Multi-Robot Collaboration

Multiple Robots Collaborating Within a Shared Workspace

Master's Thesis in Computer Science

Mads Benjamin Fjeld

May 14, 2019
Halden, Norway

Østfold University College

www.hiof.no

Abstract

This thesis seeks to answer two things: How can RoboDk with its C# API be effectively used to enable the collaboration of two robotic arms, and how can different forms of collaboration be classified based on the requirements of the system to accomplish the given collaborative scenario. The challenges of having robots collaborate within the same workspace are presented, and different solutions are discussed. Technical challenges regarding RoboDk, the API and concurrent control and monitoring of the robots, as well as collision avoidance, are evaluated, and different possible solutions are presented. A method for classifying different forms of collaborative scenarios is proposed, where the scenarios are broken down into the capabilities, or "skills" required for the system to accomplish the scenario. Two different strategies for validating planned solutions to the collaborative scenarios are implemented in a "proof of concept" windows forms application that utilizes RoboDk and its API to control the robots and simulate solutions. Three different strategies for selecting the fastest solution to a scenario is also implemented. Finally, the application is successfully tested, both for simulation and against the physical robots, and the benefits of the classification of collaborative scenarios based on required skills are discussed.

Keywords: Robot Collaboration, Multi-robot, RoboDk, C#, API, .NET, Windows Forms, Industrial Robots, Robotic Arms, Collaboration, Collaboration Scenarios, Concurrent Control, Collision Avoidance, Simulation

Acknowledgments

I want to thank my academic supervisor, Professor Øystein Haugen for all of his valuable guidance and encouragement throughout my time writing this thesis.

I would also like to thank my friends and family for their enduring support and making life at Høgskolen i Østfold enjoyable during the course of my education. Mainly I would like to express my gratitude towards Huimi Chen for her constant love and support, as well as Martin Gunnersen for providing fruitful discussion and assistance regarding the robotics lab.

Contents

Abstract i

Acknowledgments iii

List of Figures viii

List of Tables ix

1 Introduction 1
- 1.1 Background and Motivation . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Method . . . 3
- 1.4 Report Outline . . . 3

2 Analysis 5
- 2.1 Research Topic . . . 5
- 2.2 Related Work . . . 5

3 Design 9
- 3.1 Environment . . . 9
- 3.2 Concurrent Operations . . . 10
- 3.3 Collision Avoidance . . . 10
- 3.4 Execution Plans . . . 11
- 3.5 Execution . . . 12
- 3.6 Collaborative Scenarios . . . 13
- 3.7 Categorizing Collaborative Problems and Their Solutions . . . 19

4 Implementation 23
- 4.1 RoboDk . . . 23
- 4.2 Interface With RoboDk's API . . . 25
- 4.3 Concurrent Movement . . . 26
- 4.4 Collision Monitoring and Avoidance . . . 32
- 4.5 Intelligent Programming . . . 36
- 4.6 Collaboration Scenarios . . . 37
- 4.7 Responsive Programming . . . 42

5 Testing **43**
5.1 Simulation 43
5.2 Testing On the Physical Robots 56

6 Discussion **65**
6.1 The Application 65
6.2 Different Forms of Collaboration 71
6.3 What Could Have Been Done Differently and Future Work 72

7 Conclusion **73**

Bibliography **76**

List of Figures

1.1 The two robotic arms. 2
1.2 The OMRON LD60. 2

3.1 RoboDk simulation of both robots in a theoretical configuration allowing for collaboration between the robots. 9

4.1 The RoboDk station to be loaded at launch 23
4.2 The setParentStatic method in the python API. 24
4.3 The setParentStatic method imported to the C# API. 24
4.4 The interface to interact with RoboDk through the API. 27
4.5 The list of tracked bricks in the interface. 28
4.6 The NewLink method for multi-thread support. 28
4.7 Creating and executing tasks for moving the robots concurrently. 29
4.8 The robots moving concurrently with C# tasks - The tasks are started at the same time. 30
4.9 The robots moving concurrently with C# tasks - The KUKA moves faster than the UR robot, KUKA has reached the first destination, UR is still moving towards it. 31
4.10 The default collision map for the RoboDk station. 32
4.11 Creating and executing RoboDk programs for moving the robots concurrently. 35
4.12 Scoring a RoboDk program based of if a collision occurred and execution time after execution. 35
4.13 The algorithm for delivering a brick. 38
4.14 The algorithm for finding the best solution for building a wall (page 1 of 2). 39
4.15 The algorithm for finding the best solution for building a wall (page 2 of 2). 40
4.16 The algorithm for building a wall. This is an extension of the "Simulate solution and determine score" process used in figure 4.15. 41

5.1 Testing the delivery scenario within the simulation. - Starting the test. . . . 44
5.2 Testing the delivery scenario within the simulation. - KUKA places the first brick in the critical region before returning to its home position. 45
5.3 Testing the delivery scenario within the simulation. - After the KUKA robot has returned to its home position, the UR robot picks up the brick from the critical region. 46
5.4 Testing the delivery scenario within the simulation. - The UR robot places the first brick on the designated position. 47

5.5 Testing the delivery scenario within the simulation. - After the UR robot has returned to the home position, the KUKA robot places the second brick within the critical region. 48
5.6 Testing the delivery scenario within the simulation. - After the KUKA robot has returned to its home position, the UR robot picks up the brick from the critical region and places it on the first brick on the destination position before returning home. 49
5.7 Testing the build wall scenario within the simulation. - Starting the test. . 51
5.8 Testing the build wall scenario within the simulation. - When a collision is detected in the simulation, the simulation is stopped and the next solution is tried. 52
5.9 Testing the build wall scenario within the simulation. - After all potential solutions were tried, the fastest one was selected as the best one. (Green status text: "Fastest solution found! - estimated execution time 4.69 sec") . 53
5.10 Testing the build wall scenario within the simulation. - Running the fastest solution, both robots moving bricks concurrently. 54
5.11 Testing the build wall scenario within the simulation. - Test finished, all bricks placed on their respective destinations. 55
5.12 Network diagram of the system during physical testing. 57
5.13 Connecting to the robot controllers from the API. 58
5.14 The application after successfully connecting to the robot controllers. 59
5.15 The method for testing concurrent movement on the physical robots (C# tasks). 60
5.16 Testing concurrent movement on the physical robots - Starting the test. . . 61
5.17 Testing concurrent movement on the physical robots - KUKA reached first position. 62
5.18 Testing concurrent movement on the physical robots - KUKA reached second position. UR moving away from first position. 63
5.19 Testing concurrent movement on the physical robots - UR reached second position. KUKA returned to home position. 64

6.1 A mind-map showing the different problems and challenges encountered during the project. 66

List of Tables

3.1 An example of a skill required for a scenario. 14
3.2 Skills necessary for the process of delivering an object. 15
3.3 Skills necessary for the process of building a wall. 16
3.4 Skills necessary for the process of moving a fragile object. 17
3.5 Skills necessary for the process of attaching a brick to a suspended object. . 18

Chapter 1

Introduction

As the complexity, abilities, and applicability of industrial robots are ever increasing, the prospect of efficiently having multiple robots collaborate within a shared workspace is appealing. Unfortunately, the task of programming industrial robots is not trivial, as different manufacturers often have different proprietary software and languages for their robots. Albert Nubiola described this as such:

> If you drive a car, it makes little difference what brand it is: all cars are driven in essentially the same way. The same applies to computers. If you have a Windows PC, the user interfaces won't be affected by your computer hardware. This is definitely not the case for industrial robots [10].

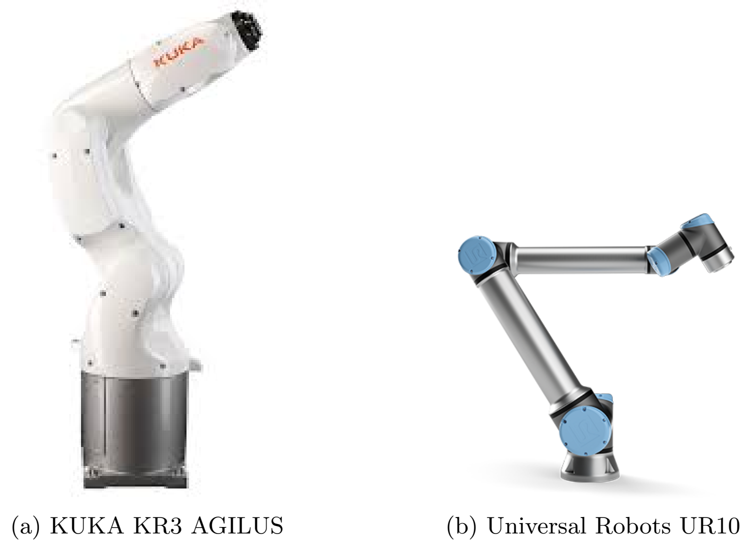
Incorporating multiple robots in a single task is already challenging, incorporating multiple robots that run on different languages makes this even harder. Conveniently, programs allowing automatic translation of commands to the correct language based on the robot's make and model do exist, and some of these programs even allow programming to be done through common object-oriented languages such as C#, Java or Python. By utilizing these programs, it should be possible to create a system capable of ensuring efficient and collision-free collaboration between multiple robots by creating a controller application in an object-oriented language, implementing the necessary logic for such operations.

This thesis attempts to investigate how current solutions that can bridge the gap between the different proprietary languages can be used in order to program robots of different brands within a single application allowing multiple robots to collaborate within the same workspace. The challenges that come with controlling multiple robots within the same workspace in an efficient way without causing collisions are also explored, to find a way to effectively determining the nature of such collaborative scenarios and coming up with suitable solutions to these challenges.

1.1 Background and Motivation

Høgskolen i Østfold is currently in the process of expanding their "robotics lab" with two new robots and a high fidelity 3D camera. The expanded lab will then have three different robots, all from different manufacturers and with different specifications. Of these three robots, two are industrial robotic arms; the first is manufactured by KUKA, model KR3 AGILUS, the second is manufactured by Universal Robots, model UR10. The last robot is a fully autonomous mobile robot designed for warehouses, and this robot is manufactured by OMRON, model LD60.

With two robotic arms available for use, it would be fascinating to create a system allowing both robots to collaborate concurrently. This collaboration should allow the demonstration of how multiple robots can speed up different tasks, or even enhance one another with, for example, increased reach.



(a) KUKA KR3 AGILUS (b) Universal Robots UR10

Figure 1.1: The two robotic arms.

Figure 1.2: The OMRON LD60.

1.2 Problem Statement

Making multiple robots collaborate in real time is a challenging endeavor. Accommodating robots from different brands in a single system is hard; having these robots with different languages collaborate in a meaningful way is even harder.

Currently, there exist a lot of different tools developed to make it easier for robots from different manufacturers to be programmed within a single language [18], the majority of these are so-called "offline-programming tools." These are programs that offer a unified environment for specifying the tasks of the robots individually, then generate brand-specific code which then can run on the robots. Of these tools, the most promising is RoboDk [13] due to its internal library of robots from more than 30 different manufacturers and a robust API allowing programs to be created in both Python as well as C#.

In order to attempt to create a system that allows meaningful collaboration between multiple robots, a way to classify different collaborative scenarios based on the requirements of the task would help identify scenarios with similar requirements. If the requirements are known, they could be implemented into the application, allowing the robots to perform tasks necessary for the scenarios. Different methods for planning, execution and determining the best plan could be tried depending on the identified classification based on the requirements.

The project seeks to explore two questions:

- How can RoboDk, and its API be used to implement collaboration between the two robots at Høgskolen i Østfold's robotics lab?
- How can different collaborative scenarios be classified based on the requirements necessary in order to accomplish said scenario?

1.3 Method

Firstly, a solution is designed by acknowledging the problems which need to be solved in order to create a working implementation of RoboDk and the API that can accommodate two robots collaborating and presenting different solutions. The most suitable solutions are then implemented in a Windows Forms application in order to form the interface that will utilize the chosen solutions, allowing the robots to collaborate through the API. The effectiveness of the solution is then discussed, and a conclusion is drawn regarding the results. Then things that could have been done differently are discussed, and possible future work is presented.

1.4 Report Outline

In chapter 2 the research topic is described in more detail, and already existing work will be presented. In chapter 3 the different challenges needing to be overcome before an effective application can be developed are presented, and different solutions are discussed and tested before the most suitable solutions are decided upon and implemented in chapter 4. Throughout chapter 5 the proof of concept solution is tested and evaluated before the findings are discussed in chapter 6. The project will be summarized in chapter 7 where a conclusion is presented.

Chapter 2

Analysis

2.1 Research Topic

Høgskolen i Østfold is now accommodating two robotic arms from two different manufacturers running on two different programming languages. It would be exciting to explore how these robots could be used collaboratively; either in order to solve complex tasks or perform more straightforward tasks more efficiently through concurrent control of both robots. The problem with controlling both robots simultaneously within a shared workspace is the difficulty of coordinating the robots such that the necessary tasks are accomplished efficiently and without having the robots collide. In order to solve this problem, a system capable of coordinating the robots is necessary, and in order to create such a system, the requirements of different forms of collaborative scenarios must be identified. This project aims to accomplish two things. Firstly to explore and evaluate possible ways to utilize RoboDk's API for controlling the robots concurrently such that they can collaborate on tasks, and to find a method for efficiently determining the required capabilities of the system in order to perform a collaborative scenario through some form of classification. Different methods for planning concurrent movement, executing the plans and identifying the most effective plans will be explored, and ultimately a proof of concept implementation of a solution capable of this will be produced.

2.2 Related Work

Literature regarding the specific scope of the problem faced in this project is quite lacking (multiple robots from different manufacturers collaborating within a shared space); however, there exist multiple recent publications utilizing simulation software. A common choice appears to be to utilize RoboDk for its in-built collision detection module, and run simulations with the collision detection enabled to make sure no collisions will happen. The simulation functions as a form of validation where if no collisions are detected the program can then be executed on the physical robots.

This approach is used in the publications [4] and [5] which both propose a new automatic method for calibrating industrial robots automatically and with cheaper calibration equipment compared to current conventional methods. Both of these publications utilize RoboDk for the validation of configurations, making sure they are collision-free during movement as well as at the final position.

The 2013 paper "A New Skill Based Robot Programming Language Using UML/P Statecharts" [17] presents a new way of programming robots with a domain specific language called LightRocks (Light Weight Robot Coding for Skills). LightRocks is built up by three different levels of abstractions, where domain experts can create "skills" that can be used on a more general level to create more complex tasks and processes by shop floor workers or technicians. This way of programming would fit very well with this project, as the skills can be coded into the software controlling the robots. These skills would include instructions for moving the robots as well as interacting with objects (grabbing and letting go); collision avoidance could also be performed at this level. The completed skills, as well as combinations of these (tasks), can then be made available in the interface for the software.

The first objective for this project, allowing multiple robots from different manufacturers to collaborate within the same workspace, can be divided into two core problems. The first problem is the challenge presented by having to rely on different languages for the robots. This reliance means that a solution would have to translate commands to the correct language or format for each robot per command. The second part to the problem is the problem of coordinating the different robots so that they can work safely and effectively within the same workspace without colliding. The second part of the problem will be the main focus of this project, as RoboDk handles the first part.

2.2.1 Multi-Arm Coordination

> The problem of finding collision-free trajectories for two arms operating independently in a shared workspace is called coordination. [3]

Kant et al. were the first to tackle the problem of multi-arm coordination in 1986 [8], [3]. Since then multiple algorithms have been developed [16], [11], [3].

Beuke et al. stated that dual-arm robots working within a shared workspace are a frequent problem in both service-oriented and industrial robotics. The specific challenge they respond to is the need to coordinated both independent arms both temporally and spatially [3]. They also point out that this problem is usually solved manually, which results in slow execution times and negative user experience. They present an algorithm which can automatically coordinate both robot arms of a two-armed robot so that they can share a workspace without colliding. They produce an algorithm that is incorporated into both the planning phase as well as the execution phase. This algorithm then allows for a solution that is both responsive, meaning that it can plan faster than it is executing, as well as reactive, meaning it allows for changes to be made to the plans when new goals are discovered during execution.

2.2.2 Software

The problem of programming robots from different manufacturers is well known, and a myriad of different software has been developed that makes it easier by defining one language or method that can be translated into different robots. A list of some of these programs can be found on Wikipedia [18]. Although these solutions make it easier to make separate programs for a multitude of different manufacturers, they do not provide much assistance in coordinating multiple robots directly. The ability to use one language for different robots makes it easier to create software that can implement this. Implementing

the necessary logical algorithms for such coordination is further enabled by the support of controlling these programs with object-oriented programming through the use of APIs. RoboDk's inclusion of such an API is what makes it a prime candidate for this project.

RoboDK

RoboDK is a commercial solution that supports both offline programming and three-dimensional (3D) simulation of industrial robots. RoboDk comes with an extensive library of over 30 different manufacturers [13]. Programs created for RoboDk is translated to the correct robot program code through the use of a post processor. The programs can be created graphically in RoboDk or programmatically with C# or Python through the use of their API [12]. RoboDK is not open source nor is it free, but comes with a 30-day trial license and retains limited functionality for experimentation (not commercial use) even without a license. The post processors used by RoboDk to translate programs to the different supported robot specific languages have been released to the open source and free robot programming software "robot operating system - industrial" (ROS-I) under an open source license [7].

Robot Operating System - Industrial

ROS-I is an open-source project that extends the advanced capabilities of "Robot Operating System" to manufacturing automation and robotics [6]. This support is provided with drivers for standard hardware used in manufacturing, like manipulators, grippers, sensors, and device networks [6]. Unlike RoboDk, ROS-I does not support graphical programming, but third-party solutions for this do exist [9].

Chapter 3

Design

3.1 Environment

The project will focus on a simulated version of the robot lab within RoboDK. RoboDK is a well-tested application with support for hundreds of different robots and manufacturers with the ability to compile programs for those robots or even directly run the programs on the robots. Having a functioning program in RoboDK should translate to a functional program on the physical robots.

3.1.1 Simulation Environment

RoboDk provides detailed and accurate simulation models for both the KUKA- and UR-robot present at the robot-lab. RoboDk will be used to simulate the robots, as well as present a visual view of the simulation (see Figure 3.1). The simulation will also include a simulated/digital view of the 3D camera, able to discover and track the position and orientation of items within the workspace. For this project, the objects in the simulation can be thought of as ***Lego bricks***.

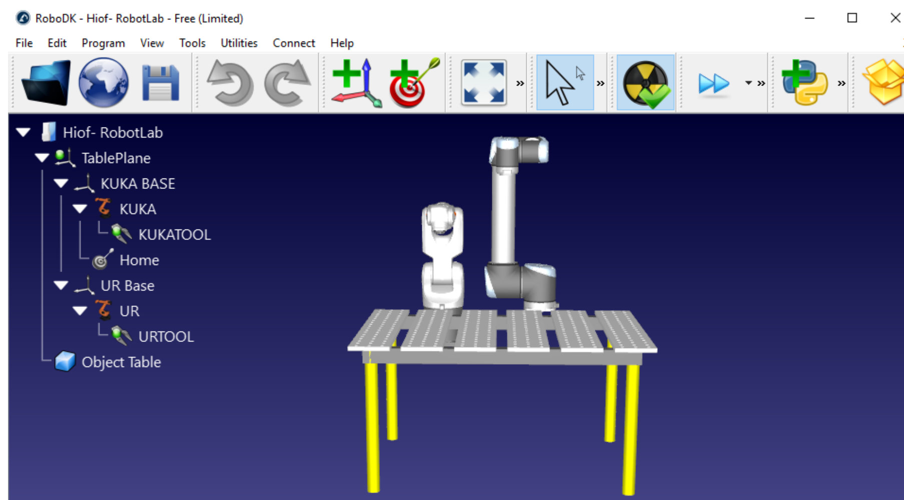


Figure 3.1: RoboDk simulation of both robots in a theoretical configuration allowing for collaboration between the robots.

3.1.2 Interface With RoboDk's API

An interface has to be created capable of sending commands to RoboDk through the C# API as well as provide a visual 3D simulation of the entire workspace so that the robots can be observed during operation. RoboDk provides a 3D visualization by itself which could potentially be utilized for this purpose. A windows forms application created in .NET would be able to implement logic through programming as well as sending commands to RoboDk through the API and could, therefore, function as a user interface. Actions such as moving and grabbing can be considered atomic and will be programmed into C# methods which will function similarly to tasks described in [17]. These skills can then be combined into tasks and processes which can all be made accessible in the interface using buttons.

3.2 Concurrent Operations

In order to be an effective tool for controlling multiple robots at the same time, as well as allowing for an effective form of collaboration between the robots, it is quintessential that the application can simultaneously control all of the robots in the simulation. The RoboDk API is stated to support controlling multiple robots in a multi-threaded application by connecting to the desired robots using the included NewLink method (see Section 4.3). After a separate link has been created for each robot, it can be assumed that the robots can be controlled separately and asynchronously through the implementation of C# Tasks. Concurrent movement of the robots could also be achieved by creating programs within RoboDk for each robot and starting these programs independently. The implementation of concurrent operations is explained in detail in Section 4.3 and 4.4.1.

3.3 Collision Avoidance

The main focus of this project will be implementing a way to enable collision free collaboration between the robots. It is essential that the application can either detect an imminent collision and take action in order to stop the robots or the objects in the workspace from colliding during execution, or guarantee that no collisions will occur by planning the movements before execution. When the robots are operating at speed, the damage inflicted to both the robots and other objects in a collision would likely be substantial even with if it would just be a glancing collision. In order to achieve non-colliding collaboration while controlling the robots with the developed tool, multiple solutions will be researched, proposed and evaluated.

3.3.1 Prevention Through Software Barriers

A simple and effective way to avoid collision between the robots would be to limit their allowed workspace in the application such that the different robots could not physically occupy the same space as each other. This restriction does, however, limit the amount of collaboration the robots can achieve. It is also possible to allow some form of collaboration by having some overlap between the robot's individual permitted workspaces, and this could be an ideal solution. This shared section of the workspace can then be seen as what is typically defined as a "critical region" in concurrent programming, exclusively

granting access to only one robot at a time [1]. The space that the robots could inhabit outside the intended workspace should be programmed as out of bounds in order to avoid having the robots collide with objects not meant to be interacted with, for example behind the robots, away from the intended workspace. By implementing a critical region, concurrent collaboration is not possible as the robots are prevented from effectively sharing the workspace.

3.3.2 RoboDk Collision Detection

RoboDk has a built-in collision detection mechanism which will stop all robots when a collision is detected [12]. This collision detection mechanism will only stop the robots after a collision has already occurred, which as explained could already have caused damage to the equipment and is therefore insufficient as a solution. It could be possible to run the robots in simulation mode and check for collisions before executing on the physical robots. By simulating before execution, it would be ensured that the robots would not collide during operation as long as the simulation is sufficiently accurate. If so, a collision avoidance solution like this would likely be the least technically challenging to develop. The accuracy of the simulations compared to execution were tested by running the same program multiple times, comparing each execution to each other; this showed a small variation well within what can be considered sufficiently accurate. This was expected as RoboDk is a well known and widely used application.

