Technical University of Denmark

Rikke Rold Bateman

Functional Modelling and Monitoring of a Robotic System

Master's thesis, July 2016

Functional Modelling and Monitoring of a Robotic System
Funktionsorienteret Modellering og Overvågning af Robotsystem

Report written by:
Rikke Rold Bateman

Advisor(s):
Haiyan Wu
Morten Lind
Xinxin Zhang
Ole Ravn

DTU Electrical Engineering
Automation and Control
Technical University of Denmark
Elektrovej
Building 326
2800 Kgs. Lyngby
Denmark
Tel : +45 4525 3576

studieadministration@elektro.dtu.dk

Project period: 2016-02-01 – 2016-07-01

ECTS: 30

Education: MSc

Field: Electrical Engineering

Class: Public

Remarks:	This report is submitted as partial fulfilment of the requirements for graduation in the above education at the Technical University of Denmark.
Copyrights:	

Abstract

Most modelling of engineering systems, such as robotic systems, is qualitative, which does not convey the intended purpose of the system. The purpose of this project was to develop a method for the functional modelling of robotic event-based systems, while simultaneously applying this method to a known robotic system. This project drew inspiration from Multilevel Flow Modelling (MFM), and used the theories that MFM were based on to develop the model and the methodology.

In order to demonstrate the performance of the model, a monitoring system was developed. The monitoring system was tested using a simulation of the robotic system, and was able to track the robotic system, as well as detect errors.

The modelling methods were found to be useful for modelling a robotic system, especially for identifying preconditions to actions, which can be used for error monitoring.

Contents

1 Introduction **1**
- 1.1 Background . . . 2
- 1.2 Introduction to Action Types, Hierarchical Models and MFM 3
- 1.3 The Robotic System . . . 3
- 1.4 Model Development and Monitoring Implementation . . . 4

2 Theory **7**
- 2.1 Hierarchical Models . . . 7
- 2.2 Action Types . . . 8
  - 2.2.1 Von Wright's Theory of Action and Elementary Action Types . . . 9
  - 2.2.2 M. Lind Interpretation: Interventions and Omissions . 9
  - 2.2.3 Action Roles . . . 10
- 2.3 Action Phases . . . 11
- 2.4 Multilevel Flow Modeling . . . 13
  - 2.4.1 MFM Library . . . 13
  - 2.4.2 MFM Water Mill Example . . . 13

3 Functional Modelling of the Robotic System **17**
- 3.1 Analysis of the Robotic System . . . 17
- 3.2 Applying Theories to the Robotic System . . . 19
  - 3.2.1 Hierarchical Model . . . 20
  - 3.2.2 Action Types and Action Roles . . . 21
  - 3.2.3 Action Phases . . . 23
- 3.3 Developing the Model . . . 25
  - 3.3.1 Model Methodology . . . 27
  - 3.3.2 Initialization Process States . . . 27
  - 3.3.3 Trigger Process States . . . 28
  - 3.3.4 Circular Process States . . . 28
  - 3.3.5 Exclusion of Information in Model . . . 29

4 Functional Model Based Monitoring System **31**
- 4.1 Monitoring System Layout . . . 32

5 Implementation of Monitoring System **35**
5.1 CLIPS . . . 35
5.2 Robotic System Monitoring Availability . . . 36
5.2.1 Signals Currently Available . . . 36
5.2.2 Unavailable Signals . . . 37
5.3 Simulation Environment: Python and MatLab . . . 37
5.3.1 Simulated Signals . . . 38
5.4 Implementation in CLIPS . . . 39
5.4.1 Rules for Main System Monitoring . . . 40
5.4.2 Rules for Error Monitoring . . . 47

6 Results and Tests **53**
6.1 Methodology Developed . . . 53
6.2 Testing the Monitoring System . . . 54
6.2.1 One Run Scenario . . . 55
6.2.2 Error: Conveyor Belt Off . . . 57
6.2.3 Error: Object Detected, System Not Ready . . . 58
6.2.4 Error: Timeout in Ready-to-Go State . . . 60
6.2.5 All Scenarios Run Sequentially . . . 62

7 Conclusion **65**
7.1 Future Work . . . 65

A Monitoring System Program **67**
A.1 CLIPS Code: monitoring.clp . . . 67
A.2 Python Code: monitoring.py . . . 74
A.3 Output . . . 75
A.3.1 error_messages.txt . . . 75
A.3.2 all_messages.txt . . . 76
A.4 MatLab Code: simulated_robotic_system.m . . . 76

Chapter 1

Introduction

The goal of this project is to contribute to the creation of a methodology for functional modelling of industrial robotic systems. The development of such a methodology is largely theoretical. For balance, this project will also contain an implementation of the developed model to show that it is useful.

