Reconfigurable Storing, Scanning and Moving Installation

Group 01

Frederik Alexander Hounsvad
University of Southern Denmark
Odense, Denmark
frhou18@student.sdu.dk

Nikolai Emil Damm
University of Southern Denmark
Odense, Denmark
nidam16@student.sdu.dk

Peter Andreas Brændgaard
University of Southern Denmark
Odense, Denmark
pebra18@student.sdu.dk

Oliver Lind Nordestgaard
University of Southern Denmark
Odense, Denmark
olno18@student.sdu.dk

Troels Zink Kristensen
University of Southern Denmark
Odense, Denmark
tkris17@student.sdu.dk

1 Group report

Extended Summary

This project covers the following subjects from Model-Driven Software Development (SM2-MSD), Software Engineering of Internet of Things (SM2-IoT), and Software System Analysis and Verification (SM-SSAV):

- SM2-MSD: External Domain-Specific Language (DSL), Xtext, Xtend, verification and scoping, code-generation, configuration language, logic language.
- SM2-IoT: Network protocols, Connectivity, Provisioning micro controllers (ESP32s, Raspberry Pi), MQTT, Placement of logic, actuators and sensors.
- SM-SSAV: Model-checking with UPPAAL, Code-generation of UPPAAL project, verification and validation of requirements.

1.1 Problem and Objective

The Problem - As the world is moving to industry 4.0 a new set of challenges arise; devices are becoming more interconnected, and a new level of autonomy is reached. How this revolutionary step will affect industrial production is exciting. In particular, it is interesting to examine some of the challenges that occur when building production systems that operate on industry 4.0 technology. Especially how such systems remain reliable and safe, and how they can be built to be re-configurable and highly flexible.

The Objective - This project sets out to examine the problem space by utilizing knowledge and tools from Model-Driven Software Development, Software Engineering of Internet of Things, and Software System Analysis and Verification to build a re-configurable storing, scanning, and moving installation that runs on industry 4.0 technology. The installation is required to operate reliably and safely.

1.2 Problem Description

The problem description describes the problem in more detail by formulating the background of the problem and a proposed solution.

1.2.1 Background of the Problem

During the industrial growth as part of industry 3.0, automated production became commonplace. Production lines that operated automatically from computer programs, revolutionized the industry, allowing for autonomous processes e.g. production lines. Manufacturing large quantities of products like furniture, electronics or food became even more accessible. [?]

However, the introduction of autonomous robots did not come without a cost. Autonomous assembly lines are for example still required to run near personnel, and in some cases, they must be operated and configured by said personnel. This requires such installations to be safe, e.g. limiting interaction, responding safely to external impediments, and operating reliably. As the world is now moving to industry 4.0, it makes everything even more complex, as interconnectivity and more autonomy put even more emphasis on reliability and safety. [? , ch. 6]

The added autonomy to assembly lines will require a new level of trustworthiness, as technology shifts often do. How can personnel be confident that the installations around them will not be dysfunctional if e.g. a camera sensor dies? Industry 4.0 systems with many moving parts are more complex, and the risk for failure is increased. In case of connectivity issues or dysfunctional devices, systems must be able to adjust behaviour accordingly. [? , ch. 5]

Without the inclusion of these subjects, it would be hard to create a re-configurable, reliable and safe assembly line running on industry 4.0 technology.

1.2.2 Proposed Solution

The proposed solution to the problem is divided into separate entities that include the three courses, as described below:

- The devices within the assembly line will be actuated and controlled by different IoT-devices, such as an ESP32 to ensure physical connection for the hardware equipment, but also a WiFi connection to ensure remote control by MQTT-topics.
- An external DSL will be created to ensure a simple and human-readable programming interface to set up the devices and create the logic of the system.
- An implementation of the DSL will be code-generated in C# to work with MQTT.
- From the implementation of the DSL, a UPPAAL-model will be code-generated as well.
- Queries will be created to verify the functional requirements defined to ensure a safe and reliable system.

1.3 Solution Approach

The system in its entirety is a prototype of an autonomous assembly line. It simulates a large-scale assembly line in storage facilities where safety is critical. The system consists of three different devices: cranes, disks and cameras. There is no limit to the number of each device type.

The responsibility of the disks is to store items in a facility. The disks have a set of slots to contain the items and several zones that correspond to input, output, and processing locations in a large-scale assembly line. Three zones will be created at a minimum; that is one for crane operations, camera scanning and incoming items on the assembly line.

The responsibility of the crane is to move items from a disk to storage containers matching the item type.

Cameras scan the colours of the items located on the disks. The colours are used to simulate item types, and as such the cranes use the information to decide which storage container to put items into.

A full overview of the system and how the physical devices are controlled remotely by software is presented in Appendix **??**.

1.3.1 Domain-Specific Language

A domain-specific language has to be developed to make programs for the assembly line. These programs should be simple and easy to write for a non-programmer. This is essentially the point of a DSL, but this language has been decided to be a natural language to make it look more like the English language. An external DSL should be implemented as opposed to an internal DSL to make the language fully configurable to fit an assembly line.

The DSL should be able to configure all three devices. Within the configuration, a name should be specified for each device, as it is possible to have multiple objects for each device. Furthermore, each device has additional configurations. The crane should have positions defined for locations it can move to, e.g. positions at the disk and the storage containers. The disk will need a predefined number of slots, and the number of zones to specify where the physical locations of the other devices are. At last, the camera will need a configuration to define which colours it will be able to scan.

The DSL should also support defining the logic of the program. This logic instructs what the devices should do in different situations. An example of both configuration and logic is shown in Appendix **??**.

Having a DSL allows for the fast and easy deployment of an assembly line. The workers, which are installing the setup, require little technical knowledge and only need to know how the DSL works. However, due to the simplicity of the DSL, it is also quite limited in functionality. Expanding functionality requires either mixing the DSL with manual implementations or expanding the DSL itself. The first option makes for a complicated working process that will most likely result in high amounts of technical debt. The second option is a time-consuming task.

1.3.2 Networking, Communication between Devices, and Logging

Networking **-** For the project, an isolated network is used. It is isolated in the sense that the network exists behind a NAT. This shields the devices from direct interaction with outside connections. Isolating them on their own wireless AP's also serves to increase the communication stability, as dysfunctional devices can make the communication to an AP very unstable.

MQTT **-** Communication across the system is done through MQTT. MQTT is a publish-subscribe system that facilitates communication via topics. Topics in MQTT are text strings that describe where messages are sent to. MQTT topics allow for only one message at a time but can be set to retain the last message such that any client who connects after the message has been sent are able to view the message. The topics in MQTT are normally structured by forward-slashes to signify association, and an example of this could be `/log/camera1/error`. In this example we see a base level of `log`; this name in our solution signifies an association with logging for any lower topics. We then reference `camera1` which likewise signifies association with the camera. We can also use wildcards to reference these topics, such as using a wildcard instead of the `camera1`. This would allow gathering all errors which are referenced by `/log/[devicename]/error`.

Logging **-** Logging is currently done by the devices publishing logs to MQTT topics depending on the severity of the data. These topics are structured such that the last part of the MQTT topic corresponds to the severity, for example, `/log/camera1/error`. Additionally, to facilitate dynamic logging levels, the devices subscribe to a `SetLoggingLevel` command, which controls what information the device will log to the MQTT broker. To collect the logs from the MQTT broker and store them in a database, a custom Python application is used. This application subscribes to all logging topics, and whenever a message is published, sends it to the influxdb database, which stores the logs. To visualize the logs, Grafana is used by connecting it directly to the InfluxDB database and creating panels that show errors over time and other such information. Such a graph can be seen in Appendix **??**.

1.3.3 The UPPAAL-Model

The DSL is used to generate a UPPAAL model. This UPPAAL model is used to verify certain requirements, which have been deemed suitable for any configuration of the system. This ensures that the configuration and factory logic implemented by a user can be verified for basic safety and reliability issues. The requirements of the model can be seen in Appendix **??**. However, some requirements will not be verified. These are: 1.b, 1.c, 1.d, 1.e, 1.i, 1.j, 2.e, 2.f, 2.j, 2.k, 3.c and 3.d. These will be omitted due to scoping.

