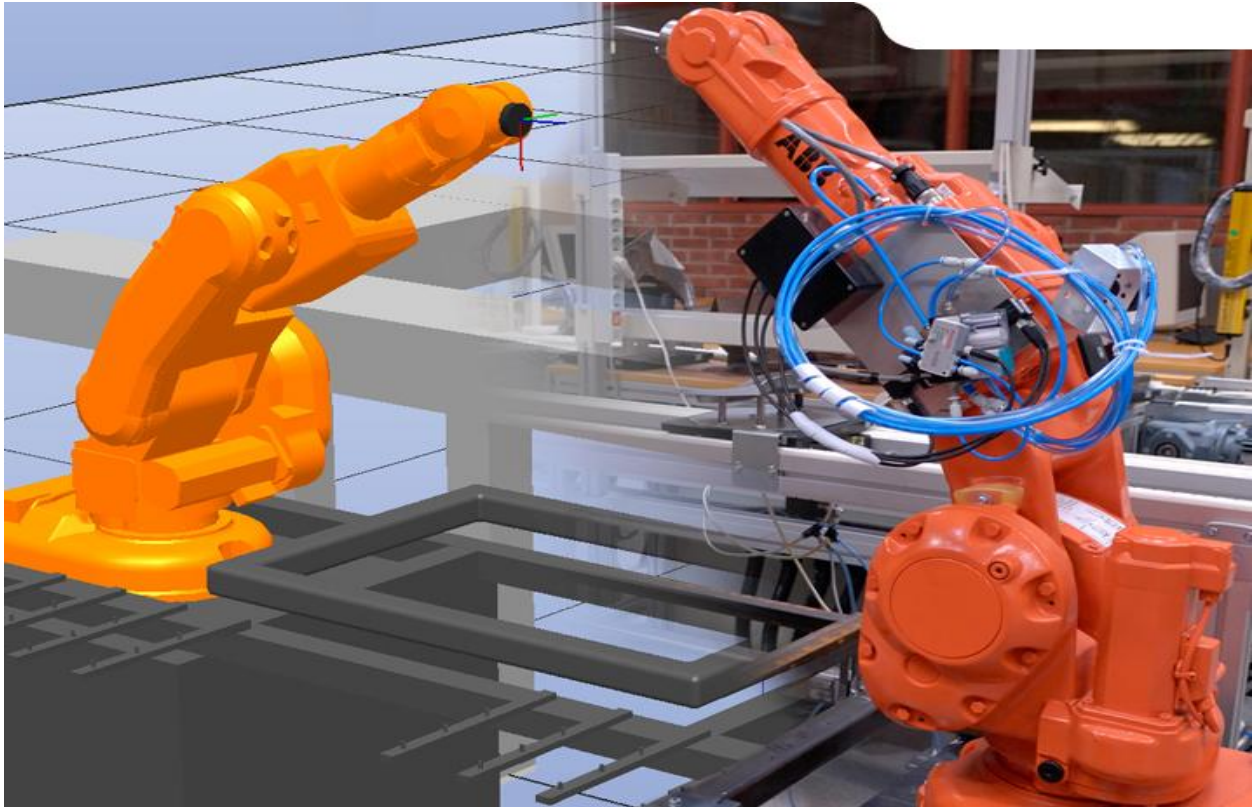


CHALMERS



Development of BoxSweeper and BoxSweeper PLC: Real verification and synthesis of a zone booking system using Supremica, RobotStudio and Simatic Step 7

Master of Science Thesis

Daniel Bengtsson
Carl-Johan Rutgersson

Department of Signals and Systems
Division of Automation research group
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2008
Report No. EX027 / 2008

Abstract

When mechanical objects such as robots and fixtures are moving in the same area there is potential risk for collisions. This is especially true when many robots are sharing a mutual workspace. To avoid collisions, bookable work zones are generated, only allowing one single moving equipment to be in that zone at a time. This behavior is supervised by a control function. A major problem is that it is very hard to guarantee that the control functions fit with the specification and thus never will reach undesirable production states such as collisions or deadlocks.

This is the third master thesis, in cooperation between Chalmers University of Technology and ABB, in a project to solve the problem with collisions. Earlier work has generated an add-in to ABBs simulation and robot programming tool RobotStudio. The add-in is called BoxSweeper and it creates a supervisor that prevents that the robots within the robot cell to collide.

The first objective for this master thesis was to verify that BoxSweeper created collision free robot programs that could be used in a real controller. The BoxSweeper add-in was unstable in the beginning, but after stability improvements, a complete robot program was generated. The code was successfully tested in a real controller.

The second objective for this master thesis was to create a simulation of a robot cell that should be used as an illustrating example for the 2008 bachelor thesis project. The simulation was successfully created in RobotStudio.

The third and final objective for this master thesis was to develop BoxSweeper so that the generated supervisor could be executed in a PLC (Programmable Logic Controller) and supervise the cell equipment via Profibus communication. The RAPID code was compared to the different PLC standard languages. Because of the similarities to RAPID, Structured Text was the language that was chosen. The BoxSweeper add-in was developed and a supervisor which executes in a Siemens PLC was successfully created. When using the BoxSweeper add-in the user now has the ability to choose which kind of supervisor to generate, BoxSweeper (operates in a robot controller) or BoxSweeper PLC (operates in a PLC).

Sammanfattning

När mekaniska föremål som robotar och fixturer rör sig i samma arbetsområde finns det en potentiell risk för kollisioner, framförallt när flera robotar delar på ett gemensamt arbetsområde. För att undvika kollisioner genereras bokningsbara arbetszoner, dessa får endast beträdas av en robot eller fixtur i taget. Detta övervakas av en kontrollfunktion. Ett stort problem är att det är väldigt svårt att garantera att kontrollfunktionen uppfyller specifikationen och därmed ser till att icke önskvärda tillstånd, som kollisioner och låsningar, inte uppnås.

Detta är det tredje examensarbetet, i samarbete mellan Chalmers Tekniska Högskola och ABB Robotics, inom problemet med att undvika robotkollisioner. Tidigare arbete har genererat en modul till ABBs RobotStudio, som är mjukvaran för att skapa och simulera robotprogram. Modulen har fått namnet BoxSweeper och den skapar en övervakare som förhindrar att robotarna i en gemensam cell kolliderar.

Första delmålet med detta examensarbete var att verifiera att genererad kod från BoxSweeper kunde laddas in i det verkliga robotsystemet och sedan köras. BoxSweeper-modulen var instabil i början men efter att stabiliteten förbättrats genererades RAPID-kod till robotsystemet. Den genererade koden kunde köras med lyckat resultat i ett verkligt system.

Det andra delmålet för detta examensarbete var att skapa en simulering av en robotcell som skulle användas som ett illustrerande exempel på det kandidatarbete som skall göras under vårterminen 2008. En cell med de uppsatta kraven skapades i RobotStudio med lyckat resultat.

Det tredje och sista delmålet för detta examensarbete var att utveckla en ny funktion, i BoxSweeper, som placerar övervakaren i en PLC (programmerbart styrsystem) och använder Profibus för att kommunicera med robotarna. RAPID-koden jämfördes med de olika språkstandarder som finns för att programmera PLC. Strukturerad Text valdes som språk på grund av likheterna med RAPID. Utvecklingen av den nya funktionen till BoxSweeper, som genererar en övervakare till en PLC, utfördes med lyckat resultat. Vid användning av BoxSweeper finns det möjlighet för användaren att välja vilken typ av övervakare som skall genereras, BoxSweeper (övervakare till en robot controller) eller BoxSweeper PLC (övervakare till en PLC).

Preface

This master thesis has been written by Daniel Bengtsson and Carl-Johan Rutgersson in cooperation with ABB Robotics and Chalmers University of Technology. This has been the last task in our education and there are quite a few people that have helped us with this project. Without your help we would still be struggling:

Petter Falkman – Chalmers University of Technology. Our supervisor and examiner, thank you for your guidance and encouragement during this project.

Magnus K. Gustafsson – ABB Robotics. Our supervisor at ABB, thank you for giving us the opportunity to write this thesis and for your support during this project.

Avenir Kobetski – PhD at Chalmers University of Technology. Thanks for all your support and answers to all our “stupid” questions.

Henrik Berlin – ABB Robotics. His help brought life to BoxSweeper. Thank you for the technical support.

Bertil Nelson – Semcon. Thanks for teaching us all about Profibus and PLCs.

Carl-Johan specially wants to thank:

My lovely wife Anna for all your support during these years

My family for always supporting and believing in me

Daniel specially wants to thank:

My mother and father for your support and for always believing in me

Gothenburg

2nd of April 2008

Daniel Bengtsson

Carl-Johan Rutgersson

Table of contents

Word List

- 1. Introduction..... 1
 - 1.1 Background 1
 - 1.2 Purpose 1
 - 1.3 Objectives 2
 - 1.4 Scope and limitations 2
 - 1.5 Conditions 2
- 2. Theoretical Background 3
 - 2.1 Discrete Event Systems..... 3
 - 2.1.1 Automata 3
 - 2.1.2 Plant..... 3
 - 2.1.3 Specification 4
 - 2.1.4 Supervisor 4
 - 2.2 Robotics 6
 - 2.2.1 Coordinate systems 6
 - 2.2.2 Configuration 7
 - 2.2.3 Robot Controller – IRC5 7
 - 2.2.4 Programming robots 7
 - 2.2.5 FlexPendant 8
 - 2.2.6 RAPID 8
 - 2.2.7 Keys..... 8
 - 2.3 RobotStudio 9
 - 2.3.1 Virtual controller 9
 - 2.3.2 Targets 9
 - 2.3.3 Paths 9
 - 2.3.4 Mechanisms..... 9
 - 2.4 BoxSweeper 10
 - 2.4.1 BoxSweeper communication..... 12
 - 2.4.2 Communication between the supervisor and the robot tasks..... 13

2.5 PLC	14
2.5.1 Ladder Diagrams.....	14
2.5.2 Instruction List.....	14
2.5.3 Structured Text.....	15
2.5.4 Sequential Function Charts	15
2.5.5 Function Block Diagrams	15
3. Implementation.....	16
3.1 Preparations	16
3.1.1 Mounting the robots to the floor.....	16
3.2 Implementation of BoxSweeper.....	17
3.3 Simulation of Bachelor Thesis project	18
3.4 Development of a PLC supervisor.....	19
3.4.1 Signal configuration for BoxSweeper PLC	23
4. Results	25
4.1 Implementation of BoxSweeper.....	25
4.2 Simulation of Bachelor Thesis project	25
4.3 Development of a PLC supervisor.....	25
5. Discussion	26
6. Future work	27

References

Appendix A - CodeWriter.cs

Appendix B – RapidCodeTemplatesPLC

Appendix C - BoxSweeper.cs

Appendix D – Example program for BoxSweeper PLC

Appendix E – Manuals

Appendix F – Symbol Table

Appendix G – Function block for communication

Word list

Automaton (Plural Automata)	Automaton is a model that describes the behavior of a discrete event system
BoxSweeper	BoxSweeper is an add-in to RobotStudio that creates a supervisor that prevents the robots within the cell to collide
Event	Events are actions that make the automaton move from one state to another
FlexPendant	The FlexPendant is a hand held operator used for controlling ABBs robot systems
IRC5	The latest generation of ABBs robot controllers
Path	A path is a sequence of move instructions to targets
PLC	Programmable Logic Controller
RAPID	A programming language developed by ABB designed to handle robot movements and time critical systems
RobotStudio	RobotStudio is ABBs software used for programming and simulating robot programs offline
Target	A target is a coordinate that contains information about position, orientation and configuration of a robot
Zone	A zone is the volume of space where collisions can occur between robots in a cell
Robot Controller	A device that controls the robot motions
Real robot system	The robot system in the PPU-lab.

1. Introduction

This section contains a short background of previous work and introduces the purpose and objectives for this master thesis.

1.1 Background

The production industry has never experienced tougher conditions than they do today. There is an increasing demand to produce more products within the same time span, while competing in a growing global market. In order to meet these challenges, automation (among other solutions) has been used to increase production volumes while keeping desired product quality. The products are getting more and more complex as well, which complicates the production processes even more.

When mechanical objects such as robots and fixtures are moving in the same area there is potential risk for collisions. This is especially true when many robots are sharing a mutual workspace. To avoid collisions, bookable work zones are generated, only allowing one single moving equipment to be in that zone at a time. This behavior is supervised by a control function. A major problem is that it is very hard to guarantee that the control functions fit with the specification and thus never will reach undesirable production states such as collisions or deadlocks. Today the production industry solves the problems with collisions between robots with manual programming. This is time consuming and yet collision avoidance cannot be guaranteed because of the human factor.

Since a few years there is cooperation between Chalmers University of Technology (henceforth Chalmers) and ABB Robotics (henceforth ABB) with the objective to create an add-in to ABBs simulation tool RobotStudio. The name of the add-in is BoxSweeper and it creates robot programs that are collision free using a supervisor that supervises the robots in real time. This add-in has been developed to an extent that real tests now are possible.

1.2 Purpose

The initial purpose of this master thesis is to verify and perform real tests of the BoxSweeper add-in in a real robot cell. ABB and Chalmers want to see if the current version of BoxSweeper works as intended, if not changes shall be made until a real test is performed successful.

Second, a simulation of a robot cell for a bachelor thesis shall be created. This simulation shall function as a suggestion of how the robot cell can be realized.

Third, the current function of BoxSweeper is limited to ABBs IRC5 robot controllers, which allows each controller to operate multiple robots. Since this option is not very common in the industry, ABB and Chalmers want to see if a modified BoxSweeper can be created so that the supervisor can be executed and supervised from a PLC.

1.3 Objectives

This master thesis has three objectives:

1. Generate a supervisor for the robot controller with the BoxSweeper add-in, to verify that collision free robot programs are created that works in a real robot controller.
2. Simulate a robot cell that shall be used as an illustrating example of the robot cell for the 2007/2008 Bachelor Thesis project.
3. Develop BoxSweeper so that the generated supervisor can be executed in a PLC and supervise the cell equipment via Profibus communication.

1.4 Scope and limitations

The real test shall be performed in the Chalmers robot cell in the PPU-lab. This robot cell contained two robots when the work on this master thesis started and therefore any real tests will be limited to two robots sharing a mutual workspace. The robot cell is equipped with a Siemens PLC, hence all generated PLC programs will have to be written to execute in a Siemens environment.

1.5 Conditions

This master thesis is based on the previous work in two master theses written by Haziraj, Fallström and Weinemyr, Zekaj. The BoxSweeper add-in has been developed since the last master thesis but the core functions in BoxSweeper is unaltered.

The work shall be performed during a period of 20 weeks, each week represents 40 hours of work.

2. Theoretical Background

This section contains the theoretical background and important concepts that are needed to understand the content of this master thesis. Initially an introduction to discrete event systems is given. Then there is a summary of the theory of robotics. This is followed by an introduction to ABBs RobotStudio and the add-in BoxSweeper. Finally there is a section about the different programming languages and theory about PLCs.

2.1 Discrete Event Systems

Discrete Event Systems (DES) are models of systems that at any time-instant occupy a unique state of being, out of a finite set of such states. Especially man-made, non-physical systems containing information, such as manufacturing systems, are profitable modeled as DES. The theory presented in this chapter is written by Fabian, M. *Industrial Automation*.

2.1.1 Automata

Automata are models that describe the behavior of discrete event systems (DES). An automaton is a directed graph where the nodes represent system states and the edges represent possible ways of transitioning between states in the occurrence of events. Events are actions that make the automaton move from one state to another.

An on/off switch can serve as a simple example of an automaton. If the switch is in its *off* state the event *Flip* causes the switch to change state into *on*. If another event *Flip* occur then the switch changes state again and returns to its initial state. This means that an automaton model of the switch has two states, *on* and *off*, and the event *Flip* is the action which causes the automata to change state. An example of the automaton for the switch can be seen in Figure 2.1.

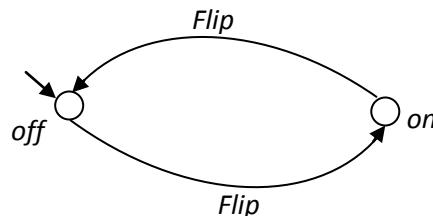


Figure 2.1: A switch represented as an automaton

2.1.2 Plant

It is important that the model captures the behavior of the real system. Relevant characteristics of the real system, that are important for the user, must be chosen when designing a model. These models are called plants, denoted P. A plant that describes unnecessary information complicates manipulation of the model. The automaton to the left in Figure 2.2 is a simple plant.

2.1.3 Specification

A specification, denoted S_p , is an automaton that expresses the allowed behavior of a given plant P . S_p should describe the allowed behavior during the work cycle. Figure 2.2 shows a plant and a specification. The robot P has four positions: $[Home, Pose1, Pose2, Pose3]$. An allowed movement between these positions is represented in the plant by the execution of one of the four events: $[mp1, mp2, mp3, mH]$. The specification S_p says that the robot should move in a certain pattern. In this example the only accepted path for the robot is from Home to Pose1, Pose2, Pose3 and finally back to Home.

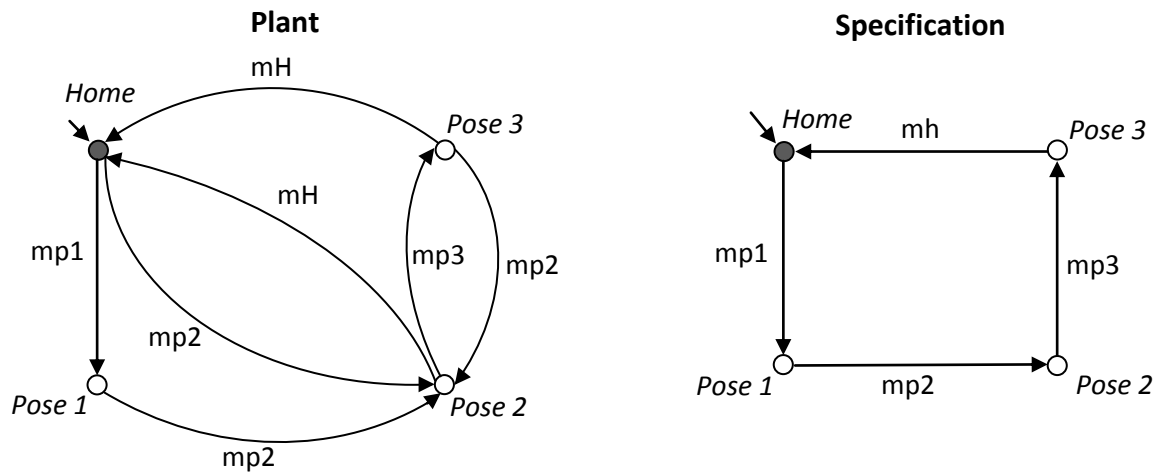


Figure 2.2: Plant and Specification for a robot

2.1.4 Supervisor

A supervisor is used to monitor and control the behavior of a plant given a specification. With the help of the specification S_p and the model P , a supervisor S can be generated. Automatic generation of a supervisor through some form of algorithm is what is referred to as synthesis.

S will observe and influence P so that it fulfills the specification. The synthesised supervisor is by construction non-blocking and controllable, these concepts will be explained below.

Observation

The supervisor will observe all events and control them by allowing and denying the execution of the events that will lead to undesired behavior.

Influence

The plant generates all events, the supervisor is a passive follower that dynamically restricts the choices of events that the plant can generate. That is the supervisor only influences the plant behavior by enabling or disabling events.

Fulfilling the Specification

There are no guarantees that the specification can be met exactly, due to limitations of the plant. A reasonable requirement is that the specification is met as well as possible, meaning the specification will describe the behavior.

Non-blocking

Some states may be of certain importance or special interest: those states may be denoted marked. The marked states are represented graphically in an automaton by a circle: all automata that have been shown in previous figures are marked. For example, no matter the movements and paths the robots take, they must always be able to get back to the home position. The home position is a natural choice for a marked state. Non-blocking means that movements should be performed in such a way that a marked state always is reachable.

Controllable

A supervisor that never disables an uncontrollable event is said to be controllable. This is important since there exists no way to execute or forbid an uncontrollable event in the real system. The events contained in a plant may be controllable or uncontrollable. A controllable event can be allowed and disallowed by the supervisor. An uncontrollable event must always be allowed. A supervisor may not be able to prevent the enabling of some of the events in a plant. For example a supervisor has no control over an emergency stop button but it still has to allow the button to be pressed. The emergency stop button is uncontrollable.

Modular and monolithic supervision

Monolithic supervision is when one single supervisor handles the whole system. A monolithic supervisor grows exponentially with cell complexity. When modeling a complex process it is beneficial to build the model out of models of less complex components. To divide the process into smaller plants and specifications and generate a supervisor for each of these plants is called modular supervision.

Assume there is a plant P modeled by two sub plants, P1 and P2. Create a supervisor S1 that supervises only P1 and S2 for P2. The modular supervisor of the system consists of the two parts S1 and S2. For an event to be allowed in the modular supervisor it needs to be able to execute in both S1 and S2. The benefits of this is the reduction of complexity, modular supervisors grows linearly and will not get as big as a monolithic supervisor.

2.2 Robotics

A robot is a mechanical device, also called manipulator arm, which can be programmed to perform various tasks. In industry robots are used for various purpose, e.g. welding and component assembly. In order to understand how the robot works some definitions need to be explained. The theory presented in this chapter is written by Bolmsjö, Gunnar S. unless there are other references.

2.2.1 Coordinate systems

A robotic arm is often modeled as a chain of rigid links connected by joints. In order to control the position of these links a couple of coordinate systems are used.

World Coordinate

Since the robot is placed somewhere in the workspace, a *world coordinate* is defined. The world coordinates can be placed anywhere in the workspace and they are used as the base reference point of the coordinate system. See the world coordinate in Figure 2.3.

Base Coordinate

The position of each robot with respect to the world coordinate is called the *base coordinate system*. The base coordinate system is placed in the base (the first link of the robot). For each link of the robot another coordinate system is created which refers to the previous link. See the base coordinate in Figure 2.3.

Tool Center Point

In order to describe the position of the tool the *tool coordinate* or the *tool center point (TCP)* is created. This is the most commonly used position when programming a robot since the position and the orientation of the tool is essential for the task to be performed. See the tool center point in Figure 2.3.

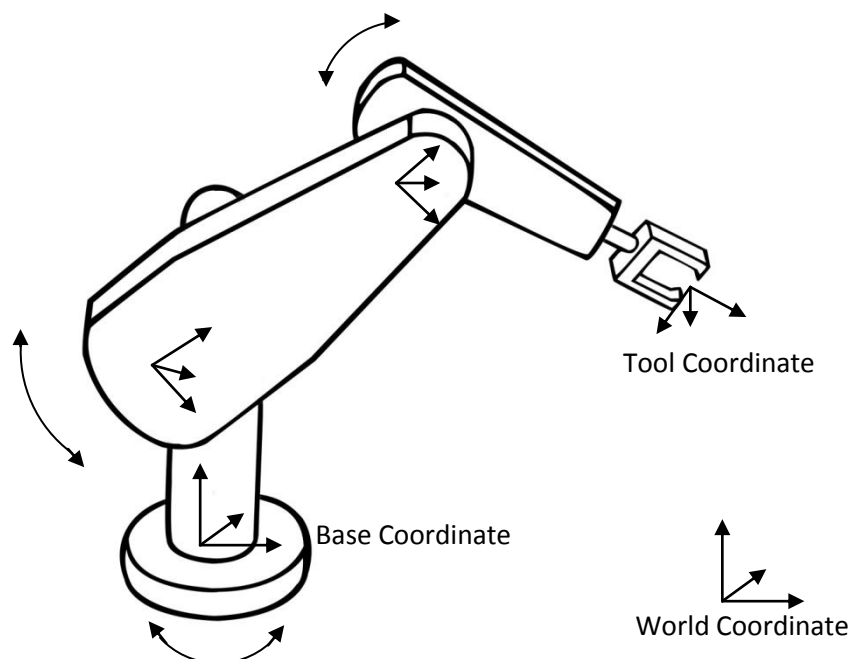


Figure 2.3: A manipulator arm and its coordinate systems

2.2.2 Configuration

Because of the mechanical construction and the many motions available for each joint the robot can reach the same position in the world coordinate in different ways, depending of the position of each joint. The different combinations of joint positions that reach a fixed position in the world coordinates are referred to as the configurations of the robot, see Figure 2.4.

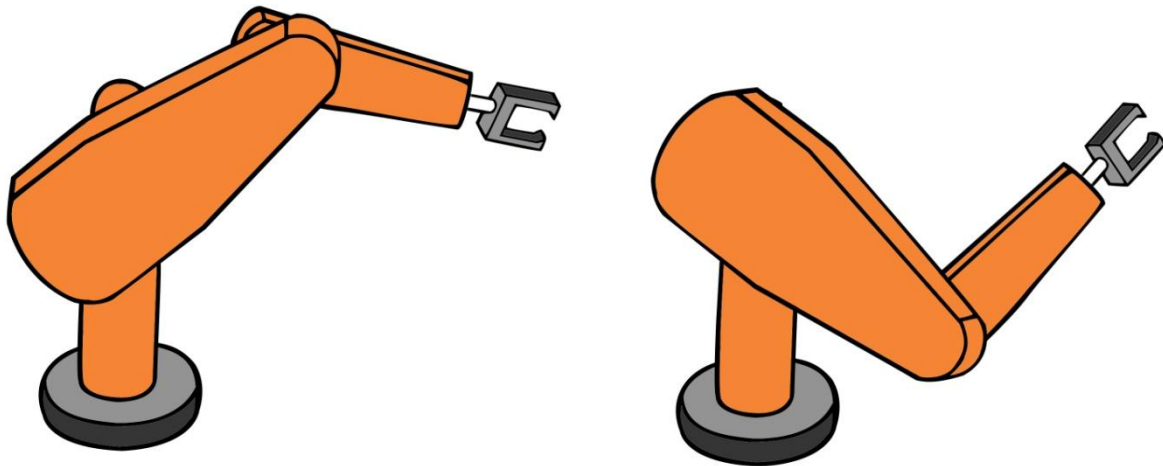


Figure 2.4: Two configurations of joint positions that reached the same position

2.2.3 Robot Controller – IRC5

An industrial robot behavior is managed by a robot controller. This controller runs the instructions the operator programs the robot to perform. The latest generation to ABBs robot controllers is the IRC5, the fifth generation of robot controllers. The controller supports fully synchronous multiple robot control through the MultiMove function. (ABB, Product Guide IRC5)

A basic IRC5 robot system consists of a robot controller, the FlexPendant, RobotStudio, and one or several robots or other mechanical units. There may also be process equipment and additional software options. (ABB, Operation Manual IRC5)

2.2.4 Programming robots

There are two different types of programming that are used with robots. The first one is called *online programming*. The behavior of the robot is controlled in real time by the programmer with the help of a controller, in ABB systems this operating unit is referred to as the FlexPendant.