The theoretical basis of the project is a concept called Multilevel Flow Modelling. Multilevel Flow Modelling, abbreviated MFM, is used for modelling continuous energy and mass flows and their interactions in complex industrial automated systems[1]. However, MFM was not designed for robotic systems. Robotic systems are often defined by a sequence of actions and this event-based nature cannot be modelled by MFM.

Since MFM was not designed for discrete systems, expanding it to work on robotic systems would be like trying to fit a round peg through a square hole. Instead, this project takes a step back, looking at the theories and concepts that MFM is built on, and using these to design a new modelling methodology for modelling robotic systems with the same features as MFM, but adapted to the robotic domain.

Since MFM has taken decades to develop, it is well beyond the scope of one Master's Project to create a fully functional modelling language, even using the same building blocks used to create MFM. Therefore, this is a case study, attempting to apply the semantic concepts used for developing MFM in order to model an industrial robotic system. Thus, the modelling methodology will be simultaneously developed and applied.

The case studied is a robotic system developed for meat production. The purpose of the system is to move meat from a conveyor belt onto meat hooks. Technically, it is a robotic arm in combination with a conveyor belt and an optical sensor.

The theories explored, which will be discussed in Chapter 2, were applied to the robotic system, and from this a model was developed alongside the

methodology. The theory application as well as the model development is discussed in Chapter 3.

The model was then used to create a monitoring system for the robotic system, and the implementation of this is discussed in Chapter 4 and 5, with the results shown in Chapter 6.

1.1 Background

Engineering deals mostly with describing systems and problems quantitatively and building them. Quantitative models do not include purpose or intent of a system, and this information is important for design, operation, and monitoring. Therefore, it is relevant to develop good models that capture this information, for describing the parts of a system that cannot be described mathematically. It is also important to create a methodology for making models that can be understood from one person to another, without having to further explain the model being presented. This makes it possible for one engineer to work on another engineer's project, and simply by updating the corresponding models both will understand what has been changed in the system.

The models developed in this project will be conceptual, since the goal is to develop a functional understanding of a system without variables and formulas.

MFM was developed to model industrial process systems[2], specifically continuous systems with identifiable mass and energy flows. While MFM works well for these types of systems, it was not designed to handle discrete systems such as the robotic system which will be investigated in this project.

A robotic system is often an event based process which moves from one action to another, whereas an industrial process system is generally made up of flows of energy and mass. As an example, imagine a water mill which grinds grain to flour. There will be a constant water flow and a constant flow of grain turning to flour. In a robotic system, like a robotic arm assembling a car, the closest to a flow would be the flow of cars coming through the assembly line, i.e on the level of overall production, and not on the level of the individual assembly actions.

While it was initially considered to expand MFM in order to model discrete as well as a continuous system, this idea was discarded as MFM was not designed to do this. Instead, the idea of this project is to take a step back and attempt to create a similar system as MFM, but for robotic sys-

tems, using the same concepts used for creating MFM in the first place.

1.2 Introduction to Action Types, Hierarchical Models and MFM

MFM is developed from a number of philosophical and semantic ideas, such as hierarchical model theory[3] and action theory[2]. Since the development of MFM is the basis for the development of this methodology, it was considered relevant to apply these theories to the robotic system.

Hierarchical modelling is the widely used practice of arranging a system by levels of importance. Section 2.1 explores this further. Action Theory, discussed in section 2.2, is based on von Wright's theory of action, which identifies basic types of action, useful for categorizing actions and events into fundamental types. Another important part of action theory is Action Roles[4]. This concerns the identification of which roles the entities performing an action hold during the specific action. The concept of Action Phases[5] is helpful for identifying the requirements for an action to complete successfully, and will be discussed in section 2.3.

1.3 The Robotic System

It is important to distinguish between what is meant by "robotic system" and "robot". A robotic system, in this case, is the entire system that the robot operates in, but also that it is limited to. The robotic system thus contains the robot arm, the conveyor belt, the camera, the meat hooks, and the controller that connects the parts. It is limited as the robotic system does not describe what the individual parts, such as the robot arm, could possibly do, but what they do within the system.