UPPAAL can then be used to verify the combined model of the generated templates. This is done by also generating queries that match the generated model. These queries try to match the requirements so that it can be verified that the requirements are true for the implemented system.

This approach means that any implementation of the DSL can easily be verified to fulfil the listed requirements. However, as the UPPAAL model is generated, it quickly becomes so big that verifying the properties is computationally heavy and takes a long time. Creating one model means that all states have to be evaluated at once.

1.4 Solution Description and Results

1.4.1 Implementation Prototype

The system is designed to support any number of devices, which are programmed with the DSL. To test the system, a prototype implementation was used throughout the project.

The prototype implementation uses one of each device to match the scope of the project and to ensure a minimal viable product is feasible in the project time frame.

The setup is shown in Appendix **??**. Here it can be seen how the orchestrator, running on a Raspberry Pi, communicates with the different devices through an MQTT broker. A program also listens for any messages on logging topics and saves them in a database.

This is all based on an example implementation of our DSL, which can be seen in Appendix **??**, and it is also the implementation that controls the program flow in the video attached to the paper. The function of this implementation is to scan incoming items, and then let them dry based on their colour. When they are done drying, they are picked up by the crane and placed in the right box, which fits with the item's colour.

1.4.2 Factory Language (.fl)

The DSL created according to section **??** is implemented with a focus on readability, usability and on providing a flexible and extensible metamodel. The grammar consists of 39 parser rules, 5 enums, and 2 terminals. Furthermore, the grammar relies a lot on the `returns` keyword to design a hierarchy for different parser rules and ultimately the metamodel. **??** in Appendix **??** gives an overview of the metamodel parsed by the grammar language.

The root parser rule, `Model`, contains one or more configurations and statements, where the configuration parser rules parse text blocks that define devices and device properties. The statement parser rules parse text blocks that define the logic of the final system.

Configurations support different device properties e.g. configurable slot count, disk zones, crane positions and scannable colours for a specific camera. All these device properties are referable by the statements, such that the logic can operate on the provided properties.

There are three different types of statements - conditionals, loops, and operations. Conditionals can parse if-statements with various conditions and operands, to support basic program flows. Loops can parse for-each statements that allow program flows where statements must be repeated while a condition is not satisfied. Operations define different instructions that the different devices can execute, e.g. `DiskMoveSlotOperation` which commands the disk to move a slot to a specific zone. The operations are split into hierarchies of `DiskOperation`, `CraneOperation`, and `CameraOperation`, such that each operation type can be extended with new commands as required.

Factory Language has a set of validation and scoping rules. The validation rules ensure that only valid input is allowed. This is very helpful to the user because Factory Language uses natural language for its grammar, and this means that there are some inconsistencies in operations and statements to ensure grammatically correct expressions. A couple of examples for validation rules are listed below:

- Crane positions must be between 0 and 359 degrees as these are the lower and upper limits of angles.
- A disk cannot have more zones than available slots.
- Drying time for items must be equal to or larger than 1 second.

Scoping rules were necessary to enable device properties to be usable by statements. As such, scoping has been implemented for all variables and references.

Lastly, Factory Language is implemented to be whitespace aware, where nested elements must be indented. This provides a better reading experience, and because validation errors tell the end-user it is required should make it obvious if indentation is new to the user. Indentation is implemented with the terminals: `synthesize BEGIN` and `synthesize END`. The terminals tell the parser when indentation is required, and using `synthesize` with terminals is recommended by Xtext documentation [**?**].

The grammar language results in the Factory Language (.fl). An example is shown in Appendix **??**.

1.4.3 Generated Artifacts

From the Xtext project, two distinct artefacts are generated - (1) an orchestrator, and (2) a UPPAAL project.

The Generated Orchestrator - The orchestrator is a C# service that translates the Factory Language into an executable program, which functions as a main controller for the assembly line.

The orchestrator consists of the following generated files and folders:

- A `Program.cs` class.
- An `Mqtt` folder with an `MqttService` and `MqttTopics`.
- An `Entities` folder with classes representing the programmable devices in the assembly line.
- A `Dockerfile`.

The `Program.cs` class is the entry point for the application, and it is also this class that defines the main loop that continuously executes code equivalent to a given program flow defined by a Factory Language's set of statements.

The `MqttService` class includes everything needed to send and receive messages from the MQTT broker. The service is injected in nearly all entity classes, as well as being initialized and used in the `Program.cs` class.

The entity classes consists of:

- `Camera.cs`
- `Crane.cs`
- `Disk.cs`
- `Slot.cs`
- `SlotState.cs`

Each of the entity classes has methods corresponding to the different operations that they provide, e.g.:

The `MoveSlot` method moves a slot from one zone to another, but it also demonstrates how many of the operation methods are asynchronous and utilizes tasks, such that operations can await each other, or run in parallel where possible.

The generated `Dockerfile` is part of a continuous integration workflow, where a GitHub Action poll for changes in the Orchestrator service, and if any changes are applied, a Docker image is built and pushed to a GitHub Container Registry. From there the latest image can be pulled to the Raspberry Pi for easy deployment of the assembly line.

The Generated UPPAAL Project - The generated model creates different templates depending on the configuration defined in the DSL. A master controller template is always created, which reflects the statements part of the DSL. Each device specified in the configuration of the DSL also results in one or more templates. A crane is represented by two templates;

- Crane template: This template reflects the position of the crane's arm and whether the magnet is raised or lowered.
- Crane Magnet template: This template only shows the state of the magnet.

A disk is the most complicated entity to model and is represented by nine templates. However, several of these templates are identical in functionality as they only deal with different variables. This means that in reality there are four different templates created for a disk;

- Disk template: This template represents how the disk is currently positioned. It is also responsible for communicating that an item has been added or removed to the correct slot. The master controller only requests adding or removing an item and not where to put it.
- Disk Slot template: This template represents a disk slot on the disk. It is therefore copied per slot defined in the DSL. The disk slot keeps track of if the slot has an item and what colour if any, the item has.
- Disk Get Empty / Complete / Free / In-progress Slot template: These templates are used by the master controller to get a slot that is in the specified state.
- Disk Slot Variable - Complete / Free / In-Progress template: These templates are used to toggle the specified state of a disk slot.

Finally, a camera is only a single template that chooses a random colour, when asked to scan.

1.4.4 IoT setup

Networking - The networking that the system requires is while simple, a very critical part of the system. For the communication between the devices and the orchestrator, a network is needed. The ESP32 devices that are to control the crane and the storage, require that the network conforms to the Wi-Fi standards IEEE 802.11b/g/n[?] on a 2.4 GHz band. To achieve this, a Raspberry Pi 4b (RPI4) is acting as the access point. It has been set up to use its RJ45 Ethernet connector as a WAN port to allow the system to be connected to any network. It provides NAT functionality on the WLAN to facilitate that the connection happens without directly exposing the IIoT devices to the rest of the network that they are connected to.

The Physical Setup - The setup itself is well defined by the diagram seen in Figure **??**. It is in theory composed of three IIoT devices. The crane, the storage disk, and the camera colour scanner. These devices would then be connected up to a dedicated network, and the services such as the orchestrator and database would be hosted on one or more servers. In practice, the system has two standalone devices in the crane and disk. The scanner, AP and service hosting are then handled by the RPI4. Optimally, these services should be deployed to different physical devices to remove the single point of failure. However, here it serves as a proof of concept.

With the programming of the devices shortcuts were taken. For the ESP32's code, the Arduino framework was used. This was done to lower complexity and shorten the prototyping time. The Arduino framework offers a lot of simplifications, for things like writing to i/o pins. It also allows the use of libraries for common peripherals such as stepper motors. Nonetheless, using the framework locks down what can be done such as the multi-threading offered by FreeRTOS. It also hampers the performance of the programs, as much of it is poorly optimized for space and speed.

1.5 Discussion

The final system has been evaluated on whether it satisfies the requirements defined for model-checking (see Appendix **??**) and if it satisfies the course objectives (see Appendix **??**).

D.1.1.* - The functional requirements for the disk are satisfied by making it possible for the disk to rotate, occupy slots, and rotate slots to positions.

D.1.2.* - The functional requirements for the crane are satisfied by making the crane able to rotate, pick up, drop, and carry around items without dropping them prematurely.

D.1.3.* - The camera can scan the colours of items, and publish the colours to MQTT.