The second type of programming is performed in a simulated environment with computer software, this is called *offline programming*. The programmer does not need a real robot while programming which means that there are no disturbances to the real production and robot programs can be made to systems that are not built.

2.2.5 FlexPendant

The FlexPendant is a hand held operator unit used to perform many of the tasks involved when operating a robot system: running programs, nr. 1 in Figure 2.5, jogging the manipulator, nr. 2 in Figure 2.5, and modifying robot programs. The touch screen, nr. 3 in Figure 2.5, allows the user to operate the controller. (ABB, Operation Manual IRC5)

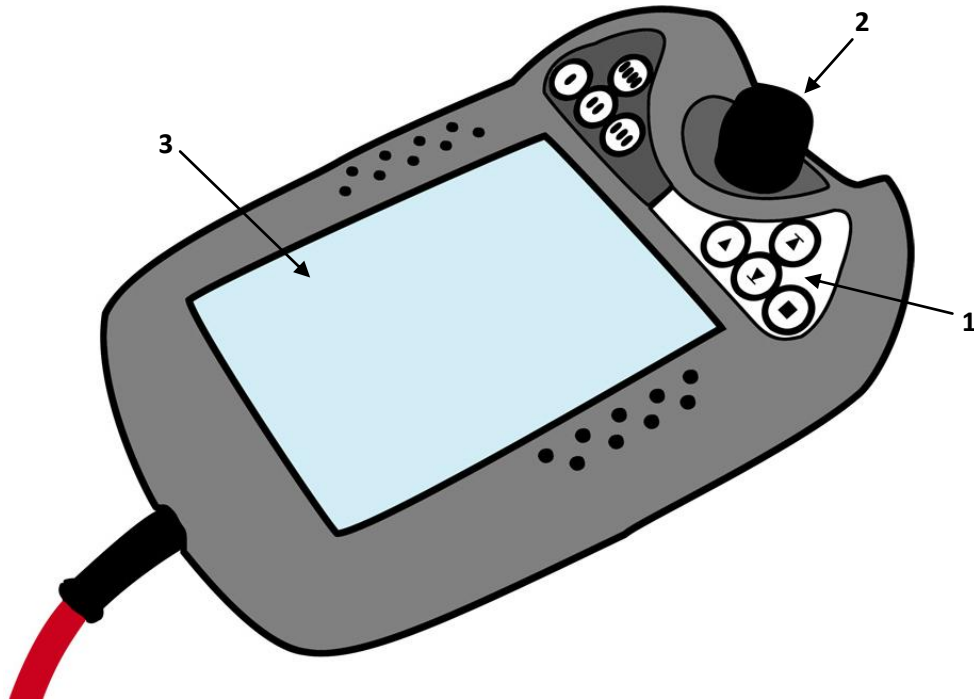


Figure 2.5: The FlexPendant

2.2.6 RAPID

The programming language developed by ABB is called RAPID and it is a high level language. The language is designed to handle robot movements and time critical systems. RAPID applications contain one or more tasks. Each task contains RAPID programs and system modules.

One of the modules contains the entry procedure, a global procedure called main. Execution of the program actually means executing the main procedure.

2.2.7 Keys

ABB uses license keys to grant access to applications or specific functions. An application may require additional license keys to allow access to optional features. (RobotStudio Help – Search – License management)

IRC5 systems uses three types of keys:

- Controller keys – Specifies the controller and software options.
- Drive keys – Specifies the robots in the system. The system has one key for each robot in the system.
- Additional options keys – Specifies additional options.

A virtual license key allows the user to select any options, but a system created from a virtual license can only be used in a virtual environment. (RobotStudio Help – Search – RobotWare concepts)

2.3 RobotStudio

RobotStudio is ABBs software used for programming and simulating robot programs offline. The software allows the user to create a virtual copy of an existing or future robot cell in 3D. There are a number of benefits with using a virtual cell to create the robot program, if the robots collide in the virtual environment there are no costs for repairs, production loss or tool re-coordination.

2.3.1 Virtual controller

The controller is represented as a virtual controller in RobotStudio. It is a true replica of the real controller and ensures the same behavior and functionality as in the real on-line system.

2.3.2 Targets

When programming a robot in RobotStudio the use of targets is essential. A target is a coordinate that contains information about:

- Position – The position of the target in a coordinate system.
- Orientation – The orientation of the target, used for aligning the TCP.
- Configuration – Values for each robot axis for how the robot shall reach the target.

2.3.3 Paths

A path is a sequence of move instructions to targets. Paths are used to make the robot move along a sequence of targets. (RobotStudio Help – Search – About targets and paths)

2.3.4 Mechanisms

A mechanism in RobotStudio is a graphical representation of a robot, tool, external axis or device. The various parts of a mechanism move along or around an axis. The use of mechanisms allows the user to create custom made objects in RobotStudio and thus use these objects to create a more detailed simulation of the users real robot cell. (RobotStudio Help – Search – Creating, compiling and modifying a mechanism)

2.4 BoxSweeper

BoxSweeper is an add-in to RobotStudio that creates a supervisor that prevents the robots within the cell to collide. The user follows a series of steps and the result is a task for each robot and an additional supervisor task. Figure 2.6 shows the BoxSweeper add-in.

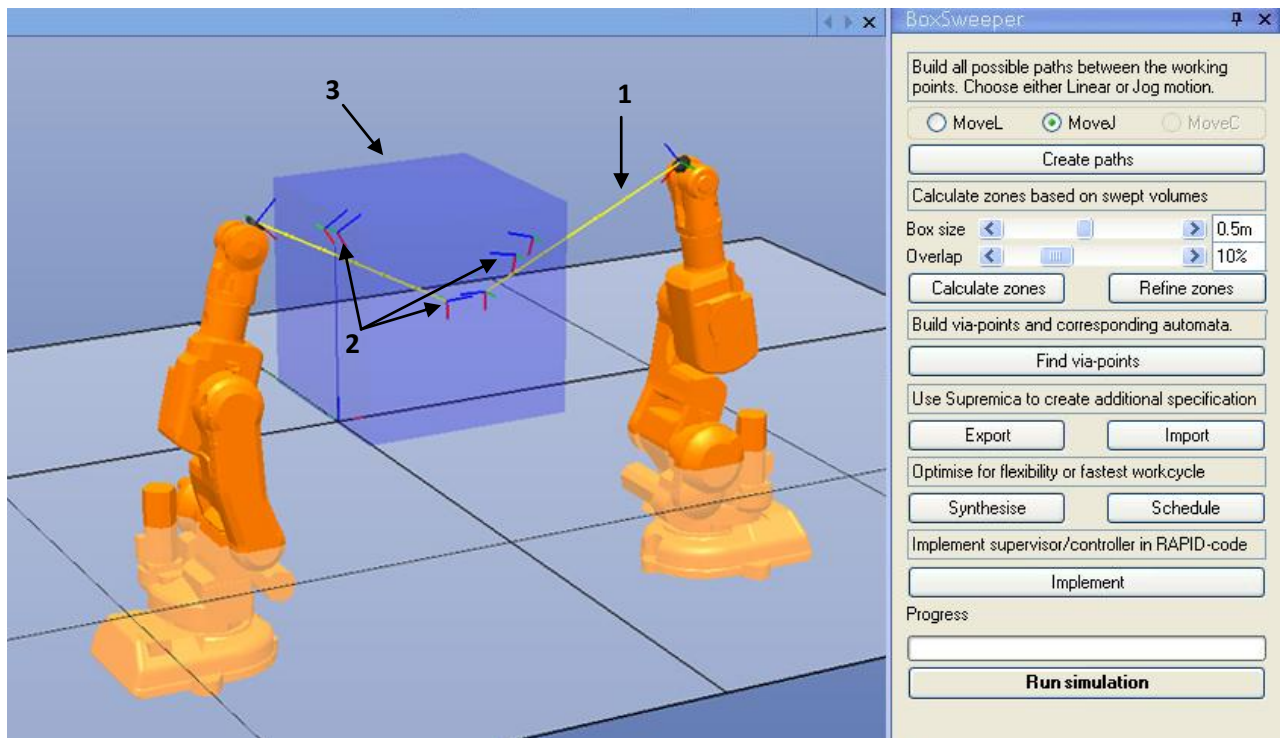


Figure 2.6: The BoxSweeper add-in, to the right, and two robots with a mutual zone

Step 1 - Create paths

If the user has not created any paths BoxSweeper will create paths, nr. 1 in Figure 2.6, between every target, nr. 2 in Figure 2.6. The user then has to choose which paths to use, since not all of the automatic generated paths correspond to the desired movement of the robots.

Step 2 - Zone calculations

To avoid collisions among robots, mutual exclusive zones (henceforth zones) will be generated. These zones can be accessed by at most one robot at a time. (Flordal, H. et al.)

These zones are represented as boxes in RobotStudio, nr.3 in Figure 2.6. The user chooses the Box size and Overlap between the boxes. These zones will be used by the supervisor to ensure that only one robot at a time will have access to a zone. A large box size will result in fewer zones, but smaller zones allow the robots to move closer to each other.

RobotStudio then generate zones around every axis of the robot. RobotStudio simulates the paths for the robot. As soon as any of the robots axis leaves the current zone a new zone is created. When the simulation is completed zones for the robots entire sweep volume has been generated.

When the sweep volume for each robot has been generated their mutual zones is identified and these zones are saved. These are the zones that the supervisor will have to supervise.

Step 3 – Find via-points

To determine when the zones need to be booked, each path is simulated a second time to find the robot positions where entry to and exit from the zones occur. This is done by having the simulating environment detecting the robots “collisions” with the zones. (Flordal, H. et al.)

RobotStudio creates via-points, new targets, for each entry and exit to the zones.

Step 4 - Export

The targets and paths are exported to Supremica, a tool developed by Chalmers that is based on Supervisory Control Theory. Supremica creates plants and specifications for the system and these are used to create a supervisor that is non-blocking and controllable.

Step 5 – Synthesise or Schedule

Next the user has to choose between a maximally permissive (Synthesise) and a cycle optimal (Schedule) supervisor. The cycle optimal supervisor only allows the events in the robot cell to happen in one optimal way, while the maximally permissive allows the events to happen in many ways.

Step 6 – Implement

This is the final step. During implement the robot programs are generated. A new task called Supervisor is created and the supervisor program is written to this task. When the implementation process is completed the “Run simulation” button simulates the final programs.

2.4.1 BoxSweeper communication

BoxSweeper is designed to operate in an ABB MultiMove system. The communication between the robot tasks and the supervisor task is achieved with global variables.

These global variables are:

- *Semaphore*, a Boolean that restricts more than one robot to communicate with the supervisor at once.
- *supervisorEvent*, a String that is given the name of the event that the robot want to perform.
- *supervisorRequest*, a Boolean that tells the supervisor task that there is a new event to evaluate.
- *supervisorAllow*, a Boolean that is given the result of the evaluation of an event.

Figure 2.5 shows the BoxSweeper procedures for a Robot and a Supervisor.

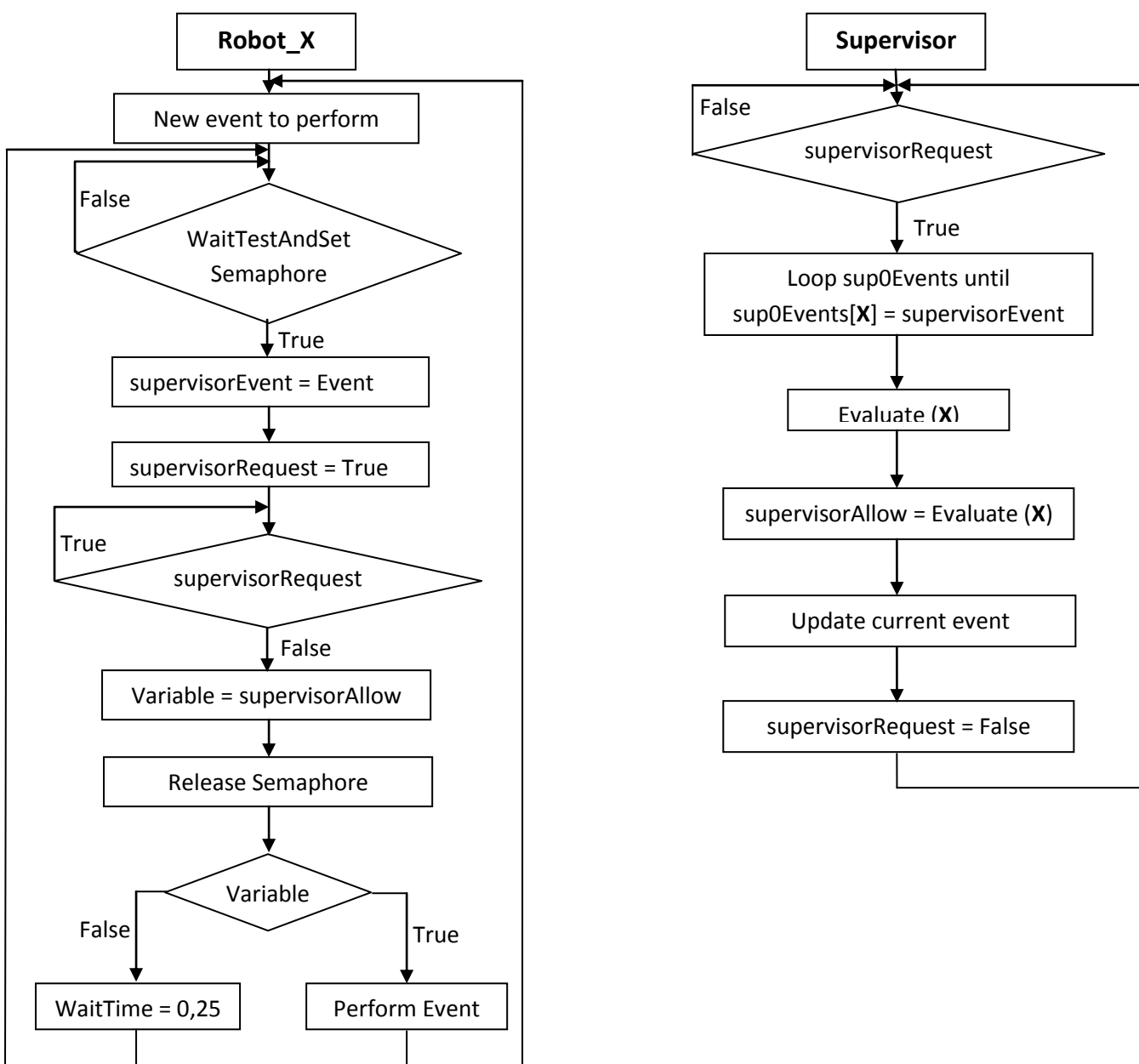


Figure 2.5: The BoxSweeper procedures for a Robot and a Supervisor

2.4.2 Communication between the supervisor and the robot tasks

Each robot has a number of events that they perform continuously in a certain order. In this example a robot named Robot_1 wants to perform an event called Event, in the real system the events have names such as "RB_u_0_m1_1". First Robot_1 needs to ensure that it is the only robot that is communicating with the supervisor. Robot_1 tries to set the Boolean Semaphore. This is done by a RAPID command called WaitTestAndSet that waits until a Boolean has the value *FALSE* and then sets it to *TRUE*. When the semaphore is set, Robot_1 is the only robot that can communicate with the supervisor task. The string in Event is copied to the supervisorEvent string. Next the Boolean supervisorRequest is set to *TRUE* and the Robot_1 program waits at this state until the supervisorRequest Boolean is reset, this is done by the supervisor task.

The supervisor task is a loop that constantly checks if there is an event to evaluate. The automaton that was created during "Step 5 – Synthesise or Schedule" is represented as arrays in the supervisor task. One of these arrays is called sup0Events and it contains all events that any robot in the system can perform. As soon as the supervisorRequest Boolean is *TRUE* the value of supervisorEvent is compared to the sup0Events array until the corresponding event is found. The value index of that event is an Integer (called X in this example) and it is used to calculate if the event is allowed or not. If the event is allowed supervisorAllow is set to *TRUE*, else it is set to *FALSE*. When the evaluation is completed the supervisor updates the current state in the automata arrays and then the supervisorRequest Boolean is set to *FALSE*. The supervisor task then continues to loop until there is a new event to evaluate.

As soon as the supervisorRequest Boolean is set to *FALSE* the program for Robot_1 continues. The program copies the value of supervisorAllow to an internal variable and then releases the semaphore. If the event is allowed the program continues and Robot_1 performs the event. If the event is not allowed the program waits 0.25 seconds before trying to set the semaphore again to start a new communication process with the same event.

If an event is allowed or not depends on the current state in the supervisors automata. If another robot Robot_2 is in a certain state and blocking the zone that Robot_1 want to enter Robot_2 will have to leave the zone before Robot_1 can enter it. The supervisor will keep denying Robot_1 to enter until Robot_2 has left the zone.

For detailed information on the evaluation procedure for the events see previous master thesis written by Haziraj, Fallström and Weinemyr, Zekaj.

2.5 PLC

A PLC (Programmable Logic Controller) is a digital computer used for automation in industrial processes. Originally the PLCs were designed to replace hard-wire relay-logic and their original heritage still influence their behavior and programming.

The operation of a PLC is very simple. It has a continuous scan cycle consisting of three events:

- 1) Read inputs
- 2) Execute program
- 3) Write outputs

The programmer writes the program which processes the inputs. There is a worldwide standard called IEC 61131-3 for PLCs which support the five most common programming languages: Ladder Diagram (LD), Function Block Diagrams (FBD), Instruction Lists (IL), Structured Text (ST) and Sequential Function Charts (SFC). IL and ST are textual languages while the others are graphical. However each manufacture of PLC systems has adapted the standard to work with their own hardware. (Jack, H.)

2.5.1 Ladder Diagrams

The Ladder Diagrams (LD) is the main programming method used when programming PLCs. It was designed to mimic relay logic. LD consists of graphic symbols laid out in networks. The result of the LD often represents a Boolean, which has the values true or false.

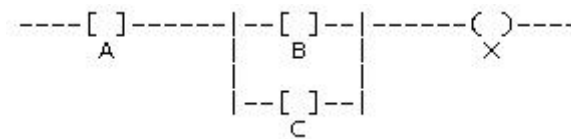


Figure 2.6: LD of the function $X = A \text{ and } (B \text{ or } C)$

Figure 2.6 is an example of a very basic LD network. The example realizes the function:

$$X = A \text{ and } (B \text{ or } C)$$

That means that if A and any of the variables B or C are TRUE, X will also be given the value TRUE. As the LD network grows more complex programs can be created, a LD program can consists of thousand of rows with function such as this. (Fabian, M.)

2.5.2 Instruction List

Instruction List (IL) is a low level language that resembles assembly. All other programming languages for the PLC can be converted to IL. The function $X = A \text{ and } (B \text{ or } C)$ written as IL: (Jack, H.)

```
LD A
LD B
LD C
ORB
AND B
ST X
```

2.5.3 Structured Text

Structured Text (ST) is a high level language, such as Basic or C, and the programs are built as function blocks. The language also supports the use of iteration loops, conditional execution and functions. The function $X = A \text{ and } (B \text{ or } C)$ will look like this as a ST program: (Jack, H.)

```
FUNCTION Example
  INPUT VAR
    A: BOOL;
    B: BOOL;
    C: BOOL;
  END_VAR
  OUTPUT VAR
    X: BOOL;
  END_VAR
  VAR
    Temp: BOOL;
  END_VAR
  Temp := OR(B,C);
  X := AND(A,Temp);
END_FUNCTION
```

2.5.4 Sequential Function Charts

Sequential Function Charts (SFC) is the most powerful graphical technique for describing the sequential behavior of a PLC program. A SFC organizes a PLC into a set of steps and transitions interconnected by directed links. (Jack, H.)

2.5.5 Function Block Diagrams

Function Blocks are program elements that appear as rectangular blocks. These blocks are “wired” into Function Blocks Diagrams. (Jack, H.)

3. Implementation

In this chapter the implementation of the three objectives of this master thesis is presented. The sequence of work, detailed solutions to selected obstacles and the results are presented.

3.1 Preparations

There is a robot cell at Chalmers called “PPU-lab” (named after the department for Product and Production development). The safety in the PPU-lab has increased after an installation of a safety cage. The safety cage was installed in the spring of 2007. When a good safety cage was in place it was decided that the robots and the Flexlink pallet system should be fixed to the floor followed by a calibration of the cell in order to complete the setup of the robot system. The world origin coordinate for the system was placed in a spike in the Flexlink pallet system, indicated by the white arrow in Figure 3.1. The spike positions a pallet moving on the pallet system so the robot knows the position of the pallet.

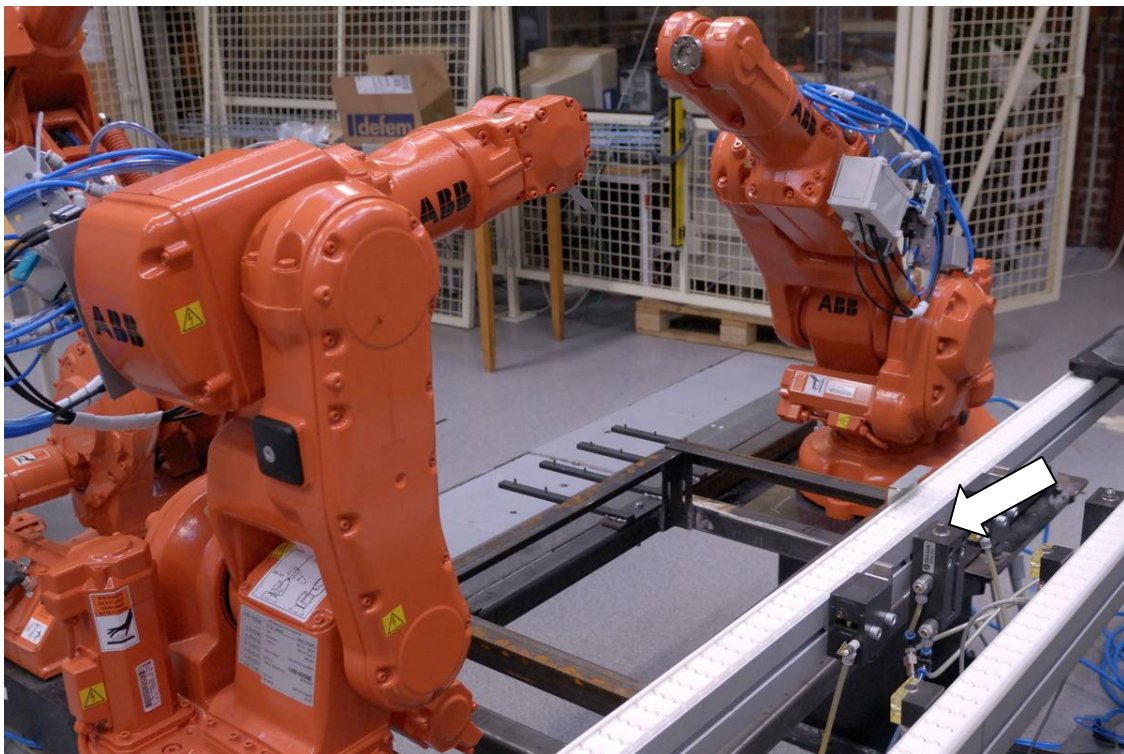


Figure 3.1: The robot cell in the PPU-lab and the world coordinate

3.1.1 Mounting the robots to the floor

Initially the Flexlink system was placed in the correct position based on openings in the safety cage. Then it was time to give the two robots the correct position. The robots have to be able to reach each other, the pallet on the Flexlink system and one robot also has to reach the rotating table.

The robots are mounted onto pillar stands. At the bottom of these stands there are three mounting holes on each side. When the maximum carrying weight of the robots was known, the maximum load on the mounting screws was calculated and a safety factor was added. Each robot was mounted to the floor with four Hilti HVU M10x90 screws, with adhesive anchor.

3.2 Implementation of BoxSweeper

When this master thesis started it showed that the BoxSweeper add-in was not as stable as expected. The first weeks were therefore spent trying to get BoxSweeper to run. Pretty much every step the user has to go through to get the final robot programs had some sort of error. For example the generation of boxes, how the via-points were selected and the data input to Supremica, all had some errors. After some help from ABB and Avenir Kobetski (PhD at the department of Signals and Systems at Chalmers) BoxSweeper became more stable and the simulation speed was slightly improved.

One error that was discovered was that the boxes were only generated for the part which was mounted to the robot. For example if a robot had a tool and this tool holds a sheet the boxes were only generated for the robot and tool, but not for the sheet. The problem was solved by doing a control if a part has a "child-link". In RobotStudio objects are attached to each other as "child" and "parent", if a tool is connected to a robot then the robot is the parent and the tool is the child. If an object is attached to the tool, the tool is the parent to that object. Child control was changed to perform on every part until a part no longer had a child-link, which resulted in a correct generation process for boxes.