The robot system that this case study is based on is a 6-joint robotic arm with a gripper, a camera, and a conveyor belt. It is intended to be part of a slaughterhouse process, where its specific function is for the robotic arm to lift a slab of meat off a conveyor belt and move it to a collection of meat-hooks on a pole, referred to from here on as the christmas tree or CT. The meat hooks will be referred to as a spike or a CT spike.

A picture of the robotic system is shown in Figure 1.1.

An analysis of the robotic system can be found in section 3.1.

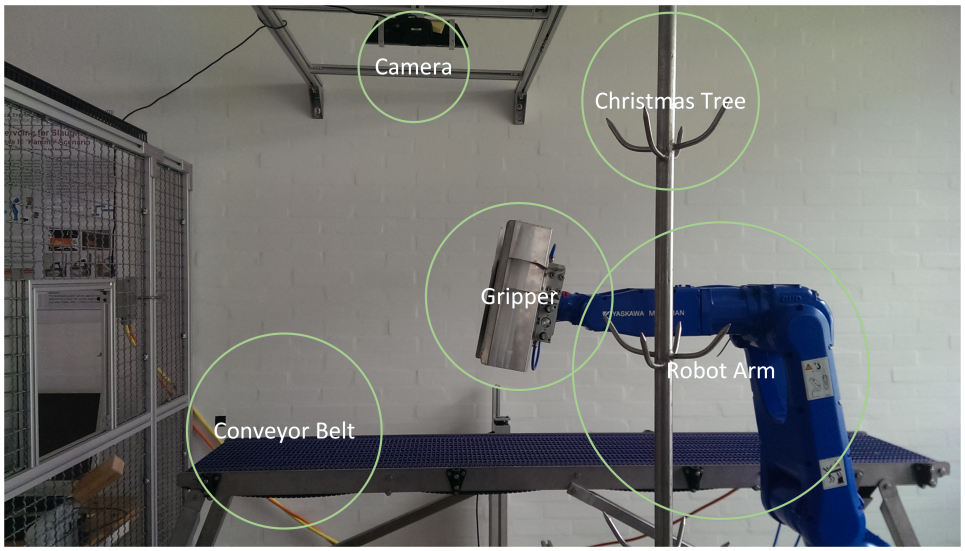


Figure 1.1: The Robotic System

1.4 Model Development and Monitoring Implementation

Initially, the robotic system was analysed, so as to identify the separate events that the system goes through. Then, the modelling theories were applied to the robotic system. A model was later developed based on these theories, of which a simplified version can be seen in Figure 1.2.

Finally the model was used to construct a monitoring system. This was implemented using the expert system language CLIPS[6] and tested using a simulation of the robotic system. Real life testing was not possible because the robotic system had to be handed back to the company that owned it, and was therefore no longer available for the project.

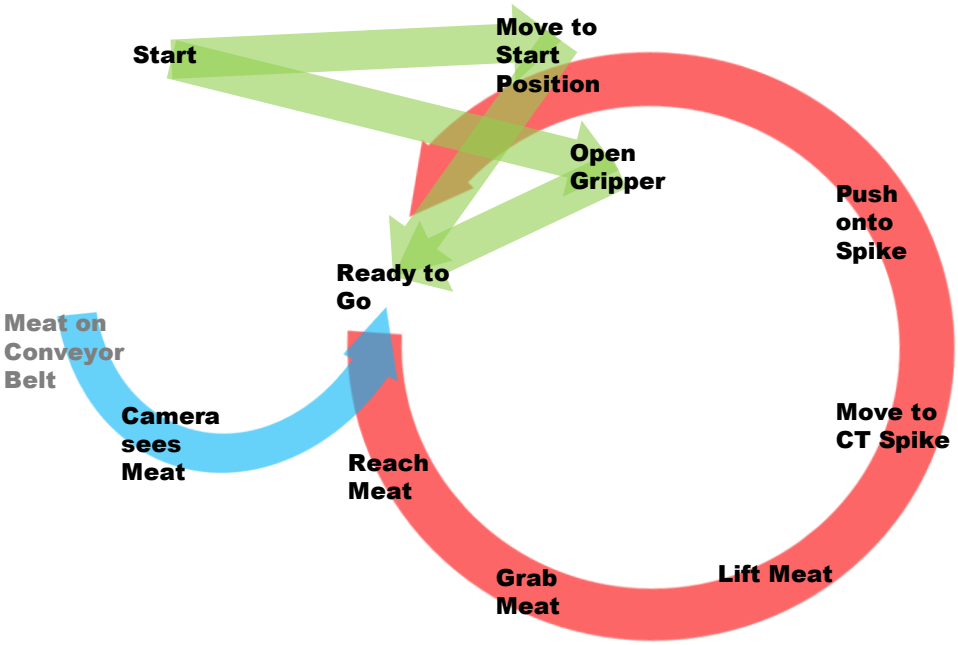


Figure 1.2: Developed Model of Robotic System, Simplified

Chapter 2

Theory

The purpose of this chapter is to introduce the concepts and theories that the project is based on. These theories will be used in Chapter 3 to construct the modelling methodology and the model of the robotic system. This chapter also includes a description of the modelling language MFM, as well as a short example of its application.

