Semester Project Technical Report

Model-Driven Software System Development: Formal Verification & validation of IoT-based Plant Health Monitoring System

1 SUMMARY

This report will present the work done for the course "Semester Project in Trustworthy Systems" of the 2nd semester of the Master in Software Engineering. Explanations regarding terms and abbreviations can be found in Appendix A.

The objective of the project was to develop a system for plant health monitoring, using IoT technologies and sensors, combined with the implementation of a custom made Domain Specific Language for easier configuration, providing a scalable and flexible system as well as automatic generation of code and UPPAAL models.

Furthermore, an auto generated system based on an exemplary implementation of the Domain Specific Language has been modeled and verified using the UPPAAL model checker, providing us with valuable insights on the reachability, liveness and safety properties of the system.

The contributions from each of the three mandatory courses of the semester are listed below.

- Model Driven Software Development
  - Meta-model design
  - Implementation of DSL in Xtext.
  - Code generation using Xtend.
- Software System Analysis and Verification
  - Modeling and verification in UPPAAL.
  - Using modeling and verification for debugging and refinement.
  - Using computer-aided design to guarantee the correctness and dependability of the system.
- Software Technology for Internet of Things
  - Designing and implementing an IoT system.
  - Interfacing with sensors and actuators.
  - Low-level C/C++ programming of an ESP32 Microcontroller unit.
  - Data transmission using MQTT.

2 PROBLEM AND OBJECTIVE

When growing plants both at home and in an industrial setting, soil and other certain environment parameters are hard to maintain without exact and frequent measurements. Humanity as a whole depends on the agricultural sector to provide us with food, which means that it is important for systems supporting this sector to be trustworthy.

The overall objective of this project is therefore to reduce uncertainties regarding the care of plants both at home and within the agricultural sector. This should be achieved through development of a pretotype system that (1) can monitor a plants health, (2) can be scaled into implementation in a small agricultural setups and (3) can be configured through a concise and simple Domain Specific Language.

3 PROBLEM DESCRIPTION

In a world with a continuously increasingly population, food security has become more of a concern. The ability to ensure people have food readily available has become more of a logistical challenge while climate-change and war change the world and introduce more and new challenges to the agricultural sector. Ukraine and Russia happen to make up a third of grain production in the world. War breaking out between them [1] is polarizing and put the food supply-chain at risk, and any countries downstream have to grapple with securing enough food to not disrupt people's lives [2].

Allowing more people to have automated home-grow, hydroponics or even indoor farm setups would allow for more food to be produced without people needing to be proficient in agriculture.

However, maintaining a plant's optimal growth parameters accurately and constantly is a difficult problem. Proper management can allow for better yields in crops and allows for usage of less resources without adversely affecting the output of a farm [3]. Interestingly, it is even possible to marginally increase efficiencies if plants are stressed correctly by small deficiencies in irrigation or nutrient supplies. This however obviously has to be carefully balanced to still allow the crop to grow properly and not suffer ill effects from such practices. Any small deviation from optimal growth parameters can and will impact yields. Prăvălie et. al. [4] found a negative correlation between temperature and yield of 1 ton/ha/year of corn per degree Celsius. Mismanagement as such needs to be eliminated, preferably by having proficient individuals share setups and configurations.

A solution that allows for shareable setups and configurations would allow for amateur setups for growing fruits, vegetables or other crops. Shareable setups would require a common board-design or schematics and identical sensors, and these factors may change

and be improved upon, but are not easily upgradeable. The problem to be solved as such lies in the configuration.

4 SOLUTION APPROACH

The solution must include certain components in order to fulfill the requirements presented in Appendices B and C. These components should be intertwined with each other and work in unison in order to achieve a solution to our problem statement. The first part of the solution is the hardware which should consist of sensors and actuators capable of gathering data on the plants' environment and acting on it, respectively. The hardware needs to include a microcontroller in order to link the sensors, actuators and publish the collected data to the MQTT broker over the network. It should also cooperate with the second component of the solution, the DSL. The code generator needs to be able to generate code for the microcontroller based on user configuration through the DSL. How the DSL is developed will determine how easy the system is to use and how powerful and configurable the system is. The more configurable the DSL is and the amount of scalability it supports will determine how large a part of the agricultural sector will gain something from adopting the system. The DSL should support the setup of the MQTT data transaction protocol within the microcontroller in order to extract and visualize data without having to access the individual hardware setups physically. The MQTT protocol will feed data to the sequential database, InfluxDB [5], which will then be presented to a user through a graphical interface called Grafana [6]. In order to connect the database with the MQTT broker Telegraf will be utilized, since it works as an adapter between these components. In Appendix D a diagram showing theses components and their relation to each other can be seen.

A separate but important part of the system is the formal verification done through modelling in UPPAAL. This part aims to verify that the logic of the system works as intended. These models should be automatically generated based on a DSL too, in order to give users access to the results of formal verification of their system without the need for them to go through the modelling process.

5 SOLUTION DESCRIPTION & RESULTS

5.1 Hardware

The hardware design of the system is based on around a microcontroller, sensors and actuators. The microcontroller has the responsibility of executing DSL generated code in order to coordinate sensor data and actuator activation. The sensors gather data about the growth environment of the plant, while the actuators react to this data as specified by the user.

5.1.1 ***Microcontroller***.

The microcontroller unit used for this project is an ESP32 Wrover E (Appendix P), which provides a low-power system with integrated WiFi capabilities. For a system like ours, it is more than capable of running the necessary logic to run the system. The ESP32 platform provides a total of 39 digital pins, which means that a single unit has the capability of monitoring a wide array of sensors maintaining several plants.

5.1.2 ***Sensors***.

The system uses different sensors to provide measurements from the soil and air around the plants. The sensors needed for the system to maintain plant health should include moisture, temperature, humidity and light sensors, which of course comes in many different variations. The sensor setup can be seen in Appendices Q and R.

Multiple sensors have been used during this project as a means of testing the capabilities of the system. Most of these were analog sensors, providing simple interfacing at the cost of having to manually adjust values to a set of parameters, since analog outputs are varying voltages with no real units of measurement applied during sampling. The only sensor with a digital output was the DHT-11, responsible for humidity and temperature measurements. Interfacing with a digital sensor is stricter than interfacing with an analog sensor since specific libraries or large code chunks are needed in order to read the output. As an upside, the readings provided by this type of sensor are given as specific numeral values related to a unit of measurement instead of a simple voltage.

An exception to the sensor selection is the CO2 sensor which optimally would be a digital sensor, but due to pricing was exchanged for a potentiometer which is an analog sensor, capable of being configured to provide a specific amount of voltage between zero and the reference voltage (5V for our system).

5.1.3 ***Actuators***.

The system should support actuators that help maintaining proper growth conditions for the plants. This could include pumps for water and nutrients, growth lamps and ventilation systems for air quality control. The only currently implemented actuator is a water pump, that is be capable of pumping water from a reservoir into a plant's soil upon activation, in order to increase the moisture level of the soil.

The final hardware setup can be seen in Appendix R. A more detailed overview of the hardware circuit can be seen in Appendix S, in the form of a KiCad sketch of the circuit.

5.2 Microcontroller's software

The microcontroller board holds the necessary logic to sample the data coming from the sensors, enable the actuators when necessary, and exchange data through the MQTT broker. In order to achieve this, we have chosen to implement it in C/C++ code for ease of low-level system management, especially memory space. In fact, the code is mostly C code in itself, but C++ functionalities were needed in order to include the Arduino libraries for pin handling and the PubSubClient[7] library for MQTT message publishing. Since we are working in an embedded system that might be used in a restrictive environment, memory usage control is of high importance. Despite our pretotype being meant to be plugged in at all times and connected to fast WiFi, future use of the system could involve it in different configuration where data has to be accumulated in memory and is only sent after extended period of times, so this is why the code was built from the beginning with such concerns in mind. Additionally, power consumption could also become a concern in such settings, therefore it is essential to avoid superfluous computing, mainly ongoing looping between the processing of the

events.

One of our end-goals is to generate the fully configured code using our custom DSL. To achieve this, we first implemented a proof-of-concept template code to serve as a structure for the code generator using the DSL. In order to limit the generated code to a coherent minimum and ease the development process of both the ESP32's code and the code generator, the code running on the microcontroller was not designed with a monolithic approach. Instead, it has been dispatched into:

- a core library containing the framework and data structures needed to run the microcontroller;
- external sensor-specific libraries for sampling sensor input;
- external actuator-specific libraries for enabling actuators;
- a general algorithm containing the configuration sections and the main logic of the microcontroller — this is the only file that is meant to be generated.

In order to define the framework that will bring our system into action, we must beforehand make sense of the concepts that it needs to employ. First, the Future Event List (FEL) is at the core of the program and consists of an ordered linked list of timestamped events that will be considered one after the other. Events within this list are used to hold the time when to sample sensor input, publish data to the MQTT broker, fetch data from a datasource and disable actuators once they have been enabled for long enough (i.e stopping the pump from watering the plants after activating it for 10 seconds). The internal clock of the ESP32 renders this possible, as we can use it to measure the passing time without the need to be synchronized to real time, and therefore use it as a reference for the relative timing of the events, which are always happening on a periodic or timed fashion. Additionally, this also allows us to put the ESP32 in deep-sleep mode in between the events in order to drastically reduce power consumption [8], since we know when the events are supposed to happen. However, due to time constraints, this functionality hasn't been implemented in our pretotype and is left as possible future work.

The events are processed sequentially at their scheduled times as specified through the configuration done with the DSL, the associated actions depending on the event type. A configuration code excerpt can be found as an example in Appendix X. Sensor sampling events are the most important ones, as they are meant to be the main source of data in the system. They are linked to Sensor data structures holding the pin number where the physical sensor is connected to the microcontroller and a function pointer to the sampling function W configured for the associated sensor. This allows us to separate the logic behind the sampling in external sensor-specific libraries, so that we can ensure compatibility and separation of concern. We have created these libraries for every sensor we used so the generated code only needs to include those that are necessary for the specified board configuration.

Using the linked sampling function to generate the data, we then store it in the corresponding data structure buffer member W, which is an array of DSL-specified length to allow the user to set how much memory space should be reserved to store each sensor's data, as some sensors might require more frequent sampling than others. If the user also specified a conversion function through the DSL, it will be converted using it before storage. This is to enable the user to translate sensory input data to human-readable values, for example to convert measured voltage on the sensor's pin to Celsius degrees.

The sensor data structure also holds an array of triggersW, each defining the user-defined condition function to enable the associated actuator, and for how long. After each sample, the data is directly inputted to each of those to determine the actuators to activate if needs be, therefore ensuring that there is no delay between the sensing and the actuation.

MQTT publish events are the second kind of event, mainly consisting of publishing the data accumulated so far from the related sensor to a MQTT topic every fixed period of time, both DSL-specified. Data that failed to be sent is cached, so it can be sent during the next attempt.