3.3.3 Calculating Distance Between Robot Joints

Another possible approach to preventing collision between the robots would be to calculate the dimensions, the position and the angle of the joints for each robot. This information could then be used to calculate the position of the robots in real time. This solution would be dependant on the processing speed of the computer, and as such, what could be considered a safe distance between the robots would change depending on the delay between angle readouts and the calculation of the position of the robot parts. The real-time calculations would be performed by a separate thread running on the application, getting the angle of the joints from the RoboDk API. The parts of the robot could be interpreted as either cylinder for best accuracy or boxes for faster calculations. This solution would not be technically challenging, but could be very demanding on the system, as well as functioning based on the distance between the moving robot parts and not predicted collisions; this form of collision avoidance is therefore not ideal for the project.

3.4 Execution Plans

In order to create execution plans for the robots, which is what actions the robots will take in what order or the programs to be executed by the robots is traditionally done in two main ways. The most common way of creating a program for the robots is by manually moving the robot with a teach pendant into each critical position for a task. An example of this would be to manually move the robot into a location above an object to be picked up, ensuring a straight path to the location the robot's tool needs to be in for the object to be grabbed successfully, also known as an "approach" location. Then after this position is taught to the robot, the operator would move the robot down to the location where the object can be grabbed by the robot, teaching this position to the robot too. Those

two locations, as well as a position out of the way of the workspace, known as the robots "home" position, can then be entered into the program in order. The program would then allow the robot to pick up the object and return "home." The second common way to program a robot is by an operator already knowing the coordinates or angles of the joints necessary to position the robot in the desired locations. These coordinates or angles can then be directly entered into the program during writing. This way of programming is a lot faster, but prone to errors due to the operator entering the wrong values.

For this project, the process of finding the necessary coordinates and angles in order for the robots to reach the desired locations should be automated. This automation will be done by tracking the locations of objects within the workspace and by predefining the robots' home locations. The software will also be responsible for solving the necessary levels of collaboration. It would then be possible to create and execute collaborative programs by utilizing predefined skills.

3.5 Execution

When it comes to executing the planned programs, there are several ways of doing this with different advantages and disadvantages. Mainly, an execution plan can be run without any prior or parallel control in the form of collision detection. This method would have the disadvantage of having a high probability of causing a collision during collaborative tasks. This form of execution does, however, require no computation before execution and few resources during execution and could be used for non-collaborative tasks such as having a robot pick up an object. A more sensible way of going about executing programs that require both robots to collaborate would be through the implementation of "intelligent programming" where the system would make sure the plan is collision free before execution by simulating the plan before executing it. This method can be achieved by allowing the application to manage a critical region, or simulate the planned motions, either in their entirety or in smaller sections. Another way of handling the execution of programs is to incorporate "responsive programming," meaning that the application can react to problems such as collisions before or when they occur. This method requires the application to monitor the position of both robots and objects within the workspace during execution. When the application detects that a problem is about to occur, the robots could be stopped or redirected in a direction that would avoid the problem until the task can continue. Alternatively, the application could detect when a collision has already occurred and reverse the movement that caused the collision and attempt a different solution. Having the robots collide before finding a collision-free solution might however not be desirable. Finally, if a process is to be repeated, previously successfully solutions can be saved and ran as the first method described, without any prior or parallel control.

3.5.1 Intelligent Programming

By implementing logic in the application controlling the API, simulations ran in RoboDk can not only be checked for collisions, but expected execution times can also be measured. This functionality allows the application to score different feasible solutions based on different criteria. The most important of such criteria would be that the solution is collision free. Secondly, it could score the solution based upon execution time or even safety (how close the robots come to colliding during execution). This evaluation can be

performed on a per command basis, or for entire processes consisting of various movements and interactions. Optimally, it is also possible to incorporate responsive programming in these solutions. If an implementation of responsive programming proves too difficult or time-consuming, evaluating programs before execution would still result in safe execution plans and might even be preferable to responsive programming, depending on the type of process to be executed.

3.5.2 Responsive Programming

By taking advantage of the possible real-time control of the robots, it could be possible to alter the movement of the robots during execution in case an undesirable situation occurs, such as an imminent collision. A practical implementation of this would be to start a simulation of a planned program shortly before the program starts execution on the physical robots. With the 3D camera able to track new objects appearing within the workspace and a connection from RoboDk to the physical robots, a functional "digital twin" would be able to make real-time adjustments to a process as it is executing on the physical robots. With such a digital twin, when the simulation foresees a problem, the system could respond by quickly simulating different changes until a movement plan that avoids the problem is found. This new plan could then be sent to the robots in order to continue an operation that would otherwise fail. Implementing responsive programming is explained further in section 4.7.

3.6 Collaborative Scenarios

Collaborative tasks in the scope of this project can be separated into two main groups, passive and active collaboration. Passive collaboration is the act of coordinating the robots such that they can both operate within the same space or for the same goal without hindering each other or colliding. Active collaboration is when both robots are required to performed synchronized tasks in order to achieve a goal that would be impossible with only one robot. The task of placing a set of bricks in a given location within the shared workspace could be classified as passive collaboration as the action of moving the bricks around can be done by either robot. A task of moving a fragile object, for example, a long thin bar that cannot support its weight when being carried by only one of the robots. Would require both robots to work together in a synchronized manner in order to achieve the task and would, therefore, be classified as active collaboration. In short, tasks requiring synchronized collaboration between the robots are active collaboration tasks. Tasks not requiring synchronization but benefit from coordination are passive collaboration tasks.

Skills

By breaking collaborative scenarios down into the necessary skills, it can easily be seen if a scenario can be classified as a passive or active collaboration scenario. These skills can be represented as rows in a table. An example of the skills required for a scenario is presented in Table 3.1.

Skill	**Platform**	**Type**
Accept a location where the object is to be placed	Software	Input
Avoid collision between the robots during execution	Software	Coordination

Table 3.1: An example of a skill required for a scenario.

The "Skill" column describes what the skill is achieving, the purpose of the skill. "Platform" refers to where the skill is happening; usually this would be on one or both robots, or the software. Where the skill is happening points to the limiting factor for the skill, for example, having the robot grab an object would be a skill happening on the physical robots, this requires the robots being physically able to grab an object by being equipped with a gripper. The skill of avoiding collisions would be handled by the application's software, with no dependence on the physical robots and would have "software" as its platform.

The "Type" column describes what kind of skill is being described. For example, the skill of being able to accept a location where an object is to be placed requires some form of input from the user. This skill would then be of type "Input," and as this is happening in the software, the platform would be "Software." Types of skills seen in this project are as follows:

- Input - some form of input to the system, normally from the user.
- Evaluation - requires the system to evaluate something, typically whether something is possible or not.
- Physical - requires a physical action to take place, normally executed on the robots.
- Simulation - a skill to be executed through simulation, normally involves simulating solutions in order to find one that can be executed on the physical robots.
- Coordination - requires some form of coordination of the system, normally the coordination of the robots such that collisions are avoided.
- Synchronization - requires some form of synchronization of the system, normally this would be to synchronize the robots such that they perform physical skills at a certain time or speed compared to each other.

When scenarios are broken down into skills in this way, the inclusion of a synchronization skill will classify the scenario as an active collaboration scenario. Another advantage of listing the skills necessary for a scenario is the ability to see what parts of the system needs to be implemented or changed in order to enable the system to perform the desired scenario. If each skill is implemented in a modular way, identical skills between two different scenarios would mean that a solution to these skills in one scenario would carry over to the solution of the next. For example, a scenario of picking up an object and placing it down would require all the same skills as a scenario of picking up and object, rotating it ninety degrees and then placing it down. As long as the first scenario is already implemented, only a skill allowing the object to be rotated ninety degrees before being placed down needs to be implemented in order to achieve the second scenario, allowing the implementation of all other skills to be reused.

3.6.1 Passive Collaboration Scenarios

Passive collaboration scenarios are processes built up by simpler tasks that each robot can accomplish alone while still benefiting from coordination between the robots. Primarily these processes would be completed faster when both robots participate compared to having only one robot perform all of the necessary tasks. What separates a passive collaboration scenario from a non-collaborative scenario is the application of coordination between the robots.

"Delivering" an Object

A simple demonstration of passive collaboration is the task of delivering an object. For delivering an object, the system must figure out how to move the object from one position within the workspace to another. If both the original position and the destination position is within reach of a single robot, this task is a simple pick and place task. The collaboration aspect comes into play when a brick is out of reach of a robot capable of reaching the destination, but within reach of another robot, unable to reach the destination. The idea is to have one robot pick up the object, then move it to a location within reach for the other robot, the second robot can then carry the object the rest of the way to the destination. The process of "delivering" an object can be broken down into several core skills that the system needs to accomplish in order to complete the task. These skills are as described in Table 3.2.

Skill	**Platform**	**Type**
Accept a reference to the object to be moved	Software	Input
Accept a location where the object is to be placed	Software	Input
Check if the object is reachable by either robot	Software	Evaluation
Check if the target location is reachable by either robot	Software	Evaluation
Decide how to deliver the object	Software	Evaluation
Move to pickup and drop-off locations	Robots	Physical
Pick up brick	Robots	Physical
Place the brick	Robots	Physical
Avoid collision between the robots during execution	Software	Coordination

Table 3.2: Skills necessary for the process of delivering an object.

The system can accomplish the input skills of accepting an object and location by including inputs in the user interface. The evaluation skills for determining if the object and target location are within reach of either robot will decide how the delivery is executed. As this task has few variables and is quite simple, it could be solved accurately and quickly by implementing an algorithm that decides the order of movements depending on what robots can reach the object and destination as well as how close those are to the respective robots. This task could also be solved by testing all possible solutions through simulation. However, running the simulations of all possible solutions would be slower than determining a functional execution plan by utilizing an algorithm.

The reach of the robots needs to be coded into the software. As RoboDk knows the location of the robots, the application can then calculate the distance to the object and destination for each robot. The physical skills are commands that need to be sent to the robots; this is accomplished by utilizing the RoboDk API, creating methods that call the

API in order to communicate with the robots. If the object needs to be transferred from one robot to another, a position reachable by both robots would need to be used for the exchange. This location could be calculated during execution or merely a pre-programmed location within reach of both robots.

Depending on whether an algorithm or a simulation is used to find the optimal execution strategy, the method of collision avoidance changes. For an algorithm, collision-free paths have to be guaranteed by not moving the robots into the same area at the same time, by for example moving one robot at a time and making sure the active robot moves back to its home position before moving the other robot. This algorithm would be very inefficient compared to simulating the movement of both robots, attempting different timings until an optimal solution was found that did not result in a collision. As the task is quite simple, and the expected execution time is brief, the extra time it takes to simulate multiple solutions would most likely be too long to make up for the more optimal execution. The exception to this would be if the process were to be repeated many times.

Building a "Wall"

By utilizing collision detection/avoidance as well as intelligent programming, it should be possible to discover and evaluate all feasible solutions to a multiple-step process through the use of simulation. For example, using multiple bricks to construct a wall. The bricks should be positioned in a spread across the workspace; the software can then automatically evaluate in which order each robot should move each brick in order to rearrange the bricks into a wall without collisions in the shortest amount of time.

The process of building a wall out of bricks can be broken down into several smaller tasks, and these tasks can be described as skills which the robots and controlling software need to be capable of performing. Skills necessary for this collaborative scenario are listed in Table 3.3.

Skill	Platform	Type
Accept location of wall	Software	Input
Accept number of bricks	Software	Input
Calculate destination position of each brick in the wall	Software	Evaluation
Check if bricks are available and reachable	Software	Evaluation
Check if wall will fit at the given location	Software	Evaluation
Move to brick locations	Robots	Physical
Pick up brick	Robots	Physical
Move to target location	Robots	Physical
Place the brick	Robots	Physical
Find optimal solution to the task	Software	Simulation
Avoid collision between the robots during execution	Software	Coordination

Table 3.3: Skills necessary for the process of building a wall.

The input skills are addressed by including inputs in the user interface. The software logic handles evaluation skills after the necessary inputs are passed to the application by the user. The application knows the reach of the robots, the dimensions of the constructed wall are calculated based on the number of bricks to be included. The application will then check if the bricks are within reach of at least one robot and that all locations for where

the bricks are to be placed when constructing the wall are also within reach of at least one robot. Each one of the physical skills represents required physical action of the robots and must be programmed into the software and then combined in order for the software to be able to simulate the process. The optimal solution is then found by running simulations of all solutions passing the initial evaluation. These simulations are then evaluated based on the total duration of the process. The simulations will also check for potential collisions during execution and only solutions without collisions are evaluated based on completion time. After all valid solutions have been evaluated the best one is presented and can be executed on the physical robots.

3.6.2 Active Collaboration Scenarios

Active collaboration scenarios are more complicated processes that require both robots to perform one or more actions that require synchronized execution of one or more actions from the other robot. For example, having one robot lift the right side of a large object while the other robot lifts the left side. What separates Active collaboration scenarios from passive scenarios is this dependency on synchronized actions from both robots, requiring both coordination as well as synchronized command execution.

Moving a Fragile Object

A solid demonstration of active collaboration between the robots is the process of handling a fragile object. The transportation of which requires both robots to support different sections of the object in a synchronized manner. For this scenario, a long and thin object will be placed within a reachable area of both robots and then moved to another location within reach of both robots. The area where the item can be moved to is limited by the reach of the shortest robot arm. Skills necessary for this process are listed in Table 3.4.

Skill	Platform	Type
Accept a reference to the object to be moved	Software	Input
Accept a location where the object is to be placed	Software	Input
Calculate valid points to lift the object from	Software	Evaluation
Check if valid points on the object are reachable	Software	Evaluation
Check if target the location is reachable	Software	Evaluation
Move to the attachment point of the object	Robots	Physical
Grab the object at the identified spot	Robots	Physical
Find an optimal solution to the task	Software	Simulation
Synchronously lift the object	Software	Synchronization
Synchronously move the object horizontally	Software	Synchronization
Synchronously place the object at the target location	Software	Synchronization
Avoid collision between the robots during execution	Software	Coordination

Table 3.4: Skills necessary for the process of moving a fragile object.

Both input skills will be achieved through the interface of the application. In order to know where the object can be grabbed, valid grabbing-points must either be manually specified in the application or calculated by the application on a per object basis; it could also be possible to allow the user to specify grabbing points through the interface. The

remaining two evaluation skills are achieved by the software logic, comparing each robot to the position of the grabbing-points and the location for putting the object down. The robot with the shortest reach will primarily restrict the range of which an object can be picked up from and how far it can be moved. The physical skills need to be programmed into the software and then combined to make up the process. The simulation will attempt to find the best solution based on execution time. For the synchronization skills, the movement speed of the robots must be set so that they move at precisely the same rate, and the movement paths must be parallel to one another. This movement speed will be calculated by the software based on the speed of the slowest robot, ensuring through the simulation that both robots can maintain the set speed for the entirety of the movement paths during execution. The fastest feasible speed for the robots to operate at will be found by the simulation, also making sure the robots do not collide when moving to the position of the grabbing-points.

Attaching Bricks to an Object While the Object is Suspended.

Another example of active collaboration is the process of manipulating an object with one robot, to grant access to otherwise inaccessible angles for the other robot to perform actions on the object. With only a single robot, the act of attaching Lego bricks to the underside of another brick would be impossible without stacking the bricks. The idea is to have one of the robots pick up a brick ("suspending" it by holding on to it) and then angling it such that the other robot can attach another brick to the underside of it. Then the first robot can place the new structure down after the second robot is done attaching the new brick. Although this exercise is not very useful in itself, the process can easily be compared to actions such as performing welding on different sides of an object. The skills necessary for such a process are listed in Table 3.5.

Skill	Platform	Type
Accept a reference to the brick to be attached	Software	Input
Accept a reference to the target brick	Software	Input
Check if both bricks are reachable	Software	Evaluation
Move to brick and target brick	Robots	Physical
Grab the target brick	Robots	Physical
Grab the brick to be attached (from the side)	Robots	Physical
Calculate a location and angle for attaching the brick	Software	Evaluation
Move target brick to the calculated position and angle	Robots	Physical
Attach the bricks together	Software	Synchronization
Avoid collision between the robots during execution	Software	Coordination

Table 3.5: Skills necessary for the process of attaching a brick to a suspended object.

Like the other scenarios, both input skills are accomplished by allowing the selection of the target brick and the brick to be attached through the interface. The software checks if the bricks are reachable by calculating the distance to the bricks for each robot and making sure both robots can pick up different bricks. The first two commands to be sent to the robots are the same as in the other scenarios, movement, grabbing and letting go.

The target brick needs to be grabbed differently than the other bricks, as the top of the brick must be available for connecting to the target brick, the robot gripper must hold onto

the brick from the side. This skill would be implemented as a separate grabbing-method in the application.

The physical movement of position and angle are also new, and in order to support this sort of process, the application must be able to communicate to the robots that the objects need to be held a specific angle. It would be possible to attach a brick to the underside of another while both were being held straight. However, as this process is supposed to resemble more useful processes, the ability to operate at an angle is deemed necessary.

For this sort of process, a simulation could be run in advance of the execution in order to make sure the planned paths did not cause a collision, but a simulation would not handle the most likely issues. The most likely problematic situations with this sort of process would emerge during the attachment task when the two robots are both exerting force on the handled bricks at the same time. The most likely issue would be that the bricks slide or drop from one or both of the robots during the attachment task; a simulation in RoboDk would not be able to foresee this issue.

A better solution for this problem is to use data from RoboDk, and the 3D-camera to monitor the actual state of the workspace and react to unforeseen situations in real time. The planned execution could be simulated in RoboDk while the data from the real execution was being compared to the simulation; if a difference was discovered, the system could halt or adjust.

3.7 Categorizing Collaborative Problems and Their Solutions

The goal of this project is not just identifying the different forms of collaboration between industrial robots, but also to categorize them and investigate the applicability of broader strategies that can guarantee a system's ability to achieve that form of collaboration. By comparing the required skills for different scenarios, groups can be formed for the different challenges presented in different forms of collaboration. Solutions that would permit a system to accommodate the skills necessary for that group can then be investigated. The goal is to identify a sufficient amount of groups such that the most common collaboration scenarios can be put into categories where the requirements of the system are already determined. Initially, two major groups for collaboration have been defined, active and passive collaboration. As defined in Section 3.6.1 both the "delivery" and "building a wall" scenarios can be classified as passive collaboration scenarios. The two other scenarios, "moving a fragile object" and "attaching bricks to an object while the object is suspended" can be classified as active collaboration scenarios, as defined in Section 3.6.2. The ability of the complete system to perform a specific ability or task can be called a "skill" that the system possesses, or is required by the given scenario. By breaking the scenarios into separate necessary skills, it is possible to classify different collaboration scenarios as either passive or active collaboration scenarios by looking for the inclusion of synchronization type skills. These skills require more precision from the robots themselves as well as the controlling system. The coordination type skills are necessary for a scenario to be considered collaboration at all.

Coordination Skills

In order to require coordination type skills, some form of coordination must be necessary to achieve the goal of the scenario. This coordination could, for example, be the management of a critical region as described in the delivery scenario by an algorithm allowing only one robot to be present within the region at a time. It could also be the designation of tasks to be completed by what robot in what order like described in the wall building scenario. Another example of collaborative coordination would be a scenario where two robots work on an assembly line with different tools. One robot could use a specialized tool in order to enable the other robot's tool to be effective. For example, having one robot with a tool specialized for rapidly stacking items on top of each other in order for the other robot's tool to be able to pick up the entire stack further down the assembly line.

Synchronization Skills

Synchronization skills are primarily operations that require the robots to move synchronized relative to each other. This type of skill could be lifting two sides of an object at the same speed and on a parallel trajectory as described in the "moving a fragile object" scenario. Another example of a synchronization skill is using both robots to manipulate a single object, as described in the "attaching bricks to a suspended object" scenario. Here the robots would not move at the same speed or on parallel trajectories, but one robot would use its tool for holding the object, while the other robot would use its tool to attach a brick to the underside of the first object. The robots could each push the object and brick together, or one could hold the object still, while the other robot pushes the brick against it. For either example, the control system should be able to detect if something goes wrong. For example, the object slips from the grip of either robot or the robots stop moving at the same speed. In order to correct an unwanted situation, the controlling software would have to be able to both detect an imminent accident and alter the execution plan during execution. This form of control is a lot more challenging than the simpler coordination type skills, as it requires some form of sensors as well as responsive programming.

Core Skills

Some skills will be required in most scenarios, both passive and active collaboration scenarios, and even scenarios that require no collaboration at all. These skills can be described as core skills. Core skills include input type skills, like allowing the designation of what objects to manipulate and where to place them and physical type skills that result in actions from the robots; like moving, grabbing or letting go of objects. These skills do not provide any information for classifying what type of collaboration a scenario is. They are still necessary for a system in order to perform a type of scenario and could be used to designate what tasks can be performed by either robot. If an object needs to be grabbed, and only one robot has a tool capable of grabbing the object, this robot would have to perform this action. As such, if a scenario requires an object to be grabbed, the solution would have to incorporate a physical skill capable of grabbing an object.

3.7.1 Utilizing New Skills For Expanding Existing Processes To Enable New Scenarios

If the system can accomplish a solution to a scenario, for example, if the system can perform the skills necessary for manipulating a suspended object. The introduction of a physical type skill capable of performing spot welding the system should then be able to perform this kind of welding on an object, given that no skills are removed (like removing the grabbing tool in order to attach the welding tool). Another example of enabling new scenarios by adding new skills is to build upon core skills by combining them with repeated or new core skills or synchronization skills. A system capable of picking up an object and placing it down is perhaps not very useful in itself, and not capable of collaboration. However, By adding a coordination skill of controlling a critical region by sequencing the movements of the robots as described in the delivery scenario. The available workspace for both robots is extended without risk of collisions.

Chapter 4

Implementation

4.1 RoboDk

For this project, the latest version of RoboDk (3.5) was downloaded and installed from the official website [13]. Although it is possible to programmatically add and configure any object to a RoboDk workstation/project through the API, a station was created with a table, the two robots and reference frames for each robot and the worktable. This station is then loaded into the RoboDk simulation through the API when the interface is launched. The created station is shown in figure 4.1.

