :: UPPAAL Documentation." Available at https://docs.uppaal.org/language-reference/requirements-specification/.

[5] "Language Reference :: UPPAAL Documentation." Available at https://docs.uppaal.org/language-reference/.

Appendix

A. *Work Distribution*

- Christian
  - Environmental Modeling
  - Environmental based verification
  - Structuring Environmental Main Template
- Renaud
  - General conception and implementation of the model
  - Coordination with the real system and its algorithm
- Mathias
  - Requirements elicitation, specification and definition
  - UPPAAL queries
- Daniel
  - System verification and validation
  - UPPAAL queries
  - General model input
- Morten
  - Requirements elicitation, specification and definition
  - Requirement and result interpretation
  - Hardware requirement coordination
- Ábel
  - General conception and implementation of the model
  - Random-number Based Modeling
  - UPPAAL queries
  - Verification

B. *Glossary*

TABLE I
Glossary of used terminology

Term	**Explanation**
DSL	Domain Specific Language used to specify the components and behaviour of the system.
V&V	Validation and Verification, the process of making sure an abstract model of a real-life system accurately represents the real system, and if the model corresponds to its specification and requirements, respectively.
Back-off Timer	A possible behavioural variable of an actuator in our system, it sets a time limit for the actuator that it can not be activated again until it's reached.
FEL	The Future Event List contains the events that will happen during the simulation, queued in ascending order based on their timestamp of firing.