D.1.1.f, D.1.1.g, D.1.2.g, D.1.2.h - The stop function requirements are satisfied by having a working emergency stop function on the crane and disk. The stop function is not able to resume operations without a full reboot, as such, it does not satisfy D.1.1.h and D.1.2.i.

D.1.1.i, D.1.1.j, D.1.2.j, D.1.2.k, D.1.3.c, D.1.3.d - The connectivity requirements are satisfied by using ESP32s and a RaspberryPi 4B as microcontrollers, as they have a WiFi module and support MQTT. Furthermore, remote control is enabled by the code-generated orchestrator utilizing MQTT and WiFi connectivity to the IoT devices.

??*, and* ?? - The learning objectives are satisfied, as topis from SM2-MSD, SM-SSAV, and SM2-IoT are used extensively to both build the system and to describe the system.

?? - Functional and non-functional requirements have been defined for the system in Appendix **??**, and the system has been built, tested and evaluated with these requirements in mind.

The state of the system is demonstrated in the video demo attached to the hand-in of this paper. It demonstrates the system's success, but also its weaknesses.

As the system does not take physical space and its surroundings into account when determining where it is concerning the disk, the system had some precision error, as the disk and crane had to be positioned and placed correctly to perform as expected. Improving the precision would require rebuilding parts of the crane or disk. It could for example require that the crane and disk were mounted together, making the crane arm extensible to adjust for precision, or replacing wires to the electromagnet with softer ones that do not affect the position. As the precision of the hardware was not critical for the project's objective, rebuilding the system to fix it was not a priority, and in the demo, the precision error is corrected by pushing the electromagnet a little bit when necessary.

In particular, three requirements were not satisfied. A resume function after an emergency stop was implemented for neither the disk nor the crane; likewise, it was not made possible for the camera to retry scanning if it fails. Due to time constraints, it was not feasible to implement the resume function, and as such requirements D.1.1.h and D.1.2.i were not satisfied. The need for a retry scanning function for the camera never occurred, so requirement D.1.3.b was not prioritized. In reality, these requirements are important for a real assembly line, as any failure can have huge costs for a production facility, and making sure a system can gracefully fail, and later resume is important to minimize maintenance.

In summary, the final system has a couple of advantages and disadvantages:

+ System is highly configurable and flexible.
+ Configuring devices and creating a program flow have very low complexity.
+ Auto code-generating UPPAAL projects with a basis in a real system are possible (with fixed queries).
+ System is operational.
– Physical system is imprecise.
– UPPAAL queries are not dynamically defined in the DSL.
– UPPAAL model checking is computationally heavy.
– IoT setup is not supported in the DSL.
– Expanding or modifying the system is a large task, as the entire flow from DSL to generated code has to be implemented.

1.6 Conclusion

To solve the problem and objective stated in **??**, a prototypical assembly line system has been built. It can move, store and scan items, with three separate devices, that are interconnected utilizing MQTT. Items put on a disk is automatically moved to a camera, that can scan items, so they can be picked up by a crane and be put into a storage container matching the scanned colour. The system operates both synchronously and asynchronously, which means every device in the system is capable of executing tasks individually, but they await each other in cases where necessary. This ensures that, for example, the disk waits for the crane to have picked up an item from the disk, before turning itself around to the next item.

The problem and objective have been satisfied by the following achievements:

1. An external DSL has been successfully developed in Xtext (consisting of both configuration and logic) to create a simple and natural language that non-programmers can use to program their assembly line. Verification and scoping rules have been added to the DSL to ensure that the end-users cannot write something that is not allowed. All necessary artefacts in C# have been code-generated in Xtend, which makes the process of creating new programs that the system understands automatic.

2. A Raspberry Pi has been included to enable the orchestrator to use Wi-Fi to send commands through MQTT-topics across all devices. All hardware in the system has been connected to two different ESP32s to make sure that all parts of the system can respond correctly to the commands sent by the orchestrator.

3. The logging level is remotely dynamically changeable for the devices, in a way where they can save on bandwidth if needed by lowering the logging level. It is stored in a database and can be searched and shown as needed.

4. Model checking has been used through UPPAAL-templates that cover all the processes within the system. All templates have been code-generated by the DSL. This means that both the C# files and the UPPAAL-templates are all being generated by the same program, automatically. Queries have been written to verify and validate the requirements. The results indicate that the system mostly behaves reliably and safely.

2 Individual Report

Extended Summary

Write an Extended Summary for the individual report. As part of this you must highlight with one bolted point the elements from each of the 3 courses.

2.1 Problem Description

Provide a more detailed problem description here. What is the problem about? Characterize the problem in a way that allows for deriving a solution.

2.2 Solution Approach

Describe the solution approach on a high-level including advantages and drawbacks based on relevant literature.

2.3 Solution Description and Results

Describe the solution, test, and evaluation results (note: remember to refer to what the video covers).

2.4 Conclusion

Short conclusion summarizing the project, achieved solution and results.

Appendix

A System Overview

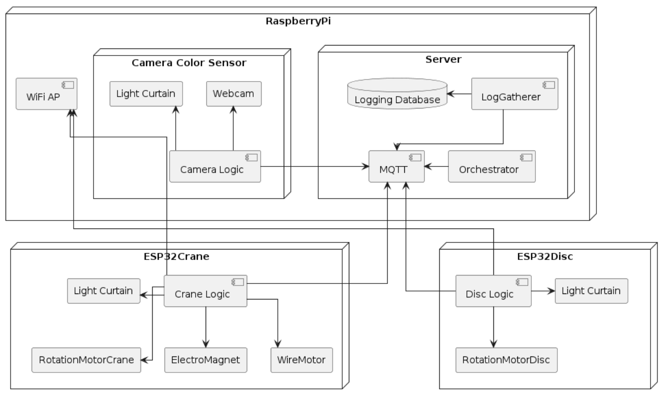


Figure 1: Overview of the system and all its components.

B Factory Language (.fl)

B.1 Metamodel

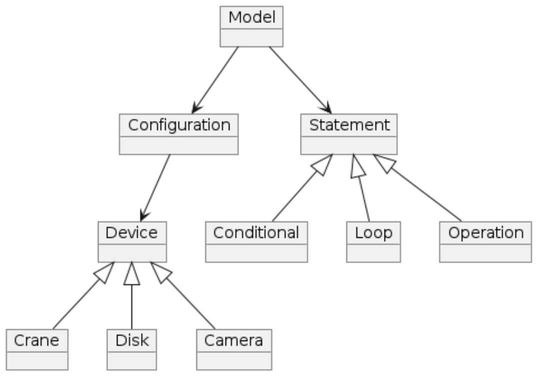


Figure 2: An overview of the metamodel.

B.2 Grammar Language

B.3 Example Factory Language program

C The Orchestrator

C.1 Program.cs

C.2 Dockerfile

C.3 GitHub Action

D Functional and non-functional requirements

D.1 Functional requirements

(1) Functional requirements for the disk.
   (a) The disk must be able to rotate items from one location to another.
   (b) The disk must be able to rotate the shortest path.
   (c) The disk must be able to avoid tangling wires while rotating.
   (d) The disk must have configurable slots for items.
   (e) The disk must have configurable zones for operations, e.g., scanning, crane pick up, intake.
   (f) The disk must have a stop function that conforms to danish machine safety standards.
   (g) The disk's stop function must halt all movement if the system is in an unsafe state.
   (h) The disk's stop function must be able to resume operations when the unsafe condition is resolved.
   (i) The disk must have WiFi connectivity.
   (j) The disk must be controllable with MQTT.

(2) Functional requirements for the crane.
   (a) The crane must be able to rotate its jib from one location to another.
   (b) The crane must be able to raise and lower with its hoist.
   (c) The crane must be able to pick up and deliver items.
   (d) The crane is not allowed to drop items before it reaches its target location.
   (e) The crane must be able to rotate the shortest path.
   (f) The crane must be able to avoid tangling wires while rotating.
   (g) The crane must have a stop function that conforms to danish machine safety standards.
   (h) The crane's stop function must halt all movement if the system is in an unsafe state.
   (i) The crane's stop function must be able to resume operations when the unsafe condition is resolved.
   (j) The crane must have WiFi connectivity.
   (k) The crane must be controllable with MQTT.

