Modelling On Industrial Robots

Comparing and exploring different methods to program industrial robots.

Master's Thesis in Computer Science

Kim-Andre Engebretsen

October 15, 2020
Halden, Norway

Østfold University College www.hiof.no

Abstract

The work in this thesis attempts to explore possible new ways to program industrial robots and compare the methods of modeling the behavior of a robot to other more direct approaches. This is done by applying different modeling techniques using a Domain Specific Language (DSL), in this case ThingML, on RoboDK which acts as a middleware between the robot and the DSL. The approaches includes testing and evaluating different strategies for robot behavior. The behaviors are defined in different scenarios where the robot is tasked to build figures of various shapes using Lego Duplo bricks with added safety precautions regarding the robots workspace. The modelling language is applied to RoboDK through its API which supports several different programming languages such as C++ and Python. The process involves different stages where the first stage is mapping out the behavior in a state-machine model which is then generated into C++ code using model transformation provided by ThingML. In the final stage we would like to attempt to use the simulation software to connect to the industrial robots Kuka KR3 R540 and UR 10 which mimics the behavior of the robot in the simulation. The more direct approaches focuses on not applying a DSL to the simulation software, implying that all the programming in these cases wont be using any modeling techniques. The modeling techniques and the direct approaches are compared to each other by evaluating the efficiency, scalability, productivity and reliability of the tested methods. The goal of this research is to find the main differences between using modeling and more direct methods when programming industrial robots.

Keywords: Kuka KR C4, Universal Robots, UR10, Robot Operating System, ThingML, Robot behavior, Robot modeling, Robot languages, Model Driven Engineering, MDE, Robot Descriptions, Cyber Physical System, CPS, Kuka.

Acknowledgments

I want to sincerely thank my academic supervisor, Professor Øystein Haugen for his exemplary guidance and constant encouragement during my time of writing this thesis. I would also like to thank my family and close friends for their love and support through my years at Østfold University College.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Research topic

This topic propose the task of assessing the means to model the industrial robots UR10 and Kuka KR 3 R540 to potentially find out the best strategies for describing robot behavior and what methods yields the most effective and best results. This involves in particular the use of the Domain Specific Language ThingML to program robots with the help of the simulation softwares RoboDK and Robot Operating Systems. What makes this interesting is that a Domain Specific Language, in this case ThingML, allows us to map out how the behavior of a robot should look like using models such as statemachines. Previous work within the field of using models to control the behaviour of a robot has been for the most part to simply show the advantages of using models in robotics, but very few have taken the extra steps in actually comparing modelling, as a mean of programming robots, to more direct procedures where modelling isn't used, which is the driving force behind this research.

1.2 Research questions

RQ 1 *Comparing development of robot systems by model orientation versus more direct programming methods*

What are the main differences between developing a robotic system using a Domain-Specific Language with code generation framework, and a system using more direct approaches on the simulation software RoboDK?

> In this master's thesis there will be developed two separate systems. One system will be a model-driven approach using a DSL with Code generation where models are designed and then transformed into source code for RoboDK. The other system will not be a model-driven approach but rather program directly to RoboDK with it's own interface or API. We would like to know what differences there are in developing these two systems, how many symbols or lines of code is needed in each development? How long does it take to compile and run the application on the systems? What are the thought process behind each development process? When and what errors occurs in the development process and how are they mitigated?

RQ 2 *Improving performance and reliability of the developed software*

What are the main differences in performance between the model-driven and direct approaches when requirements for improvements and alterations to counter unpredicted events appear?

> We would like to know what changes needs to be done with the initial application on both systems for the robot to be able to perform a more complex task than what it initially was tasked to do. Also, we would like to know when and what errors occur in this process and how they are countered. An example of a unpredicted event could be that the robot or its work plane has been moved before or during a task. Another example could be that the safety switch turns on for whatever reasons.

RQ 3 *Higher production rate*

When the robot is altered to operate at higher speeds or tasked to build something significantly larger and more complex than it has previously been tasked to build, what are the main differences in changes that has to be done between the two approaches?

> Considering a scenario where one or both of these systems have been developed for a manufacturer for automating a process like assembling a structure using bricks. We would like to know what changes needs to be done between the different programming methods for the robot to be able to operate at higher speeds and build larger objects while encountering as few errors as possible. And if/when errors do occur at what speed or construction process did it happen?

1.3 Method

As the solutions is to be tested on real robots at some point, the first stage is to learn and getting to know how the robots work. Thanks to prior experience, the UR10 robot is to be prioritized last as its interface is easy to use and connection between RoboDK and UR10 does not require much effort. After gaining sufficient knowledge and understanding how the Kuka robot works the next step is to establish a connection between the Kuka and RoboDK. Then sufficient tools needs to be considered in order to do tests with the real robots. Once this is achieved, the next stage is to develop a model-driven solution with the use of a middleware such as RoboDK. Once the model-driven solution is setup and working as intended with RoboDK, we can assure that a connection between ThingML, RoboDK and the real robots is possible. Then the next stage is develop model-driven solutions that are designed to tackle the problems which will be presented later on. Once the model-driven solution is developed, a solution using a more direct approach with only APIs or provided interface from RoboDK is developed, figure 1.2 shows how ThingML and RoboDK are linked together for the different approaches. The results of each solution are then tested and measured in scenarios relating to the research questions and suggestions regarding future work and what could have been done diferently is mentioned. The results is then discussed and a conclusion is drawn based on what was found during the evaluation process.

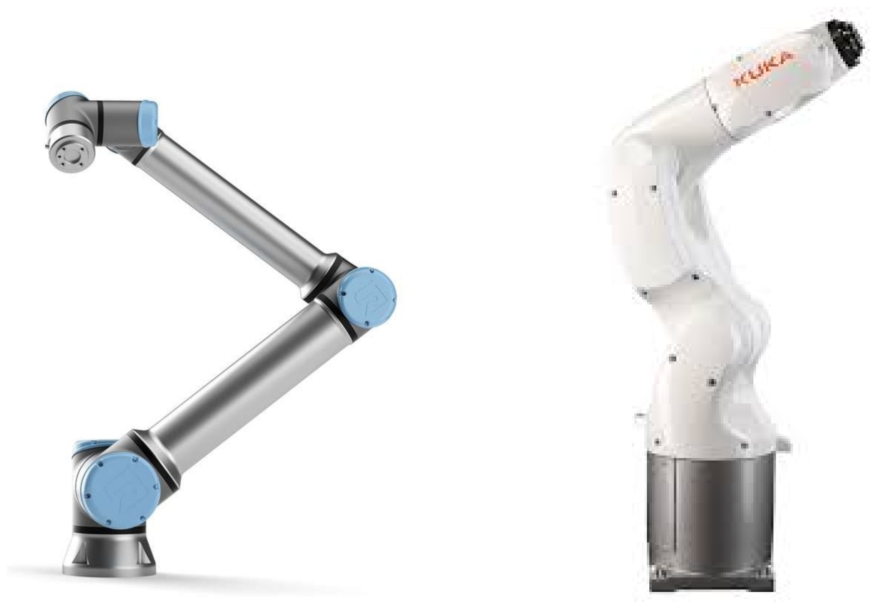


Figure 1.1: Left:The UR10 Robot. Right:The Kuka KR 3 R450.

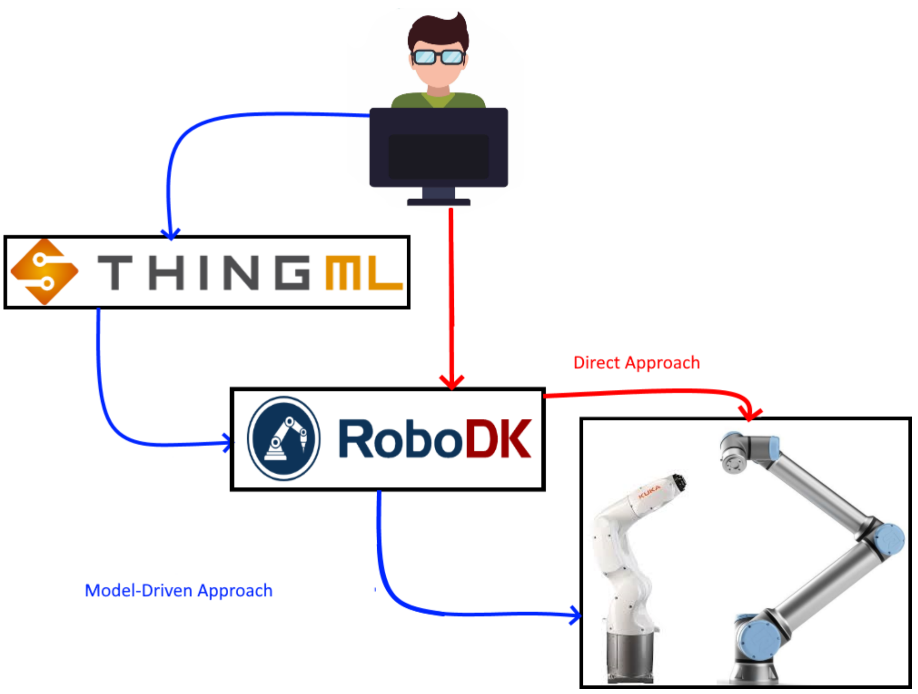


Figure 1.2: The figure shows how the softwares and ThingML are linked together in the different approaches.

1.4 Report Outline

In chapter 2, a overview over the state of the art is provided. In this chapter the methods used in retrieving the literature are described and sorted in a table together with search keywords that were used and from which database they were obtained. The literature is divided into three main topics that are related to the research done in this thesis. Each main topic provides a short introduction and previous work done in the area. Lastly, four evaluation topics on how the approaches are to be measured is presented. A general description is provided each topic and how they are linked to each research question

Chapter 3 provides an overview over RoboDK and the chosen domain specific language ThingML. A short introduction is given for each software and ThingML with what kind of functionalities they bring and how said functionalities can be used for the development of each system in this thesis. The 2nd section presents four scenarios where the robots are tasked to build structures. Each scenario is designed to call for further improvements of the systems which relates to the research questions. The 3rd section provides a short introduction to one of the required improvements which is utilizing 3D models and some related work done in the area. Lastly, a short summary is given summarising each software and ThingML and how they are used in different scenarios in order to answer the research questions.

Chapter 4 is the part where the experiments for each scenario is conducted. The first section of this chapter presents the model-driven approach which uses ThingML with RoboDK as a middleware. In the first part of the section an overview over how the steps involving a general model-driven approach with RoboDK as a middleware is set up is presented, in addition a short overview of RoboDK's C++ API which is gonna be used in combination with ThingML is provided. The last part of the first section presents the model-driven approach for each scenario and how improvements were made in order to satisfy the new requirements. The two next sections presents the direct approaches where programming is done directly on the middleware's software or utlizing a supported API. Lastly, a short summary is given summarising the approaches for each scenario.

The first section in Chapter 5 presents the results of the experiments conducted on the scenarios for each approach and how they were measured for each evaluation topic presented in the last section in chapter 2. The next section presents suggestion for further work and what could have been done differently in this thesis. The results are then discussed in chapter 6 analysing each evaluation done on the approaches. Finally, the last chapter, chapter 7, presents the conclusion of the work done in this thesis and how the research questions were answered.

Chapter 2

Background

2.1 State of the art

2.1.1 Model-Driven Engineering (MDE)

Model-Driven Engineering (MDE) is a well known software development methodology [13], [2]. Kai et al. [10] stated in their research regarding the use of DSLs in different domains that MDE provides a need based abstraction of a system to various stakeholders for different concerns, which copes with the challenge of requiring expertise from multiple domains when developing robotic applications for advanced robotics systems. By using MDE, models that reduce the complexity of robotic systems can be created, making the development process easier to understand and validated because knowledge from every domain is not needed, thus lowering the technical skills required when developing robotic systems [1, 2]. Further, MDE allows to increase the level of automation through code generation and bridge the gap between modeling and implementation, the anticipated results are usually improved efficiency and quality of the robotic system engineering process.

Brugali [17] did a survey where the aim was to analyze the role and use of MDE technologies in robotic software engineering. The research focused on the architectural model as the central artifact of most software development activities such as analysing, design, implementation, configuration, and documentation. Davide Brugali stated that recent research and development efforts in the MDE for robotics are motivated by the objective of simplifying application development by experts in robotics with limited software engineering skills. It was also stated that according to a study on the state of practice in MDE in industry reports that the gains in productivity based on automatic code generation alone is not considered significant enough to adopt MDE.

To explore the advantages of MDE, in design and development tasks specifically, Estévez et al. [12], presented an arm based robotic tasks modeling tool called ART2ool. ART2ool is an MDE framework which turned out to be very useful for application domain experts as it guided them along the design of the functionality, abstracting from the emerging techniques. This framework supports automatic code generation by using Model to Text (M2T) transformation for component-based and ROS communication middleware, this achieves reusability, flexibility, and adaptability which are software requirements for handle robots. This is because due to continuous changing market demands, modern production facilities demands reusability, integration, flexibility, and optimization of industry processes as major requirements. When performing a M2T transformation, it means transforming a model into source code or documentation, this source code can be used in multiple general-purpose programming languages such as Java and C [3], [4]. This makes M2T really flexible because we can generate code and reuse it in different languages if it is desirable. During the explanation regarding automatic code generation a rule set for M2T transformation to generate executable code for the ROS platform was proposed.

Zander et al. [23] presented a novel ontology-driven software engineering approach for developing industrial robotics control software. For this approach they used the ReApp architecture which provides a toolset for MDE which is seamlessly integrated with ontologies that ease the process by providing structural and domain knowledge. By using ReApp it is shown how different ontological classification systems for hardware, software, and capabilities help developers discovering suitable software components for their tasks and how to apply them correctly. Zander S et al. states that their proposed model-driven tooling makes it possible for developers to work at higher abstraction levels and encourages the development of automatic code generation. The viability of the presented architechture were demonstraded through three examples showing the workflow for creating and retrieving software components. The ReApp tool chain with it's ontology-based information model enables MDE of components and aid users with the knowledge encoded with the ReApp domain. It greatly enhances the chance of software reuse, lifting the burden of having to be proficient with every piece of software. Zander S et al states that the enhanced usability and reusability of software enables faster development of robotic applications and keeps them maintainable, allowing flexible response to market demands.

Wenger et al. [21] used the same architecture in their approach to provide reusable robot applications based on ROS and aimed to provide a development environment for creating, applying and managing reusable robotic software components. The reason behind their project was the requirement of experts specialized in a specific manufacturer's technology. The reuse of robot programs are, even partially, close to impossible due to different robot languages being employed by the robot vendors. The provided workbench called Skill and Solution Modeling Tool (SMT) provided a graphical editor, supporting views to compose a robot application out of different types of nodes which are a part of programming application in ROS. To demonstrate their approach, an example was presented where the task was to automate a soldering process.

Rehbein et al. [14] whose research was to combine the advantages of MATLAB based, model-based design and industrial robot programming, used Simulink together with MATLAB to control a KUKA Agilus KR6 R900. This advantage involved enabling non-programmers to be able to work with industrial robots by using graphical programming, and automatically adds robustness against code errors. The process of creating a robot program using Simulink and MATLAB involved four main steps. The robot program was first developed graphically using Simulink. Simulink is a MATLAB-based graphical programming environment for modelling, simulation, and analysis of dynamic systems using primary graphical block diagram tools [7]. After Simulink has finished compiling, every motion is saved to a MATLAB workspace which then establishes a TCP/IP connection to a PLC. The PLC receives the motion commands and passes them on, one by one to a robot controller using, in this case, mxAutomation which is a control software developed by Kuka for allowing external controllers to command Kuka robots on the basis of elementary motion instructions[1]. Lastly, the robot controller does the trajectory planning of the robot before it starts moving, simultaneously the PLC will send process data from the robot system to MATLAB. The test was conducted by feeding the robot motion blocks containing coordinates, this was done in a graphical user interface (GUI) using Simulink. Between two blocks called BEGIN and END, the robot program could be placed and coordinates were added to motion blocks which made the robot perform a linear movement accordingly. Though the research itself was successful, it was not possible to see a difference in execution time or precision when comparing their approach using mxAutomation and the Kuka Robot Language itself. It was stated that using this approach with mxAutomation didn't exactly show an improvement in execution time or precision compared to the kuka robot's own language.

2.1.2 Domain Specific Language (DSL)

The method of adding MDE usually involves using designated programming languages called Domain Specific Languages (DSL)[18]. Kai et al. [10] who presented a collection of DSLs to describe service robotics applications, stated that DSL's, or more specifically, DS(M)Ls which stands for domain-specific modelling language enables domain experts to model solutions using their own terminology of the problem domain. As there are many different DSLs used for different strategies when developing robotic applications, Nordmann et al. [18] did a survey on the available literature regarding DSLs that target core robotics concerns. In total 137 publications were identified following a set of criteria where the goal was to provide an overview on the state-of-the-art of DSL approaches in robotics. The publications were investigated from a user and developers of model-based approaches point of view. The results of their analysis provides an overview of which subdomains were used the most and Nordmann A et al. were not surprised by the fact that more than 53% of the publications covered the subdomain called Architectures and Programming. The reason behind this is the broad area which Architectures and Programming covers and includes several important aspects which has to be taken into consideration when using a DSL in a model-driven approach. Lastly, the survey also provides an overview of four different quantitative benefits and corresponding metrics which can be used to evaluate a model-driven approach, these are: Efficiency, Scalability, Productivity and Reliability.

Ringert et al. [19] sketched a concept for for code generator composition based on explicit code generator interfaces and configuration models by using MontiArcAutomaton, a modelling language that is a component and connector (CC) architecture description language (ADL). A conceptual approach based on the notion of generator interfaces were presented. They developed a MontiArcAutomaton solution using the domain-specific language (DSL) MontiCore, this was presented using a problem statement and example. The problem statement involved the CC ADL and how a framework to support code generators have to provide a mechanism for configuring CC applications with different code generators. The example involved a software engineer responsible for the development of a controller for a robotic arm to support a physically disabled person operating a toaster in a kitchen environment. During the example it was explained what elements a code generator interface should contain. Kai et al. [10], [9] used a similar solution of the combination of

MontiArcAutomaton and CC ADL with MontiCore where they proposed a modular architecture method which rests on the separation of model processing, model transformation and code generation. This method involves translating architecture models into modules that are compatible with a middleware of choice such as ROS and includes two types of modeling transformations: M2T which was a part of ART2ool like mentioned earlier and M2M which means Model to Model. The resulting code generator of the proposed method enabled translating MontiArcAutomaton architecture models to ROS Python nodes.

Miyazawa et al. [16] proposed a set of constructs suitable for modeling robotic applications and supporting verification using model checking and theorem proving. For this they used a DSL called RoboChart. RoboChart is a language based on UML state machines, some notable features which this language includes are the notions of robotic platforms, parallel controllers and machines, and synchronous and asynchronous communications. The goal behind their approach on this subject was that they wanted to support roboticists in writing models and applying modern verification techniques using a language familiar to them. To support work on case studies and validation of the presented semantics, a user-friendly tool called RoboTool was developed for editing and verifying RoboChart diagrams. This tool supports automatic generation of CSP semantics which was used to validate via model checking, besides, this tool also supports automatic generation of C++ implementations for a subset of state machines and controllers. The approach was demonstrated through three case studies where various requirements were formalised and semantics were used to verify them.

Cavalcanti et al. [11] presented an approach to verify the consistency of simulations with respect to design models using a tool-independent notation called RoboSim. The work was carried out in the context of RoboChart and RoboSim, the central concept for both languages is state machines, the difference between the two is that RoboChart targets design while RoboSim targets the simulation models. A running example were developed and three case studies were presented, illustrating how RoboSim and the proposed verification techniques works. It is shown how RoboSim models can be used to check if a simulation is consistent with a functional design written in a UML-like notation similar to those used by practitioners on an informal basis. Lastly, it was shown how to check whether the design enables a feasible scheduling of behaviors in cycles as needed for a simulation, and how to formalise implicit assumptions routinely made when programming.

Wigand Wrede [22] proposed a DSL for modeling timing behavior of real-time sensitive component-based robotic systems. The proposed DSL is integrated onto a CoSiMA framework, which offered the ability to model, simulate, deploy and analyze the behavior of robotics systems on different platforms. To show how the DSL worked in practise, an experimental system was modeled and executed using CoSiMA. CoSiMa is an architecture designed specially for supporting component-based development, analysis and execution of robotic systems on simulated and real platforms. The system consisted of a humanoid robot called COMAN which had the task of performing a zero moment point-based walk in a straight line, the actual execution was then compared to the model later on. The proposed extension for the CoSiMa framework offers developers a way to define more fine-grained requirements on real-time sensitive system. The advantage of this approach is that the system can be evaluated beforehand or in simulation which increases the reliability and safety which leads us to one of the reasons behind this approach. It was stated by

Wigand L,D., Wrede S., who claimed that correct timing is crucial for robotic systems that requires to expose highly reliable behavior. Since the demand for robots applied in collaborative environments is increasing drastically, robots need to be even more reliable and safe. It is mandatory for non real-time component-based robotic systems to guarantee the safety in sensitive environments such as faulty behavior leading to health hazards which puts robots and humans in danger. MDE techniques has in recent trends been applied to specify and analyse such a system and its behavior.

2.1.3 Summary

Using MDE techniques to create robotic software and develop robotic systems is a very common theme. Removing the necessity of requiring experts from multiple domains to program robots was a very popular topic among the presented literature and was the goal behind most of the research projects. The tools and approaches presented did well in describing how they worked, what benefits they brought, and how they worked in real life examples. However, very few mentioned comparing their approach to how it would perform using other tools or methods, or how tackling the same system development or task without using their approach would be. So far only Rehbein et al. [14] compared their approach using mxAutomation to the robot's own programming language. From a developer point of view being specialized in using the robot's programming language, would it even be necessary to apply a model-driven approach like the one Rehbein J et al. did? As mentioned by Brugali [17], the code-generation frameworks which seemingly allows to increase the automation of a process is not good enough to consider adopting to MDE. But since Rehbein J et al.'s aim was to show the advantage of model-based design in industrial programming enabling non-programmers to work with industrial robots, maybe it wasn't necessary to look at their approach from a developer's point of view who is specialized in the area. The goal of the master's thesis is essentially to compare a model-driven approach for programming industrial robots, with an approach that is not using models to program industrial robots. Evaluating the methods and DSL used in the approaches should give an answer to whether users should consider using modelling techniques to program industrial robots or if they are fine without it. Though it is worth mentioning that, one major advantage to using a DSL or similar modeling techniques to create robotic applications is the use of model transformations and code generation. Designing a model for a system by using state-machines for example, which is then transformed to source code usable in general-purpose languages such as Java or Python makes model-driven solutions very independent of the platform due to its reusability. Solutions that uses a middleware such as ROS or RoboDK between the application and the robot can simply change the type of source code generated if the developer assigned to the task was for example more comfortable programming in C++ rather than in Python. Lastly, while we were able to look at previous work using ROS based approaches and how code generation was used to create an ROS application, there were no related articles using MDE with RoboDK as a middleware.

2.2 The process of evaluating the approaches

The process of evaluating the approaches is built upon the results presented in Nordmann et al. [18]'s survey. The results of the survey presents an overview of four different quantitative benefits and corresponding metrics which can be used to evaluate a model-driven approach, these are: Efficiency, Scalability, Productivity and Reliability.

2.2.1 Productivity

In order to answer RQ 1, several aspects in the development phases is measured. Programming in ThingML compared to C++ and Python might provide different challenges when developing each system. And if the same challenges appears in other languages the problems may or may not have to be treated differently. Therefore, challenges that occurs during development for each language used, whether it be making an algorithm or missing the use of a specific library, is to be noted and compared later on.

Advantages or disadvantages that is noticed between the different languages during development is to be measured as well. Some of the programming languages might provide libraries or functionalities that works better on different aspects during development. For example ThingML might have functions or libraries (that is otherwise not found in C++, Python or ROS) that works better for handling errors that might occur in the approaches.

Lines of Code or amount of characters is to be noted as well during development. This is to measure whether getting a system up and running requires more effort in for example ThingML than in RoboDK or ROS. Effort regarding bug fixing and troubleshooting is to be measured as well for each language and software used.

2.2.2 Efficiency and Reliability

Efficiency and Reliability through performance. This is more hinted towards how the proposed method of applying a Domain Specific Language (DSL) to RoboDK are performing compared to more direct approaches. At first, in order to answer RQ 2, the robots performance in building figures of various shapes is to be measured. This could be how well it was able to pick up and place bricks in the real world in comparison with the simulation, unfortunately due to the corona pandemic, running experiments on the real robot was not possible due to the lockdown of Østfold University College. Therefore, only the performance of the simulation, and the efficiency and reliability of each solution in the development processes is measured. Efficiency-wise, run-time and the complexity of the approaches is to be measured. By measuring run-time we mean how long does it take for the approaches to execute code and by complexity we mean how complex each developed solution is. Reliability-wise we measure what kind of features is available for each approach to resolve issues as and if they occur and how to avoid further occurrences of each issue.

When the robot is moving around, picking up bricks and placing them in a set pattern, it is not unreasonable to think that it may collide with objects within its working space unless measures to counter collision is put in place. However, how well is the robot performing before such measures have to be put in place? Does it avoid collision with already placed bricks from the get go? And how many tweaks has to be done before a collision free movement when tasked to to build different structures is achieved? If this was to be measured in the real world as well, it would be appropriate to study how well the robot picks up and places bricks. Maybe the grip force is too low or too high, maybe the robot does a bad job in placing the bricks at the exact right spots, maybe it struggles hard in higher operating speeds? These are unfortunately not problems one will encounter in a simulation.

2.2.3 Scalability

Operating speed and resource usage when the demand for higher production rate is present. This is hinted towards how well the robot performs using the different programming methods when the system is altered to produce more or larger products at higher speeds, does it completely miss when trying to place the bricks at correct spots? does it end up destroying a brick or two in the process?. Unfortunately this is something that cant be properly measured for the reason that, as mentioned earlier, experiments on real robots is not possible at the time of this work and such problems does not occur in a simulation. Therefore, to answer RQ 3, only the main differences when the robot is tasked to build something of significantly larger scale than previous attempts is studied.

Chapter 3

Preparations

In this chapter it will be explained more in depth what tools and languages will be used and why. The approach for utilizing the tools is described and how it will be tested on various scenarios in order to answer the research questions. First, an in depth overview of

The goal of this chapter is to give an in-depth overview of the exact tools that will be used, which programming languages to be used on said tools and how the testing of the approaches will be done.

3.1 ThingML

ThingML is a Domains Specific Language (DSL) and has three main constructs: Thing, Message and Instances. A Thing can be determined as something that behaves like a state machine, Things have local attributes and communicates asynchronously through ports. The Messages describes the interaction between Things and are sent asynchronously. Instances are established and connected [8]. In the model-driven approach the first steps when making a program for any given scenario is designing the behavior of the robot as a state-machine in ThingML. These state-machines can be generated to usable code for several different purposes and covers multiple programming languages, this includes files using C++ which will be used as the main programming language when using the provided APIs from RoboDK. The state-machines can also be generated into a UML diagram [8].

One technical disadvantage with the available model to source code compilers in ThingML is that the generated files using C++ code is not intended for regular C++ usage but they are rather made for Linux and Arduino. To work around this, the Arduino ino file can be altered to a cpp file by picking out only the parts that is needed to run the code with the tools used in this project. How this is done will be explained in the execution phase, chapter 4. For future work however, if possible, the ThingML compiler can be altered to automatically do the editing of the ino file, since it is the same functions every time that needs to be removed or changed, the work required to do this may not require much, however, current knowledge does not allow for this alteration to happen during this thesis as time is taken into consideration for goals to be reached within a certain time. The reason why this alteration to the compiler is desired is to achieve as close to 100% automation as possible when generating usable source code.

3.2 RoboDK

RoboDK is a simulation software used to simulate one or multiple robots performing tasks in a virtual environment. RoboDK allows for offline-programming on robots using python scripts or other programming languages using the RoboDK API. Robots can also be programmed from using RoboDK's own interface using 3D reference frames and targets which serves as positions for the robot to move to. After creating and running a robot simulation through either using the RoboDK API or RoboDK's interface, the simulated robot can be turned into a script which can be run on a physical robot using an ethernet connection and have it perform the exact same tasks as the simulation [6].

In the model-driven approach, the second step involves using either RoboDK or Robot Operating System to run the generated code on the robot. Since the generated code is C++, RoboDK recommends that we use QT Creator, a cross-platform C++, JavaScript and QML IDE. RoboDK also provides a sample application designed in QT creator which includes a Graphical User Interface (GUI) that connects the application to RoboDK. To save time, an altered version of the provided GUI will be used in this master thesis for the model-driven approach using RoboDK. The altered GUI will be using a header file to connect with the functions provided in the generated C++ code. Once the program has been simulated it can then be run on a real robot from the GUI using functions provided by the RoboDK API.

In the more direct approach where offline programming is done directly from RoboDK, the generated code from the model will not be used. In this approach, the goal is to achieve close to the same algorithms using either RoboDK's interface or python scripts which allows to be directly run on RoboDK without the use of an additional IDE such as QT Creator for C++ code. The direct approach will then be compared to the model-driven approach later on by evaluating their behavior as discussed in chapter chapter 2.

3.3 Scenarios

There will be a total of four scenarios defining the behavior of the robot which the different approaches will be tested and evaluated on. Each scenario describes a task that the robot will do. These tasks involves picking up and placing 2x2 and 2x4 sized Lego Duplo bricks in a certain pattern to make figures of various different shapes.

3.3.1 Scenario 1, 2D patterns, collision avoidance and dispensers

A manufacturer wants to automate a process where bricks are placed together in a set pattern. The patterns are quite simple as they are only 2 dimensional and layers of the same pattern will be stacked on top of each other essentially making a structure where the outside shape is the same but the inside can change and be different. The manufacturer has 2 available robots for this, the UR 10 and the Kuka KR3 R540, see figure 3.4 for a overview on how the systems looks in RoboDK, the robots are placed on a table with 6 lego duplo baseplates placed together in front of them, this is shown in figure 3.2, the area which the baseplates covers defines the workspace which the robot are allowed to move and place bricks in. To the side, two dispensers are present, one dispenser contains 2x2 sized lego duplo bricks, while the other one contains 2x4 sized bricks, figure 3.3 shows a picture of how this looks in RoboDK. While there are no container for these bricks, the dispensers

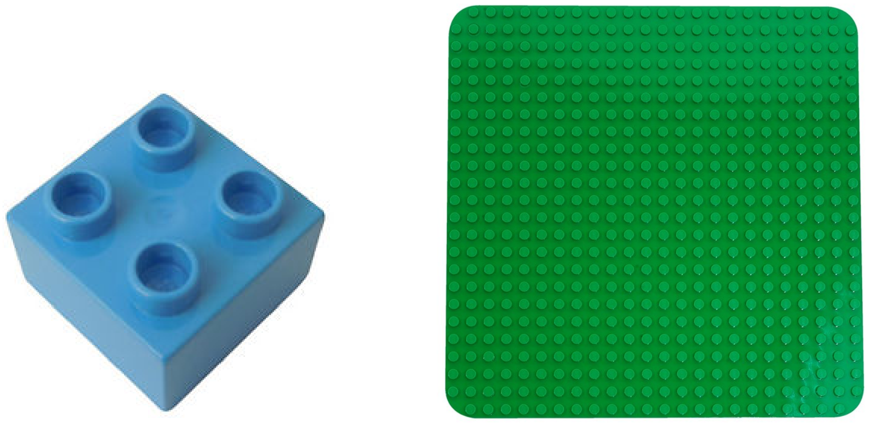


Figure 3.1: Left: 2x2 Lego Duplo block. Right: Lego Duplo Baseplate

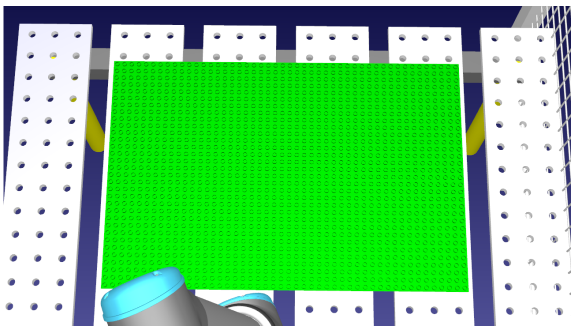


Figure 3.2: The figure shows 6 duplo baseplates together. These defines the workspace of which the robot are allowed to move and place bricks.

serves their purpose in the simulation by getting smaller each time the robot picks out a brick. Each dispenser can hold up to 14 bricks and there is sensor which keeps track of whether the dispenser are empty or not. The patterns varies in sizes and can be as big as the work space allows for. It is crucial that each brick lines up with the baseplates studs to avoid damage and place the bricks with movements that avoid collision with objects. Figure 3.5, 3.6, 3.7, and 3.8 shows some examples of these patterns, the figures have grids where each cell represents a coordinate for brick placement.