It is worth mentioning that the functional modelling of robotic systems is a fairly unexplored topic, and as such there are no text books describing these theories readily available. Therefore, the theories that this project is built on are taken from lecture notes and articles, much of which is unfortunately unpublished course material.

The theories that will be explored are Hierarchical Modelling[7], Action Types[4] and Action Phases[8].

2.1 Hierarchical Models

The theory of hierarchical modelling is taken from on[7].

A Hierarchical model is a collection of interconnected nodes separated into levels of abstraction. The highest level nodes can be connected to multiple lower nodes, but lower nodes are only connected to a single higher node.

There are many different types of hierarchical models. For example:

- Part-whole hierarchies, in which subordinate nodes are the parts that make up their immediate superior node. For example, *lightbulb* and *lampshade* would both be subordinate to *lamp*.

- Command hierarchies where the top nodes can command lower nodes to action, such as in control hierarchies or military organization.

- Means-end hierarchies where a node signifies an action or event, and the subordinate nodes connected to it are the means for accomplishing that action. Figure 2.1 shows an example of a means-end hierarchy, which is taken from [7].

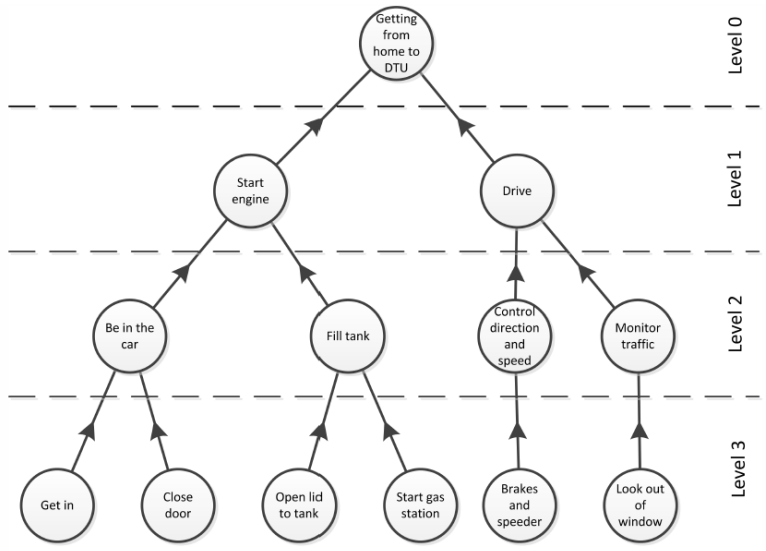


Figure 2.1: Means-end hierarchy example from [7]

A hierarchical model can illustrate complicated ideas in an approachable manner and give an understanding of the structure of a system.

A hierarchical model is useful for gaining an overview of the structure of a system, but not a thorough understanding, as communication between levels of abstraction is not included in the concept of a hierarchical model. This can be remedied by using annotations, however adding too much writing to a model could obscure its meaning.

2.2 Action Types

This section deals with the theory of action types, which is the idea that all actions can be defined as belonging to a more overall group of actions. For instance, pushing a glass or lifting a table could both be classified as moving an object.

Von Wright's theory of action[4] is such an analysis, and has identified four basic types of action. M. Lind has expanded on this theory and identified eight types, by including the role that intervention and omission plays.

Action Roles describe how the entities that are part of an action are participating in said action.

2.2.1 Von Wright's Theory of Action and Elementary Action Types

Von Wright's theory of action asserts that there are four basic types of action: happen, remain, disappear, and remain absent. These are illustrated using a pTp schema[4], where T is the action or change, and p is the object or event that this action or change affects. This schema can be observed in table 2.2.1.

Change Schema	Description
∼pTp	p happens
pTp	p remains
pT∼p	p disappears
∼pT∼p	p remains absent

Table 2.1: von Wright's Basic Action Types

For example, imagine p being an apple. Eating the apple would be pT∼p, as the apple disappears, where as putting an apple on a plate could be either ∼pTp since the apple is in a place it was not before, or pT∼p since the apple is no longer where it was.