(3) Functional requirements for the camera.
   (a) The camera must be able scan colors of items.
   (b) The camera must be able to retry scanning if it fails.
   (c) The camera must have WiFi connectivity.
   (d) The camera must be controllable with MQTT.

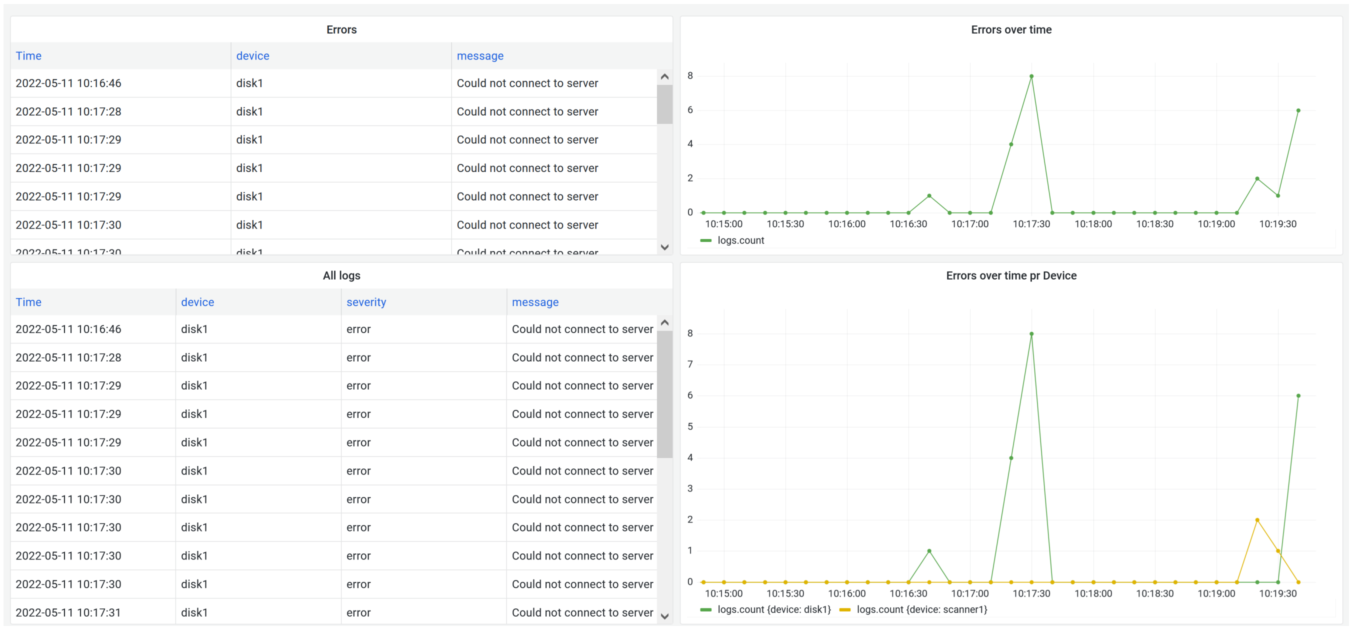
D.2 Non-functional requirements

(1) Non-functional requirements for the disk.
   (a) The disk must operate reliably.
   (b) The disk must operate safely and not cause harm to people.

(2) Non-functional requirements for the crane.
   (a) The crane must operate reliably.
   (b) The crane must operate safely and not cause harm to people.

(3) Non-functional requirements for the camera.
   (a) The camera must operate reliably.

E Logs Graph



Errors

Time	device	message
2022-05-11 10:16:46	disk1	Could not connect to server
2022-05-11 10:17:28	disk1	Could not connect to server
2022-05-11 10:17:29	disk1	Could not connect to server
2022-05-11 10:17:29	disk1	Could not connect to server
2022-05-11 10:17:29	disk1	Could not connect to server
2022-05-11 10:17:30	disk1	Could not connect to server
2022-05-11 10:17:30	disk1	Could not connect to server

All logs

Time	device	severity	message
2022-05-11 10:16:46	disk1	error	Could not connect to server
2022-05-11 10:17:28	disk1	error	Could not connect to server
2022-05-11 10:17:29	disk1	error	Could not connect to server
2022-05-11 10:17:29	disk1	error	Could not connect to server
2022-05-11 10:17:29	disk1	error	Could not connect to server
2022-05-11 10:17:30	disk1	error	Could not connect to server
2022-05-11 10:17:30	disk1	error	Could not connect to server
2022-05-11 10:17:30	disk1	error	Could not connect to server
2022-05-11 10:17:30	disk1	error	Could not connect to server
2022-05-11 10:17:30	disk1	error	Could not connect to server
2022-05-11 10:17:31	disk1	error	Could not connect to server

Figure 3: Logs graph

F Trustworthy Systems - Course objectives

F.1 Learning objectives - Knowledge

- Obtain an understanding of and explain the topics that are associated with the project.
- Obtain an understanding of the elements of engineering projects with a scientific purpose.

F.2 Learning objectives - Skills

- Identify, analyse and make qualified choices for the design of trustworthy systems given functional and non-functional requirements.
- Implement, test and evaluate trustworthy systems in regard to functional and non-functional requirements.

F.3 Learning objectives - Competences

- Conduct software development within topics that are associated with the project.
- Carry out professional engineering use of software technologies in development of software solutions within the topics that are associated with the project.
- Work structured and scientifically with engineering solutions to open-ended challenges and disseminate knowledge about the solutions orally and in writing.

G Templates

G.1 The Master Controller template

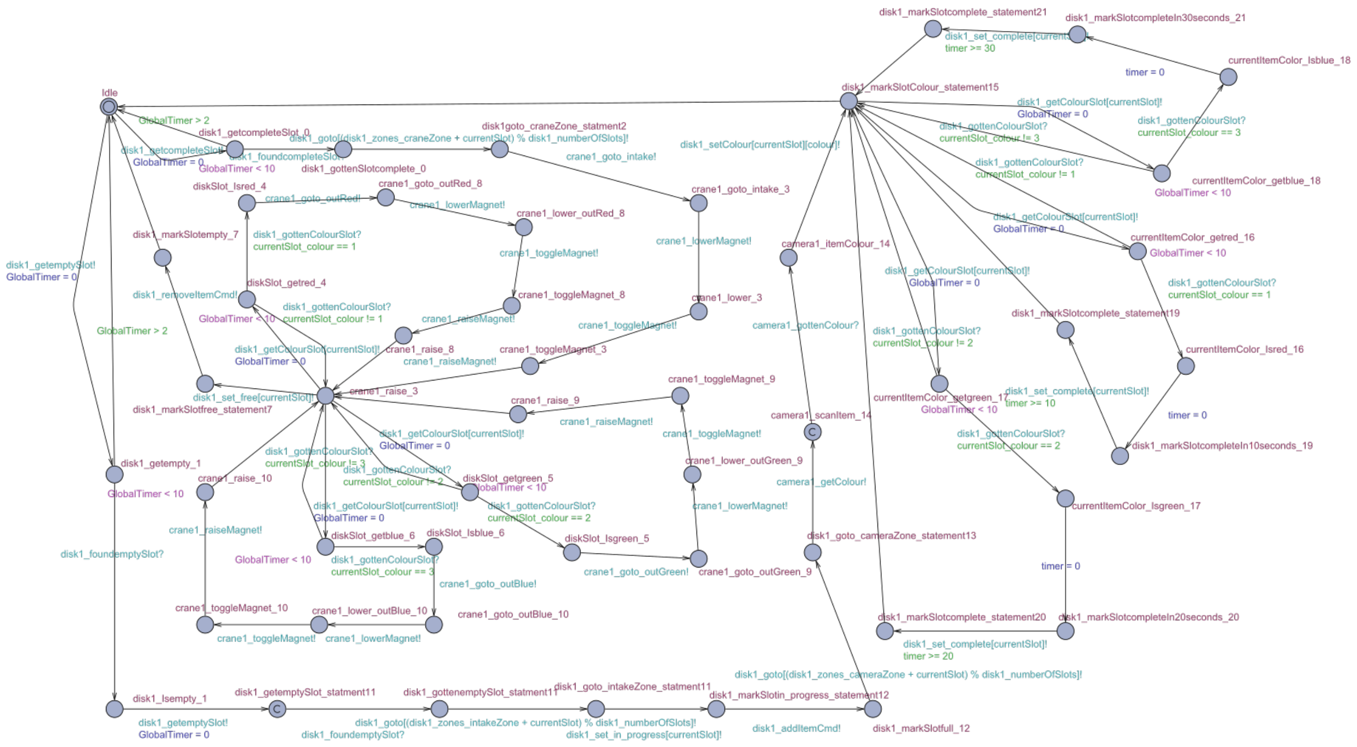


Figure 4: Master Controller.