3.3.2 Scenario 2, 3D models and Emergency button

The manufacturer contacts us for improvements of the existing system. Instead of using 2 dimensional patterns as the base ground for building the structures, the system will now build the structures based on 3D models. The 3D models can be anything the manufacturer desires as long as the size is within the reach of the robot and does not exceed the limits of the work space. The thought process here is that a user can make or find a 3D model online and insert it into the existing systems as some type of input. The building process will act in the same way as before where it builds layer by layer from bottom to top. Lastly, the manufacturer wants an emergency button installed in the system where everything stops when the button is pressed and resume from where it left off when the button is released.

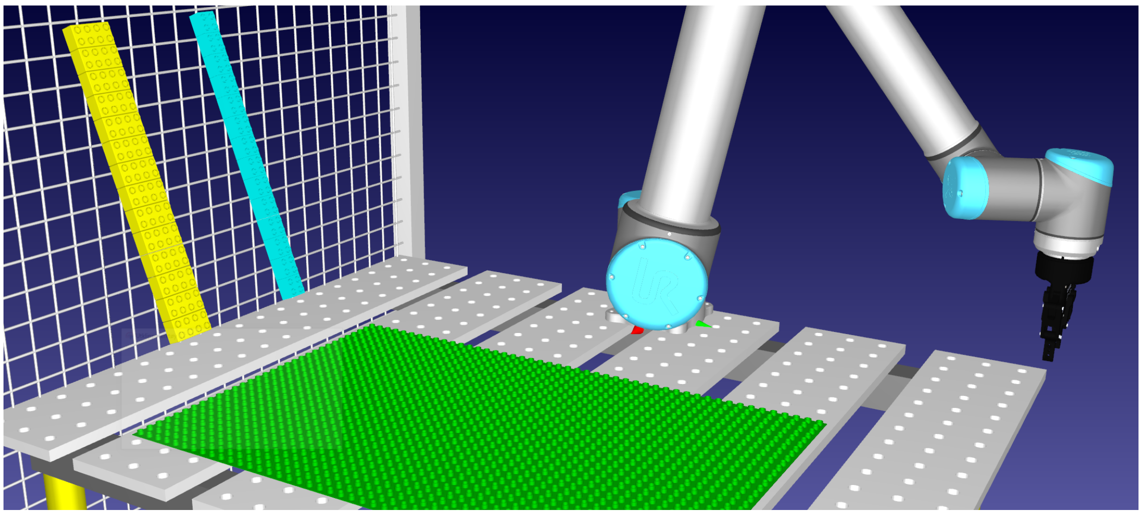


Figure 3.3: The figure shows how the dispenser looks in the setup used in RoboDK. There is no shelf model but the thoughtprocess is that the bricks are placed on top of each other in an angle so when the robot pulls out a brick the remaining bricks will not fall out of the dispenser when they're falling down.

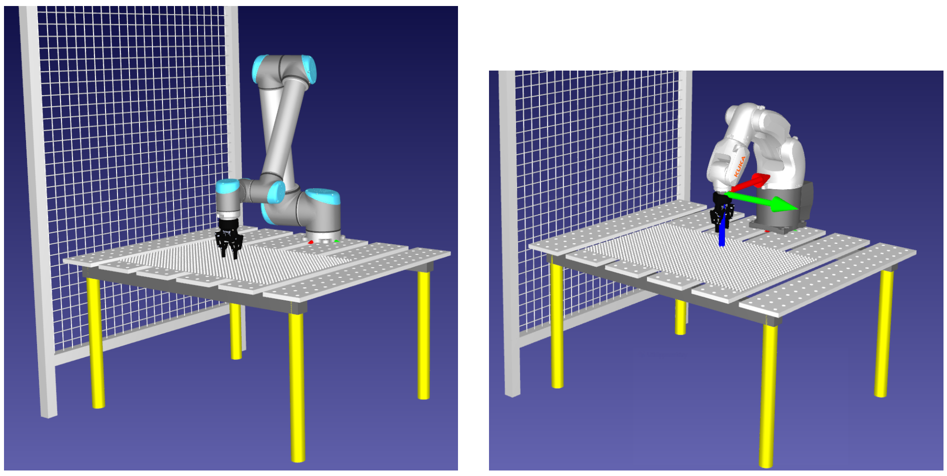


Figure 3.4: Left: The UR10 setup as shown in RoboDK. Right: The Kuka setup as shown in RoboDK.

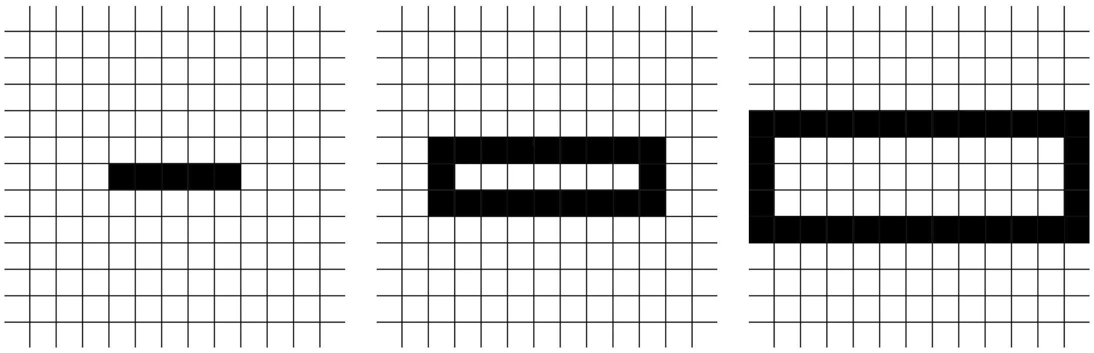


Figure 3.5: Left: Middle: 3x9 Wall figure. Right: 5x13 Wall figure.

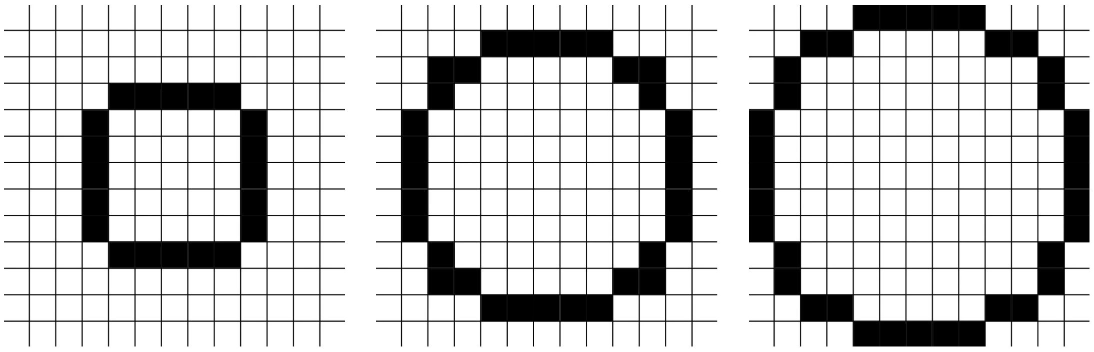


Figure 3.6: Left: 7x7 Circle figure. Middle: 11x11 Circle figure. Right: 13x13 Circle figure.

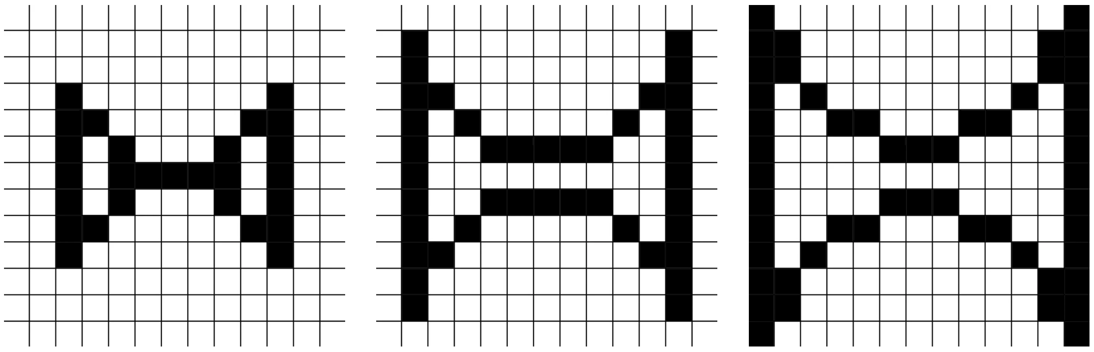


Figure 3.7: Left:7x9 Concave figure. Middle:11x11 Concave figure. Right:13x13 Concave figure.

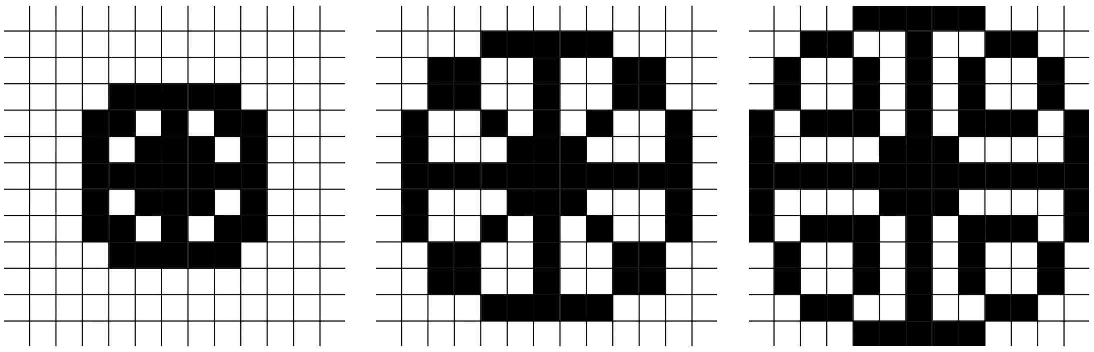


Figure 3.8: Left:7x7 Complex figure. Middle:11x11 Complex figure. Right:13x13 Complex figure.

3.3.3 Scenario 3, support bricks and human harm avoidance

While the systems are eligible to now build structures based on 3D models, the finished structure may not quite look like the initial model that was used as a input for the systems. While building the structures, the robot in use encounters bricks that could be categorized as impossible bricks, also called: "floating bricks". As the robots builds the model layer by layer, one of the layers might have a brick or two where there was not placed a brick in the same position in the previous layer, which means that the robot essentially places the brick on thin air. Just like 3D printers, when tasked to print figures with a difficult shape, additional support structures are added to avoid the figure falling over while in the middle of printing process. The system at hand must find out where floating bricks occurs and place additional bricks acting as a support beam so the bricks are not placed into thin air. Also the manufacturer wants to add a safety feature in case a user for some reason places their arm inside the work space while the robot is operating.

3.3.4 Scenario 4, better usage of studs, sub-figures and error handling if the robot drops a brick

The final improvement of the system is to have the choice of assigning each stud a coordinate placement in the algorithms instead of bricks. The existing systems before this implementation only placed single bricks on top of each other, in other words, none of the bricks in layer z+1, where z is the current layer, is placed in such manner where it is connected to two separate bricks in layer z. Also, the manufacturer wants to add a feature to the algorithm where if too many floating bricks occurs, the remaining part of the figure shall be built in separate location, which can be to the side of the original structure. This is to avoid using too many bricks for support purposes. Lastly a feature for handling when the robot accidentally drops a brick is to be implemented.

3.4 Building with 3D models. Related Works

Sugimote et al. [20] studied a robotic system which acted as a 3D printer by converting 3D CAD models to a block model which allowed for the robot to build the model using bricks resembling lego. The study was a based on their previous work where a prototype system with an industrial manipulator was implemented. In the new approach, the goal was to add three functions to allow the system to print more various models. These three functions included color printing, a method to deal with unprintable shapes, see figure 3.9 for an example of a unprintable shape, and a block feeder to generate block structures of a larger-scale. What is interesting with this study which will be utilized in this master's thesis is the study's method of dealing with unprintable structures. They introduced two methods to deal with this which both worked on their own to a certain extent or they could also be combined. These methods involved support blocks, see figure 3.10, and splitting the structure into sub-figures, see figure 3.12 for a example of sub-figures and see figure **??** for a example of combining both methods. The way these methods are handled are by looking for "floating" bricks in each layer of the structure. Since the robot always will build layer by layer there may be bricks that cant be placed because there aren't any previously placed brick in the same position in the previous layer, this will classify the brick as a "floating" brick. Other bricks that also would be considered not placeable would be bricks that may collapse the structure or at least a part of it, see figure 3.13 for an

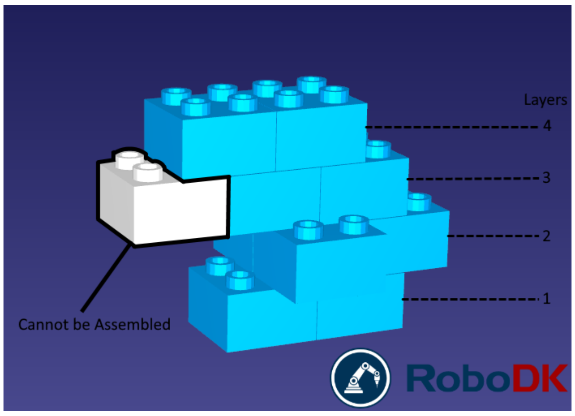


Figure 3.9: A example from Sugimote et al.'s thesis recreated in RoboDK. This structure is considered an unprintable shape as the white brick will essentially end up as a "floating brick" if the robot builds the shape layer by layer

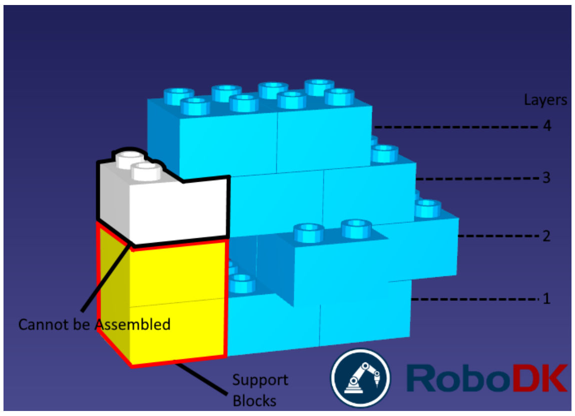


Figure 3.10: The assembly of the structure in figure 3.9 is made possible with support blocks (yellow bricks)

example of this. Consider a grid like figures: 3.5, 3.6, 3.7 and 3.8, instead of having each position in the grid defining a brick placement, it would be ideal to have them define studs instead. This allows for the system to utilize the lego bricks's studs where two lego bricks placed next to each other can be connected by placing another lego brick on top using the studs from both bricks, which is normally how lego structures are made. Maeda et al. [15] did a study prior to this where they developed an algorithm to avoid the structure collapsing upon brick placements by checking which bricks connected with which studs in the previous layers.

3.5 Summary

To summarise how it is planned to reach an answer to the research questions, two different approaches will be done, a model-driven approach and a more direct approach. The model-driven approach includes 3 steps when testing each scenario. The steps involves to first model statemachines using ThingML to define the behavior of the robot, then ThingML transforms the state machine into C++ code which will be used together with

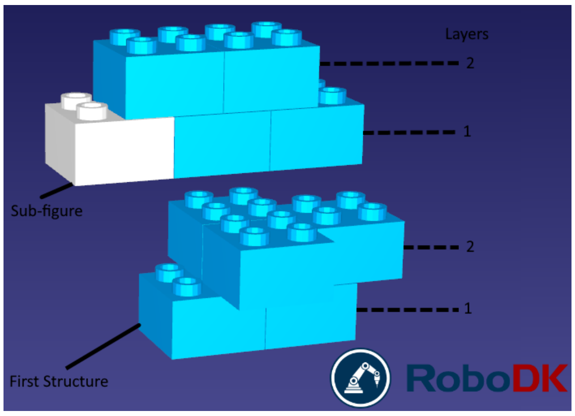


Figure 3.11: The assembly of the structure in figure 3.9 is also made possible by splitting it into two separate pieces where the last two layers are built separately

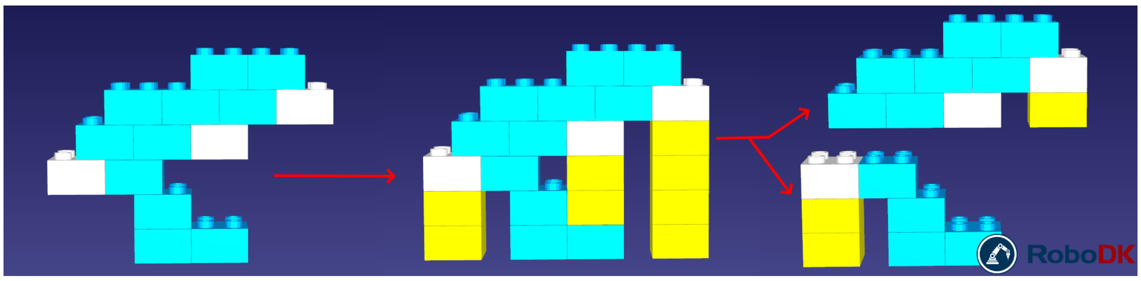


Figure 3.12: Another example from Sugimote et al.'s thesis recreated in RoboDK, this example combines both support blocks and sub-figures to build a difficult shape

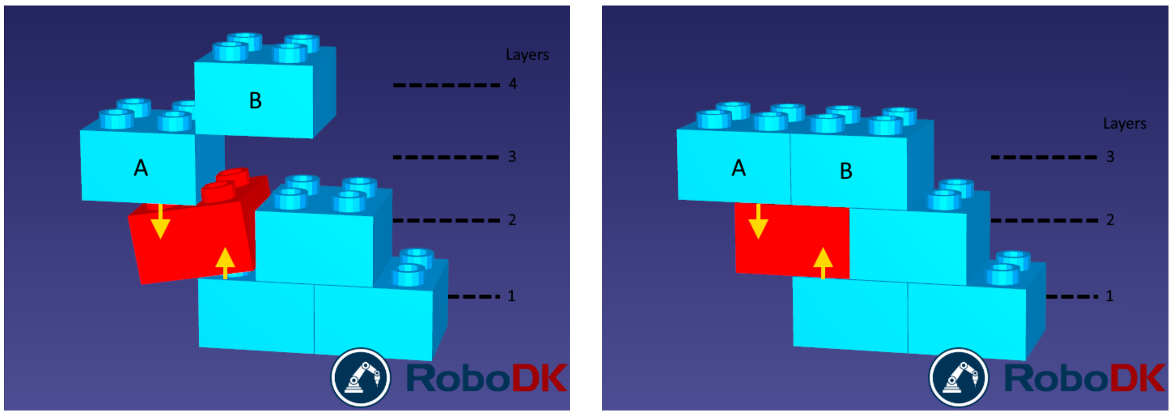


Figure 3.13: Depending on which bricks are placed first, placing bricks in positions that aren't properly supported by the previous layers may result in the structure collapsing

RoboDK's API to program and simulate the robot's behavior. After simulating and additional tweaking has been done, the next step would be to run the code on the real robots via RoboDK to see how it performs in a real world environment.

The direct approach is very similar to the model-driven approach in terms of how the robot should approach the tasks it is assigned to, the only difference is that modeling and model transformation is not used here. In this approach the robot is programmed directly from RoboDK using the C++ API.

Both approaches will be tested using different scenarios where the robot is tasked to build figures of different shapes using Lego Duplo bricks, which is placed on a Duplo baseplate. Some of these shapes are straight forward such as a simple wall, while other shapes can get more complicated. To add to the complexity of the scenarios, the robot will also be tasked to build 3D structures of various shapes and sizes where it might encounter bricks that are unplaceable, such bricks could be bricks that, due to brick-placement of previous layers of the structure, might be considered a "floating brick" because there aren't any bricks for it to be placed on or the brick could collapse the structure upon placement. To counter this, two methods, which was discovered in related works, will be tested. These methods involves the use of support blocks and splitting the structure into sub-figures.

As time is limited, the method that adds support bricks to the structure is prioritized. The testing will be evaluated using the strategies mentioned in 2 for each approach and compared with each other at the end.

Chapter 4

Performing the experiments

In this chapter each step in the phase where we will perform the experiments is thoroughly discussed, this involves: How the model-driven approach was first tested, in ThingML, then in RoboDK with QT creator, and then with physical robots, including the usage of the RoboDK's API; How the implementation of the model-driven approach was done to test each scenario; How the implementation of the direct approach was done in RoboDK.

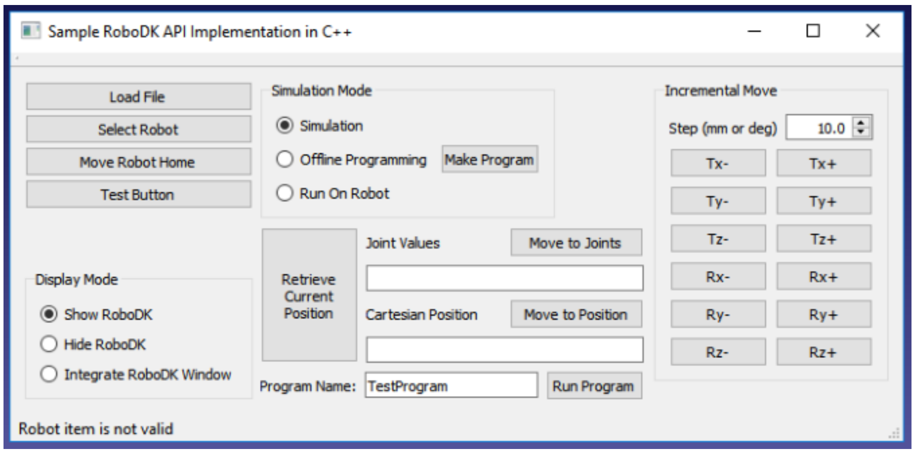
4.1 Step by Step, from ThingML to Qt to RoboDK

The first step in the model-driven approach was to consider the options of available programming languages for RoboDK, which languages can be used and if some had to be used in combination with each other. As described in chapter 3, ThingML will be used as the domain-specific language (DSL) for the model-driven approach and supports platform specific code generation for: Arduino (C++), C for Linux, Java, Javascript and Go. Unfortunately python is not one of the supported languages which was the preferable option, especially for RoboDK. The reason for this is that python files can just be dragged directly into RoboDK's interface or added directly from the interface itself and everything is ready to go (assuming the python code compiles without errors). RoboDK do support C++ but in order to utilize their C++ API we have to use an additional IDE called QT Creator which RoboDK provides API files for as well as a premade GUI application which interacts with the RoboDK interface, figure 4.1 shows how this GUI looks like.

Going the C++ route has its pros and cons. One of the cons, which we already have discussed is the requirement of an additional IDE. Another con is that ThingML does not really have a pure C++ compiler, at the time of performing these experiments we have to make do with the Arduino or C for Linux compiler. Another con is that, from a developer point of view, RoboDK seems to be in more favor of Python in terms of documentation, support and examples compared to C++ which is very limited. The pros of going the C++ route is that RoboDK provides, as mentioned earlier, an example that uses a premade GUI. This GUI can be easily altered to fit any need we may have simulation-wise which we will come back to later in the scenarios.

4.1.1 Model to Source Code Transformation

The next step was to find a way to use the C++ options that are available for code generation with ThingML. First off was the Linux option, this option was early on regarded



Simulation Mode: [✓ Simulation] Offline Programming Run On Robot

Display Mode: [✓ Show RoboDK] Hide RoboDK Integrate RoboDK Window

Step (mm or deg) 10.0

Program Name: TestProgram

Robot item is not valid

Figure 4.1: The QT application provided by RoboDK uses a GUI to interact with software

as not suitable for our needs. The reason for this is that while ThingML generates files with C code, to use the files however, a makefile is also generated which needs either Linux or a GCC compiler for windows to run and there was no available option at this time to combine this with the RoboDK API to make a solution that worked. It was also not possible to pull out code from the files to use them in a seperate cpp file in QT as this would call for missing package errors that only were resolved using Linux or a GCC compiler. It may or may not have been an option to find and replace those packages to make the code runable in a seperate cpp file but this required heavy tampering with the generated code which should be avoided in model-driven engineering.

The second option was to explore the possibility of using the generated arduino files which proved to work very well. There is just a few things one have to note going this route. The generated arduino code uses a datatype called byte, the QT application will not recognise this datatype. To find a way around this, a test sample where generated for C for Linux as well since this solution uses the byte datatype as well, what is different here is the use of "typedef unsigned char byte" which is located in the header files. Adding the "typedef unsigned char byte" on top of the arduino generated code resolved the issue of QT not recognising the byte datatype. Lastly, in order to run the arduino code in QT we have to call two functions in the generated code, setup() which initializes a configuration that is set up in ThingML, and loop() which is a simple while loop to keep the generated code running. To call upon setup and loop, a function can be added at the bottom of the generated code which can be set to interact with RoboDK GUI in QT. This way, no tampering of the generated code is needed. Figure 4.2 shows an example of how the code in ThingML will travel to be able to operate with RoboDK in the modeling solutions.

4.1.2 From ThingML to C++ to RoboDK

In this chapter we will shortly go through the full process of coding in ThingML, generate a solution to arduino code and implement it in QT to work with RoboDK's C++ API. To test this process, a simple example provided by ThingML at their github page was tested. This example consists of a single Thing with a statemachine containing two states called Greetings and Bye. Upon entering the Greetings state it will transition over to the Bye

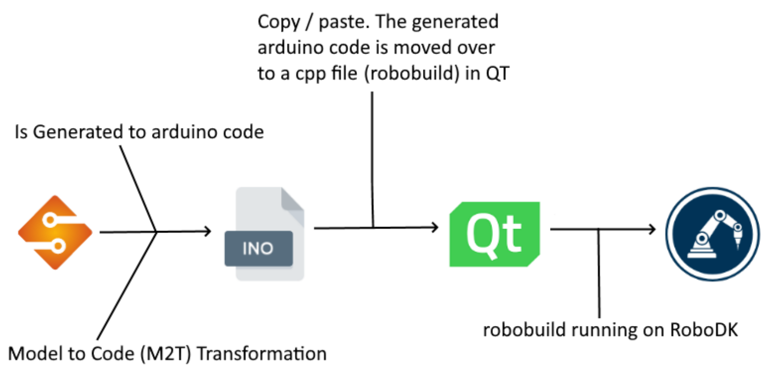
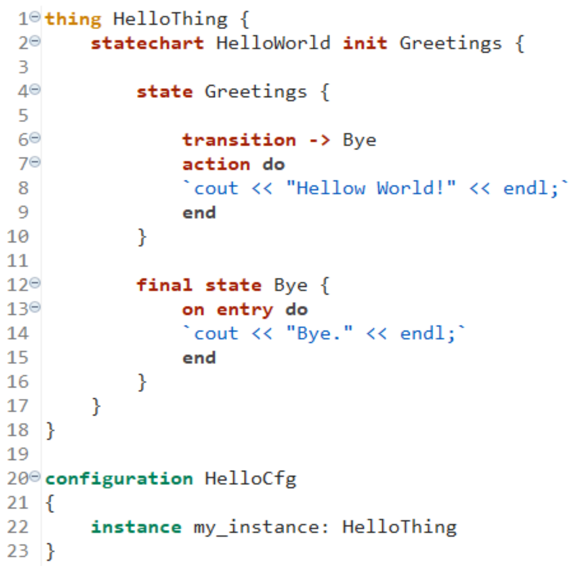

Copy / paste. The generated arduino code is moved over to a cpp file (robobuild) in QT
Is Generated to arduino code
INO
Qt
Model to Code (M2T) Transformation
robobuild running on RoboDK

Figure 4.2: The figure shows a overview over how the code will travel from ThingML to operate with RoboDK

state and print "Hello World" in the console window. Upon entering the Bye state it will print "Bye." in the console, see figure 4.3 to see how this ThingML code looks like.



```
thing HelloThing {
    statechart HelloWorld init Greetings {

        state Greetings {

            transition -> Bye
            action do
            `cout << "Hellow World!" << endl;`
            end
        }

        final state Bye {
            on entry do
            `cout << "Bye." << endl;`
            end
        }
    }
}

configuration HelloCfg
{
    instance my_instance: HelloThing
}
```

Figure 4.3: The figure shows the ThingML code of the Hello World example.

Once the ThingML coding is done it will be run through the ThingML to Arduino compiler, if everything went well a ino file will be generated. The next step is to create a cpp file and add it as a source file to the already made QT application provided by RoboDK. This application contains 6 files in total: Two header files, one for the main window and one for the robodk api which contains the class structures; Three source files for starting the application, which in turn loads the GUI and launches the RoboDK software, including a second robodk api file which contains the functions for movements, object referencing, error checks, etc.; And lastly a ui file containing the actual GUI which can be edited with additional buttons, menus, textboxes etc. See figure 4.4 for an overview of these files as shown in QT Creator.

To connect the newly made cpp file which will contain the generated code a additional header file is made, for context we will call the cpp file robobuild and the header file

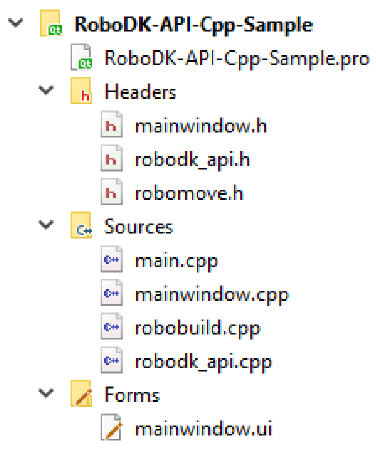


Figure 4.4: An overview of the files in the provided QT application from RoboDK.

robomove. Within robomove a class object called "TASK" is created. Next, we will split (not literally) robobuild into three portions. What we mean by this is that robobuild will always have custom code on top and bottom, figure 4.5 shows an overview of the robobuild file and the three portions and what each portion contains.

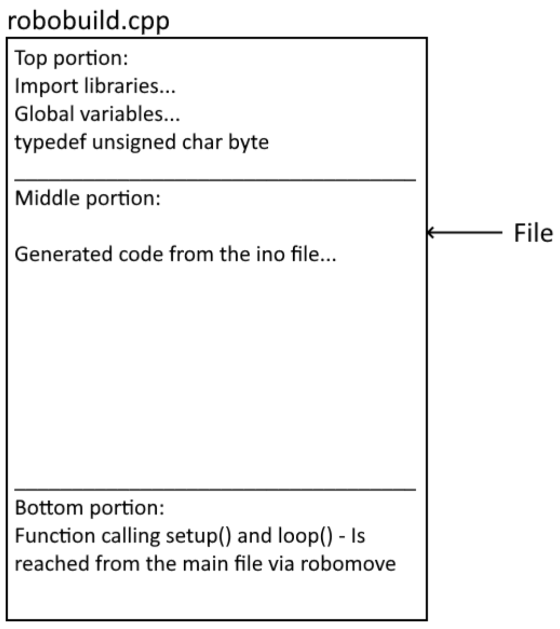

robobuild.cpp
Top portion:
Import libraries...
Global variables...
typedef unsigned char byte
Middle portion:
Generated code from the ino file...
File
Bottom portion:
Function calling setup() and loop() - Is reached from the main file via robomove

Figure 4.5: In order to make the generated code work in QT it is first copied over to a cpp file called robobuild. Then we need to add some custom code in order to run the code without errors, these codes are added at the top for importing packages, global variables, etc., and bottom for the function that will initialize the generated code

As mentioned earlier we need to add "typedef unsigned char byte" at the top to avoid errors using the byte datatype, and a function at the bottom which purpose is to call two functions (setup() and loop()) in the generated code from a button click in the GUI, once this is done the generated code can be copied over from the ino file and inserted in between the top and bottom portion. The bottom portion function will be connected to the main source file via robomove as a part of the object TASK, looking at listings: 4.1, 4.2, 4.3 we can see this in more detail. In the mainwindow source file we declare a TASK object after a button in the GUI is clicked and initializes a TASK specific function called Robobuild see listing 4.1, in listing 4.2 we can see this function in the header file robomove, and in listing 4.3 we can see the function located in the bottom portion of robobuild which is connected to robomove. Now the generated code is ready to be used in QT.