Before the generation of boxes in BoxSweeper can begin, the paths have to be defined. This can be done manually or automatically by BoxSweeper. A technical limitation in BoxSweeper is that only two targets can be used in each path. BoxSweeper does this automatically but if the paths are created manually it is important to remember this limitation. It is a good idea to simulate all targets and paths when they are defined. There are two reasons for this: check that RobotStudio executes the paths in the same order as planned and to define a Main procedure to get a robot program with the right syntax for the real system.

When BoxSweeper finally started to operate and produce rapid code to the real system it showed that the created robot system in RobotStudio was not a copy of the real system, the base frame of the robots was not the same in RobotStudio as in the real system. The consequence was that the robots reported kinematics errors because they could not reach the targets defined offline in RobotStudio. The correct operating procedure of how to create a backup of the real system and how to use it in RobotStudio are explained in Appendix E chapters "3. How to take a backup on a real system" and "7. How to create a new system in RobotStudio".

A system based on a back-up was created and BoxSweeper successfully generated code for the system. This code was loaded into the real controller. During the load of the code, the controller discovered that the variable name *supervisorRequest* was too long. The robot controller only accepted variable name with 16 tokens. Therefore the automatically generated code was modified so the variable was renamed to *supRequest*.

When BoxSweeper runs in RobotStudio a task for the supervisor is automatically created but in the real controller a task has to be created manually. How to define a new task in the real controller is explained in Appendix E chapter "5. How to create a task in the real system". When a task had been created and the variable *supervisorRequest* had been renamed it was no problem to run the code generated by BoxSweeper on the real controller.

After all the problems and errors had been fixed and BoxSweeper successfully created code, the code was executed in the real robot system. Two robot programs with different targets and paths have been validated. The result was good, the robots acted as expected and there were no difference between the offline simulation and the real system. A video of the system has been recorded. Contact the department of Signals and Systems at Chalmers University of Technology for further details.

3.3 Simulation of Bachelor Thesis project

When work began with the simulation for the Bachelor Thesis a few specifications were set for the final simulation:

- Three robots, two ABB IRB140 (robot 1 and 2) and one ABB IRB1400 (robot 3), were to operate simultaneous in a common cell. The IRB1400 was to be attached to an external track thus allowing it to move along the track.
- The product to assemble should contain 3-4 pieces and should be assembled with pop rivets.

The specification resulted in a model shown in Figure 3.2.

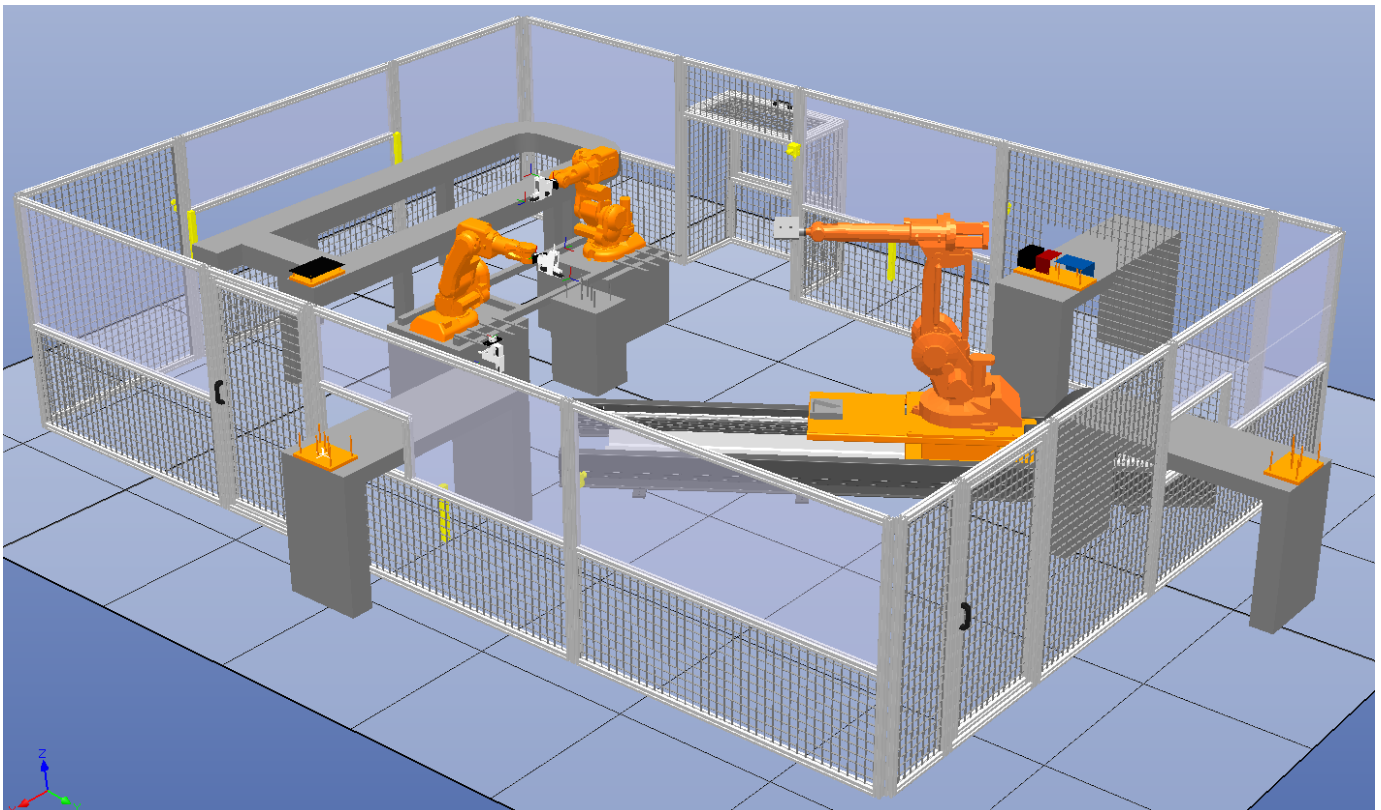


Figure 3.2: The final simulation model for the Bachelor Thesis

The cell has one input, one output and one control conveyor. The control conveyor is used to check the quality of the produced product. When the generation of concept started for the different parts, the idea was that it only should be possible to mount the parts in one way on the conveyor to prevent an operator to rotate a part and place it wrong or place a part on another place. Because of this the parts all have different shapes and the locating holes are asymmetrical. The parts are shown in Figure 3.3. A video of the system has been recorded. Contact the department of Signals and Systems at Chalmers University of Technology for further details.

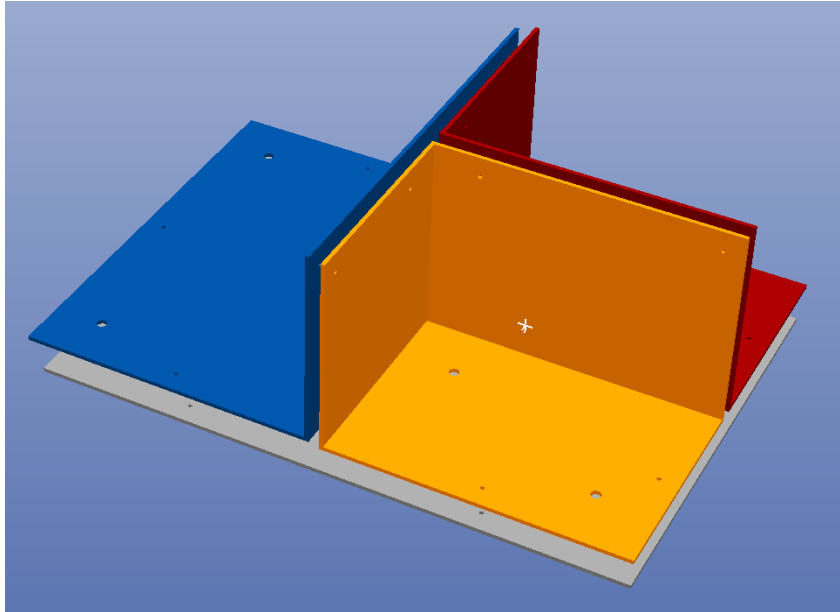


Figure 3.3: The three smaller parts positioned on top of the bottom plate

Three smaller parts are taken into the cell by the input conveyor. When the parts are in the cell the IRB 1400 robot picks up the parts with a three sided tool and places the parts in a fixture for assembly. The bottom plate is brought into the cell by the Flexlink track and is positioned in the fixture by robot 2, after the three smaller parts are in place. When the parts are in position the three robots perform pop riveting simultaneously.

When the parts have been pop riveted into one piece, robot 3 picks up the part and robot 1 and 2 perform the final pop rivet. When the pop riveting has been performed the product is delivered to the output conveyor. Every third product is delivered to the control conveyor so that a quality control of the pop rivet can be performed.

The final simulation of the robot cell contains a robot attached to an external track. The input and output conveyers are created as mechanisms. Macros are used in the simulation that controls the attachments between some of the objects in the simulation.

3.4 Development of a PLC supervisor

In part one of this master thesis the supervisor is created as a new task in the robot controller. This requires that an IRC5 controller with a multimove system is used. But multimove systems are not that commonly used in industrial robot cells. The use of PLCs is more common to control the production and therefore the third part of this master thesis is about the development of the supervisor to be able to run it on a PLC. In the PPU-lab there is a Siemens PLC, model 313, and it uses software called Siemens Simatic Step 7. A Profibus communication between the PLC and the robot controller is being used and in the robot controller a Profibus gateway is connected to the controller through device net.

To separate the different supervisors a new name for the special BoxSweeper had to be created. Since the supervisor shall supervise from a PLC the new BoxSweeper function was named "BoxSweeper PLC".

First a study in the different programming language for PLC was performed. IL, LD and ST were investigated and the ST (Structure Text) language was chosen because of its similarities to RAPID code. Siemens use a programming language named SCL which is very much alike ST but some syntax are changed to match the Siemens system.

When the programming language was decided the work begun with creating a deeper understanding for the supervisor. The global variables, which are being used to communicate with the supervisor, were detected. The robot programs use these variables to declare which zone they want to enter and at what time. When the supervisor is executed in the robot controller the variables are global because every task has to be able to write and read to these variables. For a detailed explanation of the communication process and the global variables in BoxSweeper see chapter 2.4.

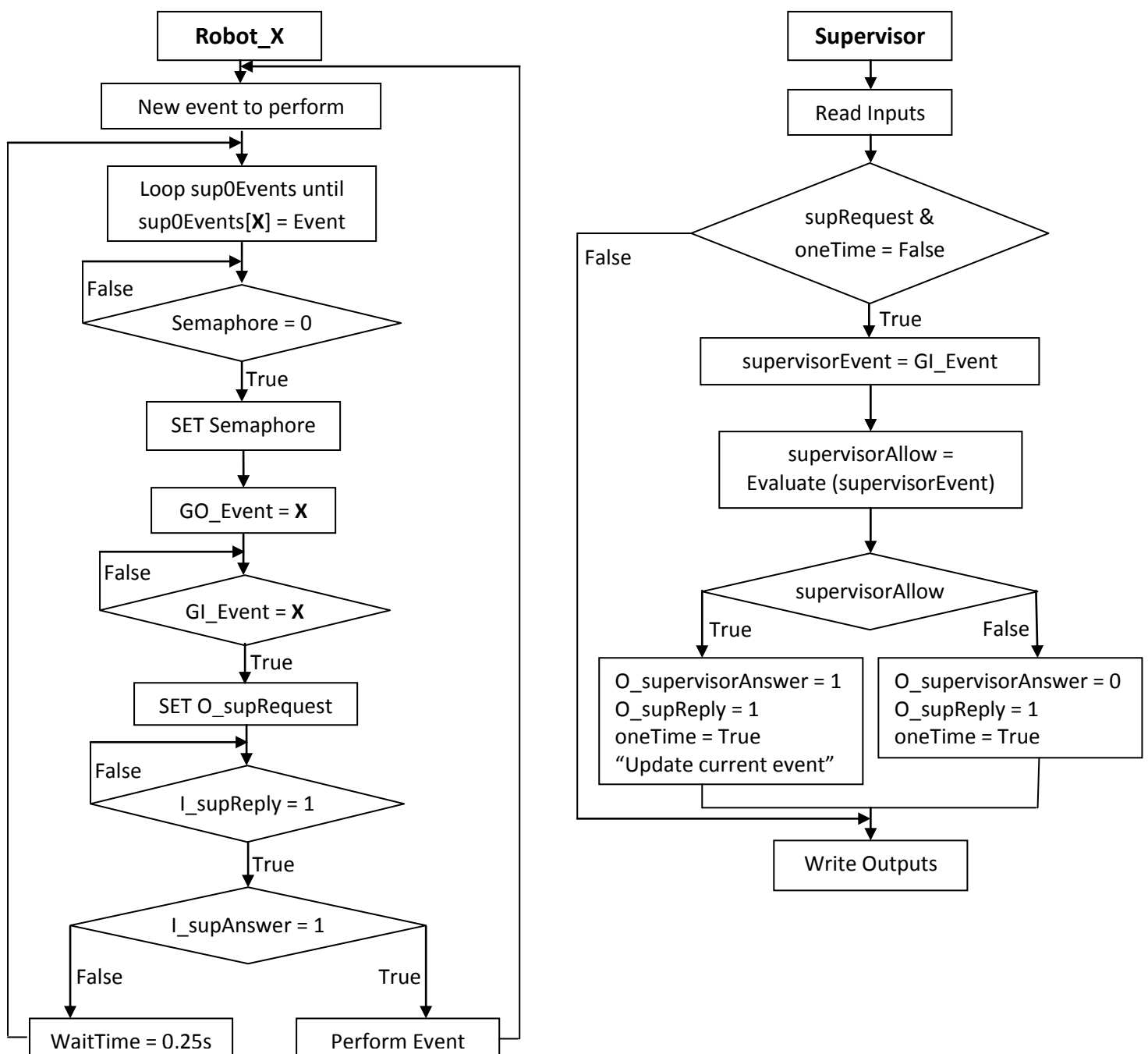


Figure 3.4: The BoxSweeper PLC procedures for a Robot and Supervisor

When a robot wants to enter or leave a zone it sends a string with an event to the supervisor, which has an array with all events in the system called *sup0Events*. The string is translated to a number and the supervisor calculates if the event is allowed.

The biggest problem when moving the supervisor task from the robot controller is that the communication processes need to be altered. The global variables cannot be shared over Profibus. This is because Profibus cannot transfer strings. Since the events have names such as “RA_u_0_m1_1” or “RB_b_0_m1_1” a translator would have had to be created in the robot program that translated the strings to bits and then another translator would be needed in the PLC to retranslate the bits back into a string.

This problem was analyzed and it was found that the first operation that the supervisor task in the robot controller does, when it receives a *supervisorRequest*, is to translate the Event (string) into an integer. The code for the translation was moved from the supervisor to the robot tasks instead, also a group signal called *GO_Event* was created. When the robot wants to perform an event the event is translated into an integer (0-255, 255 is the largest value that can be sent in one byte) in the robot task before it is sent to the PLC as *GO_Event*.

When the initial test of BoxSweeper PLC was performed a problem with the communication process was detected. The time to write the value of *GO_Event* to Profibus varied which had the result that the supervisor sometimes received corrupt events. It is believed that the PLCs scan cycle was so fast that it started working on the event data before the entire event, all the bits, had been written to Profibus. This resulted in incorrect changes of the current state in the supervisor and thus the supervisor only allowed events that the robots currently did not want to perform.

The problem was solved with a group signal called *GI_Event*. *GI_Event* is set in the PLC and it is a mirrored signal of the data on *GO_Event*. Before the robot task set *O_supRequest* the data on *GI_Event* has to have the same value as *GO_Event*. This ensures that the correct event has been received by the PLC.

When a working solution for the BoxSweeper PLC supervisor had been found the solution had to be implemented in the source code for the BoxSweeper add-in so that the code could be generated automatically. The changes that were made in the BoxSweeper procedures for the robots and the supervisor can be seen in Figure 3.4. A video of the system has been recorded. Contact the department of Signals and Systems at Chalmers University of Technology for further details.

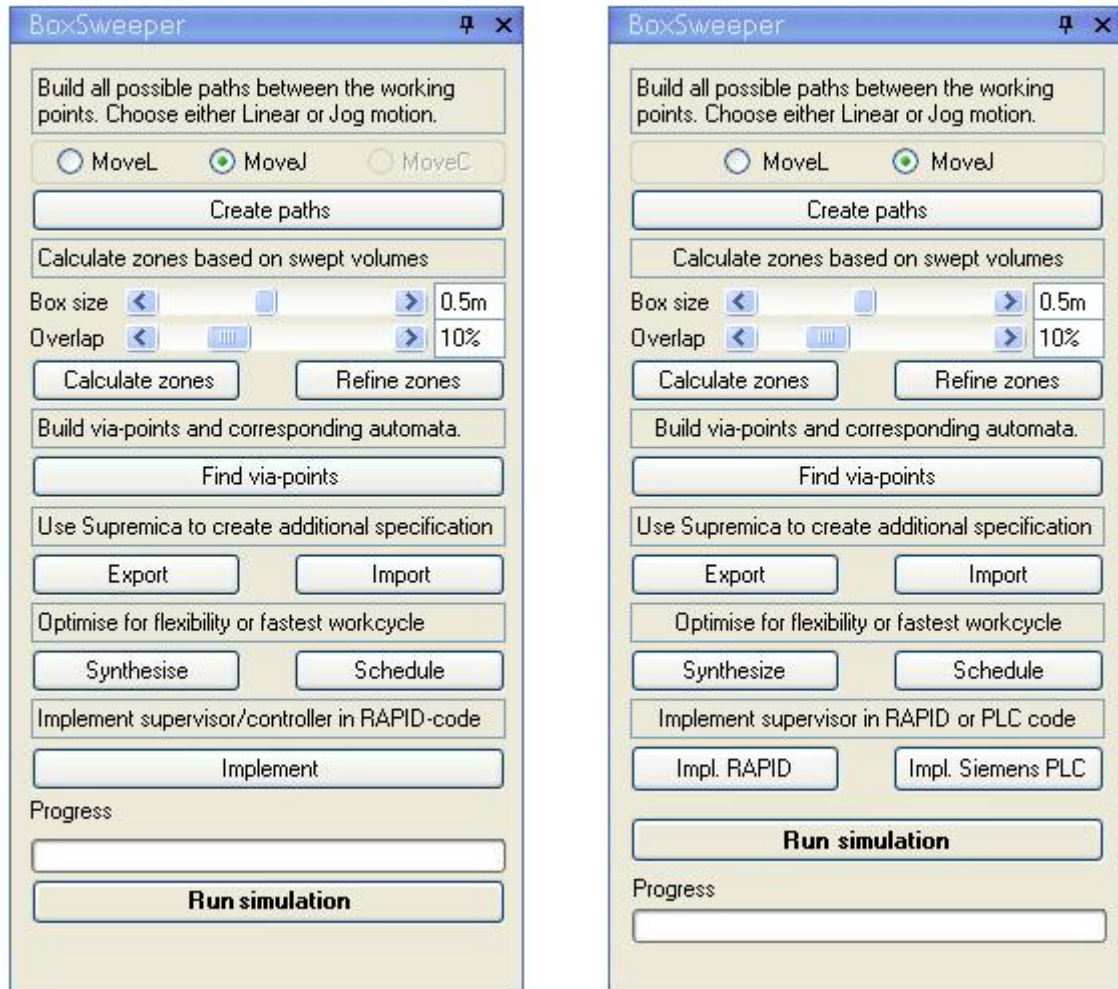


Figure 3.5: A comparison of the old BoxSweeper add-in, to the left, and the new one.

After the via-points have been placed and the supervisor has been created (Synthesise or Schedule) the user now has to choose if the code shall generate a robot task supervisor (Impl. RAPID) or a PLC supervisor (Impl. Siemens PLC). Figure 3.5 shows the new layout for the BoxSweeper add-in.

If the user chooses to create a PLC supervisor the program starts to generate the code. Before the supervisor code is written the user is asked where to save the code for the supervisor. The code will be saved as a .scl file, which is the file system for Siemens SCL files. The user then has to take this file, and the robot programs, and load them into Siemens Simatic Step 7 and the robot controller. The .scl file is then compiled in a special SCL-compiler which translates the generated code to PLC code.

During the tests of an automatically generated PLC Supervisor a problem with the program compilation was detected. The compiler can only address a certain amount of temporary data for each operation. When larger arrays were compiled the amount data was larger than the available temporary memory, this resulted in compiler errors.

The problem was solved by moving all the arrays into a separate data block and compiling each element in the array one by one.

3.4.1 Signal configuration for BoxSweeper PLC

In order to use the code generated by BoxSweeper PLC a few signals, data blocks, function blocks and variables must be defined in Siemens Simatic Manager, see Appendix F.

The signals to communicate with the robot controller are defined in two data blocks. One data block is used for the input signals and one is used for the output signals. The addresses for the signals that are received, see data block 71 in Table 3.1, starts at address 4.0 and the data type and initial value are specified in the data block. To connect the input signals of the PLC with the output signals of the robot controller, the unit mapping of the robot signal has to be the same as the bit number it is specified on in the data block. For example, signal *I_supRequest* is specified on address 5.0, see Table 3.1, and according to Table 3.3 the PLC starts to read on address 4.0 and reads 2 bytes. Therefore the signal *I_supRequest* has to be declared, with unit mapping, as number 9 in the controller since *GI_Event* is a byte (8 bits).

The output signals starts at address 3.0, see Table 3.2, and 2 bytes are being sent, see Table 3.4. To specify the right unit mapping for the signal *GO_Event*, the signal in the robot controller has to be specified as number 0-7 because *GO_Event* is a group signal and the signal has the first bits that are being sent on the Profibus. Another example is the signal *O_supReply*, which has unit mapping number 9.

Adress	Name	Type	Initial value	Comment
0.0		STRUCT		
+0.0	STATUS	WORD	W#16#0	
+2.0	NDR_ERROR_DP	WORD	W#16#0	
+4.0	GI_Event	BYTE	W#16#0	The event to be evaluated represented as a number
+5.0	I_supRequest	BOOL	FALSE	1 = A call from the robot to start an evaluation
+5.1	I_initialize	BOOL	FALSE	1 = Perform initialization
+5.2	I_Semaphore	BOOL	FALSE	1 = A robot has exclusivity to use the supervisor

Table 3.1: Data block 71 - Definition of the input signals for BoxSweeper PLC

Adress	Name	Type	Initial value	Comment
0.0		STRUCT		
+0.0	STATUS	WORD	W#16#0	
+2.0	DONE_ERROR	BYTE	W#16#0	
+3.0	GO_Event	BYTE	W#16#0	
+4.0	O_supervisorAnswer	BOOL	FALSE	0 = The event is not allowed 1 = The event is allowed
+4.1	O_supReply	BOOL	FALSE	1 = The evaluation process has been completed

Table 3.2: Data block 70 - How the output signals are defined BoxSweeper PLC.

Network 1: Receives data from the Profibus	
	CALL "DP_RECV"
	CPLADDR := W#16#130
	RECV := P#DB71.DBX4.0 BYTE 2
	NDR := DB71.DBX2.0
	ERROR := DB71.DBX2.1
	STATUS := "TaemotPro".STATUS
	DPSTATUS := DB71.DBB3
L	"DB_RECV".GI_Event
T	"GI_Event"
L	DB71.DBB 5
T	MB 111

Network 3: Sends data to the Profibus	
L	"GI_Event"
T	"GO_Event"
L	MB 112
T	DB70.DBB 4
L	"GO_Event"
T	"DB_SEND".GO_Event
	CALL "DP_SEND"
	CPLADDR := W#16#130
	SEND := P#DB70.DBX3.0 BYTE 2
	DONE := DB41.DBX8.0
	ERROR := DB41.DBX8.1
	STATUS := "SkickaPro".STATUS

Table 3.3: Network 1 in the communication block Table 3.4: Network 3 in the communication block

4. Results

This chapter presents the results for the objectives that were set for this master thesis.

4.1 Implementation of BoxSweeper

The automatically generated supervisor works as intended and it has been verified that BoxSweeper generates collision free robot programs. When the BoxSweeper add-in worked as intended, the only error that appeared was that the variable *supervisorRequest* that had too many characters. When the variable was renamed to *supRequest* the program was executed successfully. Two robot programs with different targets and paths have been validated.

4.2 Simulation of Bachelor Thesis project

A simulation for the Bachelor Thesis was successfully generated in Robot Studio. The final solution of the robot cell contains a robot attached to an external track, mechanisms that control a number of conveyors and macros that controls the attachments between some of the objects in the simulation.

4.3 Development of a PLC supervisor

The BoxSweeper add-in has been developed and now has the ability to generate a supervisor which executes in a Siemens PLC. When using the BoxSweeper add-in the user has the ability to choose which kind of supervisor to generate, BoxSweeper (operates in a robot controller) or BoxSweeper PLC (operates in a PLC). As well as BoxSweeper, verified in part 1, BoxSweeper PLC generates supervisor that guaranties collision free robot programs. However the executions of BoxSweeper PLC programs are not as fast as BoxSweeper programs.

5. Discussion

Collision free programs generated by BoxSweeper had only been verified in RobotStudio before the work began on this master thesis. When the BoxSweeper add-in finally started to generate code the program could be verified on a real robot system. The generated programs have worked as intended, the behavior of the robots is the same in both RobotStudio and in the real system.

Unfortunately the instability of BoxSweeper has caused a lot of frustration. Many of the tests that were designed could not be generated because of different errors. It has been noticed that different RobotStudio systems, with the same setup of robots, targets and paths, does not always result in the same outcome. On certain systems BoxSweeper works as intended almost every time while other systems tend to freeze during different stages of BoxSweeper.

The problems with BoxSweeper had the effect that only two systems were successfully generated to be used for tests in the real robot system. The total time to create a system, targets, paths and run the BoxSweeper add-in on this system can take almost an hour. If BoxSweeper crashes during any phase, RobotStudio must be restarted and sometimes the entire system must be built from scratch. BoxSweepers stability must improve substantially to allow further work and testing, especially if BoxSweeper shall be released as a commercial product. During the work of this master thesis the stability has improved, but there is still a lot of work to do.