As specified in the requirements, the microcontroller should also be able to receive information, such as user input for manual actuation. This is done through datasource lookup events that mainly set the ESP32 as a MQTT subscriber to a command topic to receive such data, though other means of transmission could be envisioned. It uses a callback function to check the related triggers' condition with the received data for eventual actuation. Due to time constraints, this functionality hasn't been fully implemented and is left as a possible extension.

After processing a sensor sampling event, a datasource lookup event or a MQTT publish event, its memory space is freed to avoid running out of memory at some point and a new event is created and added to the FEL, timestamped according to the DSL-specified period for it.

The last type of event isn't a periodic event but a timed one. Actuator disabling events are created when an actuator is enabled, timestamped according to the duration of said event. To turn on and off an actuator, its associated data structure embarks externally defined actuator-specific functions in a similar fashion to the sampling functions for sensors. Those are also defined in separate custom-made libraries, to allow for greater modularity of the code.

5.3 DSL

To allow the user to customize their system, a DSL was created for the project. This is essentially a way of declaring the components to be used in the system, using a grammar defined by us. To facilitate this, Xtext was used as a framework upon which the domain-specific configuration language was built that we use to define the system and later to generate code from to run on the microcontroller. Initially, to establish a baseline feature-set, an example file containing a valid configuration for the system was made. This allowed for clearer communication about the subject and set expectations for how the flow from configuration to code was going to be made. The example model can be seen in full in Appendix E.

It has been decided that only one Unit can be defined per file and that there will be no inter-connection between Units supported by

either the system implementation or the DSL. Each Unit consists of several attributes. An attribute can be, for example, a sensor or actuator declaration, a trigger defining the logic of how the system will respond to a certain input, or the physical pins that other attributes (sensors and actuators) are associated with.

An excerpt, consisting of selected lines from the grammar, defining the structure of the language can be seen in Appendix F, from which some of the structure becomes visible. From this excerpt it is noticeable for example, that the grammar has built-in support for conversion of readings, as mentioned in section 5.2, into a more useful value for known parameters relevant to the domain. This conversion consist of a small mathematical expression that is translated into code along with the other aspects of the language.

During the process of implementation, the grammar was always developed up against an example configuration that was intended to be valid and correct, and as such testing was fairly limited but quickly and easily performed. Additionally, any problems or invalid configuration being interpreted without error was addressed by further refinement or inclusion of validation rules, implemented in Xtend to ease development [9], to help warn the user of impossible or erroneous input. An excerpt of these validation rules can be found in Appendix I.

5.4 Code Generation

Once the user has configured and defined their desired system using our custom DSL, the next step is to generate code that controls and binds the physical system together. This is done using Xtext's built-in code generator feature, allowing us to create code for the sensors and actuators dynamically, based on the configuration the user has defined. Using the approaches described in the book [10] along with a predefined architectural template we created to help us aim for the desired structure of the system, we were able to take the parsed DSL and generate the necessary controller code with Xtend [9].

We used the architectural template as a blueprint to what the code file should adhere to once it ran through the generator. This made the development process much faster and easier as we had a specific structure to follow. Additionally, to help with the development, a "helper" Xtend class called CodeGenHelper was created. This class contained values of static variables, time conversions, methods for setting and retrieving specific values and logic to translate sensor data conversions using mathematical expressions. It is also in this class that user-defined values and declarations are getting extracted and saved into maps for later use, as seen in the excerpt in Appendix H. Specific values needed for the code gen can then be retrieved, by using the sensors/actuators type or identifier as a key.

Even though the auto generated code is not meant to be edited by the user and does not need to be humanly readable, it is recommended to strive after producing clean code and best practices, as this approach allows us to more easily debug and understand the generated microcontroller-code in case of errors, bugs and configuration problems. An example of this is the use of comments and annotations in the code base. By encapsulating and grouping larger and more important code chunks together, it is easier for us and the user to get a better overview and locate specific parts of the logic. Appendix G shows a small example of this.

Since the code generator was designed based on a predefined architectural template, testing whether the code-generator produced the desired output to a specific DSL was fairly easy. The two files were compared to each other after each code generation, and if any mistakes or unpredicted diversions were detected, the code generator was updated and corrected accordingly. Additionally, the most important test was to run the auto generated code on our physical system testing whether it behaved as intended, and it worked (see section 5.7).

5.5 Model Generation

As mentioned previously, a secondary goal of ours was to be able to auto-generate UPPAAL models based on the users' input in the DSL, to be able to formally verify certain properties of the system. To satisfy this requirement, we extended our code generator with the ability to write files compatible with UPPAAL. UPPAAL uses a few different file formats, easiest of which to manipulate and build are the XML file. This has the added benefit of also being the easiest to import and the most used format. To facilitate the generation of such files, a helper class with the XML relevant code was used to help extract this from the code defining the system. An excerpt of this helper class can be seen in Appendix J. This implementation has the purpose of keeping track of all things necessary to successfully build the required documents for UPPAAL. It does however require some structured code to use as it only serves to abstract unnecessary details of the underlying XML away from the user. This abstraction was introduced to speed up implementation of translation from DSL to model, and as such allow more time to be used improving the model itself before cementing it in the code-generation. This allowed for a more parallel workload to take place. An excerpt of usage of the helper can be found in Appendix K. As it can be seen, the generator is only concerned with translating the structure coming out of the parsing and linking steps in Xtext [11].

The code generation of the UPPAAL model depicting the user-defined system was done in a similar manner to the board-code generation. Using the helper functions we have set up in the previously mentioned UPPAAL.xtend file (Appendix J), the code generation itself is done by iterating through all the attributes (refer to Appendix E) that the user has defined, which can be different physical parts of the system (most notably, actuators, sensors or triggers). As a result of having defined a very uniformed and extendable UPPAAL base-model for our system previously, by looking at the type of the current attribute, we can create the necessary templates, variable declarations, locations, transitions, transition-labels (selects, guards, synchronizations and updates) and even verification queries by extracting the required values associated with the current element. An excerpt containing a piece of the code responsible for the code generation of parts of the UPPAAL model associated with sensors can be seen in Appendix L.

Testing the output of this process is as simple as putting the DSL example in Appendix E into the project and invoking the generator. The output can then be checked against expectations and previously hand-made models to ensure consistency. Additionally the output is checked by loading it into UPPAAL and inspecting the models, and running the queries to ensure validity of these too.

5.6 Modeling and Verification

During the design process of our system, we created an abstract model of it in order to see if our requirements would hold up during runtime. This has been achieved through validation of the designed model, and verification of certain properties. To model our system and to verify our requirements being satisfied at all times, we have used the UPPAAL integrated V&V model-checker tool.

As we mentioned before, as a secondary requirement, we set out a goal to be able to generate the UPPAAL model from the user's input in the DSL. With this in mind, during the modeling process we had to create rather simple, uniformed templates in UPPAAL that can be modified and regenerated based on the user's input.

To depict the system's logic, we have created a general main template, that is responsible for handling the possible events in the system and communicates with other physical parts, such as sensors and actuators, modeled in separate templates, via synchronization channels. The general template has generatable sub-sections, each of them handling one specific sensor defined by the user. One such sub-section can be activated by the system timer to request a sensory reading of the environment at user defined intervals. The model checks for the validity of the reading, in the case of 5 independent invalid readings, we assume a hardware problem with the sensor and alert the user. Valid sensory readings however are checked again if they trigger any associated actuators, and if so, the actuators are turned on for a user defined amount of time. All 3 possible outcome of each sensory reading eventually ends with a transition leading back the SystemOn idle location, so that we can continue the simulation by either checking another sensor, or shutting down an actively working actuator. This behavior is designed to satisfy our requirement that only one sensor should be checked at one given time, but still leaves us the possibility of having multiple actuators running in the background. An example of a sensory sub-section of the Generic template can be in Appendix M.

5.6.1 *Environmental modeling*.

During the modeling of templates of actuators and sensors, we have encountered an issue with the need to simulate the environment of our system to react upon. The ability to describe a complex, ever-changing system, such as nature itself is already a challenging feat in itself, but to pursue making the predictions as accurate as possible, it becomes a topic worthy of its own research. To be able to simulate the working of our model, we agreed we have to make certain assumptions to abstract the environment. During this process, we have identified two possible viable approaches. The first model has sensors generating a random number of a defined range every time a sensory reading is requested. Actuator templates have 4 possible locations, an idle, a working, startWorking and stopWorking, the latter two being auxiliary committed locations for technical reasons. The actuators are shut down by the main model if they have been working for *at least* the requested amount time.

The second model utilizes ODEs to simulate the constant changing nature of the environment to react upon. The actuator templates have only 2 possible locations, idle and working. The environment/sensor templates use the same broadcast channels as the actuators to change the flow direction of the ODE. This model uses its own "tick" integer type value instead of a clock.

We have found that both solutions can provide additional usefulness and positive functionality to the model, but both have their own shortcomings as well. The model utilizing random number generation comes in handy, when during a simulation we want to see how the model handles specific scenarios. However, it makes the simulation overall fairly unrealistic, as the working of actuators are not having an effect on the environment, as the sensory values are generated randomly for each sensory reading request. An example of a sensor template producing random numbers as sensory readings, and an example of the sensor/environmental template using an ODE can be seen in Appendices N and O, respectively.

The model using ODEs is effectively a contrast to the previous solution, providing a more realistic environment simulation, on which the actuators can have effect. The downside of this solution is that it uses fairly complex ODEs that the user should provide in DSL input, as we can not know in advance which type of environmental sensor they are going to declare. This could require some prior knowledge of environment simulation from the users as a prerequisite, which is not necessarily good practice. The other problem with this method that we failed to resolve due to time constraints, revolves around the usage of "ticks" instead of clock type variables in UPPAAL, which we mentioned above. As the whole environment simulation relies on the environment timer, which loops around to increase the ticks (essentially, the time in our system), we had to make the locations in our model committed. In committed locations the simulation is continued instantaneously, without any time delay spent in the location. This is obviously not case in the real system, as the locations symbolize the different processes, but as most of the code has a fairly short runtime, and we found the value of environment simulation beneficial to our model abstraction, we decided to include it in our report.

In both cases, we have been only able to guarantee that every activated actuator stays active for *at least* the required amount of time, as guaranteeing exact stopping points in time was found to be hard to achieve, especially with the hardware of the real system. When we became aware of this behavioral characteristics of the model, we agreed it is an acceptable feature, that still enables us to simulate the model according to our scope in a way that accurately displays the working logic.

5.6.2 *Verification*.

To verify certain properties of the model, we have translated most of our requirements declared earlier to formal expressions that

can be verified in UPPAAL. In Appendix V, we can see a table of requirements, with the same categorization as earlier, and their formal translation, implemented in UPPAAL. Note that during the code generation of the UPPAAL model, we are able to generate verification properties to check as well. As a result, we made all of our tests shown in the table pass, with the exception of deadlock checking, as due to the continuous nature of the simulation, there will be always a next transition to take, making the state space infinite, and the verification of this property impossible.

