CHALMERS UNIVERSITY OF TECHNOLOGY

The ROVU Robot System

by

Arvid Björklund

Felix Engelbrektsson

Mattias Landkvist

Andreas Mann

Siavash Paidar

Chatchai Sornkarn

in the

Department of Computer Science and Engineering

February 2018

CHALMERS UNIVERSITY OF TECHNOLOGY

Abstract

Department of Computer Science and Engineering

by Arvid Björklund
Felix Engelbrektsson
Mattias Landkvist
Andreas Mann
Siavash Paidar
Chatchai Sornkarn

This course required the participants to produce a set of models describing a system that should control robots. The different models that were created were a domain model, a use case diagram, a component diagram, a class diagram, state machines, and sequence diagrams. Code should then be generated from the class diagram to then be further implemented to conform to the project description. This report, the final product in the course, compiles all the models and describes the different qualities of each and how they are supposed to be interpreted.

Contents

Chapter 1

Introduction

This report will cover all the assignments included in the TDA593, Model-Driven Software Development course, at Chalmers University of Technology and Gothenburg University. This introduction section will describe expectations on the team as well as describe the context of the ROVU system, around which the course revolves.

1.1 Guidelines

This project work was conducted by a group of six people with various backgrounds. To make sure that the project work went smoothly, some guidelines were established to indicate what characterizes a good team member. These guidelines are:

- A team member is expected to show up on time to scheduled group meetings or give notice well in advance if the team member can't participate.
- A team member is expected to finish the tasks that have been assigned to the team member. If the team member lacks the capability to do so, the team should be notified so that the task can be solved within the time frame.
- A team member is expected to not be confrontational and put blame on other team members but should rather seek to resolve conflicts in a constructive way within the team.
- A team member is expected to make sure that the entire team understands the work that has been done.
- A team member is expected to participate in all supervision meetings.

- A team member is expected to be solution-oriented and not look for faults without contributing to solving it.
- A team member is expected to be respectful towards all members of the group.
- A team member is expected to stay up to date on the project's progress.

1.2 Goals & Objectives

The team has also set up goals and objectives so that the team can have a shared understanding of what is to be achieved in the project and the course. These goals and objectives are:

- Finish all assignments on time.
- Attend all supervision meetings.
- Learn practical applications of MDSE.
- Produce a project and report of high quality.
- Practice our presentation skills.
- Learn to work in cross-functional teams.
- Perform at the best of our capabilities, but be content with whatever grade we receive.

1.3 The Project Context

The system to be modeled is a platform, called ROVU, for a robot that will be monitored by non-technical operators. The system should display the status of the environment and is made up of multiple robots that all have:

- GPS sensors.
- Camera.
- Wheels.
- Networking device.

The system should allow robots to conduct missions. A mission is made up by points in an environment which the robot should visit in a specific sequence. An environment is built up of different areas and these areas are either physical, where physical boundaries restrict movement, or logical, e.g. a Wifi zone. These areas can be nested.

The system should provide interfaces that cater to different groups of people, e.g. visually impaired. Reward points should be presented on the platform and updated every 20 seconds. Reward points can be calculated in two ways:

- By the number of robots in a physical area.
- By the number of robots in a logical area.

Initially, reward points are calculated according to how many robots are in a physical area. At the in end of a 20 second period, if at least one robot is within a logical area, reward points should be the latter way. At the end of the next period, reward point calculation is switched back to the former way. In the last mission in the project, both of the procedures will be used.

As a mission is not constant, the platform must be able to handle change as a new mission can occur while another one is running or an operator can stop the program. A robot should also, if critical faults are detected, turn itself off.

Chapter 2

Domain Model and Use Case Diagram

For the first assignment, the group was tasked with producing a domain model and a use case diagram for the project. As with all assignments in this project, these models were iterated over as time progressed to make sure that they were in line with any changes made in the more detailed models. These two models have a high abstraction level and are modeled to visualize the main functionality and relationships within the system.

2.1 Domain Model

Below, in figure 2.1, is the domain model for the ROVU system. Following that is a description of the entities and the relationships between them to help clarify how the system is intended to operate.

The center of our domain model is the Controller. The Controller is responsible for connecting and updating every part of the system. The Controller is further divided into three controllers which each have different responsibilities and focus. They convert and send information to the main Controller. The Controller is also responsible for notifying updates to the View about information that is necessary for the user.

The EnviromentController contains areas that are built up together. Each area could be logical or physical. The EnviromentController is also responsible for giving permission or denial to robots entering the areas.

