Software design
Team project – Deliverable 3

Team number: 4
Team members:

Name	Student Nr.	E-mail
Basel Sammor	2608511	b.s.a.sammor@student.vu.nl
Yasin Aydın	2657725	yasinaydin@sabanciuniv.edu
Connor de Bont	2599228	connordebont@gmail.com
Faruk Simsekli	2657316	faruksimsekli.7@gmail.com

Table of Contents

1. The new requirement

Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli, Connor de Bont.

Our team received the following requirement change (C): ”Stop the mission - the system shall support the situation in which the operator pushes a “Stop-everything” button for immediately stopping the whole mission due to some severe circumstances.”

The implementation of our system before the requirement change did not include stop-everything button but we were already planning to add it. The implementation was also not complete so we could not run the simulator but the diagram was complete and most of the classes were implemented other than **CentralStation** and **Robot** class.

The first step was to complete the remaining implementation but before that, we controlled if there was any change we had to do on our diagrams. Firstly, there was no need to update the requirements in deliverable 1 after this requirement change. The **use case diagram** was already created in a way that the actor could end the mission so nothing had to be changed there. On the other hand, we had to add a stop-everything button to our interface on the **class diagram** but there was no need to add any new association according to the requirement change. The **object diagram** that we created did not need any change since the *robot* status “DISABLED” could be used to change *robots* statuses and we already had that specific case in our object diagram.

The requirement change was actually one of the things that we were planning so we already had an idea of how to implement that problem on our diagrams and code. The robot status “DISABLED” could be used to change any robot’s status so that when they are disabled they cannot perform anything. We could use that status when the button stop-everything is clicked so that all robots shall have the same status and they will all stop doing actions and stand still.

This requirement is also working like the requirement that we had before, which is the “70% rule”. This rule also requiring the system to stop when the covered area percentage is more than 70%. So, we decided to have a stop() function that can be used in both of these requirements. In that stop() function we only had to stop robots’ actions since the general algorithm continues with their actions. So, whenever they become disabled the game would stop because of that. We designed the algorithm in a way that when they are only continuing to move when are not “DISABLED”, otherwise they would go in functions that are containing the algorithm.

Since the requirement change is related to the action of the operator we had to change the interface. So after we implemented that in **UserInterface** class, the button had to call functions that can stop everything. As it is explained above the way to stop everything is through stopping robots. In our system, every command is given by the *central station* to robots. The *central station* has a function **sendCommand()** that can communicate with robots. So, in order to stop the robots, the operator had to communicate through the *central*

station. We already had the association between the **UserInterface** and **CentralStation** so the only thing we had to do was create a **stopMission()** in **CentralStation** and call it in **UserInterface**. So, we added the “Stop Mission” button which is actually “stop-everything” button that calls **stopMission()**. In **stopMission(),** central station sends “STOP_MISSION” *command* to every *robot* in the system. And when *robots* receive that command they immediately call the **stop()** method**,** which changes the *robot’s* status to “DISABLED” and stops the *robot’s* movement. And as stated before even if Simbad simulator calls the **performBehavior()** continuously, the if statement with the guard in this function shall not enter as running gets set to false by **stop()** and so the algorithms shall not repeat. So, everything in the system shall stop and the game shall end.

The changes that we have done according to the requirement change are below. The unrelated changes can be found in deliverable 2 with their descriptions.

Class Robot

Attributes

- **boolean running:** this static boolean acts as a guard and is checked to determine if the *robots* should continue with the mission or the mission is over.

Operations

- **void stop():** this function sets the *robot* status as disabled and stops everything to end the mission.

Associations

The **Robot** class is associated with the **Image** class. One *robot* can have many Images at the same time. *Robots* shall use Images when capturing photos of *boxes* and updating to the *central station’s* list. The navigability is undefined since in the code generation we faced issues with the creation of a *map* variable.

Class CentralStation

Operations

- **void stopMission():** this function performs the required tasks to end the mission and informing the *robots* to stop.
- **void saveImages():** this function shall be called at the end to save the images stored in the list on disk for later(after stop mission or mission ends normally).

Associations

The **CentralStation** is associated with the **Image** class. It will be using it to create an array list of Images to store the objects. The navigability is undefined since in the code generation we faced issues with the creation of a coordinate variable.

The **CentralStation** is associated with the **UserInterface** class. It needs to be able to communicate with the operator through the **UserInterface**.