2.2.2 M. Lind Interpretation: Interventions and Omissions

Morten Lind has expanded upon these, creating eight Action Types by introducing interventions and Omissions. This is shown in Table 2.2.

Omissions can be seen as the act of letting something happen, and in that way the basic action types are altered. Interventions seek to change the outcome, which is how the four action types *produce*, *maintain*, *destroy*, and *suppress* were created[4].

This can be illustrated by using the same pTp schema as in Table 2.2 and including the I in the outcome of the action. The square brackets illustrate what change would have happened without the intervention or omission. The p on the left side of the I represent what happens due to the intervention or omission, and the p on the right side of the I represents what would have happened without.

As such, *I* can be read as "instead of". Then, the line pT[∼pIp] (destroy) reads as "p changes to [not-p instead of p]".

Change Schema	Description
∼pT[pI∼p]	let p happens
pT[pI∼p]	let p remains
pT[∼pIp]	let p disappears
∼pT[∼pIp]	let p remains absent
∼pT[pI∼p]	produce p
pT[pI∼p]	maintain p
pT[∼pIp]	destroy p
∼pT[∼pIp]	suppress p

Table 2.2: Intervention and Omission Action Types

As an example, think of an apple falling from a tree. This would be pT[∼pIp] (let disappear) in relation to the branch the apple fell from. However, if a person placed their hand under the apple, it could be read as pT[pI∼p], as the apple would not disappear, i.e would be maintained on the branch.

If an apple was plucked from the tree, it would be thought of as pT[∼pIp], since the apple would have remained on the branch without the intervention, but because of the intervention the concept of the apple being on the branch was destroyed.

The four intervention action types are serving either a promotive function(produce, maintain) or an opposive function (destroy, suppress).

2.2.3 Action Roles

For each action, the entities that are engaged in the action can have one of four different roles[8]. The most important roles are the Agent and the Object. The action roles will be capitalized for easy identification.

- The **Agent** is the entity performing the action.
- The **Object** is the entity that this action is being performed upon.
- The **Helper** assists the agent in performing the task.
- The **Counter-agent** is the entity working against the action taking place.

In a case such as an object being lifted, the Agent would be the entity that does the upwards motion, the Object would be the entity being lifted,

the Helper could be a mechanism holding on to the Object, such as glue, and the Counter-agent is just gravity.

While the Agent and Object are always present in an action, a Counter-agent and Helper are not always present or easily identifiable.

In a sequence of events, the roles may be changed from one action to another. That is, the Agent of the initial action might be the Object, Helper, or Counter-agent of the second action.

2.3 Action Phases

Action Phases describe the conditions for a successful action, and are provided in Figure 2.2[5].

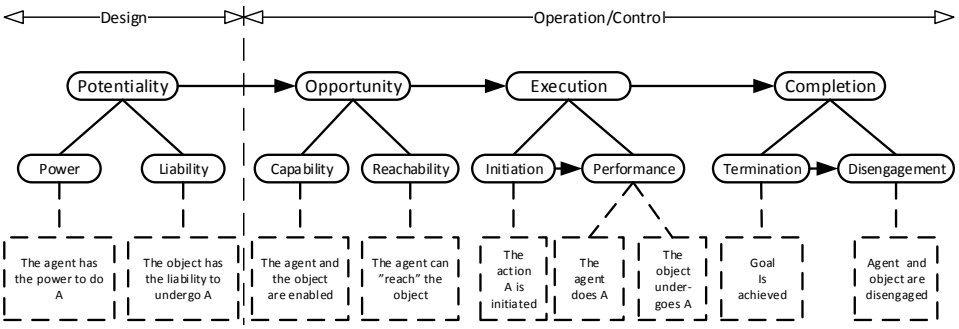


Figure 2.2: Action Phases

Each Action Type contains its own action phases that need to be fulfilled in order to complete the given action.

Potentiality describes the capability that the system should have for completing the desired tasks. During design, the potentiality of an action should be ensured. As can be seen from the figure, potentiality is the only field under Design, where the remaining fields can be grouped under Operation. It is different from Opportunity in that the system must be designed with the ability to complete said task, whereas Opportunity is the possibility of the task being completed due to circumstances. Examples of potentiality can be the strength of a robotic arm being large enough to lift an object, a camera being able to detect infra-red light, or a person having the knowledge to operate a piece of machinery.