G.2 The Disk template

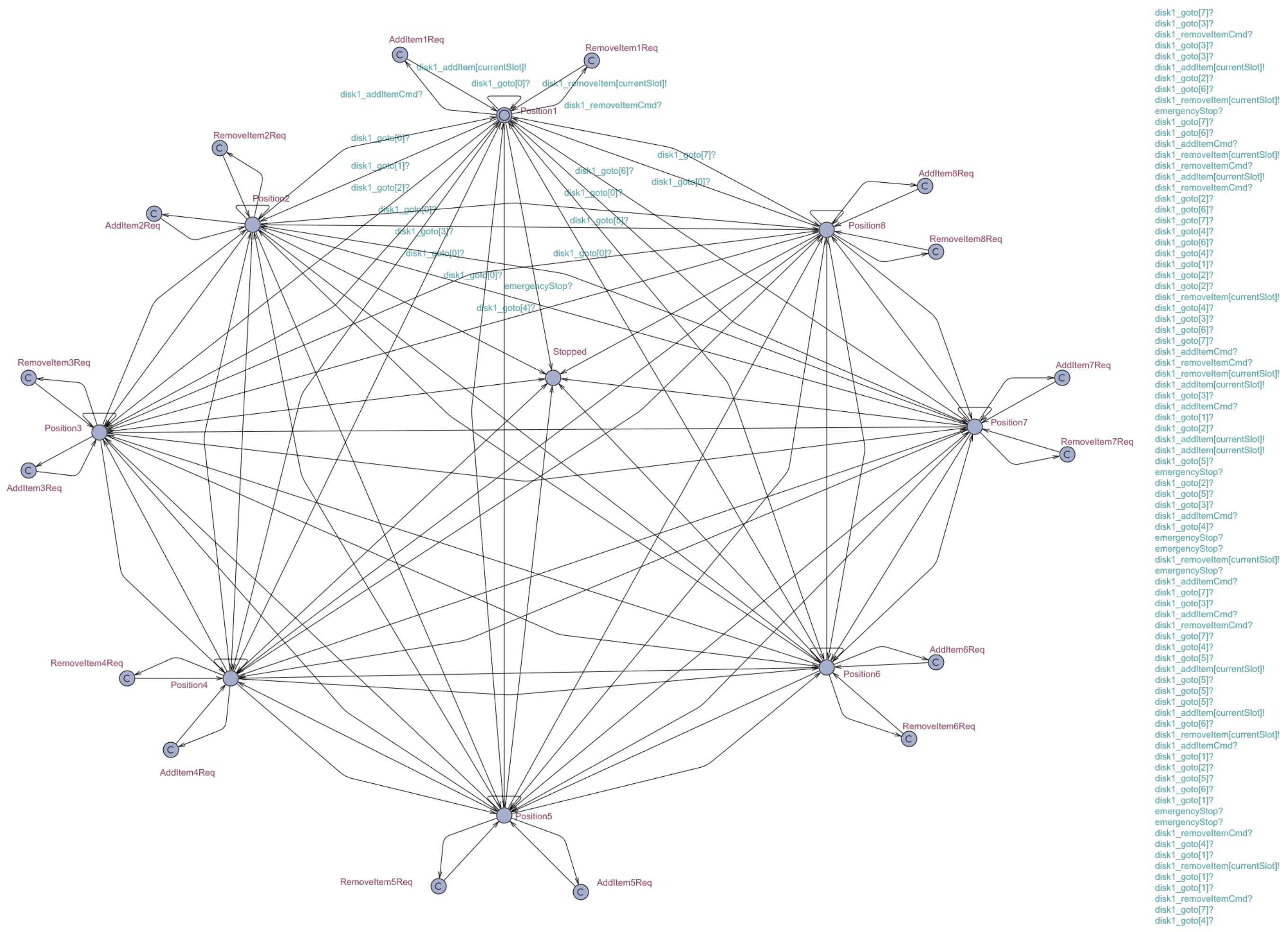


Figure 5: Disk.

G.3 The Disk Slot template

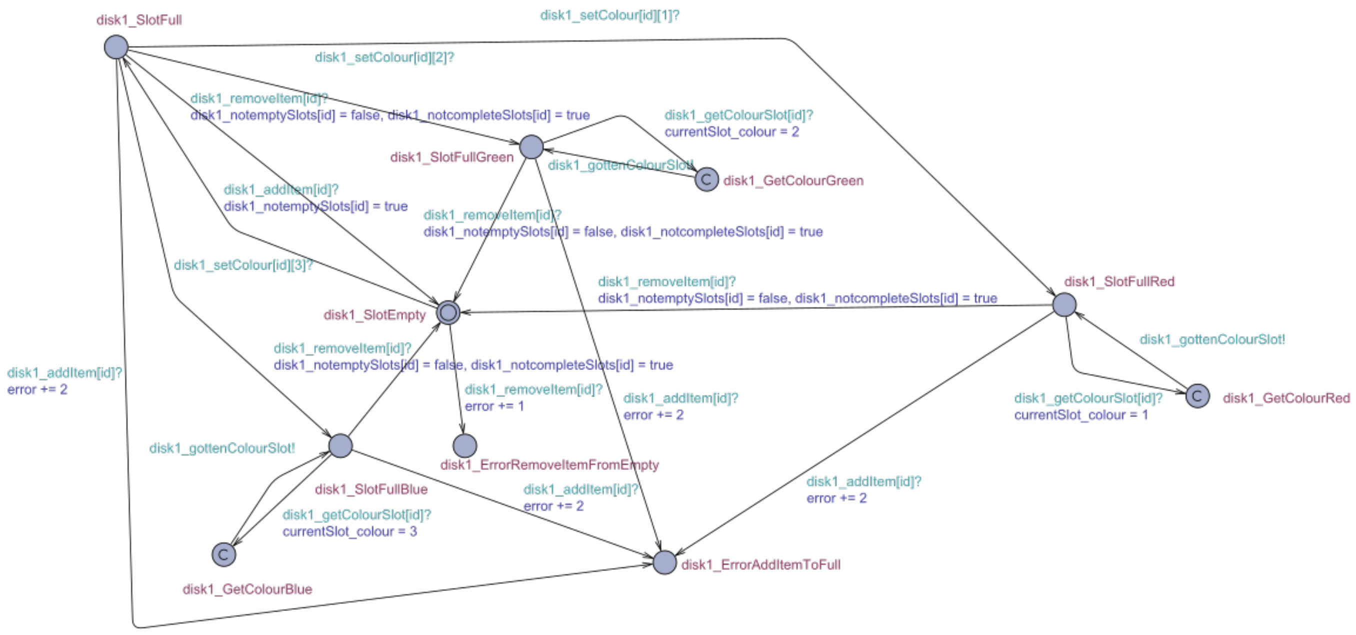


Figure 6: Disk Slot.