The RewardController is responsible for updating reward points every 20 seconds and saving the reward points in the RewardStorage. Having the reward points stored in

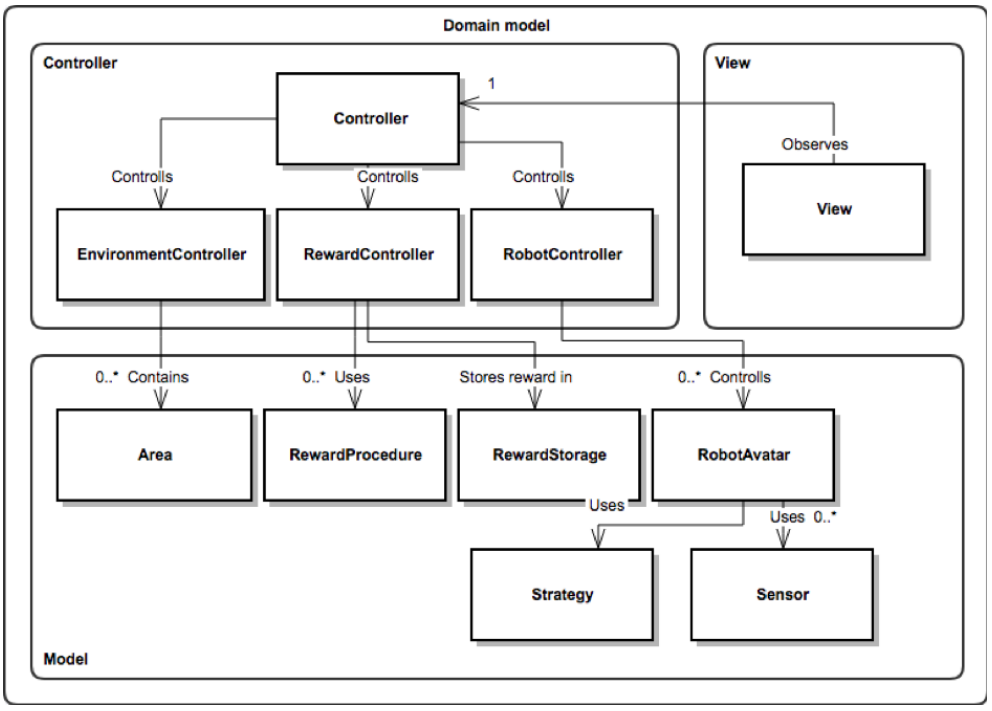

FIGURE 2.1: A domain model for the ROVU system.

the RewardStorage further decouples the logic from the data. The RobotController is responsible of telling the robot to move and to execute a mission.

Each robot has a strategy and the strategy contains the mission points and in what order they should be visited by the robot as well as how.

2.2 Use Case Diagram

Below, in figure 2.2, a Use Case diagram will be presented. After that, justification and description of selected actors and use cases will be presented.

2.2.1 Actors

The Operator is chosen as an actor as the operator is the one who the system was designed for in the first place. The operator is able to interact with the system through a view. The Operator is in this case the main user of the system and has mostly visual interaction possibilities.

The Robot is on the other side of the system. It is the actor who the system should model and it is a vital part of the system which is why it is chosen as an actor in this diagram.

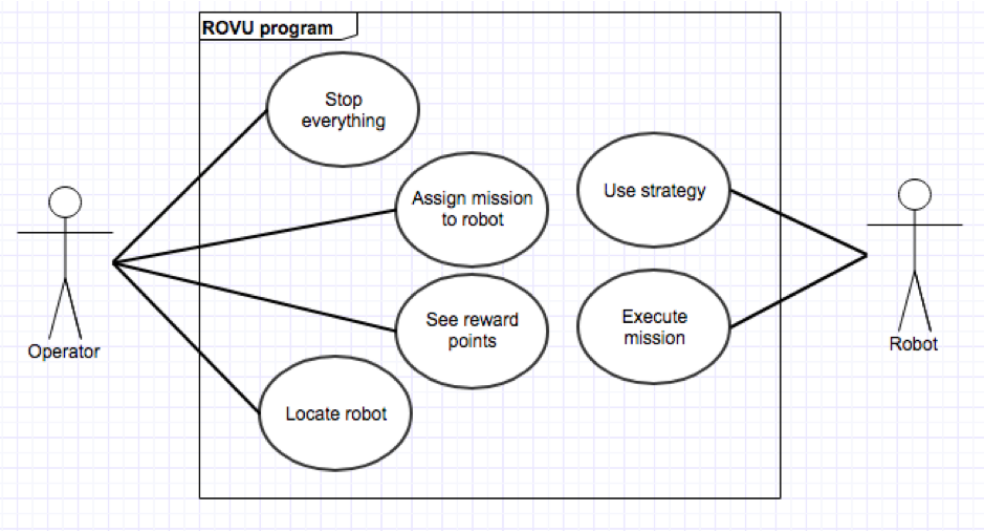

FIGURE 2.2: A Use Case Diagram for the ROVU robotic system.

2.2.2 Use Cases

This section will look at the chosen use cases and describe them, as well as justifying why they have been chosen.

Stop everything
An explicitly stated requirement is that an operator could, should the operator wish to do so, stop the entire system. This use case is chosen to fulfill this requirement.

Assigning mission to robot
A technical operator should be able to assign missions to the robot. Without missions the robots will not move and the assignments will not be able to be completed. To complete the assignments, this is essential to the project.