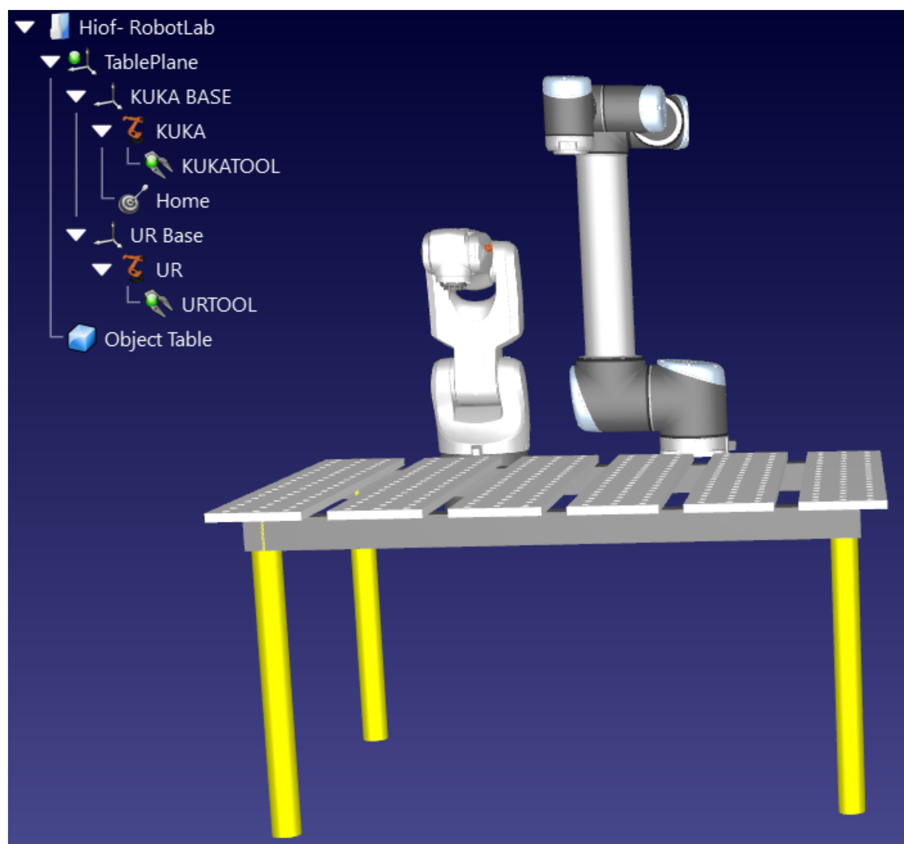


Figure 4.1: The RoboDk station to be loaded at launch

4.1.1 Changes to the API

The RoboDk API is available for C#, Python and Matlab [12]. Although it is available for C#, the API was initially created for Python; as a result, the Python version of the API has more functionality that is not yet officially implemented in the C# version. One of the commands not yet included in the C# version of the API is the "Attach" and "Detach" commands. These commands allow objects to be attached to the robot tools in the simulation. In order to use these commands in the project, they had to be manually converted from the Python API [14] to C# code and appended to the C# API. The "setParentStatic" method (shown in figure 4.2) was imported from the Python API to the C# API (as shown in figure 4.3) in order to provide the functionality of the Attach and Detach methods.

```
2805        def setParentStatic(self, parent):
2806            """Attaches the item to another parent while maintaining the current absolute position in the station.
2807            The relationship between this item and its parent is changed to maintain the abosolute position.
2808
2809            :param parent: parent to attach the item
2810            :type parent: :class:`.Item`
2811
2812            .. seealso:: :func:`~robolink.Item.setParent`
2813            """
2814            self.link._check_connection()
2815            command = 'S_Parent_Static'
2816            self.link._send_line(command)
2817            self.link._send_item(self)
2818            self.link._send_item(parent)
2819            self.link._check_status()
```

Figure 4.2: The setParentStatic method in the python API.

```
/// <summary>
/// Imported from python library by Mads Benjamin Fjeld
/// </summary>
/// <param name="item">the item to attach</param>
4 references | Mads Benjamin Fjeld, 33 days ago | 1 author, 1 change
public void Attach(Item item)
{
    link._check_connection();
    link._send_Line("S_Parent_Static");
    link._send_Item(item);
    link._send_Item(this);
    link._check_status();
}
```

Figure 4.3: The setParentStatic method imported to the C# API.

The Attach method (after importing the setParentStatic method from the Python API) in the C# API could then be used to attach the bricks in the simulation to the robots, and also detach the bricks by attaching them to the simulated table once the robots are supposed to let go of the bricks. This method only functions as a simulated pick up and drop action, allowing the visualization to depict the bricks being picked up

and placed down by the robots in the simulation. For the real robots, a separate method has to be implemented to function with the real tools attached to the robots, the actual implementation of the method will depend on the model of the tool equipped to the physical robot.

4.2 Interface With RoboDk's API

For this project, a lightweight proof of concept interface was designed and implemented using .NET Windows Forms and the RoboDk C# API. The interface implemented a set of predefined "skills." The goal is to have the robots collaborate meaningfully and effectively, with as few inputs as possible. With this sort of solution, the skills need to be defined in advance by someone with at least moderate programming experience. After the skills are defined and made accessible through the interface, they could be used as "building blocks" for more complex tasks for the robots to accomplish. The software would then use the API to make sure the desired tasks were safe to execute (possible and without collisions), after validating the tasks, the tasks could be executed. The interface could then potentially be used in the robotics lab for demonstrating how the robots can be used for collaborative scenarios. The interface is shown in figure 4.4.

4.2.1 Tracking Bricks in RoboDk

As the robotics lab at Høgskolen i Østfold also contains a high fidelity 3D camera capable of recognizing and tracking objects in 3D space, this functionality was simulated in the interface with the "Add Brick" and "Add Bricks" buttons. This simulation represents the recognition of a brick in the workspace. When a brick is "recognized," a brick is created in the RoboDk simulation at the provided coordinates through the API. The interface then tracks the reference to this RoboDk object within the windows forms application. The tracked bricks are displayed in a selectable list in the interface. This list is used to select and interact with the bricks. Selected bricks are highlighted in the simulation by changing the color of the corresponding bricks to green through the API (shown in Figure 4.5).

4.3 Concurrent Movement

The task of supporting the concurrent movement of the robots through the API presented a couple of problems. The biggest technical problem to this was the fact that programs are executed sequentially, it was, therefore, necessary to create a multi-threaded application that could send API calls for both robots concurrently and independently. The API contains a method for creating new links to the RoboDk software. This method is described to support multi-threaded applications (shown in Figure 4.6).

By utilizing this method on each robot, and then creating separate tasks for each robot (allowing the control of the robots to be run on separate threads) the desired effect of simultaneous execution was achieved. An example of how such tasks could be created and executed is shown in figures 4.7, 4.8 and 4.9.

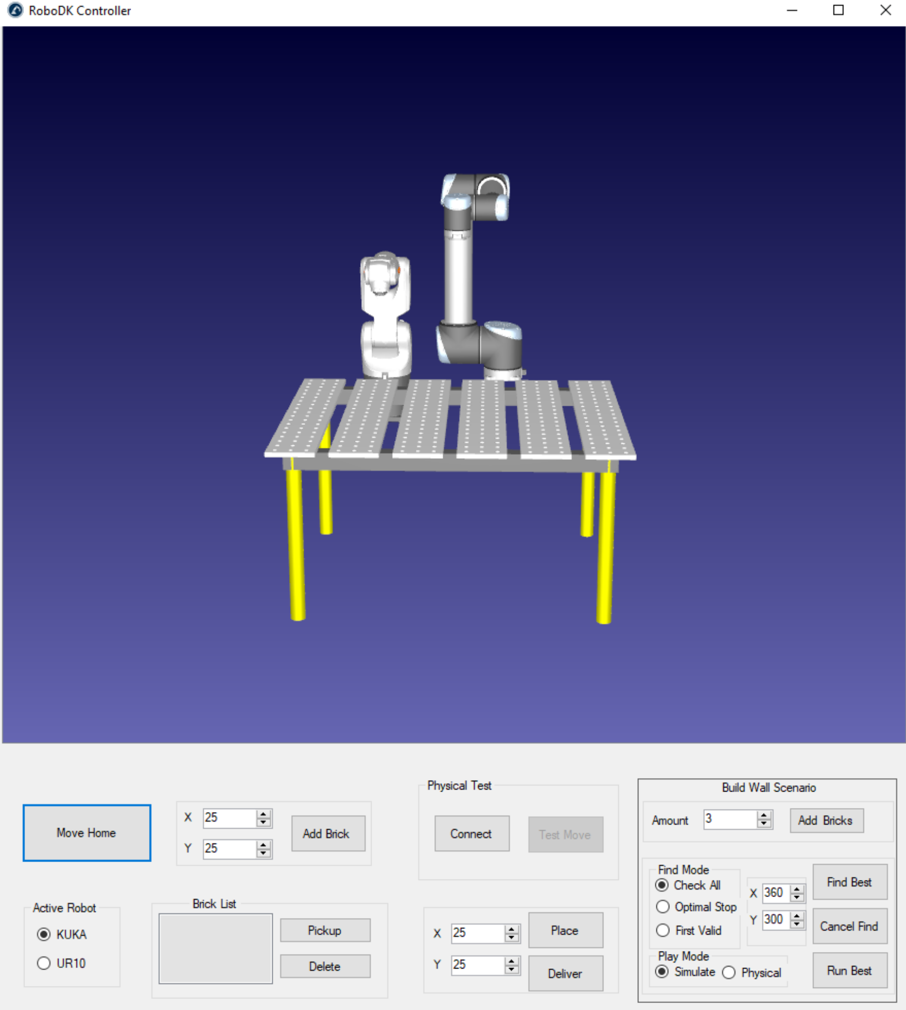


Figure 4.4: The interface to interact with RoboDk through the API.

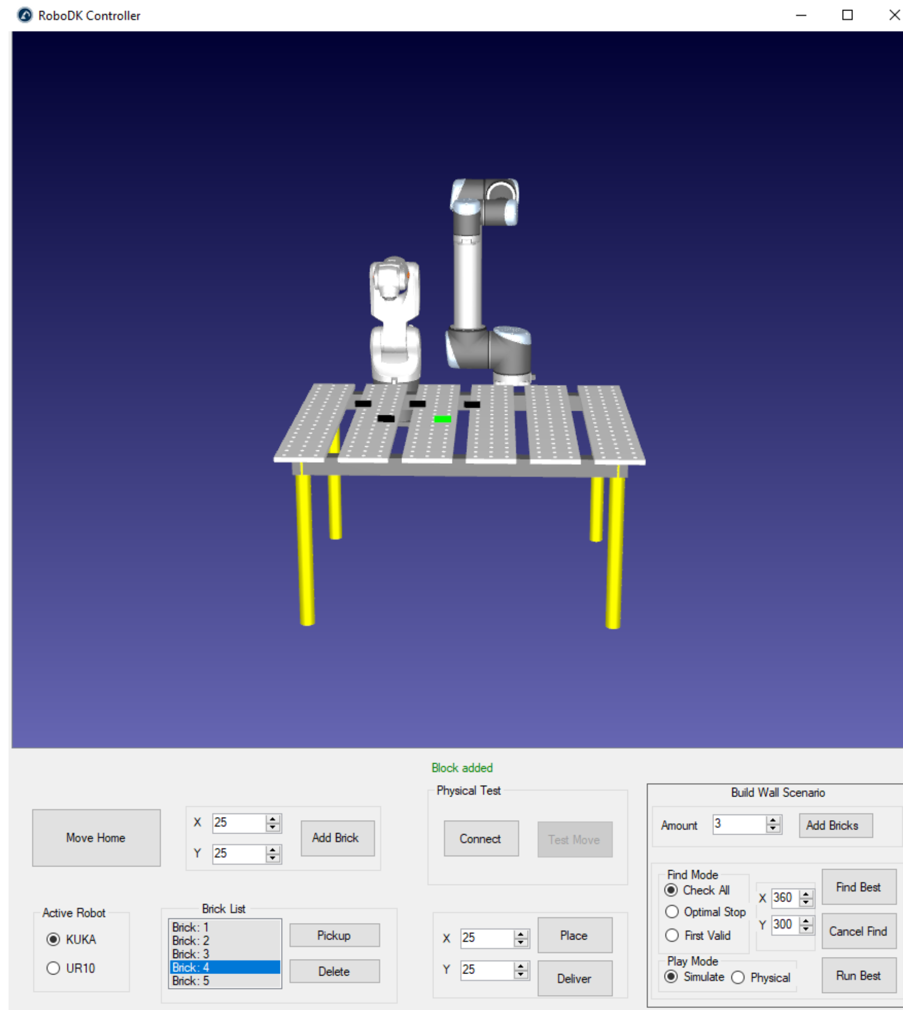


Figure 4.5: The list of tracked bricks in the interface.

```
/// <summary>
/// Create a new communication link. Use this for robots if you use
/// a multithread application running multiple robots at the same time.
/// </summary>
2 references | Mads Benjamin Fjeld, 33 days ago | 1 author, 1 change
public void NewLink()
{
    link = new RoboDK();
}
```

Figure 4.6: The NewLink method for multi-thread support.

```
private void MoveTaskExampleAsync()
{
    // Set runmode to simulate only in order to evaluate operation before running it on the physical robots.
    RDK.setRunMode(RoboDK.RUNMODE_SIMULATE);

    // Positions for a brick and the corresponding approach location.
    var PlacePos = new double[] { 600, 300, 10 };
    var APos = new double[] { 600, 300, 50 };

    // Creating movement targets for the UR robot.
    RoboDK.Item URAPos = GenerateTarget(APos, UR_ROBOT);
    RoboDK.Item URPlacePos = GenerateTarget(PlacePos, UR_ROBOT);

    // Updating the brick position and approach location.
    PlacePos = new double[] { 300, 300, 10 };
    APos = new double[] { 300, 300, 50 };

    // Creating movement targets for the KUKA robot.
    RoboDK.Item KukaAPos = GenerateTarget(APos, KUKA_ROBOT);
    RoboDK.Item KukaPlacePos = GenerateTarget(PlacePos, KUKA_ROBOT);

    // Creating a task for moving the UR robot.
    Task URMoveTaskExample = new Task(() =>
    {
        UR_ROBOT.MoveL(URAPos, true);
        UR_ROBOT.MoveL(URPlacePos, true);
        UR_ROBOT.MoveL(URAPos, true);
    });

    // Creating a task for moving the KUKA robot.
    Task KukaMoveTaskExample = new Task(() =>
    {
        KUKA_ROBOT.MoveL(KukaAPos, true);
        KUKA_ROBOT.MoveL(KukaPlacePos, true);
        KUKA_ROBOT.MoveL(KukaAPos, true);
    });

    // Starting the tasks.
    URMoveTaskExample.Start();
    KukaMoveTaskExample.Start();

    URMoveTaskExample.Wait();
    KukaMoveTaskExample.Wait();
}
```

Figure 4.7: Creating and executing tasks for moving the robots concurrently.

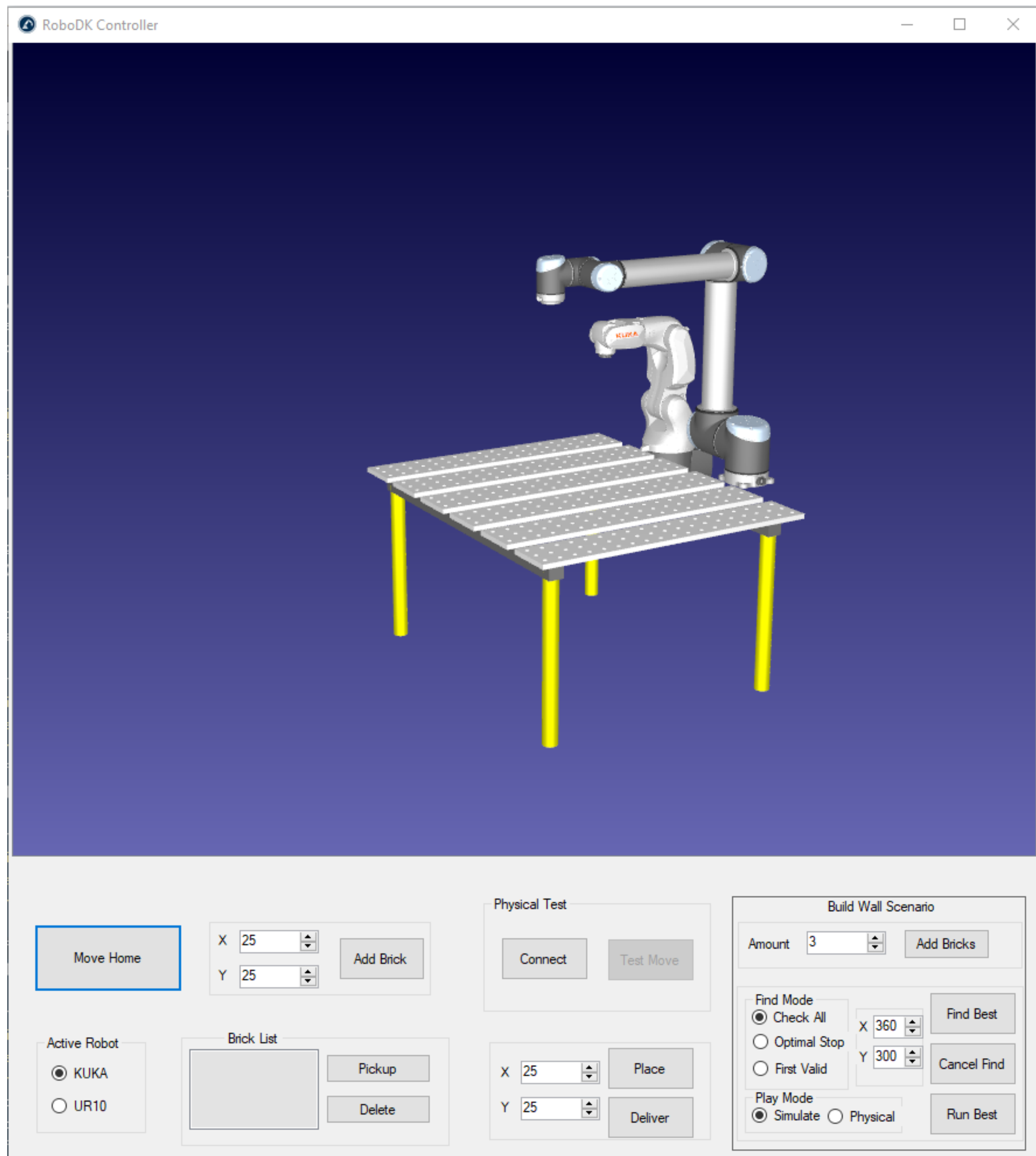


Figure 4.8: The robots moving concurrently with C# tasks - The tasks are started at the same time.

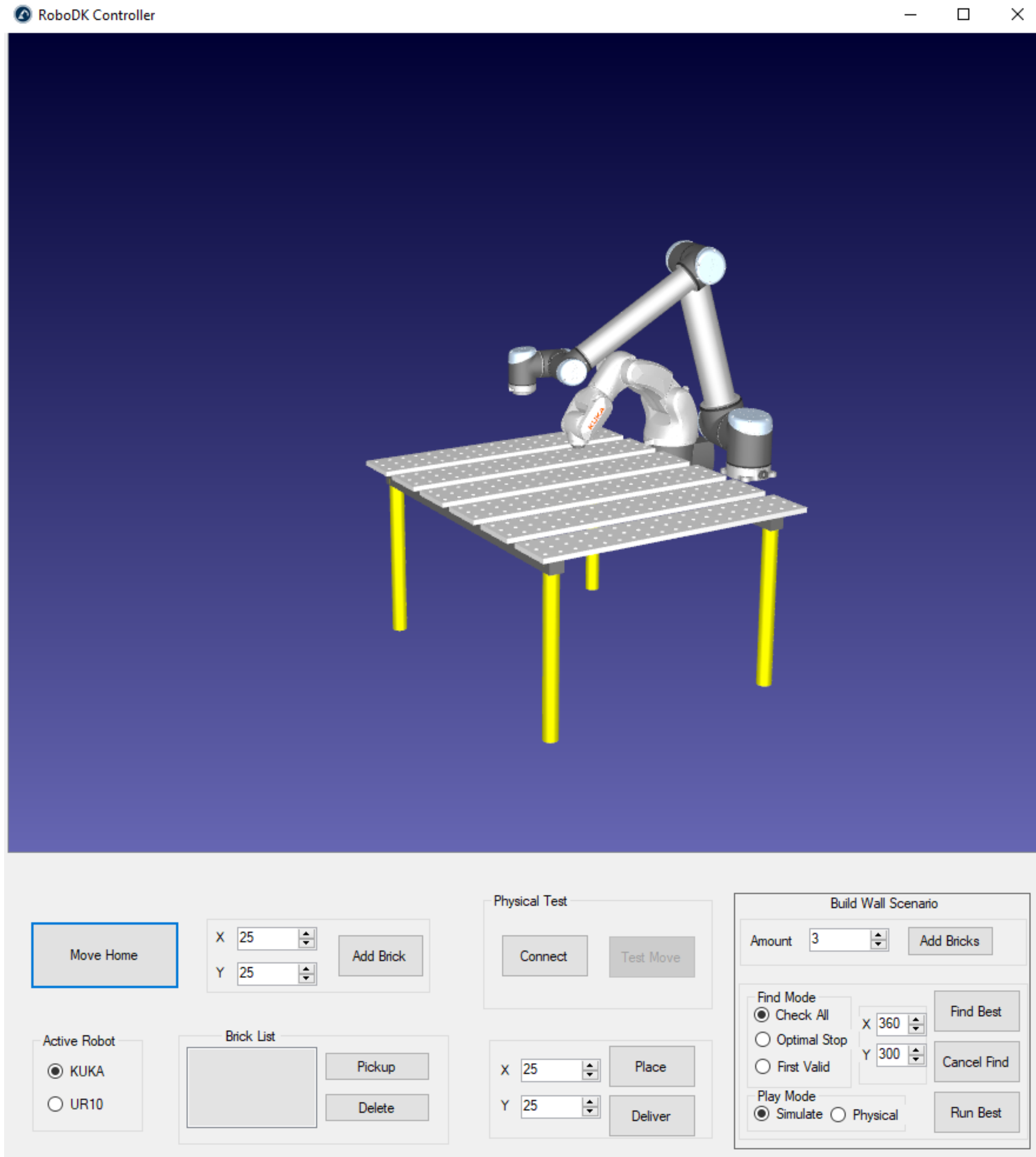


Figure 4.9: The robots moving concurrently with C# tasks - The KUKA moves faster than the UR robot, KUKA has reached the first destination, UR is still moving towards it.

Creating programs for the robots within RoboDk and then executing these programs is another way of achieving concurrent movement of the two robots, at the cost of less control when it comes to collision avoidance (see Section 4.4.1).

4.4 Collision Monitoring and Avoidance

RoboDk provides a few ways to detect when a collision has happened. When the robots are moved (by a program or task), RoboDk can continuously check for collisions between entities specified in a collision matrix (shown in Figure 4.10). Unfortunately, this way of detecting collisions presents a few different problems. Firstly, the execution of a program slows down severely when the joints of a robot approach the table, presumably this is caused by more intense checking for collisions as the distance between two entities able to collide with each other (as defined by the collision matrix in Figure 4.10) gets shorter. Additionally, this requires the robots to collide before the program is stopped, this is too late as the physical robots would already have caused damage to themselves, each-other or equipment by the time RoboDk tells them to stop. In order to overcome the problem of detecting a collision too late, the programs were run in simulation mode (preventing any real robots from moving), and then if they ended without a collision, the program could then be executed on the physical robots.