Listing 4.1: A object TASK is made which is used to access the function which initializes the generated code in robobuild

```
void MainWindow::on_btnThingMLRun_clicked(){
...

TASK buildWithThingML;

buildWithThingML.Robobuild();

...
}
```

Listing 4.2: An early stage of the header file robomove

```
class TASK : public QObject{
    Q_OBJECT
public:
    explicit TASK(QObject *parent = 0);

    void Robobuild();
};
```

Listing 4.3: The function located at the bottom portion in robobuild.cpp which is connected to the robomove header file

```
void TASK::Robobuild() {
    setup();
    loop();
}
```

To summarise: first we simply just create a cpp file in QT and divide it into three parts. On top we import the necessary packages, set up necessary global variables (if needed) and include "typedef unsigned char byte". At the bottom we have the function which is called from a button press in the GUI or something similar, this function calls upon setup and loop. Technically, the top and bottom portions will never change unless some solutions needs to use some global variables which is simply just added at the top portion. From the arduino file containing the generated code we simply just copy paste everything and insert it between the top and bottom portion, figure 4.6 shows this process which is also shown earlier in less detail in figure 4.2.

4.2 The RoboDK API

As the RoboDK API contains a multitude of different functions to achieve tasks ranging from painting parts with a spray gun to picking and placing objects from a conveyorbelt

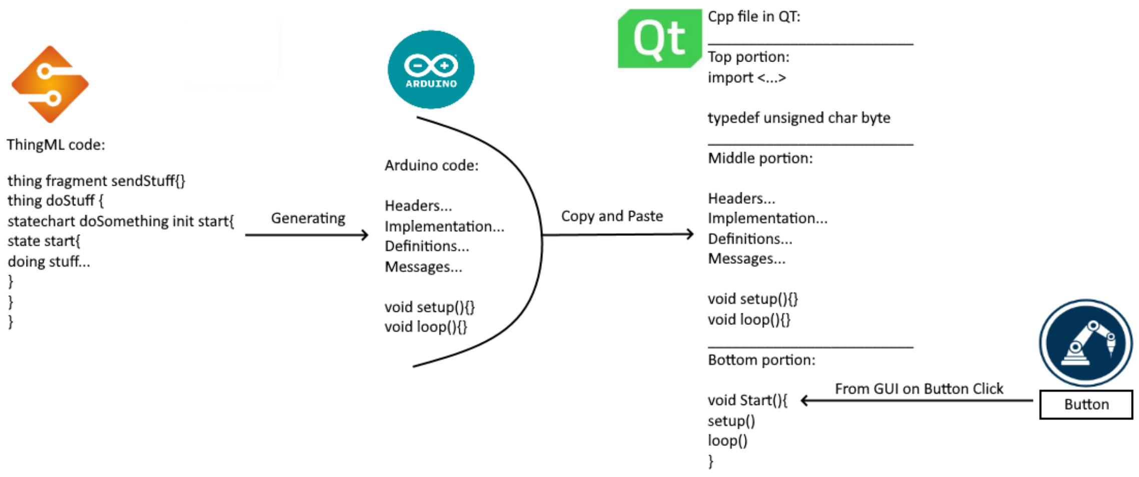


Figure 4.6: The figure shows how ThingML code are generated into arduino code and copy pasted into a cpp file in QT. This cpp file is divided into three portions where the middle portion contains the generated code. The bottom portion involves a function which initializes the generated code, this function is called upon when a button is clicked in the GUI.

with the help of a camera, this chapter will only highlight the necessary functions needed to get as close to our goals as possible. Whether the RoboDK API is being accessed through packages in python or using the provided c++ API files, the datatypes, classes and functions is for the most parts the same, making it fairly easy to adapt to the c++ supported API based on previous experience with python. Within the provided cpp files, all objects within RoboDK's virtual environment is defined as a datatype called "item", this includes robots, reference frames, 3D models, etc. The current RoboDK environment which the script or file refers to is defined as a datatype called "rdk". In QT creator the RoboDK environment (rdk) is declared at the very start after robodk is launched, all functions that are called later on will then point at the robodk environment that were launched when starting the application in QT creator.

To access reference frames, objects, robots etc. a new object item needs to be defined, then the function "getItem" within the Item class is called. Using the C++ API, this is done as such:

```
Item *varname = new Item(RDK->getItem(name, type));
```

Example when adding and retrieving Frames. These frames, also refered to as reference frames, are 3 Dimensional and can have robots, objects and other frames placed within them. Every item placed within a frame have their position defined as X, Y, Z coordinates within the frame. A frame can be added in RoboDK by pressing the button, as shown in figure 4.7, in the toolbar. To access the frame via code, the name has to match the name of the targeted frame and the type needs to be defined as: "RoboDK::ITEM_TYPE_FRAME".

```
Item *frame = new Item(RDK->getItem("framename", RoboDK::ITEM\_TYPE\_FRAME));
```

Items that are defined as tools in RoboDK is attached to the head of the robot arm and can be detached or switched with another tool. To access tools, the type needs to be "RoboDK::ITEM_TYPE_TOOL:". The following example shows how this is used to access a 3-finger gripper tool in QT:

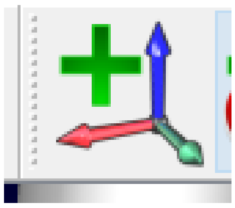


Figure 4.7: Button for adding a frame in RoboDK

```
Item *tool = new Item(RDK->getItem("3-Finger_Gripper_open", RoboDK::ITEM_TYPE_TOOL));
```

Further, the same procedure is exactly the same for adding robots and 3D models, the item type just have to be changed. For robot the item types is: RoboDK::ITEM_TYPE_ROBOT and for 3D models it is: RoboDK::ITEM_TYPE_OBJECT. Apart from manually adding reference frames in RoboDK using the button on the toolbar as shown in figure 4.7 we will otherwise only interact with RoboDK using the RoboDK API. Following is a list of tables with every RoboDK API specific functions used in each scenario.

Function: getItem	
Parameter	QString, int
Parameter Name(s)	name, itemtype
Parameter Description	Filter by item type RoboDK.ITEM_TYPE_...
Summary	Returns an item by its name. If there is no exact match it will return the last closest match.

Function: setParent	
Parameter	Item
Parameter Name(s)	parent
Parameter Description	parent item to attach this item
Summary	Attaches the item to a new parent while maintaining the relative position with its parent. The absolute position is changed.

Function: setPose	
Parameter	Mat
Parameter Name(s)	pose
Parameter Description	4x4 homogeneous matrix
Summary	Sets the local position (pose) of an object, target or reference frame. For example, the position of an object/frame/target with respect to its parent.

4.3 ThingML, what functionalities will be used

Before we enter the development phases of each scenario, we will go through a short list of components that will be used in this thesis to create a functional thingml file that will

Function: setPoseAbs	
Parameter	Mat
Parameter Name(s)	pose
Parameter Description	4x4 homogeneous matrix (pose)
Summary	Sets the global position (pose) of an item. For example, the position of an object/frame/target with respect to the station origin.

Function: setGeometryPose	
Parameter	Mat
Parameter Name(s)	pose
Parameter Description	4x4 homogeneous matrix
Summary	Sets the position (pose) of the object geometry with respect to its own reference frame. This procedure works for tools and objects.

Function: setPos	
Parameter	double
Parameter Name(s)	xyz[3]
Parameter Description	X, Y and Z coordinate within a matrix
Summary	Sets the position of an object, target or reference frame with respect to its parent.

Function: AddFile	
Parameter	const QString& , const Item*
Parameter Name(s)	filename, parent
Parameter Description	filename: absolute path of the file. parent: parent to attach. Leave empty for new stations or to load an object at the station root.
Summary	Loads a file and attaches it to parent. It can be any file supported by robodk. Returns newly added object.

Function: Pose	
Summary	Returns the local position (pose) of an object, target or reference frame. For example, the position of an object/frame/target with respect to its parent. Returns a 4x4 homogeneous matrix (pose)

Function: MoveL	
Parameter	const Mat&, bool
Parameter Name(s)	target, blocking
Parameter Description	target: pose target to move to. It must be a 4x4 Homogeneous matrix. blocking: True if we want the instruction to block until the robot finished the movement (default=true)
Summary	Moves a robot to a specific target (”Move Linear” mode). By default, this function blocks until the robot finishes its movements.

compile. A thingml file can compile with just one single Thing. A Thing in ThingML represents a component which can be instanciated through a configuration. A Thing may

Function: AttachClosest	
Summary	Attach the closest object to the tool. Returns the item that was attached. Returns attached item.

Function: DetachAll	
Parameter	Item
Parameter Name(s)	parent
Summary	Detach any object attached to a tool.

Function: Stop	
Summary	Stops a program or a robot

Function: Scale	
Parameter	double
Parameter Name(s)	scale
Parameter Description	scale to apply as scale or [scale$_x, scale_y, scale_z$]
Summary	Apply a scale to an object to make it bigger or smaller. The scale can be uniform (if scale is a float value) or per axis (if scale is a vector).

Function: PoseTool	
Summary	Returns the tool pose of an item. If a robot is provided it will get the tool pose of the active tool held by the robot. Returns a 4x4 homogeneous matrix

Function: PoseFrame	
Summary	Returns the reference frame pose of an item. If a robot is provided it will get the tool pose of the active reference frame used by the robot. Returns a 4x4 homogeneous matrix (pose)

include a statemachine, ports, messages, properties and functions. We will go into more detail about messages, ports and other functionalities used in this thesis later on in this chapter. Functions in Things acts the same as functions in most languages, it is its own snippets of code that are only used when called upon, it can utilize parameters and return values as well. Properties is how variables are declared inside a Thing and are declared as such:

```
property {Name of Variable} : {Variable Type}
```

,

so say we want to create an Integer:

```
property var : Integer = 3
```

4.3.1 Messages

In ThingML, each state can transition into a new state given a specific criteria. Such a criteria can be defined by the occurrence of a event and/or the guard (which acts as a if statement) is true. These events is set to trigger when a message is received from another statemachine. To provide an example of this, say we are monitoring a system for regulating temperature inside a room, for this we have have two state machines. One state machine for pulling information from the sensors as well as turning the oven on and off (SM1) and one state machine which requests information about temperature and determines whether the oven should be turned on or not (SM2). SM2 will frequently send a message to SM1 requesting the current reading from the temperature sensors. SM1 the moment it receives the message, will read the current sensors and send a message back with the temperature reading. SM2 receives the temperature and based on the readings will either enter a state for increasing the temperature, in which it will request the oven(s) to turn on, or wise versa. To send and receive messages we create fragments. Fragments are essentially Things and are declared as a Thing but they can not be instantiated, but rather they can be included in another Thing[REF]. Inside a fragment we declare messages. A message can be just a simple signal or it can carry with it variables. A fragment with a simple message can be declared as such:

```
thing fragment NameFrag{
    message messageName()
}
```

To include this fragment in another Thing:

```
thing DoStuff includes NameFrag{
    //statechart ... init ... etc
}
```

To send or receive a message we need to utilize ports, for this purpose we use RequiredPort and ProvidedPort. Ports in general are means of communication between Things in ThingML. The difference between Required and Provided is that we need RequiredPort for sending out messages from a Thing, while we use ProvidedPort for receiving Messages. RequiredPort can also be used for receiveing messages but is, as mentioned earlier, needed for sending a message. So if we want to send a message from a Thing we do:

```
thing DoStuff includes NameFrag{
    required port outgoingMessage{
        sends messageName
    }
    //statechart ... init ... etc
}
```

Or if we want to receive message:

```
thing DoStuff includes NameFrag{
    provided port incomingMessage{
        receives messageName
    }
    //statechart ... init ... etc
}
```

If we

Sending a message can be done when we enter or leave a state, when transitioning to a new state or inside functions. Sending a message is declared as such: Name of port!Name of message(), by using code examples above we can do:

```
thing DoStuff includes NameFrag{
    required port outgoingMessage{
        sends messageName
    }

    statechart doingStuff init Start
    {
        state Start{
            on entry do outgoingMessage!messageName
        }
    }
}
```

The code above will send a message upon entering the state called Start. For receiving messages we add events which is a part of the requirement for a state to transition to another state as mentioned earlier, see example below for a Thing receiving messages.

```
thing DoStuff includes NameFrag{
    provided port incomingMessage{
        receives messageName
    }

    statechart doingStuff init Start
    {
        state Start{
            transition -> End
            event incomingMessage?messageName
        }

        state End{

        }
    }
}
```

Here the state Start will only enter the state End if ”messageName” has been sent from either the same Thing or a different Thing. One thing to note is that a message can be a property or be sent with parameters this will be utilized later on in the scenarios.

4.3.2 Configuration

In the configuration we declare which Things that are to be instantiated upon start. If we want to send and receive messages between Things we establish the connection here in the configuration, this is done by defining from which Things messages will be traveling from to itself or other Things. To instantiate a Thing we define the configuration as such:

```
configuration configName{
    instance instanceName : ThingName
}
```

To add on the examples provided earlier for fragments and messages, for Things to send and receive messages we declare them as such in the configuration:

```
configuration configName{
    instance instanceDoStuff : DoStuff
    instance instanceDoOtherStuff : DoOtherStuff

    connector instanceDoStuff.outgoingMessage =>
              instanceDoOtherStuff.incomingMessage
}
```

4.3.3 ObjectTypes

In ThingML we can create custom datatypes that represents datatypes in the lower level. For example, we can create a mockup RoboDK datatype in ThingML type, which we otherwise use in C++, by first creating defining a "object" which we can name whatever we want. What sort of object this will be acting as in the lower level is defined by a type, which refers to the language we are using in the lower level. Since we are programming in C++, this type is defined as: c_type. The last thing we need to do is define is what sort of datatype the object will look be in C++. Since these objects only will work on the lower level, in this case c++, it is not possible for the ThingML compiler to check if they are legitimate C++ datatypes so it is important to make sure that the datatype is correctly written and that on the lower level the correct packages are imported.

4.4 Scenario 1

To summarize shortly for scenario 1, a manufacturer wants to automate processes where a robot of type UR10 or Kuka are tasked to place bricks in a set pattern. In this first scenario, these patterns are defined by 2D grids where each cell defines a x and y coordinate for brick placements. There can be multiple different patterns defined in these 2D grids. The patterns can also be stacked on top of each other, meaning that the shape of the structure stays the same but the inside is built according to the defined pattern. The workspace where the building takes place consists of a table with a robot (either UR10 or Kuka), multiple baseplates for placing the bricks which are Lego Duplo bricks in size 2x2 and 2x4, and two dispensers which feeds the robot duplo bricks. The dispenser can hold up to 14 bricks and have a sensor indicating when the dispenser are empty. The robot must also move in such a manner where it does not collide with any objects, which could potentially destroy the structure in a real life scenario.

To tackle this scenario we have to consider three tasks: How will the robot get these patterns, How do we simulate the dispensers, and how should the robot move to avoid collision.

4.4.1 How will the robot receive the building pattern

The first stage where we need to figure out how the approaches will get these patterns and how they will figure out what to do with them is universal for both approaches. As the QT sample from RoboDK's github contains a Graphical Unit Interface (GUI), the GUI will be altered to contain a group box of radio buttons where each toggled button is declared as a brick placement. Each button has a x and y value which is declared as true in a 3D array of type boolean when a button called "add layer" is clicked. At the same time, a value z is increased by one indicating that next time "add layer" is clicked, it is the next layer that we are defining.

This is done by iterating through a list containing all the radio buttons, see listings 4.4. Figure 4.8 shows this groupbox of radio buttons with the "add layer" button. This method of defining the patterns is used in both the direct and the modeling approach does not play a big part in the actual building process. If the GUI was not available, another way of doing this could be to use files of type txt or json, etc with predetermined coordinates where for example each row is a coordinate for brick placement with 3 columns for x, y and z (which is utilized later).

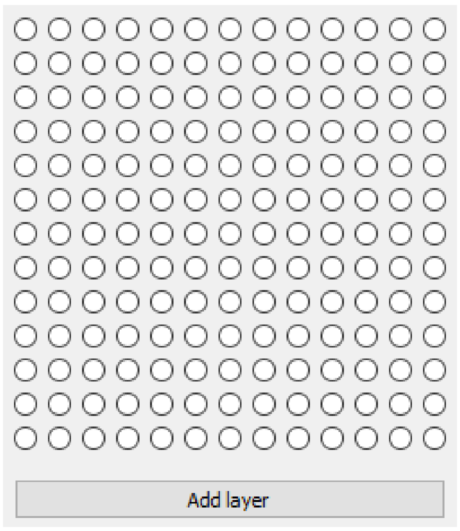


Figure 4.8: For the purposes of scenario 1 and testing, a group box of radio buttons was added to the GUI where each radio button contains an x and y value in a 2D grid.

Listing 4.4: The code initiated when the "add layer" button is clicked

```
//On button "add layer clicked"
for (it = radBtnList.begin(); it != radBtnList.end(); it++){
    if(it->rbtn->isChecked()){
        coord[z][it->xID][it->yID] = true;
        it->rbtn->setChecked(false);
    }
    else{
        coord[z][it->xID][it->yID] = false;
    }
}
z++;
```

To quickly summarize how the robot will receive the building patterns: A group box of radio buttons is added to the GUI. When toggled, a true value is declared somewhere in a 3D array. The true value indicates that a brick is to be placed somewhere using the position of the true value in the 3D array as x, y and z coordinates.

Following is the results of the robot building the patterns as shown in chapter 3.3.1. Figure 4.9, 4.10, 4.11 shows the results of the wall structures built by the robot. Figure 4.12, 4.13, 4.14 shows the results of the circle structures built by the robot. Figure: 4.15, 4.16, 4.17 shows the results of the concave structures built by the robot. And finally, figure: 4.18, 4.19, and 4.20 shows the results of the complex structures built by the robot.

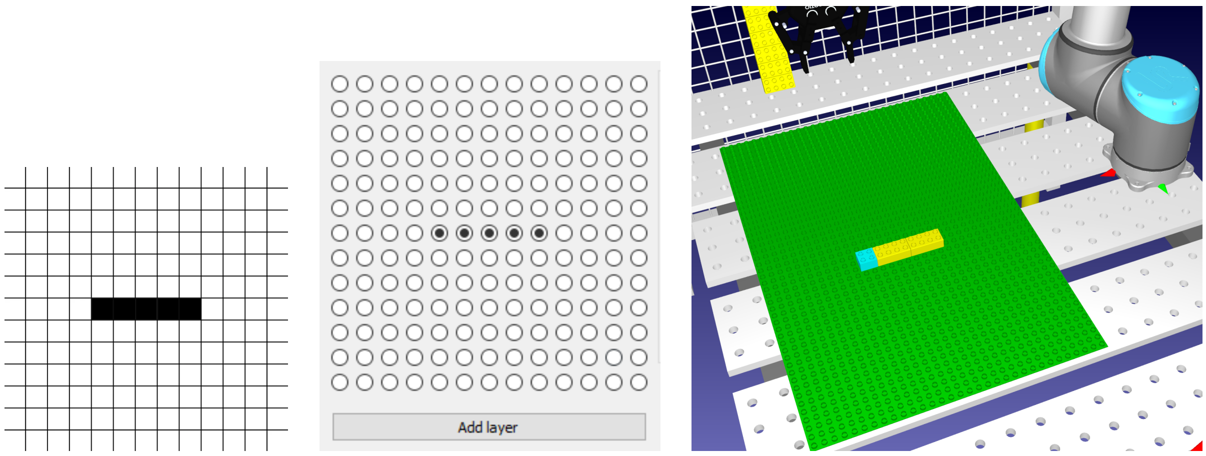


Figure 4.9: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

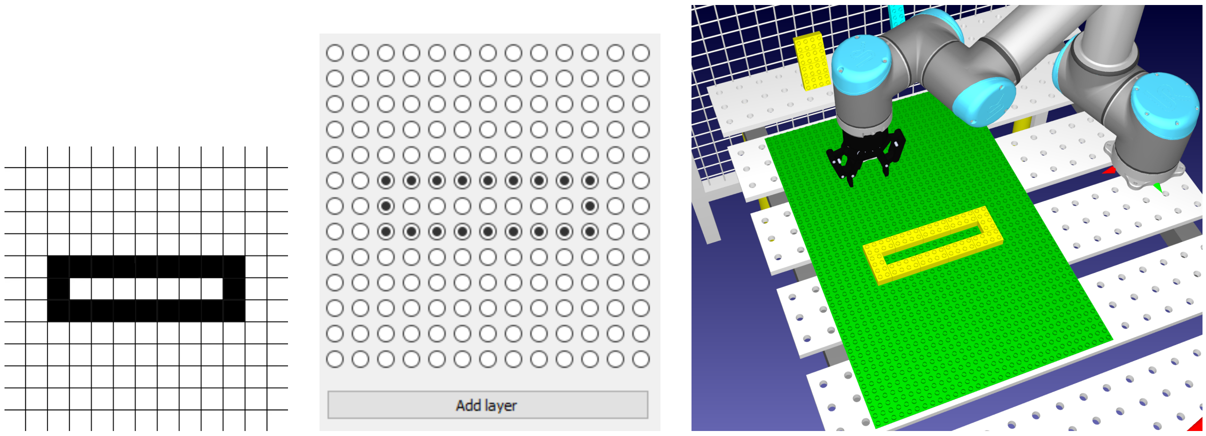


Figure 4.10: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

In the next two chapters a solution is developed in the direct approach using C++ and in the modelling approach using ThingML. In the approaches, a solution is created for how the dispensers are simulated, and how the robot picks up and places bricks in a non-colliding matter.

4.4.2 The direct approach

To start off, a simple algorithm is made for adding and sorting bricks in a list which the robot will utilize. The user defines a pattern by clicking on the radio buttons and then

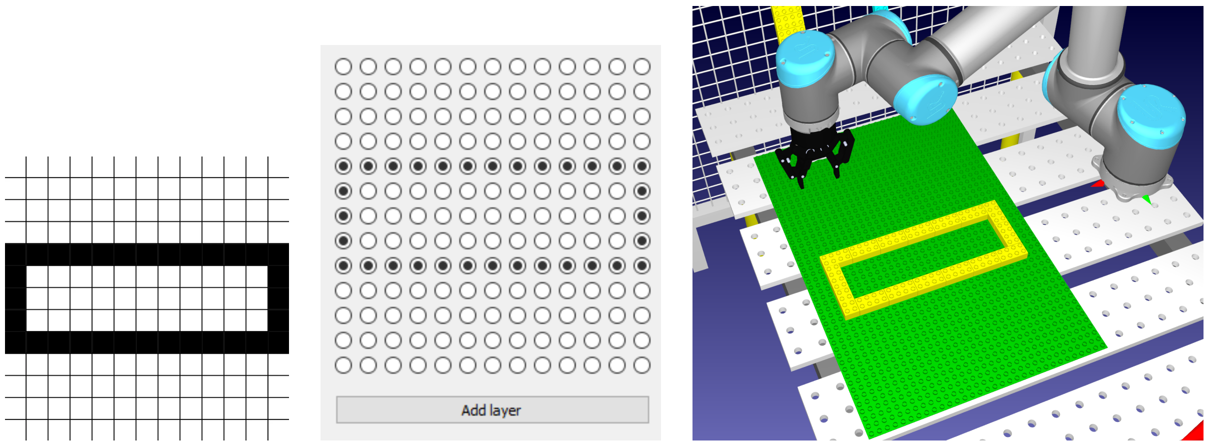


Figure 4.11: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

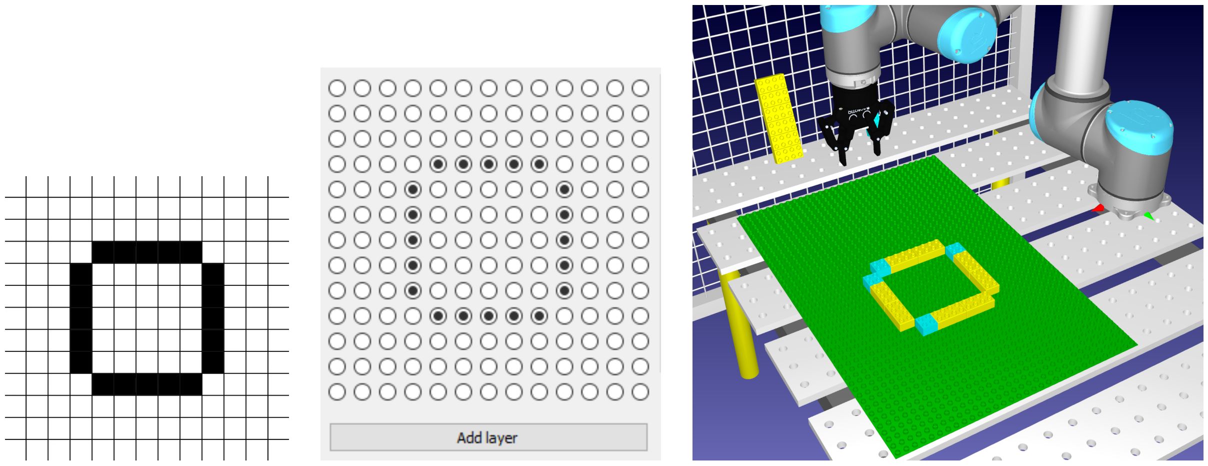


Figure 4.12: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

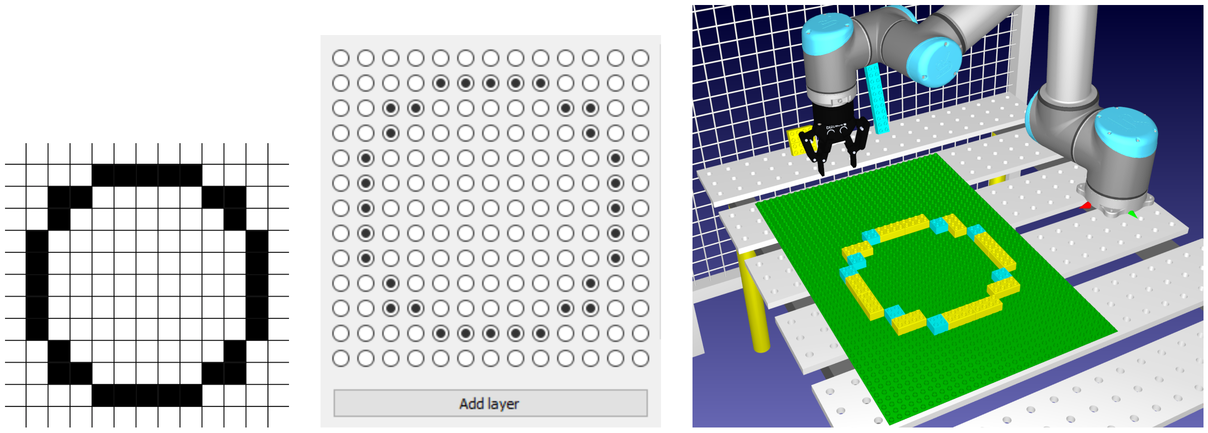


Figure 4.13: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

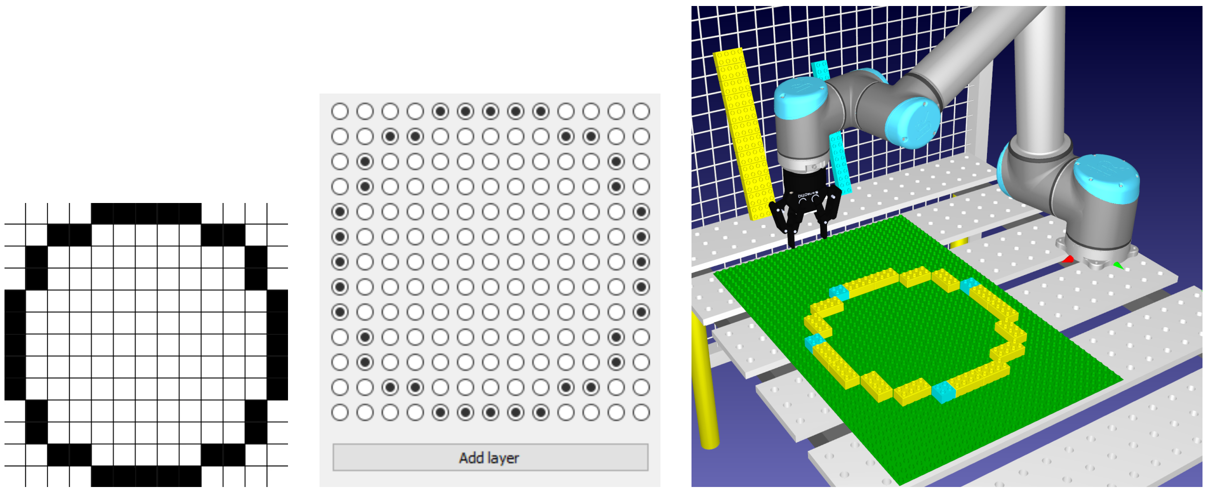


Figure 4.14: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

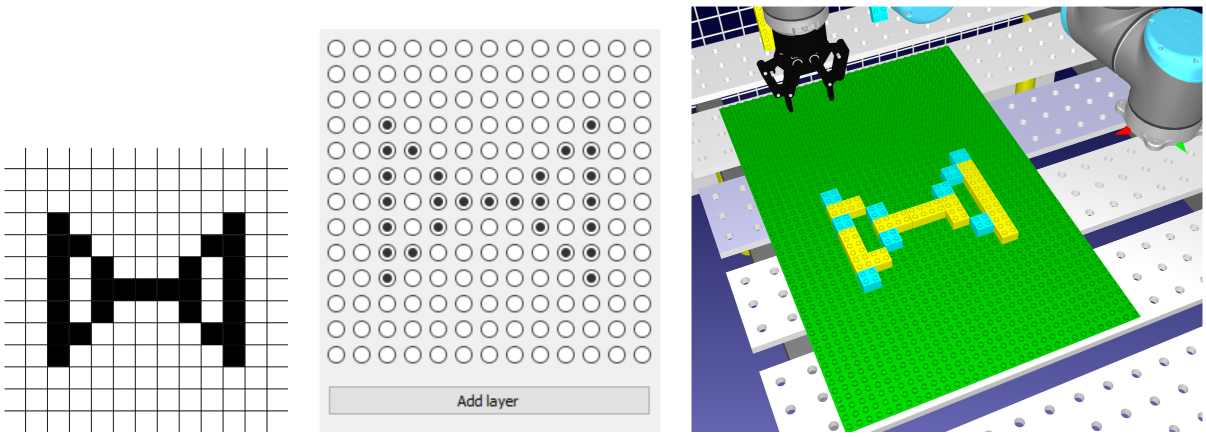


Figure 4.15: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

the pattern is saved to a 3D array. Iterating through the array is done by using a tripple-nested for-loop and once a true value is found then it will go to the next step. Duplobricks in this approach are declared as their own specific class. The Duplobrick class consists of 13 attributes:

- QString fileLocation, This is the filelocation of the 3D model which is imported into RoboDK.
- QString name, This is the name of model in the RoboDK environment, when we look for objects, frames, or robots in RoboDK from the code we need the name and the type of the item.
- QString dispenserName, This refers to the name of the designated dispenser in RoboDK which feeds bricks to the robot. "Dispenser Base" is the name of the dispenser that generates 2x2 bricks, while "Dispenser Base 2" generates 2x4 bricks.