BoxSweeper has successfully been developed with the new ability to create a PLC supervisor. The PLC supervisor has the same structure as the supervisor for the controller. The supervisor works as intended but due to a believed communication delay, between the robot controller and the PLC, the time between each evaluation is longer. This causes the robot programs to make longer stops when the robot enters or leaves a zone. The reason for the communication delay is still unknown, further work is needed to determine the cause of this delay.

The group signal GO_Event is defined as an 8 bit signal. That allows a maximum of 256 events to be sent on the Profibus, hence limiting the number of events that can be used in the robot system. The tests that have been used for validating the BoxSweeper PLC had a maximum of 59 events. By creating a larger group signal a larger amount of events can be sent.

Structured Text was chosen because of the similarities to RAPID. The supervisor task demands that mathematical operations can be performed during the event evaluation. The mathematical operations could have been created in other PLC languages, but the code would have been much more complicated. If future demands call for BoxSweeper to generate code for other PLC manufactures, the generation process can easily be modified to other manufacturing standards.

6. Future work

This master thesis has resulted in a few suggestions for future work:

- Stability - The stability for BoxSweeper must improve in order to allow further development.
- Complicated systems - Include external track and mechanisms in the zone generation process to allow BoxSweeper to be used on more complicated systems.
- PLC communication – Examine the cause of the delay when entering and leaving zones.
- HMI – Improve the human machine interface of BoxSweeper to minimize the errors that can be caused by the operator. The user is not given any feedback on which steps that has been performed since all buttons are available at all stages. For example, disable the “Run simulation” button after a PLC supervisor has been generated.
- Speed – Insert an option that allows the user to set the speed for the movements in the robot programs. Today every program is generated with a standard speed for robot movements.
- Examine – Some systems that are created refuses to work with BoxSweeper while others work as intended. The reason for this must be examined in order to increase stability.
- Paths – Today BoxSweeper only allows paths with two targets. Complicated systems will result in a large number of paths.
- Upgrade – Install a new version of RobotWare, that supports 32 tokens in the variable name, in the controller in the PPU-lab.

References

Printed sources

ABB (2007). *Application Manual Engineering Tools*. Document ID: 3HAC 020434-001
Revision: D

ABB (2007). *Operation Manual IRC5 With FlexPendant*, Document ID: 3HAC 16590-1, Revision H

ABB (2007). *Operation Manual RobotStudio 5.09*, Document ID: 3HAC 028932-001, Revision D

Bolmsjö, Gunnar S. (2006). *Industriell Robotteknik*. Third edition. Studentlitteratur, Lund. ISBN 91-44-00848-1

Fabian, Martin (2006). *Industrial Automation: Lecture Notes (SSY 065)*. Chalmers Automation. Department for Signals and Systems. Chalmers University of Technology. ISSN 1403-266X

Fallström, A. and Haziraj, A. (2006). *Development and Evaluation of Automatic Robot Interlocking Policies using Supervisory Control*. Chalmers University of Technology. Report No. EX078/2006

Flordal, H., Fabian, M., Åkesson, K., Spensieri, D. (2007) *Automatic model generation and PLC-code implementation for interlocking policies in industrial robot cells*. Control Engineering Practice 15, pages 1416–1426

Grahn, R (1995). *Engineering mechanics*. New York: Prentice Hall. ISBN 992-071673-1

Weinemyr, T., Zekaj, V. (2007). *RAPID-Code Implementation of Supervisory Control Functions for Multi Agent Robot Cells*. Chalmers University of Technology. Report No. EX080/2007

Internet sources

ABB (2008). Product Guide IRC5

<http://www.abb.com/product/seitp327/f0cec80774b0b3c9c1256fda00409c2c.aspx?productLanguage=us&country=00> (Accessed 18 Mars 2008)

Jack, H. (2007). *Automating Manufacturing Systems with PLCs*. V 5.0

<http://www.eod.gvsu.edu/~jackh/books/plcs/> (Accessed 18 Mars 2008)

Appendix A - CodeWriter.cs

These are the additions that have been made to the CodeWriter.cs file during this master thesis.

```
public bool writeLinePLC(String s)
{
    // Trim the string (remove empty spaces at the beginning and end)
    s = s.TrimStart().TrimEnd();

    // Check if file is open
    if (!fileIsOpen())
        return false;

    // Decrease indentation? Look for keywords...
    if (s.StartsWith("END") || s.StartsWith("ELSE"))
        indentation--;

    // Write indentation
    for (int i = 0; i < indentation; i++)
        rapidFile.Write(INDENTSTRING);

    // Write the line
    rapidFile.WriteLine(s);
    // Increase indentation? Look for keywords...
    if ((s.StartsWith("VAR") || s.StartsWith("FUNC") ||
s.StartsWith("STRUCT") || s.StartsWith("DATA") || s.StartsWith("BEGIN") ||
s.StartsWith("IF") || s.StartsWith("ELSE") || s.StartsWith("WHILE")))
        indentation++;
    return true;
}
/// <summary>
/// Writes a comment to the file.
/// </summary>
/// <param name="s">The comment</param>
/// <returns>False if there is no open file, True otherwise.</returns>
public bool writeComment(String s)
{
    return writeLine(COMMENTSTRING + s);
}

/// <summary>
/// Writes a blank line.
/// </summary>
/// <returns>False if there is no open file, True otherwise.</returns>
public bool blankLine()
{
    return writeLine("");
}

private const String COMMENTSTRING = "!! ";
private const String INDENTSTRING = " "; // "\t";
}
}
```

Appendix B – RapidCodeTemplatesPLC

These are the additions that have been made to the RapidCodeTemplatesPLC.cs file during this master thesis.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
using System.Windows.Forms;
using ABB.Robotics.RobotStudio;
using ABB.Robotics.RobotStudio.Stations;
using ABB.Robotics.Controllers;
using ABB.Robotics.Controllers.RapidDomain;
using System.IO;
using org.supremica.automata;

namespace BoxSweeper
{
    public static class RapidCodeTemplatesPLC
    {
        /// <summary>
        /// The maximum value of the Num type in rapid
        /// </summary>
        public const int maxNum = 300000;

        #region Robot code, for the Robot tasks
        /// <summary>
        /// For all robots, implements the robot RAPID code based on the plant
        /// model of the robot.
        ///
        /// This method differs from WriteRobotRapidCode since the events are
        /// represented as numbers when they are sent to the PLC.
        /// </summary>
        /// <param name="path">The path of the directory, code will be saved in
        /// "path/modelName/" </param>
        public static void WriteRobotRapidCodePLC(string path, Hashtable
simulationData)
        {
            //Now the code is generated from the lists of events generated during
            simulation instead
            //moveL changed to triggL, moveJ to triggJ and moveC to triggC
            string MOTIONTYPE;
            if (BoxSweeper.motionType == MotionType.Linear)
                MOTIONTYPE = "L";
            else if (BoxSweeper.motionType == MotionType.Joint)
                MOTIONTYPE = "J";
            else if (BoxSweeper.motionType == MotionType.Circular)
                MOTIONTYPE = "C";
            else
            {
                LoggHandler.Logg("No motiontype chosen.", LogMessageSeverity.Error);
                return;
            }

            //Request Mastership
            Mastership mastership = BoxSweeper.controller.GetMastershipRapid();
            if (mastership != null)
            {
                //Add all the tasks in an array
                Task[] task =
                BoxSweeper.controller.GetController().Rapid.GetTasks();
                int numberOfTasks = task.Length;

                //One task at the time...
                for (int i = 0; i <= numberOfTasks - 1; i++)
```

```

        {
            //This is for avoiding the supervisor task, will have to be
remade when we are able to create a task directly through the sdk
            if
(!task[i].Name.Trim().ToUpper().Equals(StringConstants.SUPERVISORTASK_NAME.ToUpper(
)))
        {
            ProgressHandler.PerformStep();
            Station stn =
ABB.Robotics.RobotStudio.Project.ActiveProject as Station;
            RsTask findNameTask = stn.Irc5Controllers[0].Tasks[i];
            Mechanism robot = findNameTask.Mechanism;
            string modelName = robot.Name;

            //Defines the module that is going to be used and selects
the place to save
            string modulePath = path + modelName + "\\";
            Module robotModule =
BoxSweeper.controller.GetController().Rapid.GetTask(task[i].Name).GetModule("Module
1");

            try
            {
                if (!System.IO.Directory.Exists(path))
                {
                    System.IO.Directory.CreateDirectory(path);
                }
                if (!System.IO.Directory.Exists(modulePath))
                {
                    System.IO.Directory.CreateDirectory(modulePath);
                }

                robotModule.SaveToFile(modulePath);
            }
            catch (Exception ex)
            {
                LoggHandler.Logg("Error when creating files and paths
for Rapid code: " + ex.Message, LogMessageSeverity.Error);
                return;
            }

            try
            {
                // Read data from old file (generated from
synchronization from RS, includes important declarations)
                StreamReader reader = new StreamReader(modulePath +
robotModule.Name + ".mod");

                // Write to new file
                //StreamWriter writer = new StreamWriter(path +
"temp.mod"); // Use ToRapid instead!
                CodeWriter writer = new CodeWriter(modulePath +
modelName + ".mod");

                writer.open();

                // Copy the first lines, with the important target
definitions, to the new file
                bool lookingForBreak = false;
                while (!reader.EndOfStream)
                {
                    // Read line from input file
                    string line = reader.ReadLine();

                    // When the line contains PERS or CONST we set a
flagg and start looking for an empty line
                    if (line.Contains("PERS") ||
line.Contains("CONST"))
                {

```

```

        lookingForBreak = true;
    }

    // When we find the empty line we break the reading
operations
    if (lookingForBreak == true &&
(line.Trim().Equals("") || line.Contains("PROC")))
    {
        break;
    }

    // Write line to output file
    writer.WriteLine(line.Trim());
}
// Now we don't need the read-file any more! The rest
is all new material...
reader.Close();

//When a new state is found it is added to the to do
list. Afterwards we get the todoList and
//start from that state and find our way to the next
states!

IList todoList = new ArrayList();

//The list of states we already found our way out of!
IList doneList = new ArrayList();

//Construction of the robotmotion program
//Now we continue and add the different variables that
are needed by our program
writer.WriteLine("");
writer.WriteComment("Variables for setting speed and
accuracy of the robot when moving between targets");
writer.WriteLine("VAR speeddata speed := " +
StringConstants.SIMULATION_SPEED + ";");

writer.BlankLine();
writer.WriteComment("The below code was automatically
generated by BoxSweeper PLC on " + DateTime.Now + ".");
writer.WriteComment("Edit with extreme care.");
writer.BlankLine();

// Get simulationdata
List<List<TargetInfo>> targetInfoLists =
(List<List<TargetInfo>>)simulationData[robot.Name];
writer.WriteComment("These variables are used for
interrupts that are called upon while the robot is moving.");
writer.WriteComment("It is these interrupts that do the
booking/unbooking of zones, while moving. For each");
writer.WriteComment("interrupt there is one trigger
variable and one interrupt name. Since RobotStudio only");
writer.WriteComment("has support for names up to 32
characters, they each get a short 'alias' name.");

// Generate short alias names for every trigger and
TRAP-routine
// since RobotStudio only has support for routine names
of up to 32 characters it
// is necessary to have short alias names for these
events
// this also solves previous issues with ambiguous
variables
TrapNames interruptNames = new TrapNames();
foreach (List<TargetInfo> targetInfoList in
targetInfoLists)
{
    for (int m = 0; m <= targetInfoList.Count - 2; m++)
    {

```



```

        string eventName =
RapidCodeTemplates.EventGenerator(robot, targetInfoList[m], targetInfoList[m + 1]);
        if (eventName != null)
        {
            interruptNames.addName(eventName);
            string alias =
interruptNames.getAlias(eventName);
        }
    }
    // Since we often only want to treat each alias ONCE,
even though it may appear several times in the simulation data, we need to
    // remember the ones we've treated;
    List<string> treatedAliases = new List<string>();

    // This block is for declaring all the variables used
for interrupts and triggering
    // Ie the first part of the RAPID-file
    treatedAliases.Clear();
    foreach (List<TargetInfo> targetInfoList in
targetInfoLists)
    {
        for (int m = 0; m <= targetInfoList.Count - 2; m++)
        {
            string eventName =
RapidCodeTemplates.EventGenerator(robot, targetInfoList[m], targetInfoList[m + 1]);
            // Unless it is IN followed by OUT or IN
followed by no zone we will make an interrupt
            if (!((targetInfoList[m].isEntering() &&
targetInfoList[m + 1].isExiting()) || (targetInfoList[m].isEntering() &&
targetInfoList[m + 1].isEmpty()) ||
                (eventName == null)))
            {
                string alias =
interruptNames.getAlias(eventName);

                // Treat this one if not already treated!
                if (!treatedAliases.Contains(alias))
                {
                    writer.WriteLine("VAR triggdata trigg_"
+ alias + ";" + " !! Alias trigg_" + alias + " is associated with event(s) " +
eventName);
                    writer.WriteLine("VAR intnum int_" +
alias + ";");
                    treatedAliases.Add(alias);
                }
            }
            // if it is IN followed by OUT do nothing
        }
    }
    writer.blankLine();
    writer.writeComment("Variabels and matrix for
communication with the PLC");
    if (i == 0)
    {
        writer.WriteLine("VAR bool Initialize := TRUE;");
    }
    writer.WriteLine("VAR num eventNr := -1;");

    //Creation of the Automata Matrix copied from
BoxSweeper.cs
    AutomatonMatrix[] matrix;
    java.util.Iterator supervisorIterator;
    int counter = 0;
    //iterate over all supervisors
    supervisorIterator =
BoxSweeper.supremicaFacade.GetSupervisorModel().getAutomata().iterator();

```

```

        if (supervisorIterator == null)
LoggHandler.Logg("SupervisorIterator is null!");
        //Create the automatonmatrixarray
        matrix = new
AutomatonMatrix[BoxSweeper.supremicaFacade.GetSupervisorModel().nbrOfAutomata()];
        //Loop for all supervisors
        while (supervisorIterator.hasNext())
        {
            ProgressHandler.PerformStep("Building Matrix");
            //Create the automatonmatrix
            matrix[counter] = new
AutomatonMatrix((Automaton)supervisorIterator.next(), "sup" + counter);
            ProgressHandler.PerformStep();
            ProgressHandler.PerformStep();
            counter++;
        }
        RapidCodeTemplatesPLC.SupervisorEventPLC(matrix,
writer);

        //Start writing the main procedure
        writer.blankLine();
        writer.writeLine("PROC main()");
        writer.writeComment("The TriggInt commands are used to
define the triggers to go off when the robot is leaving");
        writer.writeComment("one target and starting its move
to the next target.");
        writer.writeComment("The CONNECT statements defines
which interrupt to call upon when the triggers go off.");

        // One trigger definition and one CONNECT for each
trigger/interrupt pair
        // These are called upon while moving using TriggJ
instructions.
        // 0 \Start means that the interrupt is called upon the
instant that the robot
        // is leaving a target, and performed while moving to
the next
        treatedAliases.Clear();
        foreach (List<TargetInfo> targetInfoList in
targetInfoLists)
        {
            //This for-loop compares two rows on the simulation
list.
            for (int m = 0; m <= targetInfoList.Count - 2; m++)
            {
                string eventName =
RapidCodeTemplates.EventGenerator(robot, targetInfoList[m], targetInfoList[m + 1]);
                // Unless it is IN followed by OUT or IN
followed by no zone we will make an interrupt
                if (!(targetInfoList[m].isEntering() &&
targetInfoList[m + 1].isExiting()) || (targetInfoList[m].isEntering() &&
targetInfoList[m + 1].isEmpty()) ||
                    (eventName == null))
                {
                    string alias =
interruptNames.getAlias(eventName);

                    // Treat this one if not already treated!
                    if (!treatedAliases.Contains(alias))
                    {
                        writer.writeLine("CONNECT int_" + alias
+ " WITH trap_" + alias + ";");
                        writer.writeLine("TriggInt trigg_" +
alias + ", 0 \\Start, int_" + alias + ";");
                        treatedAliases.Add(alias);
                    }
                }
            }
            // else do nothing

```

```

    }
}

//Robot 1 is responsible for the initialization of the
PLC
if (i == 0)
{
    writer.blankLine();
    writer.writeComment("Signals the PLC-
initialization");

    writer.writeLine("IF initialize THEN");
    writer.writeLine("SET O_initialize;");
    writer.writeLine("WaitTime 0.5;");
    writer.writeLine("RESET O_initialize;");
    writer.writeLine("initialize := FALSE;");
    writer.writeLine("ENDIF");
}
//The other robots has to wait for the initialization
if (i > 0)
{
    writer.writeComment("Synchronization to initialize
(rob 1)");

    writer.writeLine("WaitTime 2;");
}
writer.blankLine();

// Sort the lists in groups originating form the same
point
// Compares the list to ALL targets, including VIA-
points,
// these will generate empty pathsFromHereLists
// therefore it is important to break the loop if the
list is empty
RsTargetCollection allTargets = robot.Task.Targets;

//Check to see which list originate from the same
point, and group these together
foreach (RsTarget target in allTargets)
{
    List<List<TargetInfo>> pathsFromHereList = new
List<List<TargetInfo>>();
    foreach (List<TargetInfo> targetInfoList in
targetInfoLists)
    {
        if
(targetInfoList[0].targetName.Equals(target.Name))
        {
            pathsFromHereList.Add(targetInfoList);
        }
    }
    //break the loop if the list is empty to avoid
index out of bounds-errors
    if (pathsFromHereList.Count == 0)
    {
        // the list is empty, break the loop
        continue;
    }

    //Make all the movement instructions from these
lists

    //It is important that a target and a label are
separated and does not have the same name
    //We print out LABEL where we are currently: e g
AT_HOME:
    writer.writeLine("AT_" +
pathsFromHereList[0][0].targetName + ":");

```

```

        foreach (List<TargetInfo> targetInfoList in
pathsFromHereList)
    {
        writer.WriteLine("IF (isEnabled(\"" +
robot.Name + "_g_" + targetInfoList[targetInfoList.Count - 1].targetName + "\"))
THEN");
        //This for-loop compares two rows on the
simulation list.
        // comparing [index] to [index + 1]
        // there are eight different combinations. IN-
IN, IN-OUT, IN-no zone, OUT-OUT, OUT-IN, OUT-no zone, no zone-IN and no zone-OUT
        for (int m = 0; m <= targetInfoList.Count - 2;
m++)
        {
            // Which zone accuracy should we use?
            string zone;
            if ((m + 1) != targetInfoList.Count - 1)
            {
                //not the last one, ie not the target
                zone =
StringConstants.ZONE_ACCURACY_VIAPOINT;
            }
            else
            {
                // fine accuracy
                zone = StringConstants.ZONE_ACCURACY;
            }

            // The movement parameter represents the
speed, trigger, zonedata, toolname and workobjectname. These are needed in the
RAPID program
            if ((targetInfoList[m].isEntering() &&
targetInfoList[m + 1].isExiting()) || (targetInfoList[m].isEntering() &&
targetInfoList[m + 1].isEmpty()) || (targetInfoList[m].isEmpty() &&
targetInfoList[m + 1].isExiting()))
            {
                // IN - OUT or
                // IN - no zone, ie the target or
                // no zone- OUT
                // No interrupt routine, just move
                through to the next point
                writer.WriteLine("move" + MOTIONTYPE +
" " + targetInfoList[m + 1].targetName + ", speed, " + zone + ", " +
robot.Task.ActiveTool.Name + "\\WObj := " + robot.Task.ActiveWorkObject.Name +
";");
            }
            else
            {
                // Call interrupt routine in a trigg
                move
                writer.WriteLine("trigg" + MOTIONTYPE +
" " + targetInfoList[m + 1].targetName + ", speed, trigg_" +
interruptNames.getAlias(RapidCodeTemplates.EventGenerator(robot, targetInfoList[m],
targetInfoList[m + 1])) + ", " + zone + ", " + robot.Task.ActiveTool.Name + "\\WObj
:= " + robot.Task.ActiveWorkObject.Name + ";");
            }
        }

        //the last event in this list, ie the target of
the path
        writer.WriteLine("waitUntilEnabled(\"" +
robot.Name + "_r_" + targetInfoList[targetInfoList.Count - 1].targetName + "\");");
        writer.WriteLine("GOTO AT_" +
targetInfoList[targetInfoList.Count - 1].targetName + ";");
        writer.WriteLine("ENDIF");
    }
    //Return to the label and wait check again to see
where to go next

```

```

        writer.WriteLine("GOTO AT_" + target.Name + ";");
        writer.BlankLine();
    }
    writer.WriteLine("ENDPROC");
    writer.BlankLine();

    //TRAP interrupt routines, these do the
    booking/unbooking while the robot is moving between targets
    interrupts that are called upon by the trigg instructions.");
    writer.WriteLine("These communicate with supervisor
    for booking/unbooking events. The interrupt ends as");
    writer.WriteLine("soon as the supervisor enables the
    event. The main program is then resumed.");
    treatedAliases.Clear();
    foreach (List<TargetInfo> targetInfoList in
targetInfoLists)
    {
        for (int m = 0; m <= targetInfoList.Count - 2; m++)
        {
            string eventName =
RapidCodeTemplates.EventGenerator(robot, targetInfoList[m], targetInfoList[m + 1]);
            string alias =
interruptNames.getAlias(eventName);

            // Treat this one if not already treated!
            if (!treatedAliases.Contains(alias))
            {
                if ((targetInfoList[m].isEntering() &&
targetInfoList[m + 1].isExiting()) || (targetInfoList[m].isEntering() &&
targetInfoList[m + 1].isEmpty()) || (targetInfoList[m].isEmpty() &&
targetInfoList[m + 1].isExiting()))
                {
                    // IN - OUT or IN-no zone or no zone-
                    // no interrupt routine, just move on
                    continue;
                }
                else if (targetInfoList[m].isExiting() &&
targetInfoList[m + 1].isEntering() && !targetInfoList[m].isEmpty() &&
!targetInfoList[m + 1].isEmpty())
                {
                    // OUT followed by IN (and neither the
                    // unbook the current zone AND book the
                    // next one
                    writer.WriteLine("TRAP trap_" + alias);
                    // The name is a combination of two
                    // event names, unbooking AND booking of zones, so there are really TWO events
                    // (slightly magical, I know)
                    writer.WriteLine("waitUntilEnabled(\""
+ RapidCodeTemplates.EventnameGenerator(robot, targetInfoList[m]) + "\");");
                    writer.WriteLine("waitUntilEnabled(\""
+ RapidCodeTemplates.EventnameGenerator(robot, targetInfoList[m + 1]) + "\");");
                    writer.WriteLine("ENDTRAP");
                    writer.BlankLine();
                }
                else
                {
                    // OUT-OUT or OUT-nozone or
                    // IN-IN or nozone-IN
                    writer.WriteLine("TRAP trap_" + alias);
                    writer.WriteLine("waitUntilEnabled(\""
+ eventName + "\");");
                    writer.WriteLine("ENDTRAP");
                    writer.BlankLine();
                }
            }
        }
    }
}

```



```

        writer.WriteLine("RESET O_semaphore;");
        writer.WriteLine("RETURN eventAllowed;");
        writer.WriteLine("ENDFUNC");
    }
    #endregion

    /// <summary>
    /// Writes the function SupervisorEventPLC that creates the "Event" array
in the robot program
    /// </summary>
    #region
    public static bool SupervisorEventPLC(AutomatonMatrix[] a, CodeWriter
writer)
    {
        // Compiles code for initializing the variables and the arrays that
defines the automaton
        StringBuilder tempString;
        System.String[] eventNames;
        if (!writer.fileIsOpen()) return false;
        writer.blankLine();
        for (int counter = 0; counter < a.Length; counter++)
        {
            writer.writeComment("The array of strings containing the name of
the events");
            tempString = new StringBuilder();
            tempString.Append("CONST string " + a[counter].GetId() + " " +
StringConstants.eventsString + "{" + a[counter].GetNrOfCols() + " } := [");

            eventNames = a[counter].GetEventNames();
            for (int i = 0; i < a[counter].GetNrOfCols(); i++)
            {
                tempString.Append("\"" + eventNames[i] + "\"");

                if (i < (a[counter].GetNrOfCols() - 1))
                {
                    tempString.Append(",");
                }
            }

            tempString.Append(";");
            writer.WriteLine(tempString.ToString());
            writer.blankLine();
        }
        return true;
    }
    #endregion

    /// <summary>
    /// Writes the function SupervisorArraysPLC that creates the arrays
sup0Pos, sup0Var and sup0Bounds for the SCL supervisor
    /// </summary>
    #region
    public static bool SupervisorBoundArraysPLC(AutomatonMatrix[] a, CodeWriter
writer)
    {
        StringBuilder tempString;
        ArrayList boundList = new ArrayList();
        boundList.Add(0);
        //Write the boundArray to the file
        tempString = new StringBuilder();
        tempString.Append("sup0Bounds: ARRAY[1.." + a[0].NrOfBounds() + "] OF
INT := [");
        object[] boundArray = boundList.ToArray();
        for (int i = 0; i < boundArray.Length; i++)
        {
            tempString.Append(boundArray[i].ToString());
            if (i < (boundArray.Length) - 1)

```

```

        {
            tempString.Append(",");
        }
    }
    tempString.Append("];");
    writer.WriteLinePLC(tempString.ToString());
    return true;
}
#endregion