Class UserInterface

Attributes

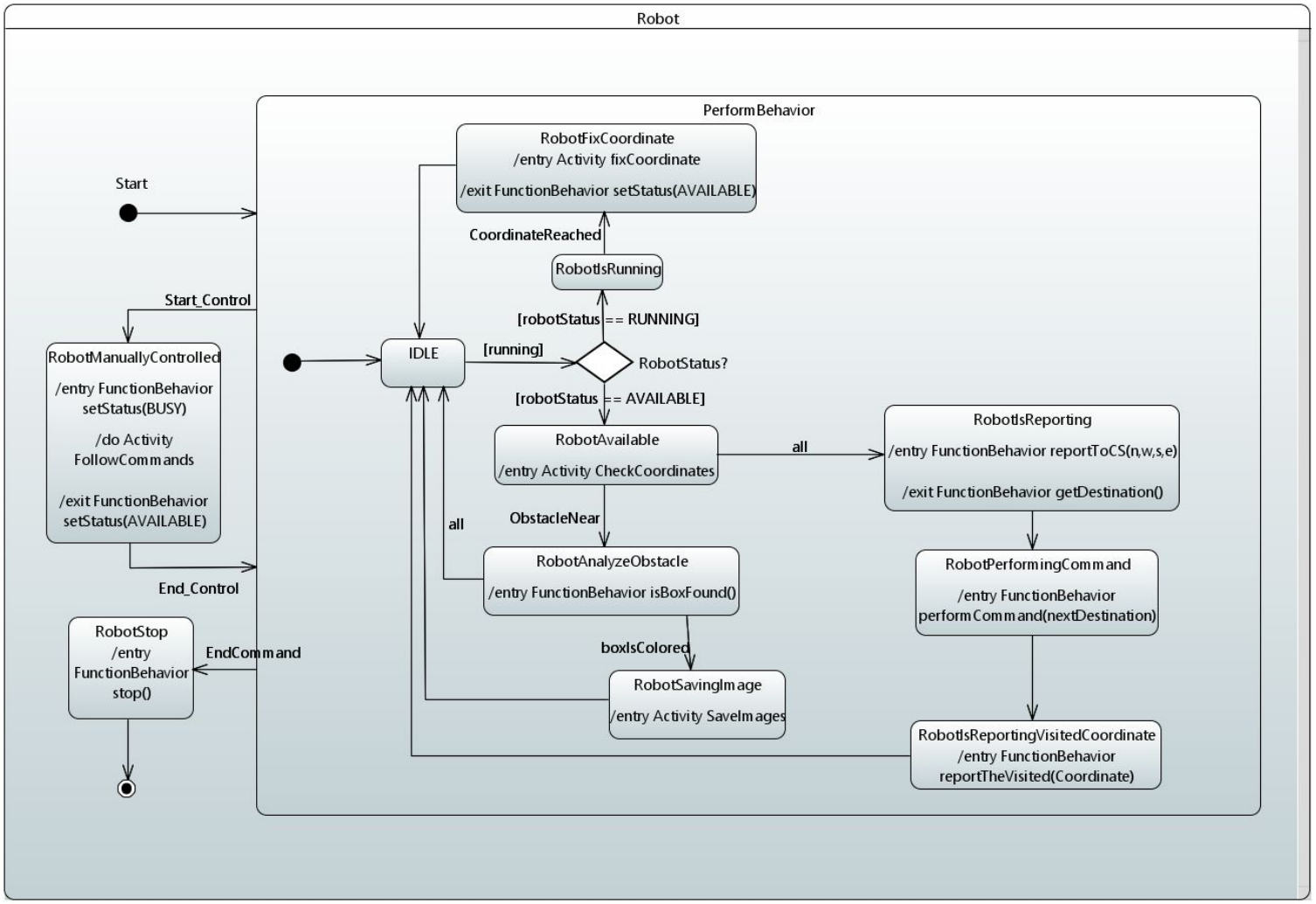
- **JButton stopMission:** this attribute will represent the go stop mission button in the manual control GUI.

Operations

- **void stopMission():** this function will stop the mission prematurely by the operator.

2. Objects internal behavior

Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli, Connor de Bont.
State Machine Diagram 1: Robot (Events are underlined.)



This diagram represents the states of the robot object during the run-time of the system.

At the start, robot directly goes to PerformBehavior state and becomes IDLE and at that point, it shall check “running” which is true when the mission is not over(game continues).

Next, the guard is checked, and if it's true the decision node is entered where again two guards are checked which are mutually exclusive. If the *robot's* status is AVAILABLE, the state of the *robot* shall become RobotAvailable and in the entry, it checks the coordinates around itself(North, West, South, East).

After that, if the *robot* detects any obstacles in those coordinates, it shall change the state to RobotAnalyzeObstacle. In this state, *robot* calls the function **isBoxFound(),** which turns to the coordinates that have an obstacle on them and captures an image and analyzes it whether the obstacle color is the same as the operator's choice. And then if the event boxIsColored doesn't happen, the *robot's* state becomes IDLE.

If the event boxIsColored happens, robot change state again to RobotSavingImage. When this state is entered, the images shall be saved to the directory that is hardcoded(/Images). And then the *robot's* state becomes IDLE.

After RobotAvailabe state if the ObstacleNear doesn't happen "all" ("any receive event") shall happen and robot's state shall become RobotIsReporting.

In the RobotIsReporting state, in the entry *robot* shall call **reportToCS()** and send checked *coordinates*. After that, the completion event occurs. Hence, before changing the state, the function(**getDestination()**, which returns a random empty coordinate around the robot) in the exit shall be executed.

When the *robot's* state changes to RobotPerformingCommand, the entry activity shall be performed. So, **performCommand(nextDestination)** shall be called and the parameter in this function comes from the **getDestination()**. At this point, the *robot* starts to move towards the nextDestination with some(0.5) velocity.

Then, the completion event occurs and the *robot's* state changes to RobotReportingVisitedCoordinate. As the *robot* enters this state it reports the last coordinate that it has been in, so that up can be updated on the *map*. And then again *robot's* state changes to IDLE(with the completion event).