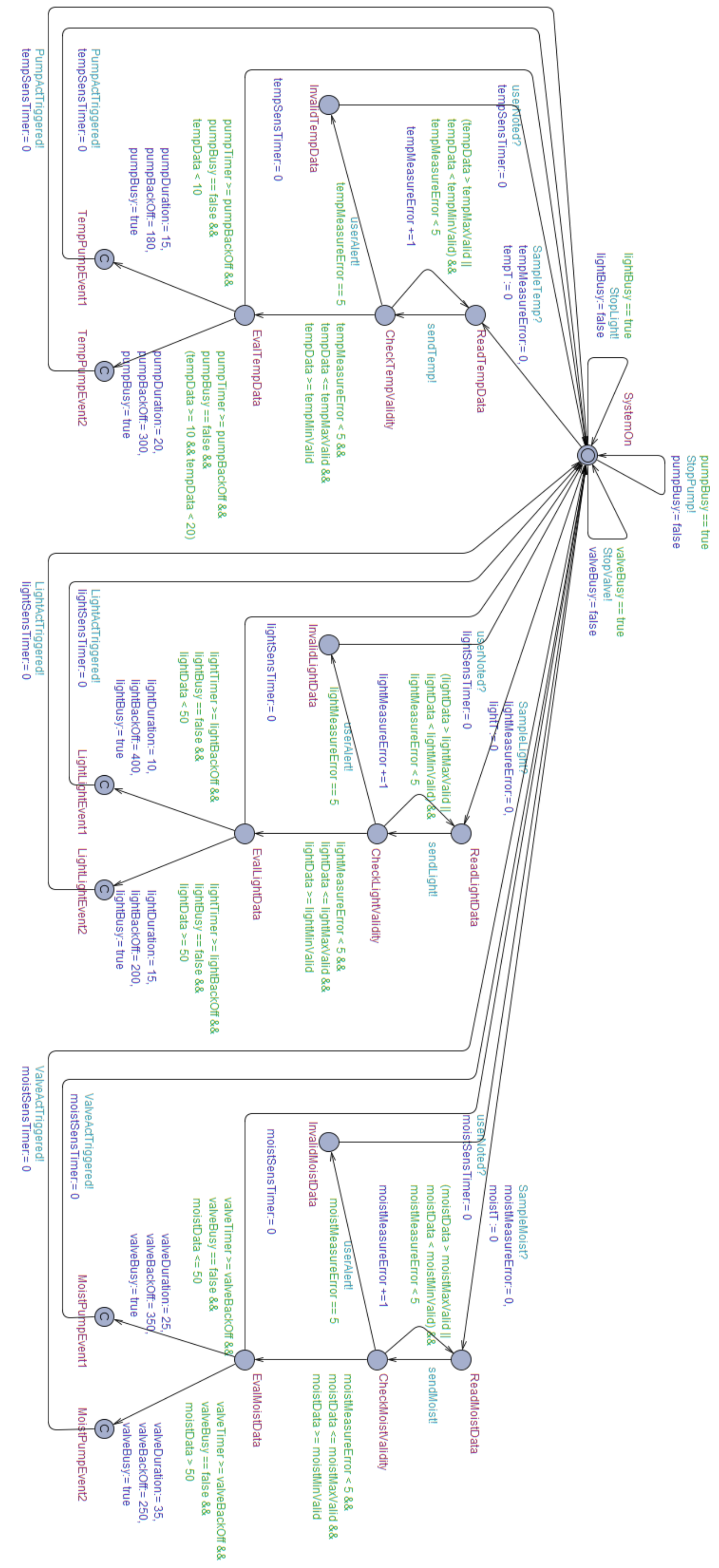


Fig. 1. Random-number General Template

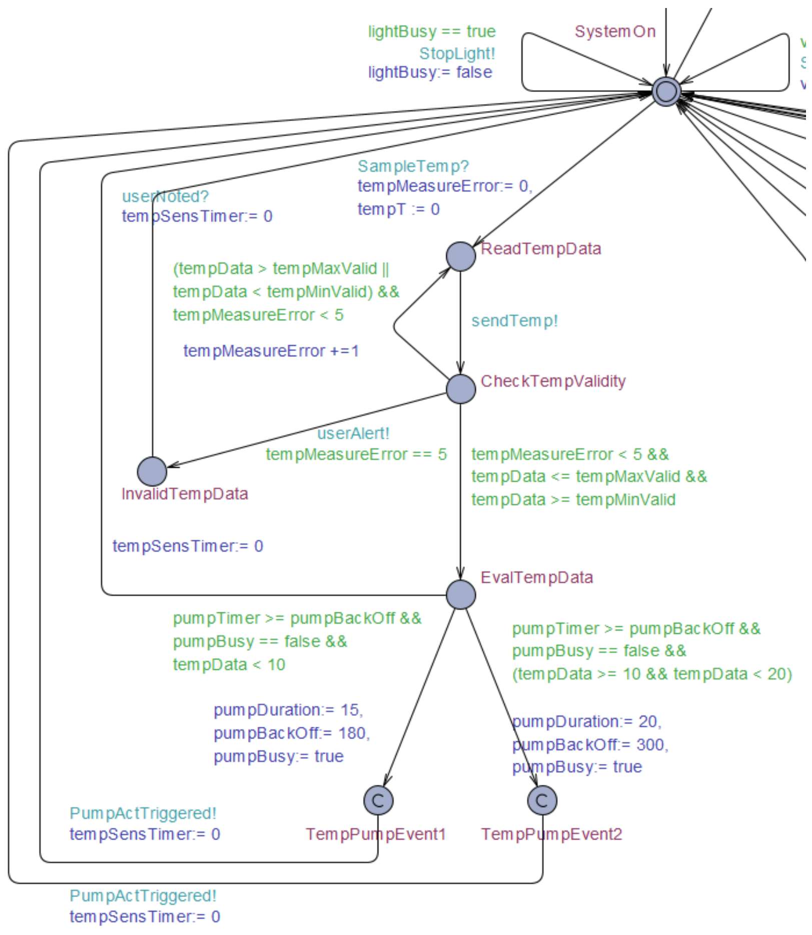


Fig. 2. Generatable sub-section of the main template responsible to handle sensory working