We interpreted these results as proof that the system's major components worked on paper, and that it satisfies our requirements. During the modeling we had to omit certain additional requirements, that we haven't been able to incorporate in the model, mostly due to time constraints and relevance to our goals. The real system has the ability to post sensory readings, this was left out from the model due to irrelevancy to our modeling goals. Another requirement for the real system is user's ability at all times to override the system, this was initially included in the model (using the Notify template), but only added towards the complexity of the state space, without adding utility to our model, so we removed it.

We believe that there are more left to improve upon with our model, mainly the introduction of additional timing constraints to processes other than the actuators. We have found, after experimentation with our above mentioned two different approaches, that a potential solution to combine the strengths of both solutions would be the creation of a scheduling system in UPPAAL. However, due to time constraints, we decided to leave it out for a potential individual extension of our project.

5.7 Results

The prototype of the system, while not tested in the intended context (i.e. maintaining a plants environmental parameters), has been tested separately and functions without the context — and it is therefore why we call it a pretotype. Rigorous design efforts and modeling gives confidence in the ability of the system to perform its intended purpose in the problem context. The Tables 4 and 5 in Appendix T & U respectively sums up the implemented functionality while comparing it to the previously listed requirements in Appendices B and C. From these it becomes apparent that most functionality and non-functionality is as intended, with some small deviations from expectations like Functional Requirement (FR) 1, 2, 5, 8 and 9. FR1 fails partially only because of the lack of testing with a CO2 sensor, but it could easily be tested and would likely work fine if equipment was available. FR5 and FR8 are almost identical in meaning but the first is not fulfilled because the ESP32 does not respond to other stimuli but sensor input, while the other fails because the user in the current iteration can not invoke / override behaviour remotely. The latter of the two is very close to being implemented, but is not there yet. FR9 was not intended to be implemented in this iteration, and as such has not been worked on. This function was scoped out of the first iterations very early in the development. The rest of the system functions as intended and an example of the system components working together can be seen in the video at:

Mirror #1: https://www.youtube.com/watch?v=9GMhw8HLA-4
Mirror #2: https://ln5.sync.com/dl/a48fe8360/d6ux6kjs-cueycs7r-6esj9end-gvj4c8kk
In the video from the start to 01:01 the code-gen of the board-code is run through, after which code-gen for the models are demonstrated until 02:03 in the video. Then follows a short bit of run-down of the hardware until 03:11, and finally a demonstration of the system working until the end of the video.

In regards to the non-functional requirements (NFR) a lot of the desired properties are fulfilled to satisfaction. The notable exceptions are NFR 1, 5 and 11. NFR 1 was never intended to be fulfilled but would be a nice addition to allow further self-regulation in the system. NFR 5 is by nature very hard to ensure, but to attempt achieving this property a model was developed and constantly compared to implementation to help ensure consistency and translatability of qualities from model to system. NFR 11 is fulfilled in the tested system as the longest sensor reading takes 23 ms [12] to read and that allow 977 ms to react, which by design happens immediately after reading of the sensor value. The fulfillment of NFR 2 - 4 are summed up in 1, and NFR 6 - 10 are fulfilled by building the system from the ground-up to uphold these principles.

5.7.1 ***Future work.***

Some of the aspects of our DSL are not utilized or implemented in our physical system, due to time limitation or scope reduction. Therefore, including these will not only increase the usability of our product, but also the overall quality, as some of the missing elements are needed to customize the system further. An example of one of the missing features that we did not mange to implement, is the ability to control actuators on a more granular level, by specifying voltage levels or the intensity a specific actuator should operate at, in form of a specific PWM duty-cycle, as seen in Appendix E. Since our DSL grammar supports this configuration, to realize this option for the user, the specific logic for the board-code should be implemented together with the respective section of the code generator.

Besides the example above, another feature that could further elevate the quality of our system is the ability to link multiple "units" together. This could allow the user to create a coupling between units that share some common properties and have similar growing conditions, to make monitoring and potentially optimizing how much yield a cluster of crops can produce. This scalability aspect was specified earlier as requirement FR 9, but has not been fulfilled.

Supporting more sensors and actuators in our system, would allow the user to monitor and control more aspects of the crops environment and potentially provide better growing conditions for the plants. Examples of actuators that we see likely to be needed support for, are air filtration systems, lighting systems in form of special light bulbs with specific wavelengths or linear actuators to open/close curtains to control the amount of sunlight getting in. This additional hardware support would entail integrating the sensors and actuators into our DSL and writing custom sampling/-controlling libraries that are supported by our microcontroller code.

Our pretotype is meant to be plugged in at all times for testing purposes, but the final system may need to be deployed on limited power supply such as batteries. Power consumption is therefore an important aspect of it, and as such the computation should only be done when necessary. To improve this aspect, we could use the event system as it provides knowledge of the timeframes where putting the device in deep-sleep mode is possible, and therefore greatly lower the power consumption by avoiding constant looping of the program.

In order to have a better overview of the system, increase predictability and troubleshoot problems more easily, it is also necessary to be able to send notifications or error notices to the user. While this requirement (NFR1) is already partially fulfilled through the serial output, the final version of the device should allow the user to receive such messages remotely, through a notification or error MQTT topic for example.

6 CONCLUSION

The project sets out to produce a pretotype proving the viability of having a customisable platform with exchangeable peripheral units to maintain optimal growth parameters. Most aspects, as seen in Appendices T and U, have been fulfilled as intended and the project could, with the correct configuration and peripherals, maintain environmental factors. The pretotype is capable of reading from a range of sensors, push the data to a broker and control actuators based on a configuration in a custom DSL. The application of proper knowledge about optimal parameters could allow the project to completely automate the growth of crops or other vegetation of interest. It has however become clear that before adoption of the system a lot more work has to be put into refining the DSL and software components. Additionally, if the verified guarantees are to be useful for anything but liveness or reachability checks, real-world values of the systems making up the environment and/or microprocessor need to be collected, analysed and fed into the model. Alternatively conservative fuzzy logic or machine learning could be utilised to allow for more flexible setups, this could potentially simplify configuration too.

REFERENCES

[1] Natalia Zinets and Aleksandar Vasovic. "Missiles rain down around Ukraine". In: *Reuters* (Feb. 25, 2022). URL: https://www.reuters.com/world/europe/putin-orders-military-operations-ukraine-demands-kyiv-forces-surrender-2022-02-24/ (visited on 05/12/2022).

[2] Michael Hogan and Gus Trompiz. "Grain exporters tap EU supplies as war shuts Ukraine ports -traders". In: *Reuters* (Feb. 25, 2022). URL: https://www.reuters.com/business/energy/grain-exporters-tap-eu-supplies-war-shuts-ukraine-ports-traders-2022-02-25/ (visited on 05/12/2022).

[3] Mohammad Hooshmand et al. "The effect of deficit irrigation on yield and yield components of greenhouse tomato (Solanum lycopersicum) in hydroponic culture in Ahvaz region, Iran". In: *Scientia Horticulturae* 254 (Aug. 25, 2019), pp. 84–90. ISSN: 0304-4238. DOI: 10.1016/j.scienta.2019.04.084. URL: https://www.sciencedirect.com/science/article/pii/S0304423819303383 (visited on 05/12/2022).

[4] Remus Prăvălie et al. "Spatio-temporal trends of mean air temperature during 1961–2009 and impacts on crop (maize) yields in the most important agricultural region of Romania". In: *Stochastic Environmental Research and Risk Assessment* 31.8 (Oct. 2017). Num Pages: 1923-1939 Place: Heidelberg, Netherlands Publisher: Springer Nature B.V., pp. 1923–1939. ISSN: 14363240. DOI: https://doi.org/10.1007/s00477-016-1278-7. URL: https://www.proquest.com/docview/1951917324/abstract/F0FF118CC17A4742PQ/1 (visited on 05/12/2022).

[5] *InfluxDb*. InfluxData Inc. URL: https://www.influxdata.com/ (visited on 05/13/2022).

[6] *Grafana: The open observability platform*. Grafana Labs. URL: https://grafana.com/ (visited on 05/13/2022).

[7] knolleary. *PubSubClient library website%*. Section: Microcontroller. May 20, 2020. URL: https://pubsubclient.knolleary.net/ (visited on 05/15/2022).

[8] cdaviddav. *Guide to Reduce the ESP32 Power Consumption by 95%*. Section: Microcontroller. Sept. 23, 2020. URL: https://diyi0t.com/reduce-the-esp32-power-consumption/ (visited on 05/15/2022).

[9] Eclipse.org. *Xtend - Documentation*. URL: https://www.eclipse.org/xtend/documentation/index.html (visited on 05/13/2022).

[10] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Second Edition. Livery Place 35 Livery Street Birmingham B3 2PB, UK.: Packt Publishing Ltd. ISBN: 978-1-78646-496-5.

[11] *Xtext - The Grammar Language*. URL: https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html (visited on 05/10/2022).

[12] "DHT11 Humidity & Temperature Sensor". In: (), p. 10. URL: https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf.

APPENDIX

A Glossary

Term	Explanation
Back-off	Also called cooldown. Refers to a required amount of time spent inactive before subsequent actuator activity.
FR	Functional requirement .
NFR	Non-functional requirement .
PWM	Pulse-width Modulation
V&V	Validation and Verification

Table 1: Glossary of Terms

B Functional Requirements

Number	Requirement
1	The system should be able to sense moisture, humidity, temperature, Co2 and light.
2	Users are notified in case of measurement errors.
3	The system should be able to provide water for the plant.
4	The system should use collected sensor data to determine whether an actuator needs to be activated.
5	The system should be accessible over a network.
6	The system should publish sensor readings to a MQTT broker.
7	Sensor measurement intervals should be definable for the user.
8	Actuators can be individually controlled/invoked by a user, temporarily overriding defined system behaviour.
9	The system should be scalable combining the functionality of multiple similar units.
10	Actuators should not run within their set cooldown/back off time.
11	The user should be able to debug their configuration through the serial output.
12	The user should be able to see sensor readings on a webpage.
13	The systems should be configurable using a Domain Specific language.
14	UPPAAL files modeling the system based on the user's input should be automatically generated from the DSL.
15	Board controller code for the system should be automatically generated from the DSL.