Locate robot
The operator should be able to retrieve information about where the robot is located. This is required so that the operator can supervise the robot and see that it actually completes the missions that has been assigned to it.

Execute mission
As mentioned earlier, missions are what are required for a robot to move about within a given environment. This system does not support robots walking around aimlessly, and only with a mission will they have the ability to move. This use case means that a robot will visit the points specified in a mission while taking any restrictions in performance due to strategy or system requirements.

Use a strategy
As mentioned in the previous use case, a robot might have conditions to performing

missions. This is modelled as a strategy that a robot uses. The strategy will dictate how the robot moves through the system. It is chosen as a use case because it will be important for a robot to accept a strategy and perform a mission while conforming to requirements specified in current assignments.

See reward points
Another core functionality of the system is that of reward points. Different reward points will be given by different areas and depending on the robots positions a tally of points will be added and stored in the program. The operator should be able to see these reward points the robots have collected.

2.3 Responsibility & Control

In figure 2.3, responsibility and control for each use case has been assigned to the group's members.

Use Case	Responsible	Controller
Execute mission	Andreas	Felix
Use strategy	Arvid	Mattias
Stop everything	Siavash	Chatchai
Assign a mission to robot	Chatchai	Siavash
Get reward points	Mattias	Arvid
Locate robot	Felix	Andreas

FIGURE 2.3: Division of responsibility and control.

Chapter 3

Component Diagram and Architecture

In this chapter, a component diagram will be presented. This diagram will not explain functionality but rather describe the components that execute the desired functionality. This chapter will also look at the chosen architectural styles and finish off with a report on contributions of each team member.

3.1 Component diagram

In figure 3.1, the component diagram is displayed.

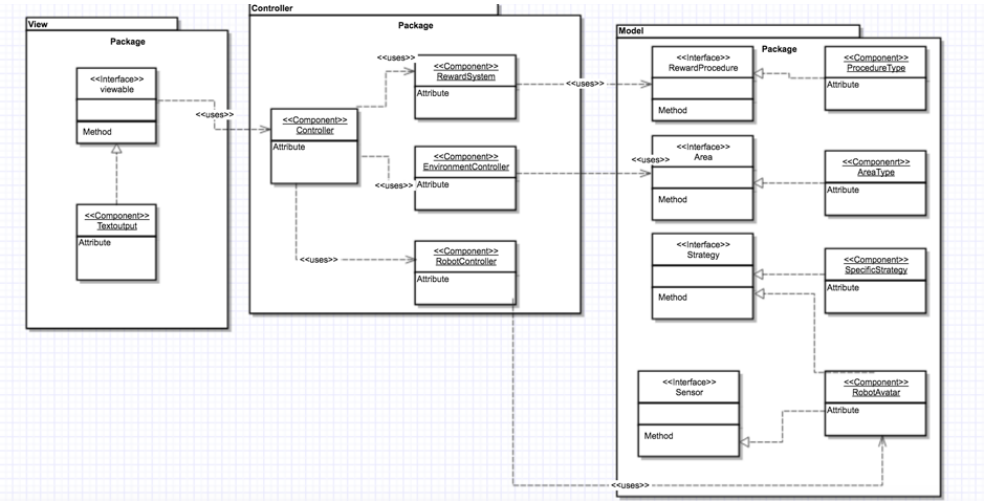

FIGURE 3.1: A component diagram for the system.

Below, a description of the components and interfaces will follow.

3.1.1 Components

Robot - *Component*

The robot component is responsible for the robot's behavior, managing its data, logic, and rules. This component communicates with RobotController to do the action that the Controller requests and also notifies the change to the controller that in turn updates the view.

RobotController - *Component*

Controller doesn't have direct access to our RobotAvatar and all the orders go through RobotController to the robot avatar.

TextOutput - *Component*

Its a component that is realized from our view interface.

AreaType - *Component*

The AreaType will be responsible for the area that robot is going to do its mission in.

ProcedureType - *Component*

The ProcedureType is one part of the system that keeps classes for the different procedures to calculate the reward points. This component will also keep track of when the reward points were last calculated so that the system knows when to do so again.

SpecificStrategy - *Component*

The SpecificStrategy is one part of the system that identifies which strategy should be used by the RobotAvatar to do its mission. In the project, only two strategies are employed. But as we shall see in later chapters, it can easily be extended to more strategies.

EnvironmentController - *Component*

The EnvironmentController is a component that specifies in which area our robot is going to do its mission. The EnvironmentController part of the system is responsible for the operations that concerns different areas, i.e. permissions to enter them, their sizes, and their current state in terms of occupation.

Controller - *Component*

This component keeps track of all the activities that are related to our robot avatar such as its reward, its movement and the area its going to do its mission in. So all the orders

are going through this component and all the information that is going to be displayed to our users is also going to be updated by this component. This component does not handle data, only the logic that manipulates it.

3.1.2 Interfaces

Viewable - *Interface*

The View interface is responsible for representing information to an operator(user) and forwarding the operator's order to the controller.

Sensor - *Interface*

This interface is created to record and handle the data that is collected through the robot's sensors.