/// <summary>
/// Writes the first Function Block for the SCL supervisor
/// </summary>
#region
public static bool SupervisorPLCFB2(AutomatonMatrix[] a, CodeWriter
writer)
{
    writer.WriteLinePLC("FUNCTION_BLOCK FB2");
    writer.WriteLinePLC("VAR_INPUT");
    writer.WriteLinePLC("  supervisorEvent: INT; //supervisorEvent is the
event that the robot want the supervisor to evaluate");
    writer.WriteLinePLC("  supRequest: BOOL; //supRequest is a call to tell
the Supervisor that there is a new event to evaluate");
    writer.WriteLinePLC("  initialize: BOOL; //initialize is a
initialization from Robot 1 that sets the initial state (sup0Current) for the
Supervisor");
    writer.WriteLinePLC("END_VAR");
    writer.WriteLinePLC("VAR_OUTPUT");
    writer.WriteLinePLC("  supervisor_Answer: BOOL; //supervisorAnswer is
the respons from the supervisor wheter the event is allowed or not");
    writer.WriteLinePLC("  O_supReply: BOOL; //O_supReply is a call to tell
the Robot that the evaluation is complete");
    writer.WriteLinePLC("END_VAR");
    writer.WriteLinePLC("VAR");
    writer.WriteLinePLC("  supervisorAllow: BOOL; //supervisorAllow is a
local Boolean that recives the result of the evaluation from the calculation");
    writer.WriteLinePLC("END_VAR");
    writer.WriteLinePLC("BEGIN");
    writer.WriteLinePLC("IF initialize THEN");
    writer.WriteLinePLC("  sup0Current := " + a[0].GetInitialState() +
"); //Sets the initialstate
    writer.WriteLinePLC("  oneTime := FALSE; //oneTime is a Boolean that
garantees that the evaluation process is only run once for each set of input
values");
    writer.WriteLinePLC("END_IF");
    writer.WriteLinePLC("");//This is only true if there is a call from a
Robot and no evaluation has taken place");
    writer.WriteLinePLC("IF supRequest AND NOT oneTime THEN");
    writer.WriteLinePLC("  oneTime := TRUE;");
    writer.WriteLinePLC("");//If the eventnumber is larger then the total
number of events in the matrix it is always allowed");
    writer.WriteLinePLC("  IF supervisorEvent > 12 THEN");
    writer.WriteLinePLC("    supervisor_Answer := TRUE;");
    writer.WriteLinePLC("    O_supReply := TRUE;");
    writer.WriteLinePLC("    RETURN;");
    writer.WriteLinePLC("  END_IF");
    writer.WriteLinePLC("");//This is a call for Function Block 3, sup0isReady
");
    writer.WriteLinePLC("");//The input supEvent_sup0isReady is given the
value OF supervisorEvent");
    writer.WriteLinePLC("");//The respons sup0isReady_Output is returned to
supervisorAllow");
    writer.WriteLinePLC("  sup0isReady.sup0isReadyDB3(supEvent_sup0isReady
:= supervisorEvent); // IN: INT");
    writer.WriteLinePLC("  supervisorAllow :=
sup0isReadyDB3.sup0isReady_Output; // OUT: BOOL");

```



```

        writer.blankLine();
        writer.writeLinePLC("//If supervisorAllow is true then the event is
allowed.");
        writer.writeLinePLC(" IF (supervisorAllow) THEN");
        writer.writeLinePLC("     supervisor_Answer := TRUE;");
        writer.writeLinePLC("     sup0current := sup0Target; //Update the
current value of sup0current");
        writer.writeLinePLC(" ELSE");
        writer.writeLinePLC("     supervisor_Answer := FALSE; //The event is
not allowed");
        writer.writeLinePLC(" END_IF;");
        writer.writeLinePLC(" O_supReply := TRUE; //The evaluation is
complete, and there is an answer");
        writer.writeLinePLC("END_IF;");
        writer.blankLine();
        writer.writeLinePLC("//When there is no call from the robot these
values are set to false.");
        writer.writeLinePLC("IF supRequest = FALSE THEN");
        writer.writeLinePLC(" O_supReply := FALSE;");
        writer.writeLinePLC(" supervisor_Answer := FALSE;");
        writer.writeLinePLC(" supervisorAllow := FALSE;");
        writer.writeLinePLC(" oneTime := FALSE;");
        writer.writeLinePLC("END_IF;");
        writer.writeLinePLC("END_FUNCTION_BLOCK");
        writer.blankLine();
        ProgressHandler.PerformStep();
        return true;
    }
#endregion

/// <summary>
/// Writes the second Function Block for the SCL supervisor
/// </summary>
#region
public static bool SupervisorPLCFB3(AutomatonMatrix[] a, CodeWriter writer)
{
    writer.writeLinePLC("//sup0isReady: The name sup0isReady is a heritage
from the original supervisor for a robot controller.");
    writer.writeLinePLC("FUNCTION_BLOCK FB3");
    writer.writeLinePLC("VAR_INPUT");
    writer.writeLinePLC(" supEvent_sup0isReady: INT; //The event to
evaluate, from supervisorEvent");
    writer.writeLinePLC("END_VAR");
    writer.writeLinePLC("VAR_OUTPUT");
    writer.writeLinePLC(" sup0isReady_Output: BOOL; //The respons of the
evaluation");
    writer.writeLinePLC("END_VAR");
    writer.writeLinePLC("VAR");
    writer.writeLinePLC(" eventNr: INT; //The current event to evaluate");
    writer.writeLinePLC("//The following variabels are used to evaluate the
event");

    writer.writeLinePLC(" searchPos: DINT;");
    writer.writeLinePLC(" tempPos: DINT;");
    writer.writeLinePLC(" sup0_left: INT;");
    writer.writeLinePLC(" sup0_right: INT;");
    writer.writeLinePLC(" sup0_mid: INT;");
    writer.writeLinePLC(" bound: INT;");
    writer.writeLinePLC(" loopcounter: INT;");
    writer.writeLinePLC("//sup0Pos and sup0Val are Arrays that represent
the automata matrix from Supremica");
    RapidCodeTemplatesPLC.SupervisorBoundArraysPLC(a, writer);
    writer.writeLinePLC(" maxNum: DINT := 300000; //Used for faster
evaluation of larger arrays");
    writer.writeLinePLC(" sup0Cols: INT :=" + a[0].GetNrOfCols() + ";
//The number of columns in the compressed matrix");
    writer.writeLinePLC(" sup0Length: INT :=" + a[0].NrOfnzv() + "; //Sets
the size of the arrays, i.e. The number of non zero values in the matrix.");
    writer.writeLinePLC("END_VAR");
}

```

```

        writer.writeLinePLC("BEGIN");
        writer.writeLinePLC("//Further details of the evaluation process bellow
can be found in these master thesis:");
        writer.writeLinePLC("//HAZIRAJ & FALLSTROEM Report No.EX078/2006");
        writer.writeLinePLC("//WEINEMYR & ZEKAJ Report No.EX080/2007");
        writer.blankLine();
        writer.writeLinePLC("eventNr := supEvent_sup0isReady;");
        writer.writeLinePLC("searchPos := (sup0Current*sup0Cols)+eventNr;");
        writer.writeLinePLC("sup0_left := 1;");
        writer.writeLinePLC("sup0_right := sup0Length;");
        writer.writeLinePLC("// Binary search through the arrays");
        writer.writeLinePLC("IF (searchPos>maxNum) THEN");
        writer.writeLinePLC("    bound:=1;");
        writer.writeLinePLC("    searchPos:=eventNr;");
        writer.writeLinePLC("    FOR loopcounter := 1 TO sup0Current DO");
        writer.writeLinePLC("        tempPos:=searchPos+sup0Cols;");
        writer.writeLinePLC("        IF (tempPos>maxNum) THEN");
        writer.writeLinePLC("            bound:=bound+1;");
        writer.writeLinePLC("            tempPos:=searchPos-maxNum;");
        writer.writeLinePLC("            tempPos:=tempPos+sup0Cols;");
        writer.writeLinePLC("        END_IF;");
        writer.writeLinePLC("        searchPos:=tempPos;");
        writer.writeLinePLC("    END_FOR;");
        writer.writeLinePLC("    sup0_left := sup0Bounds[bound];");
        writer.writeLinePLC("    sup0_right := sup0Bounds[bound+1]-1;");
        writer.writeLinePLC("    END_IF;");
        writer.writeLinePLC("    WHILE (sup0_left<sup0_right) DO");
        writer.writeLinePLC("        sup0_mid :=
DINT_TO_INT(TRUNC((sup0_left+sup0_right)/2));");
        writer.writeLinePLC("        IF (searchPos>DB55.sup0Pos[sup0_mid])
THEN");
        writer.writeLinePLC("            sup0_left := sup0_mid+1;");
        writer.writeLinePLC("        ELSIF (searchPos<DB55.sup0Pos[sup0_mid])
THEN");
        writer.writeLinePLC("            sup0_right := sup0_mid-1;");
        writer.writeLinePLC("        ELSE");
        writer.writeLinePLC("            sup0Target := DB55.sup0Val[sup0_mid];");
        writer.writeLinePLC("            sup0isReady_Output := TRUE;");
        writer.writeLinePLC("            RETURN;");
        writer.writeLinePLC("        END_IF;");
        writer.writeLinePLC("    END_WHILE;");
        writer.writeLinePLC("    IF (DB55.sup0Pos[sup0_right] = searchPos)
THEN");
        writer.writeLinePLC("        sup0Target := DB55.sup0Val[sup0_right];");
        writer.writeLinePLC("        sup0isReady_Output := TRUE;");
        writer.writeLinePLC("        RETURN;");
        writer.writeLinePLC("    END_IF;");
        writer.writeLinePLC("    IF (DB55.sup0Pos[sup0_left] = searchPos) THEN");
        writer.writeLinePLC("        sup0Target := DB55.sup0Val[sup0_left];");
        writer.writeLinePLC("        sup0isReady_Output := TRUE;");
        writer.writeLinePLC("        RETURN;");
        writer.writeLinePLC("    END_IF;");
        writer.writeLinePLC("    sup0Target := sup0Current;");
        writer.writeLinePLC("    sup0isReady_Output := FALSE;");
        writer.writeLinePLC("END_FUNCTION_BLOCK");
        return true;
    }
#endregion
/// <summary>
/// This function writes the Data Block that creates the arrays sup0Pos and
sup0Var for the SCL supervisor
/// </summary>
/// <param name="a"></param>
/// <param name="writer"></param>
/// <returns></returns>
public static bool SupervisorPLCDB55(AutomatonMatrix[] a, CodeWriter
writer)
{

```

```

writer.blankLine();
writer.writeLinePLC("//The Data Block for the sup0Pos and sup0Val
arrays");
writer.writeLinePLC("DATA_BLOCK DB55");
writer.writeLinePLC("STRUCT");
writer.writeLinePLC("sup0Pos: ARRAY [0.." + (a[0].NrOfnzv()-1) + "] OF
INT := " + (a[0].NrOfnzv()-1) + "(1);");
writer.writeLinePLC("sup0Val: ARRAY[0.." + (a[0].NrOfnzv()-1) + "] OF
INT := " + (a[0].NrOfnzv()-1) + "(1);");
writer.writeLinePLC("END_STRUCT");
writer.writeLinePLC("BEGIN");

for (int i = 0; i < a[0].NrOfnzv(); i++)//Assign all the values in the
postionarray
{
writer.writeLinePLC("sup0Pos[" + i + "]:= " +
(int)(a[0].getPosByIndex(i) % RapidCodeTemplates.maxNum) + ";");
}
writer.blankLine();
for (int i = 0; i < a[0].NrOfnzv(); i++)//Assign all the values in the
valuearray
{
writer.writeLinePLC("sup0Val[" + i + "]:= " + a[0].GetValByIndex(i)
+ ";");
}
writer.blankLine();
writer.writeLinePLC("END_DATA_BLOCK");
return true;
}
}
}

```

Appendix C - BoxSweeper.cs

These are the additions that have been made to the BoxSweeper.cs file during this master thesis.

```
public static void GenerateSupervisorPLC(System.String path)
{
    try
    {
        AutomatonMatrix[] matrix;
        java.util.Iterator supervisorIterator;
        int counter = 0;
        string name = "";
        //Create filepaths if they doesnt exist
        if (!System.IO.Directory.Exists(path))
        {
            System.IO.Directory.CreateDirectory(path);
        }
        SaveFileDialog dialog = new SaveFileDialog();
        dialog.DefaultExt = ".SCL";
        dialog.FileName = BoxSweeper.Station.Name + ".SCL";
        DialogResult result = dialog.ShowDialog();
        if (result == DialogResult.OK)
        {
            name = dialog.FileName;
            LoggHandler.Logg("File will be saved as " + dialog.FileName +
".");
        }

        //iterate over all supervisors
        supervisorIterator =
supremicaFacade.GetSupervisorModel().getAutomata().iterator();
        if (supervisorIterator == null)
        LoggHandler.Logg("SupervisorIterator is null!");
        //Create the automatonmatrixarray
        matrix = new
AutomatonMatrix[supremicaFacade.GetSupervisorModel().nbrOfAutomata()];
        //Loop for all supervisors
        while (supervisorIterator.hasNext())
        {
            ProgressHandler.PerformStep("Building Matrix");
            //Create the automatonmatrix
            matrix[counter] = new
AutomatonMatrix((Automaton)supervisorIterator.next(), "sup" + counter);
            ProgressHandler.PerformStep();
            ProgressHandler.PerformStep();
            counter++;
        }

        //LoggHandler.Logg("PLC code will be written to file: "+ name);
        //Check that the file is up and working
        CodeWriter rapidWriter = new CodeWriter(name);
        if (rapidWriter == null)
            LoggHandler.Logg("PLCcode is null.");
        rapidWriter.open();
        if (!rapidWriter.fileIsOpen())
            LoggHandler.Logg("PLCfile is not open.");
        //Write RAPIDcode to the file
        RapidCodeTemplatesPLC.SupervisorPLCFB2(matrix, rapidWriter);
        ProgressHandler.PerformStep();
        RapidCodeTemplatesPLC.SupervisorPLCFB3(matrix, rapidWriter);
        ProgressHandler.PerformStep();
        RapidCodeTemplatesPLC.SupervisorPLCDB55(matrix, rapidWriter);
        ProgressHandler.PerformStep();
        rapidWriter.close();
    }
}
```

```
        LoggHandler.Logg("Supervisor written to file.");
        controller.DisconnectController();
    }
    catch (Exception ex) { LoggHandler.Logg("Exception:" + ex.Message,
LogMessageSeverity.Error); }
}

//Write Robotcode for BoxSweeper PLC
public static void GenerateRobotCodePLC(string path)
{
    RapidCodeTemplatesPLC.WriteRobotRapidCodePLC(path, simulationData);
}
```

Appendix D – Example program for BoxSweeper PLC

This appendix contains an example program that can be used to present BoxSweeper PLC. The example is made for the two robots in the “PPU-lab” and can be supervised from a Siemens 313 PLC with SCL.

Supervisor - Siemens SCL

The following code shall be copied into the SCL compiler in Siemens Simatic Step 7 to generate the supervisor code. See appendix X for a detailed explanation of how to create the plc code.

```
FUNCTION_BLOCK FB2
VAR_INPUT
    supervisorEvent: INT; //supervisorEvent is the event that the robot want the supervisor to evaluate
    supRequest: BOOL; //supRequest is a call to tell the Supervisor that there is a new event to evaluate
    initialize: BOOL; //initialize is a initialization from Robot 1 that sets the initial state (sup0Current) for the
Supervisor
END_VAR
VAR_OUTPUT
    supervisor_Answer: BOOL; //supervisorAnswer is the respons from the supervisor wheter the event is
allowed or not
    O_supReply: BOOL; //O_supReply is a call to tell the Robot that the evaluation is complete
END_VAR
VAR
    supervisorAllow: BOOL; //supervisorAllow is a local Boolean that recives the result of the evaluation from the
calculation
END_VAR
BEGIN
    IF initialize THEN
        sup0Current := 7;
        oneTime := FALSE; //oneTime is a Boolean that garantes that the evaluation process is only run once for
each set of input values
    END_IF;

    //This is only true if there is a call from a Robot and no evaluation has taken place
    IF supRequest AND NOT oneTime THEN
        oneTime := TRUE;
        //If the eventnumber is larger then the total number of events in the matrix it is always allowed
        IF supervisorEvent > 12 THEN
            supervisor_Answer := TRUE;
            O_supReply := TRUE;
            RETURN;
        END_IF;

        //This is a call for Function Block 3, sup0isReady
        //The input supEvent_sup0isReady is given the value OF supervisorEvent
        //The respons sup0isReady_Output is returned to supervisorAllow
        sup0isReady.sup0isReadyDB3(supEvent_sup0isReady := supervisorEvent); // IN: INT
        supervisorAllow := sup0isReadyDB3.sup0isReady_Output; // OUT: BOOL

        //If supervisorAllow is true then the event is allowed.
        IF (supervisorAllow) THEN
            supervisor_Answer := TRUE;
            sup0current := sup0Target; //Update the current value of sup0current
        ELSE
```

```

    supervisor_Answer := FALSE; //The event is not allowed
END_IF;
O_supReply := TRUE; //The evaluation is complete, and there is an answer
END_IF;

```

```

//When there is no call from the robot these values are set to false.

```

```

IF supRequest = FALSE THEN
    O_supReply := FALSE;
    supervisor_Answer := FALSE;
    supervisorAllow := FALSE;
    oneTime := FALSE;
END_IF;
END_FUNCTION_BLOCK

```

```

//sup0isReady: The name sup0isReady is a heritage from the original supervisor for a robot controller.

```

```

FUNCTION_BLOCK FB3

```

```

VAR_INPUT

```

```

    supEvent_sup0isReady: INT; //The event to evaluate, from supervisorEvent

```

```

END_VAR

```

```

VAR_OUTPUT

```

```

    sup0isReady_Output: BOOL; //The respons of the evaluation

```

```

END_VAR

```

```

VAR

```

```

    eventNr: INT; //The current event to evaluate

```

```

    //The follwing variabels are used to evaluate the event

```

```

    searchPos: DINT;

```

```

    tempPos: DINT;

```

```

    sup0_left: INT;

```

```

    sup0_right: INT;

```

```

    sup0_mid: INT;

```

```

    bound: INT;

```

```

    loopcounter: INT;

```

```

    //sup0Pos and sup0Val are Arrays that represent the automata matrix from Supremica

```

```

    sup0Bounds: ARRAY[1..2] OF INT := [0];

```

```

    maxNum: DINT := 300000; //Used for faster evaluation of larger arrays

```

```

    sup0Cols: INT :=12; //The number of columns in the compressed matrix

```

```

    sup0Length: INT :=48; //Sets the size of the arrays, i.e. The number of non zero values in the matrix.

```

```

END_VAR

```

```

BEGIN

```

```

    //Further details of the evaluation process bellow can be found in these master thesis:

```

```

    //HAZIRAJ & FALLSTROEM Report No.EX078/2006

```

```

    //WEINEMYR & ZEKAJ Report No.EX080/2007

```

```

    eventNr := supEvent_sup0isReady;

```

```

    searchPos := (sup0Current*sup0Cols)+eventNr;

```

```

    sup0_left := 1;

```

```

    sup0_right := sup0Length;

```

```

    // Binary search through the arrays

```

```

    IF (searchPos>maxNum) THEN

```

```

        bound:=1;

```

```

        searchPos:=eventNr;

```

```

        FOR loopcounter := 1 TO sup0Current DO

```

```

            tempPos:=searchPos+sup0Cols;

```

```

            IF (tempPos>maxNum) THEN

```

```

                bound:=bound+1;

```

```

                tempPos:=searchPos-maxNum;

```

```

                tempPos:=tempPos+sup0Cols;

```

```

    END_IF;
    searchPos:=tempPos;
END_FOR;
sup0_left := sup0Bounds[bound];
sup0_right := sup0Bounds[bound+1]-1;
END_IF;
WHILE (sup0_left<sup0_right) DO
    sup0_mid := DINT_TO_INT(TRUNC((sup0_left+sup0_right)/2));
    IF (searchPos>DB55.sup0Pos[sup0_mid]) THEN
        sup0_left := sup0_mid+1;
    ELSIF (searchPos<DB55.sup0Pos[sup0_mid]) THEN
        sup0_right := sup0_mid-1;
    ELSE
        sup0Target := DB55.sup0Val[sup0_mid];
        sup0isReady_Output := TRUE;
        RETURN;
    END_IF;
END_WHILE;
IF (DB55.sup0Pos[sup0_right] = searchPos) THEN
    sup0Target := DB55.sup0Val[sup0_right];
    sup0isReady_Output := TRUE;
    RETURN;
END_IF;
IF (DB55.sup0Pos[sup0_left] = searchPos) THEN
    sup0Target := DB55.sup0Val[sup0_left];
    sup0isReady_Output := TRUE;
    RETURN;
END_IF;
sup0Target := sup0Current;
sup0isReady_Output := FALSE;
END_FUNCTION_BLOCK
//The Data Block for the Arrays
DATA_BLOCK DB55
STRUCT
    sup0pos: ARRAY [0..48] OF INT := 48(1);
    sup0Val: ARRAY[0..48] OF INT := 48(1);
END_STRUCT
BEGIN
    sup0Val[0]:= 5;
    sup0Val[1]:= 9;
    sup0Val[2]:= 17;
    sup0Val[3]:= 20;
    sup0Val[4]:= 27;
    sup0Val[5]:= 35;
    sup0Val[6]:= 39;
    sup0Val[7]:= 45;
    sup0Val[8]:= 51;
    sup0Val[9]:= 56;
    sup0Val[10]:= 62;
    sup0Val[11]:= 71;
    sup0Val[12]:= 74;
    sup0Val[13]:= 81;
    sup0Val[14]:= 86;
    sup0Val[15]:= 92;
    sup0Val[16]:= 101;
    sup0Val[17]:= 111;
    sup0Val[18]:= 118;

```


sup0Val[19]:= 123;
sup0Val[20]:= 127;
sup0Val[21]:= 135;
sup0Val[22]:= 138;
sup0Val[23]:= 146;
sup0Val[24]:= 154;
sup0Val[25]:= 158;
sup0Val[26]:= 163;
sup0Val[27]:= 170;
sup0Val[28]:= 174;
sup0Val[29]:= 191;
sup0Val[30]:= 192;
sup0Val[31]:= 201;
sup0Val[32]:= 204;
sup0Val[33]:= 212;
sup0Val[34]:= 220;
sup0Val[35]:= 225;
sup0Val[36]:= 232;
sup0Val[37]:= 236;
sup0Val[38]:= 241;
sup0Val[39]:= 249;
sup0Val[40]:= 253;
sup0Val[41]:= 260;
sup0Val[42]:= 274;
sup0Val[43]:= 283;
sup0Val[44]:= 288;
sup0Val[45]:= 294;
sup0Val[46]:= 304;
sup0Val[47]:= 313;

sup0Val[0]:= 3;
sup0Val[1]:= 1;
sup0Val[2]:= 4;
sup0Val[3]:= 8;
sup0Val[4]:= 5;
sup0Val[5]:= 3;
sup0Val[6]:= 6;
sup0Val[7]:= 4;
sup0Val[8]:= 7;
sup0Val[9]:= 11;
sup0Val[10]:= 15;
sup0Val[11]:= 6;
sup0Val[12]:= 16;
sup0Val[13]:= 7;
sup0Val[14]:= 17;
sup0Val[15]:= 14;
sup0Val[16]:= 11;
sup0Val[17]:= 12;
sup0Val[18]:= 10;
sup0Val[19]:= 13;
sup0Val[20]:= 2;
sup0Val[21]:= 14;
sup0Val[22]:= 9;
sup0Val[23]:= 22;
sup0Val[24]:= 13;
sup0Val[25]:= 23;
sup0Val[26]:= 5;

```
sup0Val[27]:= 24;
sup0Val[28]:= 12;
sup0Val[29]:= 16;
sup0Val[30]:= 18;
sup0Val[31]:= 17;
sup0Val[32]:= 19;
sup0Val[33]:= 24;
sup0Val[34]:= 20;
sup0Val[35]:= 19;
sup0Val[36]:= 21;
sup0Val[37]:= 25;
sup0Val[38]:= 0;
sup0Val[39]:= 21;
sup0Val[40]:= 1;
sup0Val[41]:= 26;
sup0Val[42]:= 23;
sup0Val[43]:= 15;
sup0Val[44]:= 25;
sup0Val[45]:= 22;
sup0Val[46]:= 26;
sup0Val[47]:= 8;
```

END_DATA_BLOCK

Robot programs

This is the code that is needed for the robots.

Robot A

%%%

VERSION: 1

LANGUAGE: ENGLISH

%%%

MODULE Module1

```
CONST robtarget HOME:=[[722.670795738948,-
572.507975708975,774.632508373243],[0.00374907167687908,0.340169778169403,-0.00398507149409638,-
0.940348119414788],[0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
```

```
CONST robtarget T1:=[[151.080307059091,-
577.672297703063,507.49374557365],[0.00505000202216584,0.694660512107159,-0.00210566208758497,-
0.71931692360584],[0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
```

```
CONST robtarget vp1:=[[275.548323364846,-
575.972133455599,622.961409319486],[0.00478789890902568,0.611369532838123,-0.00264808558926063,-
0.791326328377898],[0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
```

```
CONST robtarget vp2:=[[329.354382576334,-
575.372240943603,659.42842056366],[0.00467170285097202,0.577521591974197,-0.00284721122951585,-
0.816357078357428],[0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
```

!! Variables for setting speed and accuracy of the robot when moving between targets

```
VAR speeddata speed := v100;
```

!! The below code was automatically generated by BoxSweeper PLC on 2008-03-05 17:56:28.

!! Edit with extreme care.

!! These variables are used for interrupts that are called upon while the robot is moving.