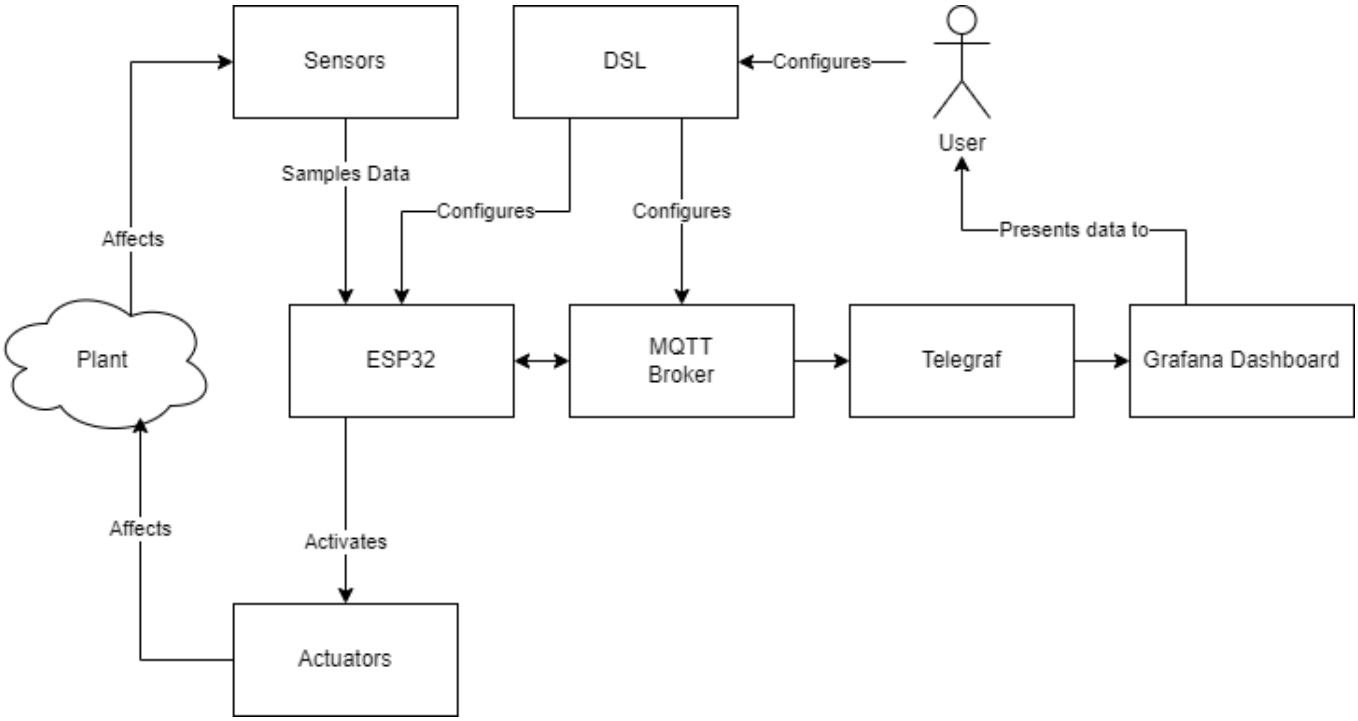
Table 2: Functional Requirements

C Non-Functional Requirements

Number	Requirement
1	Actuators should report any error obstructing them from running. Eg. A pumps water reservoir being empty.
2	The system must make use of Internet of Things concepts.
3	The system must make use of Software Systems Analysis and Verification concepts.
4	The system must make use of Model-driven Software Development concepts.
5	There should not be deadlocks in the system.
6	The system must continue functioning despite non-fatal errors.
7	The system must continue functioning despite network outages.
8	The systems behaviour and interaction with peripheral devices (sensors, actuators, messaging system) must be user configurable
9	Only one sensor should be interacted with at a time.
10	An activated actuator should not block execution of a different part in the remaining system.
11	The system should read and react upon sensor data within 1 second.

Table 3: Non-Functional Requirements

D Architecture Diagram



E DSL Configuration Example

```
Unit:

var host = https://localhost               //(default localhost)

Sensor: Temperature Temperatur
Sensor: Light Lys_Sensor
Sensor: Moisture Fugt

Actuator: Pump Pumpe
Actuator: Valve Ventil
Actuator: Light Lys
Actuator: Piezo Alarm

Trigger: Pumpe for 15s when Fugt < 10 back-off 5min
Trigger: Pumpe for 15s @ 50% when Fugt < 20 back-off 3min
Trigger: Lys when time 06:00 - 18:00
Trigger: Lys when time 18:00 - 21:00 @ 50%

PinIn: Temperatur Digital 7
PinIn: Lys Sensor Analog 10 v->((v + 13) / 5)
PinIn: Fugt Analog 9 v->(3+2) @ 10Hz

PinOut: Pumpe 16 PWM(1% - 2% @ 10kHz)
PinOut: Ventil 17 Voltage(0V - 5V)
PinOut: Lys 18 Voltage(0V / 5V)
PinOut: Alarm 12 PWM(0% / 50% @ 1 kHz)

Receiver: Water_Rec @ 0.1Hz
```

```
Receiver: Light_Rec @ 0.1Hz

Trigger: Pumpe for 30s when Water_Rec
Trigger: Lys for 5min when Light_Rec
Trigger: Alarm for 30s when Fugt < 10 back-off 1min

Sender: host/moisture_data Fugt Running_Avg(10) @ 1Hz
Notify: host/moisture_notify when Fugt < 20 back-off 5min saying 'I'm dying over here'
```

F DSL Grammar Excerpt

```
Unit: {Unit} "Unit:" (attr += Attribute)*;
Attribute: Sensor | Actuator | PinIn | PinOut | Trigger | Receiver | Sender | Notify;

// -----Custom Terminals----- //
terminal TIMESTAMP: ((('0'|'1')?('0'..'9'))|(('2')('0'..'3')))':'('0'..'5')('0'..'9');
NUMBER returns ecore::EFloat: INT ('.' INT)?;
URL: (ID|INT) (('.'|'/'|':')?(ID|INT))*;
ValueUnit: unit=(ID|'%');

PinIn: "PinIn:" ref=[Sensor] type=("Digital" | "Analog") pin=INT (conv=Conversion)?
    (rate=Rate)?;
// Inspired by MDSD 2 hand-in solution
Conversion: name=ID "->" exp=Expression;
Expression returns Expression: Factor (( {Plus.left=current} '+' | {Minus.left=current} '-'
    ) right=Factor)*;
Factor returns Expression: Primary (({Mult.left=current} '*' | {Div.left=current} '/' )
    right=Primary)*;
Primary returns Expression: {MathNumber} value=NUMBER | Parenthesis | VariableUse;
Parenthesis returns Expression: {Parenthesis} '(' exp=Expression ')';
VariableUse returns Expression: {VariableUse} ref=[Conversion];

Trigger: "Trigger:" ref=[Actuator] ("for" trigger_time=INT time_unit=ValueUnit
    (intensity=Intensity)?)? cond=Condition;
Condition: "when" trigger=ConditionTrigger (compare_op=("<" | ">" | "<=" | ">=")
    compare_val=INT "back-off" backoff_time=INT time_unit=ValueUnit)? (intensity=Intensity)?;
ConditionTrigger: ref=[Input] | time=TimeRange;
// Validation and retrieving of ref is done manually (Attempting to use a reference results
    in RULE_ID collisions for the lexer)
Input: Receiver | Sensor;
```

G DSL Code Generator Excerpt

```
def compileFile(Unit unit)'''
//-------------------------------
//---------- Libraries ----------
//-------------------------------
<<unit.add_Libraries>>

//-------------------------------
//--------- WIFI Config ---------
//-------------------------------
<<unit.add_Wifi_Config>>
```

```
//-------------------------------
//--------- MQTT Config ---------
//-------------------------------
<<unit.add_Mqtt_Config>>

//-------------------------------
//------ Sensors Config ---------
/ /-------------------------------
<<unit.add_Sensor_Config>>

//-------------------------------
//------ Actuators Config -------
//-------------------------------
<<unit.add_Actuator_Config>>

//-------------------------------
//------- Trigger Config --------
//-------------------------------
<<unit.add_Trigger_Config>>

//-------------------------------
//---------- Setup --------------
//-------------------------------
<<unit.add_Setup>>

//-------------------------------
//-------- Main Loop  -----------
//-------------------------------
<<unit.add_Loop>>

//-------------------------------
//------ MQTT Callback  ---------
//-------------------------------
<<unit.add_MqttCallback>>
'''
```

H Map population Excerpt

```
........
                timeMap.put("s", 1000)
                timeMap.put("m", 60000)
                timeMap.put("hrs", 3600000)
        }

        def populateLibraryMap() {
                libMap.put("CO2", "<potentiometers/linear_10kohm.h>");
                libMap.put("OutLight", "<light_sensors/alspt19.h>");
                libMap.put("InLight", "<light_sensors/dfr0026.h>");
                libMap.put("Temperature", "<temperature_sensors/dht11Temp.h>");
                libMap.put("Humidity", "<humidity_sensors/dht11Hum.h>");
                libMap.put("Pump", "<pumps/pump.h>");
```

```
        libMap.put("Moisture", "<soil_moisture_sensors/parallax.h>");
    }

    def populateSamplingMap() {
        samplingMap.put("CO2", "sampleLinear_10kohm");
        samplingMap.put("OutLight", " sampleAlspt19");
        samplingMap.put("Moisture", "sampleParallax");
        samplingMap.put("Temperature", "sampleDht11Temp");
        samplingMap.put("Humidity", "sampleDht11Hum");
        samplingMap.put("InLight", "sampleDfr0026");
    }

    def populateMap() {
        for (item : this.unit.attr) {
            switch item {
                PinIn: {
                    pinMap.put((item.ref as Sensor).name.toString,
                        item.pin)

                    var conv = item.conv !== null ? item.conv : null;
                    convMap.put((item.ref as Sensor).name.toString, conv)

                    var rate = item.rate !== null ? item.rate : null;
                    rateMap.put((item.ref as Sensor).name.toString, rate)
                }
                PinOut: {
                    pinMap.put((item.ref as Actuator).name.toString,
                        item.pin)
                    sigMap.put((item.ref as Actuator).name.toString,
                        item.signal_out)
                }
                Sender: {
                    pubMap.put((item.sensor_ref as
                        Sensor).name.toString, item)
                }
                Trigger: {
                    triggerMap.put((item.ref as Actuator).name.toString,
                        item)
                }
            }
        }
    }
    ......
```