Potentiality is achieved by ensuring that the object has the liability[9] to undergo the event (undergo change), and the agent has the power to perform the event (perform change). In this way, liability and power are two sides of the same coin, and one cannot exist without the other. For example, a hammer smashing a plate[9]. For this action to be possible, not only must

the hammer be strong enough to smash the plate, the plate must also be fragile, i.e have the liability, to be smashed. Another example could be a robot having the strength to bend a metal object of a certain size, but if the object is made of wood it is unbendable, and thus the object does not have the liability to be bent, just as the robot does not have the power to bend it.

Opportunity is ensured by ensuring the capability and reachability of an agent and object. Both need to be "enabled" and the agent also needs to be physically able to complete its task. Considering a robotic arm, it could be strong enough to lift the object it is the agent of, thereby having the potentiality for completing the action, but if it is in another room than the object (and unable to transport itself from that room to the object) it lacks the opportunity to complete the action. Other examples of opportunity are, using the previous examples, that the camera is turned on, and turned the correct direction, and that the person is awake and knows where the machinery that needs operating is placed.

Execution is a two-part process that begins with initiation and is completed by the task being performed. In some cases execution can be simply the triggering of an event, in other cases the performing part is more cumbersome, such as in maintaining a situation for an extended period of time.

An action can be initiated by the flip of a switch, or in the case of a person performing a jump, the initiation would be the straightening of the legs from a crouching position, which would result in upwards motion of the body. The performing part in this case would be the follow-through, of continuing to tighten the muscles to propel oneself upwards. Another example could be in the case of a maintenance action, where the performance portion of the action would have a much longer duration.

Completion is achieved when the goal of the action is achieved. This could be a short instance in time such as a picture being taken and saved to the camera memory, or it could be a much longer time, such as in a maintenance case where completion would be an ongoing process.

Action Phase theory ties in with the concept of action roles, since the roles are the entities which ensure that all the phases of the action can be completed. The role of counter-agent is important to consider in the design phase of a system, as it could be used to ensure potentiality. For instance, if the counter-agent is gravity, potentiality is ensured by having an agent with the strength to counter the force of gravity.

2.4 Multilevel Flow Modeling

The point of this section is to give a brief explanation of MFM, in order to give the reader an understanding of what theories the main project is based on.

Multilevel Flow Modeling, abbreviated MFM, is a tool developed mainly by M. Lind[1], and is used for describing industrial systems using means-end and part-whole abstractions. It has been utilized successfully to describe industrial situations. However, when MFM is applied to robotics, a number of inadequacies appear. This is mainly due to the fact that MFM was developed for continuous systems, and most, if not all, robotic systems are event-based.

Following is the MFM library and an example of how MFM can be used to model a water mill.

2.4.1 MFM Library

Figure 2.3 shows the MFM library of symbols. These are provided as reference for the following water mill example.

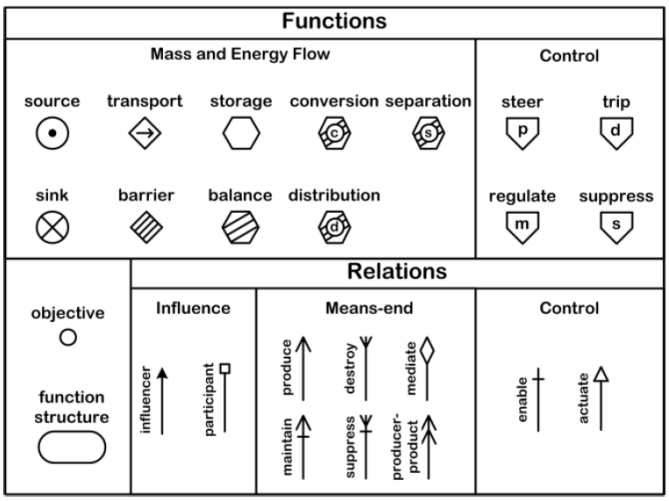


Figure 2.3: MFM Library

2.4.2 MFM Water Mill Example

This example is from [1].

The water mill is shown in Figure 2.4. The water mill works by letting water run over the main water wheel. This turns the drive shaft which causes the runner stone to be turned. The grain is funnelled into the middle of the two stones, and when the stone is turned, the grain is ground into flour (and shells).