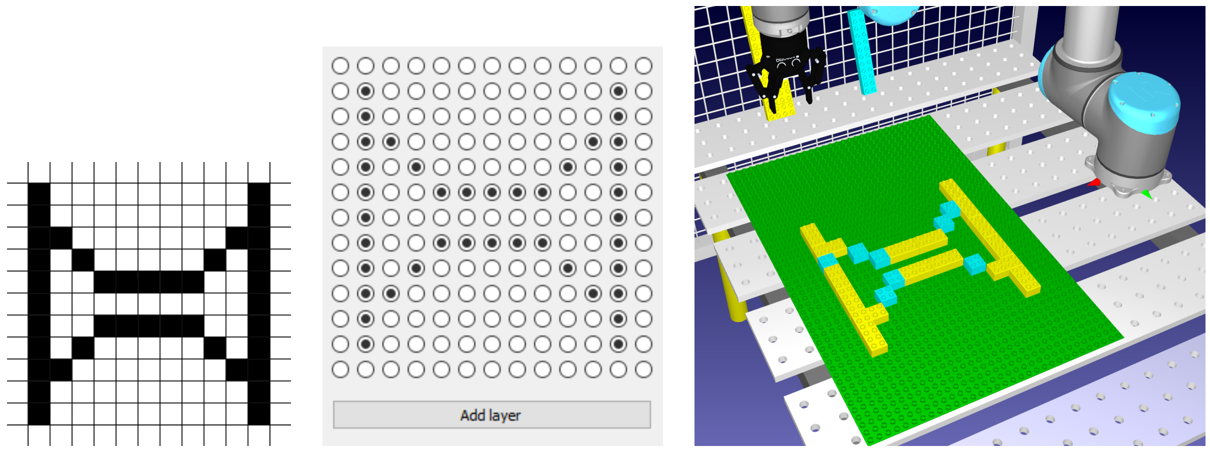


Figure 4.16: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

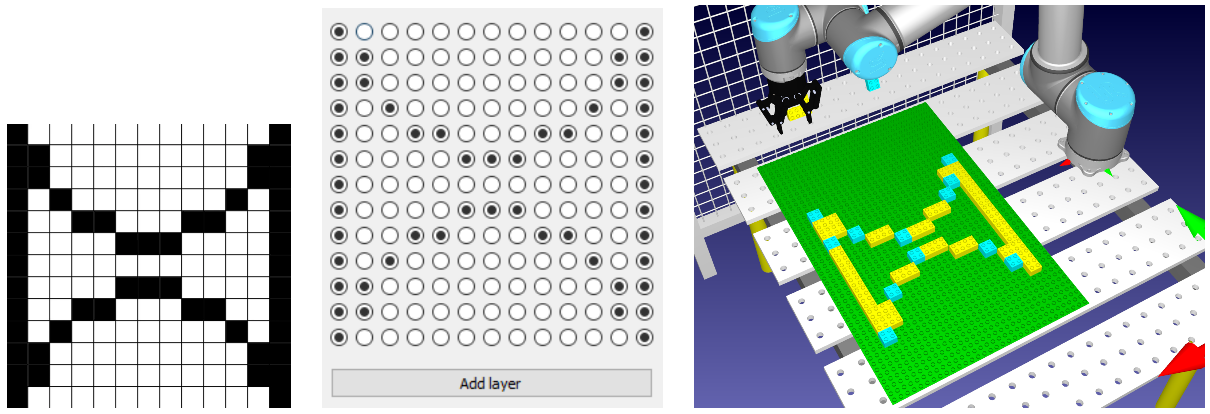


Figure 4.17: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

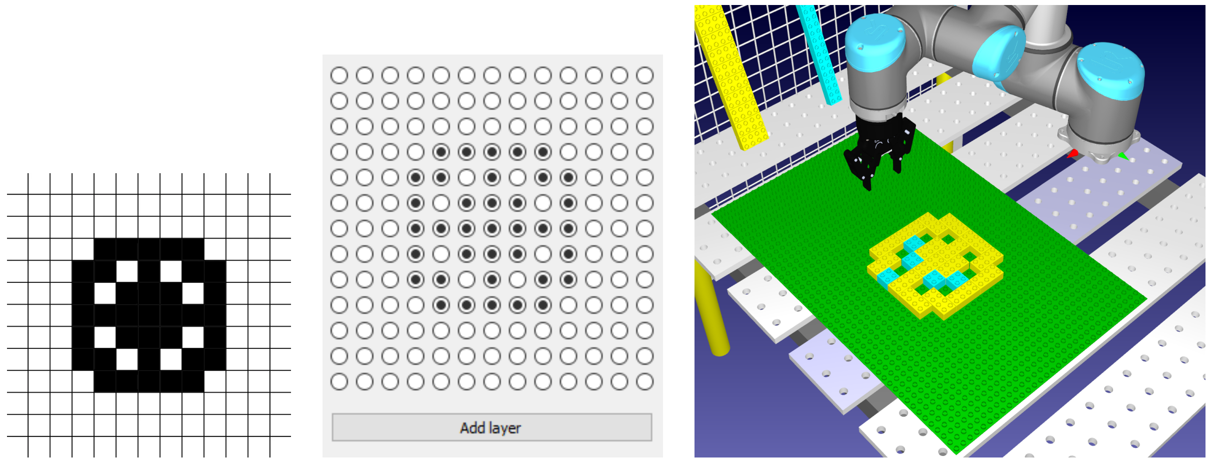


Figure 4.18: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

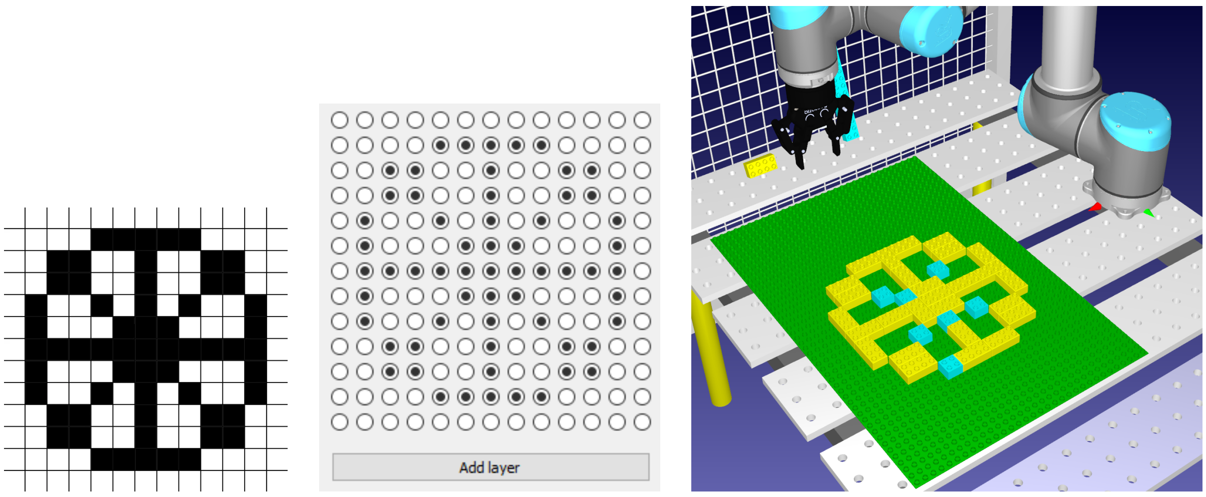


Figure 4.19: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

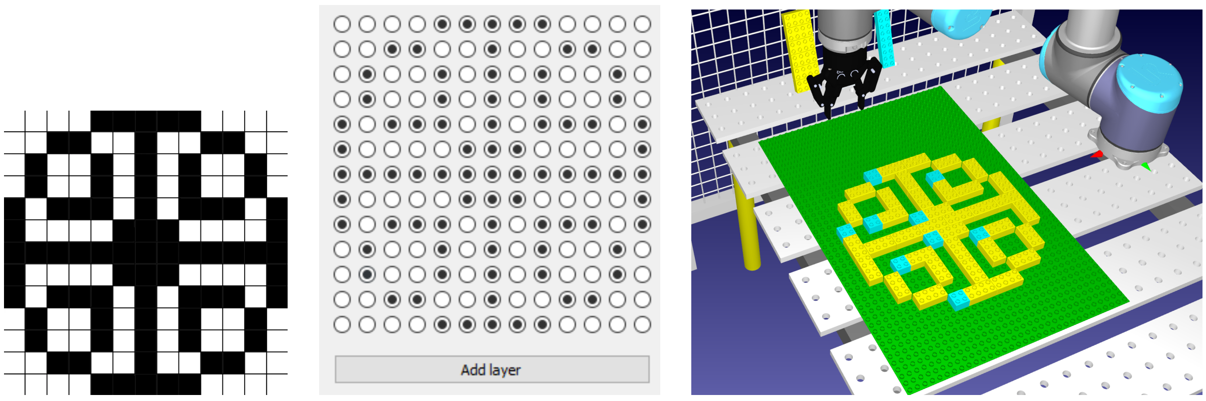


Figure 4.20: Left: The desired pattern. Middle: The radio buttons toggled to resemble the desired pattern. Right: The desired pattern built by the robot

- int studsXDir, This is the amount of studs in X direction. (From left to right).
- integer: studsYDir, This is the amount of studs in Y direction. (From top to bottom).
- integer: xCoord, This is the x coordinate from where in the array the algorithm found a true value.
- integer: yCoord, This is the y coordinate from where in the array the algorithm found a true value.
- integer: zCoord, This is the z coordinate from where in the array the algorithm found a true value.
- integer: height, This is the height of the brick in milimeters which RoboDK operates with. The height for a normal duplobrick is 19mm.
- integer: width, This is the width of the brick in milimeters. In the later scenarios, this value is declared as 16mm for both the 2x2 and the 2x4 Duplo bricks. The

reason for this is that later on each coordinate is equal to the position of a stud in stead of a brick, therefor the width of the brick does not matter but the width of the stud does which is 16mm.

- integer: depth, This is the length of the depth of the bricks.
- double: rotation, this defines the rotation of which the brick will be placed. This is only used for bricks that are of size 4x2. A 4x2 brick may not always be able to be placed vertically but in these cases they might be allowed to be placed horizontally.
- int priority, This defines the priority of the bricks. A priority value defines which bricks should be placed first.

When iterating through the array and the program finds a true value, then it will first check if there is possible to place a 4x2 brick at the current coordinates. Since every position in the grid is defined as 2x2 brick, the program will check its neighbours to see if it finds a true value. The program iterates through the array horizontally from left to right. For example it finds a true value at X = 2, Y = 3. It will first check the neighbour to the right, in this case, X = 3, Y = 3. If the neighbour to the right is a true value then a 4x2 brick can be placed on position X = 2, Y = 3 horizontally facing. If the neighbour to the right turns out to be false then check the neighbour beneath, in this case, X = 2, Y = 4. If this neighbour is a true value then a 4x2 brick can be placed at position X = 2, Y = 3 vertically facing. If it turns out that both the neighbour to the right and the neighbour beneath is false then a 2x2 brick is placed at position X = 2, Y = 3. When the algorithm decides a brick is place-able, they are saved to a list with their assigned coordinates and rotation in case it is a 2x4 brick that will be placed vertically.

To quickly summarize: If it finds a value true at position z, x, y, then first declare duplobricks, one for each type which is 2x2 and 4x2, we do not know which we will be using yet. Then it checks the neighbours, the one to the right (x + 1), and the one beneath (y + 1) to see if a 4x2 is place-able vertically or horizontally. Else, the duplobrick will be a 2x2. The desired brick is then placed in the front of the list with the function: bricklist.push_back(*brick). Listing 4.5 shows what this algorithm looks like in the direct approach using C++.

Listing 4.5: This code snippets checks the neighbours to see if a 4x2 brick can be placed at current x and y position by checking the neighbour to the left (x+1) and the neighbour beneath (y+1). If a 4x2 brick placement is not possible then just assign a 2x2 brick.

```
if(bricklist.empty()){
        for(int z = 0; z < 30; z++){
            for(int x = 0; x < 30; x++){
                for(int y = 0; y < 30; y++){
                    if(coord[z][x][y]){
                        DuploBrick *brick = new DuploBrick();

                        DuploBrick *TwoByTwo = new DuploBrick();
                        TwoByTwo->name = "duplo_brick_2x2";
                        TwoByTwo->priority = 1;
                        TwoByTwo->width = 32;
                        TwoByTwo->depth = 32;
                        TwoByTwo->height = 19;
                        TwoByTwo->xCoord = x;
```

```
TwoByTwo->yCoord = y;
TwoByTwo->zCoord = z;
TwoByTwo->rotation = 0;
TwoByTwo->dispenserName = "Dispenser_Base";

DuploBrick *TwoByFour = new DuploBrick();
TwoByFour->name = "duplo_brick_2x4";
TwoByFour->priority = 1;
TwoByFour->width = 32;
TwoByFour->depth = 32;
TwoByFour->height = 19;
TwoByFour->xCoord = x;
TwoByFour->yCoord = y;
TwoByFour->zCoord = z;
TwoByFour->rotation = 0;
TwoByFour->dispenserName = "Dispenser_Base_2";

if(coord[z][x + 1][y]){
    brick = TwoByFour;
    coord[z][x + 1][y] = false;
}
else if(coord[z][x][y + 1]){
    brick = TwoByFour;
    coord[z][x][y + 1] = false;
    brick->rotation = 90.0;
    brick->yCoord = y + 1;
}
else{
    brick = TwoByTwo;
}

bricklist.push_back(*brick);
```

Next, when the user runs the program, several necessary RoboDK variables are declared before going through the list of bricks. First, a new instance of the RoboDK API is declared.

```
RoboDK *rdk = new RoboDK;
```

Then from within the new instance we get the robot, tool, and all the necessary reference frames. This uses the getItem function, see table 4.2.

```
Item *table_frame = new Item(rdk->getItem("Table Base", RoboDK::ITEM_TYPE_FRAME));
Item *tool = new Item(rdk->getItem("Gripper RobotiQ 85", RoboDK::ITEM_TYPE_TOOL));
Item *work_frame = new Item(rdk->getItem("Work Frame", RoboDK::ITEM_TYPE_FRAME));
```

These variables are necessary for determining the robots movement and for simulating the robot being able to pick and place bricks. One issue that was ran into using the RoboDK API was that simulation was only able to have one active frame. For context, the robot is only able to do movements inside an active frame and it was currently not possible to change the active frame during simulation. Therefore the simulation uses work_frame as the active frame at all time and we simply move the work_frame inside the desired frame from which we want the robot to move within.

The direction in which the robots gripper tool will point to is determined by the z direction (Blue arrow), see figure 4.21. In order to set the direction in which the gripper tool will be pointed towards is done by using the API function setGeometryPose, see table 4.2. The setGeometryPose has one parameter which in this case is the position (Pose) of the table_frame. Retrievieng the position of the table_frame is done using the Pose function, see table 4.2.

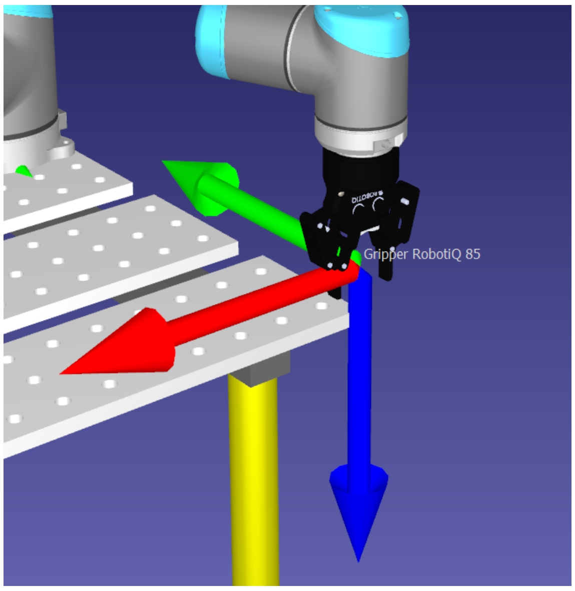


Figure 4.21: Figure shows the direction in which the robots gripper tool will point to. This is determined by the z value (blue arrow)

Now the robot is ready to begin working, the next procedure is to iterate through the list of bricks. Before the program checks the next brick in the list it will first make sure that the dispensers are not empty or if a emergency stop button is triggered (this is for scenario 2). Then a function for handling the behavior of the robot is called. This function is called with 13 parameters. These parameters consists of the RoboDK instance, robot, tool, frames, and information about the current brick in front of the list, see listing: 4.6. This function returns a list of the remaining bricks in the dispenser. Which dispenser this points towards is depends on the current brick from the list.

Listing 4.6: The function which is called for every brick when iterating through the list of bricks. This function returns how many bricks are left in the dispenser. Which dispenser depends on the current brick from the list

```
list<Item> pickUpBrick_and_Move(RoboDK *rdk, Item *robot, double rotation,
                                QString fileLocation, QString dispenserName,
                                QString brickName, Item *parent_frame,
                                Item *work_frame, Item *tool,
                                list<Item> dispenser, int x, int y, int z);
```

Inside the function pickUpBrick_and_Move from listing 4.6, the robot is programmed to move to a dispenser pick up a brick, then move and place the brick at the correct spot. This is done by first moving the active frame, in this case called work_frame over to the frame of the current dispenser depending on which brick is next on the list. This is done by setting the parent frame of the work_frame to the frame of the dispenser using a function called setParent, see table 4.2. Then the work_frame is moved to position x: 0, y: 0, z: 0

aligning perfectly with dispenser frame. Now the robot will move over and position itself in front of the bottom brick in the current dispenser. To simulate the robot picking up the brick, the function AttachClosest is used, see table 4.2. As the name suggests, the closest object is attached to the robot tool, in this case the gripper. Once the robot has picked up a brick, a new thread is initiated. In this thread a function for updating the dispensers are ran.

```
simulateDispenser = QtConcurrent::run(this, &MainWindow::UpdateDispenser,
                    dispenser, brickName);
```

This is to achieve concurrency and reduce down-time, by having the dispensers update while the robot is placing down a brick. Notice how the thread is initiated using QtConcurrent::run, the reason behind this is that QtConcurrent has functionality where we are able to check if the thread is finished or not. This is used because we have to make sure that dispensers are actually done updating before the robot picks up the next brick. To simulate the updating of the dispensers, a simple for loop is used. This for loop iterates through a list of the remaining bricks in the current dispenser and uses the function setPos, see table 4.2, to move all the bricks in the dispenser one brick down.

Lastly, when the robot has finished moving to the right position and the brick is aligned and placed correctly, to simulate the robot tool letting go of the of the brick the function DetachAll is used, see table 4.2.

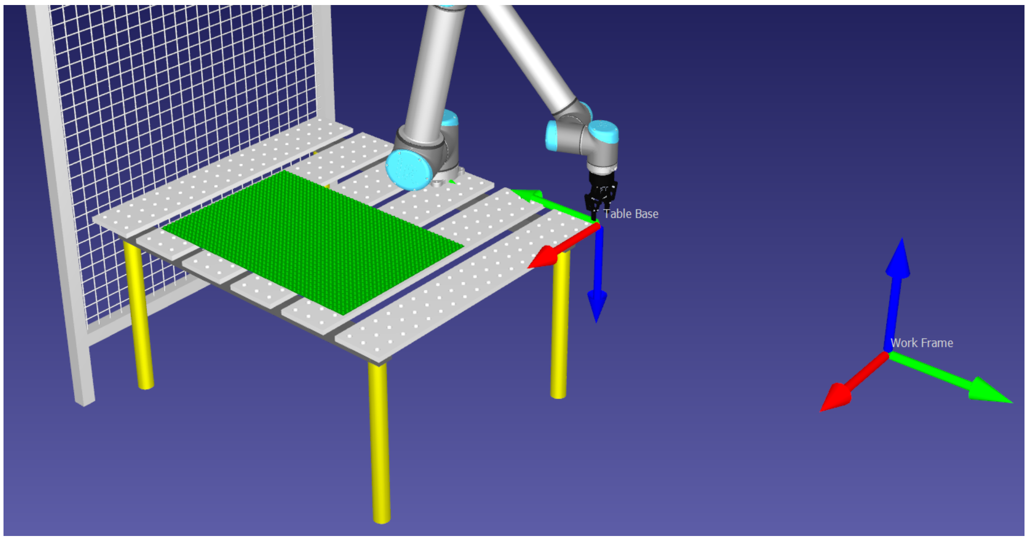


Figure 4.22: Figure shows the simulation at startup, the work_frame have not been assigned a new position yet.

4.4.3 Filling up the dispensers

Before we dive into the modelling approach, it will quickly be described how the dispensers are filled up in the simulation. This is done in a for loop that counts to 14 which is set to be the limit of dispensers in this thesis. Inside this for loop the function AddFile, see table 4.2, is used to add the brick 3D models into the simulation. To access the brick model in the simulation, the getItem function is used. Then the appropriate color is set so we are

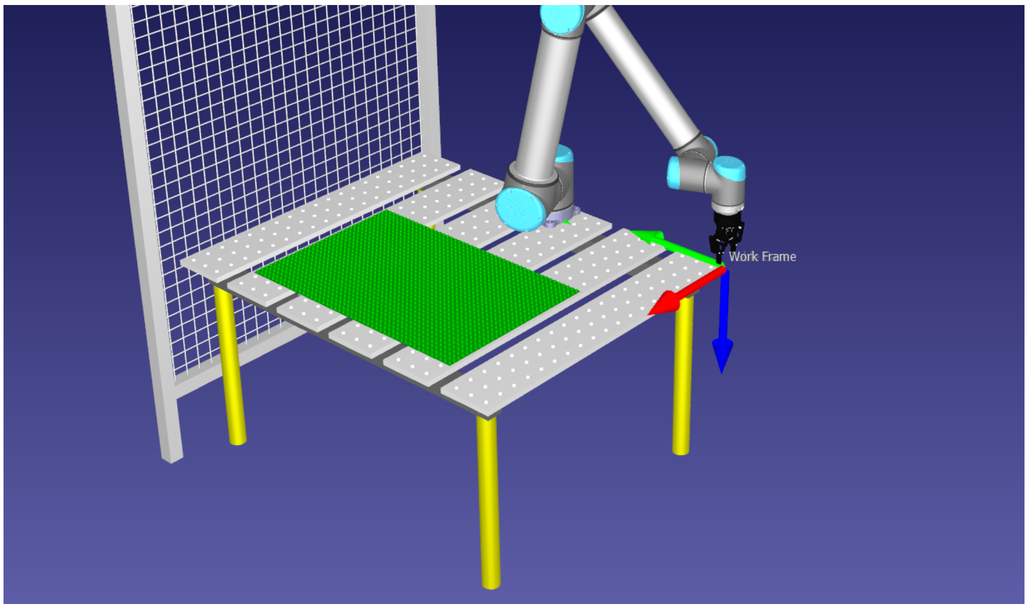


Figure 4.23: Figure shows the simulation after the program is run, the work_frame have been assigned the same position as the the table base reference frame.

able to distinguish them, in the simulation the 4x2 bricks are yellow colored and the 2x2 bricks are light blue colored. Then the setPose and rotate functions are used to place the bricks at their correct spots. At the end list is returned containing 14 bricks of either 2x2 or 4x2 depending on which dispenser was requested to be filled up.

4.4.4 The modelling approach. First solution

In this first solution in the modelling approach an attempt was made to recreate the algorithm in ThingML. While in the process of making a statemachine that would work the same way as the algorithm that was made in the direct approach, it was noted that ThingML did not support multidimensional arrays, and for-loops had to be treated a bit differently. The statemachine that will act as the algorithm starts by first entering a state called start, here all the necessary variables for RoboDK are declared. To keep everything organized, all code for RoboDK specifically was placed in functions, figure 4.25 shows the function that are called in the start state and declares all the necessary variables.

For handling the need of a multidimensional array and RoboDK functions and variables, the ThingML functionality called kickdown is used. Using kickdown we can use code from the lower level, in this case C++, in ThingML and the compiler will not complain. One thing to note however is that kickdown can be seen as a double-edged sword. The advantages is obviously that we are able to include lower level language (in this case C++) code inside the ThingML code without the compiler complaining about it but it does not check if the code is viable either. Therefore it is generally a good advice to make sure the kickdown code used actually works and compile correctly in the lower level.

After the initialization, the next state the program will enter is the state called "Iterate Layer". This state has four transitions. Two of these transitions are for iteration purposes only and use counters to determine where in the 3D array the program currently is.

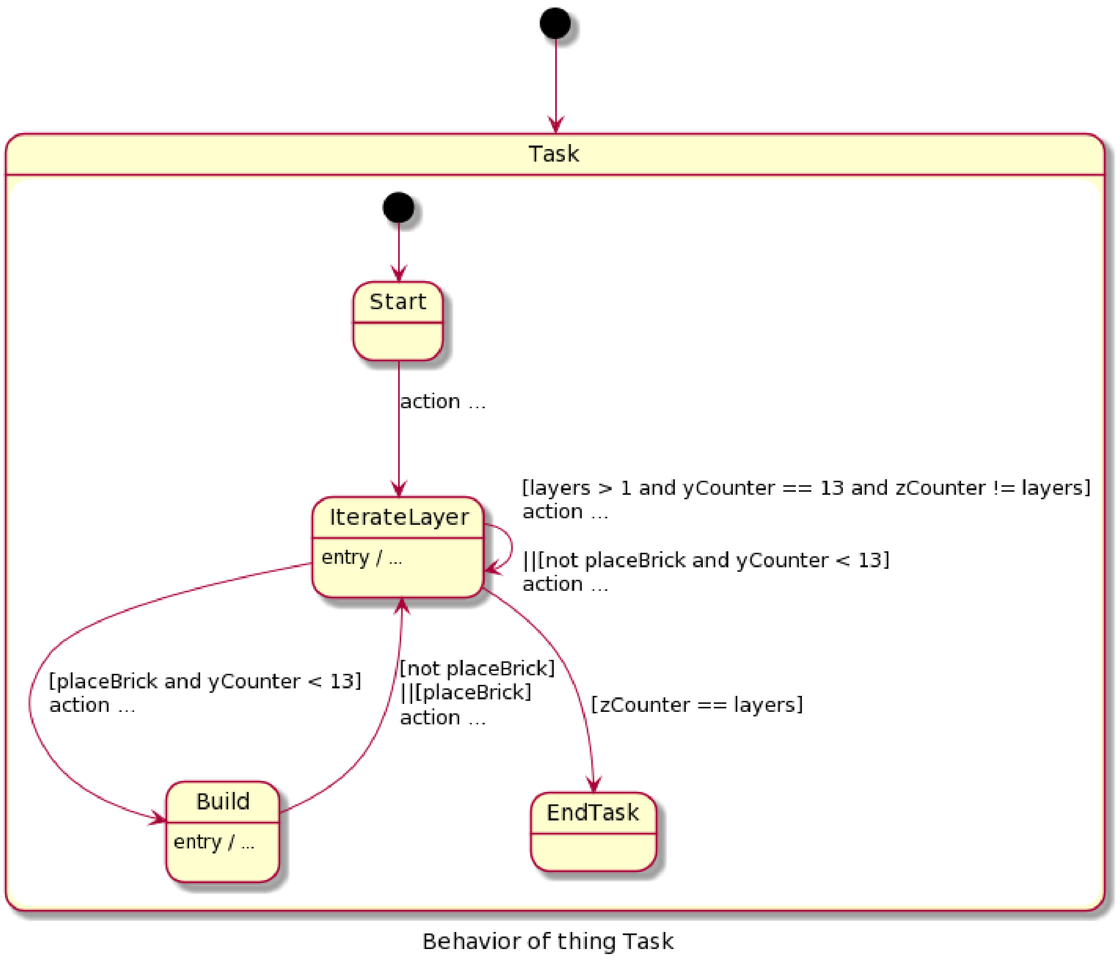


Figure 4.24: Statemachine for handling the algorithm in Scenario 1 created in ThingML, it acts the exact same way as a tripple-nested for-loop and uses counters for x, y and z for iterating through the array containing the coordinates.

```
//RoboDK Stuff
//Get the necessary Items from RoboDK and set robot variables
function declareRoboDKVariables() do
    `pose_ref = robot->Pose();`
    `robot->setSpeed(100);`
    `table_frame = new Item(rdk->getItem("Table Base", RoboDK::ITEM_TYPE_FRAME));`
    `dispenser_frame = new Item(rdk->getItem("Dispenser Base", RoboDK::ITEM_TYPE_FRAME));`
    `pose_frame_table = table_frame->PoseFrame();`
    `robot->setPoseAbs(table_frame);`
    activateTool()
end
```

Figure 4.25: This is a function within ThingML (modelling approach, first solution) that declares all the necessary variables for RoboDK, as this not code handled by ThingML kickdown is used. (Blue text)

Then we have a transition from IterateLayer to Build, the condition for this to happen is that the current value found in the array is true. For safety reasons, the Build state also checks if the current value the program is looking at in the array is true. The reason behind this is that sometimes during implementation the Iteratelayer state would transition over to the Build state even though the current value in the array was false, double checking this value in the Build state and transitioning back to the Iteratelayer state if the value was actually wrong fixed this issue. If the value was true it would transition back to the Iteratelayer state while doing the building process. See figure 4.24 for a full overview over this statemachine.

4.4.5 The modeling approach. Second solution

Before diving into the second solution of this approach, it should be noted that the idea of adding threads to achieve concurrency in the direct approach was only brought up after reconsidering the usage of ThingML and its features, which we will get into in this chapter.

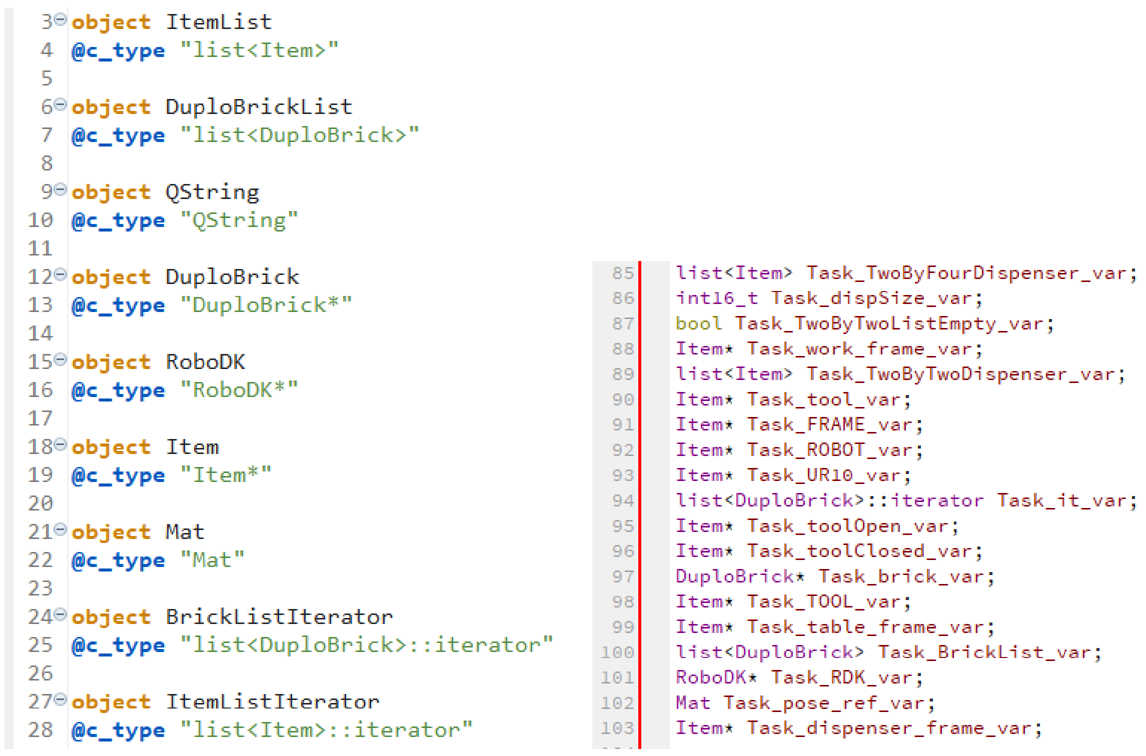
After the first solution it was soon realised that it was not the correct way to go, what was created was essentially just a translated version of the algorithm which works and can be used if one would desire to do so. However, we later realised that we should focus more on what parts of the developing process ThingML would be useful for. One of the key features with ThingML is its ability to have multiple state machines observing and performing tasks at the same time, in other words, working on several different tasks concurrently. For example, while the robot is building a structure, another statemachine can be running at the same time which monitors the brick dispensers. Once one of the brick dispensers are empty the state machine can send a message to the main program telling the robot to stop and wait until the dispensers are full again. This lead us to create a solution that uses two main state machines, one state machine that overlooks the building process and one statemachine that overlooks the dispensers making sure that there is a brick available for pick up until the dispensers are empty. At the start we also have one additional statemachine with a singular state which only purpose is to declare some variables needed for RoboDK, once the variables are declared it will send a message to the state machine that overlooks the building process to start building.

For this solution we first define custom objects that otherwise do not exist in ThingML but is a part of RoboDK or QT in the lower level language, in other words C++. We talked about how to make these objects in chapter 4.3.1. In this solution, a total of 6 objects are created, in figure 4.26 we can see to the left, the objects created for this solution, and to the right we can see generated variables using datatypes we defined in the objects we created in ThingML.