I DSL Validation Excerpt

```
public static val INVALID_REF = 'invalidRef'
public static val DUPLICATE_NAME = 'duplicateName'
public static val INTENSITY_VALPERCENT = 'invalidIntensityValue'
public static val RANGE_SAME_UNIT = 'sameUnitRange'
public static val HIGHLOW_SAME_UNIT = 'sameUnitHighLow'
public static val RANGE_PERCENT_RANGE = 'invalidPercentRange'
```

```
public static val HIGHLOW_PERCENT_RANGE = 'invalidPercentHighLow'

def EObject getContainerOfType(EObject object, (EObject)=>boolean predicate) {
        val container = object.eContainer
        if( predicate.apply(container) )
                return container
        return getContainerOfType(container, predicate)
}

/* // Found a fix for this to be unnecessary
@Check
def checkConditionTriggerRefIsValid(ConditionTrigger ct) {
        if(ct.ref !== null) {
                val (EObject)=>boolean predicate = [ EObject eCon | eCon instanceof Unit ]
                val Unit unit = (getContainerOfType(ct, predicate) as Unit)

            val Attribute attr = unit.attr.findFirst[ a | (((a instanceof Sensor) && (a as
                Sensor).name == ct.ref) || ((a instanceof Receiver) && (a as Receiver).name
                == ct.ref))]
                if(attr === null) {
                        error('Reference must refer to a Sensor or a Receiver',
                            DSLPackage.Literals.CONDITION_TRIGGER__REF, INVALID_REF)
                }
        }
}
*/

@Check
def noRepeatNames(Attribute attr) {
        val (EObject)=>boolean predicate = [ EObject eCon | eCon instanceof Unit ]
        val Unit unit = (getContainerOfType(attr, predicate) as Unit)

        val name = (attr instanceof Sensor) ? (attr as Sensor).name : ((attr instanceof
            Actuator) ? (attr as Actuator).name : ((attr instanceof Receiver) ? (attr as
            Receiver).name : null))
        if(unit.attr.findFirst[ a | (a !== attr && name !== null &&
                (((a instanceof Sensor) && (a as Sensor).name == name)
                        || ((a instanceof Receiver) && (a as Receiver).name == name)
                        || ((a instanceof Actuator) && (a as Actuator).name == name))
        )] !== null) {
                if(attr instanceof Sensor)
                        error('Name cannot be the same as any other name',
                            DSLPackage.Literals.SENSOR__NAME, DUPLICATE_NAME)
                else if(attr instanceof Actuator)
                        error('Name cannot be the same as any other name',
                            DSLPackage.Literals.ACTUATOR__NAME, DUPLICATE_NAME)
                else if(attr instanceof Receiver)
                        error('Name cannot be the same as any other name',
                            DSLPackage.Literals.RECEIVER__NAME, DUPLICATE_NAME)
        }
}
```

```
@Check
def intensityInValidRange(Intensity intensity) {
        if(intensity.percent > 100 || intensity.percent < 0) {
                error('Intensity must be between 0 and 100 percent',
                    DSLPackage.Literals.INTENSITY__PERCENT, INTENSITY_VALPERCENT)
        }
}

@Check
def rangeSameUnit(Range range) {
        if(range.low_unit.unit != range.high_unit.unit) {
                error('Units must match across a range',
                    DSLPackage.Literals.RANGE__HIGH_UNIT, RANGE_SAME_UNIT)
                error('Units must match across a range',
                    DSLPackage.Literals.RANGE__LOW_UNIT, RANGE_SAME_UNIT)
        }
}
```

J UPPAAL XML Helper Excerpt

```
class UPPAAL {
        Map<String, String> uppaalLocNameToId = new HashMap()
        Integer uppaalLocIdCounter = 0

        var Document document;

        def initUPPAALXML(Unit unit) {
                uppaalLocNameToId = new HashMap()
                uppaalLocIdCounter = 0

                document =
                    ((DocumentBuilderFactory.newInstance()).newDocumentBuilder()).newDocument()

                val Element nta = document.createElement("nta")
                document.appendChild(nta)

                return nta
        }

        def convertToXML() {
                val Transformer transformer =
                    (TransformerFactory.newInstance()).newTransformer()
                val StringWriter sWriter = new StringWriter()

                transformer.transform(new DOMSource(document), new StreamResult(sWriter))
                val String XML = sWriter.buffer.toString()
                return XML
        }

        def AddNamedNode(Element root, String name, String type) {
                val Element node = document.createElement(type)
                root.appendChild(node)
```

```
        val Element eName = document.createElement("name")
        eName.appendChild(document.createTextNode(name))
        node.appendChild(eName)

        return node
}

def getNodeAttr(Element el, String attrName) {
        for(var i = 0; i < el.childNodes.length; i++) {
                val child = el.childNodes.item(i)
                if(child.nodeName == attrName)
                        return child.firstChild//.nodeValue
        }
}

def makeLocMapName(Element template, String name) {
        return '''<<getNodeAttr(template, "name").nodeValue>>/<<name>>'''
}

def AddTemplate(Element nta, String name) {
        return AddNamedNode(nta, name, "template")
}

def AddLocation(Element template, String name) {
        val Element node = AddNamedNode(template, name, "location")

        val id = '''id<<uppaalLocIdCounter++>>''' // Increment counter
        node.setAttribute("id", id)
        uppaalLocNameToId.put(makeLocMapName(template, name), id)

        return node
}
```

K UPPAAL Generator Excerpt

```
override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
    IGeneratorContext context) {
         val unit = resource.allContents.filter(Unit).next
         //fsa.generateFile(foldername+"/" +filename+ ".cpp",compileFile(unit))
         uppaal = new UPPAAL()

         uppaal.initUPPAALXML(unit)
         val Element nta = uppaal.initUPPAALXML(unit)

         fsa.generateFile('''<<folderName>>/<<fileName>>.xml''',
             compileUPPAALTemplates(unit, nta, true))
}

def addSimulatedSensor(Sensor sensor, Element nta) {
        val name = '''<<sensor.name>>Sensor'''
        val sensorTemplate = uppaal.AddTemplate(nta, name)
```

```
        val sensorIdleLoc = uppaal.AddLocation(sensorTemplate, '''<<name>>Idle''')

        val sensorSampleLoc = uppaal.AddLocation(sensorTemplate,
            '''<<name>>Sample''')
        uppaal.SetInitLoc(sensorTemplate, sensorIdleLoc)

        val sampleTransition = uppaal.AddTransition(sensorTemplate, sensorIdleLoc,
            sensorSampleLoc)
        uppaal.AddSynchronisationToTransition(sampleTransition,
            '''sample<<sensor.name>>?''')

        val returnTransition = uppaal.AddTransition(sensorTemplate, sensorSampleLoc,
            sensorIdleLoc)
        uppaal.AddAssignmentToTransition(returnTransition, '''<<sensor.name>>Data =
            <<sensor.name>>Level''')
        uppaal.AddGuardToTransition(returnTransition, '''<<sensor.name>>Data ==
            -1000''')

        return sensorTemplate
}

def addSimulatedEnv(Sensor sensor, Element nta) {
        val name = '''<<sensor.name>>Env'''
        val envTemplate = uppaal.AddTemplate(nta, name)

        val actuatorOffLoc = uppaal.AddLocation(envTemplate, '''<<name>>ActOff''')

        val actuatorOnLoc = uppaal.AddLocation(envTemplate, '''<<name>>ActOn''')
        uppaal.SetInitLoc(envTemplate, actuatorOffLoc)

        val startTransition = uppaal.AddTransition(envTemplate, actuatorOffLoc,
            actuatorOnLoc)
        uppaal.AddSynchronisationToTransition(startTransition,
            '''Start<<sensor.name>>?''')

        val stopTransition = uppaal.AddTransition(envTemplate, actuatorOnLoc,
            actuatorOffLoc)
        uppaal.AddSynchronisationToTransition(stopTransition,
            '''Stop<<sensor.name>>?''')

        // TODO: Add envTick transitions

        return envTemplate
}

def addSimulatedEnvTimer(Element nta) {
        val envTimerTemplate = uppaal.AddTemplate(nta, "EnvTimer")

        val envTLoopLoc = uppaal.AddLocation(envTimerTemplate, "loop")
        uppaal.SetInitLoc(envTimerTemplate, envTLoopLoc)
        uppaal.AddInvariantToLocation(envTLoopLoc, "envTimer <= 1")
```

```
        val envTLoopTrans = uppaal.AddTransition(envTimerTemplate, envTLoopLoc,
            envTLoopLoc)
        uppaal.AddGuardToTransition(envTLoopTrans, "envTimer >= 1")
        uppaal.AddSynchronisationToTransition(envTLoopTrans, "doEnvTick!")
        uppaal.AddAssignmentToTransition(envTLoopTrans, "envTimer = 0")

        return envTimerTemplate
    }

    def compileUPPAALTemplates(Unit unit, Element nta, Boolean simulated) {
        for (e : unit.attr.filter(Sensor)) {
            if(simulated) {
                addSimulatedSensor(e, nta)
                addSimulatedEnv(e, nta)
            }
            else {
                addRandomSensor(e, nta)
            }
        }

        if(simulated)
            addSimulatedEnvTimer(nta)

        for (e : unit.attr.filter(Actuator)) {
            addActuator(e, nta)
        }

        return uppaal.convertToXML()
    }
```

L UPPAAL Code Generation Excerpt

```
def compileUPPAALModel(Unit unit, NTA nta){

    eventList = new HashMap();
    val main = uppaal.AddTemplate(nta, "General")
    uppaal.AddToDeclaration(nta, "clock SysTimer;")
    val mainIdle = uppaal.AddLocation(main, "SystemOn")


    for (a : unit.attr){
        elementName = ""
        sensName = ""
        actName = ""
        switch(a) {
            Sensor case a:
                {
                elementName = a.name
                sensorList.add(elementName)

                //Declarations
                if (a.range !== null){
```

```
        val Low = a.range.low
        val High = a.range.high
        uppaal.AddToDeclaration(nta, "typedef int[" + (Low -
            10).toString() + "," + (High + 10).toString() +
            "] " + elementName + "Rand;")
        uppaal.AddToDeclaration(nta, "int " + elementName +
            "MaxValid = " + High + ";")
        uppaal.AddToDeclaration(nta, "int " + elementName +
            "MinValid = " + Low + ";")
        }
uppaal.AddToDeclaration(nta, "int " + elementName + "Data =
    0;")
        uppaal.AddToDeclaration(nta, "chan Send" +
            elementName + ";")
uppaal.AddToDeclaration(nta, "clock " + elementName +
    "SensTimer;")
uppaal.AddToDeclaration(nta, "chan Sample" + elementName +
    ";")
uppaal.AddToDeclaration(nta, "int " + elementName +
    "MeasureError = 0;")
uppaal.AddToDeclaration(nta, "clock " + elementName + "T;")

//Sensor template
val sensorTemp = uppaal.AddTemplate(nta, elementName +
    "Sensor")
val sensorIdle = uppaal.AddLocation(sensorTemp, elementName
    + "SensIdle")
val sensorSample = uppaal.AddLocation(sensorTemp, "Sample" +
    elementName)
val SensorTrans = uppaal.AddTransition(sensorTemp,
    sensorSample, sensorIdle)
uppaal.AddSynchronisationToTransition(SensorTrans, "Send" +
    elementName + "?")
val SensorTrans2 = uppaal.AddTransition(sensorTemp,
    sensorIdle, sensorSample)
uppaal.AddSelectToTransition(SensorTrans2,
    elementName.charAt(0) + " : " + elementName + "Rand")
uppaal.AddAssignmentToTransition(SensorTrans2, elementName +
    "Data:= " + elementName.charAt(0))
uppaal.SetInitLoc(sensorTemp, sensorIdle)
uppaal.MakeLocationCommitted(sensorSample)

//Sensor timer template
val sensorTimer = uppaal.AddTemplate(nta, elementName +
    "Timer")
val sensorTick = uppaal.AddLocation(sensorTimer, elementName
    + "Tick")
val SensorTimerTrans = uppaal.AddTransition(sensorTimer,
    sensorTick, sensorTick)
uppaal.AddInvariantToLocation(sensorTick, elementName +
    "SensTimer <= " + elementName + "ReadInterval")
```