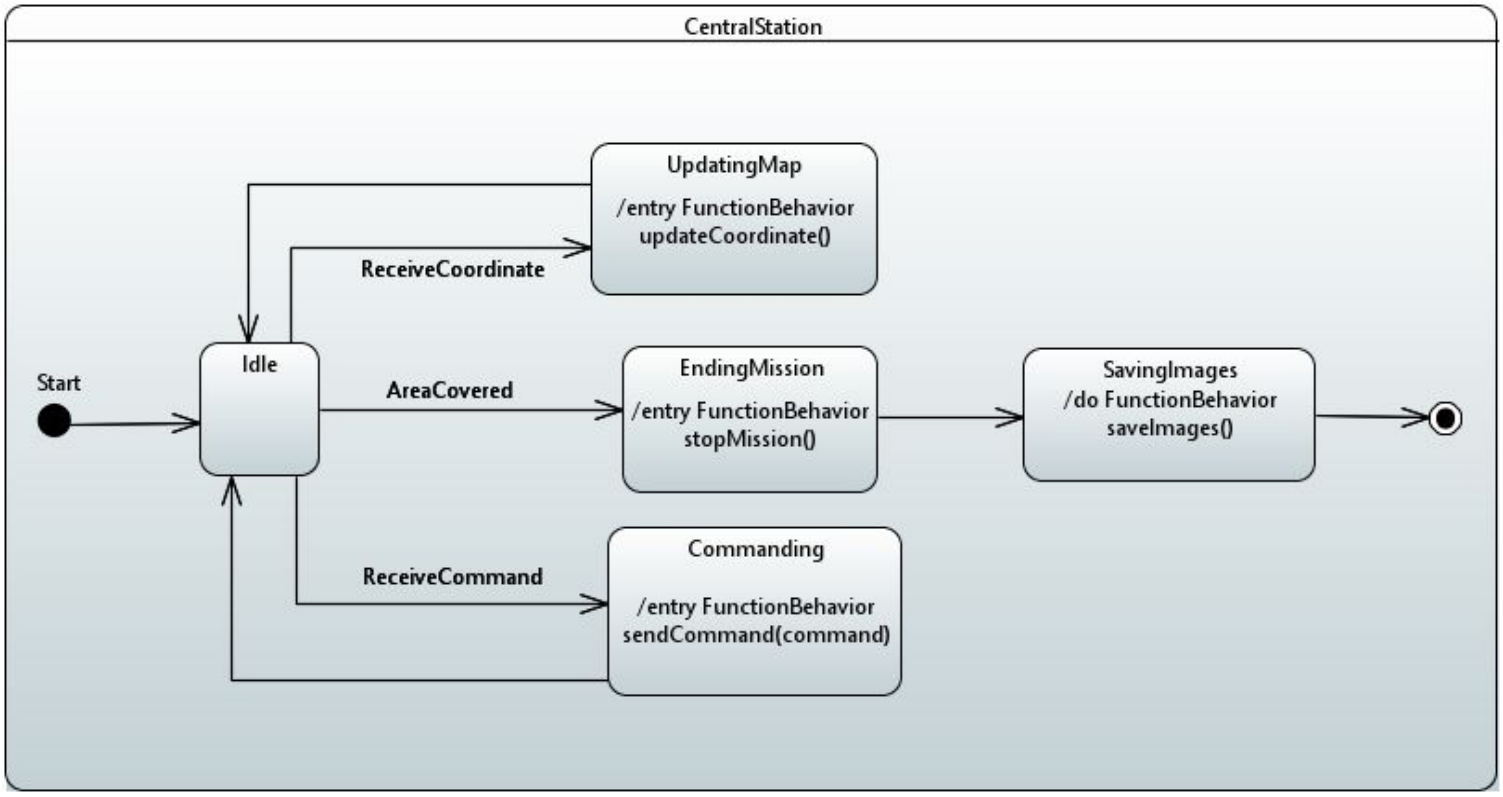
The previous part would happen if the robotStatus is "AVAILABLE" but after the **performCommand(nextDestination)** function *robot's* status turns to "RUNNING" since it starts running. So, the next time IDLE state is reached the robotStatus shall be RUNNING so that the guard [robotStatus == RUNNING] shall become true. And then, the *robot's* state shall change to RobotIsRunning. Here robot shall wait until the CoordinateReached happens and then it changes state to RobotfixCoordinate. As an entry activity robot fixes it's coordinate(this is explained in the implementation part). And then the status of the robot is changed to AVAILABLE so that everything shall repeat and robots can continue moving.

Also, there are events that can happen at any time when the *robot* is in the PerformBehavior. So, when the Start_Control happens *robot* state changes to RobotManuallyControlled. Entry activity sets the *robot* status to BUSY so that when the Simbad simulator calls **performBehavior()** robot won't react to that. As the robot stays in this state the operator can

control the robot with the given user interface. When the End_Control occurs as a exit activity the status of *robot* shall become “AVAILABLE” and then the *robot* goes to the PerformBehavior state again.

Also, at any given time EndCommand can happen. When this event happens the *robot’s* state changes to RobotStop in which **stop()** function shall be called to end the activities(final state).

State Machine Diagram 2: Central Station



This diagram represents the states of the *central station* object and how it transitions from one state to another during run-time.
Initially, the *central station* immediately enters the Idle state and waits for any of the following events to occur: ReceiveCoordinate, AreaCovered, and ReceiveCommand.

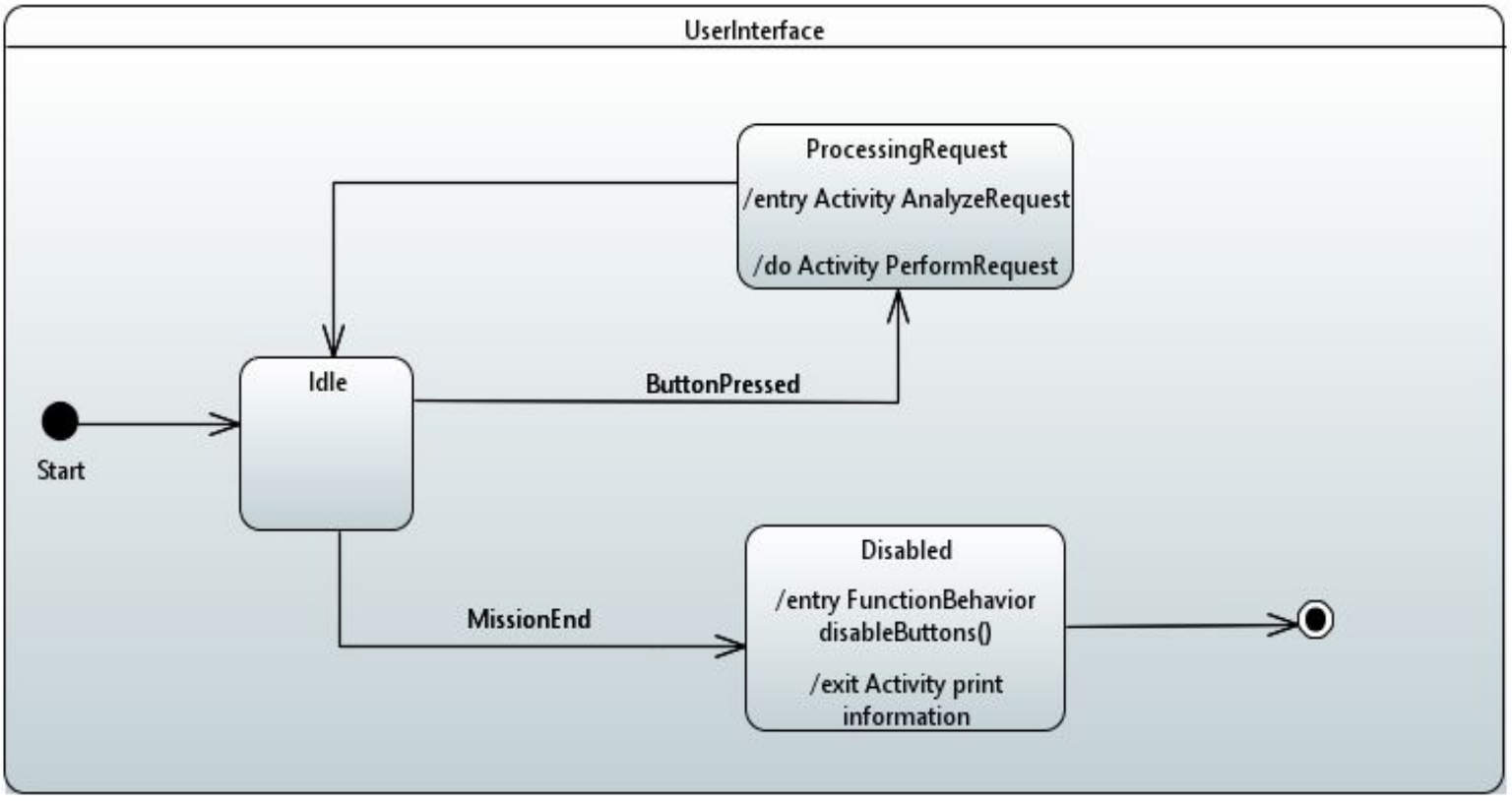
When ReceiveCoordinate occurs (from the Robot), the *central station* transitions into the UpdatingMap state and upon entry executes the **updateCoordinate()** function which updates its *map*.