RewardSystem - *Interface*

An interface for calculating the reward points acquired during the missions. By using positional data from the set of robots, it will calculate points dynamically.

Strategy - *Interface*

Strategy will handle the strategy that a robot takes to do its mission. So this interface is connected to both the robot avatar component and the strategy component.

Area - *Interface*

This interface is connected to our EnvironmentController component and AreaType component to manage which area a robot should be in.

RewardProcedure - *Interface*

This interface is connected to our RewardSystem component and ProcedureType component to keep track of reward points that a robot gathers and the type of reward points it is in particular.

3.1.3 Architecture

We chose to use the architectural design called Model-View-Controller(MVC). This choice was made because we wanted a clear separation of the logic, the data and the user interface(UI). The logic of the system is placed within the controller package while all data handling is done in the model package and finally the user interface, or view,

is located in the view package. This makes other good practices in programming a lot easier, like for example separation of concern which means that a component does not handle any unnecessary data or logic. It also makes the system more modular as you can for example add more models to the project without having to do large re-factoring to the rest of the components.

3.1.4 Team contributions

All team members were involved in updating the previous assignment according to the feedback given at the supervision. All team members were also part of discussing how this assignment should be handled. Below is a summary of the contribution of each team member.

Mattias did the first sketches of the component diagram and the remakes of the domain model and use case diagram. This made it easier for the team to continue the work as it gave us a starting point.

Arvid did the updates on the Use Case Diagram and worked with the justification of architectural decisions.

Siavash Was in charge of building the component diagram in Papyrus and did the updates on the Domain Diagram.

Felix worked on the descriptions as well as did final retouching of all parts of the assignment.

Chatchai also worked on description of components, interfaces, and operations and laid down specifics together with Felix and Andreas.

Andreas finalized the updates on the previous assignment, compiled the hand-in for this assignment and worked together with Chatchai and Felix to specify the component diagram in detail.

Chapter 4

Class Diagram & Code Implementation

This chapter will present and describe a class diagram of the ROVU system. From this model, code was generated using code generation functionality available for Papyrus. All the implemented code can be found in the package provided at the final hand-in and by visiting the Github repository at https://github.com/Alllle/TDA593-Group-16.

4.1 Class Diagram

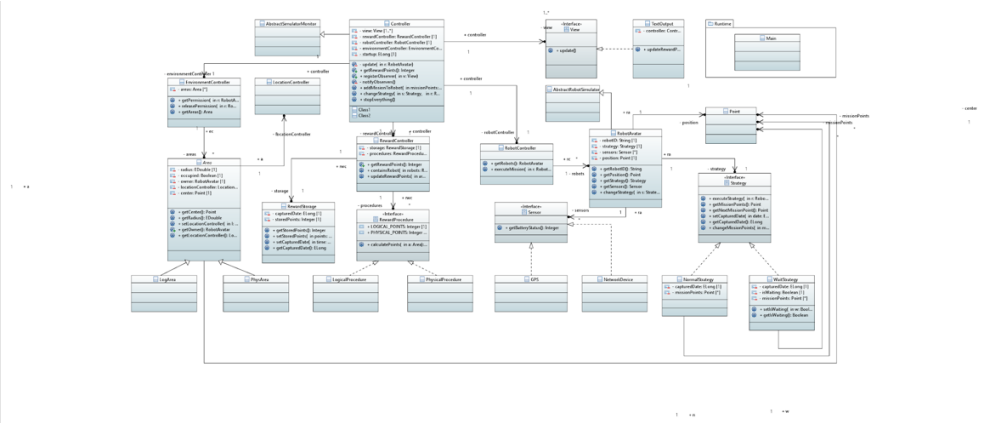

FIGURE 4.1: A class diagram representing the ROVU system.

The Controller class is the core of the model. This class is responsible for making sure that communication between all parts of the system is done in a good fashion. There are three additional sub-controllers all responsible for specific parts of the system, i.e. EnvironmentController, RewardController, and RobotController.

EnvironmentController is basically in control of a list of all the areas collected in a class (all the rooms together become an environment). Areas are divided in to two different types. The logical areas and the physical areas. Physical areas can represent rooms while Logical areas can represent zones such as Wifi or eating zones. The two kinds of areas have several attributes in common which can be seen in the diagram in Fig 4.1, which they inherit from the abstract class Area. The EnvironmentController controls the action of giving and releasing permission to enter a room depending on if a room is occupied or not which is needed when implementing the missions described in the assignment. In order to know if a room is occupied, the Area class has a LocationController, which is provided with the project, to control if the room is inhabited by another robot or not.

The reward calculation part of the system keeps classes for the different procedures to calculate the reward points. But it also includes a class that works as a storage for the current points. This class will also keep track of when the reward points were last calculated so that the system knows when to do so again, The main RewardController is the class that updates the reward points to the system.

The robot part of the system controls the robots themselves, what strategies they employ and the missions they perform. There is also a GPS and a Network device, which are realizations of the Sensor interface, from which battery status can be deducted. The robot part of the system is in charge of executing the missions but also provides information about robots whereabouts, so that checks for occupied rooms can be done in other parts of the system.