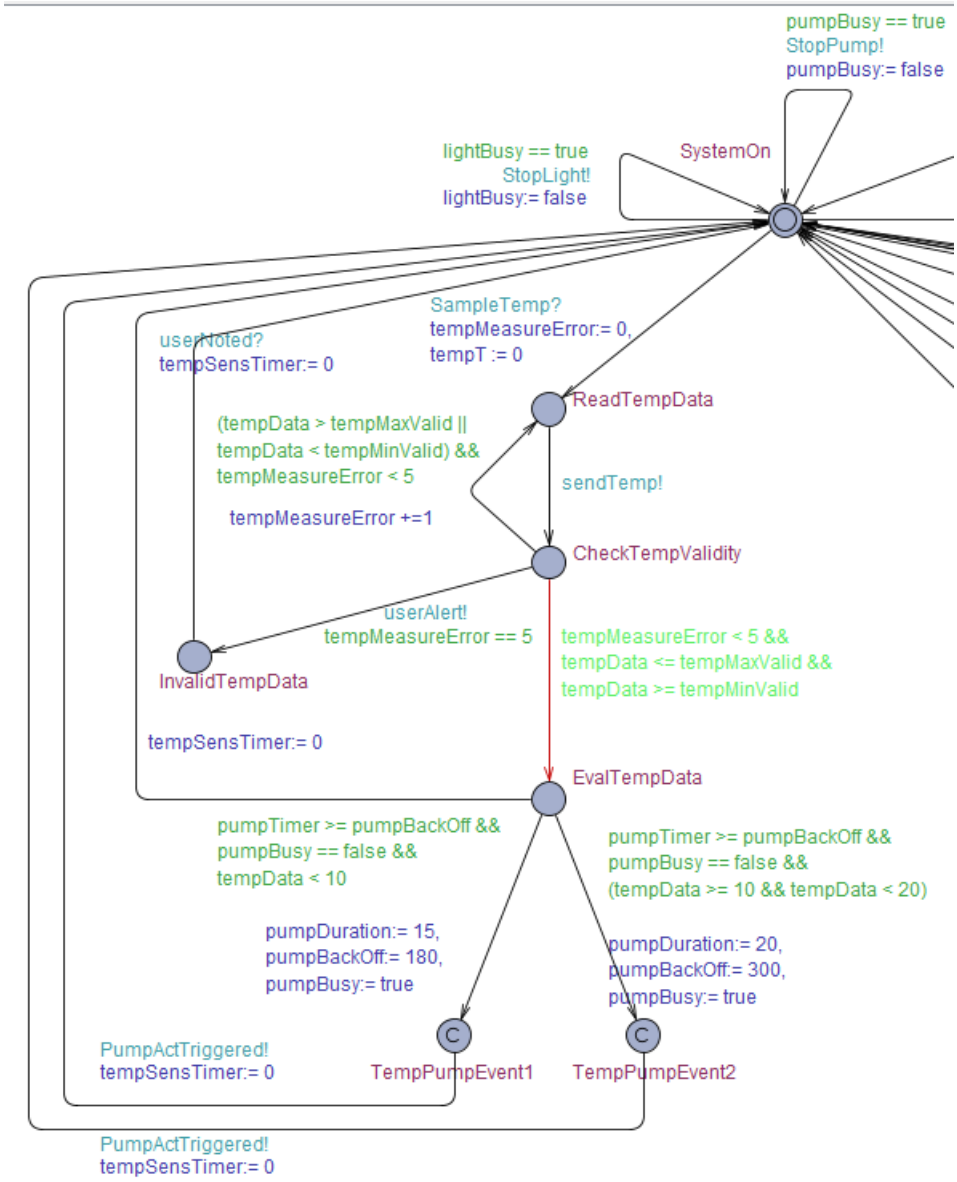
```
uppaal.AddSynchronisationToTransition(SensorTimerTrans,
    "Sample" + elementName + "!")
uppaal.AddGuardToTransition(SensorTimerTrans, elementName +
    "SensTimer >= " + elementName + "ReadInterval")
uppaal.SetInitLoc(sensorTimer, sensorTick)

//Adding sensory section to main template
val ReadData = uppaal.AddLocation(main, "Read" + elementName
    + "Data")
val CheckValid = uppaal.AddLocation(main, "Check" +
    elementName + "Validity")
val InvalidData = uppaal.AddLocation(main, "Invalid" +
    elementName + "Data")
val EvalData = uppaal.AddLocation(main, "Eval" + elementName
    + "Data")
val mainTrans = uppaal.AddTransition(main, ReadData,
    mainIdle)
uppaal.AddSynchronisationToTransition(mainTrans, "Sample" +
    elementName + "?")
uppaal.AddAssignmentToTransition(mainTrans, elementName +
    "MeasureError:= 0, " + elementName + "T:= 0")
val mainTrans2 = uppaal.AddTransition(main, CheckValid,
    ReadData)
uppaal.AddSynchronisationToTransition(mainTrans2, "Send" +
    elementName + "!")
val mainTrans3 = uppaal.AddTransition(main, ReadData,
    CheckValid)
uppaal.AddGuardToTransition(mainTrans3, "(" + elementName +
    "Data > " + elementName + "MaxValid || " + elementName +
    "Data" + " < " + elementName + "MinValid)" + " && " +
    elementName + "MeasureError" + " < 5")
uppaal.AddAssignmentToTransition(mainTrans3, elementName +
    "MeasureError += 1")
val mainTrans4 = uppaal.AddTransition(main, InvalidData,
    CheckValid)
uppaal.AddSynchronisationToTransition(mainTrans4,
    "userAlert!")
uppaal.AddGuardToTransition(mainTrans4, elementName +
    "MeasureError == 5")
val mainTrans5 = uppaal.AddTransition(main, mainIdle,
    InvalidData)
uppaal.AddSynchronisationToTransition(mainTrans5,
    "userAlerted?")
uppaal.AddAssignmentToTransition(mainTrans5, elementName +
    "SensTimer:= 0")
val mainTrans6 = uppaal.AddTransition(main, mainIdle,
    EvalData)
uppaal.AddAssignmentToTransition(mainTrans6, elementName +
    "SensTimer:= 0")
val mainTrans7 = uppaal.AddTransition(main, EvalData,
    CheckValid)
```

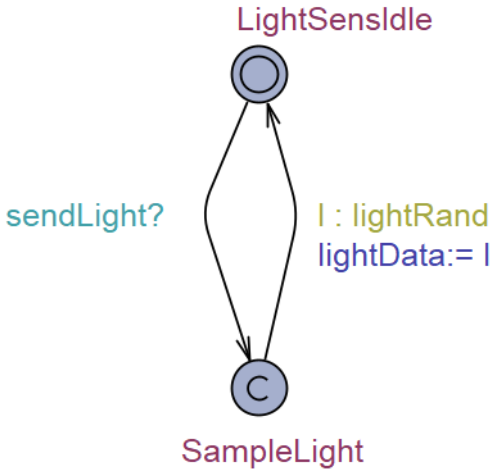
```
uppaal.AddGuardToTransition(mainTrans7, elementName +
    "MeasureError < 5 && " + elementName + "Data <= " +
    elementName + "MaxValid && " + elementName + "Data >= " +
    elementName + "MinValid")

//Queries
uppaal.AddQuery(nta, "E<> (p" + elementName + "Sensor." +
    elementName + "SensIdle imply " + "p" + elementName +
    "Sensor.Sample" + elementName + ")", "Eventually all
    sensors are able to be use")
uppaal.AddQuery(nta, "A<> (" + elementName + "MeasureError
    >= 5) imply (pGeneral.Invalid" + elementName + "Data)",
    "")
uppaal.AddQuery(nta, "E<> (" + elementName + "Data > " +
    elementName + "MaxValid or " + elementName + "Data < " +
    elementName + "MinValid) and " + elementName +
    "MeasureError <= 5 imply pGeneral.Read" + elementName +
    "Data", "")
}
```

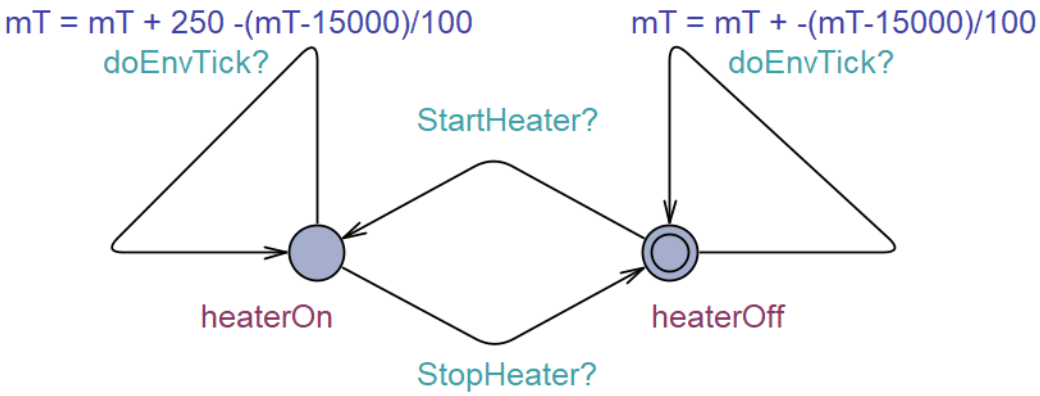
M UPPAAL Model General Template Sensor Section Example