In total this solution utilizes five Things. Where two of the Things are fragments for messages and where the other three Things utilizes the fragments and includes statemachines, functions and ports. Figure 4.30 shows all the 5 Things. Here we can see the Fragments and how they are connected to each Thing.

First we will take a look at the Thing: RoboDKManager, figure 4.28 shows the statemachine in this Thing and its purpose is to declare all the necessary RoboDK variables at the start of the program, the declaring of the variables is done in a function called "declareRoboDKVariables" see figure 4.27. As we can see the function only consists of kickdown code. Functions in ThingML is a great way to handle kickdown code because generally we dont want to overcrowd the statecharts with it.

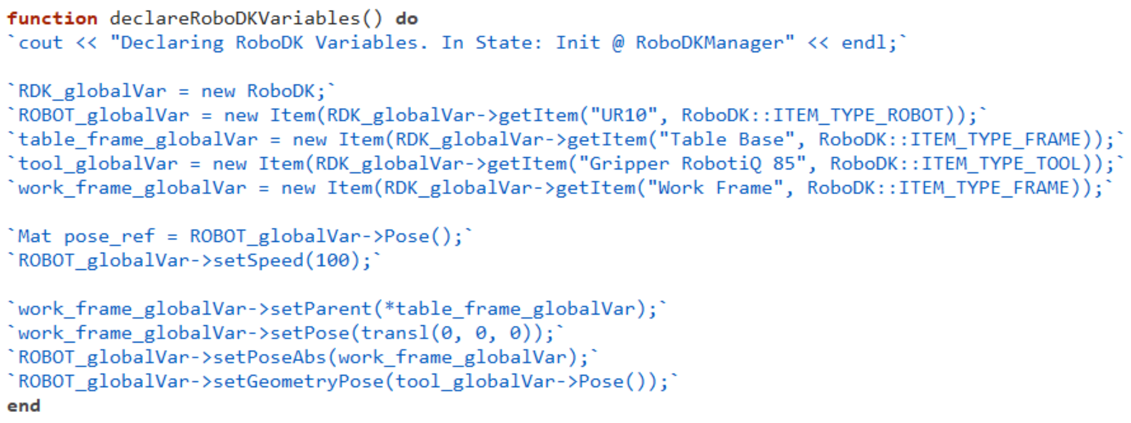
Looking at the kickdown code we can see that it declares some variables in the lower level only, some of which is called _globalVar at the end. This is because this solution relies on a lot of global variables due to not being able to send data between Things in ThingML. The initial thought when creating this solution was to have the Things pass



```
object ItemList
@c_type "list<Item>"

object DuploBrickList
@c_type "list<DuploBrick>"

object QString
@c_type "QString"

object DuploBrick
@c_type "DuploBrick*"

object RoboDK
@c_type "RoboDK*"

object Item
@c_type "Item*"

object Mat
@c_type "Mat"

object BrickListIterator
@c_type "list<DuploBrick>::iterator"

object ItemListIterator
@c_type "list<Item>::iterator"
```

```
list<Item> Task_TwoByFourDispenser_var;
int16_t Task_dispSize_var;
bool Task_TwoByTwoListEmpty_var;
Item* Task_work_frame_var;
list<Item> Task_TwoByTwoDispenser_var;
Item* Task_tool_var;
Item* Task_FRAME_var;
Item* Task_ROBOT_var;
Item* Task_UR10_var;
list<DuploBrick>::iterator Task_it_var;
Item* Task_toolOpen_var;
Item* Task_toolClosed_var;
DuploBrick* Task_brick_var;
Item* Task_TOOL_var;
Item* Task_table_frame_var;
list<DuploBrick> Task_BrickList_var;
RoboDK* Task_RDK_var;
Mat Task_pose_ref_var;
Item* Task_dispenser_frame_var;
```

Figure 4.26: Left: Custom objects created in ThingML that does not exist in ThingML but rather in the lower level, in this case, C++. Right: Here we can see the code generation with generated variables that uses the custom objects we created in ThingML (Indicated by the red dots next to the variables).



```
function declareRoboDKVariables() do
`cout << "Declaring RoboDK Variables. In State: Init @ RoboDKManager" << endl;`

`RDK_globalVar = new RoboDK;`
`ROBOT_globalVar = new Item(RDK_globalVar->getItem("UR10", RoboDK::ITEM_TYPE_ROBOT));`
`table_frame_globalVar = new Item(RDK_globalVar->getItem("Table Base", RoboDK::ITEM_TYPE_FRAME));`
`tool_globalVar = new Item(RDK_globalVar->getItem("Gripper RobotiQ 85", RoboDK::ITEM_TYPE_TOOL));`
`work_frame_globalVar = new Item(RDK_globalVar->getItem("Work Frame", RoboDK::ITEM_TYPE_FRAME));`

`Mat pose_ref = ROBOT_globalVar->Pose();`
`ROBOT_globalVar->setSpeed(100);`

`work_frame_globalVar->setParent(*table_frame_globalVar);`
`work_frame_globalVar->setPose(transl(0, 0, 0));`
`ROBOT_globalVar->setPoseAbs(work_frame_globalVar);`
`ROBOT_globalVar->setGeometryPose(tool_globalVar->Pose());`
end
```

Figure 4.27: This is a function within ThingML (modelling approach, first solution) that declares all the necessary variables for RoboDK, as this not code handled by ThingML kickdown is used. (Blue text)

data to each other through messages. Unfortunately ThingML's message functions could not be fully utilized for this purpose. When first creating the RoboDKManager, we wanted to declare the RoboDK variables by using the custom object datatypes from figure 4.26 and send these through messages as parameters. However, in attempting to do so we get the following error: "ERROR: Attempting to deserialize a pointer (for type RoboDK).

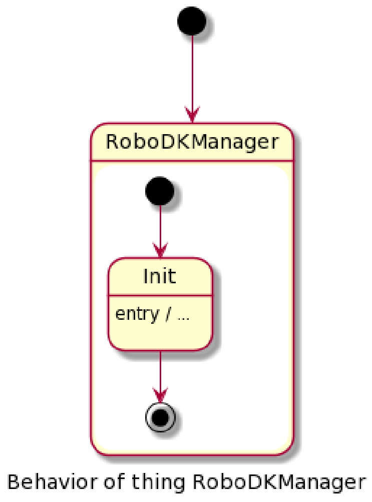


Figure 4.28: Figure shows the RoboDKManager state machine.

This is not allowed.", wether this is only the case for the arduino compiler or not is not tested. The compiler do allow for using Integers and Booleans as parameters but have no use for our purposes. Therefore, it was decided to use global variables in the lower level but declaring them via kickdown in ThingML. What was done instead is once the the RoboDK variables are declared, a simple message is sent to the Thing Task which will use the global variables declared in RoboDKManager.

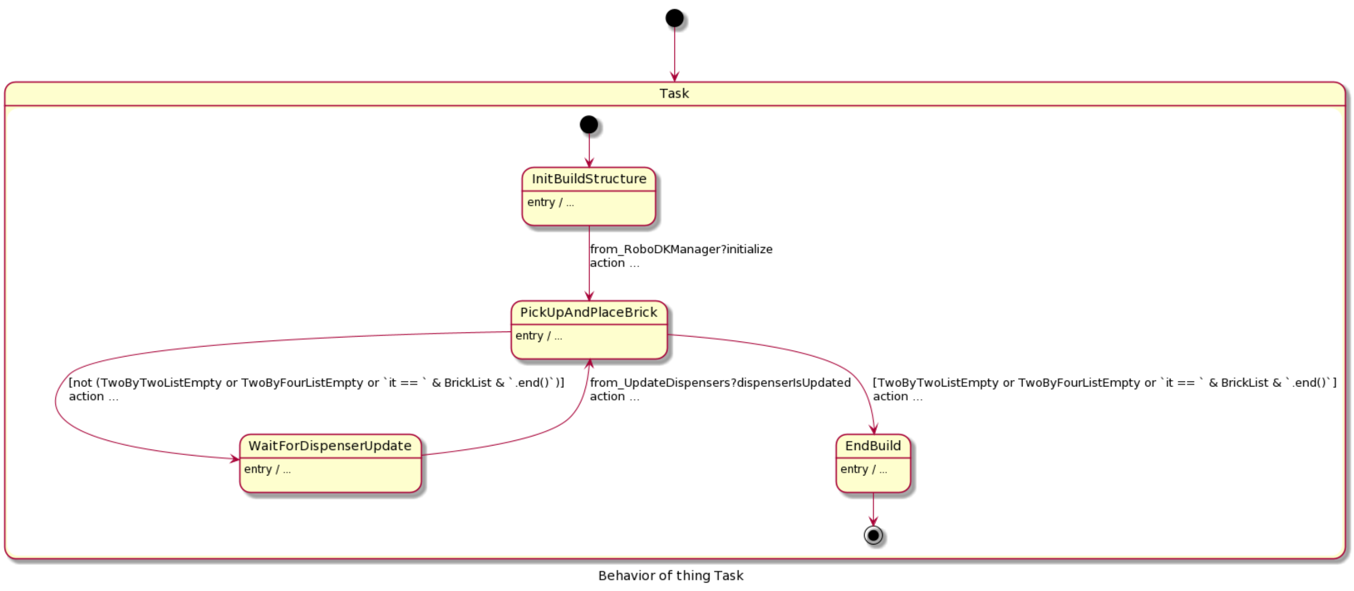

Task
InitBuildStructure
entry / ...
from_RoboDKManager?initialize
action ...
PickUpAndPlaceBrick
entry / ...
[not (TwoByTwoListEmpty or TwoByFourListEmpty or `it == ` & BrickList & `.end()`)]
action ...
from_UpdateDispensers?dispenserIsUpdated
action ...
[TwoByTwoListEmpty or TwoByFourListEmpty or `it == ` & BrickList & `.end()`]
action ...
WaitForDispenserUpdate
entry / ...
EndBuild
entry / ...
Behavior of thing Task

Figure 4.29: Figure shows the Task state machine

Figure 4.29 shows the statemachine in Thing Task and its purpose is to take care of the building process. At the start the statemachine will enter the state called Init-BuildStructure. This state will transition over to the state called PickUpAndPlaceBrick once it receives a "initialize" message from RoboDKManager. Looking at figure 4.30 we

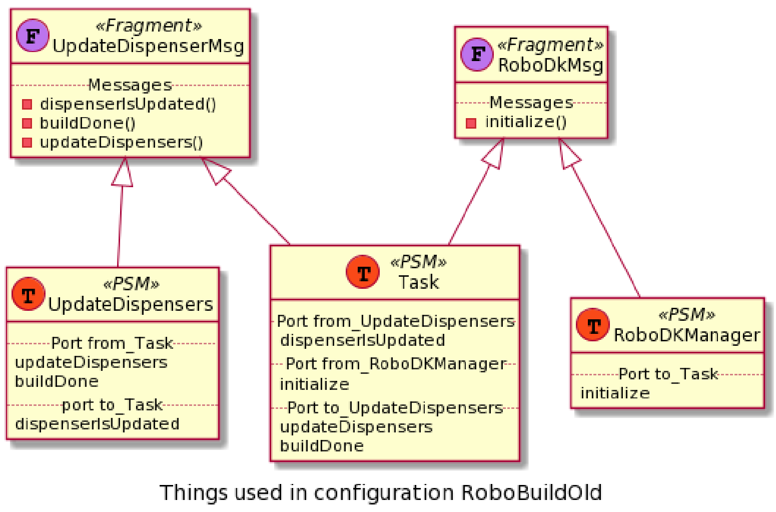


Figure 4.30: Figure shows the Things and Thing Fragments and how they are connected to each other in scenario 1 modelling approach second solution

can see how each Thing is connected to each other and which Thing has access to which Messages. Once in the state PickUpAndPlaceBrick it will first check if the dispensers are empty. Then we have two possible transitions: 1. If either of the dispensers are empty or the list of bricks is empty then it will transition over to the final state called Endbuild which stops the robot and ending the program; 2. If none of the dispensers and the list bricks are empty then it will transition over to the state called WaitForDispenserUpdate;

When PickUpAndPlaceBrick transitions over to WaitForDispenserUpdate the same procedure will happen as in the direct approach except that no loops are used to iterate through the list of bricks. Instead the next time PickUpAndPlaceBrick transitions over to WaitForDispenserUpdate the program will already be looking at the next brick in the list. Aside from this, the procedure starts by declaring a new duplobrick and assign the same information as the brick that the program is currently looking at in the list. Then a function called buildStructure which is similar to the pickUpBrick_and_Move function in the direct approach is called. Function buildStructure utilizies the exact same RoboDK API functions as pickUpBrick_and_Move with the help of kickdown and the same robotmovements are initialized. There is one difference however which is that threads is not used to initialize the updating of dispensers to run in parallel. Instead, once the robot has picked out a brick from one of the dispensers a message is sent to the Thing UpdateDispensers, see figure 4.31 for what the statemachine in this Thing looks like.

The message sent is the updateDispensers message as seen in the UpdateDispenserMsg Fragment in figure 4.30. At program start inside UpdateDispensers it will enter the state called Wait which, as the name suggests, wait untill it receives the updateDispensers message. Once UpdateDispensers receives this message, the Wait state will transition over to the Updated state and while it transitions the current dispenser is updated. The procedure for updating either of the dispensers is exactly the same as in the direct approach and uses the same functions with the help of kickdown. In theory the updating of the dispensers should run as a parallel process while the robot finishes its movements. Once the updating of the dispensers are done and the statemachine has transitioned from Wait to Updated, it will wait for the buildDone message from the Thing Task. Once the robot has

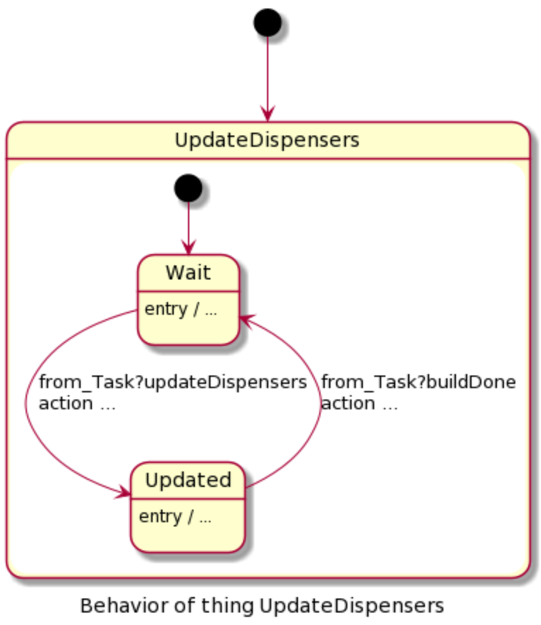


Figure 4.31: Figure shows the UpdateDispensers state machine.

finished placing a brick it, it will send a buildDone message to UpdateDispensers and the statemachine in Thing Task will transition to the state called WaitForDispenserUpdate. While inside WaitForDispenserUpdate it will wait for the dispenserIsUpdated message from UpdateDispensers.

So what happens next is that UpdateDispensers receives the buildDone message and the statemachine will transition from Updated to Wait and in doing so it will send a dispenserIsUpdated message back to Task where its statemachine will transition from WaitForDispenserUpdate to PickUpAndPlaceBrick. Figure 4.32 shows the configuration for how the three Things communicate with each other.

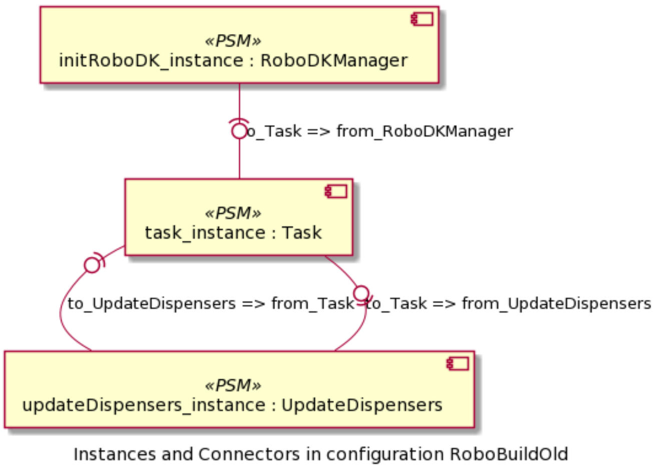
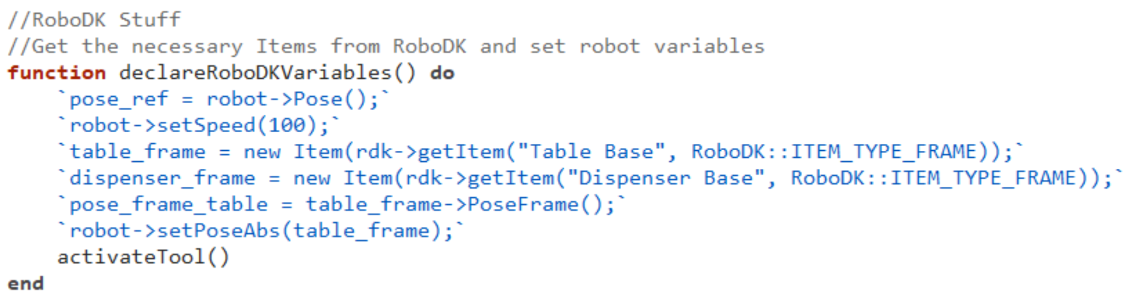

«PSM»
initRoboDK_instance : RoboDKManager
to_Task => from_RoboDKManager
«PSM»
task_instance : Task
to_UpdateDispensers => from_Task
to_Task => from_UpdateDispensers
«PSM»
updateDispensers_instance : UpdateDispensers
Instances and Connectors in configuration RoboBuildOld

Figure 4.32: Figure shows the configuration and how the instances of each Thing communicates with each other in scenario 1 modelling approach second solution

4.4.6 The modeling approach. Third and final solution

After creating the second solution, a few noticeable flaws were noticed. While the second solution did work as intended, cleaning up those flaws would make the solution overall more clean and end up with less lines of code. Firstly, the Thing RoboDKManager serves no real purpose other than just declaring RoboDK variables. This can just be done when instanciating the Thing Task instead. The reason behind including RoboDKManager in the first place was to utilize messages and trying to separate kickdown from actual ThingML code as much as possible.

The goal with creating this third and final solution is to do a thorough check of the second solution, remove unnecessary parts and reduce kickdown code down to a minimum. First off, we completely remove RoboDKManager where we also remove the RoboDKMsg fragment since it has no use anymore. What was initially declared in RoboDKManager is now done in Task when entering the firste state: InitBuildStructure. See figure 4.33 and figure 4.34 for the before and after.



```
//RoboDK Stuff
//Get the necessary Items from RoboDK and set robot variables
function declareRoboDKVariables() do
    `pose_ref = robot->Pose();`
    `robot->setSpeed(100);`
    `table_frame = new Item(rdk->getItem("Table Base", RoboDK::ITEM_TYPE_FRAME));`
    `dispenser_frame = new Item(rdk->getItem("Dispenser Base", RoboDK::ITEM_TYPE_FRAME));`
    `pose_frame_table = table_frame->PoseFrame();`
    `robot->setPoseAbs(table_frame);`
    activateTool()
end
```

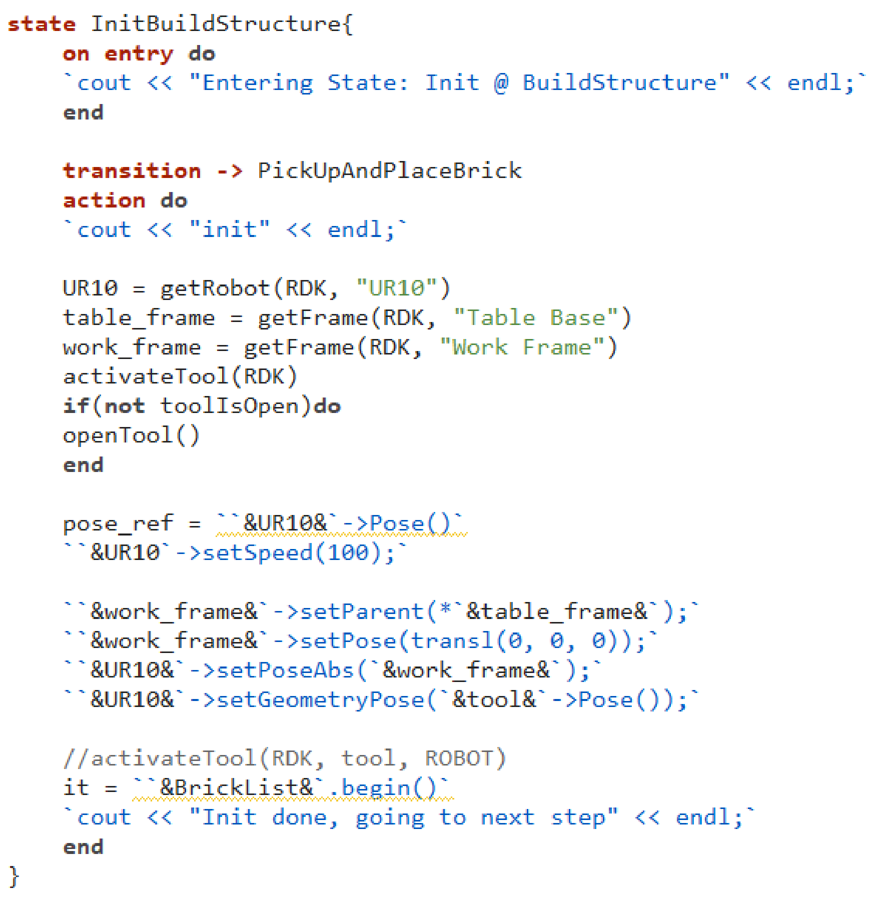
Figure 4.33: Declaring global RoboDK variables using only Kickdown

Notice how the amount of kickdown is reduced by utilizing the object datatypes which also gives us a better overview because we are using variables defined on the ThingML level rather than variables defined on the lower level behind kickdown. On thing that is not apparent when comparing figure 4.33 and 4.34 is that some kickdown code is now also moved to their own functions, see figure 4.35, which was the next step in trying to clean up as much as possible.

To summarize this third and final solution: We removed the RoboDKManager and RoboDK variables is not instead instanciated in the Thing Task. This allowed for using the custom objects created in figure 4.26 which combined with placing as much kickdown code in functions as possible cleaned up a lot of the ThingML code. We believe this third and final solution is at this point as optimized as it can be. Figure 4.36 and 4.37 shows the final version of Things and Fragments, and how they are configured to communicate with each other in the modelling approach in scenario 1.

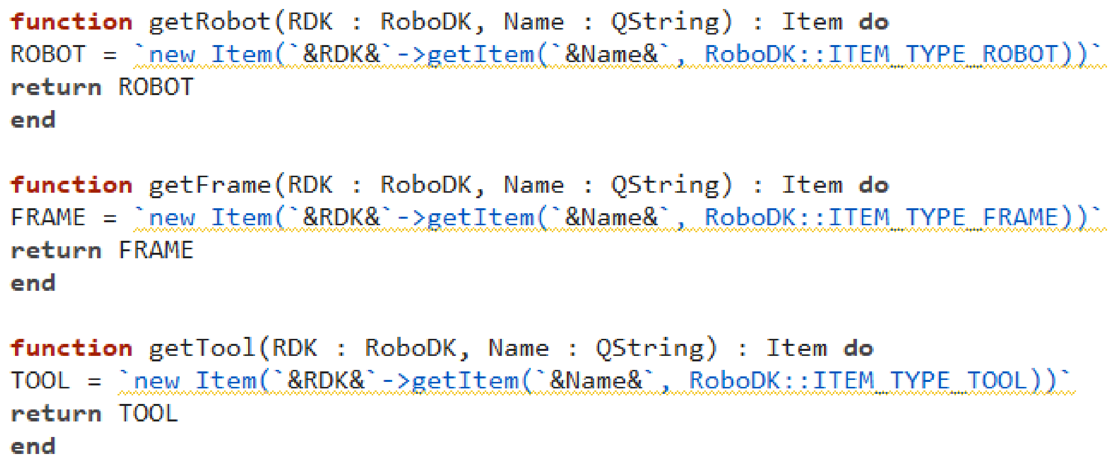
4.5 Scenario 2

To summarize this scenario shortly, the manufacturer wants to upgrade the system to handle 3D models of the user's choice. These 3D models can be anything the user desire as long as it is within the range of the robot's workspace. The manufacturer also wants to install an emergency button to the system, when pushed the system will shut down and the robot will come to a complete stop. Since the robots work with coordinates in a 3D plane,



```
state InitBuildStructure{
    on entry do
    `cout << "Entering State: Init @ BuildStructure" << endl;`
    end

    transition -> PickUpAndPlaceBrick
    action do
    `cout << "init" << endl;`

    UR10 = getRobot(RDK, "UR10")
    table_frame = getFrame(RDK, "Table Base")
    work_frame = getFrame(RDK, "Work Frame")
    activateTool(RDK)
    if(not toolIsOpen)do
    openTool()
    end

    pose_ref = ``&UR10&`->Pose()`
    ``&UR10`->setSpeed(100);`

    ``&work_frame&`->setParent(*`&table_frame&`);`
    ``&work_frame&`->setPose(transl(0, 0, 0));`
    ``&UR10&`->setPoseAbs(`&work_frame&`);`
    ``&UR10&`->setGeometryPose(`&tool&`->Pose());`

    //activateTool(RDK, tool, ROBOT)
    it = ``&BrickList&`.begin()`
    `cout << "Init done, going to next step" << endl;`
    end
}
```

Figure 4.34: Kickdown is reduced by declaring the RoboDK variables in Task instead and using the custom objects that we created in figure 4.26. A lot of the kickdown code from the left picture is also now moved into their own functions.



```
function getRobot(RDK : RoboDK, Name : QString) : Item do
ROBOT = `new Item(`&RDK&`->getItem(`&Name&`, RoboDK::ITEM_TYPE_ROBOT))`
return ROBOT
end

function getFrame(RDK : RoboDK, Name : QString) : Item do
FRAME = `new Item(`&RDK&`->getItem(`&Name&`, RoboDK::ITEM_TYPE_FRAME))`
return FRAME
end

function getTool(RDK : RoboDK, Name : QString) : Item do
TOOL = `new Item(`&RDK&`->getItem(`&Name&`, RoboDK::ITEM_TYPE_TOOL))`
return TOOL
end
```

Figure 4.35: Declaring global RoboDK variables using only Kickdown

the 3D models needs to be transformed into a pixelated version of themselves. This is done through a voxel transformation. To make the process simple, the voxel transformation is

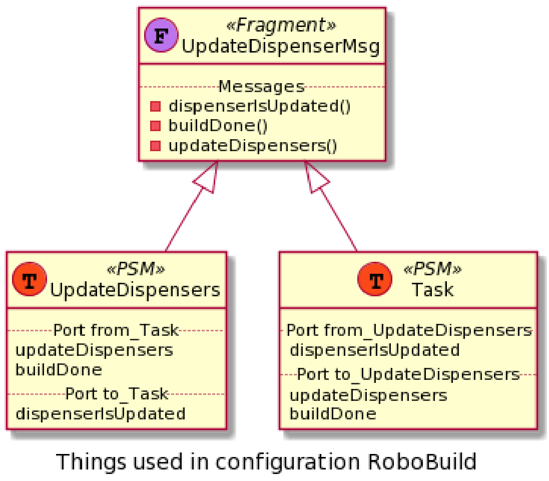


Figure 4.36: Figure shows the Things and Thing Fragments and how they are connected to each other for the final solution to Scenario 1.

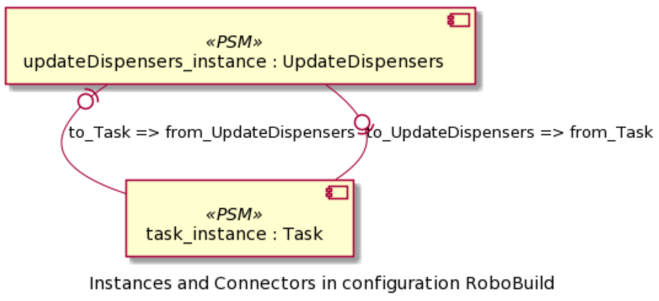

«PSM»
updateDispensers_instance : UpdateDispensers
to_Task => from_UpdateDispensers
to_UpdateDispensers => from_Task
«PSM»
task_instance : Task
Instances and Connectors in configuration RoboBuild

Figure 4.37: Figure shows the configuration and how the instances of each Thing communicates with each other for the final solution to Scenario 1.

done by a online voxelizer through the website drububu.com [5]. This voxelizer allows the user to choose the size, density and what type of file the transformed model will be saved as.

For the purposes of this work, the transformed models will be saved as txt files. Each line in the txt file defines each voxel as such: "coordinate 1, coordinate 2, coordinate 3", with the middle value representing the layers. The algorithm picks out these coordinates and inserts them into an array through a filestream in C++, listing 4.7 shows an overview of this filestream. Each line is seperated by the comma and the values in between are converted into integers as xCoord, yCoord and zCoord which are then used to set a value "true" in the 3D array.

Example, the first line in the txt file is: "2, 0, 3". This tells us that there is a voxel (or in our case, a brick) at the first layer (z = 0) on position x = 2 and y = 3. The filestream seperates the characters by the comma and converts the values into integers, xCoord is then assigned the value "2", zCoord is assigned the value "0" because it represents the layers, and yCoord is assigned the value "3". Then we use these variables to assign a true value in the 3D array like this: coord[zCoord][xCoord][yCoord] = true. So in short we read the line in the txt file: "2, 0, 3", and assign a true value in the 3D array: coord[0][2][3] = true, repeat until end of txt file is reached.

Listing 4.7: A simple filestream in C++ for retrieveing the coordinates inside txt files from the voxel transformation of the 3D models.

```
string readX, readY, readZ;
int xCoord, yCoord, zCoord;

ifstream fileStream("filelocation");
        if(!fileStream.is_open()){
            cout<<"Error when opening file"<<endl;
        }
        string line;

        while(getline(fileStream, line)){
            stringstream ss(line);
            getline(ss, readX, ',');
            xCoord = stoi(readX);

            getline(ss, readY, ',');
            zCoord = stoi(readY);

            getline(ss, readZ, ',');
            yCoord = stoi(readZ);

            coord[zCoord][xCoord][yCoord] = true;
        }
        fileStream.close();
```

For testing purposes we will use three different 3D models: a boat, a beach ball and bowling pins. figure 4.38, 4.39, and 4.40 shows the 3D models of these and a voxelized version as example. Note that the textures are not included as the systems for the time being are not designed to build with colored bricks and the testing is overall more focused on how well the robot is at building different shapes. Though the online voxelizer do support textures as well so a possible upgrade in the future could definitely be for the robots to build with colored bricks as well.

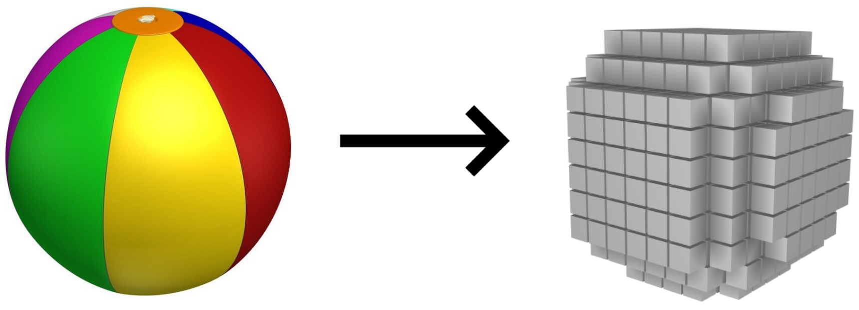


Figure 4.38: Left: 3D model of a beach ball. Right: The beach ball transformed into a voxelized version of itself

4.5.1 The direct approach

Because the program is running on a single thread, the GUI, which we will be using to simulate the emergency button, is locked and non-interactive until the code that is

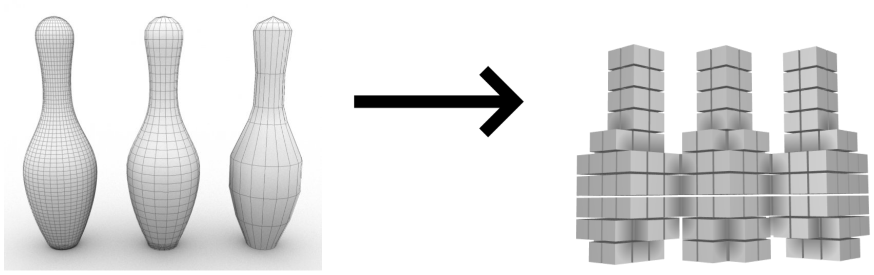


Figure 4.39: Left: 3D model of three bowling pins. Right: The bowling pins transformed into a voxelized version of themselves

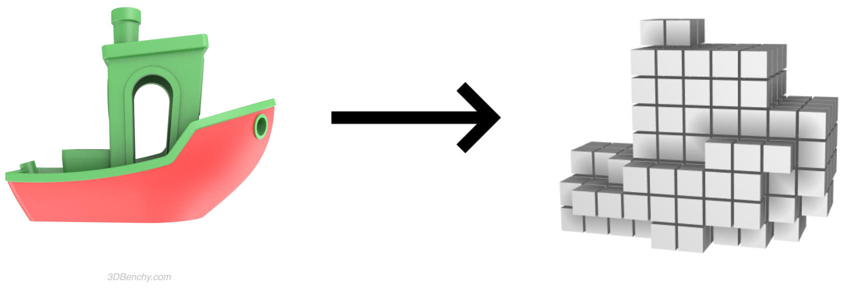


Figure 4.40: Left: 3D model of a boat. Right: The boat model transformed into a voxelized version of itself

currently running is finished. To handle this we use threads and give each function their own thread. For this we can either use std::thread which is C++'s specific class for running single threads or we can use a handful of QT specific classes if desired. Though it is recommended by RoboDK's C++ documentation to use std::thread and declare new instances of the RoboDK variable if we want to run systems in parallel in the simulation, which will be done in this thesis.