```

!! It is these interrupts that do the booking/unbooking of zones, while moving. For each
!! interrupt there is one trigger variable and one interrupt name. Since RobotStudio only
!! has support for names up to 32 characters, they each get a short 'alias' name.
VAR triggdata trigg_1; !! Alias trigg_1 is associated with event(s) RA_b_0_m1_1
VAR intnum int_1;
VAR triggdata trigg_2; !! Alias trigg_2 is associated with event(s) RA_u_0_m1_1
VAR intnum int_2;

!! Variabels and matrix for communication with the PLC
VAR bool Initialize := TRUE;
VAR num eventNr := -1;

!! The array of strings containing the name of the events
CONST string sup0Events{12} :=
["RA_b_0_m1_1","RA_g_HOME","RA_g_T1","RA_r_HOME","RA_r_T1","RA_u_0_m1_1","RB_b_0_m1_1","RB_g
_HOME","RB_g_T1","RB_r_HOME","RB_r_T1","RB_u_0_m1_1"];

PROC main()
!! The TriggInt commands are used to define the triggers to go off when the robot is leaving
!! one target and starting its move to the next target.
!! The CONNECT statements defines which interrupt to call upon when the triggers go off.
CONNECT int_1 WITH trap_1;
TriggInt trigg_1, 0 \Start, int_1;
CONNECT int_2 WITH trap_2;
TriggInt trigg_2, 0 \Start, int_2;

!! Signals the PLC-initialization
IF initialize THEN
    SET O_initialize;
    WaitTime 0.5;
    RESET O_initialize;
    initialize := FALSE;
ENDIF

AT_HOME:
IF (isEnabled("RA_g_T1")) THEN
    triggJ vp1, speed, trigg_1, z20, tool0\WObj := wobj0;
    moveJ T1, speed, fine, tool0\WObj := wobj0;
    waitUntilEnabled("RA_r_T1");
    GOTO AT_T1;
ENDIF
GOTO AT_HOME;

AT_T1:
IF (isEnabled("RA_g_HOME")) THEN
    moveJ vp2, speed, z20, tool0\WObj := wobj0;
    triggJ HOME, speed, trigg_2, fine, tool0\WObj := wobj0;
    waitUntilEnabled("RA_r_HOME");
    GOTO AT_HOME;
ENDIF
GOTO AT_T1;

ENDPROC

!! TRAP routines, these are the interrupts that are called upon by the trigg instructions.
!! These communicate with supervisor for booking/unbooking events. The interrupt ends as

```

!! soon as the supervisor enables the event. The main program is then resumed.

```
TRAP trap_1
  waitUntilEnabled("RA_b_0_m1_1");
ENDTRAP
```

```
TRAP trap_2
  waitUntilEnabled("RA_u_0_m1_1");
ENDTRAP
```

!! Description: Query the supervisor if an event is allowed.

!! Parameter : event - The name of the event.

!! Return : TRUE if the event is allowed, FALSE otherwise.

```
FUNC bool isEnabled(string event)
  VAR bool eventAllowed;
  !! Get the number associated with the event
  FOR loopcounter from 1 TO 12 DO
    IF (event = sup0Events{loopcounter}) THEN
      eventNr := loopcounter-1;
    ENDIF
  ENDFOR
  !! Waiting for semaphore
  WaitDO O_Semaphore, 0;
  SET O_Semaphore;
  SETGO GO_Event, eventNr;
  WaitUntil GI_Event = eventNr;
  SET O_supRequest;
  WaitDI I_supReply, 1;
  !! Now the supervisor has answered the question
  IF I_supAnswer = 1 THEN
    eventAllowed := TRUE;
  ENDIF
  !! Reset O_supRequest to stop the PLC from changing current state
  RESET O_supRequest;
  RESET O_semaphore;
  RETURN eventAllowed;
ENDFUNC
```

!! Description: Pause the task until the event is allowed.

!! Parameter : event - The name of the event.

```
PROC waitUntilEnabled(string event)
  VAR bool done := FALSE;
  WHILE (NOT done) DO
    IF (isEnabled(event)) THEN
      done := TRUE;
    ELSE
      WaitTime 0.25;
    ENDIF
  ENDWHILE
ENDPROC
ENDMODULE
```

Robot B

```
%%%
VERSION: 1
LANGUAGE: ENGLISH
```

%%%

MODULE Module1

```
CONST robtarget HOME:=[[-479.616711685908,-
586.997164597069,774.464366934938],[0.927673900592607,-0.00202913444571806,0.373369995540176,-
0.00344430007821262],[0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
CONST robtarget T1:=[[33.6162553592659,-591.220317584882,502.883986452669],[0.720429216881672,-
0.000592102528900461,0.693516952116986,-0.00395348001422098],[0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
CONST robtarget vp3:=[[-266.288621761906,-588.375498619176,721.074441875323],[0.865364535374578,-
0.00152028983016734,0.501126973954923,-0.00369670252445165],[0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
CONST robtarget vp4:=[[-300.886265669508,-588.123391433669,734.240632221305],[0.877078539233799,-
0.00160811068341455,0.480330364487038,-0.00365936432796496],[0,0,0],[9E9,9E9,9E9,9E9,9E9,9E9]];
```

```
!! Variables for setting speed and accuracy of the robot when moving between targets
VAR speeddata speed := v100;
```

```
!! The below code was automatically generated by BoxSweeper PLC on 2008-03-05 17:56:29.
!! Edit with extreme care.
```

```
!! These variables are used for interrupts that are called upon while the robot is moving.
!! It is these interrupts that do the booking/unbooking of zones, while moving. For each
!! interrupt there is one trigger variable and one interrupt name. Since RobotStudio only
!! has support for names up to 32 characters, they each get a short 'alias' name.
VAR triggdata trigg_1; !! Alias trigg_1 is associated with event(s) RB_b_0_m1_1
VAR intnum int_1;
VAR triggdata trigg_2; !! Alias trigg_2 is associated with event(s) RB_u_0_m1_1
VAR intnum int_2;
```

```
!! Variabels and matrix for communication with the PLC
VAR num eventNr := -1;
```

```
!! The array of strings containing the name of the events
CONST string sup0Events{12} :=
["RA_b_0_m1_1","RA_g_HOME","RA_g_T1","RA_r_HOME","RA_r_T1","RA_u_0_m1_1","RB_b_0_m1_1","RB_g
_HOME","RB_g_T1","RB_r_HOME","RB_r_T1","RB_u_0_m1_1"];
```

PROC main()

```
!! The TriggInt commands are used to define the triggers to go off when the robot is leaving
!! one target and starting its move to the next target.
!! The CONNECT statements defines which interrupt to call upon when the triggers go off.
CONNECT int_1 WITH trap_1;
TriggInt trigg_1, 0 \Start, int_1;
CONNECT int_2 WITH trap_2;
TriggInt trigg_2, 0 \Start, int_2;
!! Synchronization to initialize (rob 1)
WaitTime 2;
```

```
AT_HOME:
IF (isEnabled("RB_g_T1")) THEN
triggJ vp3, speed, trigg_1, z20, tool0\WObj := wobj0;
moveJ T1, speed, fine, tool0\WObj := wobj0;
waitUntilEnabled("RB_r_T1");
GOTO AT_T1;
ENDIF
GOTO AT_HOME;
```

```

AT_T1:
IF (isEnabled("RB_g_HOME")) THEN
  moveJ vp4, speed, z20, tool0\WObj := wobj0;
  triggJ HOME, speed, trigg_2, fine, tool0\WObj := wobj0;
  waitUntilEnabled("RB_r_HOME");
  GOTO AT_HOME;
ENDIF
GOTO AT_T1;

```

ENDPROC

!! TRAP routines, these are the interrupts that are called upon by the trigg instructions.
 !! These communicate with supervisor for booking/unbooking events. The interrupt ends as
 !! soon as the supervisor enables the event. The main program is then resumed.

```

TRAP trap_1
  waitUntilEnabled("RB_b_0_m1_1");
ENDTRAP

```

```

TRAP trap_2
  waitUntilEnabled("RB_u_0_m1_1");
ENDTRAP

```

!! Description: Query the supervisor if an event is allowed.
 !! Parameter : event - The name of the event.
 !! Return : TRUE if the event is allowed, FALSE otherwise.

```

FUNC bool isEnabled(string event)
VAR bool eventAllowed;
!! Get the number associated with the event
FOR loopcounter from 1 TO 12 DO
  IF (event = sup0Events{loopcounter}) THEN
    eventNr := loopcounter-1;
  ENDIF
ENDFOR
!! Waiting for semaphore
WaitDO O_Semaphore, 0;
SET O_Semaphore;
SETGO GO_Event, eventNr;
WaitUntil GI_Event = eventNr;
SET O_supRequest;
WaitDI I_supReply, 1;
!! Now the supervisor has answered the question
IF I_supAnswer = 1 THEN
  eventAllowed := TRUE;
ENDIF
!! Reset O_supRequest to stop the PLC from changing current state
RESET O_supRequest;
RESET O_semaphore;
RETURN eventAllowed;
ENDFUNC

```

!! Description: Pause the task until the event is allowed.
 !! Parameter : event - The name of the event.

```

PROC waitUntilEnabled(string event)
VAR bool done := FALSE;
WHILE (NOT done) DO
  IF (isEnabled(event)) THEN
    done := TRUE;
  ENDIF
ENDWHILE

```