Collision Map Settings ? ✕

Double click a cell to turn ON or OFF collision check ☐ Include hidden objects [Select all] [Select none] [Set default selection]

	KUKA (Base)	KUKA (J1)	KUKA (J2)	KUKA (J3)	KUKA (J4)	KUKA (J5)	KUKA (J6)	KUKATOOL	UR (J0)	UR (J1)	UR (J2)	UR (J3)	UR (J4)	UR (J5)	UR (J6)	URTOOL	Object Table
KUKA (Base)	✥	✖	✓	✓	✓	✓	✖	✔	✓	✓	✓	✓	✓	✓	✖	✔	✖
KUKA (J1)	✖	✥	✖	✓	✓	✓	✖	✔	✓	✓	✓	✓	✓	✓	✖	✔	✓
KUKA (J2)	✓	✖	✥	✖	✓	✓	✖	✔	✓	✓	✓	✓	✓	✓	✖	✔	✓
KUKA (J3)	✓	✓	✖	✥	✖	✓	✖	✔	✓	✓	✓	✓	✓	✓	✖	✔	✓
KUKA (J4)	✓	✓	✓	✖	✥	✖	✖	✔	✓	✓	✓	✓	✓	✓	✖	✔	✓
KUKA (J5)	✓	✓	✓	✓	✖	✥	✖	✔	✓	✓	✓	✓	✓	✓	✖	✔	✓
KUKA (J6)	✖	✖	✖	✖	✖	✖	✥	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
KUKATOOL	✔	✔	✔	✔	✔	✔	✖	✥	✔	✔	✔	✔	✔	✔	✖	✔	✔
UR (J0)	✓	✓	✓	✓	✓	✓	✖	✔	✥	✖	✓	✓	✓	✓	✖	✔	✖
UR (J1)	✓	✓	✓	✓	✓	✓	✖	✔	✖	✥	✖	✓	✓	✓	✖	✔	✓
UR (J2)	✓	✓	✓	✓	✓	✓	✖	✔	✓	✖	✥	✖	✓	✓	✖	✔	✓
UR (J3)	✓	✓	✓	✓	✓	✓	✖	✔	✓	✓	✖	✥	✖	✓	✖	✔	✓
UR (J4)	✓	✓	✓	✓	✓	✓	✖	✔	✓	✓	✓	✖	✥	✖	✖	✔	✓
UR (J5)	✓	✓	✓	✓	✓	✓	✖	✔	✓	✓	✓	✓	✖	✥	✖	✔	✓
UR (J6)	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖	✥	✖	✖
URTOOL	✔	✔	✔	✔	✔	✔	✖	✔	✔	✔	✔	✔	✔	✔	✖	✥	✔
Object Table	✖	✓	✓	✓	✓	✓	✖	✔	✖	✓	✓	✓	✓	✓	✖	✔	✥

Figure 4.10: The default collision map for the RoboDk station.

Another possible and faster way to detect and prevent collisions is through the API's Collisions method. This method checks the current orientation and location of objects in the station and returns the number of pairs of objects which are currently colliding. This function could be called rapidly in a separate thread in order to detect when a collision has occurred without slowing down the simulation when objects get closer to each other. This approach would be a better alternative to the first method described above (utilizing RoboDk's collision matrix during program execution). However, when attempted, this rapid use of the API calls caused the API to throw an exception and cease unless the delay was too long to be useful (checking for collisions slower than once every two seconds). Because of this, the first method of detecting collisions was chosen.

4.4.1 C# Tasks and RoboDk Programs

C# Tasks

Creating C# tasks for longer processes proved to be quite challenging, as creating movement targets through the API within the tasks would often result in the API throwing an exception and stop the execution. It appears as having items in RoboDk being created on a separate thread from the one having the primary connection causes some referencing issue. When the targets are created and referenced in new threads, the API will throw exceptions explaining that the target does not exist. When the targets were created on the main thread, and the new threads exclusively ordered the movement of the robots to these targets, no exceptions were thrown. A possible solution to this could be to have all targets created in advance from the main thread of the application and then added to a list, available to the tasks controlling the robots. In this thesis, C# tasks that control the robots are referred to as either C# tasks or just tasks.

RoboDk Programs

An easier way of handling the concurrent movement of robots for longer operations is to create programs within RoboDk through the API. These programs can then be started asynchronously. These programs will run within RoboDk itself, as opposed to running within the control application. When all the moves for each robot are placed within RoboDk programs, RoboDk will allow the robots to plan their motions based on future movements. This allows the robot to more efficiently reach each position in the program compared to individually running each movement command one at a time through a C# task. The increased ease of generating these programs and the ability to plan multiple movements at once comes with the disadvantage of not being able to change the planned movements of the robots during execution. For dynamic processes, where there is a high chance of something changing during execution, this would not be a satisfactory solution. However, for static processes where a complete plan can realistically be determined before starting the execution, a wide variety of solutions can be planned and simulated before execution on the physical robots. When generating RoboDk programs, the attach method from the API does not function correctly as the programs do not support it. The inability to use the attach method means that when executing RoboDk programs, the simulation cannot show the bricks being picked up and moved by the robots (see Section 4.1.1). In this thesis, RoboDk programs are referred to as either "RoboDk programs" or just programs for short.

Using RoboDk Programs for Preventing Collisions and Scoring

Assuming RoboDk's simulation of the robots is accurate, these simulations can check for collisions with RoboDk's collision matrix as well as expected execution time. A practical way to use the RoboDk simulation to achieve this is to create a program for each robot consisting of the desired movements, then run these programs concurrently on a separate thread. When the program finishes (by running through all the commands or stopping due to a detected collision) the interface can call the Collisions method of the API to see if the system is currently in a collided state. If there are no object pairs in a collided state after the programs finish executing, the programs are considered safe. If multiple variations of the desired programs are evaluated this way, they can be scored based on their simulated

execution time, this allows for "intelligent programming." An example of how the robots can be controlled concurrently using RoboDk programs are shown in figure 4.11 and an example for collision check as well as execution time scoring is shown in figure 4.12.

Implementation

Because programs do not support the use of the attach method, C# tasks were implemented for all of the tasks that do not require concurrent movement of the robots (move home, pick, place, deliver) so that the bricks could be visualized being picked up and placed down. The collaboration task of building a wall implements tasks when running the best solution, for the same reason. However, because utilizing RoboDk programs allows for more optimal movement paths between positions, it would likely be better to use programs for all tasks. Because the bricks do not need to be shown being moved during the simulations, programs were implemented when simulating the different solutions for the task of building a wall. If the system was fully implemented with real robots and a 3D camera, the system could rely on the camera tracking the bricks, thereby not needing the attach method to show the bricks moving, allowing for programs to be used for all movements, this would be ideal. Because the simulations utilize programs, while tasks are used when running the best solution, the actual movements of the robots might not match the simulation when running the solution on the robots. When the solution is running in the simulation, this is not problematic, but if the solution is to be run on physical robots, the simulation and execution of the best solution should both utilize tasks or programs in order to avoid discrepancies in movement and possible collisions.

```
private void MoveProgramExampleAsync()
{
    // Positions for a brick and the corresponding approach location.
    var PlacePos = new double[] { 600, 600, 10 };
    var APos = new double[] { 600, 600, 50 };

    // Creating movement targets for the UR robot.
    RoboDK.Item URAPos = GenerateTarget(APos, UR_ROBOT);
    RoboDK.Item URPlacePos = GenerateTarget(PlacePos, UR_ROBOT);

    // Updating the brick position and approach location.
    PlacePos = new double[] { 300, 300, 10 };
    APos = new double[] { 300, 300, 50 };

    // Creating movement targets for the KUKA robot.
    RoboDK.Item KukaAPos = GenerateTarget(APos, KUKA_ROBOT);
    RoboDK.Item KukaPlacePos = GenerateTarget(PlacePos, KUKA_ROBOT);

    // Create new programs for the robots, ensuring that they are empty.
    ClearProgram(PROG_KUKA);
    ClearProgram(PROG_UR);

    // Add the movement commands to the KUKA program.
    PROG_KUKA.MoveJ(KukaAPos);
    PROG_KUKA.MoveL(KukaPlacePos);
    PROG_KUKA.MoveJ(KukaAPos);

    // Add the movement commands to the UR program.
    PROG_UR.MoveJ(URAPos);
    PROG_UR.MoveL(URPlacePos);
    PROG_UR.MoveJ(URAPos);

    // Create a task that will run the programs on a separate thread (allowing the program to not be frozen during execution).
    var ProgramTask = new Task(() =>
    {
        PROG_UR.RunProgram();
        PROG_KUKA.RunProgram();

        // wait for both programs to stop before reaching the end of the task.
        while (PROG_KUKA.Busy() || PROG_UR.Busy())
        {
            Thread.Sleep(200);
        }
        return;
    });

    // Score the programs based on execution time, score will be -1 if a collision was detected during execution.
    var score = MonitorCollisions(ProgramTask);
}
```

Figure 4.11: Creating and executing RoboDk programs for moving the robots concurrently.

```
private async Task<double> MonitorCollisions(Task ProgramTask)
{
    bool collision = false;

    // Set runmode to simulate only in order to evaluate operation before running it on the physical robots.
    RDK.setRunMode(RoboDK.RUNMODE_SIMULATE);
    // Time the program
    var timer = new Stopwatch();
    timer.Start();
    ProgramTask.Start();
    await ProgramTask;
    timer.Stop();

    // Check if there are any collisions after the programs have stopped (ran to completion or collided).
    collision = RDK.Collisions() > 0;

    // Return the execution time if there were no collision, otherwise return -1.
    return collision == true ? -1 : timer.ElapsedMilliseconds;
}
```

Figure 4.12: Scoring a RoboDk program based of if a collision occurred and execution time after execution.

4.5 Intelligent Programming

One of the biggest advantages with the RoboDk API is that more sophisticated and "intelligent" programs can be built for controlling the robots, the interface in this project is an example of this. By scoring different solutions to the desired task on selected parameters like execution time (how long the whole task takes from start to finish), safety (how close the robots get to colliding during execution) and if the task is collision free. The score of each solution can be compared, and an optimal execution strategy can be found. The task of optimizing robot programs is a huge field in its own right, having a myriad of different parameters that could be considered for optimization, such as energy consumption, execution time, wear and tear on the robots. For this project, only execution time will be considered when finding an "optimal" solution. The ability to compare the execution time of each potential solution not only enables the interface to allow the execution of collaborative and complicated tasks easily but also evaluate and select the best solutions as the task is given. To evaluate processes and come up with solutions, a categorization of the requirements based on the nature of the process is needed. This categorization would allow the system to know what sort of functionality (or skills) are needed, and what parameters are important in order to achieve the best results.

4.5.1 Selecting Optimal Execution Strategies

An optimal solution is in the scope of this project, the solution that uses the least amount of time while still achieving the goal of the scenario and being collision free. Different levels of complexity and effectiveness were implemented.

For the delivery scenario, a simple critical region algorithm was implemented (see Figure 4.13). This algorithm does not result in the fastest possible solution, but guarantees that no collisions will occur between the robots.

For the wall building scenario, three different strategies were implemented, all based on first simulating possible solutions before execution. Firstly all solutions would be simulated, and the fastest collision free solution would then be selected, this guarantees that the fastest solution will be found, at the cost of taking a long time if there are a large number of possible solutions.

The second strategy was to apply a solution to the "secretary problem" which is a scenario involving the optimal stopping theorem. This strategy provides a $\frac{1}{e}$ ($\approx 37\%$) probability of picking the best solution by picking the best solution after observing the first $\frac{1}{e}$ solutions [2]. This strategy is most effective with a high number of possible solutions, and if the best solution was discovered in the first 37% it is identical to the first strategy, where all solutions are simulated. This strategy attempts to balance the time spent simulating solutions, and the reliability of the selected solution being the best one, with the drawback of the best solution being selected is not guaranteed.

The final strategy implemented for selecting the optimal execution strategy is to select the first collision free solution discovered. This strategy guarantees the least time spent simulating solutions. However, by selecting the first collision free solution, the probability of the chosen solution being the fastest of all possible solutions is very low. This strategy could be used if the process is short or not to be repeated. The algorithm is described in figures 4.14 and 4.15. The algorithm for building a wall is described in figure 4.16, this algorithm is executed when simulating the solutions in order to find the best score. This is the predefined process of "Simulate solution and determine score" used in figure 4.15.

4.6 Collaboration Scenarios

4.6.1 Physical Implementation

As the robots were in the process of being installed during this project, it was not realistic to reconfigure the robots in such a way that they could effectively attempt the collaboration scenarios described. The main issue to this was that the KUKA robot was to be installed with an optical safety barrier, as well as solid glass walls designed to prevent people from entering the workspace of the robot, this also prevents the UR robot from sharing a workspace with the KUKA robot. However, the methods for connecting to the robots as well as sending commands were implemented in the application. These methods can prove that the application can indeed control the robots concurrently following the planned motions; this was tested in chapter 5.

4.6.2 Simulated Implementation

The application implements the core skills of picking, placing as well as tracking bricks within the workspace of the robots. The passive collaboration scenarios of delivering a brick and building a wall were also implemented in the application. This implementation includes two different ways of planning execution strategies, an algorithm applying a critical region for delivery, as well as pre-execution simulation of the different solutions for building a wall. The building scenario also implements three different strategies for selecting the best scenario. Both C# tasks as well as RoboDk programs were implemented in the building a wall scenario.

4.6.3 Theoretical Implementation

As the implementation of synchronization skills, allowing for responsive programming was deemed too time-consuming to implement during the duration of this project, no form of active collaboration or adjustment of robot movements during the execution were implemented. It is however very likely that this is possible to achieve as described in section 4.7. With a successful implementation of synchronization skills, machine learning could be applied in order to solve the different scenarios without having the robots collide. As the planning algorithms implemented in the application are quite conservative, that is to say; they attempt to guarantee a successful execution before the robots take any action, they would not be suitable for scenarios requiring synchronization skills. Responsive programming would allow for eager algorithms to be implemented, where the robots would start moving towards their goal positions right as the command was issued, while the system constantly monitors the environment for situations that requires the robots to be redirected. Eager algorithms, capable of adjusting the solutions during execution would allow active collaboration scenarios to be implemented.

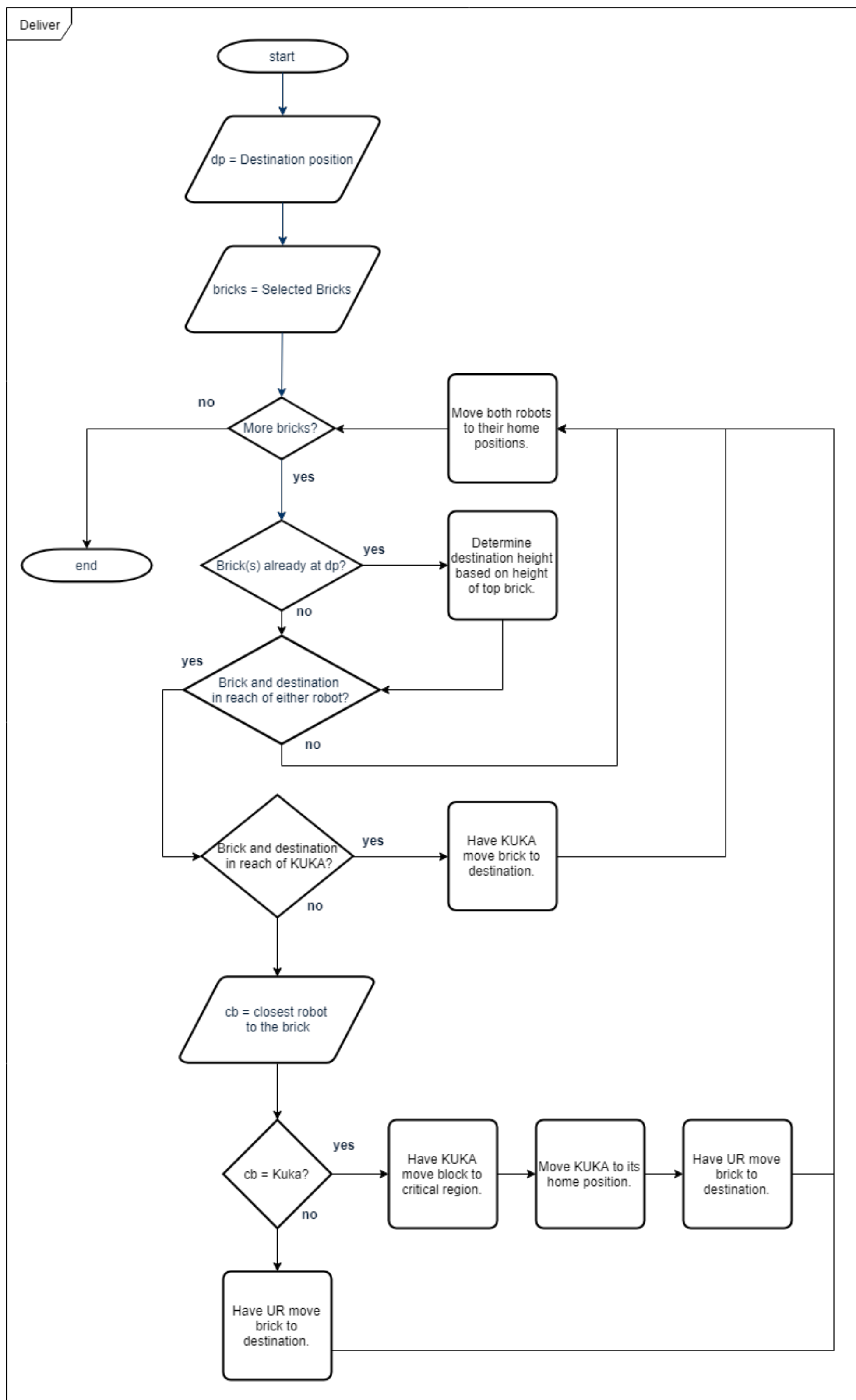


Figure 4.13: The algorithm for delivering a brick.

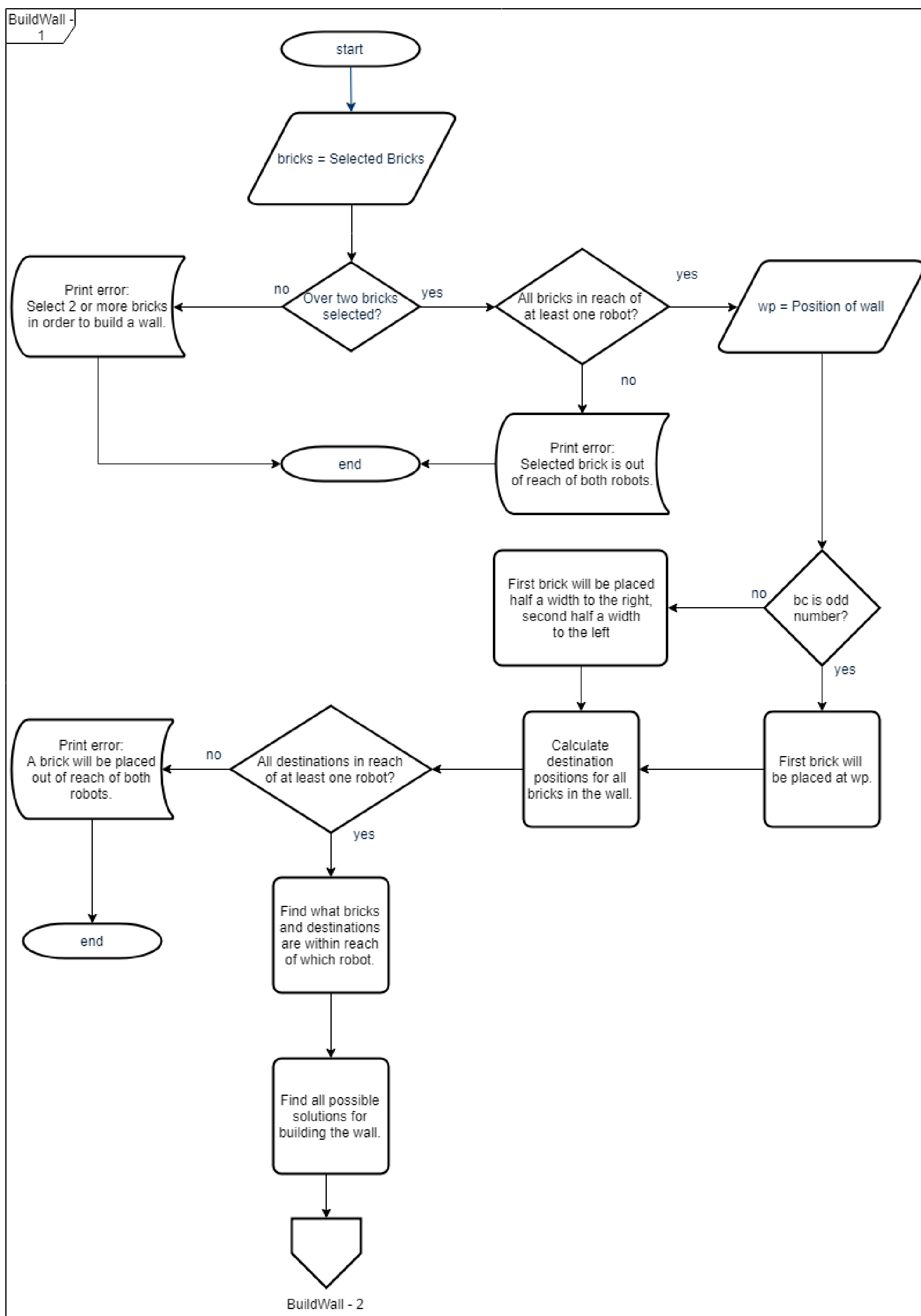


Figure 4.14: The algorithm for finding the best solution for building a wall (page 1 of 2).

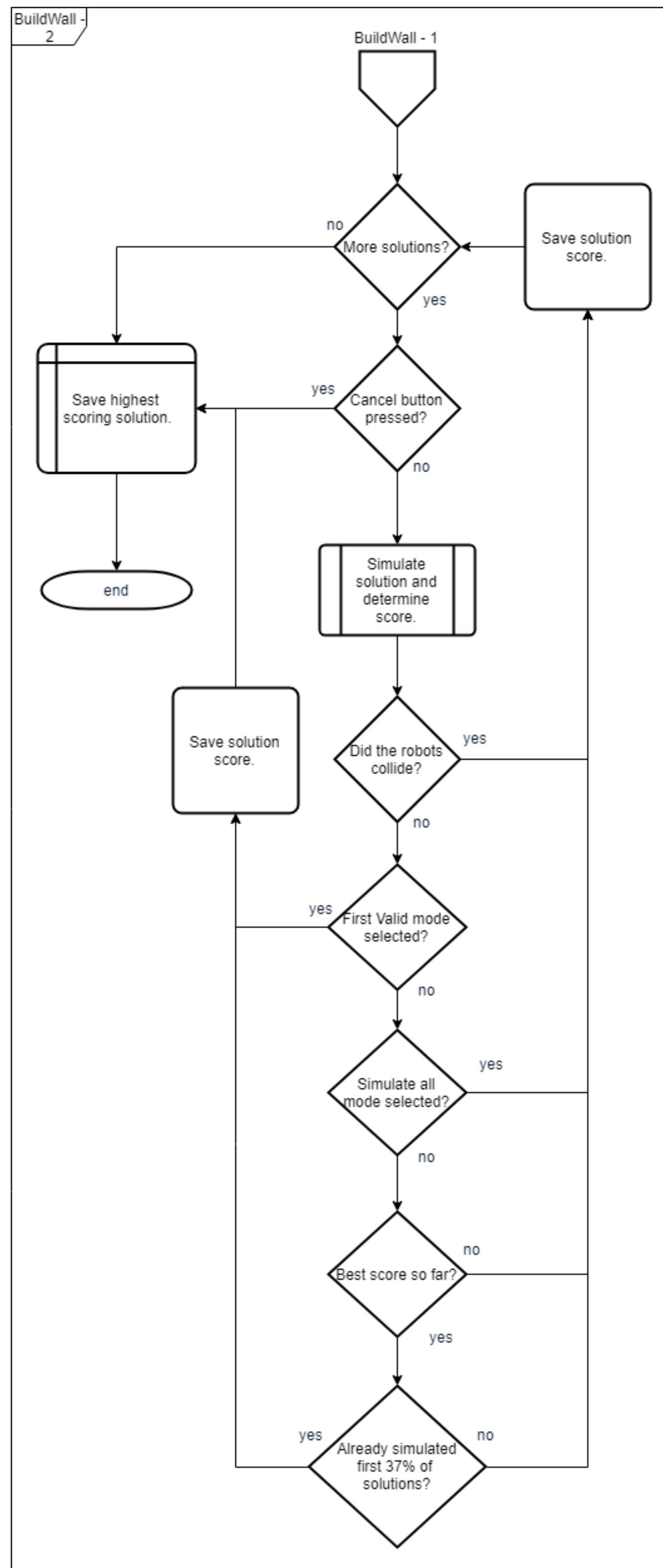


Figure 4.15: The algorithm for finding the best solution for building a wall (page 2 of 2).