So far for the direct approach we have in total 3 functions, the algorithm which is ran at the program start, the function for moving the robot and placing bricks at the correct spots, and the function which simulates and updates the dispenser. In this scenario we are adding one additional function, a function where if called, is to simulate someone pushing down the emergency stop button. Firstly we now need 2 threads, a thread for the moving and building part and a thread for the updating of dispensers. When the emergency stop button is pushed down, the main thread handling the building processes is forcefully ended, a function for stopping the robot is called, and a global boolean value which the function in the main thread checks before getting the next brick is set to true as shown in listing 4.8. The purpose for which this boolean value is checked is to see if the emergency stop button is pushed down or not.

Listing 4.8: The listing shows what the function called when the emergency stop button is pushed looks like

```
void on_emergencyStopbtn_clicked(){
    ROBOT->Stop();
    mainThread->detach();
    EmergencyStopTriggered = true;
}
```

4.5.2 The modeling approach

To start off, a new Thing called RobotIdleStop is created see figure 4.41 to get an overview of the statemachine in this Thing.

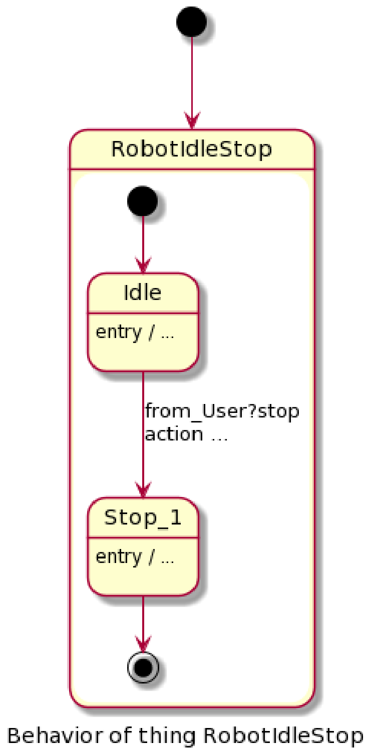


Figure 4.41: Figure shows the RobotIdleStop statemachine created in scenario 2, its purpose is to listen for when the emergency stop button is pushed.

The purpose of RobotIdleStop is to wait and listen for the emergency stop button to be pushed. Once the emergency button is pushed then RobotIdleStop will send a message to the Thing handling the building process: Task, and to the Thing handling the updating of dispensers: UpdateDispensers. Once Task and UpdateDispensers receives this stop message, the purpose is for them to stop all processes as soon as possible. This is done by putting said processes into a composite state.

Looking at figure 4.42 showing the Task statemachine and figure 4.43 showing the UpdateDispensers statemachine we can see that all the previous states which handles the building or updating of dispenser processes is now put into a composite state called Running. Once Task and UpdateDispensers receives a stop message, their Running state will transition over to a state called Stop which stops the robot and the program. The reason the UpdateDispensers is also added a composite state is because every Thing is instanciated at start. When Running in Task transitions to Stop and the program stops. Without the added composite state in the UpdateDispensers it's instance would still be

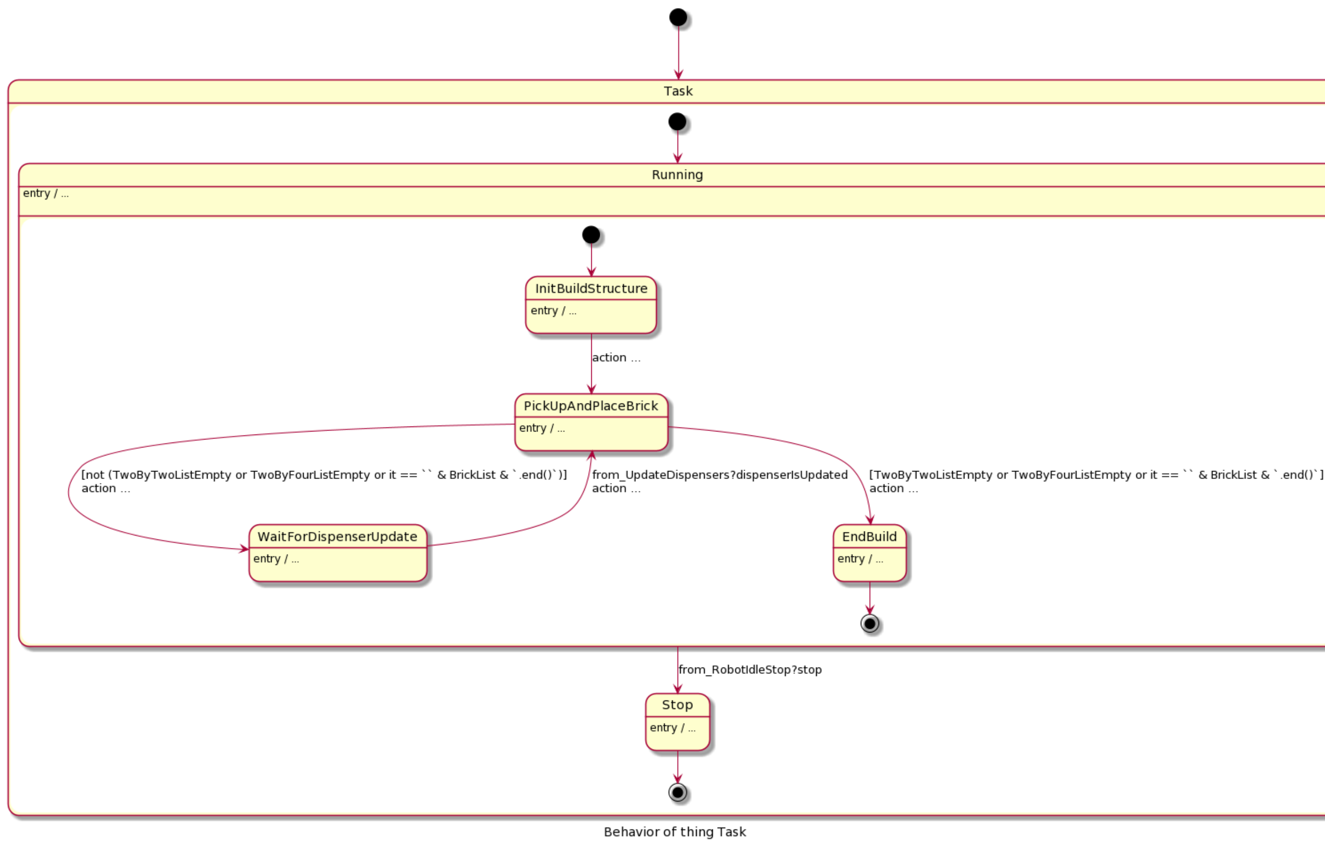


Figure 4.42: Figure shows the Thing Task used in Scenario 2. It has been edited to now have a composite state called Running. The states and functions from scenario 1 is now moved inside this composite state.

active. When Things are instantiated again while they have an active instance the program will crash.

Optimally RobotIdleStop will listen for a button push from the GUI, just like in the direct approach when Emergency Stop is pushed in the GUI the Robot will stop. Unfortunately, as of the time of this implementation, connecting a button from the GUI in QT to a Thing in ThingML was not possible. Instead a new Thing called User is implemented, the purpose of User is to simulate someone pushing a emergency stop button. User utilizes a timer and once it runs out a stop-message will be sent to RobotIdleStop which in turn sends a stop-message to Task and UpdateDispensers. If the program finishes before the timer in User runs out it will receive a stopTimer-message from Task. This is mainly due to avoid the program crashing.

For the stop- and stopTimer-messages a new fragment called StopMsg was implemented, see figure 4.44 for a overview of the fragment StopMsg and the Things that utilizes it and see figure 4.45 for a overview of how the Things are configured communicate with each other.

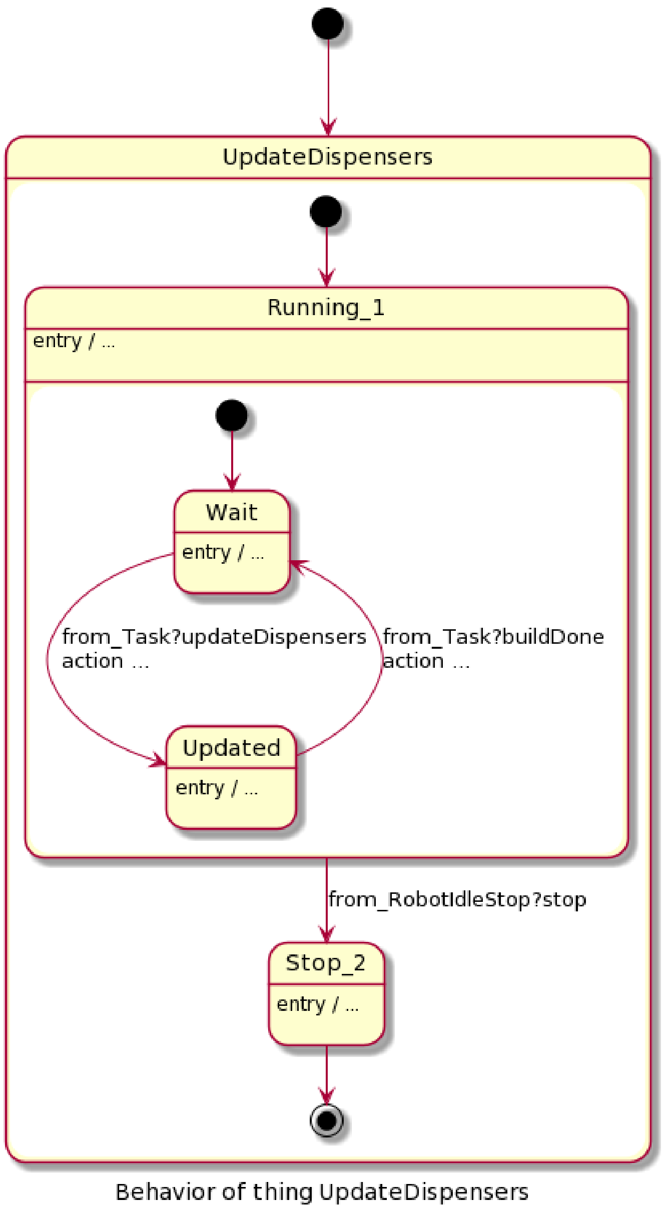


Figure 4.43: Figure shows the Thing UpdateDispensers used in Scenario 2. It has been edited to now have a composite state called Running. The states and functions from scenario 1 is now moved inside this composite state.

4.6 Scenario 3

To summarize scenario 3 shortly, in this scenario, the manufacturer now wants to add functionality to the systems which adds support bricks to structures where bricks with no support from previous layers occurs, otherwise known as floating bricks. The manufacturer also wants to add a safety switch which triggers when something from the outside enters the robots work space, such as a human arm. For the safety switch trigger, the implementation in this scenario is similar to the kuka robot setup at Østfold University College. The set up uses light curtains as a safety measure in case someone or something enters the robots

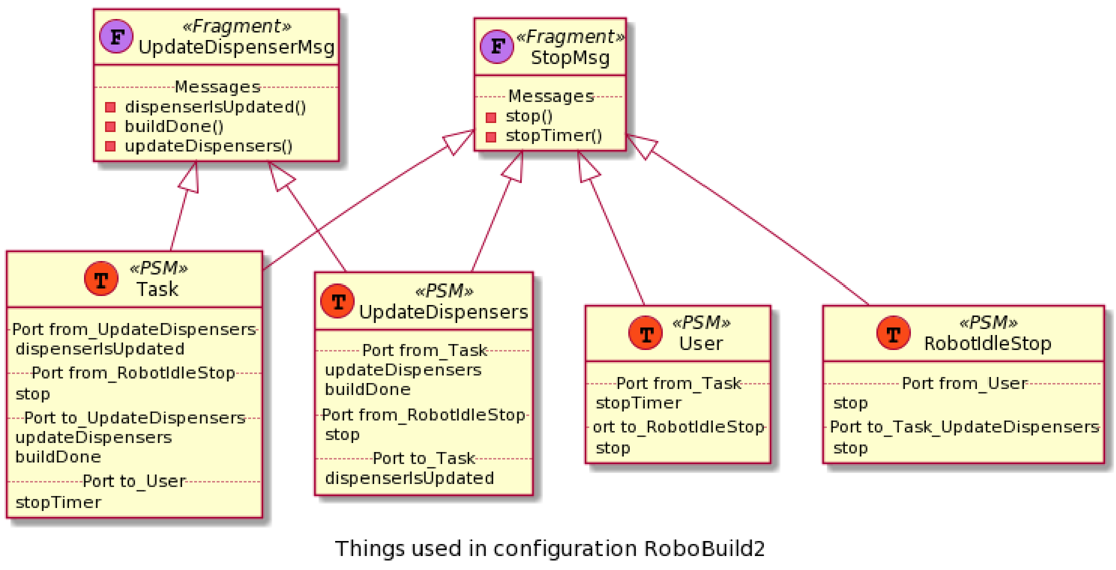


Figure 4.44: Figure shows the Things and Thing Fragments and how they are connected to each other for scenario 2

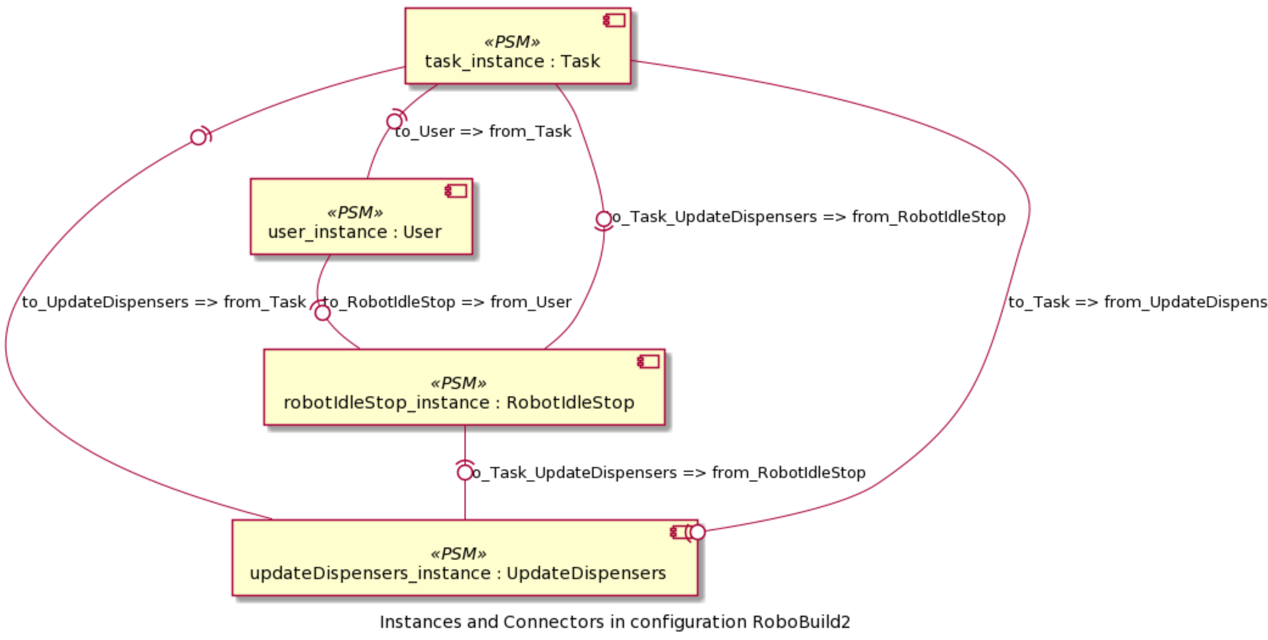

«PSM»
task_instance : Task
to_User => from_Task
«PSM»
user_instance : User
o_Task_UpdateDispensers => from_RobotIdleStop
to_UpdateDispensers => from_Task
to_RobotIdleStop => from_User
to_Task => from_UpdateDispens
«PSM»
robotIdleStop_instance : RobotIdleStop
o_Task_UpdateDispensers => from_RobotIdleStop
«PSM»
updateDispensers_instance : UpdateDispensers
Instances and Connectors in configuration RoboBuild2

Figure 4.45: Figure shows the configuration of how Things communicate with each other in scenario 2

workspace while it is active which fits the problem-statement. Therefore, the goal of both the approaches in this scenario is to recreate light curtains feature in the simulation.

4.6.1 The direct approach

First off, the algorithm is upgraded to include support bricks to avoid floating bricks. If the current layer is 1 or more (first layer start at 0) then for each brick we check all the previous layers at the same position and assign a new brick to be placed on each layer where a brick at the given position is missing. These support bricks are added in a separate list, for context called supportBrick list. This supportBrick list is merged into the main list containing the building bricks. Then the main list is sorted on the layer number so all the bricks that belongs to the current layer is placed before the robot starts placing bricks in the next layer, this includes of course also the support bricks. In order to distinguish the support bricks from the building bricks they are set to be the color purple. See listing **??** for a overview of this added algorithm. One thing worth mentioning is that the algorithm only places 2x2 bricks as support bricks even if the current brick is a 4x2. Depending on the placement and the rotation of a 4x2 brick only half of its studs may be supported but this should not cause a problem. If any 4x2 brick calls for support bricks in the previous layer then it should be enough to only use 2x2 bricks. Figure 4.46 provides two examples where one structure is built without support bricks and the same structure built with support bricks. Lastly, the support brick algorithm uses a separate 3D array which is a exact copy of the 3D array used by the main algorithm. The reason for this is that the main algorithm sets a false value on the position it finds a true value on after assigning a brick to that position. This is to avoid overlaps of the 4x2 bricks. For this reason the support algorithm cannot use the same 3D array because then it will place support bricks on positions where there already exists a brick.

Listing 4.9: The algorithm is upgraded to now also check the previous layers and assign bricks to be placed at the current position for each layer missing a brick. This is done to avoid floating bricks

```
if(z > 0){
    int childLayerCounter = 1;
    while(childLayerCounter <= z + 1){
        if(!dCoord[childLayerCounter - 1][x][y]){
            DuploBrick *supportBlock = new DuploBrick();

            supportBlock = TwoByTwo;
            dCoord[childLayerCounter - 1][x][y] = true;
            supportBlock->zCoord = childLayerCounter - 1;
            supportBlock->priority = 3;

            supportbricklist.push_back(*supportBlock);
        }
        childLayerCounter++;
    }
}
```

To implement the light curtain solution for this approach a new button was added to the GUI and a new function was added to the main program. The purpose of this new button and function is to simulate a person sticking their arm into the robots work-space triggering the light curtains and thus stopping the program.

When this new button is clicked the program will start a new thread, inside the thread, a function called EmergencyTriggered is ran. In order to simulate a person sticking their arm into the robot's workspace. Right before the robot is stopped due to the light curtains sensor being triggered, a human arm 3D model is generated in the simulation. This is done using the RoboDK API function AddFile, see table 4.2. What follows next is similar procedure to the emergency stop button in scenario 2. The Robot is stopped and the main thread is forcefully ended. In case of the thread failing to stop, a global boolean value is

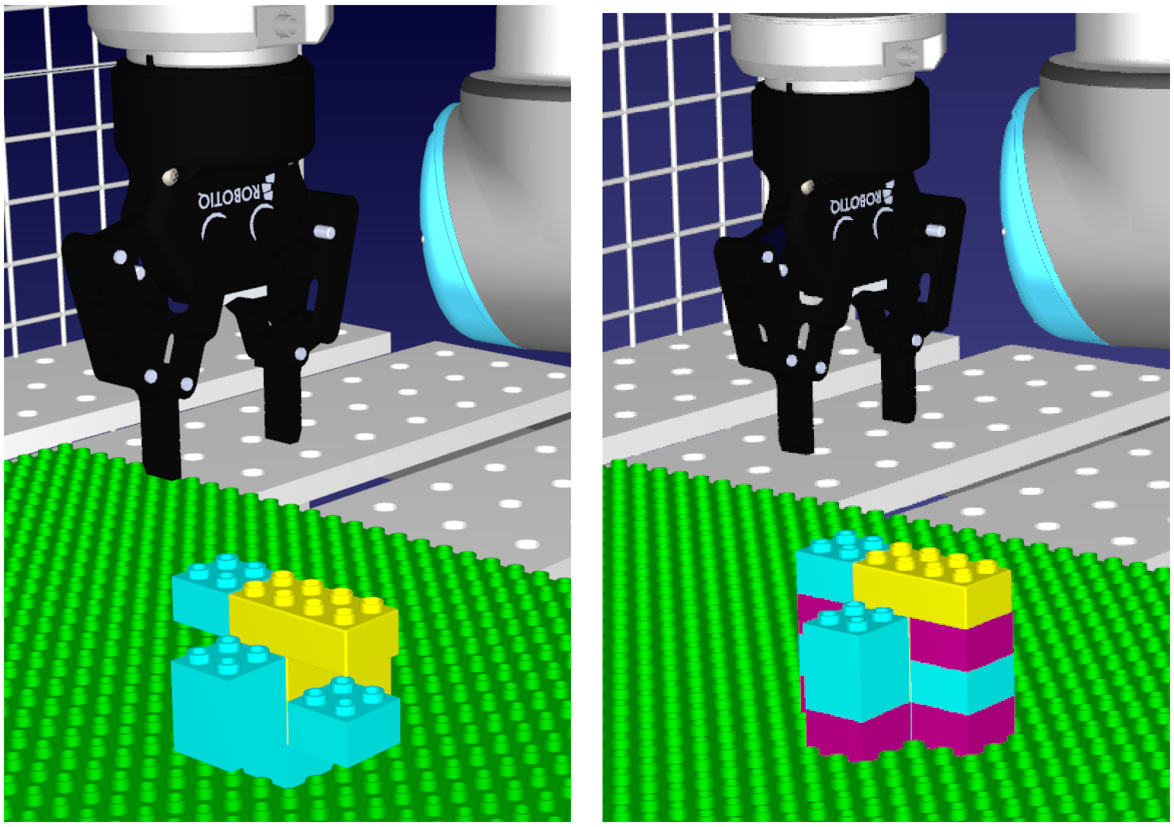


Figure 4.46: "Left: A structure without using support bricks. Right: The same structure but with support bricks."

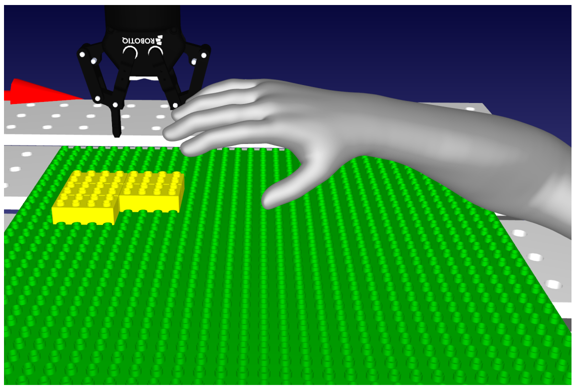
set to true which the main program checks before getting the next brick as a safety feature see listing 4.10 for a overview of this code. See figure 4.47 to see how this looks in the simulation.

Listing 4.10: The code of the function that is ran when simulating the sensors of light curtains spotting an arm entering the robot's work-space

```
RoboDK *rdk = new RoboDK;
Item *frame = new Item(rdk->getItem("Person", RoboDK::ITEM_TYPE_FRAME));
rdk->AddFile("C:/RoboDK/Library/11535_arm_V3_.obj", frame);
Item *arm = new Item(rdk->getItem("11535_arm_V3_", RoboDK::ITEM_TYPE_OBJECT));
arm->Scale(8.00000);
ROBOT->Stop();
mainThread->detach();
EmergencyStopTriggered = true;
```

4.6.2 The modeling approach

In the modelling approach, a new Thing was added called LightCurtain, see figure 4.48 for how the statemachine in this Thing looks like. LightCurtain has two ports: a provided port for receiving lightCurtains_Triggered messages which mimics that the sensors are triggered by something entering the robot's workspace and a required port for sending a stop message to the Thing RobotIdleStop. Since were not able to connect buttons from



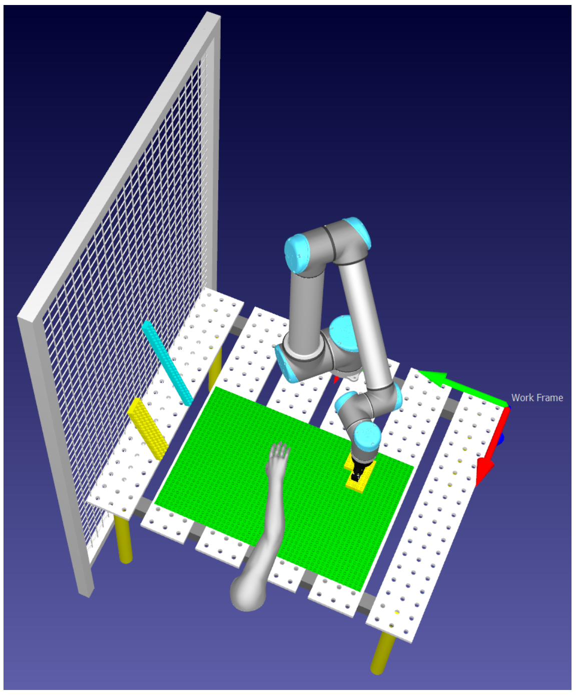


Figure 4.47: "Figure shows how it is simulated when an arm from a person enters the robot's work-space, triggering the light curtains sensors."

the GUI to Things in ThingML the Thing User was altered here and used in the same way it was used in the modelling approach in scenario 2. At program start, a timer will start in the Thing User. Once this timer runs out, a lightCurtains_Triggered message is sent to LightCurtains. Upon receiving this message, the statemachine transitions from the Idle state to the final state called CurtainsTriggered. When transitioning, the same procedure for adding a 3D model of a human arm in the direct approach is done here as well using the same RoboDK API specific functions with the help of kickdown. Upon entering the CurtainsTriggered state, a stop message is sent from LightCurtain to RobotIdleStop which in turn sends a stop message Task and UpdateDispensers which in turn stops the robot and the program. For a full overview of how the Things and Fragments are connected to each other via ports see figure 4.49 and see figure 4.50 for how the Things are configured to communicate with each other.

4.7 Scenario 4

To summarize shortly, in this scenario, the manufacturer wanted one last improvement to the existing system. Here, instead of one coordinate equals one brick placement, one coordinate now equals the position of a stud. In addition, functionality is to be added that handles situations where the gripper looses a brick, the thought-process here is that the tool is added a sensor which lets the program know when the gripper have lost a brick.

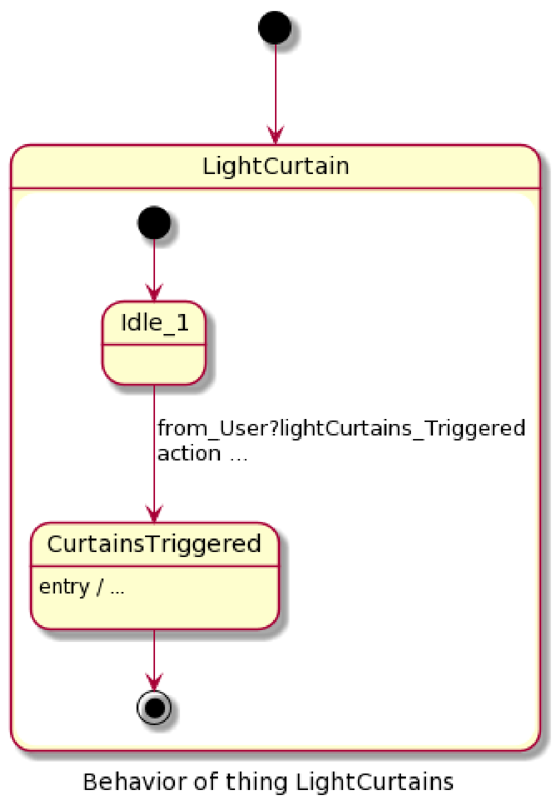

LightCurtain
Idle_1
from_User?lightCurtains_Triggered
action ...
CurtainsTriggered
entry / ...
Behavior of thing LightCurtains

Figure 4.48: Figure shows the LightCurtains statemachine created in scenario 3, its purpose is to listen for when the sensors of the Light Curtains is triggered from something entering the robot's work-space.