```
ELSE
  WaitTime 0.25;
ENDIF
ENDWHILE
ENDPROC
ENDMODULE
```

Appendix E – Manuals

In this appendix the manuals for how to use and operate RobotStudio, BoxSweeper and PLC are explained. Chapters 1-5 explains how to work with the real controller, chapters 6-14 explains the work with RobotStudio and finally the work with the PLC are explained in chapters 15-16.

1. The steps and the correct way to run BoxSweeper generated programs

The process from the definition of targets, to load the generated program into the robots is explained in this chapter.

1. First create a system and define the targets, see chapter “7. How to create a new robot system in RobotStudio”.

After the system and targets has been defined there are two ways to proceed depending on where the supervisor is going to be executed, in the robot controller or in the PLC. Steps 2-5 are for the creation of a supervisor for the robot controller and steps 6-9 are for a PLC supervisor.

2. Run BoxSweeper, see chapter “6. How to run BoxSweeper”.
3. Save the robot programs and the supervisor, see chapter “12. How to save program and supervisor”.
4. Create a task in the real robot controller, if a supervisor task does not exist. See chapter “5. How to create a task in a real system”.
5. Load the robot programs and supervisor to the controller, see chapter “4. How to load a robot program to a real system”. Proceed with step 10.
6. Run BoxSweeper PLC, see chapter “6. How to run BoxSweeper”.
7. Save the robot programs, see chapter “12. How to save robot program and supervisor”.
8. Load the basic program to the PLC, see chapter “15. How to load a BoxSweeper PLC program to the PLC”.
9. Load the robot programs to the controller, see chapter “4. How to load a robot program to a real system”.

The programs have now been loaded and the next step is to initialize the robot controller.

10. For every task, enter and press “Debug” and select “PP to Main”.

When this is performed the robot system is ready to run, release the two emergency stops and push the flashing button on the robot controller. Press and hold the guard stop on the FlexPendant. When the Play-button, see chapter 2.2.5 FlexPendant, on the FlexPendant is pressed the system will start moving.

2. How to start the robot system in the PPU-lab

1. First start both the drive modules, nr. 1 in Figure 2.1.
2. Start the controller, nr. 2 in Figure 2.1.
3. If robot tools are going to be used, check if the tube for the compressed air is connected correctly. Then turn on the compressed air, Figure 2.2.
4. To be able to run the robots both the emergency stops, on the FlexPendant and on the controller cabinet, have to be released, nr. 3 in Figure 2.1.
5. After the emergency stops have been released the system demands a confirmation. Press the flashing button (the button will still flashing after it has been pressed), nr. 5 in Figure 2.1.

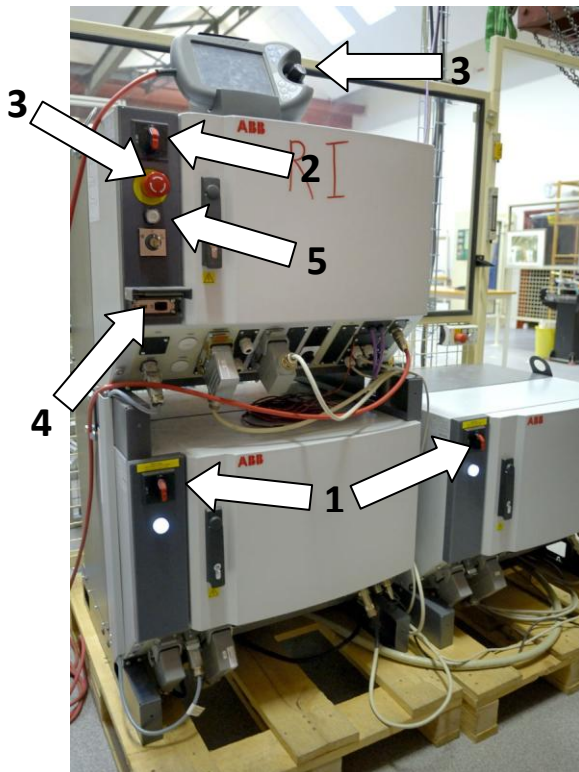


Figure 2.1: The robot system

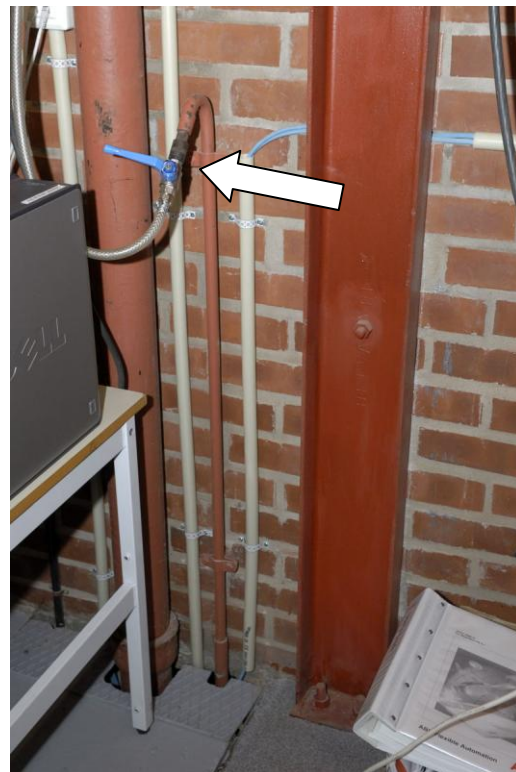


Figure 2.2: Where the valve for the compressed air is

3. How to take a backup on a real system

1. Press the ABB symbol on the FlexPendant, nr. 1 in Figure 3.1.
2. Press “Backup and restore”.
3. Press “Backup Current System”, nr. 2 in Figure 3.1.

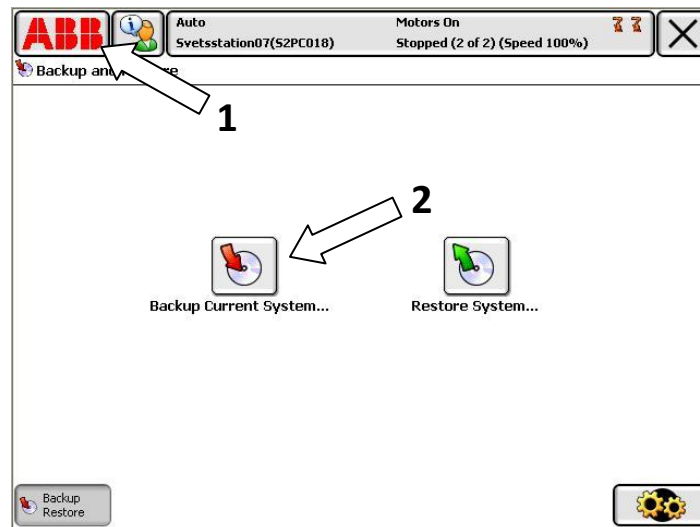



Figure 3.1: Backup of the system

The next step is to define the name of the backup and where the backup is going to be stored. The controller has a hard drive but it is recommended to store the backup on a USB memory. Connect the USB memory, nr. 4 in Figure 2.1 point.

4. Choose the name for the backup. The backup is named “Backup_*” by default, where the * is the current date. To change the name on the backup press “ABC...”, nr. 1 in Figure 3.2.
5. Change the path where the backup is going to be stored, nr. 2 in Figure 3.2. Change the path by pressing  several times to select the removable memory.
6. Finally press “Backup”, nr. 3 in Figure 3.2, to start the backup.

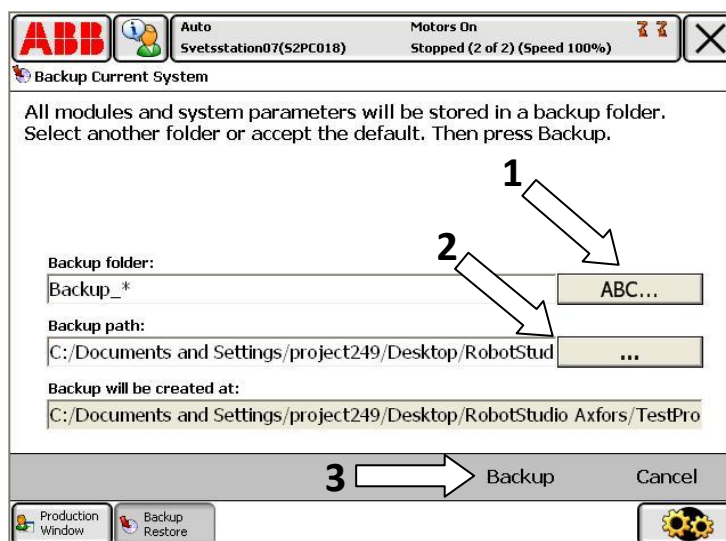



Figure 3.2: Name and path for saving the backup

4. How to load a robot program to a real system

1. Press the ABB symbol on the FlexPendant, nr. 1 in Figure 4.1, and then press “Program Editor”.
2. Press the task you want to load a program to, for example T_ROB1, nr. 2 in Figure 4.1.
- 3a. If no program is loaded to the system, the controller notifies that no program exists, choose “Load”. Select the path where the program is stored. If the program is stored on a USB memory, press  several times to change path from the internal hard drive to the USB memory. Select the program and then press “OK”.
- 3b. If a program is loaded to the system, select “Save” to save the program if the program has been modified since it was saved. Otherwise select “Don’t Save” and then load the desired program.

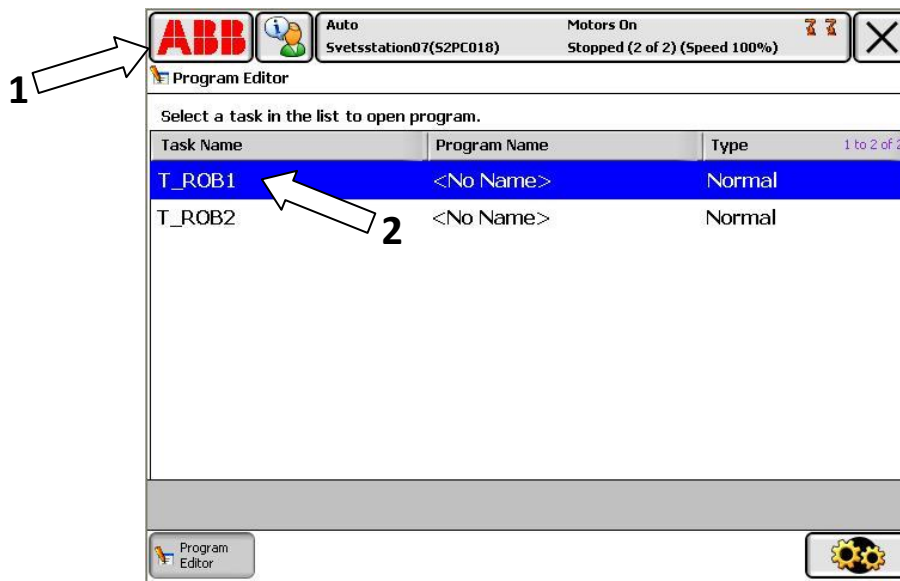


Figure 4.1: Load a program to a task

After a self check that the right robotprogram has been loaded, it is time to load the next task. To be able to load a new task you have to go back. Step 4 can be performed in two ways, preferably choose 4b since this will be assumed in the next step.

- 4a. Close the Program Editor, nr. 1 in Figure 4.2, and then repeat step 1-3.
- 4b. Press the “Tasks and Programs”, nr. 2 in Figure 4.2.

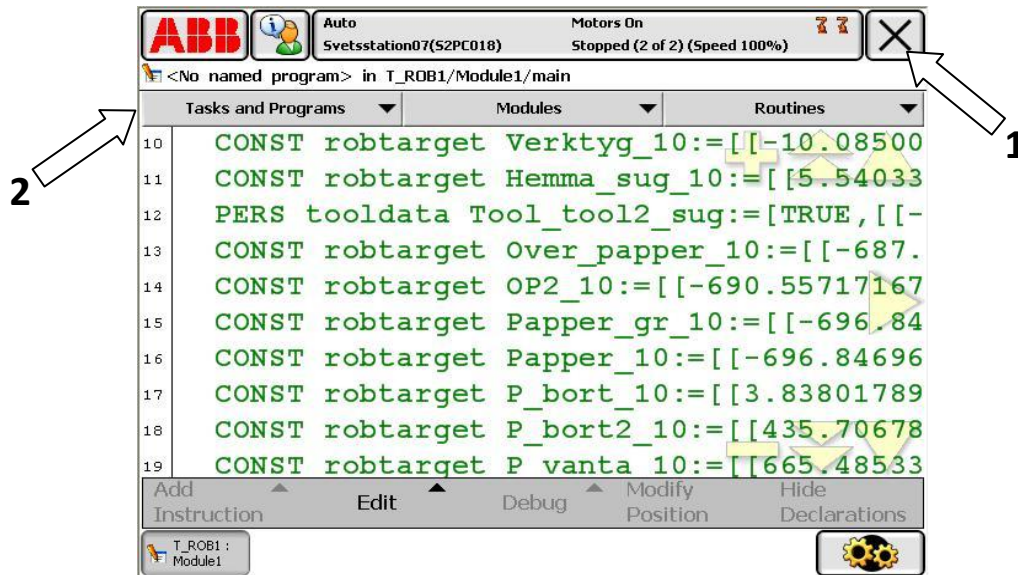


Figure 4.2: The alternatives to go back and load a new program

The following procedure are based on that alternative 4b was used. When you go back like in 4b, a harmless bug in the software is the reason for that a new menu shows, compare Figure 4.1 and Figure 4.3.

5. Select the task you want to load.
6. Press “File”, nr. 1 in Figure 4.3, and then press “Load Program...” nr. 2 in Figure 4.3.
7. Find the path were the program is saved and load the program.

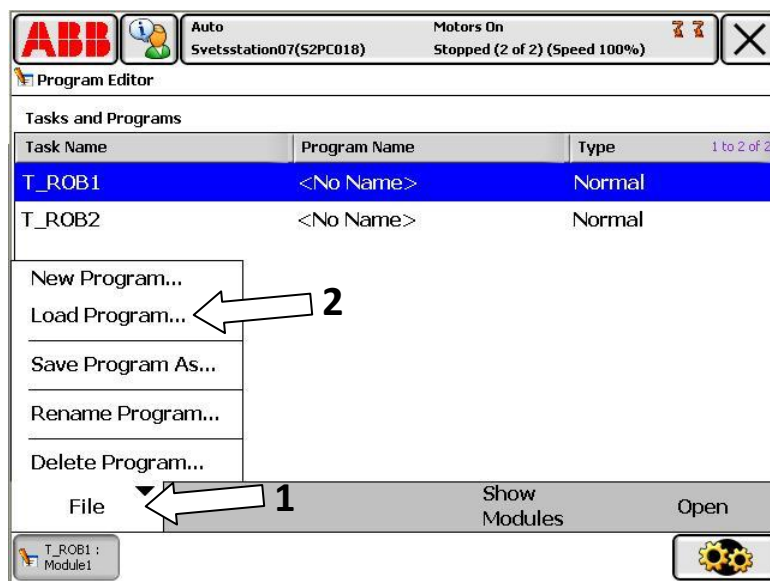


Figure 4.3: Load a program from the file menu

5. How to create a task in a real system

When BoxSweeper generated code is going to be loaded into a real system, a task for the supervisor has to be created in the controller.

1. Press the ABB symbol on the FlexPendant, nr. 1 in Figure 4.1, and then press “Control Panel”.
2. Press “Configuration”.
3. Change the topics by pressing “Topics” and then select “Controller”.
4. Press “Task” two times.
5. Press “Add”.
6. Name the new task, for example “Supervisor”. Press on the default name “tmp0” two times, nr. 1 in Figure 5.1, and type the new name.
7. Change the type of task to “Normal” by pressing on the default value “Semistatic”, nr. 2 in Figure 5.1. Read more about the different task types in “Operating manual – IRC5 with FlexPendant”, chapter 7.1.3 - Using multitasking programs.
8. Then press “OK” and choose to perform a warm start to update the controller.

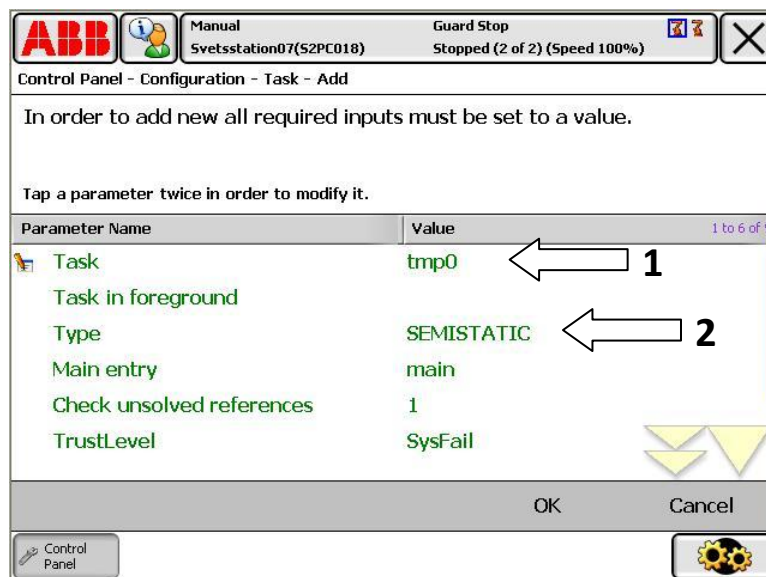


Figure 5.1: Defining a new task

6. How to run BoxSweeper

1. Rename the robots, in the objects menu, to a short name like RA and RB.

Name the targets for Robot1 T11,T12...Txx and T21,T22...Tyy for Robot2. A short name on the targets and robots is desired to allow BoxSweeper to work properly. It is now time to start BoxSweeper.

2. Press BoxSweeper on the menu bar to open the BoxSweeper add-in.
3. Perform a coldstart of the system. From the Controller menu select Start/Shutdown and then Coldstart.

The steps 4-5 are not necessary if paths already have been created.

4. Select the type of movement for the robots, MoveL or MoveJ, nr. 1 in Figure 6.1. MoveJ is to prefer.
5. Press "Create paths", nr. 2 in Figure 6.1. Every possible path between all targets is created. Remove not desired paths.
6. Select the box size. Use the scroll bar, do not enter the desired box size in the box where the size is presented, nr. 3 in Figure 6.1.

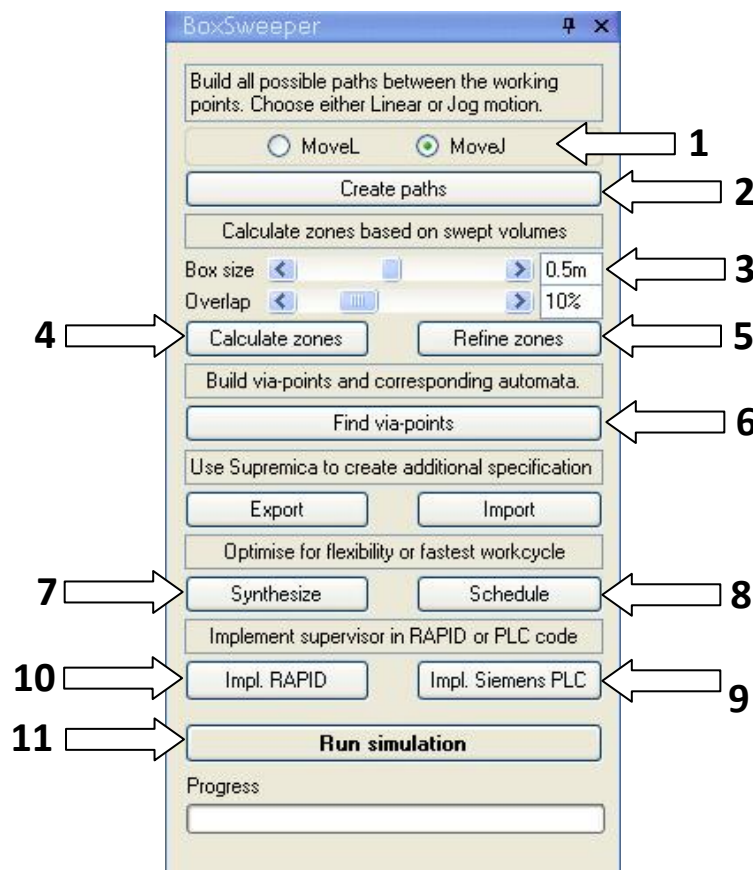


Figure 6.1: BoxSweeper add-in

7. Select the desired overlap for the boxes.
8. Now BoxSweeper is ready for creating boxes. Press "Calculate zones", nr. 4 in Figure 6.1. The robots will then move through its paths to detect mutual zones.

If smaller mutual boxes are desired when the zones have been created, perform step 9, otherwise continue to step 10.

9. When mutual boxes have been generated the alternative "Refine zones" is possible. Press "Refine zones", nr. 5 in Figure 6.1, to divide every box into 8 smaller boxes. Then the robot moves through its paths to detect mutual zones one more time.
10. When mutual zones are generated press "Find via-points", nr. 6 in Figure 6.1. The robots will then again move through its paths and place a via-point in the TCP for ever time some link of the robot enter or leaves a mutual zone.

Next step is to decide if the most flexible solution without dead-locks, step 13, or the fastest work cycle shall be generated, step 14.

11. Press "Synthesise", nr. 7 in Figure 6.1, for the most flexible solution.
12. Press "Schedule", nr. 8 in Figure 6.1, for the fastest work cycle and then select MILP as the optimization method.

All the information is collected and the generation of a supervisor and robot code can begin. If a supervisor for a robot controller is wanted, perform step 13 and 14. Step 15 is for the generation of a PLC supervisor.

13. Press "Impl. RAPID" to configure robot targets and to write the rapid cod.
14. When the rapid code has been generated, the system is ready. Press the "Run simulation", nr. 11 in Figure 6.1.
15. Press "Impl. Siemens PLC" to configure robot targets and to write the PLC code. The system cannot be simulated as in step 14, because of the supervisor has to be executed on a PLC.

7. How to create a new robot system in RobotStudio

1. Start RobotStudio 5.09
2. From the File menu select “New Station”.
3. By default “Template system” is selected. Select “No system” and then press “OK”.
4. Check that the “Layout” view is selected, see Figure 7.1.

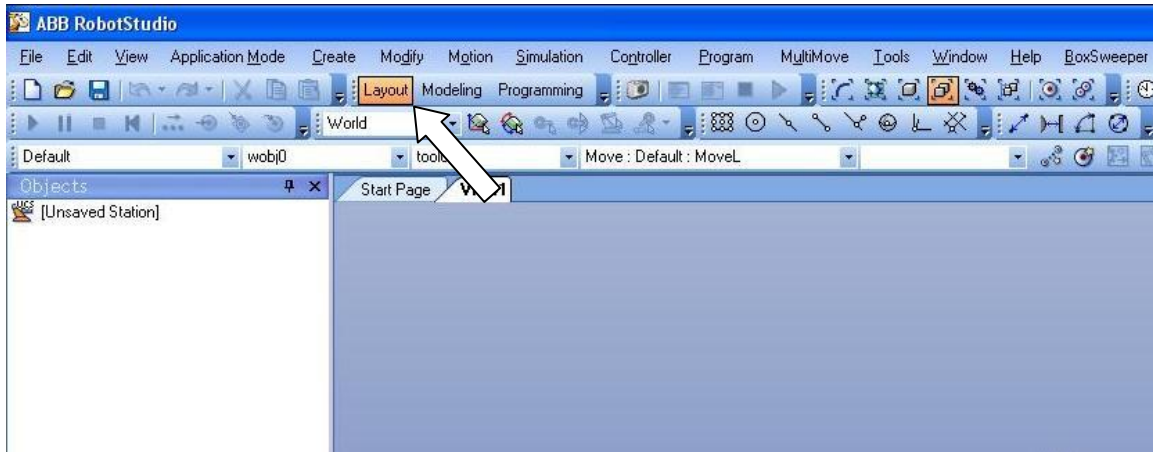


Figure 7.1: Check that layout view is chosen

5. From the Controller menu select “System Builder”
6. Decide where to save the new system by pressing the button labeled “...”.

When the place to save the system has been decided, there are two ways to proceed.

- 7a. Create a new system, which is explained in this chapter.
- 7b. Create a system based on a real system. How to do this is explained in the chapter “How to create a system in RobotStudio from a backup of the real system”.
8. To create a new system, press “Create New”.
9. First some information is prompted, read and then press “Next”.
10. Name the system under “Name”, nr. 1 in Figure 7.2. Check that the system is saved in the desired path by pressing “...”, nr. 2 in Figure 7.2. Then press “Next”.

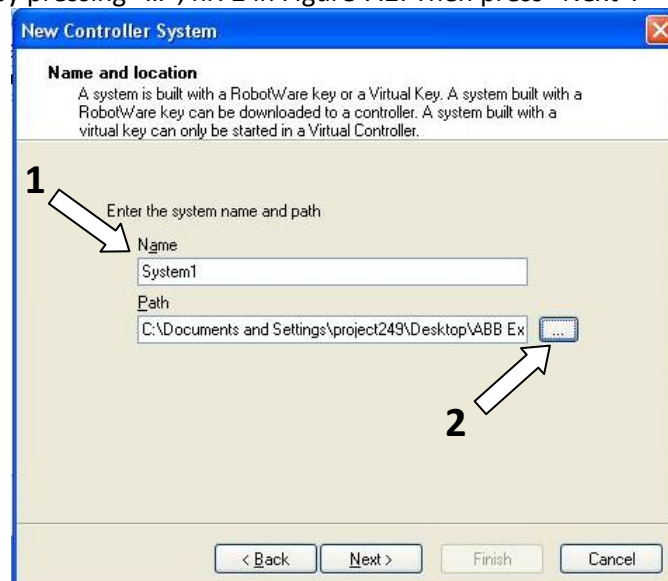


Figure 7.2: Name and path to save the system

11. Now it is time to enter a controller key, if there is one. Otherwise just check the “Virtual Key” box. If a virtual key is chosen the system will create a key and the information about RobotWare version is automatically inserted. The differences between a real- and a virtualkey are explained in chapter “2.2.7 Keys”. Press “Next” to proceed.
12. Add as many drive keys as the number of robots wanted in the system by pressing “→” one time for each robot, see Figure 7.3. When all robot keys are added, press “Next”.

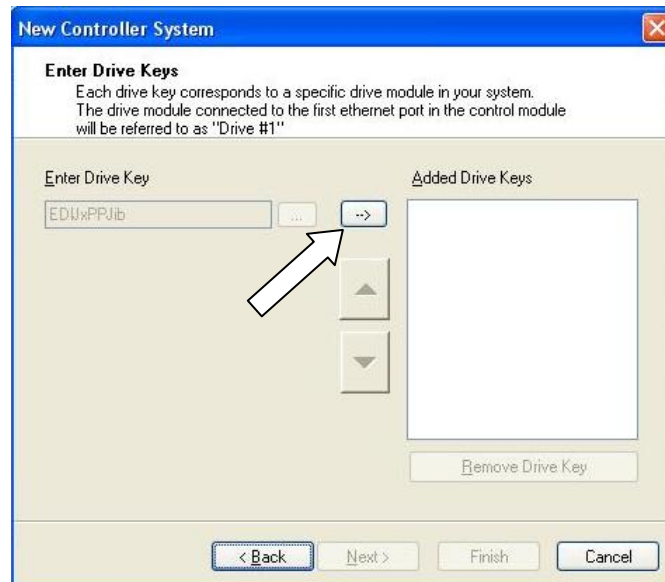


Figure 7.3: Enter drive keys to the system

13. If there is a file with the key which define the system, then add this in “Add Additional Options”. A key file is not necessary to create a system. When a key file is added or no file is used, press “Next”.
14. Specify the model type of the robots in “Modify Options”. The additional support for the system is specified here, for example if a robot that is attached to a track is wanted. See chapter “10. How to configure a system with a robot attached to a track” for how to set up a system with a robot attached to a track. When all options are selected choose “Next”.

In “Add Parameter Data” the system can be configured. Parameter data can for example be signals or robot positions. Information about the signals is saved in a file called *eio.cfg* and the positions of the robots in *moc.cfg*.

15. If parameter data is going to be added to the system, press "...", nr. 1 in Figure 7.4, and specify the path where the parameter data are being stored.
16. Then select the parameter data and add that to the system by pressing "→", nr. 2 in Figure 7.4. When all parameter data has been added, press "Next".

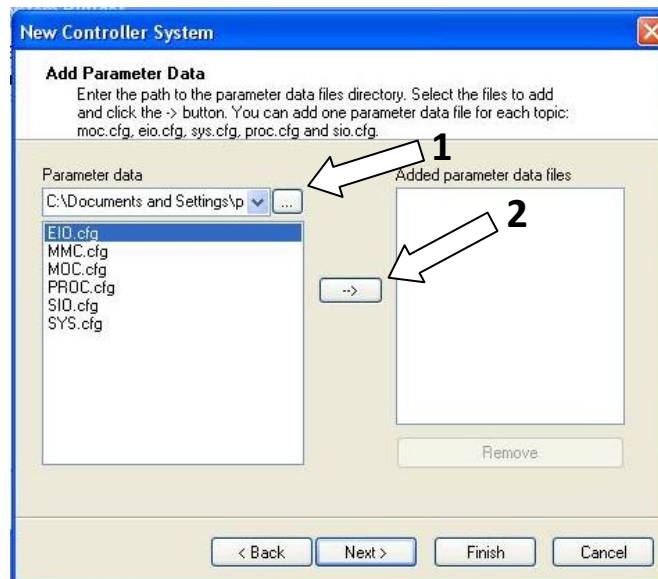


Figure 7.4: Add parameter data

17. If files shall be automatically loaded with the system in the home directory, in that case add these files. Then press "Next".
18. To complete the system setup, press "Finish".

Now the system is created and has to be added to a cell, see chapter "How to add an existing robot system to a cell".

8. How to create a system in RobotStudio from a backup of the real system

In order to create a system from a backup, a backup is needed. See chapter “3. How to take a backup on a real system” to be able to follow the steps in this chapter. When there is a backup, follow the steps 1-6 in chapter “7. How to create a new robot system in RobotStudio” and then perform the following:

1. Press “Create from backup”.
2. First some information is prompted, read and then press “Next”.
3. Name the system under “Name”, nr. 1 in Figure 7.2. Check that the system is saved in the desired path by pressing “...”, nr. 2 in Figure 7.2. Then press “Next”.
4. Select the backup folder of the real system, which is stored on the USB memory, by pressing “...”, nr. 1 in Figure 8.1. If there is no Media Pool displayed as in Figure 8.1, the default path is C:\Program Files\ABB Industrial IT\Robotics IT\Mediapool. When the backup folder is selected and there is a Media Pool, press “Next”.

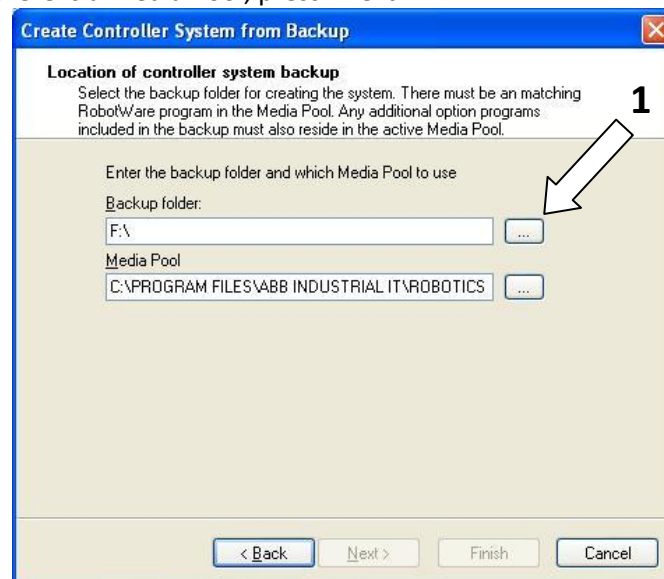


Figure 8.1: Select the path for the backup of the real system

Now the system is ready and there are two options for how to proceed. Press “Finish” to start using the system or press “Next” to do a control of what the system contains. The following steps are based on that “Next” was pressed.

5. The next dialog box is for changing the RobotWare version. If the RobotWare version is the wanted, press “Next”. Otherwise select “No, replace it” and check if the desired version is installed.
6. Check that the drive modules keys are correct and then press “Next”.
7. Follow the steps 13-18 in chapter “7. How to create a new robot system in RobotStudio” to complete the creation of a system.

9. How to add an existing robot system to a cell

See chapter “7. How to create a new robot system in RobotStudio”, steps 1-4, for how to create a new empty cell where the system is going to be added.

1. From the Controller menu select “Add system”.
2. By default “Add Template System” is selected. Select “Add Existing System”, nr. 1 in Figure 9.1.
3. Check if the desired system is listed under “Systems Found”, nr. 2 in Figure 9.1.
4. If the system is listed under “Systems Found” go to step 7, else the path where the system is saved has to be specified.
5. Press “Add...”, nr. 3 in Figure 9.1, and find the path where the system is stored. If the system called “BoxSweeper_System” is stored in for example “C:\RobotStudio\My_RobotSystems\BoxSweeper_System” the folder “My_RobotSystems” is the folder which shall be added.
6. When the correct path is displayed in “Select System Pool”, nr. 4 in Figure 9.1, all systems in this folder are then presented in “Systems Found”, nr. 2 in Figure 9.1.

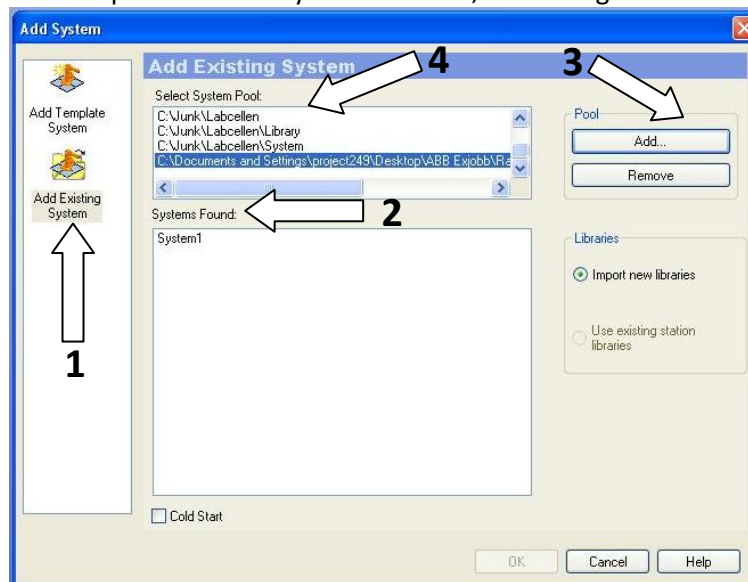


Figure 9.1: Add existing system to cell

7. Select the desired system and press “OK”
8. When the “Virtual Controller” has started, the work with the system can start.

10. How to configure a system with a robot attached to a track

Follow the steps in “7. How to create a new robot system in RobotStudio” with exception for step 14. In step 14 the following shall be performed:

1. In “Hardware”, select “709-x Devicenet”.
2. In “I/O control”, select “621-1 Logical Cross Connections”.

If there is more than one robot in the system, the following shall be performed under the desired DriveModule to specify which robot the track shall be attached to.

3. Check under “Drive module application” the chosen robot type, by default the “*IRB140 Type A*” is chosen.
4. In “Additional axes configuration”, select “R2C2 Add drive”.
5. Only one position needs to be selected for the track. Therefore select “753-1 Drive C in pos Z4”, like in Figure 10.1.
6. Press “Next” and follow the steps in “7. How to create a new robot system in RobotStudio”.

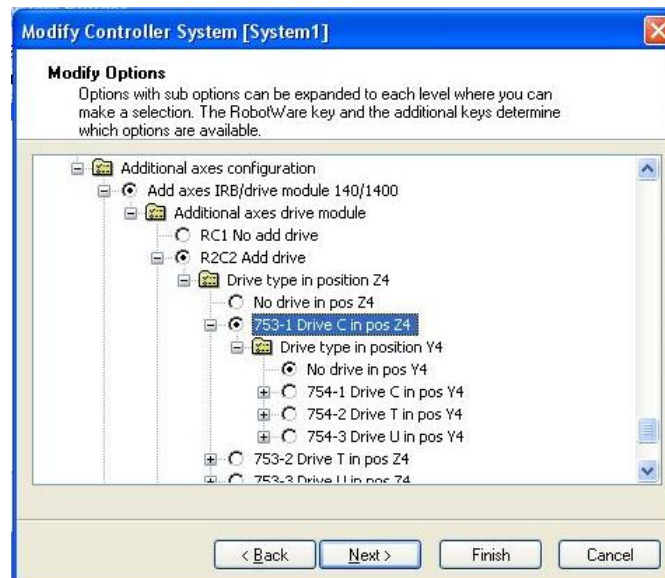


Figure 10.1: Configure a robot attached to a track.

11. How to configure a system for BoxSweeper

Follow the steps in “7. How to create a new robot system in RobotStudio” with exception for step 14. In step 14 the following steps have to be performed:

1. In “Motion coordination part 1”, select “604-2 MultiMove Independent”