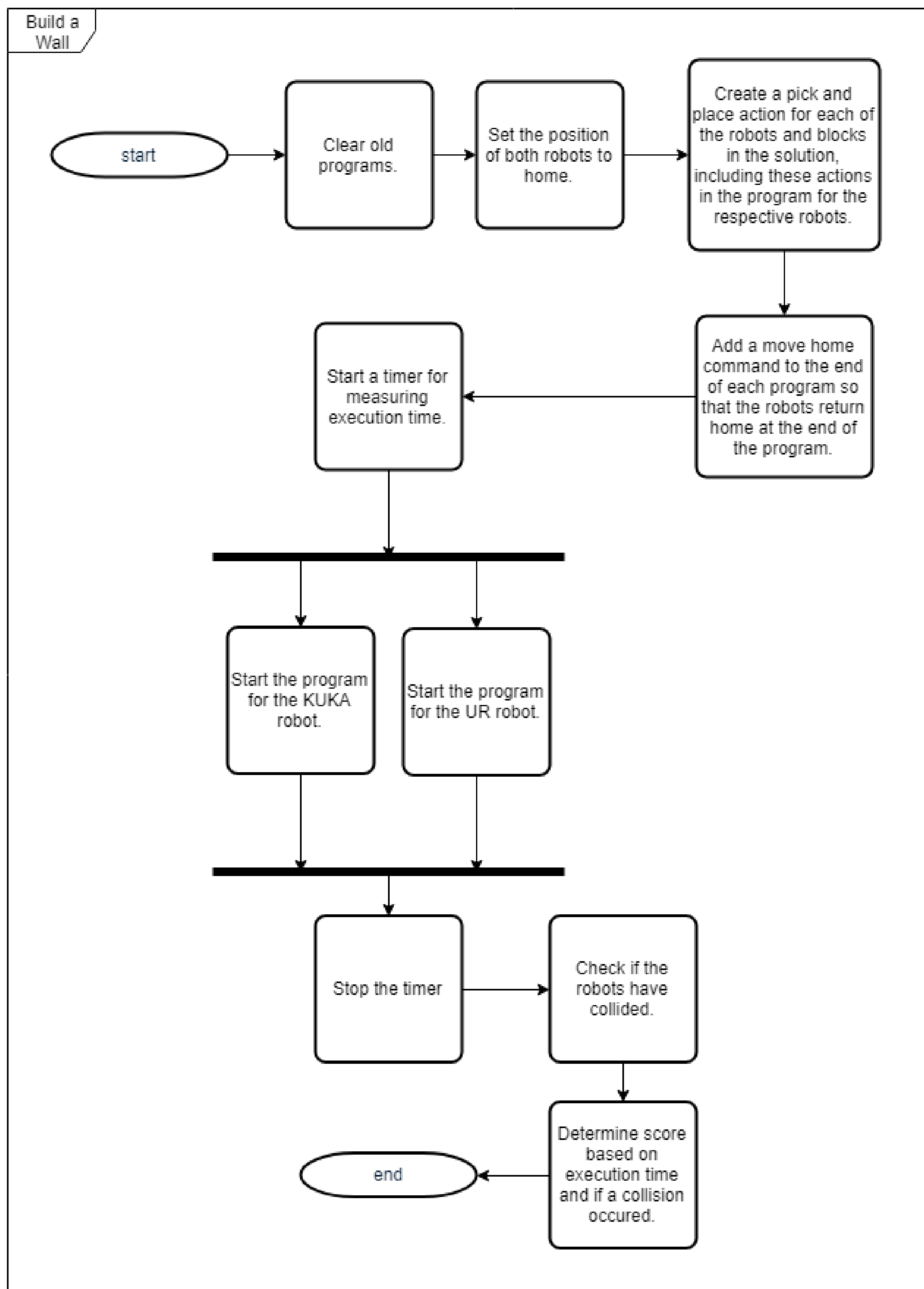


Figure 4.16: The algorithm for building a wall. This is an extension of the "Simulate solution and determine score" process used in figure 4.15.

4.7 Responsive Programming

Collisions and other unwanted events could be prevented by running a simulation separate from the execution of a program on the physical robots a little bit in advance, this ”digital twin” would then run in a slightly forward-shifted synchronized execution with the execution of the program on the physical robots. This digital twin would allow the system to foresee problems by monitoring the state of the simulation. When a problem is detected there would be an amount of time where the digital twin system can attempt to find a solution to the problem and convey this solution to the physical robots before the real robots encounter the problem. Depending on how far in advance the simulation is running, the system would have more or less time to react. A short delay would result in a short time to discover and convey a discovered solution whereas a long delay would increase the risk of something happening that causes a problem to occur in the period between the two executions, making the system unable to react correctly.

Problems

As RoboDk is responsible for both controlling the robots, as well as simulating the movement of the robots, it would be necessary to have two separate instances of RoboDk running in parallel in order to achieve a digital twin. Unfortunately, RoboDk cannot start as two separate instances on one computer at the same time. When RoboDk is running, it prevents new instances from being started. A possible solution to this would be to load two separate ”stations” within the same instance of RoboDk, that is to load two versions of the simulated robot lab at the same time within the same instance of RoboDk. This solution proved to be very challenging as the API uses names of items in order to reference objects within RoboDk and no way of selecting what station is being searched for the named item. Another possible solution would be to connect to an instance of RoboDk on a separate computer or virtual machine in order to use that instance for simulation while the original computer is responsible for controlling the robots. This solution proved to not be realistically achievable for this project as the code required to make this work needs to be a lot more complicated, where delay in the communication between the computers is a significant factor.

Chapter 5

Testing

The goal of the testing is to evaluate the effectiveness of the solutions to the technical and collaborative challenges described in chapter 3 and 4.

5.1 Simulation

Firstly, the core skills for the implemented scenarios were tested. Both robots in RoboDk could successfully move to the designated bricks, pick them up, and move them to the given or calculated destination position. After the core pick and place skills were confirmed to be working, the first collaborative scenario of delivering bricks from within the KUKA robots reach to a destination outside of its reach through the critical region algorithm (see Figure 4.13). The test resulted in both bricks successfully being stacked on the destination position. For the test, two bricks were added to the simulation at positions 25,25 and 150,150. This emulates the bricks being seen by a 3D camera. Both bricks were then selected for delivery to the position 700,700, and the "Deliver" button was pressed, starting the algorithm. Key moments in the test are shown in figures 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6.

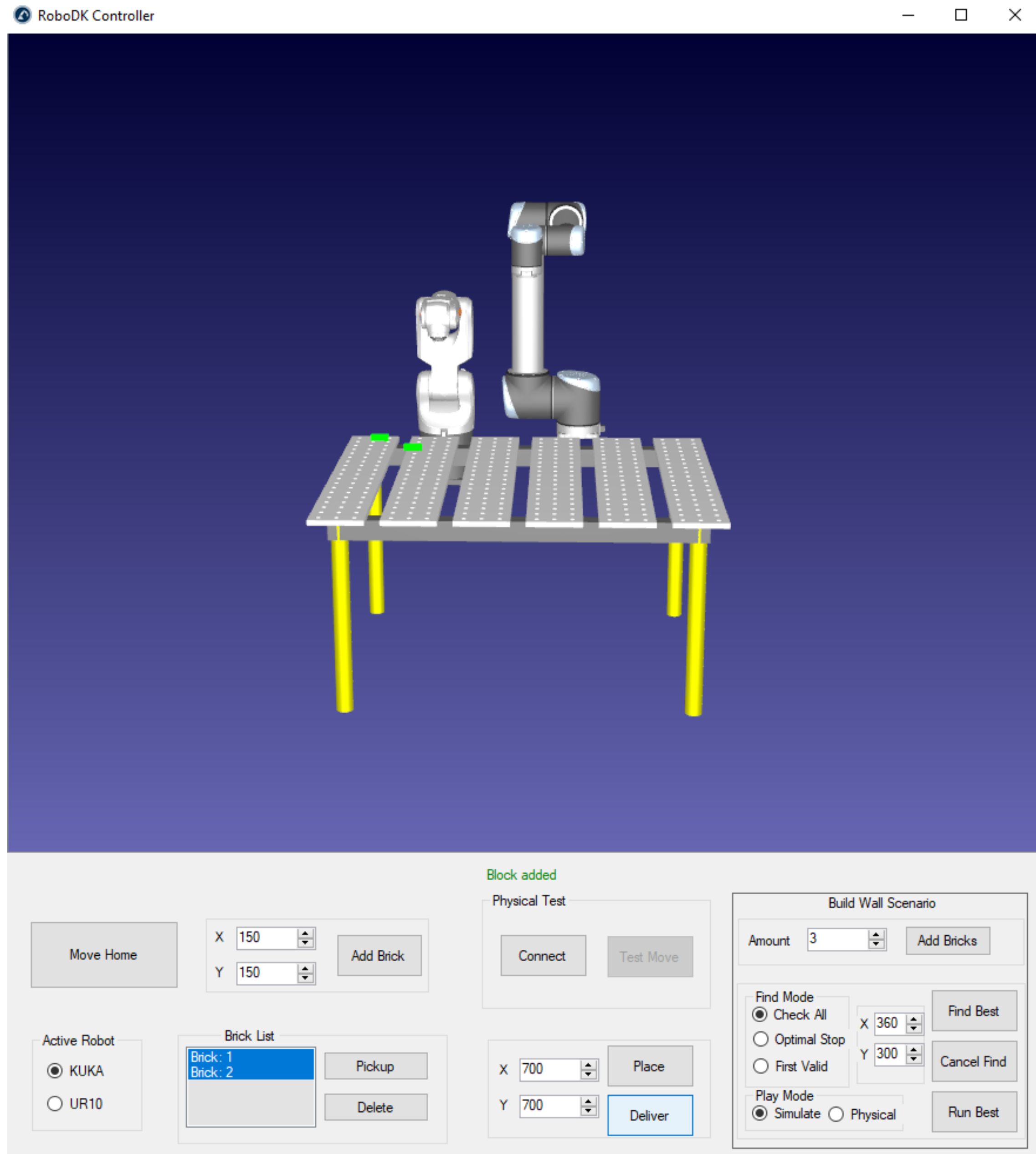


Figure 5.1: Testing the delivery scenario within the simulation. - Starting the test.

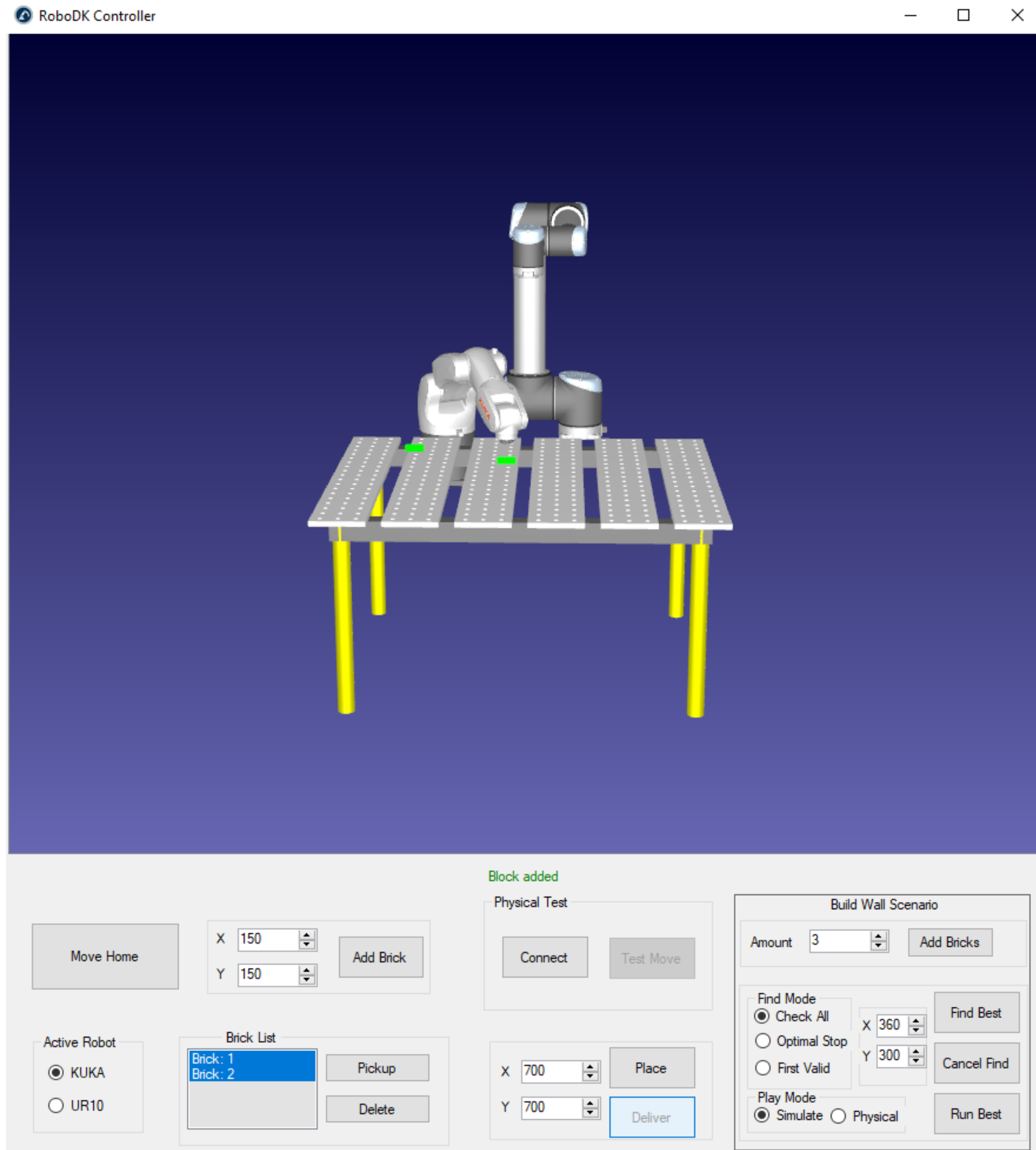


Figure 5.2: Testing the delivery scenario within the simulation. - KUKA places the first brick in the critical region before returning to its home position.

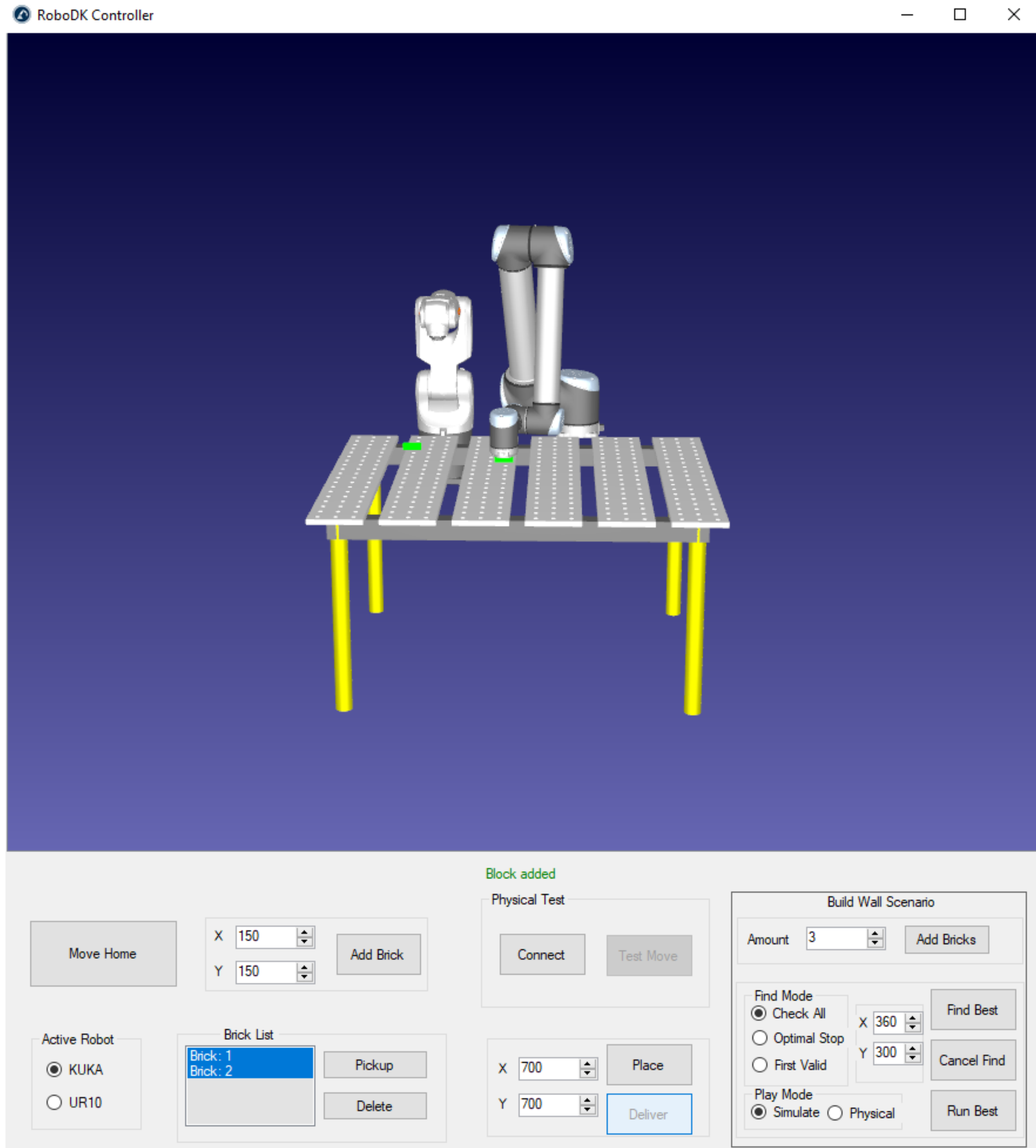


Figure 5.3: Testing the delivery scenario within the simulation. - After the KUKA robot has returned to its home position, the UR robot picks up the brick from the critical region.

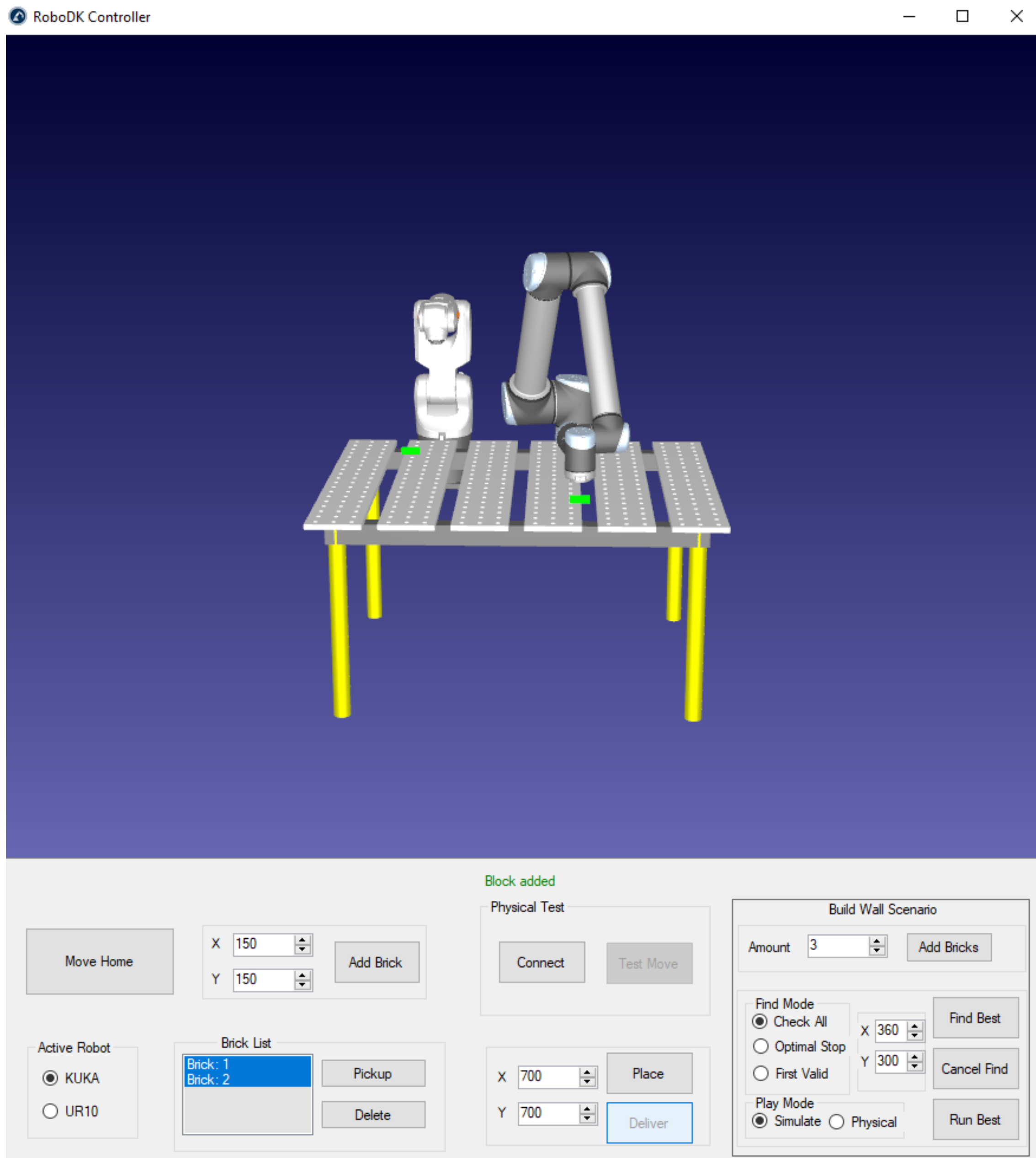


Figure 5.4: Testing the delivery scenario within the simulation. - The UR robot places the first brick on the designated position.