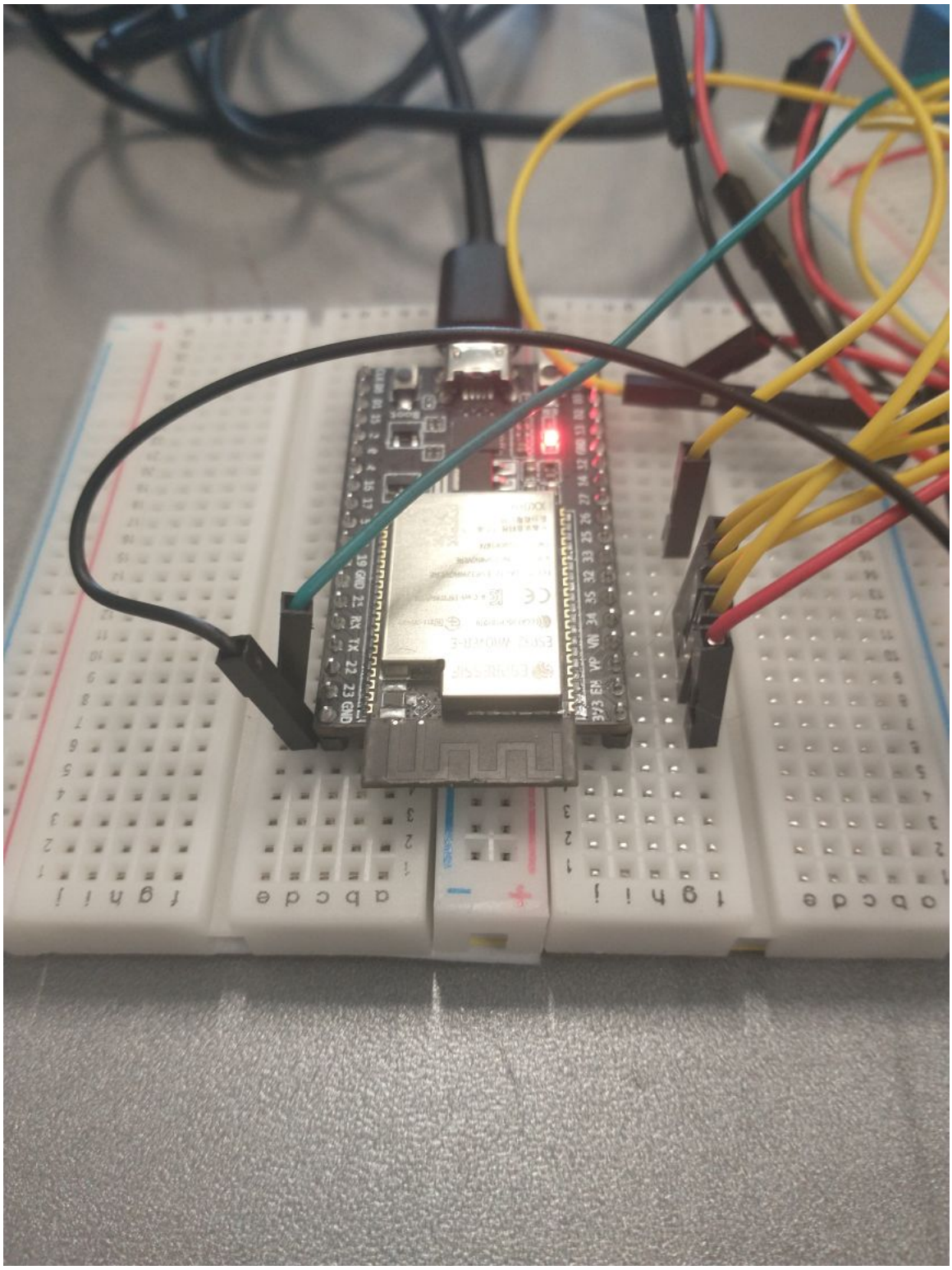
N UPPAAL Model Random Number Sensor Example



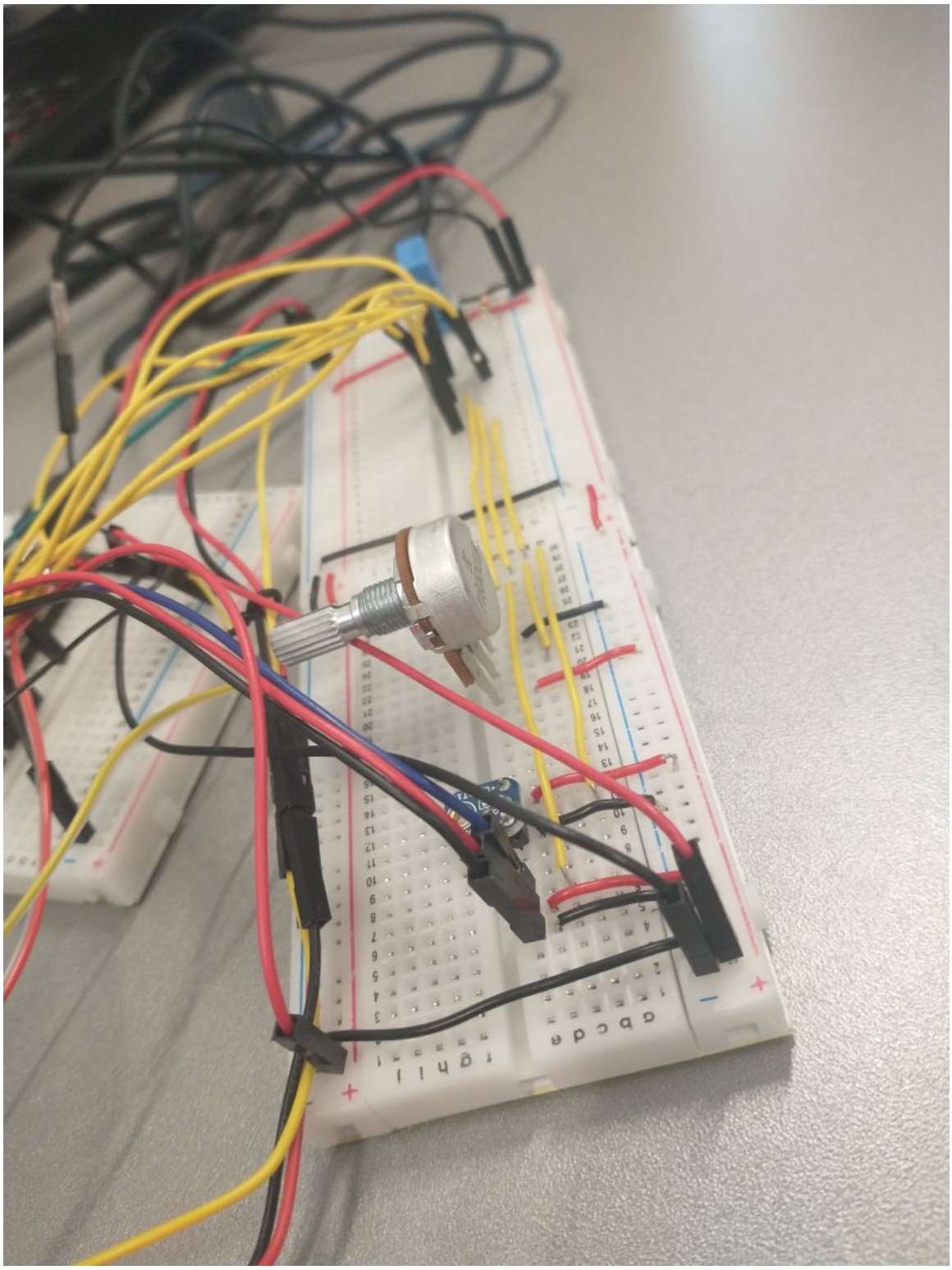
O UPPAAL Model ODE Sensor/Environment Example