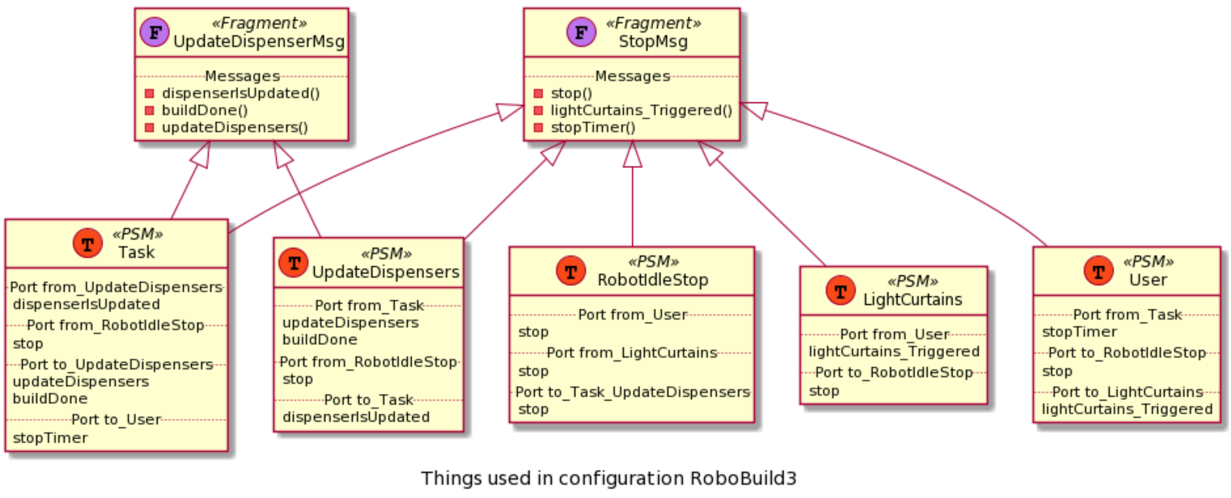


Figure 4.49: Figure shows the Things and Thing Fragments and how they are connected to each other for scenario 3

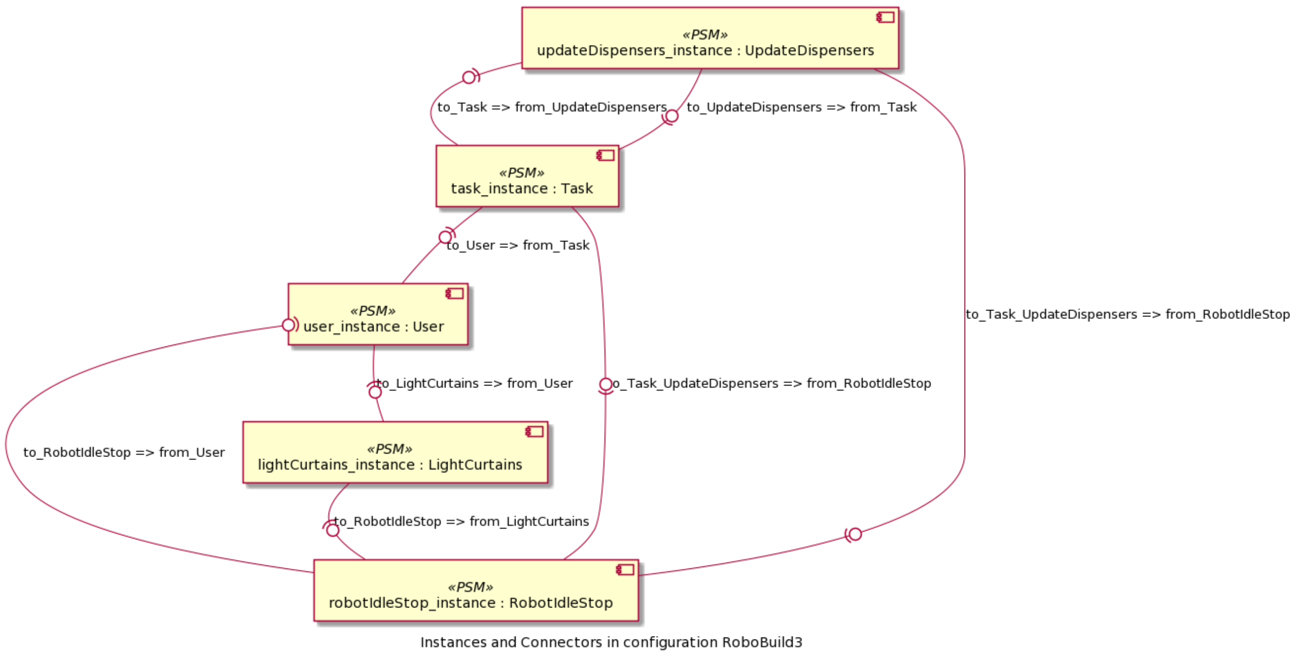

«PSM»
updateDispensers_instance : UpdateDispensers
to_Task => from_UpdateDispensers
to_UpdateDispensers => from_Task
«PSM»
task_instance : Task
to_User => from_Task
to_Task_UpdateDispensers => from_RobotIdleStop
«PSM»
user_instance : User
to_LightCurtains => from_User
to_Task_UpdateDispensers => from_RobotIdleStop
to_RobotIdleStop => from_User
«PSM»
lightCurtains_instance : LightCurtains
to_RobotIdleStop => from_LightCurtains
«PSM»
robotIdleStop_instance : RobotIdleStop
Instances and Connectors in configuration RoboBuild3

Figure 4.50: Figure shows the configuration of how Things communicate with each other in scenario 3

4.7.1 The direct approach

Due to limited time, this upgraded algorithm does not include sub-figures or rotation of bricks. As mentioned in chapter 3.5 the method for adding support bricks is prioritized, in addition, the limited works-pace for the robot could potentially make the method for splitting structures into sub-figures complicated if dealing with very complex structures. To start off, the new algorithm uses three nested for loops here as well. Once it finds a true value in the 3D array it will start counting the neighbours. It will always count from left to right, from top to bottom. It will also always check if a 4x2 brick is placeable before checking if a 2x2 brick is placeable. Both these bricks have their own values in regards to how many studs they have in each direction. These values are called studsXDir and studsYDir. The algorithm uses these values to count the neighbours of the first true value that is found. For example: say the algorithm found a true value at $x = 0$ $y = 0$, when it then checks if it can place a 4x2 brick it will check the positions on $x = 1$ $y = 0$, $x = 2$ $y = 0$, $x = 3$ $y = 0$, $x = 1$ $y = 1$, $x = 2$ $y = 1$, and $x = 3$ $y = 1$. If the counted studs is eight, then a 4x2 brick can be placed at $x = 0$ $y = 0$. If not, it will check if 2x2 brick is placeable at $x = 0$ $y = 0$ and it will follow the same procedure here. In order to avoid overlapping of brick placements the position of all the studs each brick covers is set to false in the Main 3D array and is set to true in the support 3D array which is used by the support bricks algorithm.

Listing 4.11: The new algorithm where one coordinate equals one stud

```
for(int z = 0; z < 50; z++){
    for(int x = 0; x < 50; x++){
        for(int y = 0; y < 50; y++){
            if(coord[z][x][y]){
                DuploBrick *TwoByTwo = new DuploBrick();
                TwoByTwo->fileLocation = "C:/RoboDK/Library/
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣duplo_brick_2x2.stl";
                TwoByTwo->name = "duplo_brick_2x2";
                TwoByTwo->studsXDir = 2;
                TwoByTwo->studsYDir = 2;
                TwoByTwo->width = 16;
                TwoByTwo->depth = 16;
                TwoByTwo->height = 19;
                TwoByTwo->dispenserName = "Dispenser␣Base";

                DuploBrick *TwoByFour = new DuploBrick();
                TwoByFour->fileLocation = "C:/RoboDK/Library/
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣duplo_brick_2x4.stl";
                TwoByFour->name = "duplo_brick_2x4";
                TwoByFour->studsXDir = 4;
                TwoByFour->studsYDir = 2;
                TwoByFour->width = 16;
                TwoByFour->depth = 16;
                TwoByFour->height = 19;
                TwoByFour->dispenserName = "Dispenser␣Base␣2";

                int countStudsForTwoByFour = CheckIfPlaceable(
                TwoByFour->studsXDir, TwoByFour->studsYDir, coord, x, y, z);

                int countStudsForTwoByTwo = CheckIfPlaceable(
                TwoByTwo->studsXDir, TwoByTwo->studsYDir, coord, x, y, z);

                int countStudsForChildLayer;
                int maxSupport;

                if(countStudsForTwoByTwo >= countStudsForTwoByFour
                   && countStudsForTwoByTwo == TwoByTwo->getTotalStuds()){
                    if(z > 0){
                        //If needed
                        addSupportBricksThread = QtConcurrent::run(
                        this, &MainWindow::AddSupportBricks, TwoByTwo, x, y, z);
                        int result = addSupportBricksThread.resultCount();
                    }
                    TwoByTwo->setPriority(countStudsForTwoByTwo);
                    TwoByTwo->xCoord = x;
                    TwoByTwo->yCoord = y;
                    TwoByTwo->zCoord = z;
                    bricklist.push_back(*TwoByTwo);

                    if(TwoByTwo->priority == 2){
                        for(int i = 0; i < TwoByTwo->studsXDir; i++){
                            for(int j = 0; j < TwoByTwo->studsYDir; j++){
                                coord[z][x+i][y+j] = false;
                                dCoord[z][x+i][y+j] = true;
                            }
                        }
                    }


                }
                else if(countStudsForTwoByFour == TwoByFour->getTotalStuds()){
                    if(z > 0){
                        //If needed
                        addSupportBricksThread = QtConcurrent::run(
                        this, &MainWindow::AddSupportBricks, TwoByFour, x, y, z);
                        int result = addSupportBricksThread.resultCount();
                    }
                    TwoByFour->setPriority(countStudsForTwoByFour);
                    TwoByFour->xCoord = x;
                    TwoByFour->yCoord = y;
                    TwoByFour->zCoord = z;
                    bricklist.push_back(*TwoByFour);

                    if(TwoByFour->priority == 2){
                        for(int i = 0; i < TwoByFour->studsXDir; i++){
                            for(int j = 0; j < TwoByFour->studsYDir; j++){
                                coord[z][x+i][y+j] = false;
                                dCoord[z][x+i][y+j] = true;
                            }
                        }
                    }
                }
                if(addSupportBricksThread.isRunning()){
                    addSupportBricksThread.waitForFinished();
                }
}
```

When it comes to adding support bricks, the procedure is pretty similar to the algorithm in scenario 3. Here, if the layercount is one or higher it will check for true values in the support 3D array at the positions where the current brick is placed. Here the condition is that the amount of supporting studs must be 80% in the previous layer for the current brick. This means in the lower layers the amount of studs available for the current brick must be roughly 6 studs 4x2 bricks and 3 studs for 2x2 bricks. If there are no studs available, a support brick will be placed at that layer. Of course, the same rules goes for placing support bricks as well. When placing a support brick, the brick must have 80% support in the previous layer as well. See listing 4.12 for a overview of this supportbrick algorithm.

Listing 4.12: The algorithm for adding support bricks when one coord equals one stud

```
int maxSupport = nearbyint((brick->getTotalStuds()) * 0.8);
cout << "Max support: " << maxSupport << endl;
int countStudsForChildLayer = CheckIfPlaceable(brick->studsXDir,
brick->studsYDir, dCoord, x, y, z - 1);

cout << "Studs in Childlayer: " << countStudsForChildLayer << endl;
if(countStudsForChildLayer == 0){
    int childLayerCounter = 1;
    while(childLayerCounter <= z){

        int countStudsForChildLayers = CheckIfPlaceable(brick->studsXDir,
        brick->studsYDir, dCoord, x, y, childLayerCounter - 1);

        if(countStudsForChildLayers == 0 ||
           countStudsForChildLayers >= maxSupport){

            DuploBrick *supportBlock = new DuploBrick();
            supportBlock = brick;
            supportBlock->xCoord = x;
            supportBlock->yCoord = y;
            supportBlock->zCoord = childLayerCounter - 1;
            supportBlock->priority = 3;

            for(int i = 0; i < brick->studsXDir; i++){
                for(int j = 0; j < brick->studsYDir; j++){
                    dCoord[childLayerCounter - 1][x+i][y+j] = true;
                }
            }
            supportbricklist.push_back(*supportBlock);
        }
        childLayerCounter++;
    }
}
```

The functionality for handling when the gripper accidentally looses a brick is pretty straightforward. The thought process here is that the moment the gripper looses a brick the robot should immediately halt before it resumes the building process by going back to one of the dispensers to pick up a new brick. To simulate this, a new button is added to the GUI. When this button is pressed, a new function is called which removes the brick that the gripper tool is currently holding. This is done by first accessing the tool with the getItem function, see table 4.2. Then we can access every children that the tool is currently a parent of. When the robot picks up a brick and the brick is attached to the tool, then that brick is considered the child of the tool. Then for simulation purposes we delete all childs that the tool is currently a parent of while the robot is moving. Upon removing the brick from the tool, the robot is stopped and the main thread is ended. Then before starting the main thread again we first check and see if the dispensers has updated. See listing 4.13 for a overview of the function that is ran when simulating the tool dropping a brick.

Listing 4.13: The code of the function that is ran when simulating the tool dropping a brick

```
RoboDK *rdk = new RoboDK;
Item *tool = new Item(rdk->getItem("Gripper␣RobotiQ␣85", RoboDK::ITEM_TYPE_TOOL));
QList<Item> children;
children = tool->Childs();
if(!children.empty()){
    ROBOT->Stop();
    mainThread->detach();
    tool->Childs().back().Delete();
}
if(!simulateDispenser.isFinished()){
    simulateDispenser.waitForFinished();
}else if(!TwoByTwoBlockDispenser.empty() || !TwoByFourBlockDispenser.empty()){
    mainThread = new std::thread(&MainWindow::BuildStructure, this);
}
```

4.7.2 The modeling approach

When implementing the functionality for handling when the gripper accidentally looses a brick, a new Thing was added called ClawSensor, see figure 4.51 for a overview over the statemachine in this Thing.

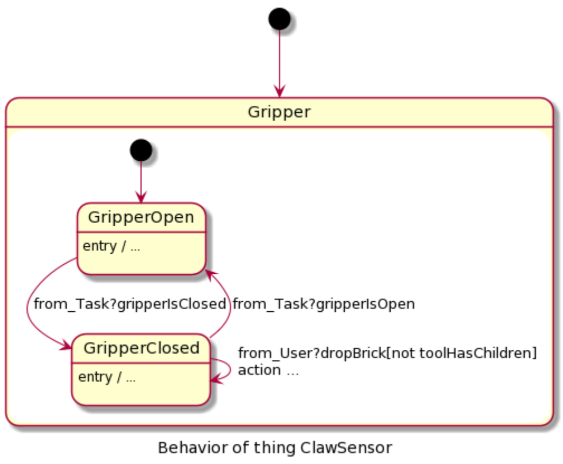

Gripper
GripperOpen
entry / ...
from_Task?gripperIsClosed
from_Task?gripperIsOpen
GripperClosed
entry / ...
from_User?dropBrick[not toolHasChildren]
action ...
Behavior of thing ClawSensor

Figure 4.51: Figure shows the ClawSensor statemachine created in scenario 4, its purpose is to listen for when a brick is dropped while the gripper tool is holding it.

ClawSensor consists of three ports for messages: a provided port for receiving gripperIsOpen and gripperIsClosed messages from Thing Task; a provided port for receiving a dropBrick message from the User Thing, User and it's timer function is utilized here as well for the same reason as in scenario 3 and 2; a required port sending a brickIsDropped message to Thing Task. Since the modelling approach is relying on the User and its timer function to tell ClawSensor when a brick is dropped, the messages gripperIsOpen and gripperIsClosed is utilized to make sure the tool actually has a brick or not when ClawSensor receives the message that the brick is dropped. At program start up, the statemachine in ClawSensor enters the state GripperOpen. In the modelling approach, the simulation is programmed to always have the gripper tool open when starting the program. A functionality which the thing Task has that the direct approach do not have, is the constant closing and opening of the gripper when it is picking up bricks and placing them down. In this scenario the message gripperIsClosed is sent to ClawSensor from Task when the gripper is closing, and vice versa. When ClawSensor is in the state GripperOpen and receives the gripperIsClosed message then it will transition over to its other state called GripperClosed. Upon entering the GripperClosed state it checks the childs of the tool to make sure it is actually holding a brick. While in this state we have two possible transitions. One transition takes us back to to the state GripperOpen, this happens when ClawSensor receives a gripperIsOpen message from Task. In the other transition GripperClosed will transition to itself, this happens when ClawSensor receives a dropBrick message from User after its timer has run out, also in order for this to happen the gripper must be holding a brick. Upon transitioning it will stop the robot, delete brick that the tool is currently holding and send a brickIsDropped message to Thing Task.

Upon receiving the brickIsDropped message, the running state in Task will transition to itself and start the building process again. For a full overview of how Fragments and Things are connected to each other in this final version see figure 4.52 and see figure 4.53 for a overview over how the Things are configured to communicate wit each other in this approach.

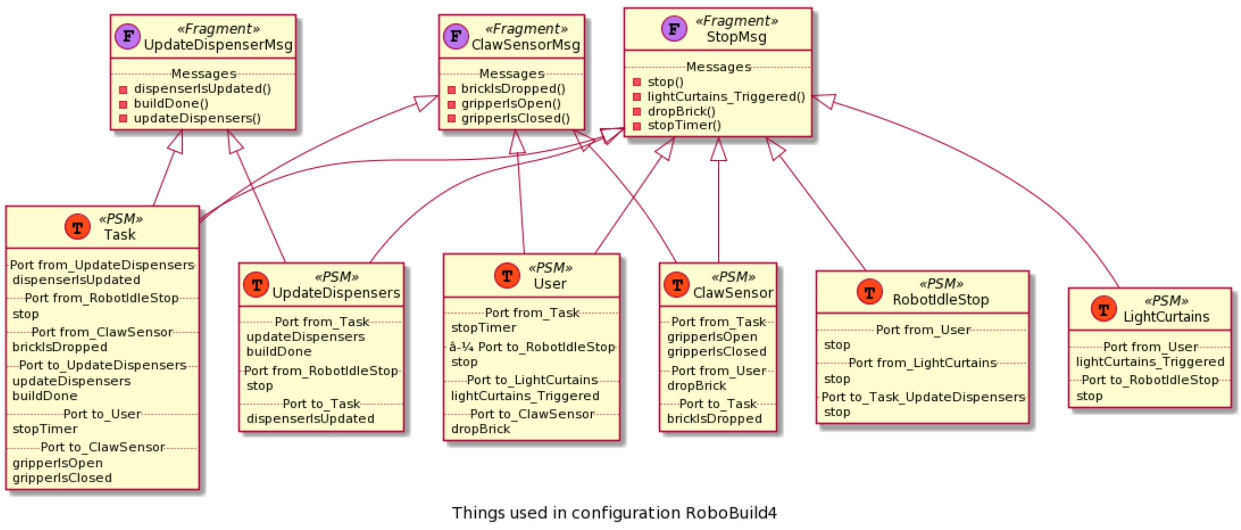


Figure 4.52: Figure shows the Things and Thing Fragments and how they are connected to each other for scenario 4

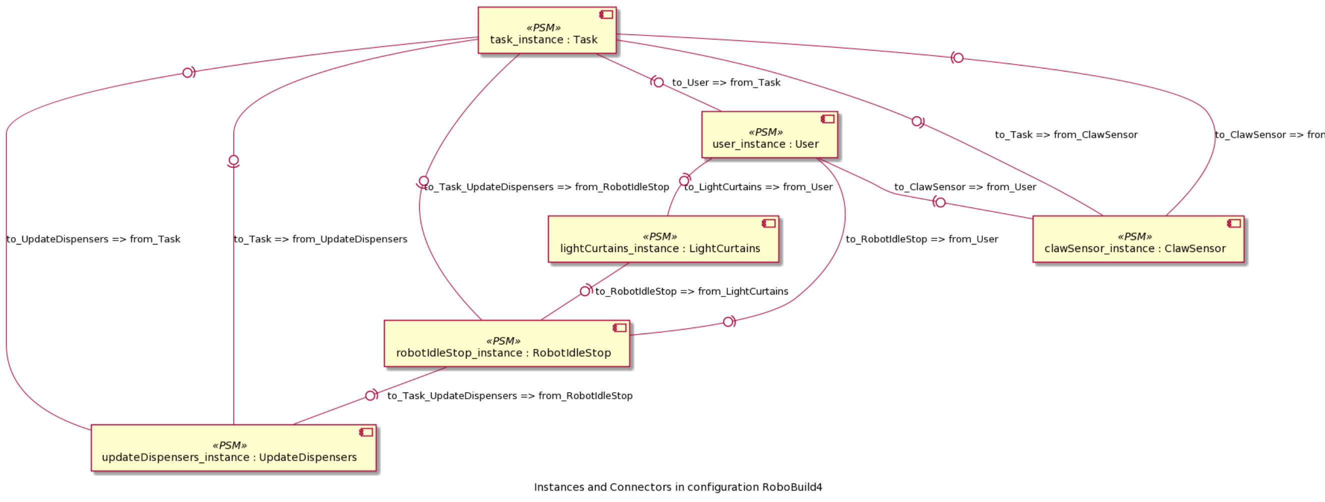

«PSM» task_instance : Task
to_User => from_Task
«PSM» user_instance : User
to_Task => from_ClawSensor
to_ClawSensor => fron
to_Task_UpdateDispensers => from_RobotIdleStop
to_LightCurtains => from_User
to_ClawSensor => from_User
to_UpdateDispensers => from_Task
to_Task => from_UpdateDispensers
«PSM» lightCurtains_instance : LightCurtains
to_RobotIdleStop => from_User
«PSM» clawSensor_instance : ClawSensor
to_RobotIdleStop => from_LightCurtains
«PSM» robotIdleStop_instance : RobotIdleStop
to_Task_UpdateDispensers => from_RobotIdleStop
«PSM» updateDispensers_instance : UpdateDispensers
Instances and Connectors in configuration RoboBuild4

Figure 4.53: Figure shows the configuration of how Things communicate with each other in scenario 4

Chapter 5

Evaluation

In this chapter we will discuss the results of the development and testing for each scenario and compare the modelling and direct approach with each other. This chapter is divided into 4 sections where we will discuss each evaluation process consisting of Productivity, Efficiency and Reliability, and Scalability as discussed in chapter 2, followed by a summary at the end.

5.1 Productivity

In this section we will discuss the thought process that went into each scenario and evaluate the results based on the productivity aspect as discussed at the end of chapter 2. Each scenario is presented with a two tables showing the character and line count for each approach and the differences between them. Challenges that occurred during development including algorithmic difficulties is discussed.

5.1.1 Scenario 1

Statistics for the modelling approach (MA) and the direct approach (DA) in Scenario 1		
Language - Approach	Character Count	Line Count
ThingML - MA	9827	308
C++ - MA (Generated)	36575	778
C++ - DA	9358	234

Table 5.1: Table shows the character and line count for both approaches for Scenario 1. Do note that these are approximations. While blank spaces are ignored, comments and console outputs for troubleshooting purposes are still included in the statistics

Looking at table 5.1 we can see that if we were to compare the modelling approach to the direct approach that they are pretty similar in terms character and line count. One thing to note is that The algorithm used in this thesis is by far not a perfect algorithm and by only including 2x2 and 4x4 bricks the algorithm became a bit more complicated than it should have been. Also the algorithm is not recreated in ThingMl in the modelling approach. The algorithm was recreated in ThingML at first but was decided to exclude it from the modelling solution for the reason that ThingML is more of a monitoring and event

Difference			
Language - Approach	Comparing Language - Approach	Difference Characters	Difference Lines
ThingML - MA	C++ - DA	469	74
C++ - MA (Generated)	C++ - DA	27217	544

Table 5.2: Table shows the difference in characters and lines between the approaches for Scenario 1

based language. If we were to recreate the whole algorithm in ThingML it would end up just being a translated version of C++ which would make development more complicated which is unnecessary. Based on this, it was concluded that ThingML should rather receive a already sorted and complete list of building bricks and then work from there. Lastly, a lot of the times (this counts for Scenario 2, 3 and 4 as well), developing the algorithm ended up being a huge part of the direct approach, based on this, the statistics in the table represents the ThingML code without counting the algorithm which was developed in C++, as mentioned, the ThingML code receives the list of bricks which this algorithm sorts out. The statistics for the direct approach however do include the line and character count of the algorithm. If the algorithm was recreated in ThingML then the modelling approach would result in more character and line counts, but this is mainly due to how ThingML code is built up compared to C++ and additionally we had to create a lot of custom objects and use a lot of kickdown.

The amount of effort was very similar in both approaches in terms of setting up the code to have it up and running. What required the most effort was firstly how this algorithm should look like in the direct approach. Secondly, how the approaches should use the list created from the algorithm, and thirdly, how the dispensers should be simulated. Since the GUI was already on the C++ level it was easier to tackle these problems first in C++. Also, ThingML do not support multi dimensional arrays which made creating an algorithm in ThingML first very complicated, and, as mentioned, was not necessary but this conclusion came very late during development. Another point to add is that since everything was running on C++ level it was easier to just create a direct approach first since testing and troubleshooting is more comfortable in QT rather than creating a solution in ThingML with a lot of kickdown (which ThingML cannot check for errors), generating code, inserting it into QT, run and then troubleshoot.

Realizing it was not the best approach to recreate an exact copy of the algorithm in ThingML it was decided to rethink the problems at hand in a different view. One of the advantages we noticed with ThingML is its ability to create a system with multiple parts working in parallel. Therefore, instead of thinking how the direct approach can be recreated in ThingML, the thought process changed to what parts of the solution could be divided up and work together concurrently. Not only did this open up new challenges to the modelling approach but made us rethink the direct approach as well. And having parts of the system work concurrently makes sense as well since if for example the dispensers can update at the same time the robot is building then downtime will be reduced because the robot does not have to wait for the dispensers to update once its done placing a brick.

To achieve this, threads was used in the direct approach which was pretty straightforward, as the RoboDK documentation for C++ suggests, should we wish to run multiple instances of RoboDK which we have to do to simulate processes concurrently, we have to use threads. Using this method, a thread was assigned to the function for updating the dispensers, after the robot has picked up a brick, the thread is declared and the updating of the dispensers runs concurrently. The results were as expected and the updating of dispensers was called each time the robot picked out a brick and worked concurrently to the main function.

In the modelling approach however, the thought process was a bit different. The goal was to here as well achieve dispensers and robot-movements that work in parallel but without the thread, since ThingML already contains functionality to achieve concurrency. Instead of separating the updating of dispensers into its own function and assigning it its own thread, both the main process of picking up and placing bricks, and the updating of dispensers are separated into their own Things. These Things communicate via messages, when the robot has picked up a brick from a dispenser, a message is sent to the Thing that handles the dispensers which in principal will happen in parallel while the robot finishes its movements. The modelling approach has a lot of its functionality based on the direct approach in the form of kickdown because of the C++ RoboDK API.

Therefore the conclusion is that (this goes for Scenario 2, 3 and 4 as well), given a problem, it is recommended that the main idea of a solution should first be tested on the direct approach since the modelling approach depends a lot on the kickdown for its solution to work. The solution can of course be theorized and a prototype can be made in the modelling approach first but since both approaches depends on RoboDK's C++ API it was generally better to test a solution in C++ first, as a developer we have to keep in mind that the ThingML compiler cannot check if the kickdown code is viable or not. Because of this, a prototype was often made first in the direct approach. Once it was tested and worked as intended then we had a foundation for how the kickdown in ThingML should look like and what variables will be needed. Then a prototype in the modelling approach could be made without worrying if the kickdown would work or not. Developing solutions this way however made our modelling approaches in a way reliant on a "direct way first" thought process but we believe that this is the best way to do it when the ThingML solution are as reliant on a lot of kickdown code as in the modelling approaches in this thesis.

To summarize scenario 1:

- The direct approach have to use threads to achieve concurrency.
- The modelling approach which uses ThingML have built in functionality to achieve concurrency and is not reliant on threads.
- Due to the amount of kickdown needed in the modelling approach, it is recommended that a prototype is made and tested in the direct approach first.
- A solution can of course be theorized and a prototype can be made in the modelling approach first, but the developer have to keep in mind that the ThingML compiler cannot check if the kickdown code is viable or not.
- Making an algorithm in ThingML was complicated and did not utilize ThingML's key features. It was concluded that the generated code should rather receive the output of the algorithm which in this case was a sorted list of bricks and were to put them.

5.1.2 Scenario 2

Statistics for the modelling approach (MA) and the direct approach (DA) in Scenario 2		
Language - Approach	Character Count	Line Count
ThingML - MA	12871	428
C++ - MA (Generated)	52770	1142
C++ - DA	11695	293

Table 5.3: Table shows the character and line count for both approaches for Scenario 2. Do note that these are approximations. While blank spaces are ignored, comments and console outputs for troubleshooting purposes are still included in the statistics

Difference			
Language - Approach	Comparing Language - Approach	Difference Characters	Difference Lines
ThingML - MA	C++ - DA	797	126
C++ - MA (Generated)	C++ - DA	41075	849

Table 5.4: Table shows the difference in characters and lines between the approaches for Scenario 2

In scenario 2 the problem statement involved adding a emergency stop button as well as upgrade the application to utilize text files consisting of voxel coordinates for building a custom 3D model of the user's choice. In the direct approach a extra function was added at start up before the algorithm sorts the list of building bricks. This function goes through the text file and creates a list of building bricks which the algorithm then sorts accordingly. Apart from this, the challenge was to implement a functionality that would stop the robot immediately or as close to immediately as possible. Based on the development in scenario 1, we would have to use threading here as well. A emergency button was added to the GUI and ended up being non-interactable until the program had finished. This was solved by assigning a thread (for context called main thread) to the main program which handles the picking up of bricks and placing them at the correct spots. Then a new function was added, this function is connected to the button in the GUI representing an emergency stop button and calls for the robot to stop all movements, this is done by killing the main thread. In case the main thread for whatever reason will not end, a global boolean variable is set to be true once the emergency button is pressed. The function handling the building process checks this boolean variable each time before getting a new brick in case the main thread fails to stop or the robot fails to stop in RoboDK, ofcourse this variable should not be used for the program to stop but exists purely for safety reasons.

In the modelling approach using ThingML we were sadly not able to connect a button from the GUI to somehow work with the generated code. Instead a new Thing was added to the ThingML code, this Thing simulates a user and uses a timer to indicate when the user has pressed the emergency button. Once the timer is finished a message will be sent out to another new Thing which monitors when a emergency occurs, this Thing then in turn sends out messages to other Things (For robotmovement and updating of dispensers) currently running telling them to as soon as possible stop all execution of codes internally. To be able to stop on command the Things which handles the building process and updating dispensers, were both added a composite state. The composite states in both Things has a state called running which handles the updating and building processes, then when a stop message is received these composite states will enter a stop state ultimately stopping all internal processes in the Things.

To summarize scenario 2:

- In the direct approach a thread had to be assigned to the main function, or else we were unable to interact with the emergency button in the GUI.
- In the direct approach when emergency stop is triggered, the main thread is stopped, with added safety features.
- In the modelling approach we did not have to worry about stopping threads or adding safety features. Using composite states and messages solved this for us.
- In the modelling approach we are currently not able to connect buttons in the GUI to Things and Messages in ThingML, therefore a Thing was added which simulates a user.

5.1.3 Scenario 3

Statistics for the modelling approach (MA) and the direct approach (DA) in Scenario 3		
Language - Approach	Character Count	Line Count
ThingML - MA	14088	473
C++ - MA (Generated)	64138	1355
C++ - DA	14614	351

Table 5.5: Table shows the character and line count for both approaches for Scenario 3. Do note that these are approximations. While blank spaces are ignored, comments and console outputs for troubleshooting purposes are still included in the statistics

Difference			
Language - Approach	Comparing Language - Approach	Difference Characters	Difference Lines
ThingML - MA	C++ - DA	-526	122
C++ - MA (Generated)	C++ - DA	49524	1004

Table 5.6: Table shows the difference in characters and lines between the approaches for Scenario 3

In scenario 3 the problem statement involved adding some sort of human harm avoidance similar to the Kuka robot set-up at Østfold University College's robot lab. In addition to this, the algorithm is to be upgraded to include support bricks for uneven structures. This was done by adding a extra function to the algorithm, the way this function works is by checking previous layers to see if there are previously placed bricks to support the new bricks getting placed. This is to avoid floating bricks like we mentioned in chapter 3. Apart from this, scenario 3 is pretty similar to scenario 2. For human avoidance, the mentioned kuka set-up utilizes light curtains to keep the robot from colliding from objects or limbs that may enter its work space while its working. The solutions for both the approaches in this case attempts to simulate this as well. Once the light curtains triggers from something entering the robot's workspace, it sends a emergency stop signal to the robot. This lets us reuse some features used in scenario 2.

To Start off, the direct approach is added an additional function which acts the same as the emergency button in scenario 2. A button is added to the GUI which purpose is to simulate a person sticking their arm into the robot's workspace. When this happens an arms will spawn in the simulation and then the same procedure as when emergency stop is triggered is executed: Stopping the main thread, Stopping the robot, and setting a global boolean variable true which the main program checks before getting the next brick (again as a safety feature). The triggering of the light curtains is assigned its own thread here as well because we are adding a object into the simulation.

In the modelling we approach we used the User Thing for this solution as well since, as mentioned in scenario 2, we were not able to connect buttons in the GUI to Things and messages in ThingML. This time though, the User Thing is set to send a message (again, based on a timer) to a new Thing created for simulating the Light Curtains. This message tells the Light Curtain Thing that a user has entered the workspace with their arm and thus a emergency stop is called. This is done by sending a message to the Thing handling emergency stops which we added in in scenario 2, which in turn sends out a message to the Things doing the building and dispenser updating processes telling them to immediately stop.

To summarize scenario 3:

- The improvements to the algorithm only applied to the direct approach. The list sorted by the algorithm is utilized by the modelling approach here as well. Since no algorithmic changes is added to the modelling approach, as we can see from table 5.5 and 5.6, the direct approach now counts more characters and lines.

- In the direct approach, a new thread was assigned to a new function which purpose is to simulate a person sticking their arm into the robot's workspace. The thread is declared and is assigned to the new function upon a click from the newly added button to the GUI.

- In the modelling approach, a new Thing was added which acted as the light curtains. When the User Thing sends a message to the light curtains Thing, it will see this message as a User sticking their arm into the robot's workspace. Then when the light curtains are triggered a message is sent to the emergency stop Thing from scenario 2 which in turn sends a stop message to the Things handling the building and updating of dispenser processes.