The view observes the changes done to the model in the system, and updates the elements in the view that have been changed in the model. In this diagram, only the interface realization TextOutput is presented. This is because the graphical representations are provided in the course and not created by the team. This design allows extension of the system with additional views that would seem fitting to someone developing the system further beyond the scopes of this course.

4.2 Design Patterns

This system makes use of the Observer pattern between the View and the Controller. In this case, the View subscribes as an observer to our Controller. That way, the MVC architecture is fulfilled by having the View updated by the Controller without the Controller having direct access to the View and information altered in the model can be viewed.

The system also uses the Strategy pattern both regarding strategies for the robot but also for the different reward point calculations. If additional strategies or reward point calculation methods would need to be added, that could be done effortlessly thanks to the implementation of this design pattern.

4.3 Team contributions

This section describes the contributions of all group members for the class diagram and code implementation assignment.

All - The team made a joint effort to build the Class Diagram. The team put Papyrus up on a big screen and went through all parts of the Class Diagram as a group and from there divided the work fork code implementation and generation.

Andreas - Made the very first sketch of the class diagram together with Siavash, Chatchai and Arvid. Andreas was then in charge of writing the description of the Class Diagram, Design Patterns and the Team Contribution as well as producing the assignment report.

Arvid - Made the very first sketch of the class diagram together with Siavash, Chatchai and Andreas. Arvid was then responsible for implementing the environment for the mission specified in the assignment description.

Chatchai - Made the very first sketch of the class diagram together with Siavash, Arvid and Andreas. Chatchai was then responsible for implementing the Robot class together with Siavash.

Felix - Felix did the first implementation of the Class Diagram into Papyrus before the team met and went through it as a group. Felix was then responsible for implementing the Mission class.

Mattias - Made changes to the initial sketch created by Andreas, Arvid, Chatchai and Siavash so that Felix could implement it into Papyrus. Mattias was then responsible for implementing the MissionController class and the Main class.

Siavash - Made the very first sketch of the class diagram together with Arvid, Chatchai and Andreas. Siavash was the in charge of implementing the Robot class together with Chatchai.

Chapter 5

State Machine

This chapter consists of state machines that are produced using the software Yakindu. The assignment consisted of three scenarios that each have their own separate state machine presented in this chapter.

5.1 First task: Synchronization

In this task, a provided skeleton of a state machine was to be further developed. The scenario modeled the behaviour of a receptionist in a hotel who should be able to check people in and out of rooms. Two rooms were introduced to the model and a room switch that simulated the behaviour of switching between rooms. The state machine for this scenario is presented in figure 5.1.

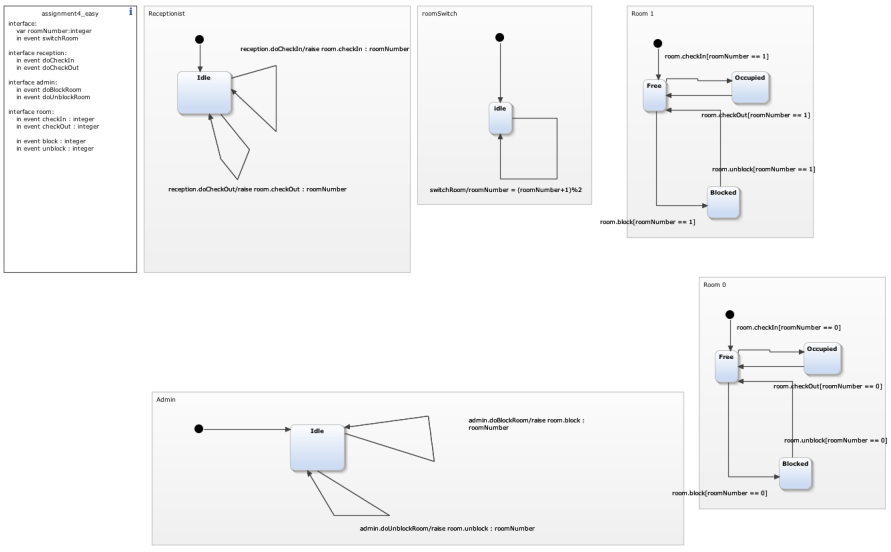

FIGURE 5.1: A state machine representing the first task.

Our first task is to continue from the pre-provided skeleton and add two different regions to simulate the status of two different rooms that could be triggered by the admin and receptionist. The added region simulate the status of free, occupied and blocked. The simulated rooms can be checked in to or blocked while they are free, but not blocked while they are occupied or the other way around.

In this task, two regions simulating the status of two rooms were added (Room0 and Room1). These rooms' statuses can switch from Free to Blocked, Free to Occupied and vice versa. Admin and receptionist can trigger the events and change the status of the room.

The implementation and the architecture of the model allows us to move from one state to another. It is impossible to go from occupied state to block state ,or vice versa, without passing the free state.