After the state is done with updating the coordinates, it automatically exits the UpdatingMap state and enters the Idle state again.

When the ReceiveCommand happens (from the Operator using the UI), the *central station* enters the Commanding state and starts executing **sendCommand()** to the *robots* in our system. Again, when this is done with, the object transitions into the Idle state where it waits for events.

The last event that can cause a transition is when the area is covered (mission is over with 70% of the area is covered). When this event occurs, the *central station* enters the EndingMission state where it takes care of every detail required to end the mission and invokes the **stopMission()**. After that's done with, the *central station* enters the state called SavingImages where it takes care of the final task of the mission which is saving the images locally by using the **saveImages()** function. The central station then proceeds to the final state and remains there.

State Machine Diagram 3: UserInterface



This diagram represents the states of the *UserInterface* object and how it transitions from one state to another during run-time.

Since this object is simple, it only includes a minimal number of states that it can be in.

The object starts and immediately transitions into the idle state where it waits for an activity where a button is pressed by the Operator.

Whenever a ButtonPressed occurs, it transitions into the ProcessingRequest state to analyze the button on entry then performs the requested task. Once that's done with, the state transitions back to Idle to wait for more events.

The other event that can cause a transition is the MissionEnd event. This signifies that the mission is over and causes the UI to transition into the Disabled state where it executes the **disableButtons()** to not allow any further requests and prints the mission information on the screen.

Lastly, it transitions into its final state where it remains due to the mission ending.