G.4 The Get Empty Slot template

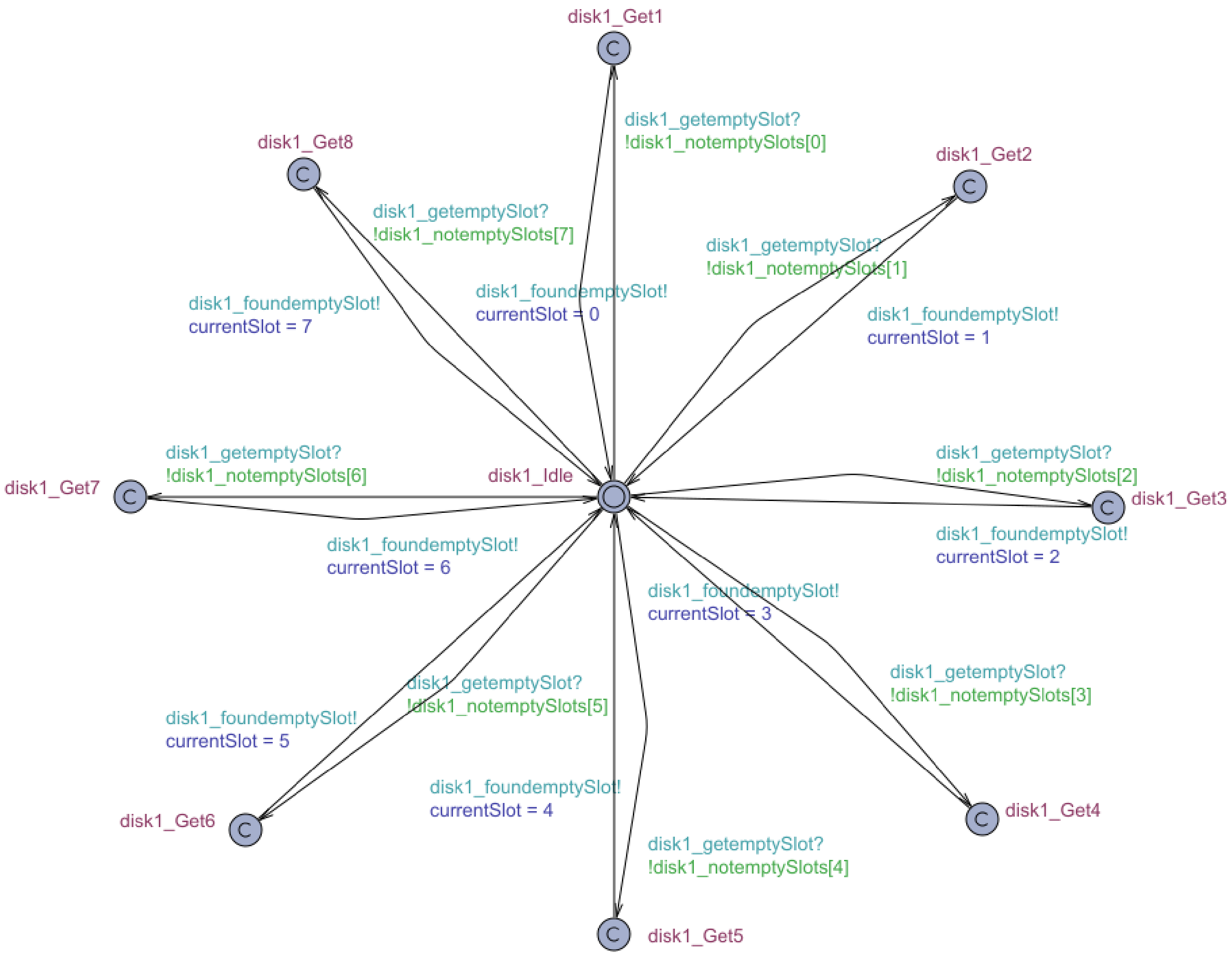


Figure 7: Disk - Get Empty Slot.