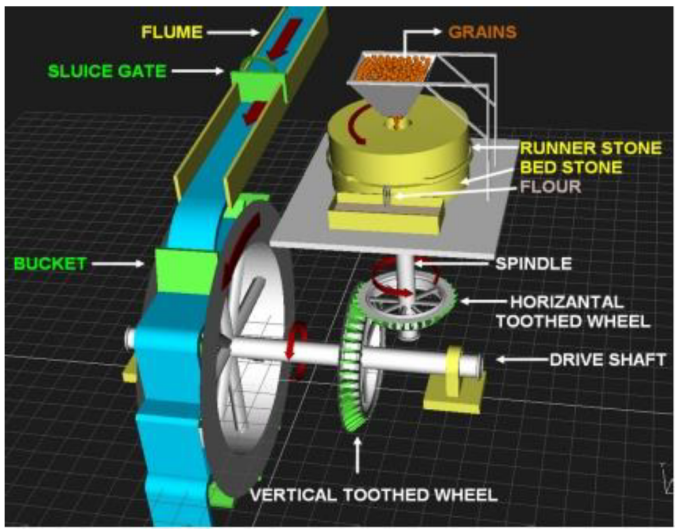


Figure 2.4: Grinding Mill

The MFM model of the water mill is shown in Figure 2.5. As can be seen, it is separated into three function structures, S1, S2 and S3.

S1 describes the flow of grain. From a storage (s01) the grain goes (tr1) into the wheel (st1), and is ground (tr2) and seperated (bl1) into shells and flour, which are transported (tr3, tr4) out of the system (si1, si2)

S2 describes the flow of energy in the system, specifically the kinetic energy of the water flow (tr5) to both (bl2) heat produced by friction(tr7, si4) and rotational energy of the grinding stone (tr6, si3).

S3 describes the flow of water. The water from the river (so3) flows (tr8) into the flume (st2), through the sluice gate (tr9), into the bucket (st3), and out (tr10) of the wheel (si5).

O1 is the goal of the system, which is the flour produed in S1 and transported in tr3. tr6 connects the rotational energy in S2 to the grinding stone in S1. tr10 shows the connection between the water and the kinetic energy (so2).

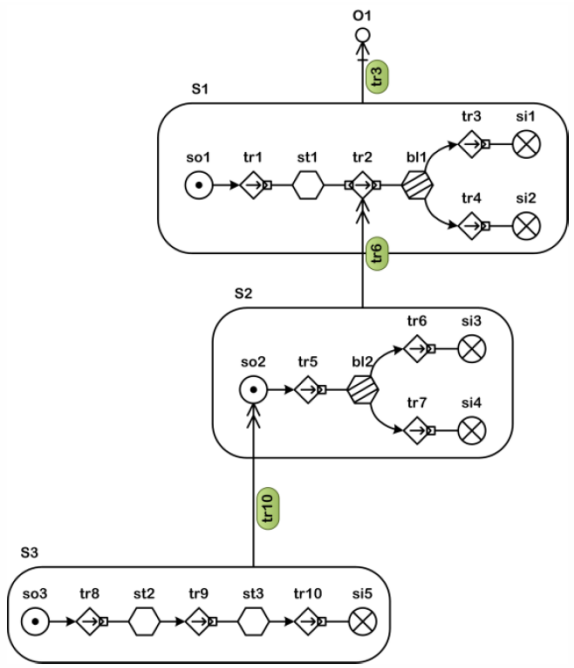


Figure 2.5: Grinding Mill in MFM

Chapter 3

Functional Modelling of the Robotic System

This chapter describes the methods used to model the robotic system. The first part describes the various approaches to describing the robotic System using the theories introduced in Chapter 2. The final part of this chapter describes the model developed.

No model can completely describe a system, whether mathematical or semantic in nature. A model will always be an approximation of the physical system, and various elements will have to be either simplified or ignored. The question is then: what is acceptable in an incomplete model? The purpose of this chapter is to investigate this.

The models developed using the different theories are all included in this chapter as a way to show the process of the methodology development.

3.1 Analysis of the Robotic System

This is simultaneously an overall description of the robotic system and an elementary analysis, and is provided here to give the reader a more thorough understanding of the robotic system. This was done as one of the first things in the project, before the theories were applied, in order to gain an understanding of the system.

For easy identification, the picture of the robotic system from figure 1.1 is repeated here. This image shows the parts that make up the robotic system. They will throughout the report be referred to as "components" and are: the camera, the conveyor belt, the gripper, the robot arm, and the christmas tree.

Not shown on this image is the controller. The controller communicates with the control box of the robot arm, the camera and the gripper. The conveyor belt can be turned on/off manually through a standalone control box.