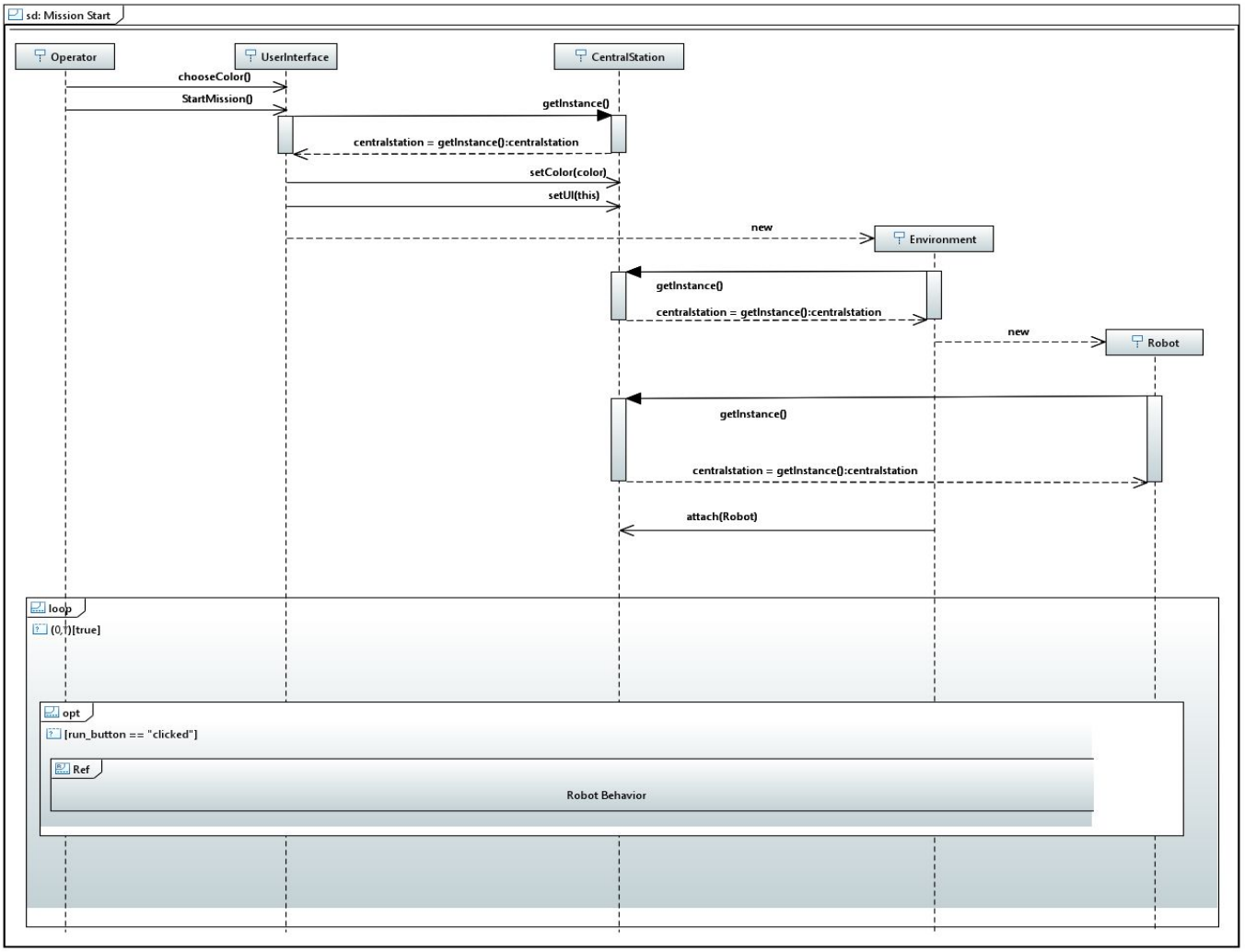
3. Interactions

Author(s): Basel Sammor, Connor de Bont, Yasin Aydın, Faruk Simsekli.

Disclaimer: When creating the sequence diagrams, we made sure to not include most of the internal behaviors of a specific object unless we made sure it adds clarity to the specific interaction sequence overall. Additionally, execution specifications in the diagrams were only added in synchronous scenarios or in self-messages in order to keep the diagrams clean as most of the calls are asynchronous.

Moreover, only one robot is shown in the sequence diagrams as having 3 additional identical *robots* in the diagrams doesn't add any extra information and just causes unnecessary clutter.

Sequence Diagram 1: Mission Start



The main members of the start mission interaction are the Operator (an actor that is outside the system. Should have been portrayed as a stick figure but papyrus doesn't allow that so we had to show him as a regular lifeline), *UserInterface, CentralStation, Environment,* and *Robot* objects.

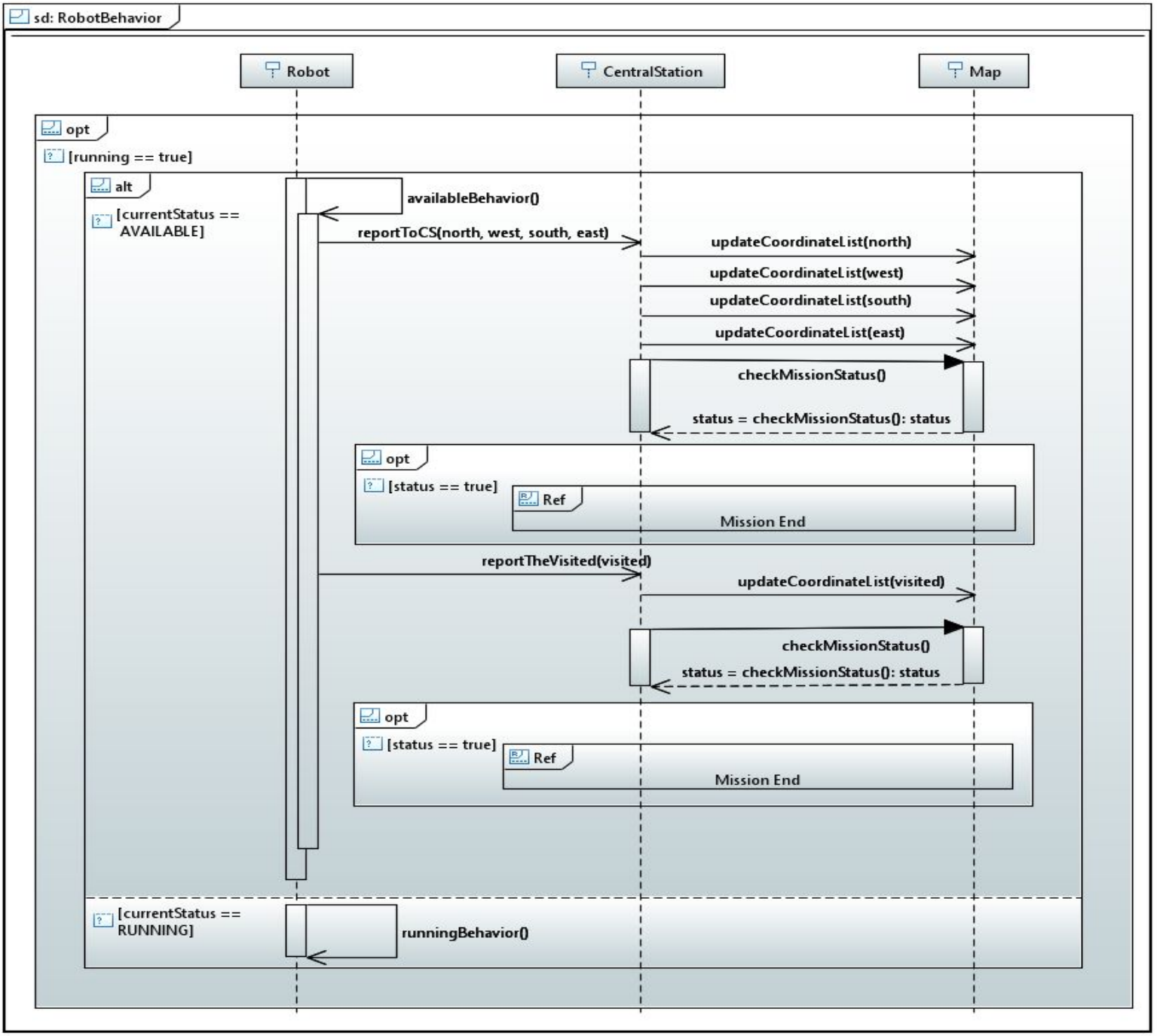
A ROBOSEARCH mission has to be started by an actor, who is the Operator here. The Operator can choose the box color and then initiate the mission by clicking the Start Mission button which calls the **startMission()** method as seen above.

Next, The *UserInterface* invokes **getInstance()** to get the singleton instance of the central station and proceeds to set the central station's box color that the operator chooses using **setColor()** while additionally setting the central station's *UI* object to itself using **setUI()**, this is done because, in later stages of the mission, the *central station* requires a reference to this very same instance in order to carry out some tasks such as ending the mission.