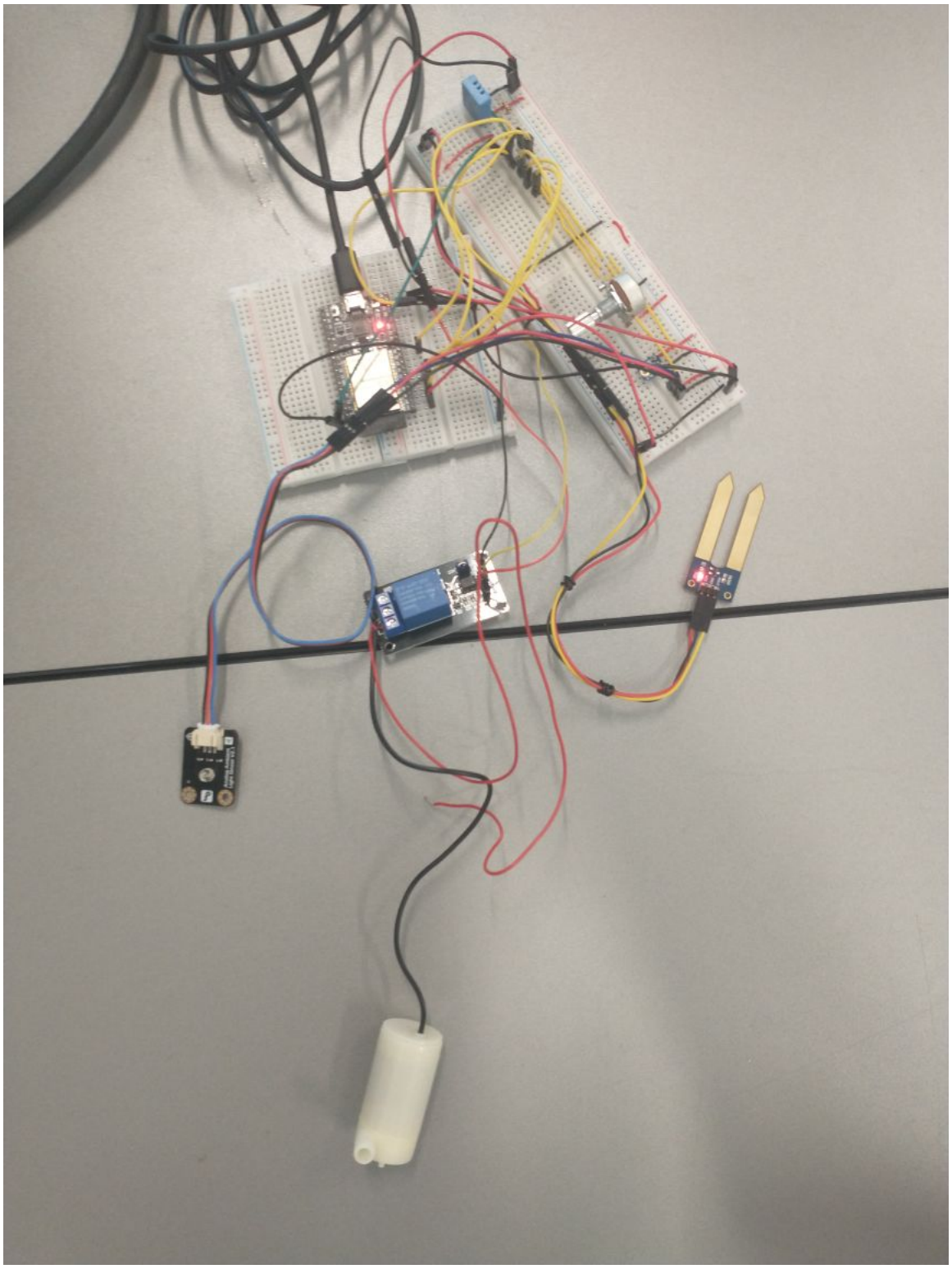
P Microcontroller



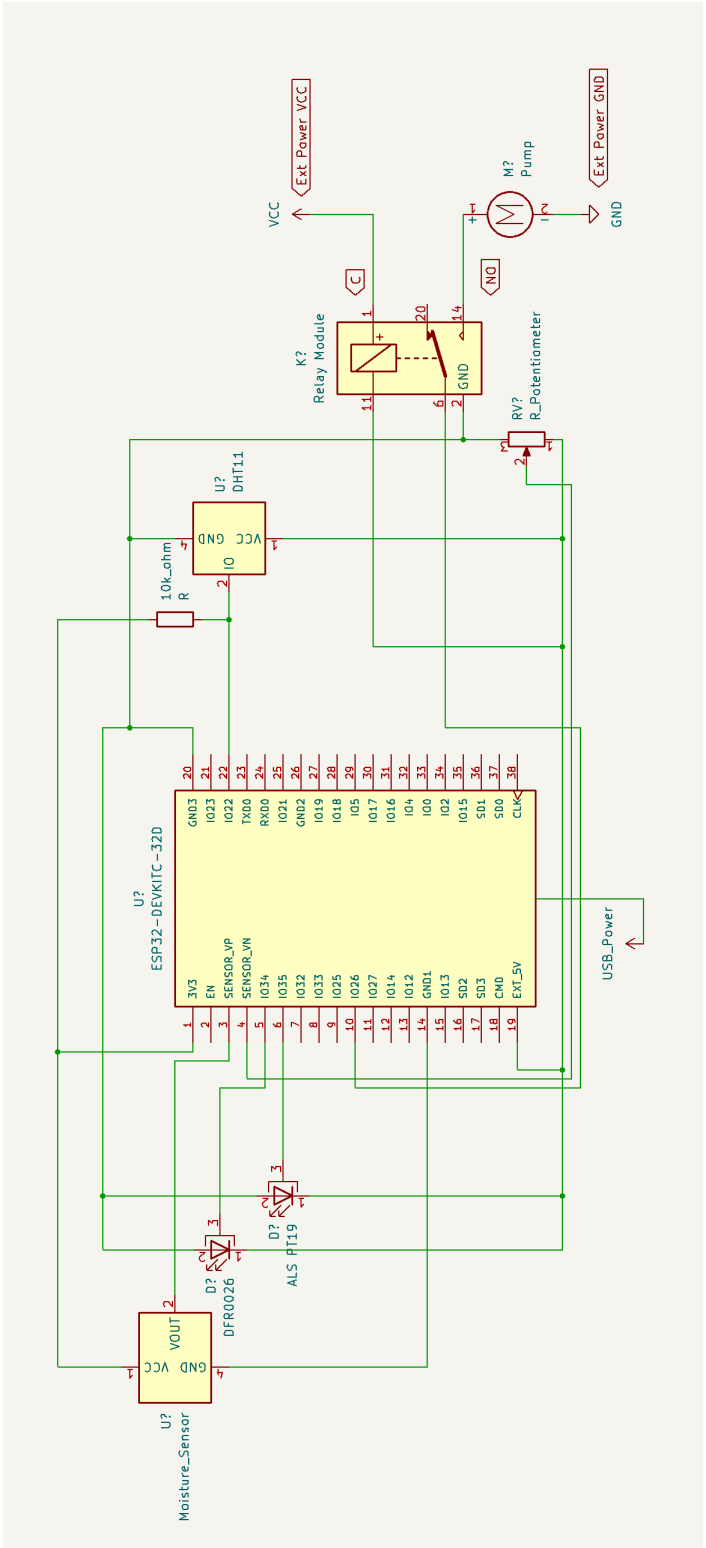
Q Sensors



R Setups



S KiCad Sketch of Complete Circuit



T Functional Requirement Results

Number	Requirement	Result
1	The system should be able to sense moisture, humidity, temperature, Co2 and light.	Partially fulfilled - Future work
2	Users are notified in case of measurement errors.	Not fulfilled - Future work
3	The system should be able to provide water for the plant.	Fulfilled
4	The system should use collected sensor data to determine whether an actuator needs to be activated.	Fulfilled
5	The system should be accessible over a network.	Not fulfilled - Future work
6	The system should publish sensor readings to a MQTT broker.	Fulfilled
7	Sensor measurement intervals should be definable for the user.	Fulfilled
8	Actuators can be individually controlled/invoked by a user, temporarily overriding defined system behaviour.	Not fulfilled - Future work
9	The system should be scalable combining the functionality of multiple similar units.	Not fulfilled - Not intended / Future work
10	Actuators should not run within their set cooldown/back off time.	Fulfilled
11	The user should be able to debug their configuration through the serial output.	Fulfilled
12	The user should be able to see sensor readings on a webpage.	Fulfilled
13	The systems should be configurable using a Domain Specific language.	Fulfilled
14	UPPAAL files modeling the system based on the user's input should be automatically generated from the DSL.	Fulfilled
15	Board controller code for the system should be automatically generated from the DSL.	Fulfilled

Table 4: Results of Functional Requirements

U Non-Functional Requirement Results

Number	Requirement	Result
1	Actuators should report any error obstructing them from running. Eg. A pumps water reservoir being empty.	Not fulfilled - Not intended
2	The system must make use of Internet of Things concepts.	Fulfilled
3	The system must make use of Software Systems Analysis and Verification concepts.	Fulfilled
4	The system must make use of Model-driven Software Development concepts.	Fulfilled
5	There should not be deadlocks in the system.	Partially fulfilled - Tested in model
6	The system must continue functioning despite non-fatal errors.	Fulfilled
7	The system must continue functioning despite network outages.	Fulfilled
8	The systems behaviour and interaction with peripheral devices (sensors, actuators, messaging system) must be user configurable	Fulfilled
9	Only one sensor should be interacted with at a time.	Fulfilled
10	An activated actuator should not block execution of a different part in the remaining system.	Fulfilled
11	The system should read and react upon sensor data within 1 second.	Fulfilled - Not fully tested

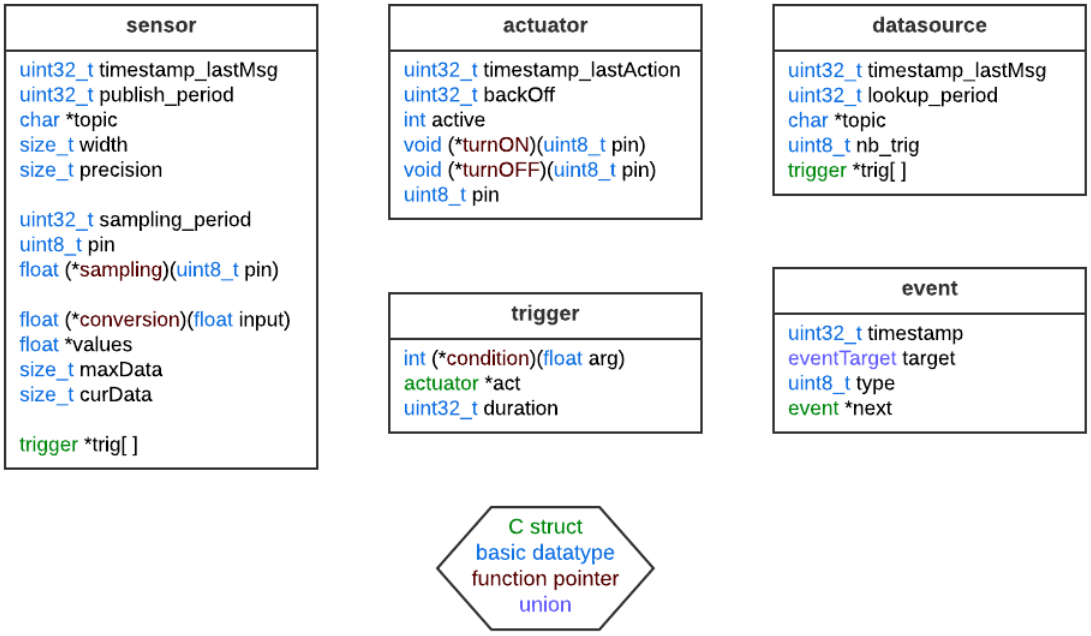
Table 5: Results of Non-Functional Requirements

V UPPAAL Verification Queries

ID	Requirement	UPPAAL query
Performance		
1	The system should read and react upon sensor data within 1 second.	— No query, not satisfied requirement —
Predictability		
2	There should not be deadlocks in the system (Not satisfied due to the continuous nature of simulation)	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 >= pumpBack-Off)
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	Sensor measurement intervals should be definable for the user.	— No query —
11	Actuators can be individually controlled/invoked by a user, temporarily overriding defined system behaviour.	— Removed functionality —
Error Detection		
12	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
13	Actuators should report any error obstructing them from running. Eg. A pumps water reservoir being empty.	— No query, not implemented functionality —

Table 6: Translation of Requirements into UPPAAL Queries

W Data structures of the controller



X Microcontroller sensor configuration Excerpt

```
// DSL-specified sensor-specific parameters defined as explicit preprocessor constants
#define DATA_BUFFER_SIZE_POTENTIOMETER_1     128
#define PUBLISH_PERIOD_POTENTIOMETER_1       100000
#define SAMPLING_PERIOD_POTENTIOMETER_1      10000
#define WIDTH_POTENTIOMETER_1                8
#define PRECISION_POTENTIOMETER_1            5
#define PIN_POTENTIOMETER_1                  14

// DSL-specified MQTT topic mean to receive the data related to this sensor
char mqtt_topic_potentiometer_1[] = "testtopic/potentiometer_1";

// DSL-specified conversion function
float conversionPotentiometer_1(float input)
{
  return input * 32 + 27;
}

// Sensor data structure creation helper function
sensor *potentiometer_1 = createSensor(
    (float *)malloc(sizeof(float) * DATA_BUFFER_SIZE_POTENTIOMETER_1),
    WIDTH_POTENTIOMETER_1,
    PRECISION_POTENTIOMETER_1,
    DATA_BUFFER_SIZE_POTENTIOMETER_1,
    PUBLISH_PERIOD_POTENTIOMETER_1,
```

```
    SAMPLING_PERIOD_POTENTIOMETER_1,
    mqtt_topic_potentiometer_1,
    conversionPotentiometer_1,
    sampleLinear_10kohm,
    PIN_POTENTIOMETER_1);

// DSL-specified duration of actuation
#define DURATION_TRIG_1 15000

// DSL-specified trigger condition function
int conditionTrig_1(float arg)
{
  if (arg > 556)
    return true;
  return false;
}

// Trigger data structure creation helper function
trigger *trig_1 = createTrigger(conditionTrig_1,
                                pump, // Pointer to an actuator
                                DURATION_TRIG_1);

// Initialisation helper funciton
sensor_addTrigger(potentiometer_1, trig_1);
```