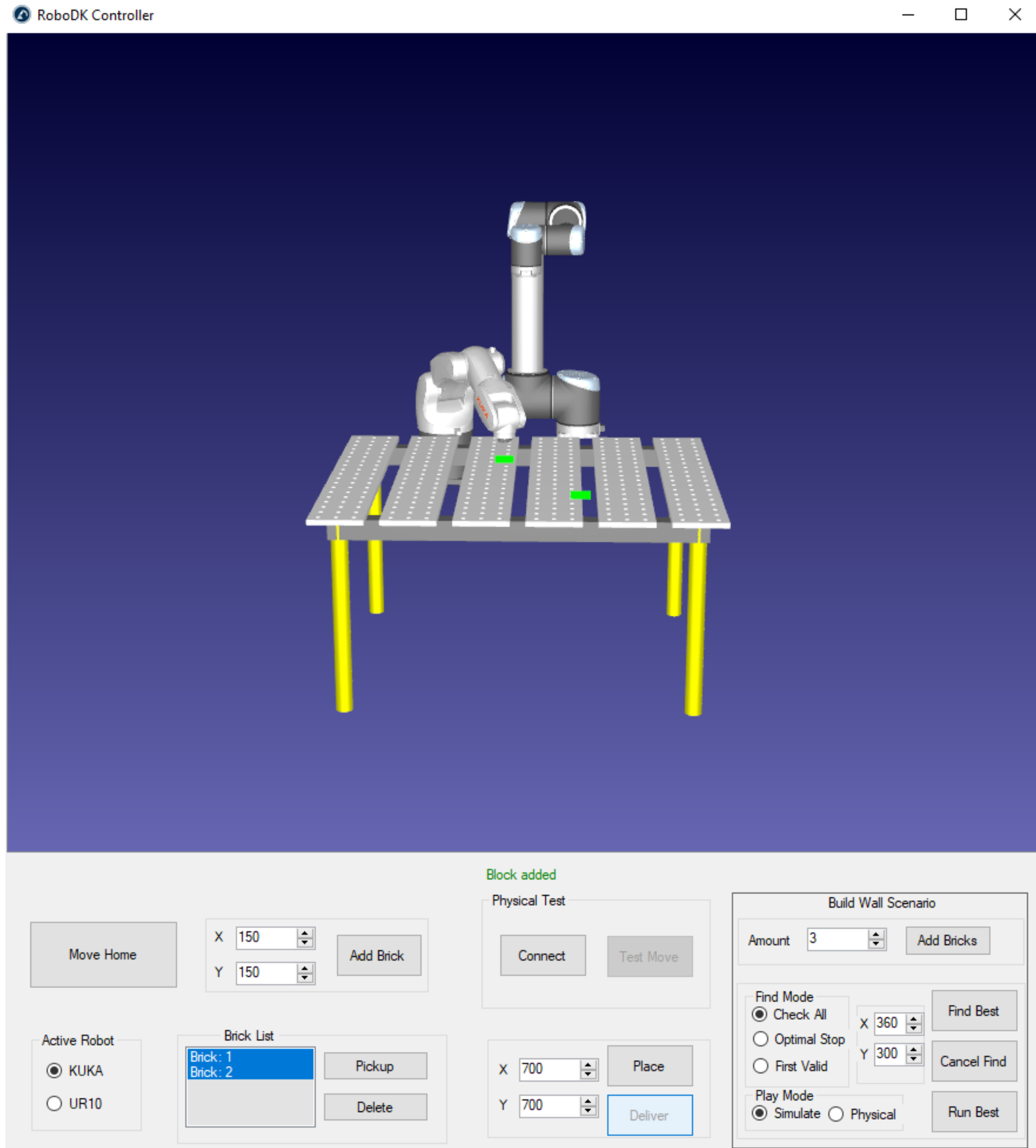


Figure 5.5: Testing the delivery scenario within the simulation. - After the UR robot has returned to the home position, the KUKA robot places the second brick within the critical region.

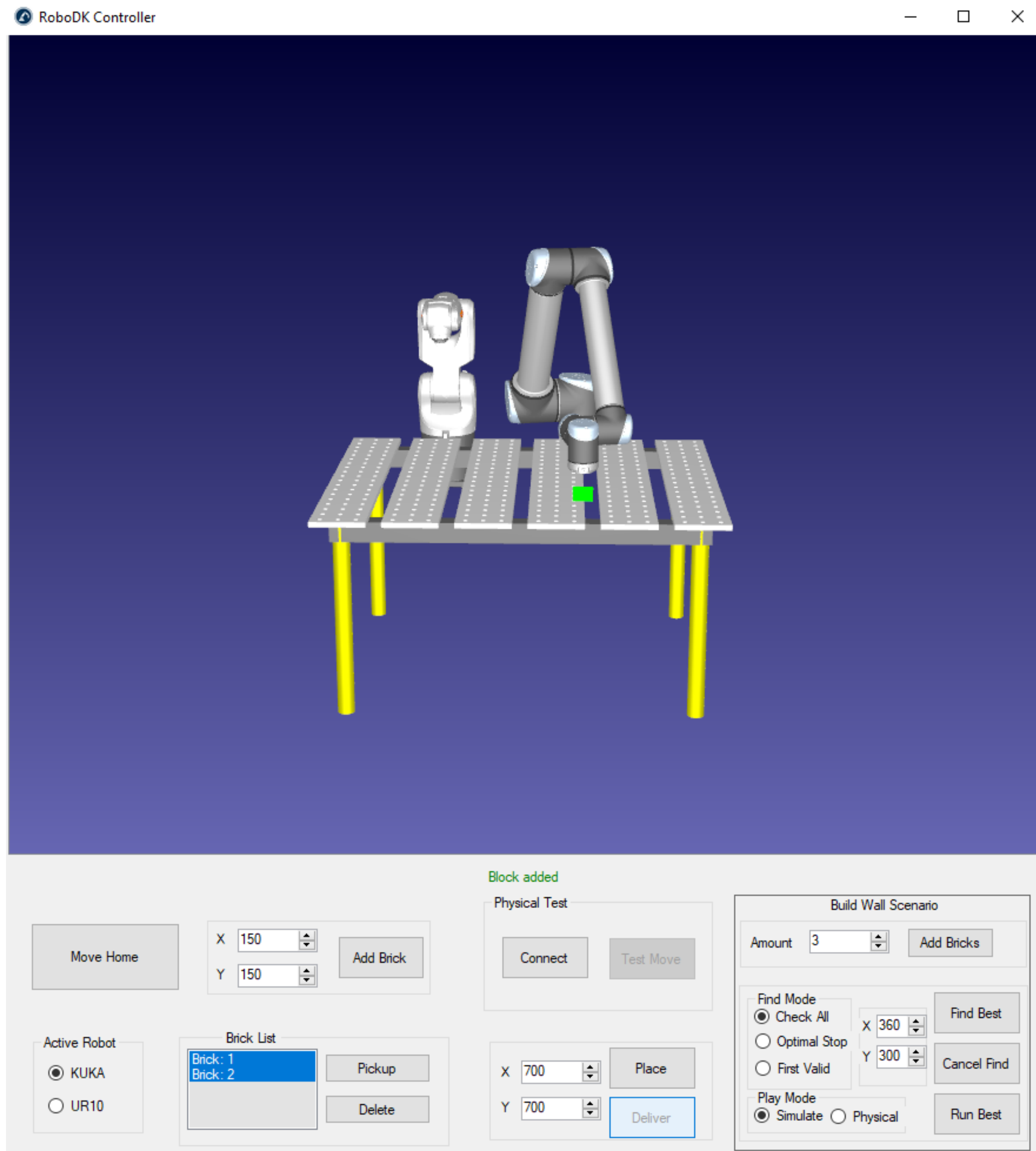


Figure 5.6: Testing the delivery scenario within the simulation. - After the KUKA robot has returned to its home position, the UR robot picks up the brick from the critical region and places it on the first brick on the destination position before returning home.

The scenario for building a wall by firstly simulating all possible solutions then selecting the fastest one (see Figure 4.14 and 4.15) was also tested in the simulation. The test resulted in all possible solutions being simulated, then the fastest one without collisions were selected as the best one. For this test, four bricks were added to the simulation through the "Add Bricks" button with the amount of 4 specified. The center position of the wall was designated to be position 360,300, and this ensures that some of the bricks would be within reach of the KUKA robot so that both robots would work concurrently to accomplish the task. The test was then started by pressing the "Find Best" button. After the fastest solution was found, it was executed by pressing the "Play Best" button. The simulations are run using RoboDk programs, allowing for optimal movement of the robots with regards to the next movements in the program. For demonstration purposes, C# tasks are utilized when executing the fastest solution such that the robots can be shown moving the bricks with the "attach" method. Key moments in the test are shown in figures 5.7, 5.8, 5.9, 5.10 and 5.11 (planned motion of the robots are shown with yellow lines).

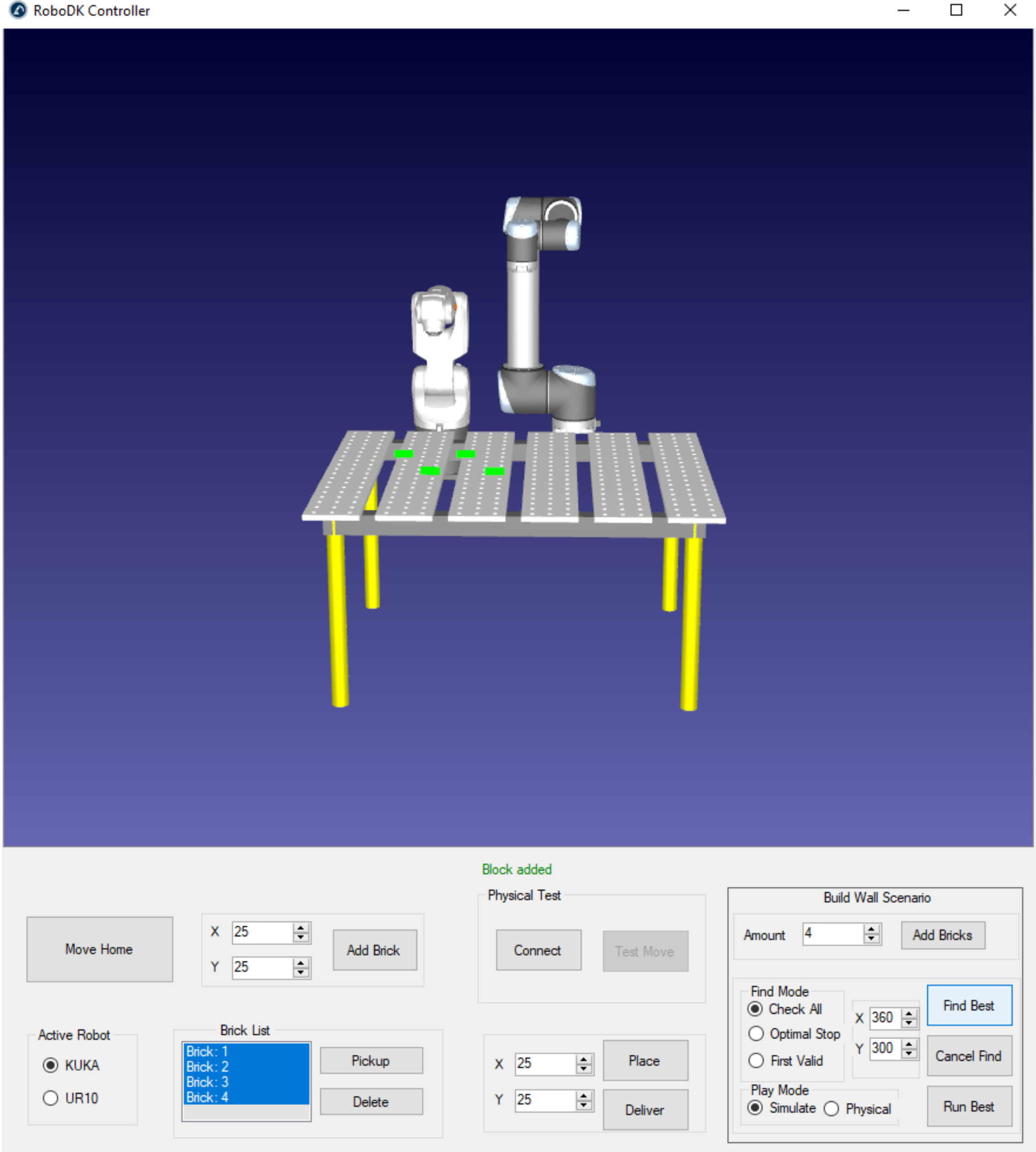


Figure 5.7: Testing the build wall scenario within the simulation. - Starting the test.

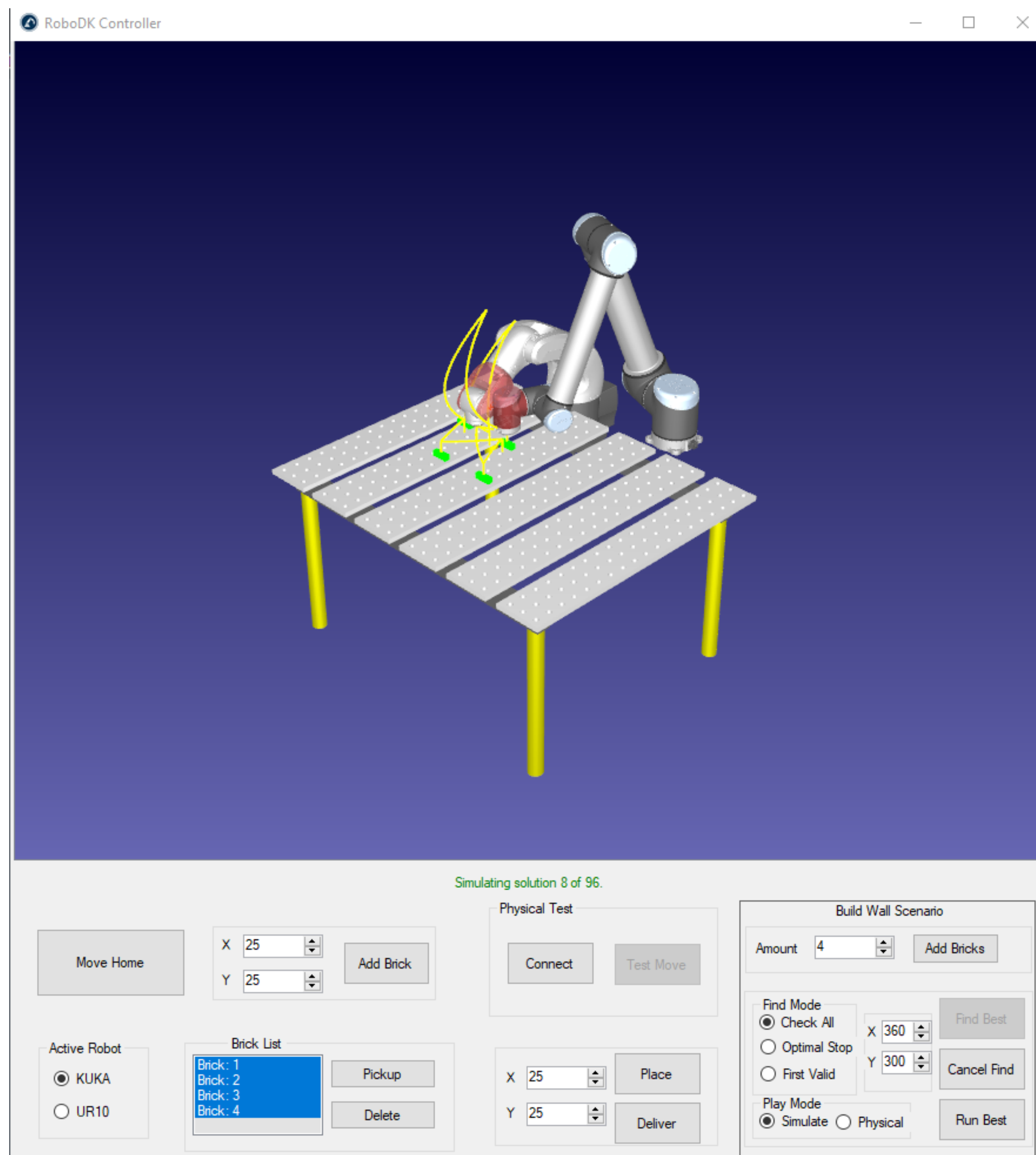


Figure 5.8: Testing the build wall scenario within the simulation. - When a collision is detected in the simulation, the simulation is stopped and the next solution is tried.

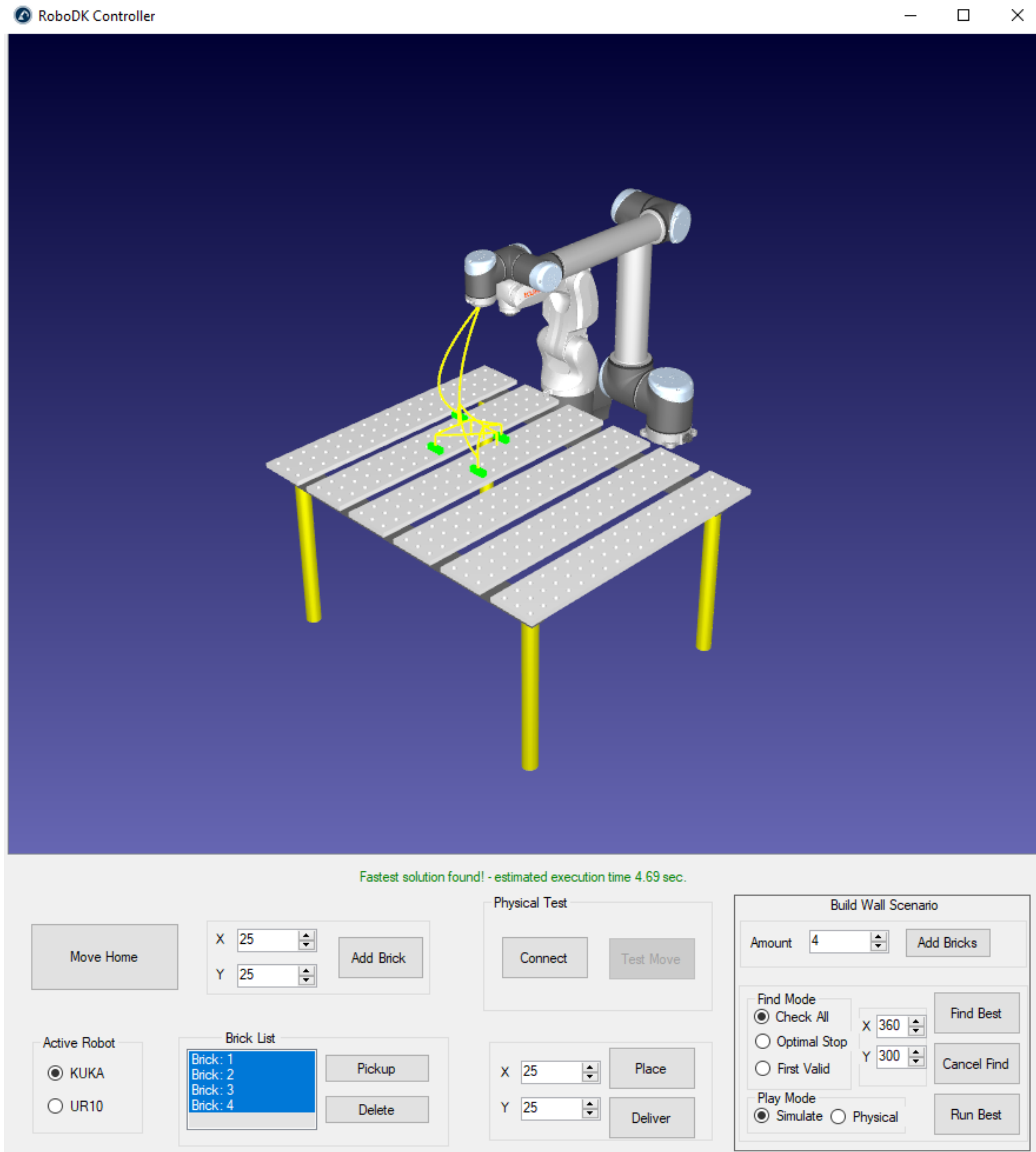


Figure 5.9: Testing the build wall scenario within the simulation. - After all potential solutions were tried, the fastest one was selected as the best one. (Green status text: "Fastest solution found! - estimated execution time 4.69 sec")

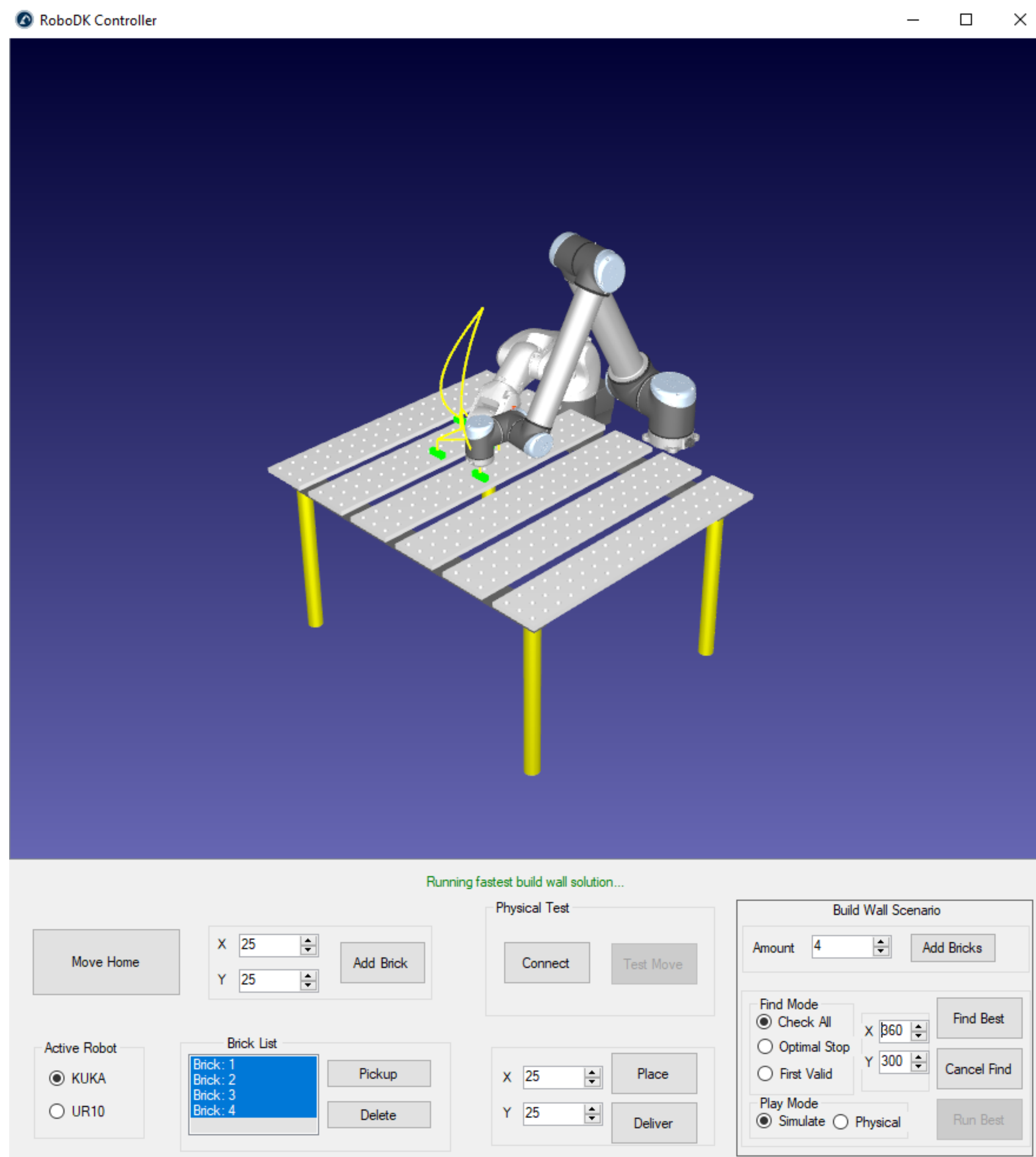


Figure 5.10: Testing the build wall scenario within the simulation. - Running the fastest solution, both robots moving bricks concurrently.