5.1.4 Scenario 4

In scenario 4 the problem-statement involved a improvement to the algorithm where one coordinate in a 3D array equals one stud on the Brick. In addition, functionality was to be added that could handle situations where the gripper tool accidentally looses a brick, the idea here was that the tool is to be added a sensor which lets the program know when the gripper have lost a brick. What would also to be included in the algorithm was the ability to rotate 4x2 bricks like in the previous algorithm and build sub-figures for structures that were too uneven. Due to the limited time left as mentioned in earlier

Statistics for the modelling approach (MA) and the direct approach (DA) in Scenario 4		
Language - Approach	Character Count	Line Count
ThingML - MA	17602	570
C++ - MA (Generated)	79581	1681
C++ - DA	16848	387

Table 5.7: Table shows the character and line count for both approaches for Scenario 3. Do note that these are approximations. While blank spaces are ignored, comments and console outputs for troubleshooting purposes are still included in the statistics

Difference			
Language - Approach	Comparing Language - Approach	Difference Characters	Difference Lines
ThingML - MA	C++ - DA	754	183
C++ - MA (Generated)	C++ - DA	62733	1294

Table 5.8: Table shows the difference in characters and lines between the approaches for Scenario 4

during implementation this was dropped and the new algorithm focused mainly on its ability to add support bricks where its needed.

To start off in the direct approach, the algorithm was upgraded to now have one coordinate be equal to a singular stud on brick. The algorithm followed the same principals as the previous one developed in scenario 1 but more complex since it now has to check and count several more neighbour positions from the first true value it finds in the 3D array. When the layer count is one or higher the algorithm also have to take into consideration the available studs in the previous layer. At specifically this, the algorithm has a major flaw. As of now when a brick is placed down, wether it is a 4x2 or a 2x2 it needs to be placed on top an equal amount of studs. This is not really optimal, a 4x2 brick can be placed on top of just 4 or even 3 studs without too much of a problem. This also causes some issues where we might end up with a lot of spots where there could potentially have been placed a brick but is empty due to this algorithm's flaw. When adding support bricks, the algorithm also follows the same principals as the one in scenario 3, here if the layer count is one or higher it will iterate through all previous layers and check available studs at the current brick's position x coordinate- and y coordinate-wise.

Lastly, the functionality for handling situations for when the gripper tool looses a brick was implemented. In the direct approach this was done by adding an additional button to the GUI where, when clicked, removes the brick that gripper tool is currently holding. Then inside a function in the program, the robot will stop, wait until dispensers are updated in case the robot has very recently picked up a new brick, and then start building again. This is achieved by monitoring and stopping and starting threads.

In the modelling approach, an additional Thing called ClawSensor was added which acted as a sensor on the gripper tool. Since it was not possible to connect a button from the GUI to Things in ThingML we had to utilize the User Thing which uses a timer. When this timer runs out it will send a message to ClawSensor telling it that the brick has been dropped. Then depending on if the gripper is closed and is actually holding a brick it will stop the robots movement and send a message to Thing Task which handles the robots movements to initiate with the building process again.

To summarize:

- Further improvements to the algorithm only applied to the direct approach. The list sorted by the algorithm is utilized by the modelling approach here as well. Looking at table 5.7 and 5.8 the modelling approach has now surpassed the direct. This is just mainly due to how many characters and lines there is in creating a Thing compared to just adding a button and a function. More kickdown is added aswell which only adds to the character count.
- In the direct approach, a new button and a function were added to simulate the dropping of the brick.
- In the modelling approach, a new Thing and four messages were added to simulate the dropping of the brick.

5.1.5 Summary

In early development it was decided to not include the algorithm for brick sorting in the modelling approach because recreating it in ThingML did not really show the advantages with using modelling and ThingML to create a solution. It was rather decided that the algorithm should look at the problem statements at hand and see how ThingML can tackle the problems differently instead of just recreating what was done in the direct approach. This led us to create solutions with multiple parts of the system working in parallel with each other. The observations from the development and testing phases proved that the modelling approach using ThingML was much more reliable in creating systems with multiple parts working asynchronous. It may or may not be possible to create an equal reliable solution through the direct approach but it requires knowledge on how to make a threadsafe application which is not needed going the ThingML route. In ThingML each part of the system, or Thing rather, is instanciated at start. What happens next is based off of how these Things communicate with each other, a Thing can also work alone completely independent on what the other Things are doing. This made creating solution consisting of parallel working systems much easier in the modelling approaches rather than in the direct approaches. With that said, the modelling approaches where heavily relying on a already working direct solution. This is due to the amount of kickdown needed in ThingML for simulating the necessary parts specified in the problem statements. It is a general rule that the kickdown in ThingML should be confirmed to be working in the lower level language, in this case C++, because the ThingML compiler cannot tell whether the kickdown code will work or not. So in conclusion to the development through all the scenarios, a direct approach has to be made and tested first. This is also easier for troubleshooting since RoboDK is directly connected to QT through the C++ API. The modelling approach also utilizes an already sorted list of bricks which is created by the algorithm which is only updated and edited in the direct approaches. Once a direct solution has been made, then we can continue on to the modelling approach looking at the problem at hand with a different mindset.

5.2 Efficiency and Reliability

In chapter 2 we mentioned how we wanted to test the performance of the of the different solutions and what kind of features where available to resolve issues and avoid further occurrences of said issues. Performance wise we wanted to test how well the robot was able to move and place the bricks without colliding with previously placed bricks or if it was able to pick up bricks in such a manner where it did not drop the brick immediately after picking it out of the dispenser. As mentioned, this was planned to be tested on a real robot but due to the circumstances of the pandemic we were only able to measure this simulation-wise. A solution for testing the approaches on real robots was proposed in chapter 4 where a connection and test run was conducted. The test run showed that after a connection between the QT application and RoboDK was done it was possible to have a simultaneous run between the simulation and a real robot from the GUI in QT.

Next we are shortly going to go over the scenarios discussing run-time and reliability by looking at what issues occurred and if any of the approaches provided any features to counter said issues. The discussion of the scenarios are divided into two parts the first one and then the rest. The reason for this is that scenario 2, 3 and 4 uses the same features for stopping the program since the problem statements for each of those scenarios involved adding safety features to the system. In addition the run-time did not change that much over the coarse of these scenarios which will be discussed when covering this

5.2.1 Scenario 1

The direct approach managed to achieve a simulation where the dispensers and the robot worked concurrently using threads. The run-time of the approach was tested on a simple pattern, and ended up being approximately: 28.161 seconds.

For the modelling approach however, it should be noted that the solution did not manage to achieve 100% concurrency during the testing of the scenarios. While in principal a program created in ThingML already achieves concurrency, when running the generated code, in the simulation the approach looked more like a turn-based execution based on a queue system where the next action is of highest priority. The priority of the next action is then based on how the Things works internally and how they communicate with each other via messages. The run-time of the approach was tested on the same pattern, as the one mentioned in the direct approach above, because the modelling approach did not achieve the same level of concurrency as the direct approach using threads, the run-time here ended up being approximately: 35.599 seconds.

5.2.2 Scenario 2, 3 and 4

Aside from updating the application to include the usage of 3D models as input, the problem-statement included adding an emergency button to the systems, as described in chapter 2.2.1, this meant assigning more threads to the direct approach. Since the application was locked when running the building process, we assigned a thread to this too. The main point of this was to have a button which was suppose to be the emergency stop button. And we had to be able to press it any time in order to get a as close to real world simulation as possible. Since the emergency button was designed forcefully end the main thread, some issues regarding bugs was observed. Depending on what the robot was doing it would sometimes not stop at all until it checked the global boolean value (which is set to true upon emergency button click) or do some bugged movements which it was not programmed to do. Overall this solution did not prove to be as reliable as we wanted since the robot sometimes did not stop after ending the thread or using the stop function

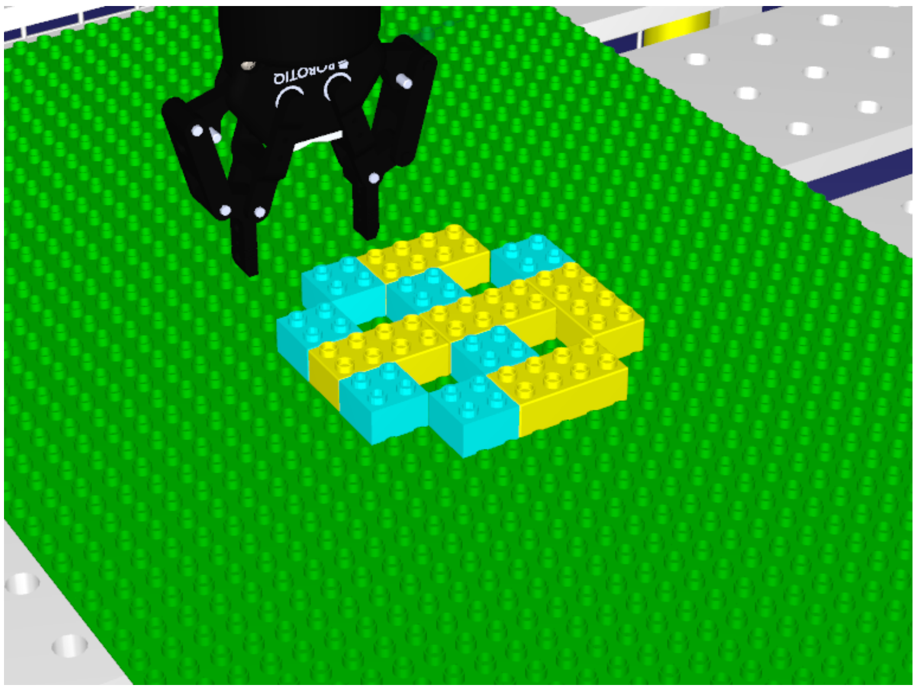


Figure 5.1: Figure shows the pattern which both approaches were tested on to calculate run-time

but by checking the global boolean value before getting the next brick. This is obviously not optimal because if the application fails to end the thread or the thread takes too long before it is stopped then the robot will continue to work and finish its task before stopping.

In the modelling approach we added composite states to the Things that take care of the building process and updating of the dispensers. This proved to be a much more reliable method of dealing with immediately, or as close to immediately as possible, stopping the robot on command. When the Things receive the stop message, the composite states will transition from the Running state which is the building process or the update dispenser process (depending on which Thing we are looking at), and enter a stop state. In the stop state the function for stopping the robot is ran and then the program is stopped. This proved to be much more reliable than what was done in the direct approach, every time the Things received a stop message, the robot would finish its current movement before stopping completely. This result was the same on every testing.

Based on the testing the conclusion that was made was that the modelling approach proved to be the best for the types of situations as described in scenario 2 through 4 based on the following points: 1. No need for threads or thread killing, composite states handles the types of situations mentioned above much better because they will constantly monitor and immediately be the first in the queue to execute code, which in this case is stopping the robot, once it receives a stop message; 2. There is no need for a global boolean variable for fail-safeing in case the processes are not stopped correctly; 3. There are no bugs caused from ending threads. The modelling approach might not have been able to completely stop the robot on command but proved to be more reliable than the direct approach in terms of bugs and unwanted movements occuring.

To shortly summarize in terms of efficiency in performance based on run-time and reliability:

- Following the direct approach we sacrifice reliability for faster execution.
- Following the modelling approach we sacrifice faster execution for better reliability.

5.3 Scalability

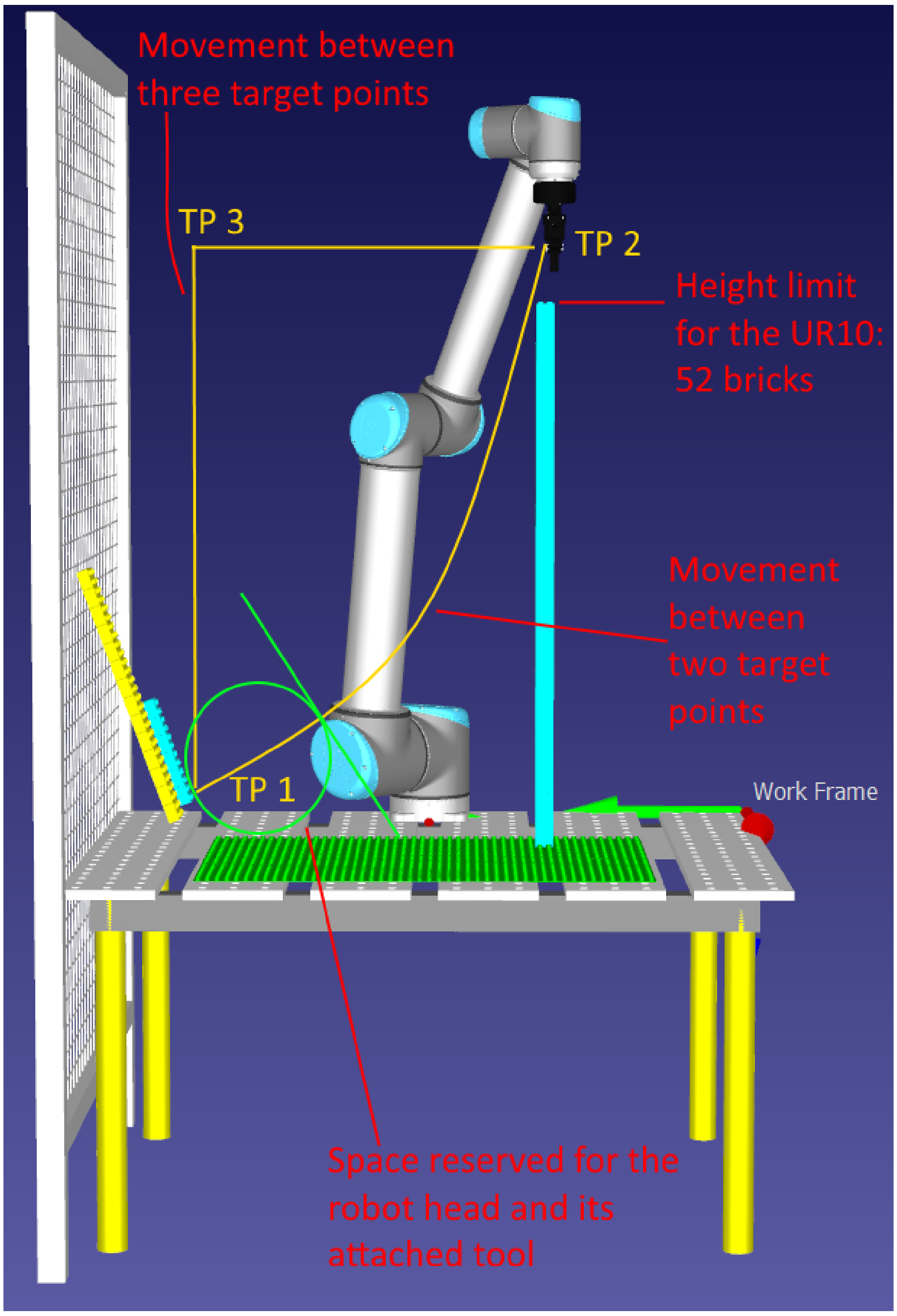

Movement between three target points
TP 3
TP 2
Height limit for the UR10: 52 bricks
Movement between two target points
TP 1
Work Frame
Space reserved for the robot head and its attached tool

Figure 5.2: Figure shows the height limit of specifically the UR 10 robot. In addition we can see the robots trajectory movement between the target points which suggests that an additional target point should be added if the structure ends up too close to the dispensers.

In this chapter we will quickly go over the scalability part of the testing of each scenario. In the scalability part we wanted to test how well the robot does when tasked to build something significantly larger or operate at much higher speeds. This was something we initially wanted to test on the real robots to see if the robots could potentially run into trouble when constructing large scale figures causing more challenges for each approach. As an example, one of these challenges could be a large and very uneven shape. Depending on height and how uneven the shape is the robot might have to pre-calculate its trajectory when placing bricks in order to avoid collision. When testing the robots limit in terms of height, it was observed that this limit is 52 bricks when the structure is as close to the robot as it can be. This limit is of course decreased the further out the structure spans. Unfortunately testing on the real robots is something that could not be done due to not having access to the real robots. Therefore we mentioned that we wanted to see how well this goes in a simulation environment. One thing that was observed was when the structure gets too close to the dispensers, the robots trajectory between picking up a brick and placing it on top of a large structure might cause a collision. If that is the case then an additional target point have to be added to the robots movement. Figure 5.2 illustrates this.

Lastly, testing the speed of the robot in a simulation environment does not give us any insight to what challenges this could have in a real life situation. When the robot is tasked to do something faster in the simulation, it will do the same as before just faster without causing any issues to the construction process. However, one challenge that could potentially be a issue in a real life setting would be that since we want the robot to operate faster it might not be able to place the bricks correctly, the robot may actually have to be given some time to place the bricks down in a safe manner. Another challenge could be vibration, when the robot is moving too fast it might cause vibrations in the robot's workspace potentially collapsing the structure or the gripper tool loosing the brick that its holding.

Chapter 6

Discussion

In this chapter the results and findings of the scenarios are discussed. Each scenario as described in chapter 3 was conducted and additional changes including pros and cons for each development process was noted which is going to be discussed further in this chapter.

As mentioned when the background for each evaluation process was explained in chapter 1, the testing part ended up being limited to only simulations. As described in each scenario in chapter 3, each scenario called for new improvements and added functionality to the system to handle unpredictable events. These events included checking if the dispensers were empty, how to handle when a person sticks their arm within the robot's reach while its working, and adding an emergency stop button or trigger. These events may only be some of many others that might occur in the real world. Being able to test each scenario on the real robot might have been able to reveal more unpredictable events that requires some sort of error handling which could be included in new additional scenarios where the task was to handle the newly explored events.

6.1 RQ 1, Comparing development of robot systems by model orientation versus more directprogramming methods

In this chapter we will discuss Research Question 1. What are the main differences between developing a robotic system using a Domain-Specific Language with code generation framework, and a system using more direct approaches on the simulation software RoboDK?

While it was not obvious at first, programming through a model-driven approach compared to a direct approach required different mindsets. Rather than asking ourselves "How can I do the same in ThingML?", we should rather ask "Are there other ways to tackle the problem in ThingML?". A lot of the time when developing and improving the systems throughout the scenarios the direct approach was often the first approach to go in to. The reason for this is that it was much more comfortable development wise since it is faster to set up some code, run it, see what went wrong, fix it and then run the program again. In this case we did not really have that freedom if we decided to develop in ThingML first because of the amount of kickdown needed for the solution to work. After all it is preferable that the kickdown works correctly because there is no way for ThingML to know if our kickdown code will compile in the lower level language and our solutions required a lot of kickdown or references to the lower level language, in this case C++. The mindset was often: If we have a direct solution that works then it is easy to implement it in ThingML as well, this made our decisions biased towards a direct thought-process which in the early development phases ended up with creating solutions in ThingML which was just translated versions of the direct approaches. The development of the algorithm which purpose was to add bricks in correct order in a list served as a example of this. While it was possible to make a new or remake the algorithm in ThingML, this did not utilize some of the key features which ThingML has to offer, which led to the conclusion that the problem statement at hand should be tackled differently than the direct approach. So what is difference between developing direct compared to modeling? With the direct approaches the thought process was pretty much straight forward, we look at the problem and design functions and classes that will create a solution. With the modeling approaches, we rather look at the problem and discuss what parts of this system can work in parallel. The advantages with ThingML is creating a system with two or more parts that work together in parallel to each other, if we did not take advantage of this then there was really no point in creating a solution with ThingML at all. If we wanted to do the same in C++ we had to use threads. In conclusion, compared to the direct approach using C++, ThingML used in the modelling approach already comes as a package containing functionalities to achieve concurrency in a system where it is desirable to have multiple parts working in parallel. Execution-wise the generated code will take care of that for us, thus we are able to just develop a state-machine based system without dealing with how we can assign certain parts of the simulation to work in parallel with each other. In the direct approach we have to assign threads to each element we want to work in parallel in the simulation, and then how to interact with and stop these threads based on which event occurs.

6.2 RQ 2, Improving performance and reliability of the developed software

In this chapter we will discuss Research Question 2. What are the main differences in performance between the model-driven and direct approaches when requirements for improvements and alterations to counter unpredicted events appear?

In order to answer this we wanted to know what changes needs to be done with the application when improvements is needed. When and what errors occurs during implementation of the new improvements and how these were countered. During the scenarios we added in total five improvements where two of these were algorithmic, these were added to the already existing system created in scenario 1. These improvements consisted of a emergency stop button, light curtains for when something enters the robot's workspace while it is working, and how the system should handle situations where the gripper looses the brick while holding it. Firstly, we wanted to measure the robots performance in the real world which was changed to simulation only due to not having access to the real robots, however, in early development it was noted that measuring the robots performance in building structures in simulation only was sort of meaningless. While RoboDK does let us know when the robot has collided with something or is on route to collide with something we simply just adapt to it in the code. If the gripper is too low we can just program it to be a little higher up than the structure and then when it is right above where it has to place a brick it can just move to the center point of the assigned brick placement, hover down and then just place the brick. There were not exactly any new challenges that we could discover by just observing the robot in the simulation which was the purpose for testing the approaches on the real robots. When it comes to the approaches time-wise both approaches were pretty similar. The run-time of each approach were measured which showed that the modelling approach was a bit slower but was more reliable because of not having to rely on threads. The direct approach was faster but was not as reliable due to high chances of bugs using threads and killing threads where we want the robot to stop did not provide the same results every time. Sometimes the robot would do some bizarre movements which it were not programmed to do, and other times it only stopped after checking the global boolean variable which is set to true upon emergency stops. We believe this is mainly due to forcefully closing threads whenever we want the robot to stop. What made the modelling approach more reliable was how Things communicate with each other using Messages where instead of forcefully stopping a thread, the procedure for stopping whichever process is running gets the highest priority in a turn-based execution which made the modelling approach a little bit slower than the direct. In theory, all the Things should operate in parallel to each other but the generated code ended up, based on the testing, as turn-based execution based on a queue system where the next code running is of the highest priority. This priority ofcourse is based on how the Things works internally and how they communicate with each other.

6.3 R3, Higher production rate

In this chapter we will discuss Research Question 3. When the robot is altered to operate at higher speeds or tasked to build something significantly larger and more complex than it has previously been tasked to build, what are the main differences in changes that has to be done between the two approaches?

This was more hinted towards when the robot is operating at higher speeds or is tasked to build something significantly larger than before, what changes needs to be done to the approaches to encounter as few errors as possible. And if or when errors occurs then at what speed or construction process did it happen? This question ended up being a bit complicated to find a definitive answer for since we were not able to test on the real robots. For the first part we wanted to increase the robots speed to see what potential errors could occur, simulation-wise moving at a fast pace was not a problem for the robot since the simulation does not exactly represent a complete real world scenario. What could for example have been an issue is that when the robot is moving too fast it could cause vibrations or not be able to place bricks correctly which could result in a unstable structure. For the second part we wanted to increase the size of the structure which the robot would be tasked to build. Ofcourse, there will be limitations anyways but the question is when and why. Simulation-wise one potential problem we could see was the robot's movement, using the MoveL function in RoboDK, the movement of the robot will be a linear line between two target-points. Moving in a linear line from the dispensers to the top of a really high structure would make the robot crash into the structure if it was too close to the dispensers. This however can be fixed by just adjusting the target-points or adding a new one as suggested in chapter 5.3, and either of the approaches did not provide any specific method for handling this differently that would require less effort or better results. If we were able to study the robots in the real world constructing large structures it is not unlikely that we might have encountered some new challenges which was desirable in the first place in order to answer this research question. Some of these challenges could have been large and very uneven structures where one side reaches the robots height limit and with the help of sensors or cameras as examples, the program then has to carefully map out how the robot should move so the highest side of the structure is built last.

6.4 What could have been done differently and future work

At the start of this thesis we had a very clear goal in mind, we wanted to create several scenarios containing different problem statements where we would develop two different approaches, a direct one using C++ and a modelling one using ThingML. At first it was not clear what we wanted the robot to do in order to answer the research questions. In the end it was decided that these scenarios and problem statements would be created around the real world setup at Østfold University College, which consists of a robot: either Kuka or UR 10, a gripper tool connected to the robot, emergency stop buttons, light curtains, and for testing purposes, lego bricks with a baseplate for building structures. However, in early development after creating the first algorithm and recreating it in ThingML we had to reconsider the development process of both approaches because we realised that we did not utilize some of the key features ThingML has to offer. As mentioned, we just ended up creating a translated version of the direct approach. While we did manage to utilize ThingML's features for creating a solution with several parts working in parallel to each other, it was concluded that in a future work the robot should do more monitoring related tasks instead of algorithmic related tasks in order to reach a more precise conclusion when exploring different methods to program robots using modelling as one of the main components. An example of such tasks could be a robot spray painting large objects such as cars, here we constantly have to monitor the amount of paint left in the robots container, once it runs out, the robot either has to pause or refill on its own. For example we could have a Thing here which constantly monitors the paint container and once it is empty it sends a message to Thing handling the robots movements. It is of course also recommended in a future work that that the approaches is tested on the real robots since it might open up to new issues or challenges in which the final solution for both approaches have to adapt to.

Chapter 7

Conclusion

In this thesis we have created solutions for two different approaches in order to explore different methods to program industrial robots, specifically this was aimed towards Kuka KR3 C4 and the UR 10. These two approaches consisted of a modelling route using ThingML as the Domain Specific Language, and a direct route using C++. Both approaches were connected and executed on the simulation software RoboDK. The solutions were developed and tested during four different scenarios. For each scenario starting from scenario 1 the complexity and difficulty increased by a small amount where several improvements and functionality were added to solve the problem-statements. These problem-statements consisted of: adding and improving an algorithm which assigned coordinates to different duplo bricks for the robot to place them correctly. This algorithm included support bricks and sub-figures to avoid floating bricks while the robot builds the structure; a emergency stop button which purpose was to instantly stop all robot movements; light curtains which purpose was to trigger a emergency stop whenever something entered the robot's workspace; and functionality for handling when the robot accidentally drops a brick. In order for the added functionalities to work we needed each process in the simulation to work in parallel with each other. To achieve this, threads were used in the direct approach while ThingML in the modelling approach already comes with functionality designed for concurrency. The results of the testing suggests that the modelling solution using ThingML is more reliable when designing a system using robot with a lot of monitoring or concurrency than the direct approach because of how messages and Thing's event listeners works. However the the modelling approach did prove to be slower because of its turn-based execution, but we believe this is mainly due to how RoboDK works. If we want to run or monitor different processes in parallel in RoboDK, then their document states that we have to create new instances of RoboDK have them run in their own threads. We did not run threads in ThingML because we believe it could have complicated the messages ending up in for example the Task Thing receiving a message that the dispensers are updated while in reality it is not. However, we do not completely dismiss the idea that perhaps finding a way to use threads inside the ThingML code would make a smoother execution on specifically the RoboDK software. The results of the testing and evalutation suggests that going the modelling approach, specifically ThingML, a given problem should be tackled with a different mindset than when programming with C++. When we program in ThingML the idea is to utilize its key features such as having multiple Things work in parallel and communicate with each other via Messages. Recreating algorithms in ThingML ended up just being translated C++ versions and did not come with any advantages and only complicated things due to not having access to for example 3 Dimensional arrays. For achieving concurrency in a system we add Things, Fragments and Messages in ThingML for each part we want to work concurrently while in C++ we add functions and assign them with their own threads. If one were to choose between the two for programming industrial robots via RoboDK specifically, it should be noted that both approaches comes with their own pros and cons. Going the modelling route we sacrifice fast run-time for good reliability. Going the direct route we sacrifice good reliability for fast run-time. In terms of scalability, we were sadly not able to test limits in terms of structure size and speed on the real robots, which meant the end results was solely based on the observation of the simulation. The results suggested that when it comes to very large and uneven structures the robot might have to pre-calculate its trajectory before-hand based on perhaps sensors or something similar. It would need more target points or edit current target points in order to avoid collision.

Bibliography

[1] KUKA.PLC mxAutomation. URL: `https://www.kuka.com/en-us/products/robotics-systems/software/hub-technologies/kuka,-d-,plc-mxautomation`.

[2] Model-driven engineering. URL: `https://en.wikipedia.org/wiki/Model-driven_engineering`.

[3] Model transformation language. URL: `https://en.wikipedia.org/wiki/Model_transformation_language`.

[4] MOF Model to Text Transformation Language. URL: `https://en.wikipedia.org/wiki/MOF_Model_to_Text_Transformation_Language`.

[5] Online Voxelizer. URL: `https://drububu.com/miscellaneous/voxelizer/?out=obj`.

[6] RoboDK: Simulate Robot Applications. URL: `https://robodk.com`.

[7] Simulink. URL: `https://en.wikipedia.org/wiki/Simulink`.

[8] TelluIoT/ThingML: The ThingML modelling language. URL: `https://github.com/TelluIoT/ThingML`.

[9] Kai Adam, Katrin Holldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, pages 172–179, Taichung, Taiwan, April 2017. IEEE. URL: `http://ieeexplore.ieee.org/document/7926535/`, `doi:10.1109/IRC.2017.16`.

[10] Kai Adam, Katrin HO Lldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. page 14, 2017.

[11] Ana Cavalcanti, Augusto Sampaio, Alvaro Miyazawa, Pedro Ribeiro, Madiel Conserva Filho, André Didier, Wei Li, and Jon Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37, April 2019. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0167642318301655`, `doi:10.1016/j.scico.2019.01.004`.

[12] E. Estévez, Alejandro Sánchez García, Javier Gámez García, and Juan Gómez Ortega. ART2ool: a model-driven framework to generate target code for robot handling tasks. *The International Journal of Advanced Manufacturing Technology*, 97(1-4):1195–1207, July 2018. URL: `http://link.springer.com/10.1007/s00170-018-1976-z`, `doi:10.1007/s00170-018-1976-z`.

[13] Yingbing Hua, Stefan Zander, Mirko Bordignon, and Bjorn Hein. From AutomationML to ROS: A model-driven approach for software engineering of industrial robotics using ontological reasoning. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Berlin, Germany, September 2016. IEEE. URL: `http://ieeexplore.ieee.org/document/7733579/`, `doi:10.1109/ETFA.2016.7733579`.

[14] T. Wrütz J. Rehbein and R. Biesenbach. Model-based industrial robot programming with MATLAB/Simulink. In *2019 20th International Conference on Research and Education in Mechatronics (REM)*, pages 1–5, 2019. `doi:10.1109/REM.2019.8744113`.

[15] Yusuke Maeda, Ojiro Nakano, Takashi Maekawa, and Shoji Maruo. From CAD models to toy brick sculptures: A 3D block printer. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2167–2172, Daejeon, South Korea, October 2016. IEEE. URL: `http://ieeexplore.ieee.org/document/7759340/`, `doi:10.1109/IROS.2016.7759340`.

[16] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18(5):3097–3149, October 2019. URL: `http://link.springer.com/10.1007/s10270-018-00710-z`, `doi:10.1007/s10270-018-00710-z`.

[17] Hochgeschwender N. Wrede S. Wigand D. Nordmann, A. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering for Robotics*, pages 75–99, 2016. URL: `https://aisberg.unibg.it/handle/10446/87804#`.

[18] Hochgeschwender N. Wrede S. Wigand D. Nordmann, A. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering for Robotics*, pages 75–99, 2016. URL: `https://aisberg.unibg.it/handle/10446/87804#`.

[19] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. page 12.

[20] Chiharu Sugimoto, Yusuke Maeda, Takashi Maekawa, and Shoji Maruo. A 3D block printer using toy bricks for various models. In *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, pages 958–963, Xi'an, August 2017. IEEE. URL: `http://ieeexplore.ieee.org/document/8256227/`, `doi:10.1109/COASE.2017.8256227`.

[21] Monika Wenger, Waldemar Eisenmenger, Georg Neugschwandtner, Ben Schneider, and Alois Zoitl. A model based engineering tool for ROS component compositioning, configuration and generation of deployment information. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Berlin, Germany, September 2016. IEEE. URL: `http://ieeexplore.ieee.org/document/7733559/`, `doi:10.1109/ETFA.2016.7733559`.

[22] Dennis Leroy Wigand and Sebastian Wrede. Model-Driven Scheduling of Real-Time Tasks for Robotics Systems. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 46–53, Naples, Italy, February 2019. IEEE. URL: `https://ieeexplore.ieee.org/document/8675604/`, `doi:10.1109/IRC.2019.00016`.

[23] Stefan Zander, Georg Heppner, Georg Neugschwandtner, Ramez Awad, Marc Essinger, and Nadia Ahmed. A Model-Driven Engineering Approach for ROS using Ontological Semantics. *arXiv:1601.03998 [cs]*, January 2016. arXiv: 1601.03998. URL: `http://arxiv.org/abs/1601.03998`.