The *UserInterface* object then instantiates an Environment object after setting up the *centralstation*. Immediately again, the Environment gets the singleton instance of the central station for later communication. The environment then creates the boxes and walls in the environment and the robots. the boxes and walls were not shown for clarity's sake. Once a robot is instantiated again it gets the central station instance and the environment then attaches them to the Central Station through the attach(robot) method following the Observer design pattern.

A loop combined fragment follows the interaction mentioned above because due to how the Simbad simulator works, the system doesn't officially start until the "run" button is clicked. an infinite loop keeps running until the run button is clicked by the operator. whenever that's done, the mission officially begins and a referenced sequence diagram titled "Robot Behavior" continues the interaction between the objects of the system.

Sequence Diagram 2: Robot Behavior



The main members of this interaction sequence are a *Robot* object, the *central station* object, and a *map* object.

This sequence diagram portrays what occurs after the operator begins the mission and clicks the run button in order to proceed to this part. It focuses mainly on the interaction between the different objects and disregards internal behavior that is not needed to be shown.

First, an opt combined fragment which covers the entire sequence diagram is shown with the guard variable “running”. the operand block won't be entered unless the running == true. this is used in order to make sure that if a mission ends and running is set as false, the robot behavior interaction sequence won't occur as we want the mission to stop.

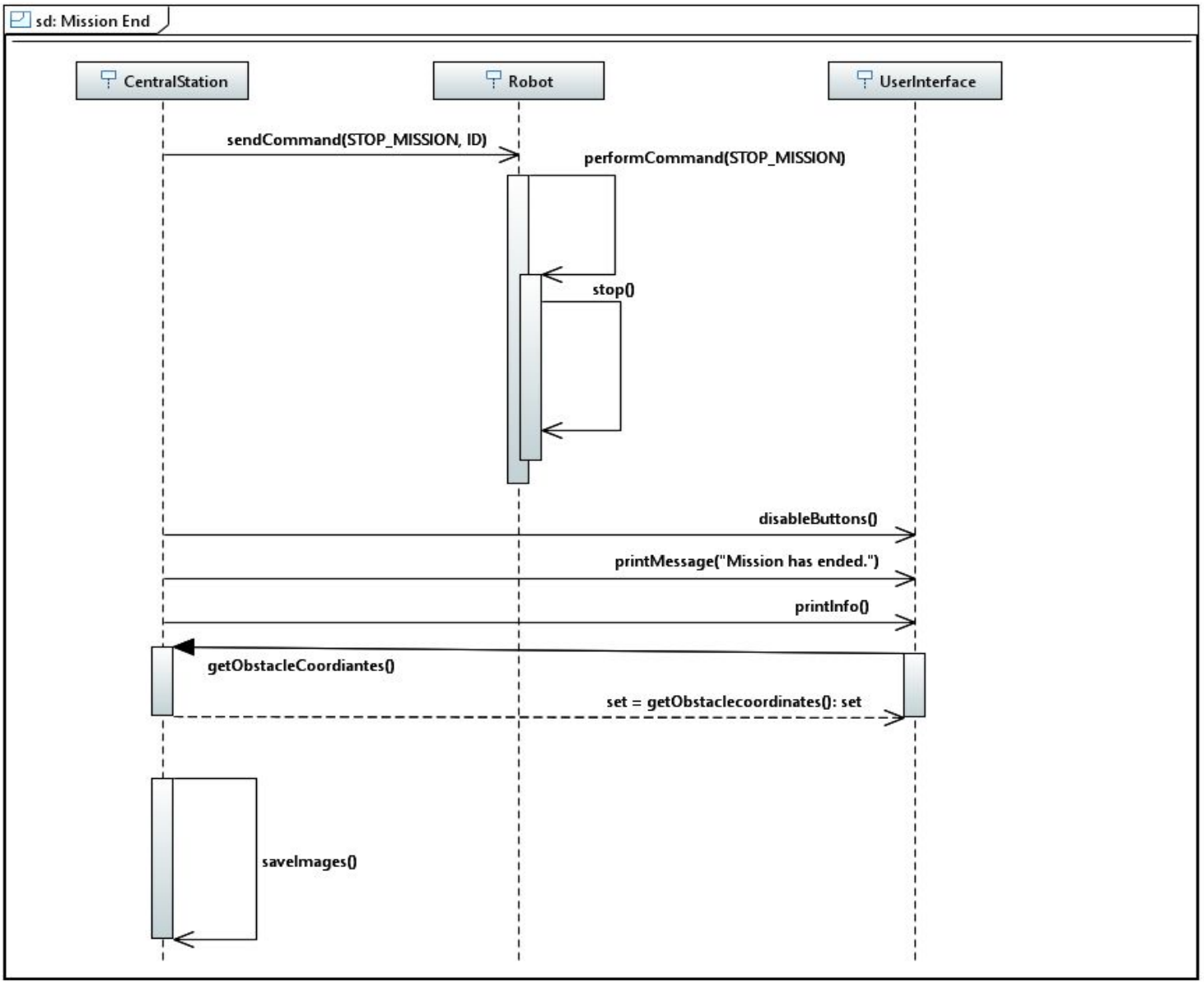
Next, another combined fragment can be seen, this time an alt fragment is used in order to make the robots follow different behaviors depending on the currentStatus of the robot. if it equals “AVAILABLE” then the robot will be executing **availableBehavior()** and communicating with other objects to pursue the mission goals. On the other hand, if it is “RUNNING” then it won't be communicating with any other objects but only executing the **runningBehavior()** method internally. This was added for clarity.

When the top operand is executing, the robot will internally execute the **availableBehavior()** method as shown. During the execution, the robot first reports the coordinates in its four directions to the *central station* by calling the **reportToCS()** method with the appropriate parameters. This, in turn, causes the central station to send an **updateCoordinate()** four times for each direction to the map object to update the status of the four coordinates in the map. Once that’s over, the central station sends a synchronous message to the map object and waits for a reply before it continues. The map object internally checks if the area is 70% covered or more and returns a boolean value.