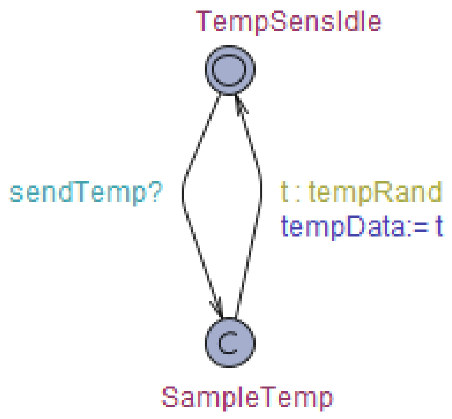


Fig. 3. Temperature Sensor Random-number Template

TABLE II
UPPAAL QUERIES

ID	Requirement	UPPAAL query
Performance		
1	The system should read and react upon sensor data within 1 second.	— No query —
Predictability		
2	There should not be deadlocks in the system	A[] not deadlock
3	Only one sensor should be interacted with at a time	A[] ((TempSens.SampleTemp and not (LightSens.SampleLight or MoistSens.SampleMoist)) and (LightSens.SampleLight and not (TempSens.SampleTemp or MoistSens.SampleMoist)) and (MoistSens.SampleMoist and not (TempSens.SampleTemp or LightSens.SampleLight)))
4	An activated actuator should not block execution of a different part in the remaining system.	E<> PumpAct.PumpingWater and Main.SystemOn
5	Actuators should not run within their set cooldown/back off time.	A<> Main.TempPumpEvent1 imply (pumpTimer >= pumpBackOff)
Functionality		
6	The system should be able to sense moisture, humidity, temperature, Co2 and light.	E<> Main.ReadTempData or Main.ReadLightData or Main.ReadMoistData
7	Users are notified in case of measurement error.	A<> (Main.InvalidTempData or Main.InvalidMoistData or Main.InvalidLightData) imply UserNot.UserAlerted
8	The system should be able to provide water for the plants	E<> PumpAct.PumpingWater
9	The system should use collected sensor data to determine whether an actuator needs to be activated.	A<> Main.TempPumpEvent1 imply (tempData < 10)
10	The system should be accessible over a network.	— No query —
11	The system should publish sensor readings to a MQTT broker.	— No query —
12	Sensor measurement intervals should be definable for the user.	— No query —
13	Actuators can be individually controlled/invoked by a user, temporarily overriding defined system behaviour.	— No query —
14	The system should be scalable combining the functionality of multiple similar units.	— No query —
Error Detection		
15	The system should report errors on invalid measurement read from moisture, humidity, temperature, Co2 and light sensor.	E<>(tempData > tempMaxValid or tempData < tempMinValid) and tempMeasureError <= 5 imply Main.ReadTempData
16	Actuators should report any error obstructing them from running. Eg. A pumps water reservoir being empty.	— No query —