G.5 The Get Complete Slot template

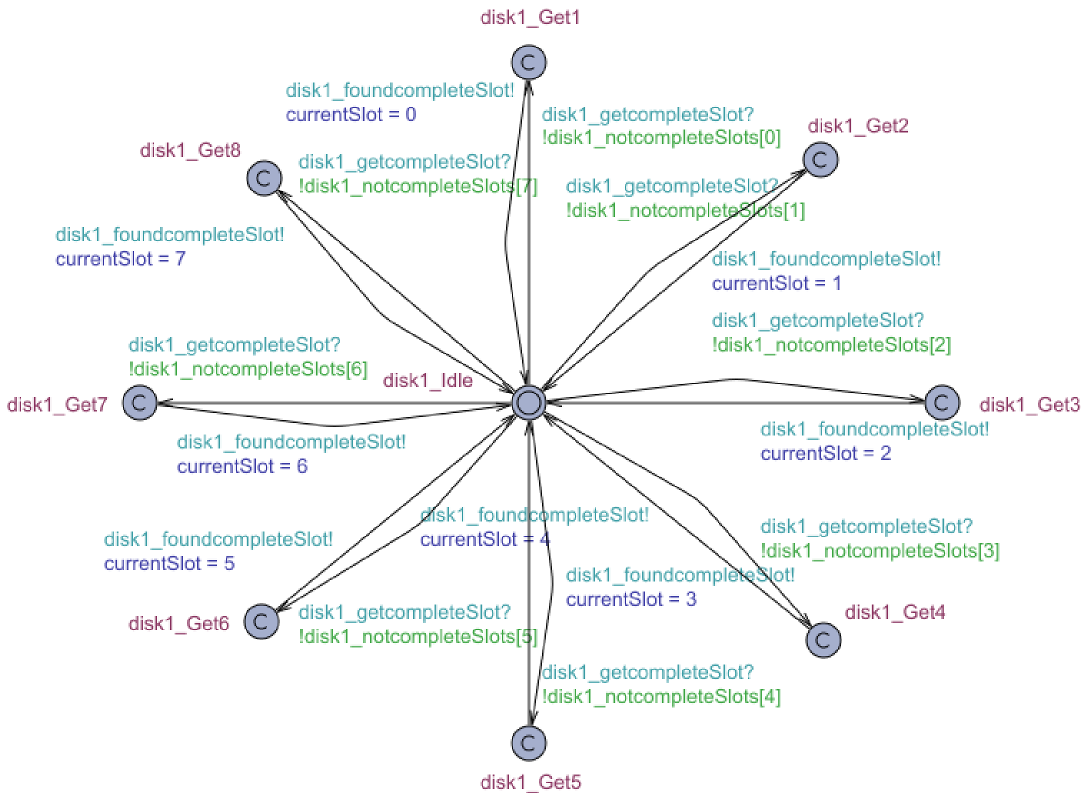


Figure 8: Disk - Get Complete Slot

G.6 The Disk Get Free Slot template

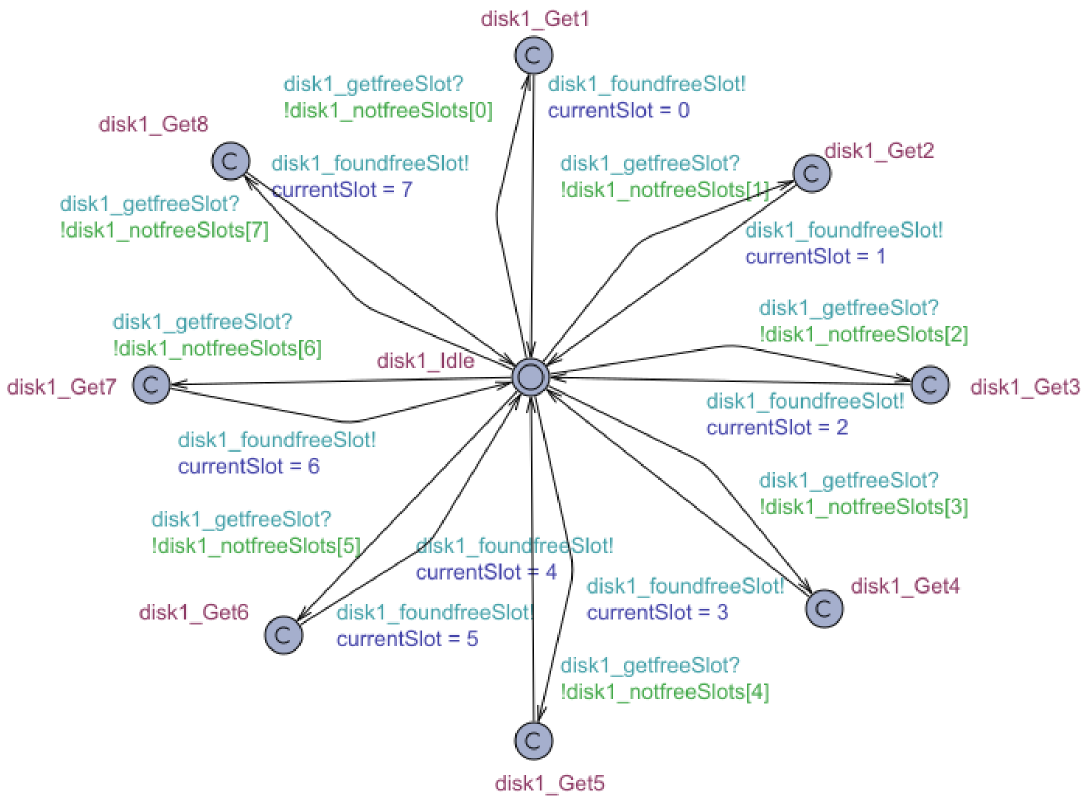


Figure 9: Disk - Get Free Slot

G.7 The Get In Progress Slot template

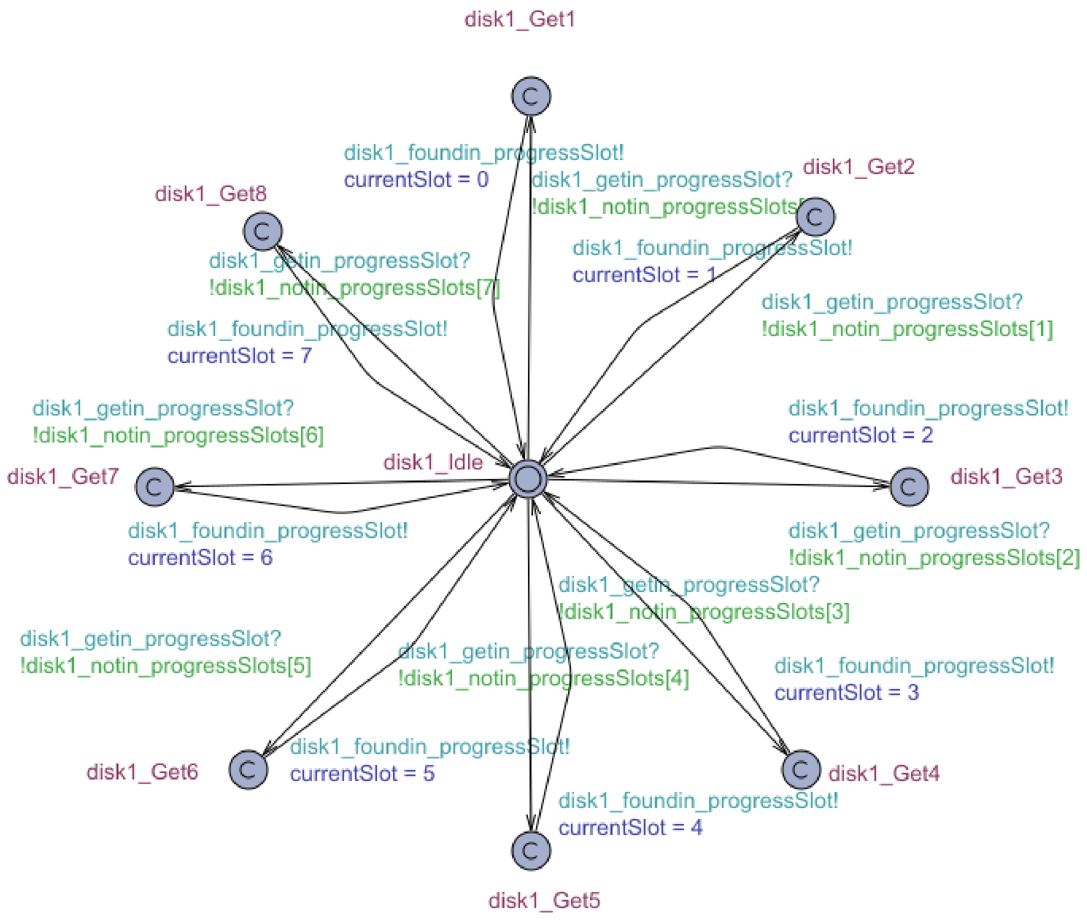

disk1_Get1
disk1_foundin_progressSlot!
currentSlot = 0
disk1_getin_progressSlot?
!disk1_notin_progressSlots[0]
disk1_Get2
disk1_Get8
disk1_getin_progressSlot?
!disk1_notin_progressSlots[7]
disk1_foundin_progressSlot!
currentSlot = 1
disk1_getin_progressSlot?
!disk1_notin_progressSlots[1]
disk1_foundin_progressSlot!
currentSlot = 7
disk1_getin_progressSlot?
!disk1_notin_progressSlots[6]
disk1_foundin_progressSlot!
currentSlot = 2
disk1_Idle
disk1_Get7
disk1_Get3
disk1_foundin_progressSlot!
currentSlot = 6
disk1_getin_progressSlot?
!disk1_notin_progressSlots[2]
disk1_getin_progressSlot?
!disk1_notin_progressSlots[3]
disk1_getin_progressSlot?
!disk1_notin_progressSlots[5]
disk1_getin_progressSlot?
!disk1_notin_progressSlots[4]
disk1_foundin_progressSlot!
currentSlot = 3
disk1_foundin_progressSlot!
currentSlot = 5
disk1_Get6
disk1_Get4
disk1_foundin_progressSlot!
currentSlot = 4
disk1_Get5

Figure 10: Disk - Get In Progress Slot

G.8 The Slot Variable - Complete template

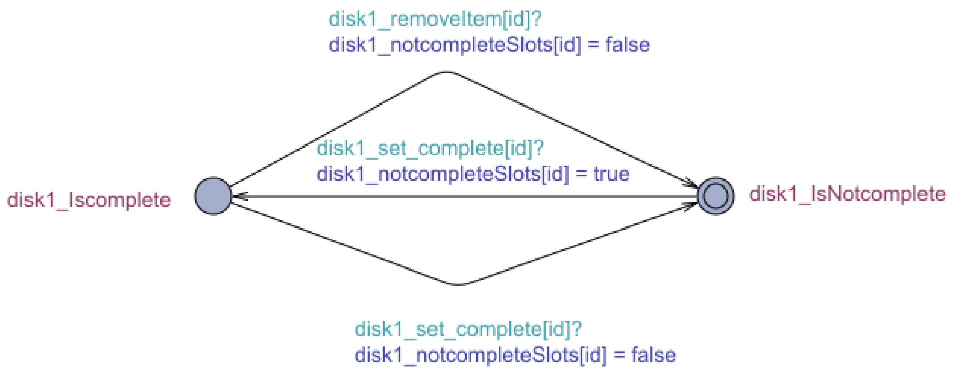


Figure 11: Disk - Slot Variable Complete

G.9 The Slot Variable - Free template

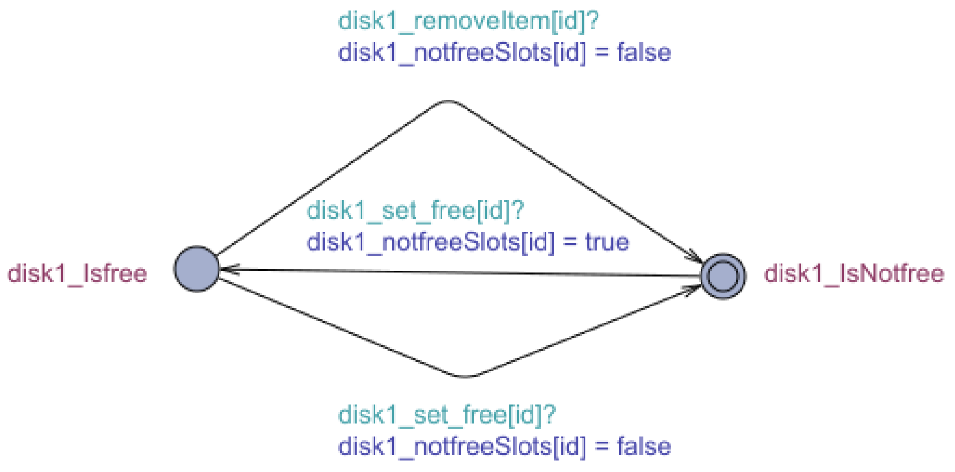

disk1_removeItem[id]?
disk1_notfreeSlots[id] = false
disk1_set_free[id]?
disk1_notfreeSlots[id] = true
disk1_Isfree
disk1_IsNotfree
disk1_set_free[id]?
disk1_notfreeSlots[id] = false