5.2 Second task: Operations

This task includes assigning operations that regions use to communicate with each other. The state machine for this task i presented in figure 5.2.

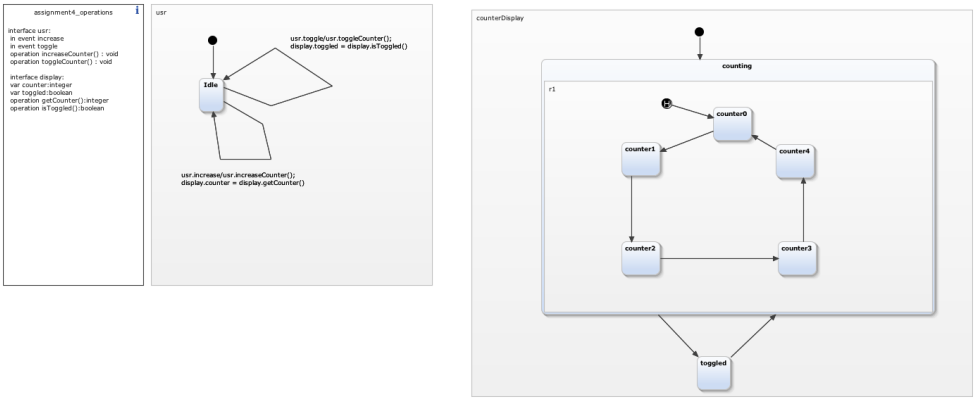

FIGURE 5.2: A state machine representing the second task.

The intention of this second task is to introduce the concept of operations in state machines and show that the counter works by using operations.

IncreaseCounter() operation in the *usr* interface increases the counter variable until it reaches five, and then it goes back to zero and gives us a range of zero to four. The display interface has the operation called *getCounter()* and this operation affects *counterDisplay* and *increaseCounter* affects it indirectly.

5.3 Third task: Formalizing a mission

This diagram has one state machine for each Robot and one for each Mission plus a Lock and a Timer. Robots can either be Idle or Wait(ing). The move event only affect the Mission if the Robot is Idle. The Wait state is triggered if the Robot move into a new Room and triggers back to Idle when > 2 Timer events have passed. For every robot there is a Mission where the Robot operates. For every Room there is a Lock. Transitions between Rooms set the position of the Robot to the new Room. The Lock results in that only one Robot can be in that Room at a time as the Robot checks if the Room is locked before entering. By having the Lock and by having Robots able to move within Rooms, the deadlock issue is removed.

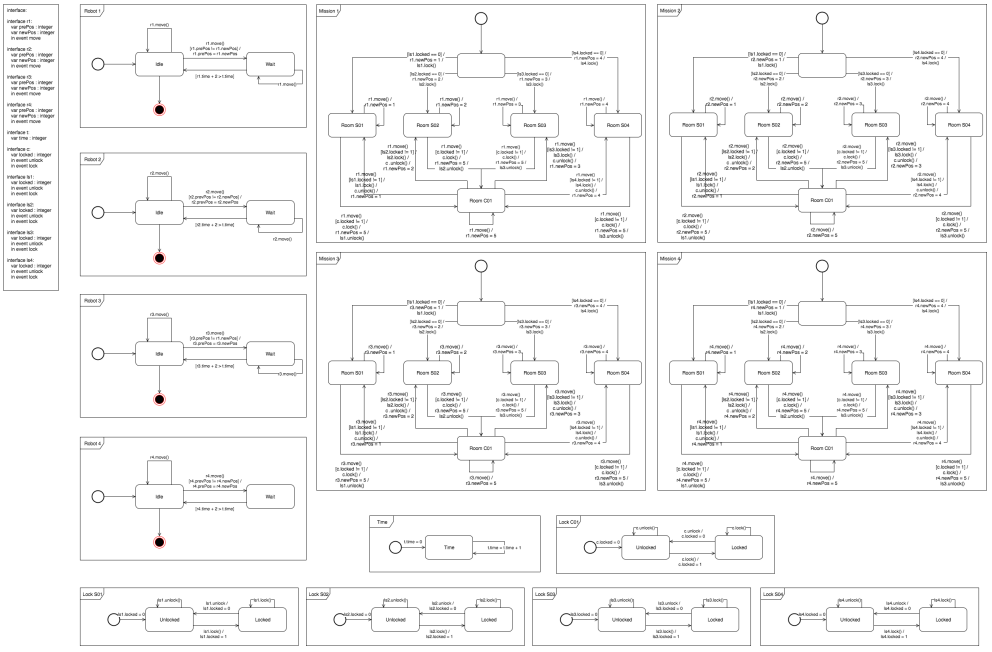

FIGURE 5.3: A state machine representing the third task.

Chapter 6

Changes in the project and Sequence Diagrams

This assignment included creating sequence diagrams for three different scenarios specified in the assignment description. It also included updating the implemented code and the class diagram according to new specifications for a mission.

6.1 Project update

The implemented code and the class diagram was updated according to specification. A description of the changes are presented here. Much of that which was excluded in the previous mission but was included in the project description was now expected to be implemented.