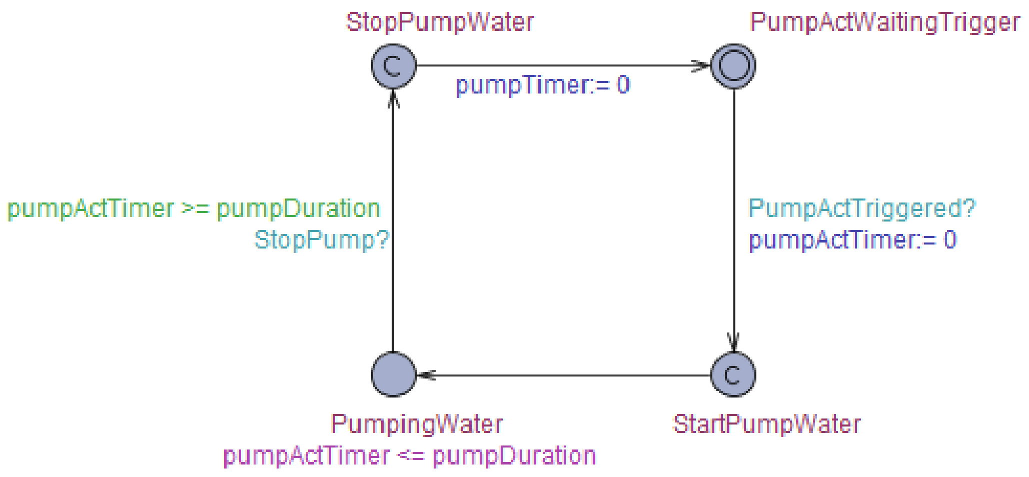


Fig. 4. Waterpump Actuator Template

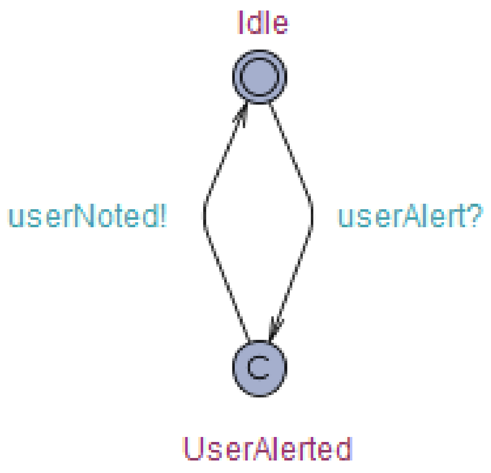


Fig. 5. User Interface Template

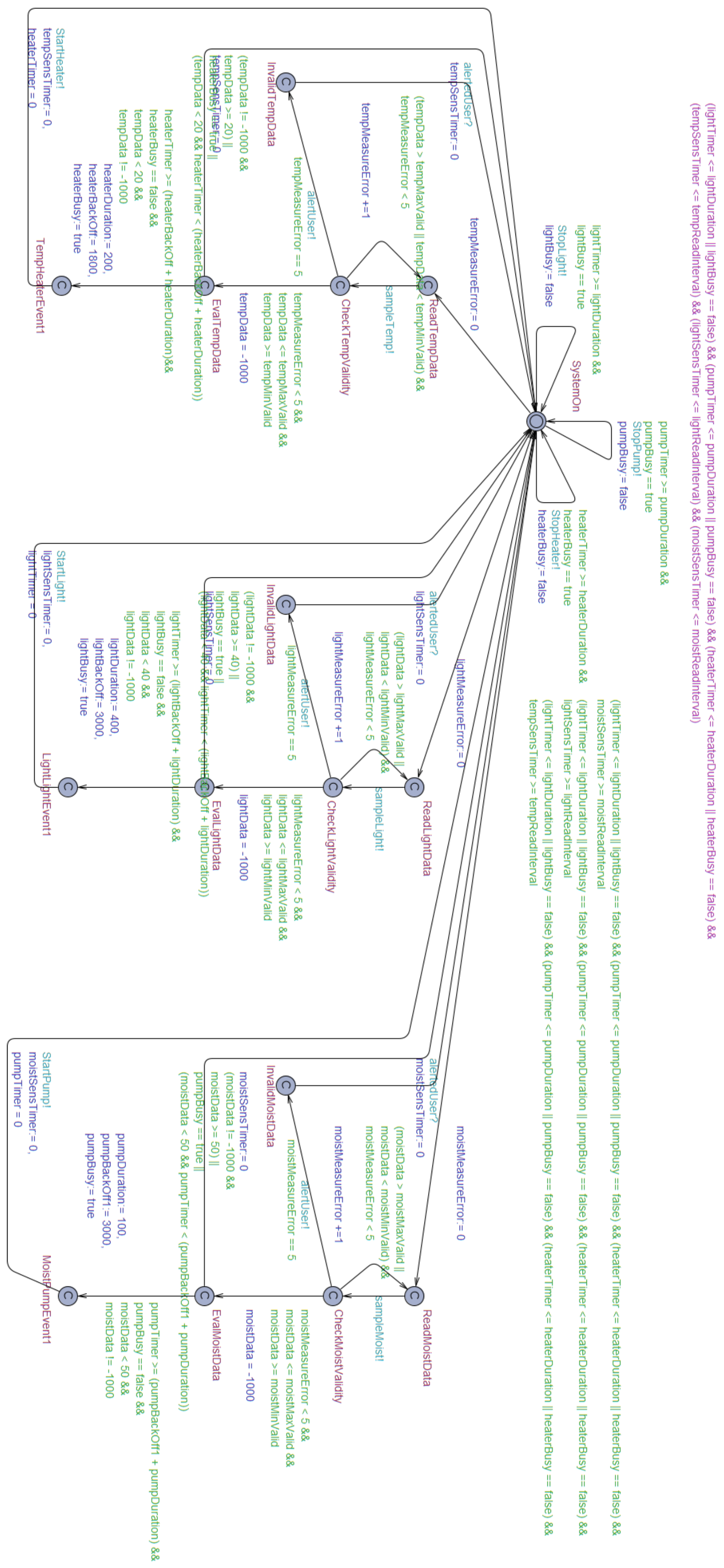


Fig. 6. Environment General Template

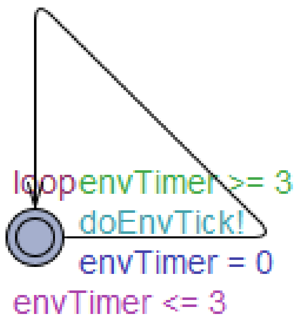


Fig. 7. Environment Timer Template

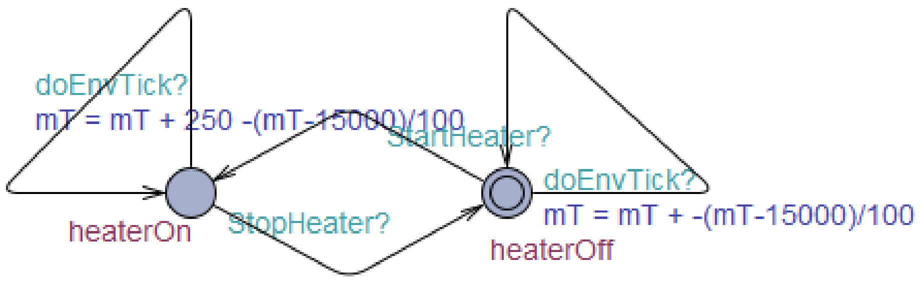
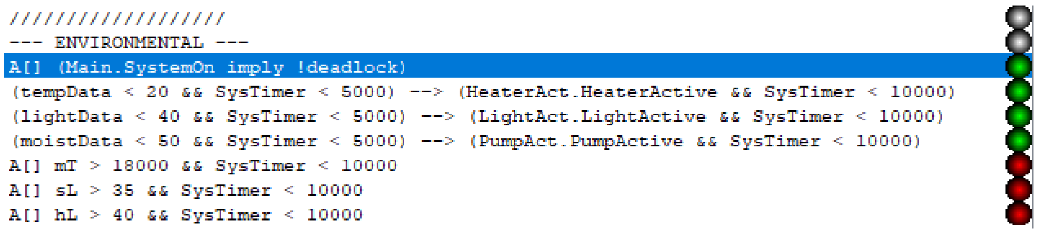


Fig. 8. Environment Temperature Template



```
///////////////////
--- ENVIRONMENTAL ---
A[] (Main.SystemOn imply !deadlock)
(tempData < 20 && SysTimer < 5000) --> (HeaterAct.HeaterActive && SysTimer < 10000)
(lightData < 40 && SysTimer < 5000) --> (LightAct.LightActive && SysTimer < 10000)
(moistData < 50 && SysTimer < 5000) --> (PumpAct.PumpActive && SysTimer < 10000)
A[] mT > 18000 && SysTimer < 10000
A[] sL > 35 && SysTimer < 10000
A[] hL > 40 && SysTimer < 10000
```

Fig. 9. Environment Model Queries