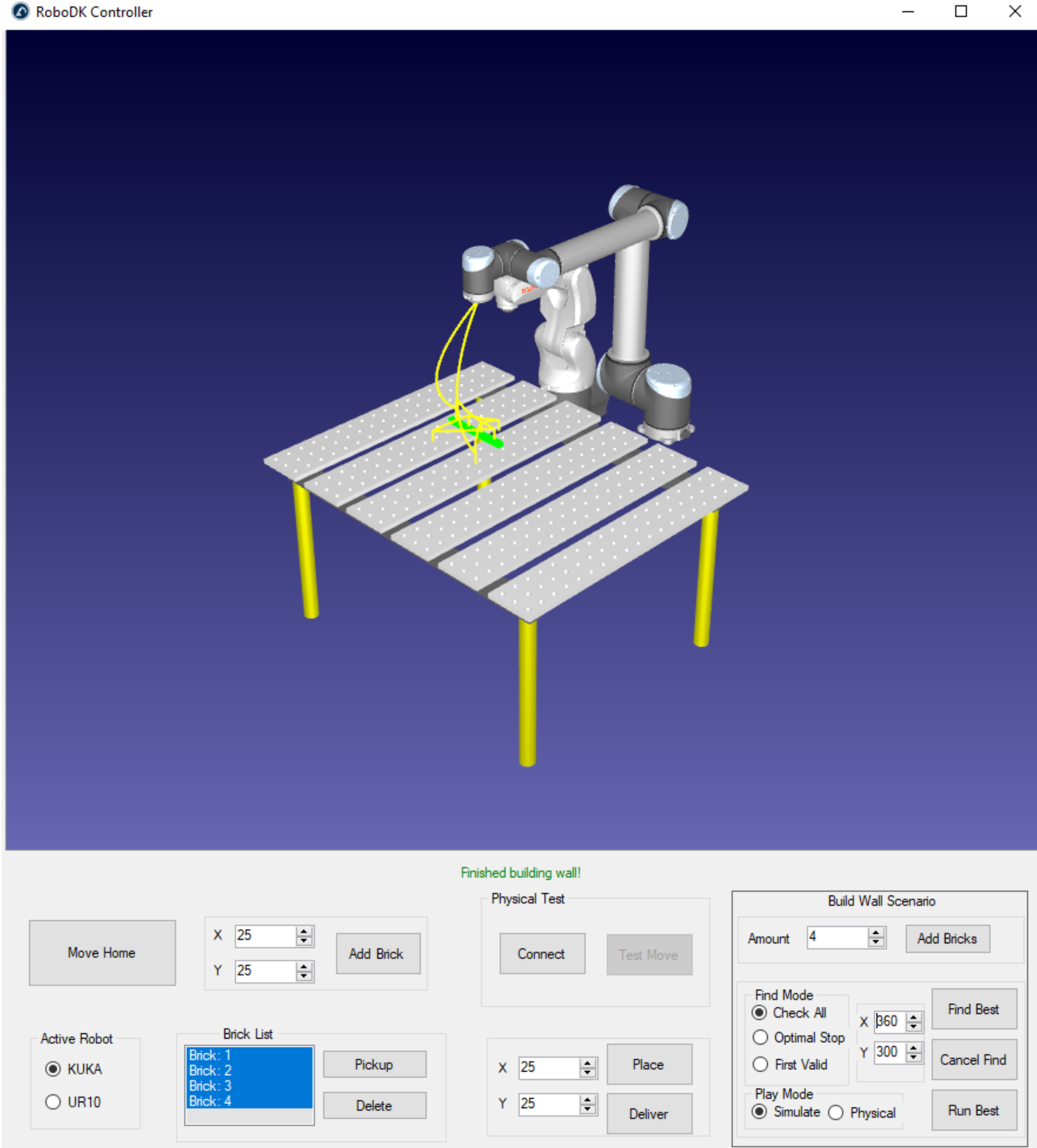


Figure 5.11: Testing the build wall scenario within the simulation. - Test finished, all bricks placed on their respective destinations.

5.2 Testing On the Physical Robots

The physical robots could not be arranged in a suitable configuration for performing the collaborative scenarios as the robotics lab was still being made ready for use during the time of this project. The KUKA robot is enclosed by physical barriers, and an optical barrier in order to provide the necessary safety while being operated by students utilizing the lab. As having the robots collaborate would require this barrier to be removed, a barrier enclosing both robots would have to be used as a replacement. This sort of barrier could not realistically be put in place during the duration of this project. The robots were however fully functional, and proof of concept testing was successful. The robots were connected to RoboDk at the same time, and both robots responded to commands sent through RoboDk, and the API. With the API, concurrent execution of commands was accomplished. The configuration of the robots during testing is shown in figure 5.12.

5.2.1 Procedure

In order to connect the robots to the application, several steps are necessary.

1. The IP addresses of both robot controller must be known. At Høgskolen i Østfold's robot lab, they should be as specified in figure 5.12.
2. The firewall on the computer running the application must be shut off; alternatively the individual robot drivers can be allowed through the firewall manually (see RoboDk documentation [12]).
3. The robot controllers must be reachable from the computer running the application. In the robot lab this is ensured by having the controllers and computer on the same local network (see Figure 5.12).
4. The robots in the RoboDk simulation must be connected to the corresponding robot controllers through the API (shown in Figure 5.13).
5. After the robots are connected, RoboDk needs to be in live programming mode (see RoboDk documentation [12] and Figure 5.13).
6. The robots can now be controlled by executing the movement or program commands from the API.

5.2.2 Testing Concurrent Movement Through the API

After ensuring RoboDk will be able to connect to both robot controllers, the application successfully connected to both robots by running the ConnectToRobots method (see Figure 5.13) after the "Connect" button was pressed, as shown in figure 5.14. After a connection is established, the "Test Move" button is enabled, when pressed, the TestMovePhysical method is executed (see Figure 5.15). This execution resulted in a successful demonstration of concurrent movement of the robots through the use of the API, with movement commands being executed through the use of C# tasks. During the test, both robots started in their home positions, and then both robots rotated 45 degrees to the right relative to their home position. The robots then rotated 45 degrees left of their starting position before returning to their home position (see Figures 5.16, 5.17, 5.18 and 5.19).

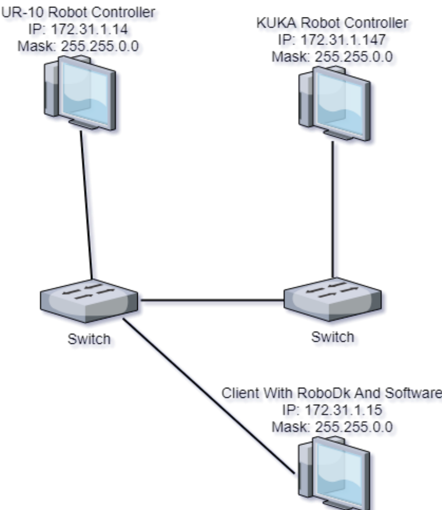


Figure 5.12: Network diagram of the system during physical testing.

```
private void ConnectToRobots()
{
    try
    {
        // Variables to hold the result of the attempted connections, True if connected successfully, False if connection failed.
        bool KukaConnected = false;
        bool UrConnected = false;

        // Attempt to connect to the KUKA robot controller.
        KukaConnected = KUKA_ROBOT.Connect(KukaIp);
        UrConnected = UR_ROBOT.Connect(UrIp);

        // Sometimes it takes two connectionattempts in order to connect, if it fails after the second try, we give up.
        if (KukaConnected == false)
        {
            KukaConnected = KUKA_ROBOT.Connect(KukaIp);
        }
        if (UrConnected == false)
        {
            UrConnected = UR_ROBOT.Connect(UrIp);
        }

        // Display the outcome of the connection attempt in the program.
        if (KukaConnected == true && UrConnected == true)
        {
            messageBox.ForeColor = MSG_SUCCESS_COLOR;
            messageBox.Text = "Successfully connected to both robot controlles!";

            // Set RoboDk to execute the commands on the robots (should be activated automatically after using Robot.Connect()).
            RDK.setRunMode(RoboDK.RUNMODE_RUN_ROBOT);

            //Get the current position of the robots.
            KUKA_ROBOT.setJoints(KUKA_ROBOT.Joints());
            UR_ROBOT.setJoints(UR_ROBOT.Joints());

            KUKA_ROBOT.setPoseFrame(KUKA_ROBOT.PoseFrame());
            KUKA_ROBOT.setPoseTool(KUKA_ROBOT.PoseTool());

            // Enable the test move button.
            BtnTestMove.Enabled = true;
        }
        else
        {
            RDK.setRunMode(RoboDK.RUNMODE_SIMULATE);

            messageBox.ForeColor = MSG_ERROR_COLOR;
            messageBox.Text = "Error: Failed to connect to one or both of the robot controlles!";
            BtnTestMove.Enabled = false;
        }
    }
    catch (Exception ex)
    {
        RDK.setRunMode(RoboDK.RUNMODE_SIMULATE);

        messageBox.ForeColor = MSG_ERROR_COLOR;
        messageBox.Text = "Error: Exception occured while attempting to connecting to the robot controlles! Ex: " + ex;
        BtnTestMove.Enabled = false;
    }
}
```

Figure 5.13: Connecting to the robot controllers from the API.

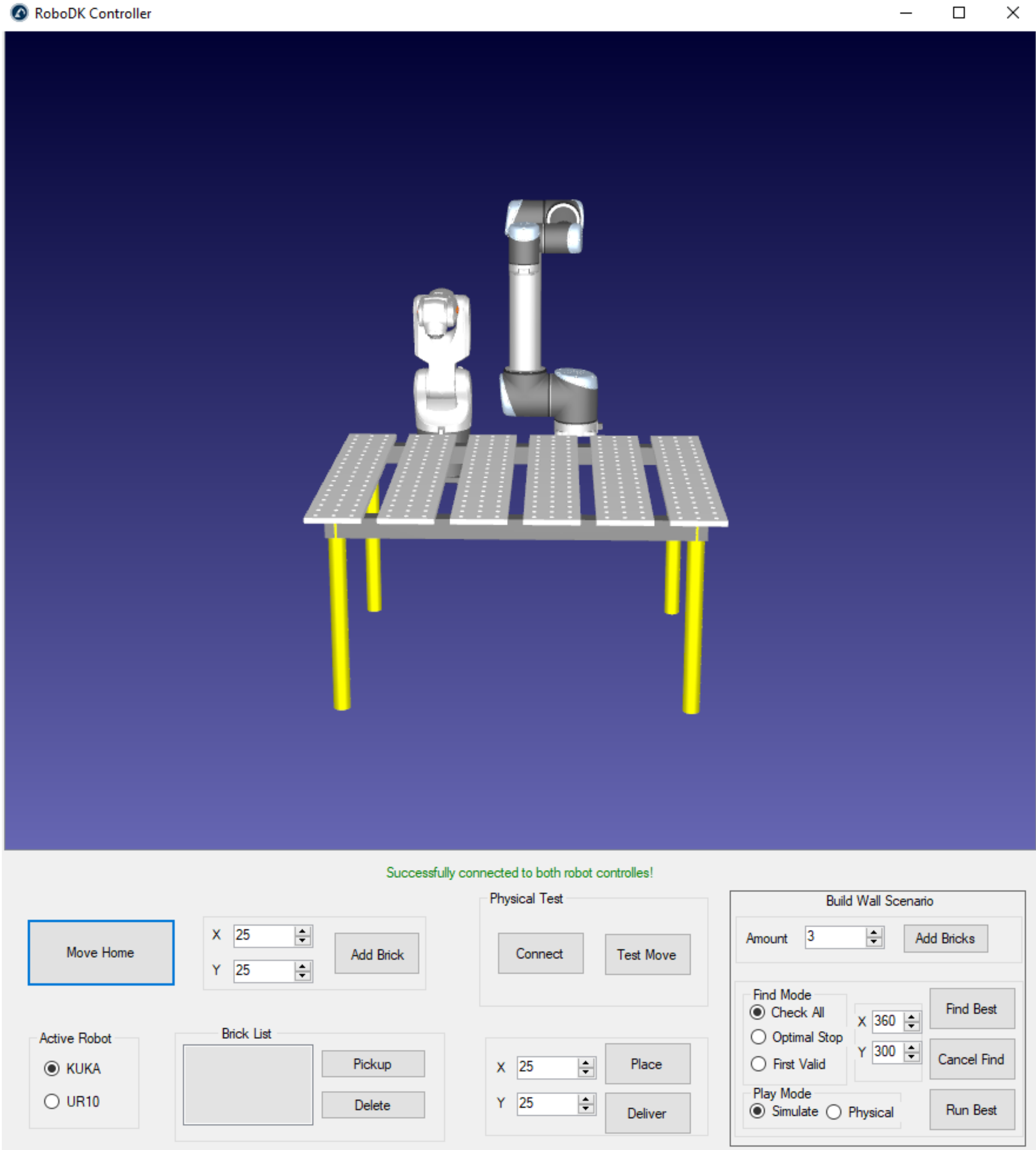


Figure 5.14: The application after successfully connecting to the robot controllers.

```
private void TestMovePhysical()
{
    // Make sure RoboDk is in run on robot mode.
    RDK.setRunMode(RoboDK.RUNMODE_RUN_ROBOT);

    // Create positions for the UR robot with 45 degrees rotation from the home position.
    var UrJointsRight = new double[6];
    var UrJointsLeft = new double[6];
    HOME_UR.CopyTo(UrJointsRight, 0);
    HOME_UR.CopyTo(UrJointsLeft, 0);
    UrJointsRight[0] = 45;
    UrJointsLeft[0] = -45;

    // Create positions for the KUKA robot with 45 degrees rotation from the home position.
    var KukaJointsRight = new double[6];
    var KukaJointsLeft = new double[6];
    HOME_KUKA.CopyTo(KukaJointsRight, 0);
    HOME_KUKA.CopyTo(KukaJointsLeft, 0);
    KukaJointsRight[0] = 45;
    KukaJointsLeft[0] = -45;

    // Create tasks for moving both robots.
    var URMoveTask = Task.Run(() =>
    {
        UR_ROBOT.MoveJ(HOME_UR);
        UR_ROBOT.MoveJ(UrJointsRight);
        UR_ROBOT.MoveJ(UrJointsLeft);
        UR_ROBOT.MoveJ(HOME_UR);
    });
    var KukaMoveTask = Task.Run(() =>
    {
        KUKA_ROBOT.MoveJ(HOME_KUKA);
        KUKA_ROBOT.MoveJ(KukaJointsRight);
        KUKA_ROBOT.MoveJ(KukaJointsLeft);
        KUKA_ROBOT.MoveJ(HOME_KUKA);
    });

    // Start both movement taks simultaneously.
    KukaMoveTask.Wait();
    URMoveTask.Wait();
}
```

Figure 5.15: The method for testing concurrent movement on the physical robots (C# tasks).

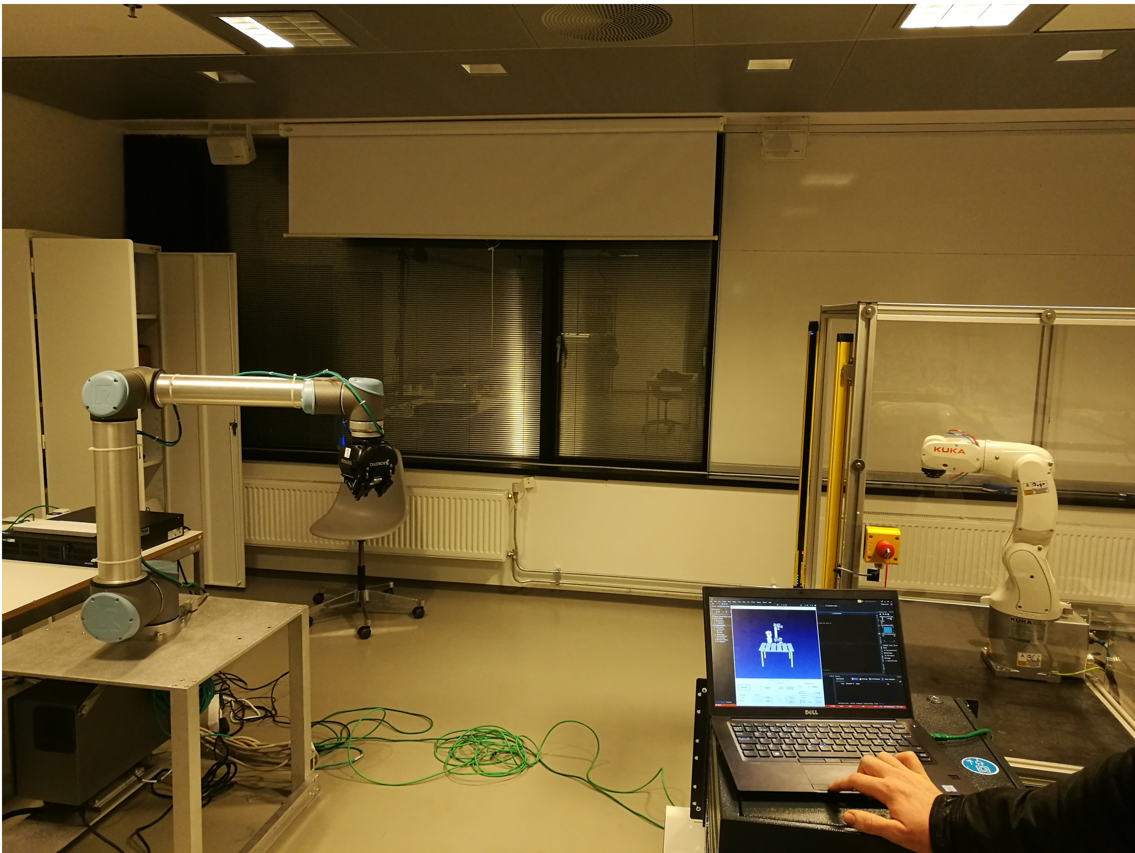


Figure 5.16: Testing concurrent movement on the physical robots - Starting the test.

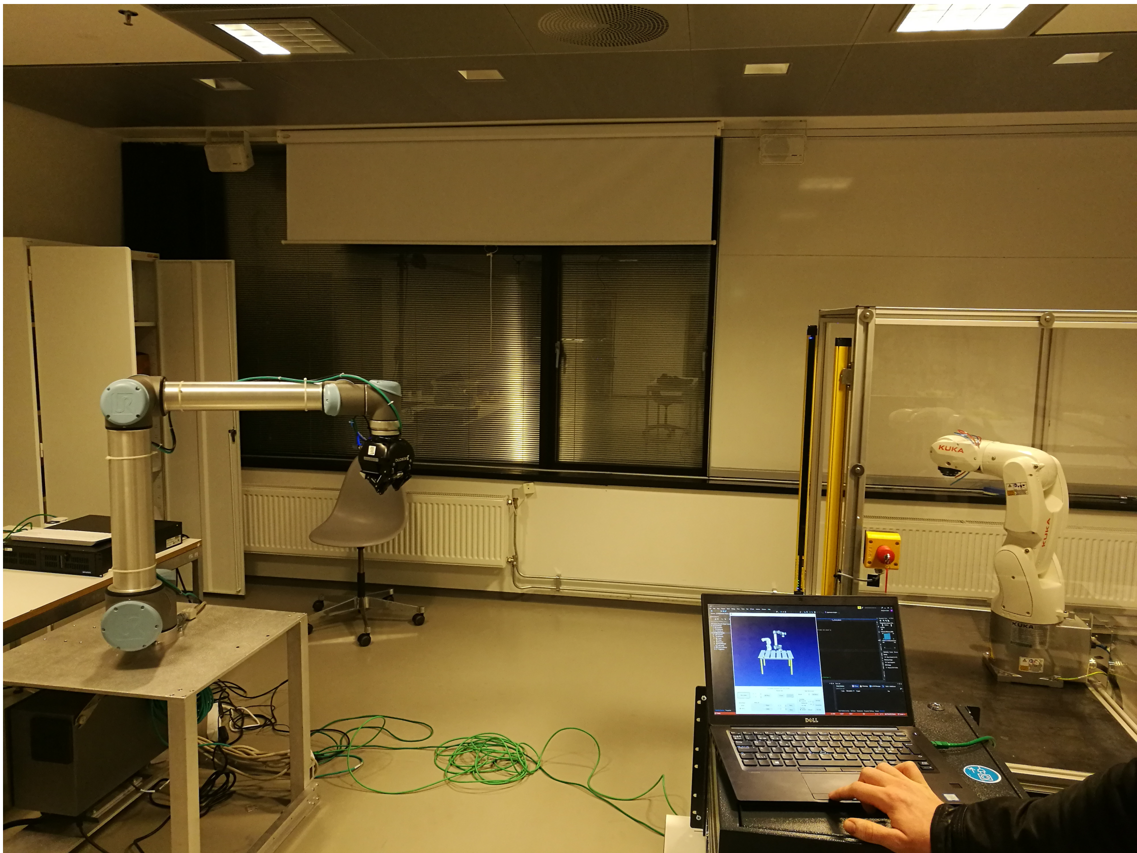


Figure 5.17: Testing concurrent movement on the physical robots - KUKA reached first position.