2. In “Motion events”, select “609-1 Fixed Position Event”.
3. In “I/O control”, select “626-1 Advanced RAPID”.
4. Press “Next” and follow the steps in “7. How to create a new robot system in RobotStudio”.

12. How to save robot programs and supervisor in RobotStudio

When BoxSweeper has successfully generated robot programs the programs have to be saved, to be able to load these to the real controller.

1. Select the view “Programming”, nr. 1 in Figure 12.1.
2. Right click on one robot, nr. 2 in Figure 12.1.
3. Select “Save Program As..”, nr. 3 in Figure 12.1.
4. Change the path where the program is going to be saved.
5. Specify a name for the saved object, the files for the object will be saved in a sub-directory with the specified name.
6. Press “Save”.

Follow this method for all the robots. If a supervisor for the robot controller is created, the supervisor is stored in the same way.

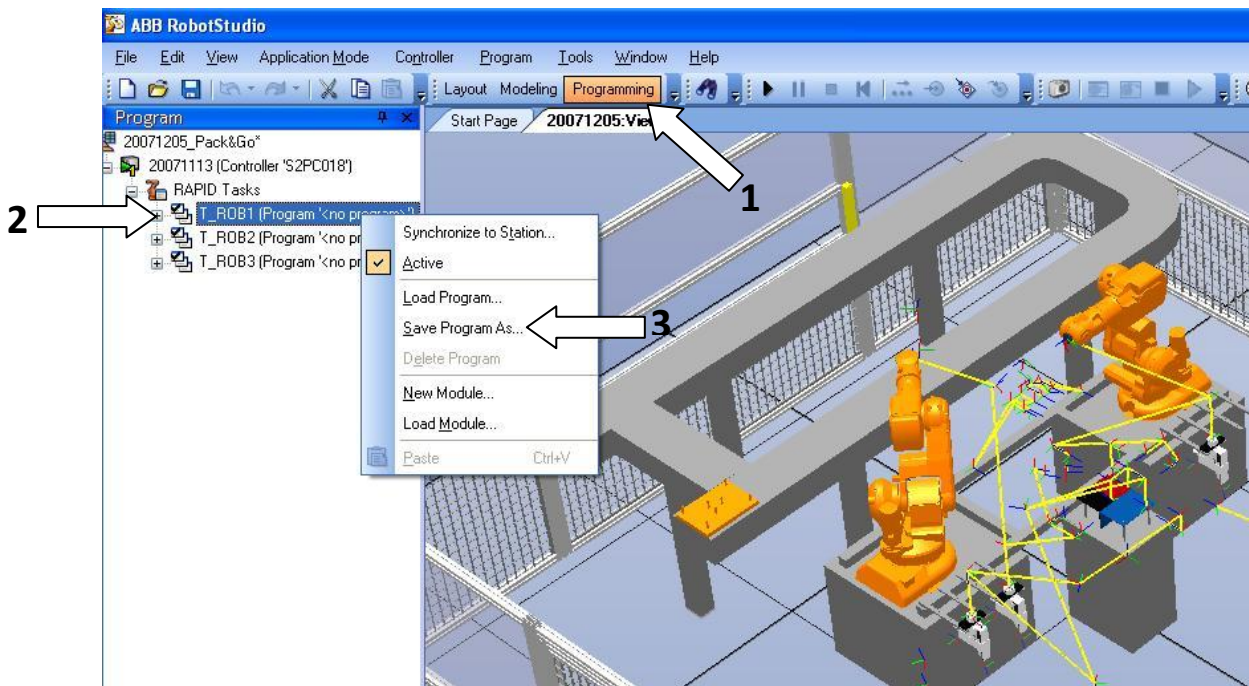


Figure 12.1: View of how to save robot programs and supervisor.

13. How to define signals in the robot controller

To be able to define signals, chapters “7. How to create a new robot system in RobotStudio” and “9. How to add an existing robot system to a cell” have to be performed.

1. Press the ABB symbol on the FlexPendant, nr 1 in Figure 4.1, and press “Control Panel”.
2. In the control panel, press “Configuration”.

If the dialog box “Functionality disabled as the robot controller is in automatic mode” appears, the key on the robot controller cabinet is in automatic mode, nr. 1 in Figure 13.1. The key has to be changed to manual mode, nr. 2 in Figure 13.1. If the signal is to be defined in RobotStudio the virtual FlexPendant also has this protection. To be able to set the controller to manual mode the key has to be turned like on the real system. Press the button with a picture of the real cabinet, to the left of the jog dial, and then Figure 13.1 will be displayed.

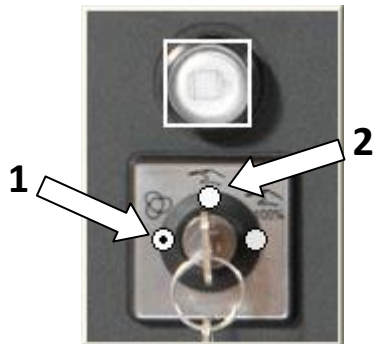


Figure 13.1: Operation mode of the robots

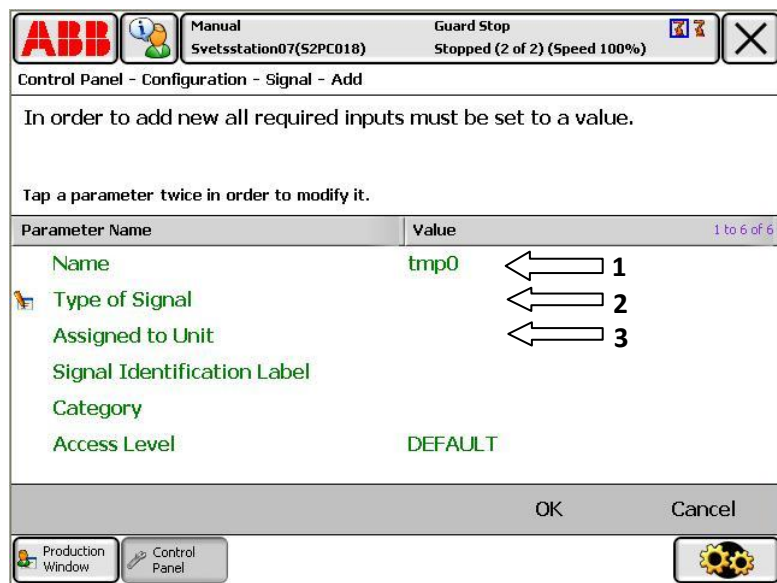


Figure 13.2: Add a signal to the system

3. Press “Topics” and make sure that “I/O” is selected.
4. Press “Signals” twice.

Now all the signals in the system is presented in the FlexPendant and new signals can be created or existing signals can be configured.

5. Press “Add” to create a new signal, the dialog box in Figure 13.2 will appear.
6. Give the signal a name by pressing on the default name “temp0” two times, nr. 1 in Figure 13.2, and type the new name.

A good naming standard is to name the signal depending on the type. For example a signal that is a digital output is preferably named DO_”SignalName” and for a digital input signal, DI_”SignalName”.

7. Select which type of signal to be defined, nr. 2 in Figure 13.2.
8. If the signal is going to be used with the Profibus, the signal has to be assigned to the PLC_bus, nr. 3 in Figure 13.2.
9. When the signal is assigned to the PLC unit, the mapping value also has to be set. Press the Unit mapping and specify the value of the signal. More about how to connect signals to the PLC is explained in chapter 3.4.1 Signal configuration for BoxSweeper PLC.
10. Change the “Access Level” to “ALL” and then press “OK”.
11. After a signal has been defined, the system wants to restart the controller. If there are more signals to define, press “No” and repeat steps 5-10. After the last signal has been defined, press “Yes” to restart the controller.

14. How to create mechanism in RobotStudio 5.08

This chapter is based on the procedure of how to create a mechanism like the fixture that was used in the automation project 2007. Movements that the fixture performs are a vertical (1) and a horizontal (2). When movement 1 and 2 are performed a rotation (3) movement is performed, see Figure 14.1 and 14.2. The parts of the mechanism were created in a Cad software and saved to a universal format to be able to import the parts to RobotStudio. This fixture has only 5 parts to make the definition of the mechanism a little easier.

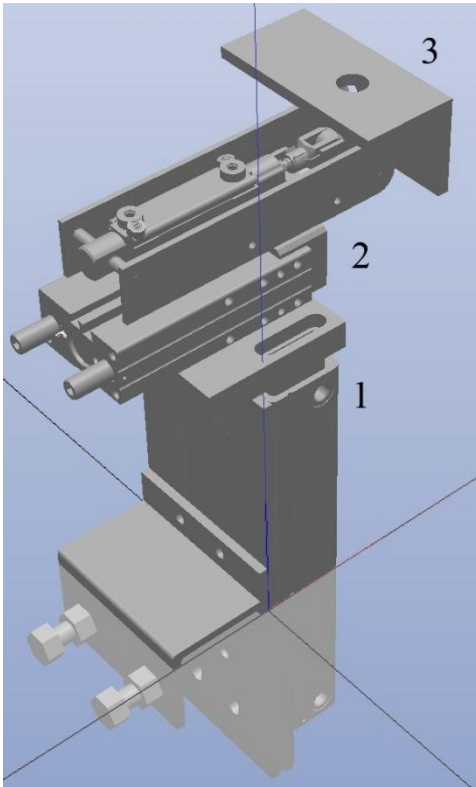


Figure 14.1: Fixture in start position

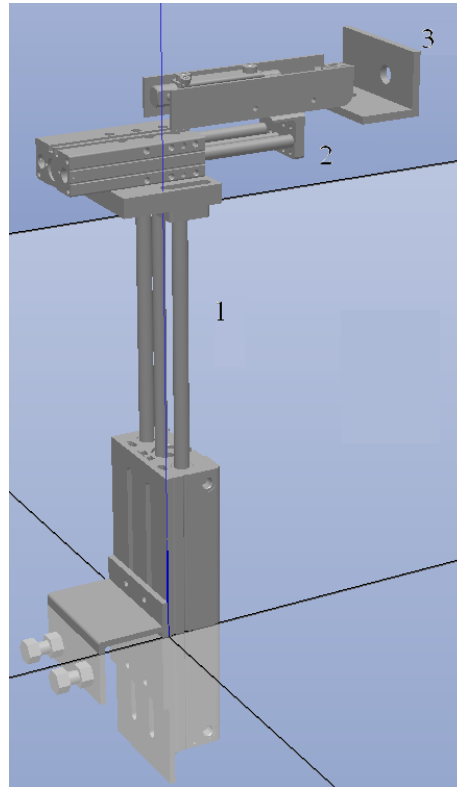


Figure 14.2: Fixture in end position

Define the movements

After all the parts are imported and placed, with the right relationship to each other, the work can begin with creating movements. In RobotStudio be sure that Layout view is selected. There are three views, Layout, Modeling and Programming.

Step 1 – Create Mechanism

From the Create menu, select “Create Mechanism”.
As “Mechanism Type” select “Device”.

Step 2 – Create Link

Right click on Links and select “Add Link”. First a base part has to be selected. Select the base part from “Selected Part” and also check “Set as Baselink”, see Figure 14.3. Confirm the selection by pressing the green arrow. When this is performed, select “Apply”.

Now the first link is created and to create the rest of the links follow the same method, to get a result like in Figure 14.4. If the parts of the fixture are imported separate, parts that are connected has to be added to the same link.

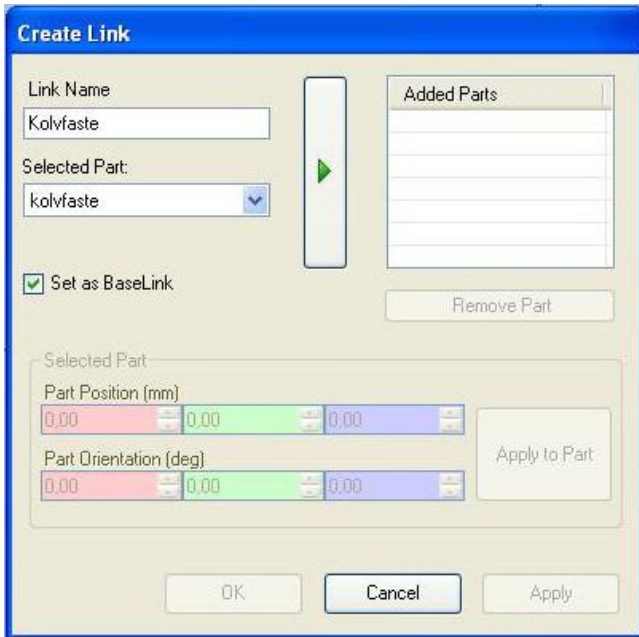


Figure 14.3: Create links

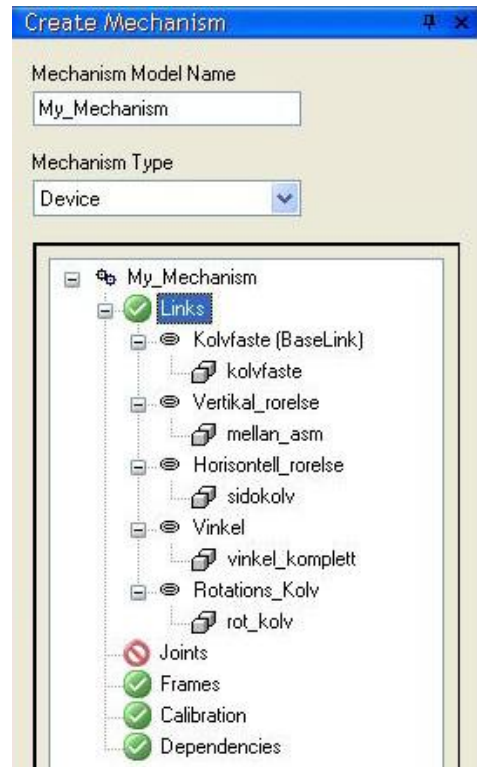


Figure 14.4: All links created

Step 3 – Define movements

Right click on "Joints", in the "Create Mechanism" dialog box, and select "Add joint".

First the vertical movement is defined by changing "Joint Type" to "Prismatic".

This because to get a straight movement. The part that was selected as base link in step 2 is automatically selected as "Parent Link". The part to select as "Child Link" is the part that is going to move towards the base link. The positions define where start and end positions are for the movement, see Figure 14.5. When the vertical movement is defined, the same method is used to create the horizontal.

For the rotating movement the method is a bit different. To create a rotating movement the "Joint Type" has to be "Rotational", see Figure 14.6. Thereafter the axis, which the part is going to be rotating around, and the size of the angle has to be defined.

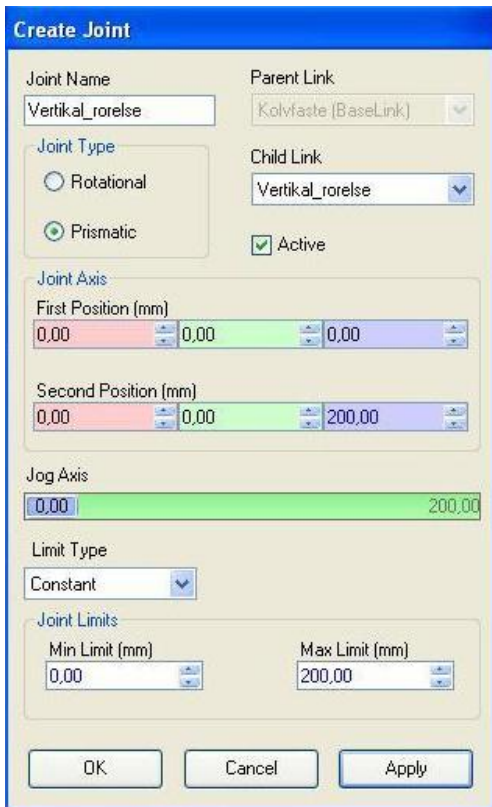


Figure 14.5: Create the vertical movement



Figure 14.6: Create the rotating movement

Step 4 – Compile the mechanism

When Links and Joints are defined, select "Compile Mechanism". When the compilation is performed the different parts are integrated in the mechanism, see Figure 14.7 and 14.8. If there is a desire to get access to a part after the compilation, the mechanism has to be modified and the link to the specific part has to be deleted.

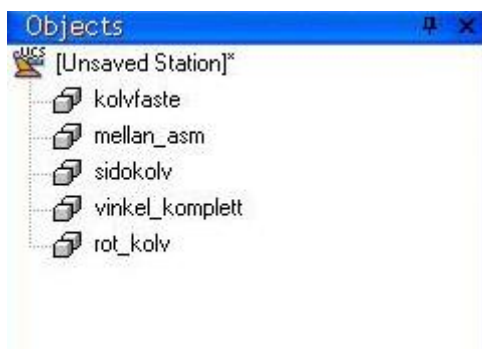


Figure 14.7: Parts before compile



Figure 14.8: Parts after compile

Step 5 – Create pose for movement

After the compilation a dialog box will appear where the poses of movement are defined.

A “Home Pose” has to be defined by check “Home Pose”, see Figure 14.9. Give every pose a name and define the movement. SyncPose and HomePose has to be defined in the system, otherwise the system will not work correctly.

Create Pose

Pose Name:

Home Pose

Launch Joint Jog Tool

Use Current

Reset Values

Joint Values

Horisontell_ror: 0,00

Vertikal_rorels: 0,00

Vinkel (deg): 0,00

Rotations_kolv: 0,00

0,00

0,00

OK Cancel Apply

Figure 14.9: Definition of Home Pose

Create Pose

Pose Name:

Home Pose

Launch Joint Jog Tool

Use Current

Reset Values

Joint Values

Horisontell_ror: 200,00

Vertikal_rorels: 75,00

Vinkel (deg): -90,00

Rotations_kolv: 25

0,00

0,00

OK Cancel Apply

Figure 14.10: Definition of rotation movement

Step 6 – Define the movement times.

The time to perform a movement can be defined in “Set Transition Times”, see Figure 14.12.

Set Transition Times

Transition Times (s)

From Pose:

To Pose:	SyncPose	HomePose	Vertikal_uppe	Horisontell_ute	Vinkel_nere
SyncPose	0,000	0,000	0,000	0,000	0,000
HomePose	0,000	0,000	3,000	0,000	0,000
Vertikal_upp	0,000	3,000	0,000	3,000	0,000
Horisontell_l	0,000	0,000	3,000	0,000	3,000
Vinkel_nere	0,000	0,000	0,000	3,000	0,000

OK Cancel

Figure 14.12: Definition of how long time it takes between two poses

Step 7 – Movement control

By right clicking on the created mechanism and select "Mechanism Joint Jog", the panel in Figure 14.13 can be used to verify the movements.



Figure 14.13: The panel to verify the movements

If the fixture works as desired, it is now time to program the fixture so the movements can be controlled by signals.

Programming

The desired movements can be programmed in different ways. The following suggestion shows a solution that will work on this fixture, but there are more solutions for how to do this.

Step 1 – Create Signals

From the Controller menu, select "I/O Simulator".

Select "Edit Signals" and create signals for every pose the fixture can be at. For example up, down and angle_down.

Step 2 – Define the action to be performed when a signal change

From the Simulation menu, select "Event Manager". To add a signal, select "Add".

In the "Create New Event" dialog box, select "On" from the "Activation" options bar and "I/O signals changed" from the "Event Trig Type". Select "Next" when this is performed.

Select a signal and check that the signal will trigger on True ("1"), then press "Next".

Now it is time to define the type of action. "Set Action Type" to "Move Mechanism to Pose" and proceed by pressing "Next".

Select the mechanism which has been created and select the Pose the mechanism will move to. Then press "Finish".

The first movement has now been created and to define the remaining movements press "Add" and repeat the method explained above.

Step 3 – Create one Start and one Return signal

For the signals “Start” and “Return”, select “Change I/O” as the “Set Action Type”.

Now it is time to define the series of event that will be executed on a signal change on the “Start” or “Return” signal. The chain of event is created through that one event is activated when another is performed. This is performed by selecting one created signal and in “Station signal to set when Pose reached” select a other signal to set when the pose is reached, nr. 1 in Figure 14.14. Do this method for all the signals.

When the last signal, in the “Start” or “Return” movement chain, is defined it can be a good idea to reset all the used signals. When the signal is selected, press “Add Action” and select “Change I/O”. Select one signal that has been used in the movement chain and set the value to “False”, nr. 2 in Figure 14.14.

When this method has been performed for both signals, a mechanism that can be controlled by two signals has been created.

The screenshot displays the 'Event Manager' window with a table of events and two detailed configuration panels.

Activation	Trigger Type	Trigger System	Trigger Name	Trigger Parameter	Action Type	Action System	Action Name	Action Parameter
Dn	I/O	Station Signals	Upp	1	Move Mech...		Move Mechanism to P...	My_Mechanism : V
Dn	I/O	Station Signals	Ner	1	Move Mech...		Move Mechanism to P...	My_Mechanism : H
Dn	I/O	Station Signals	Ut	1	Move Mech...		Move Mechanism to P...	My_Mechanism : H
Dn	I/O	Station Signals	In	1	Move Mech...		Move Mechanism to P...	My_Mechanism : V
Dn	I/O	Station Signals	Vinkel_nere	1	Move Mech...		Move Mechanism to P...	My_Mechanism : V
Dn	I/O	Station Signals	Vinkel_uppe	1	Move Mech...		Move Mechanism to P...	My_Mechanism : H
Dn	I/O	Station Signals	Starta	1	Change I/O	Station Signals	Upp	1
Dn	I/O	Station Signals	Återgå	1	Change I/O	Station Signals	Vinkel_uppe	1

Trigger: I/O Signal Trigger

Activation: Dn

Comments:

Signal Name | Signal Type

In	
Ner	
Starta	
Upp	
Ut	
Vinkel_nere	
Vinkel_uppe	
Återgå	

Signal Source: Station Signals

Trigger Condition:

Signal is true ('1')

Signal is false ('0')

Action: Move Mechanism to Pose

Added Actions:

Seq. No	Action
1	Move Mechanism to Pose

Cyclic

Mechanism: My_Mechanism

Pose: Vertikal_uppe

Station signal to set when Pose reached:

Name	Type
In	Digital
Ner	Digital
Starta	Digital
Upp	Digital
Ut	Digital
Vinkel_nere	Digital
Vinkel_uppe	Digital
Återgå	Digital

1

Apply Close

Figure 14.14: Create a chain of movements

15. How to load a BoxSweeper PLC program to the PLC

The computer we have used to communicate with the PLC is named *s2pc021*. On this computer the necessary components to be able to communicate with the PLC are installed. If another computer is being used, read Appendix E in the Bachelor Thesis 2007 (SSYX02) from the group PLC A. How to setup another computer correctly is explained in the appendix.

1. Start the Simatic Manager.
2. On the “File” menu, select “Retrieve”.
3. Select the path where the basic program, named “BoxSweeperPLC_program” is stored and then press “OK”. Contact the department of Signals and Systems at Chalmers University of Technology for further details on the basic program.
4. Select the path where the basic program is going to be stored and then press “OK”.
5. Select “Yes” to open the basic program.
6. Then expand the file tree like in Figure 15.1.

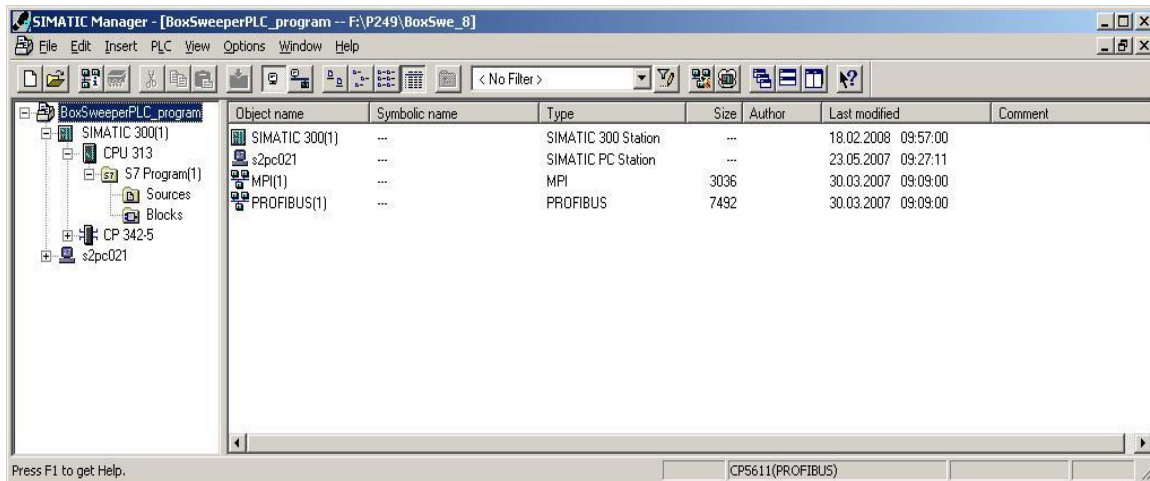


Figure 15.1: Simatic Manager

7. Select the folder “Sources” in the folder “S7 Program”.
8. Open the file “BoxSweeper”.

It is now time to copy the generated PLC code to the basic program, go to step 9.

9. Open the SCL file which was created in RobotStudio.
10. Select all text and copy this to the file in step 8.
11. Press the compile button, nr. 1 in Figure 15.2. Select “All” to overwrite existing blocks and then close the SCL-editor.

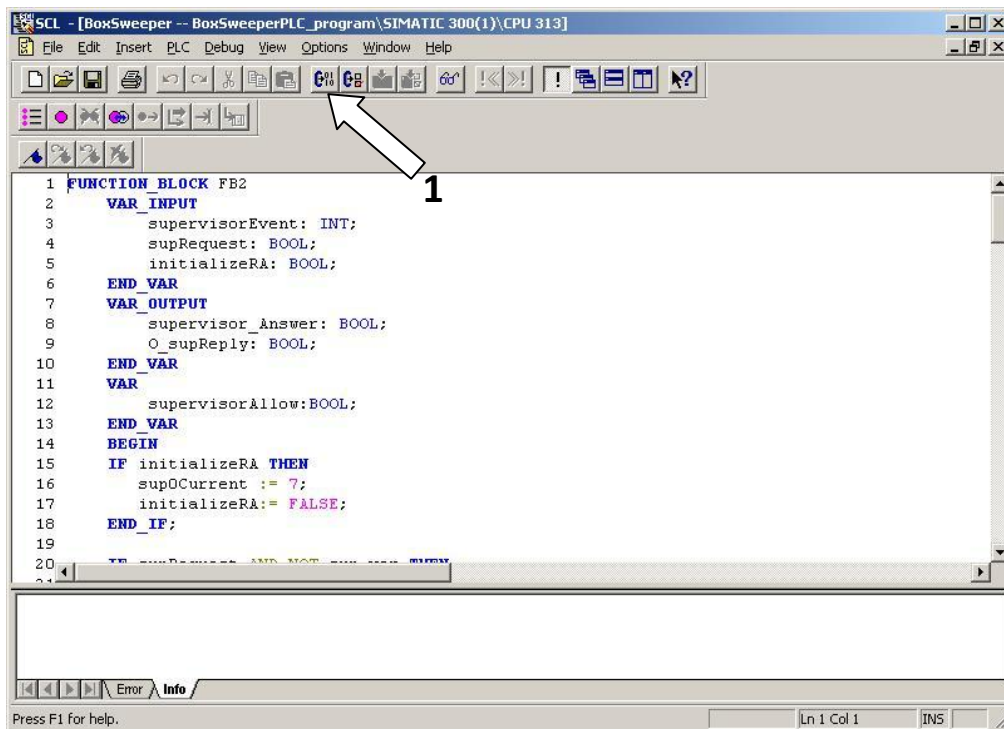


Figure 15.2: The SCL editor

Now BoxSweeper PLC is ready to work and it is time to load the program into the PLC.

12. Select the “SIMATIC 300” and then press “Download” on the PLC menu.
13. Select “Yes” to download the HW configuration.
14. If the Dialog box “Insert Organization Block” appears, select “All” to overwrite the existing blocks.
15. Select “Yes” to restart the module.

16. How to fix the Profibus communication with the PLC

If a problem occurs when using Profibus to download a program to the PLC, the best way to solve this is to use the MPI connection, see Figure 16.1. Connect the MPI adapter to the PLC CPU and to the USB on the computer. To setup the connection do the following steps:

1. In the Simatic Manager, select PG/PC Interface in the Options menu.
2. Select the "PC Adapter(MPI)" and press "Ok".
3. Press "Ok" when the next dialog box appears.
4. Download the "BoxSweeper PLC" basic program by following the steps 12-15 in chapter "How to load a BoxSweeper PLC program to the PLC".

Hopefully the connection has now been fixed and the Profibus can be used again. To change the communication back to the Profibus do the following steps:

1. In the Simatic Manager, select PG/PC Interface from the Options menu.
2. Select the "CP5611(PROFIBUS)" and press "Ok".
3. Press "Ok" when the next dialog box appears.



Figure 16.1: The MPI adapter

Appendix F – Symbol Table

The symbol table used to specify the variables, function blocks (FB), data blocks (DB) and signals for BoxSweeper PLC.

Symbol	Adress	Data type	Comment
KommunikationDB	DB 1	FB 1	
SupervisorDB	DB 2	FB 2	
sup0isReadyDB3	DB 3	FB 3	
TaemotPro	DB 40	DB 40	Recive data on Profibus
SkickaPro	DB 41	DB 41	Variabels to send over Profibus
DB_SEND	DB 70	DB 70	DB for communication to robot controler
DB_RECV	DB 71	DB 71	DB for communication from robot controller
Kommunikationsblock	FB 1	FB 1	
Supervisor	FB 2	FB 2	
sup0isReady	FB 3	FB 3	
DP_SEND	FC 70	FC 70	DP SEND Profibus
DP_RECV	FC 71	FC 71	DP RECEIVE Profibus
sup0Current	IW 101	INT	
sup0Target	IW 103	INT	
sup0Cols	IW 105	INT	
sup0Length	IW 107	INT	
supervisorEvent	IW 109	INT	
GI_Event	IW 110	INT	The event recieved from robot controller
oneTime	M 100.0	BOOL	
supervisorAllow	M 100.2	BOOL	
isReady_Output	M 100.3	BOOL	
I_supRequest	M 111.0	BOOL	A robot has an event to evaluate
initialize	M 111.1	BOOL	
I_Semaphore	M 111.2	BOOL	
O_supervisorAnswer	M 112.0	BOOL	The answer from the supervisor
O_supReply	M 112.1	BOOL	
GO_Event	QW 110	INT	Mirrored signal of GI_Event

Table 1: Signals, DB, FB and variables to be able to run BoxSweeper PLC

Appendix G – Function block for communication

The function block for communication in BoxSweeper PLC

Network 1: Receives data from the Profibus	
CALL "DP_RECV"	
CPLADDR := W#16#130	
RECV := P#DB71.DBX4.0 BYTE 2	
NDR := DB71.DBX2.0	
ERROR := DB71.DBX2.1	
STATUS := "TaemotPro".STATUS	
DPSTATUS := DB71.DBB3	
L	"DB_RECV".GI_Event
T	"GI_Event"
L	DB71.DBB 5
T	MB 111
Network 2: The call for the evaluation of the input data	
CALL "Supervisor" , "SupervisorDB"	
supervisorEvent := "GI_Event"	
supRequest := "I_supRequest"	
initialize := "initialize"	
supervisorAnswer := "O_supervisorAnswer"	
O_supReply := "O_supReply"	
Network 3: Sends data to the Profibus	
L	"GI_Event"
T	"GO_Event"
L	MB 112
T	DB70.DBB 4
L	"GO_Event"
T	"DB_SEND".GO_Event
CALL "DP_SEND"	
CPLADDR := W#16#130	
SEND := P#DB70.DBX3.0 BYTE 2	
DONE := DB41.DBX8.0	
ERROR := DB41.DBX8.1	
STATUS := "SkickaPro".STATUS	