6.1.1 Reward system

The team now had to make sure that a proper reward system was in place to calculate and display the reward points in the system. The team implemented the functionality for reward point calculation as described in the project context which is based on the project description given out by the course admins.

6.1.2 Delayed movement

Another update was to make sure that the robots stopped when entering a new room. To make this work, the team interpreted it as enough for the robot to stop when it

reaches its mission point in a room, as it only has one mission point in a room and can stop when it reaches said point. The team implemented a variable that checked if the robot was in a waiting state. Once a waiting state was initiated, a variable with the current Date was created. When time reached this Date plus two seconds, the robot is taken out of its waiting state and is assigned its next mission point.

6.1.3 New environment

A new environment was also a condition for this assignment. Four surgery rooms were created with an adjacent consulting room. The mission also specified that there should be different colors on the walls, for instance red or blue. The team gave the rooms some beautiful colors and set up the environment according to the project description including the logical areas that were required as well. There was no instruction on where to put the logical areas exactly so they were placed at random.

6.2 Sequence diagrams

As mentioned, the assignment included modeling three sequence diagrams according to three different scenarios. These scenarios were:

- **Scenario 1**: Create a diagram that shows how a change of strategy is done in the system.
- **Scenario 2**: Create a diagram that shows a mission in an environment.
- **Scenario 3**: Draw a diagram that shows how reward points are calculated in the system.

The different scenarios are represented as sequence diagrams in figure 6.1, figure 6.2, and figure 6.3 respectively.

6.2.1 Scenario 1:

When the RobotController executes a mission on a RobotAvatar through the method executeMission(RobotAvatar r), the method calls executeStrategy(RobotAvatar r) on the robot. If the strategy is WaitStrategy and the robot has entered a new room the current time is saved and a boolean, isWaiting, is set as true. If 2000 milliseconds have passed since the current time was saved, the isWaiting boolean is set to false and the

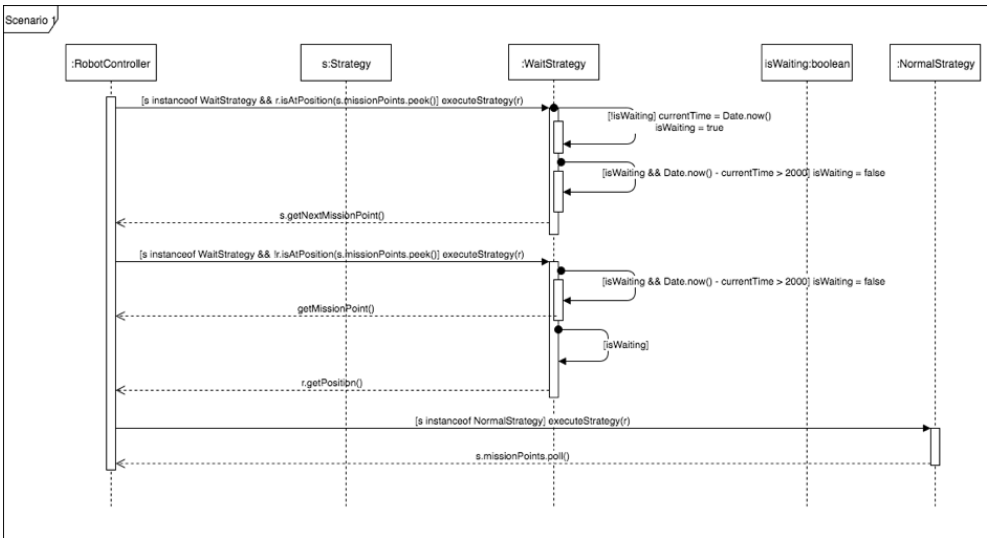

FIGURE 6.1: A sequence diagram for the first scenario.

next mission point is returned through getNextMissionPoint(). Otherwise the current position is returned through r.getPosition().

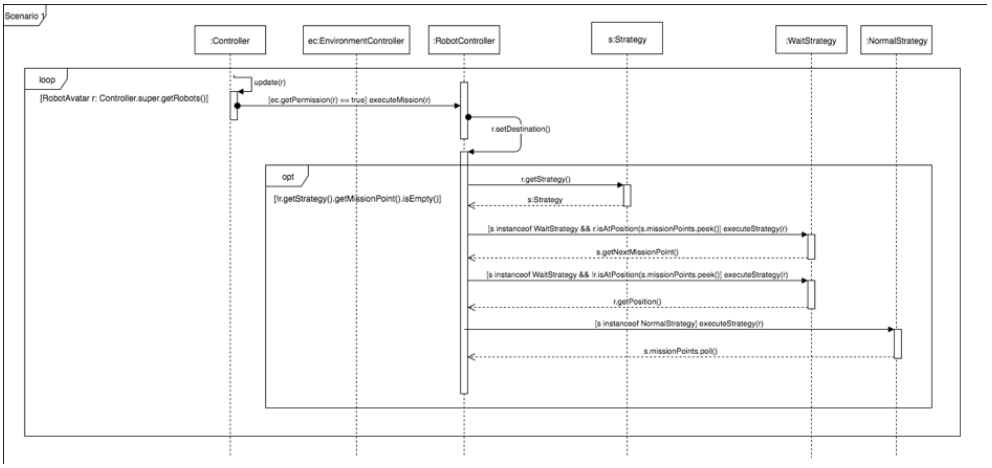

FIGURE 6.2: A sequence diagram for the second scenario.