Figure 12: Disk - Slot Variable Free

G.10 The Slot Variable - In-progress template

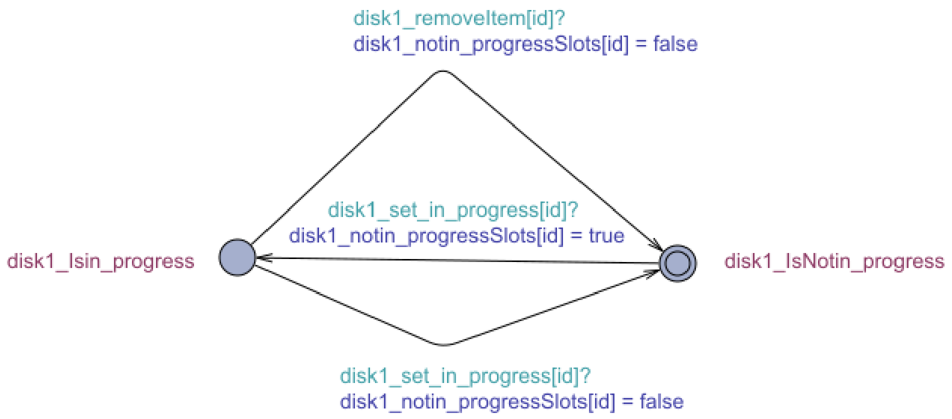

disk1_removeItem[id]?
disk1_notin_progressSlots[id] = false
disk1_set_in_progress[id]?
disk1_notin_progressSlots[id] = true
disk1_Isin_progress
disk1_IsNotin_progress
disk1_set_in_progress[id]?
disk1_notin_progressSlots[id] = false

Figure 13: Disk - Slot Variable In Progress

G.11 The Crane template

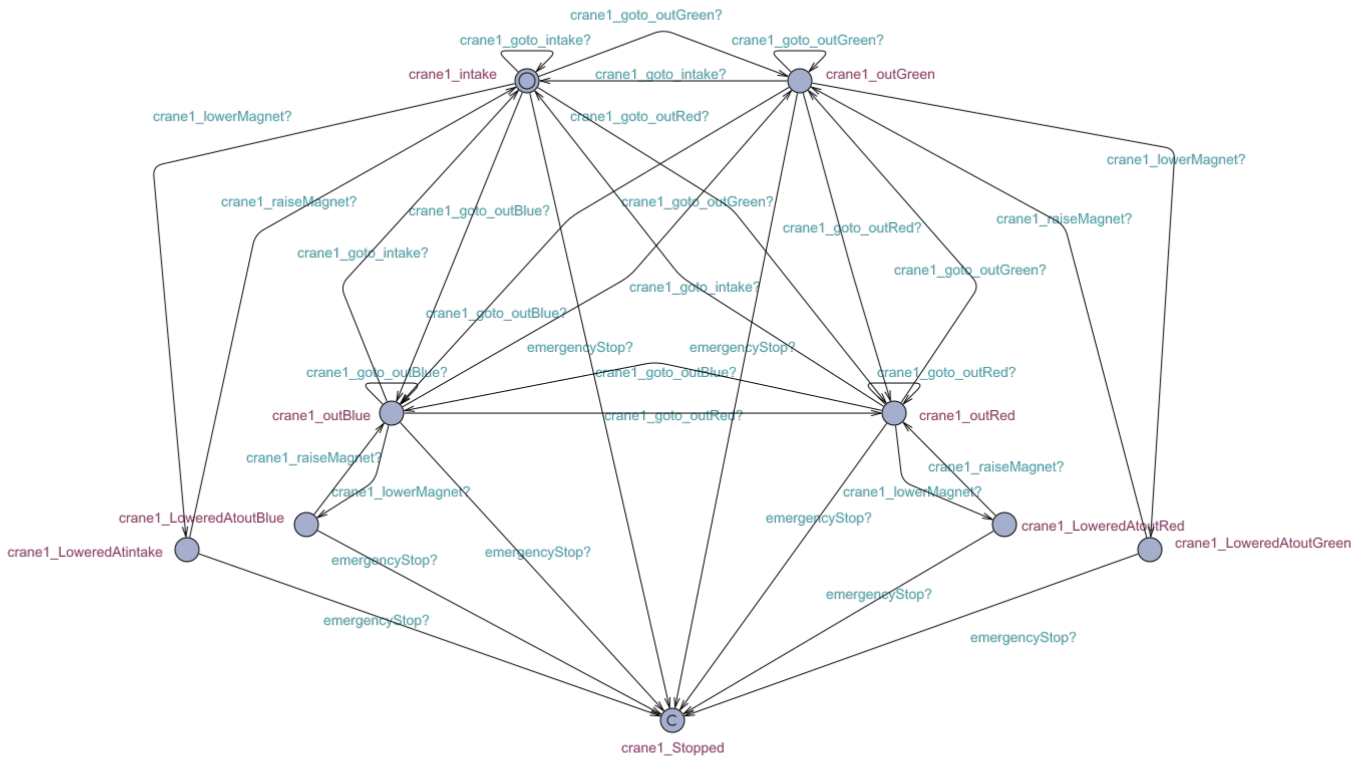


Figure 14: Crane.

G.12 The Crane Magnet template

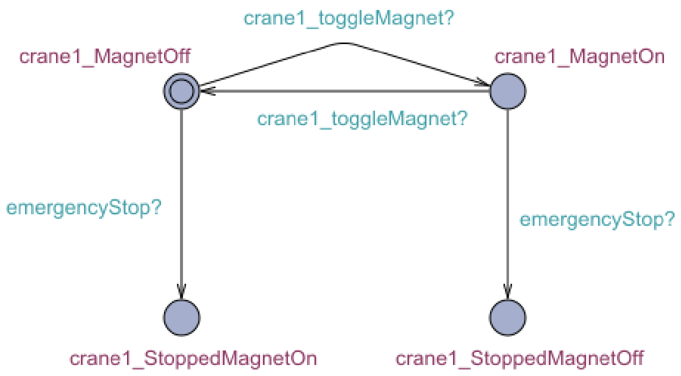


Figure 15: Crane Magnet.

G.13 The Camera template

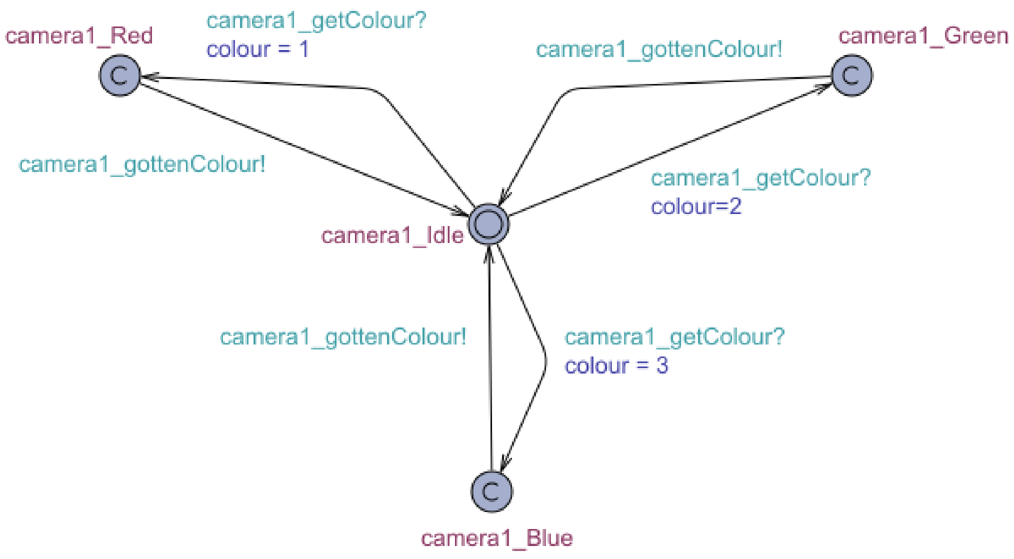


Figure 16: Camera.

G.14 The Emergency Button template

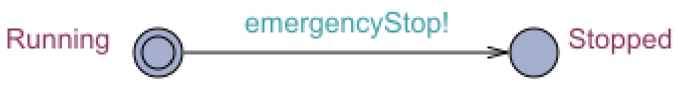


Figure 17: Emergency Button.