Another opt combined fragment can be seen when **checkMissionStatus()** is called. this ensures that if the map returns the value “true”, the *central station* takes the appropriate action of ending the mission by executing the referenced End Mission sequence.

If not, then-then execution continues and **reportTheVisited()** is called which then follows a similar sequence of interactions due to how our algorithm works by first updating the one visited coordinate instead of four and then waiting for a reply from **checkMissionStatus()**.

Sequence Diagram 3: Mission End



When the conditions for a mission ending are met, the mission end sequence starts.
This only shows the case where a mission succeeds (covering 70% of the area) and doesn't include the premature ending that the Operator can perform.

The interaction partners involved in this sequence are the **CentralStation** class, the **Robot** class, and the **UserInterface** class.

The sequence starts with the **CentralStation** calling the **sendCommand(STOP_MISSION, ID)** with the corresponding robot id to all *robots*.

The robot then receives the command and performs a self-message with the command as a parameter in its **performCommand()** function. In this function, the parameter is checked and since it is equal to “STOP_MISSION”, another self-message **stop()** is called as shown in the diagram inside the robot. This sets its current status to “DISABLED”, and translational and rotational speed to zero. Setting its status to disabled will result in the robot not being able to perform the *central station* commands and ensures that the mission ends.

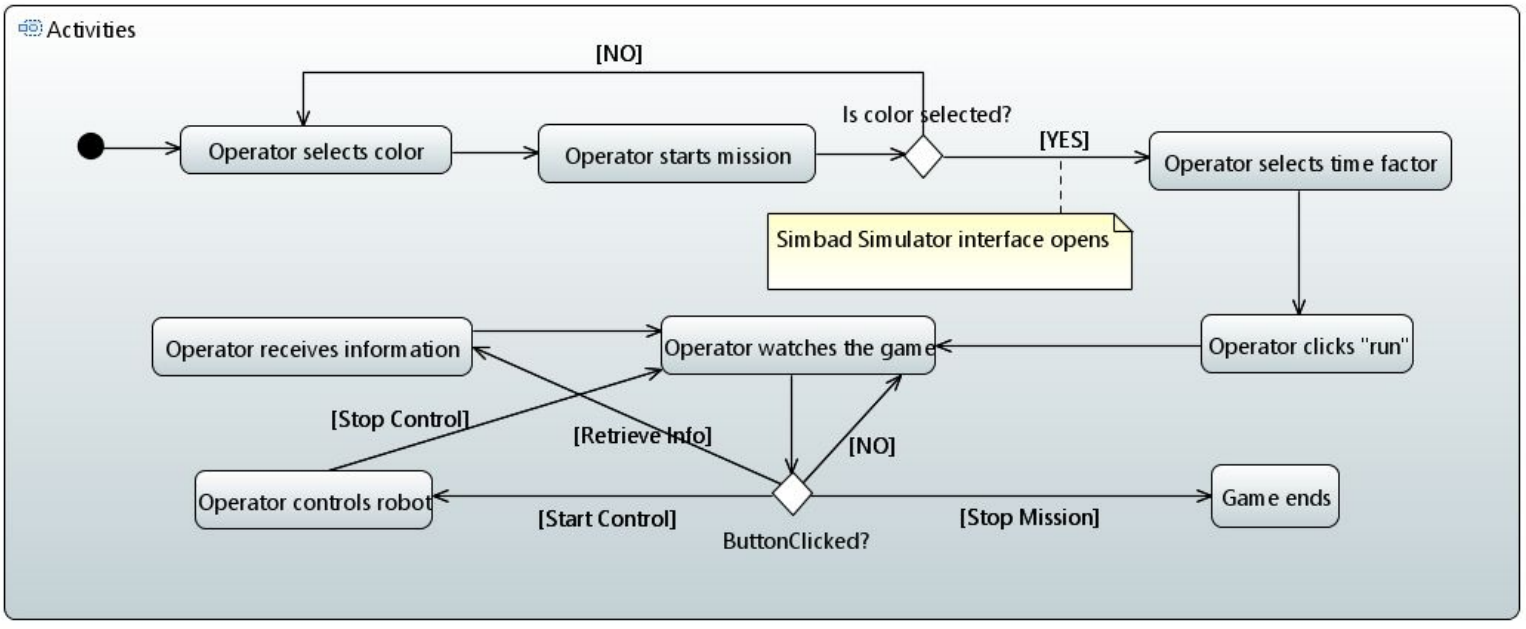
Meanwhile, the *central station* starts to inform the operator that the mission ended through the *UserInterface* object. It starts by calling the User Interfaces’ disableButton function which disables most of the UI in order to not allow the operator to issue calls that should not be issued at this stage. Next, it calls **printMessage()** with the “Mission has ended.” parameter to inform the operator clearly of the mission ending. Afterward, it calls **printInfo()** to show the coordinates of the boxes that were found in the area that matched the color chosen by the operator at the Start Mission sequence.
After that, the *UserInterface* in turn sends a synchronous message called **getObstacleCoordinates()** in order to get the set of the obstacle coordinates that the central station holds in order to print it to the operator.

Finally, the *central station* performs its **saveImages()** function internally, which saves the images and their respective coordinates locally on disk for later lookup. this marks the end of the mission.

4. Activity Flows

Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli.

Activity Diagram 1: Operator Activities



This activity diagram represents the **UserInterface** class and the actions that the operator can perform. As it is clear on the diagram the operator cannot start the game without selecting the color(of the box). After that Operator can choose the time factor from the opened Simbad simulator interface(for optimal results the time factor should be 10.0). When the operator watched the game he/she can press buttons on the control interface. If the “Retrieve Info” button is pressed the operator shall receive information about the boxes that are found. If the “Start Control“ button is pressed, the operator controls the robots by using the buttons on the control interface. If the “Stop Mission” button is pressed, the game shall end without any problem and the system shall stop.

Also, the buttons shall be activated and deactivated during the activity change so that when the mission ends the operator cannot control the robots anymore. And the operator shall not control the robot if he/she didn’t select the robot id.