6.2.2 Scenario 2:

For every RobotAvatar, r, in the AbstractSimulatorMonitor, an update(r) call is made in the Controller. If the robot is permitted to move, the method executeMission(r) is called. When setting the next destination/position of the robot the RobotController checks if the robot has a strategy. If it has a strategy and that strategy is WaitStrategy and the robot has entered a new room, it returns the position it's at right now; if it hasn't entered a new room it returns the next point in the mission. If the strategy is NormalStrategy, the robot moves to the next mission point.

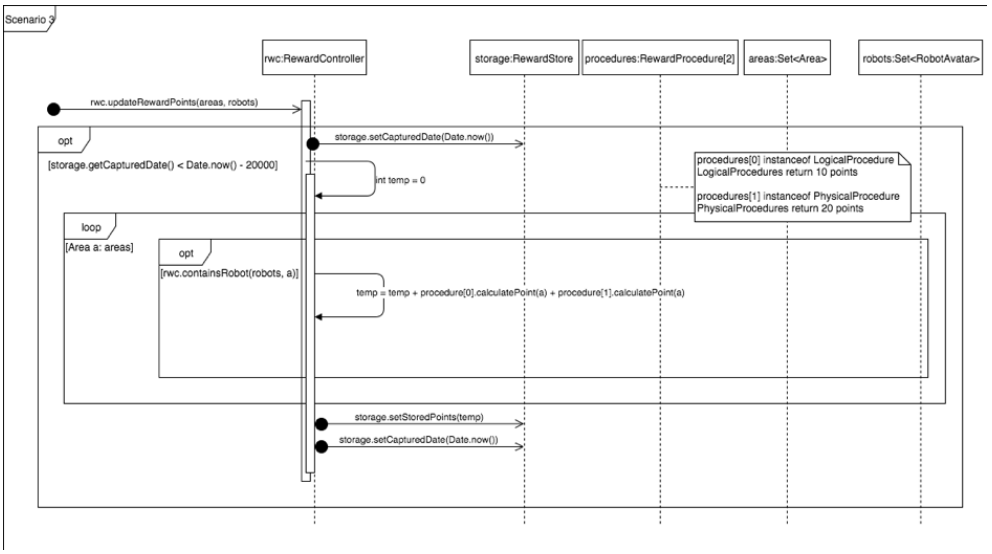

FIGURE 6.3: A sequence diagram for the third scenario.

6.2.3 Scenario 3:

As part of the run cycle in the AbstractSimulatorMonitor, a call to Controller.update (RobotAvatar r) is made. As part of this method, an update call for the reward computation logic is made by calling rwc.updateRewardPoints(areas, robots) where areas are a Set<Area> from the EnvironmentController and robots are a Set<RobotAvatar> from the RobotController. If there is a time stored in RewardStore < the current time - 20000ms, the current time will be stored in RewardStore instead. A temporary int, temp, will be created to hold the current reward points. For every Area in areas, the temp variable will be incremented by 10 points per robot, if the area is a logical area, and 20 points per robot, if the area is a physical area. This is governed by the RewardProcedures being either a LogicalProcedure or a PhysicalProcedure. When every Area in areas is covered the temp variable is stored as the current value in the RewardStore by calling storage.setStoredPoints(temp). In addition to this, the current time is stored within RewardStore by the call storage.setCapturedDate(Date.now()).

6.3 Team contributions

This section describes the contributions of all group members for updating the class diagram and code implementation as well as for creating the sequence diagrams.

Andreas - Organized the work, produced hand-in, implemented strategy functionality in code.

Arvid - Updated the environment.

Chatchai - Participated in updating code.

Felix - Updated reward system functionality and strategy implementation.

Mattias - Produced sequence diagrams and updated class diagram.

Siavash - Produced sequence diagram and participated in discussion of updating code.

Chapter 7

Conclusion

In this report, the group has demonstrated how they successfully produced all the required models and implemented code that performed the specified missions. While there have been specific contributions noted throughout the report, all group members were part of discussing and gaining understanding of all the produced models. The models can be used to describe the system. The choice of which model to present depends on the interest and knowledge of the recipient.

Throughout this course, the group has not only learned how to create different types of models, but also why they should produced in the first place.

This report successfully provided the reader with:

- **Domain model**.
- **Use case diagram**.
- **Component diagram**.
- **Class diagram**.
- **State machines**.
- **Sequence diagrams**.

This report has been a final product and basis for grading in the course DIT945/TDA593 *"Model Driven Software Development"* at Gothenburg University and Chalmers University of Technology.