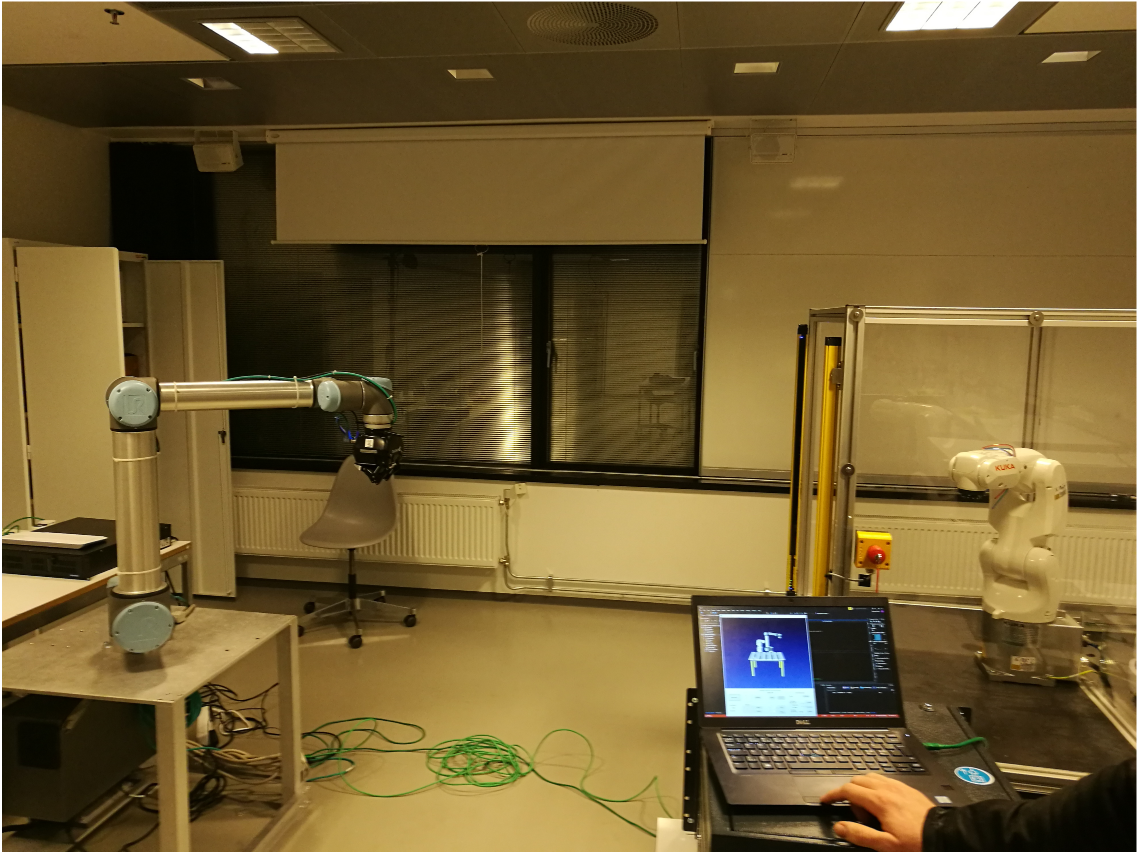


Figure 5.18: Testing concurrent movement on the physical robots - KUKA reached second position. UR moving away from first position.

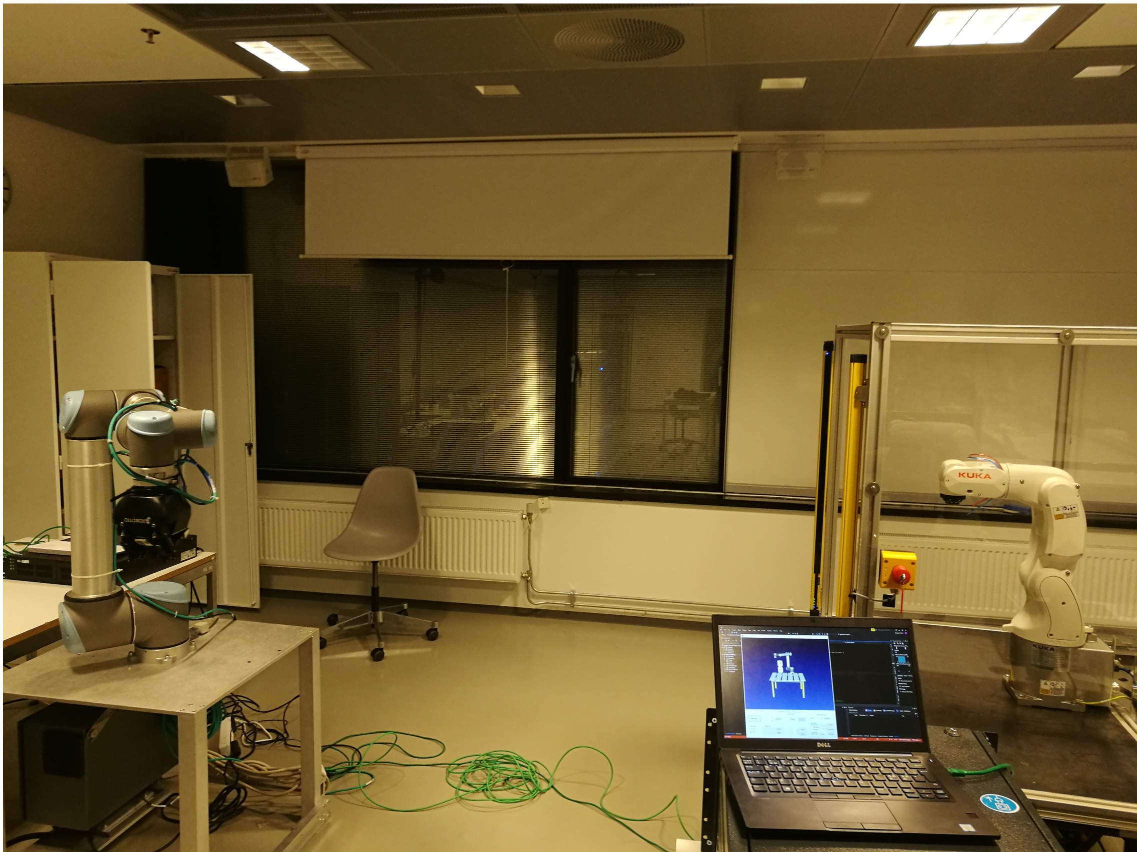


Figure 5.19: Testing concurrent movement on the physical robots - UR reached second position. KUKA returned to home position.

Chapter 6

Discussion

The project sought to answer two problems:

- How can RoboDk, and its API be used to implement collaboration between the two robots at Høgskolen i Østfold's robotics lab?
- How can different collaborative scenarios be classified based on the requirements necessary in order to accomplish said scenario?

The first problem was answered through the implementation of the application. The second problem was answered through breaking scenarios down into their required skills.

6.1 The Application

In summary, a Windows Forms application was developed, concurrent movement of the robots was achieved by utilizing both C# tasks as well as RoboDk programs through the RoboDk API. Different methods for finding solutions to the given scenarios were implemented, and several strategies for finding the fastest solution were implemented and tested. As a proof of concept, the application was a success, showing how RoboDk and its API can be used to implement collaboration between the two robots at Høgskolen i Østfold's robotics lab. Concurrent control of the physical robots was also performed, proving that this kind of implementation would indeed allow the robots to follow the execution plans decided by the application.

Unfortunately, responsive programming was not implemented due to time restrictions. Based on experience gained during the development of this project, it is very likely that responsive programming could be implemented. By performing more experimentation with multiple instances of RoboDk, perhaps running one implementation on a separate machine, the real-time monitoring required for responsive programming could be achieved.

Only passive collaboration scenarios were implemented due to the time required for the more complex active collaboration scenarios not fitting within the project's time-frame. The reason that implementing active collaboration is so complicated, is the requirement of responsive programming. For the implementation of the scenario, "Building a wall," simulations use programs, while running the best solution uses tasks, causing the possibility of the execution of the fastest solution on the robots not being identical to the simulation. This possible discrepancy should be corrected by implementing a real 3D camera for tracking bricks or somehow making the attach method work with programs if the application were to be used with the physical robots to perform collaborative scenarios.

6.1.1 Problems, Challenges, and Solutions

Different challenges, problems, and solutions encountered during the project is shown in figure 6.1.

Figure 6.1: A mind-map showing the different problems and challenges encountered during the project.

Visualization

The implementation of an application that could control the robots through the RoboDk API presented a large number of problems and challenges. The first challenge was the task of visualizing the robots to the user through the application. RoboDk already did represent the robots visually in its simulation and was integrated into the application. This integration was accomplished by utilizing methods available on Windows as described in the RoboDk C# example (available at [15]). This solution was the easiest to implement, as well as giving the most accurate visual representation.

The next challenge was the implementation of a simulated 3D camera, as the physical robot lab does contain a 3D camera capable of tracking objects in real time within a three-dimensional space, but the camera and robots are not set up to function collaboratively. Because the real camera could not be utilized for this reason, a way to simulate input from the camera had to be implemented. For this purpose, a way of manually adding bricks to the workspace was implemented. Both single bricks could be added by specifying the desired coordinates of the new brick, as well as a way to quickly add multiple bricks for use with the building a wall scenario was implemented. This implementation emulates how the actual camera would be implemented, when new bricks are discovered, they would have to be added to RoboDk, the camera would then track movements.

As the real camera could not be utilized for this project, a way of tracking the bricks as the robots were moving them was necessary. With a real camera, the location of the bricks would be updated in real time as the camera saw the bricks being moved by the robots. In order to emulate this, the "Attach" method had to be implemented from the Python API. This method allows the robots to simulate grabbing onto and to let go of the bricks by attaching the bricks to the simulated robots. This method only functions as a simulation and would have to be altered to allow the physical robots to grab onto bricks with their grippers.

Concurrent Movement

The next major challenge was enabling the concurrent movement of the robots. As the application code is executed sequentially, a way of allowing commands to be executed concurrently on the robots had to be implemented. For this purpose, two different solutions were implemented. By utilizing the "NewLink" method included in the RoboDk API, it is possible to send commands to the API from separate threads for each robot. These threads were created and executed through the creation of C# tasks.

These tasks had a few problems; firstly, when creating movement tasks within a separate thread (task), the API would not be able to reference this new task. In order to avoid this problem, all movement targets had to be created on the main thread and then made available to the tasks through a shared list. This solution made creating complex programs with many movement targets complicated, as what target belonged to each action and robot had to be determined by the tasks running asynchronously. A more straightforward solution to this could potentially be implemented, where what target belonging to which task and the order in which they should be utilized could perhaps be included in a new class within the application.

Another problem with utilizing C# tasks for controlling the robots concurrently is that when giving one command at a time to the robots, the controller can not plan movement paths based on further movement. When creating a program that includes all movements

for the robot, the robot controllers can optimize the movement of the robots based on where the robot needs to move after reaching the next target. In order to take advantage of this, another solution for concurrent movement was implemented. For this solution, programs were created in RoboDk and then started for both robots simultaneously.

These RoboDk programs presented a different set of problems; firstly they can not be modified after being started. This restriction prevents the application from being able to adjust the execution plan while the robots are moving, making responsive programming impossible. It is also not possible to make a program call the attach method during execution, meaning that the bricks could not be shown being moved by the robots during execution. Because of the benefit of allowing the robot controller to plan more efficient movements, RoboDk programs were implemented when simulating different solutions to the building a wall scenario. However, because the robots could not be simulated to picking up bricks, C# tasks were implemented for actually executing the best solution found. This solution is not ideal, as the discrepancy between simulations using programs, and the execution using tasks can lead to "safe" solutions causing collisions. If used on real robots, both simulation and executing the best solution should use programs.

Collision Avoidance

Another major challenge was to implement collision avoidance in the application. Four different solutions were tested. The first solution would be to specify "software barriers" that each robot had to stay within at all times. If these barriers were specified such that the robots could never collide, they would also not be able to collaborate in any meaningful way. It could be possible to specify barriers such that the risk of collisions was lowered and the robots still had a shared space to collaborate within. This shared portion would then either not prevent collisions if both robots could occupy this space simultaneously, or not allow for active collaboration if it is managed as a critical region. Only allowing one robot to occupy the shared section of the workspace is also very inefficient, software barriers were therefore only implemented and tested with the simplest passive collaboration scenario of delivering an object.

Collision Detection

The remaining solutions were first to simulate the solutions, and then deciding if the solution was safe if so it could be run on the physical robots. These solutions required some way of detecting collisions; three different approaches to this were tested. The first approach was to utilize RoboDk's collision detection while the robots were moving. This collision detection would stop the robots when a collision was detected and throw an exception from the API, allowing the application to know a collision had occurred. This way of detecting collisions presented two major challenges; firstly it would not prevent collisions from occurring, only detect collisions that had already occurred. In order to prevent collisions from occurring, the solutions were simulated before being executed. This simulation allowed the application to know if a solution resulted in any collisions before executing the solution on the physical robots.

The second challenge was that while RoboDk is checking for collisions, the calculations for this cause the robots to slow down when near other objects. The robots slowing down would not be a problem in itself, but the robots would slow down independently. The robots independently slowing down means that a solution that did not cause any collisions

when one robot was moving slower would end in a collision when both robots were moving at normal speed. In order to make this discrepancy as small as possible, the simulation speed had to be kept at real-time; this made sure the robots were moving at a consistent speed during the entire process. Keeping the simulation speed at real-time means that simulations utilizing this method will take as long to simulate as actually running the process. With the goal of allowing substantial amounts of simulations to be tested in a short amount of time, the third way of detecting collisions was tested.

The second way of detecting collisions was to run the robots with RoboDk's collision detection turned off, making sure the robots did not slow down when near other objects. RoboDk's "Collisions" method was then called continuously on a separate thread. This method would return how many object pairs were in a collided state when the method was called. This way of detecting collisions would enable simulations to be run at speeds way faster than real time. However, the API would throw exceptions when the collisions method was called with short intervals. This problem made this solution useless for simulation speeds above real-time, as the collisions method would have to be called with very short intervals in order to catch collisions happening at fast speeds.

The last way of detecting collisions that were implemented was to calculate how close each joint of the two robots were from each other, and then determining if this distance meant the robots had collided. This solution would allow the application to run simulations at high speeds, while not requiring the collisions method to be called. However, in order to get the angle of each robot joint, the API had to be utilized. The high frequency of retrieving the angles of the robots caused the same issue as calling the collisions method at high frequencies, where the API would throw exceptions. A way to solve this issue could be to somehow calculate the movements of the robots within the application itself or perhaps finding a way of getting the angles of the robot joints from RoboDk without causing the API to throw exceptions. No solution to this was discovered during this project. Because of the problems described with the different solutions, the first solution of utilizing RoboDk's collision detection and simulating solutions before execution was chosen, limiting simulation speeds to real-time.

Execution Plans

The next challenge was to generate execution plans for the solutions to a scenario. These execution plans, or programs, could be found with two different methods. The first method was to simulate solutions first, the problem with simulating solutions before execution is that no responsive programming can be implemented. This way of generating execution plans is a very conservative algorithm and relies on the execution going according to the simulation, and the application cannon react upon unforeseen problems. This limitation prevents the application from solving active collaboration scenarios. In order to allow the system to implement responsive programming and thereby enable active collaboration scenarios to be solved, a second solution to determining execution plans was explored.

The second solution to selecting execution plans is to implement some form of "digital twin" that would allow the application to operate based on eager algorithms, starting execution right away, and adapting based on how the execution performs in real time. The implementation of such a digital twin presented quite a few significant problems. Firstly, in order to simulate a digital twin, two instances of the robots had to run on RoboDk. The problem with running two instances of the robots is that RoboDk does not provide a separate environment for this. This means that referencing objects in one

specific instance may be complicated, furthermore, when the digital twin encounters a collision, the instance controlling the physical robots would also stop. These problems make having an instance of the digital twin and the physical robots within the same instance of RoboDk complicated and not very functional. It might be possible to find a solution to these problems, that would allow a digital twin to function with only one instance of RoboDk, no such solution was discovered during this project, however.

Another option would be to run two separate instances of RoboDk; however, RoboDk does not support multiple instances of the application on the same machine. A new instance will not start if RoboDk is already running. This could be overcome by having RoboDk run on a separate machine and communicating between the two machines over the network. This solution is however even more complicated than the first solution of having both the digital twin and the physical robots within the same RoboDk instance as network related issues like latency is introduced.

Because of the problems described, a digital twin and responsive programming implementation were not achieved during this project. It does, however, seem like this would achievable given enough time to test different solutions to the problems mentioned.

Selecting Optimal Execution Strategies

After having generated different execution strategies for a scenario, the next challenge is to select the optimal execution strategy. Optimal can mean something completely different depending on the scenario, for this project, optimal was chosen to mean the fastest solution to a given scenario. In order to determine the fastest solution, three different strategies were implemented. The first strategy is to simulate all possible solutions, then selecting the one with the shortest execution time as the best one. Finding the fastest solution presents the problem of potentially taking a long time, as there can be a huge number of potential solutions and the simulations having to be executed in real time due to the described problems with collision detection.

The second solution implemented was to solve the problem of finding the fastest solution as the mathematical problem known as the "secretary problem," a scenario involving optimal stopping theory, where the idea is to chose the first solution that scores better than the first 37% of solutions. This solution attempts to use as short of a time as possible simulating different solutions, while still having a relatively high chance of finding the best one. This solution is a good choice when there is a vast number of possible solutions, the chance of selecting the very best solution approaches 37% as the number of possibilities approaches infinite. If the best solution was found in the first 37% however, the algorithm will never find a better solution and end up simulating all possible solutions.

In order to use the least amount of time, the final strategy, selecting the first solution without collisions was implemented. This solution has a slim chance of selecting the absolute best solution. However, if the scenario takes a short time to execute, the difference between the solutions without collisions might be too small to warrant spending a significant amount of time simulating different solutions. In this case, this strategy might be ideal.

Physical Implementation

The last major challenge was to have the application control the physical robots. As the physical robots were not set up in a way that allowed collaboration, only concurrent movement was tested. The KUKA robot had a security barrier around it as students were utilizing the lab. In order to test the collaborative scenarios against the physical robots, this barrier would have to be deconstructed, and a new barrier would have to be created around both robots. Creating the barrier would take too much time and effort for being realistic during this project. The successful demonstration of concurrent movement on the robots, in accordance to commands sent by the application, did, however, prove that the application could successfully control both robots concurrently, implying that the scenarios could function on the physical robots as well as the simulation.

6.2 Different Forms of Collaboration

In summary, during this project, two major groups of collaboration scenarios were identified, passive and active collaboration. Passive collaboration scenarios include tasks that have the robots working independently towards the same goal, often resulting in a faster execution time when both robots are participating. Active collaboration scenarios include tasks where both robots are required to work in synchronization with one another. Active collaboration requires more complex control logic in order to be effective and avoid accidents. Some form of responsive programming is necessary in order to allow the robots to adapt to changes in the environment during execution. These changes are a lot more likely to occur during active collaboration compared to passive collaborations due to the movements of both robots affecting each other during execution. Such unforeseen changes could be a slight delay in the motion of one robot, a robots grip slipping or the robots being on a collision course. With passive collaboration scenarios, the only significant risk is the robots colliding with each other, as only one robot will be manipulating a single and separate object at a time.

To answer the second research question, the different collaborative scenarios were broken down into the skills required of the system in order to achieve the desired result of the scenario. This breakdown is beneficial for both a logical and technical analysis of the scenarios. For example, if all the skills necessary for performing the building a wall scenario is implemented successfully and the need for the building of a square consisting of four walls arose, the new scenario of building a square can be broken down into the necessary skills. The skills required for building a square are identical as the skills required for building a wall, with the only difference being that the application needs to calculate the destination positions of each brick such that a square is formed instead of a single wall. This observation would highlight that only a new method capable of this calculation is needed, with this new method implemented, all other skills can be reused for accepting inputs, simulating the different solutions and finding the fastest one. Breaking down scenarios into skills and categorizing them as either active or passive collaboration scenarios shows how different collaborative scenarios can be classified based on the requirements necessary in order to accomplish said scenario.

6.3 What Could Have Been Done Differently and Future Work

If I were to start over with the development of the application, I would have designed the application to implement each system skill as different classes or modules. An implementation where all skills are defined as separate classes would further demonstrate the practical and technical advantages of breaking scenarios down into the required skills. If the time restriction on the project allowed for it, the logical next step would be to implement active collaboration scenarios into the project. The application developed and the thoughts presented regarding collaboration between robots could provide a beneficial starting point for further experimentation and research into different methods for achieving efficient collaboration between robots. The implementation of responsive programming and machine learning could be fascinating when it comes to finding practical solutions to collaborative scenarios quickly.

Chapter 7

Conclusion

This thesis has aimed to present the challenges that come with having multiple robots collaborate within a shared workspace, and to discover different methods for solving these challenges. As well as to prove the applicability of the discovered solutions by implementing a proof of concept application capable of controlling both robots concurrently in such a way that collaboration was possible. In chapter 3 the design of the application, examples of different forms of collaborative scenarios and a method for classifying different forms of collaboration based on the requirements of the system in order to accomplish the scenario was presented. Predicted challenges were also presented, with possible solutions being discussed. In chapter 4 different solutions to the predicted and encountered challenges were evaluated and implemented in a windows forms application that controlled the robots through RoboDk's API. The proof of concept application was finally tested in chapter 5, where the feasibility of the system was proven on a proof of concept level. Although only passive collaboration scenarios were implemented, and only the act of concurrently controlling the robots was tested against the physical robots, the system functioned as expected and could be used as a strong starting point for future research. The benefit of classifying different forms of collaboration by breaking the scenarios down into the required skills was discussed in section 3.6 and chapter 6. However, the classification and benefits were not demonstrated in the implemented code due to the time required for restructuring the whole application would not fit within the time-frame of the project.

Bibliography

[1] Er Ankit Gupta, Er Arpit Gupta, and Amit Mishra. Research paper on software solution of critical section problem. *International Journal of Advance Technology & Engineering Research (IJATER)*, 03 2012.

[2] J. Neil Bearden. A new secretary problem with rank-based selection and cardinal payoffs. *Journal of Mathematical Psychology*, 50(1):58 – 59, 2006.

[3] F. Beuke, S. Alatartsev, S. Jessen, and A. Verl. Responsive and reactive dual-arm robot coordination. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 316–322, May 2018.

[4] Alexandre Filion, Ahmed Joubair, Antoine S. Tahan, and Ilian A. Bonev. Robot calibration using a portable photogrammetry system. *Robotics and Computer-Integrated Manufacturing*, 49:77 – 87, 2018.

[5] M. Gaudreault, A. Joubair, and I. A. Bonev. Local and closed-loop calibration of an industrial serial robot using a new low-cost 3d measuring device. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4312–4319, May 2016.

[6] Robot Operating System Industrial. About. `https://rosindustrial.org/about/description/`. Online; accessed 2019-01-19.

[7] Robot Operating System Industrial. Robodk post processors. `https://github.com/ros-industrial/robodk_postprocessors`. Online; accessed 2019-01-19.

[8] Kamal Kant and Steven W. Zucker. Toward efficient trajectory planning: The path-velocity decomposition. *The International Journal of Robotics Research*, 5(3):72–89, 1986.

[9] Aditya Narayanamoorthy, Renjun Li, and Zhiyong Huang. Creating ROS launch files using a visual programming interface. In *2015 IEEE 7th International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. IEEE, jul 2015.

[10] Albert Nubiola. The future of robot off-line programming. `http://coro.etsmtl.ca/blog/?p=529`, 2015. [Online; accessed 2019-01-09].

[11] Jufeng Peng and Srinivas Akella. Coordinating multiple robots with kinodynamic constraints along specified paths. *I. J. Robotic Res.*, 24:295–310, 04 2005.

[12] RoboDk. Documentation. https://robodk.com/doc/en/General.html. Online; accessed 2019-01-11.

[13] RoboDk. Homepage. https://www.robodk.com. Online; accessed 2019-01-10.

[14] RoboDk. Python API. https://github.com/RoboDK/RoboDK-API/blob/master/Python/robolink.py. Online; accessed 2019-01-11.

[15] RoboDk. RoboDk C# API Example Project. https://github.com/RoboDK/RoboDK-API. Online; accessed 2018-12-09.

[16] G. Sanchez and J. . Latombe. Using a prm planner to compare centralized and decoupled planning for multi-robot systems. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, volume 2, pages 2112–2119 vol.2, May 2002.

[17] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann. A new skill based robot programming language using uml/p statecharts. In *2013 IEEE International Conference on Robotics and Automation*, pages 461–466, May 2013.

[18] Wikipedia contributors. Robot software — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Robot_software&oldid=876308969, 2019. [Online; accessed 15-January-2019].