5. Implementation remarks

Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli, Connor de Bont.

Main class: The main class of our program is the class that calls **UserInterface.java** to allow the operator to configure and control of the mission starting which will then create all necessary objects and setup every object that is needed in the simulation.

Environment class: This is the class that we chose to use as a generator of every physical object on the map, which are consists of *robots, boxes, walls.* The environment class is like a factory but hardcoded since at the beginning we decided to have a fixed map and fixed robot number. Also, box locations were fixed since the information about their location is never used in any implementation part(They are not there until a robot discovers them).

CentrolStation class: Central station is the singleton of the system so there is only one of it. *Robot* objects are using the *central station* to update the *map*, get notified with the observer pattern that has been used. In order to do that we had to create a *central station* reference in each *robot* object. Robots are controlled by the *central station*. The operator shall use the *central station* through the user interface to command *robots* to change directions, retrieve information… To do that we created an interface that helps the operator to control any *robot* he/she wants.

To run the program/simulator, running **main.java** is enough but after simulation opens the operator must choose the color that he/she wants *robots* to search from the ComboBox provided on the interface. We didn’t add environment size values that can be picked by the operator since the 25x25 was the fixed size.

UML models to code Strategy

The code has been generated by the UML class diagram using Papyrus’s automatic code generator function. The first generated code only contained the attributes and the methods that were added during the creation of the class diagram. After fixing every issue in deliverable 2, the only remaining part was to implement the system as planned. During the

implementation, we realized that more functions and attributes were needed. So, we added and implemented them in a way that shall make system work as we would like. On the other hand, we realized some attributes and functions were not required or could be improved upon so they were either removed or changed.

Problems and key solutions

For the algorithm, we created our own, which basically makes robots choose not visited coordinates whenever it is possible. In order to achieve that we had to find a way to store and communicate with the shared map. We solved that with map class and central station class which are working together to update the map(2D array) and notify robots afterward. So, after fixing this issue we had to find a way to make direct robots to the “EMPTY” coordinates rather than the “VISITED” ones. We solved that by using an updated map and checking if there are any “EMPTY” coordinates near the robot and directing them if there is one. If not robot can choose any coordinate that is not “OBSTACLE” and tries to get away from the state that it is stuck(surrounded by “VISITED” coordinates). Eventually, the robot shall move to a position where it can find “EMPTY” coordinates and continue with the same algorithm that searches for empty coordinates(the reason behind this idea is to cover map as soon as possible).

The Simbad simulator did not have any class methods that we could use for our algorithm to work. The algorithm consists of using coordinates like a grid point and store them in a 2D array. In order to do that we needed integer values and we could only achieve that if robots move from one integer coordinates to another. For instance, from (1,1) to (1,2). Although it seems like this is an easy job there was no way to stop robots when that reach exactly (1,2) since the coordinate updater function was not updating in real time and the way double values work. For instance, in one update coordinate can be (1, 1.9995) and then in the next one, it can become (1, 2.005). This was the hardest problem that we faced and since the Simbad simulator did not have any functionality other then **moveToPosition()** that helps us. We had to use the teleportation function in SimpleAgent.java. In order to limit this and make it a more realistic case, we set an error rate of 0.05.

In the beginning, we implemented *robots* and their *cameras* in a way that they would capture and save pictures after analyzing it. And in those functions, we had **isBoxFound()**, which analyzes if the *box* is in the captured image, that was creating an issue since it can take pictures from far and detect the colored box there. So, we had to change the strategy to take and save pictures if and only if there is an obstacle near(maximum 0.71 meters away) and the color of the *box* is the same as asked. So, every time robot moves to a position it checks if there is a box 0.71 meter away and if there is one it turns that way and saves the picture after **isBoxFound()** turns true.

Another issue we faced had to do with the fact that **performBehavior()** is called 20 times a second and it meant that the *robot* would rotate faster than the camera had time to update its view. we managed to fix that issue with using a separate counter as a guard that did rotation or image capturing depending when it reached a certain threshold. This made sure that a

robot will not invoke any other part of the algorithm until the robot has rotated and analyzed the directions with obstacles.

The provided Simbad simulator works with the class Vector 3d from the Java 3D API as coordinates. The way of the coordinate system was really problematic since we wanted to store coordinates in a 2D array which starts with [0][0] coordinate and goes to [25][25], which is map size so for every coordinate there is one status(the status of the coordinate). To fix that issue we used Adapter pattern and implemented in a way that every time we use the coordinate system of the Simbad simulator, the code converts it to X- and Y- axis form. With this conversion, it was also much easier to control *robots* with non-negative coordinates.

Other Remarks
While analyzing images the colors were not exactly blue or red or whatever color we wanted so we had to include other ranges of colors into the analyze function since the lighting and the position of the robot affects the color and the range of the color.
(Use maximum 10.0 time factor for the simulator speed!)

Some times the saved images in the /Images file are darker than it should be because of Simbad simulator's lighting on the map. The **analyzeImage()** detects the color without any problems.

Video
Video link: https://youtu.be/ItTotCG_GIs
In this video, you will see that robots are moving meter by meter and they generally don't visit the same place twice since the algorithm works that way. Also, you can see that robots are taking pictures while they are near an object but they are only saved at the end of the mission by the *central station*. These saved images' names consist of coordinates that the box has been found at.
When the 70% is reached or the operator clicks "Stop Mission" button game shall stop every action on the simulator and stop everything accordingly. In the video, the simulation stops because of the 70% rule.

6. References

For the state machine diagrams:
https://www.altova.com/manual/UModel/umodelbasic/index.html?umcode_generation_from_state_mac.htm
https://www.lucidchart.com/pages/uml-state-machine-diagram
https://sparxsystems.com/resources/tutorials/uml2/state-diagram.html
For the activity diagram:
https://stackoverflow.com/questions/22178978/java-colour-detection
https://www.tutorialspoint.com/uml/uml_activity_diagram.htm
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/
For the user interface:
http://www.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html

https://www.thoughtco.com/example-java-code-for-building-a-simple-gui-application-2034066