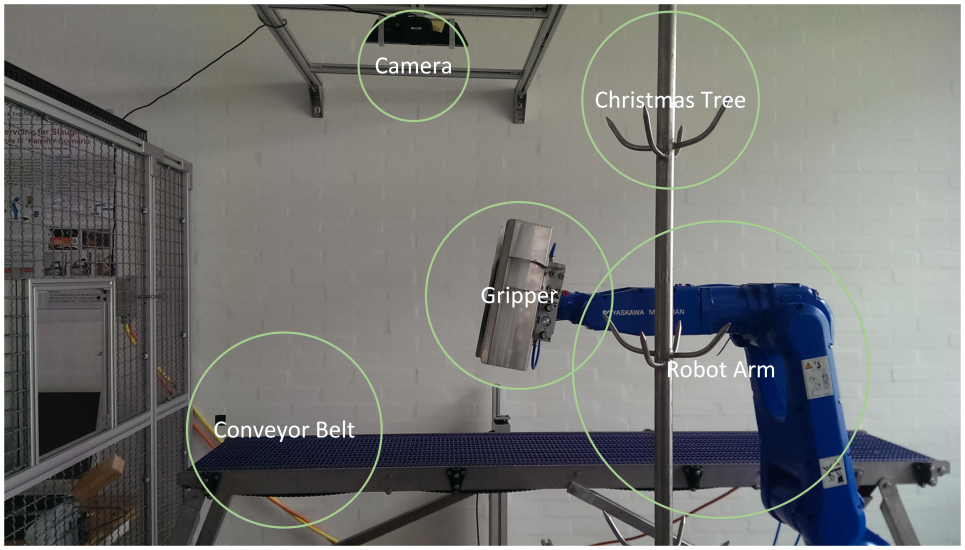


Figure 3.1: The Robotic System (again)

The signals submitted to the controller from the robotic system in its current state are:

- The position of the gripper.
- The distance between the gripper fingers
- Whether there is an object between the gripper fingers or not. This signal is only available when the gripper fingers are moving.
- Whether the conveyor belt is on or off.
- An image sent from the camera every 30 ms, which is analysed by the computer the controller is on.

This means that some preconditions for events must be assumed, as they cannot be measured. For instance, since the gripper cannot register an object between the gripper fingers when the fingers are not moving, it must be assumed that the meat stays in the gripper when it is lifted off the conveyor belt.

The controller will not be discussed much outside of this section, as this project is an analysis based on actions, where there is an agent and an object of each action, and the controller fits neither of these. It could be reasoned that, since the controller tells the robot arm to lift the meat, the controller is the agent of this action, as opposed to the robot arm being the agent. However, this would mean that technically the programmer is the agent, as the programmer tells the controller to tell the robot arm to lift the object. Or, when taken to extremes, the programmer's boss is the agent. This is why the role of agent will be the "closest" agent in relation to the action.

Initially, the system is started by initializing communication with the components and running the code that controls the system. Then, the robot arm moves to the start position, the gripper is opened, and a timer is started, which runs for approximately 1.2 seconds. Before the timer has run out, the robot arm will not reach for any objects detected. The timer is to ensure that the robot arm has reached the start position, and that the gripper is open.

Once the timer has finished, meat can be placed on the conveyor belt. The image processing algorithm detects the meat as soon as the meat appears in the camera's field of vision. Then, the position and orientation of the meat will be sent to the controller. With this information the controller calculates the path from the start position to the position where it has to grab the meat. The robot arm is then triggered to reach for the meat, and the gripper is closed to grab the meat. As soon as the meat is in the gripper, the robot arm will lift the meat off the conveyor belt.

Having lifted the meat off the conveyor belt, the robot arm moves the meat to a spike on the christmas tree, and pushes the gripper down, impaling the meat on the spike. Once the gripper has reached the base of the spike, the gripper opens. The sensor inside the gripper checks if the meat is on the spike when the gripper is being opened.

Finally, the robot arm moves back to the start position, ready to move if another piece of meat is detected. The timer is started again when the robot arm starts moving back to the start position.

3.2 Applying Theories to the Robotic System

This section describes the application of the theories introduced in Chapter 2 to the robotic system.

First, the system is modelled using hierarchical modelling. This type of model is useful for gaining an overview and are generally easy to un-

derstand. Action type theory was then applied, specifically Action Roles, which gave more insight into the components of the system and the tasks they perform. Finally, Action Phase theory was utilized, in order to gain a thorough understanding of the identified actions.

3.2.1 Hierarchical Model

Of the different types of hierarchies introduced, a means-end hierarchy was used to model the system, where the levels of abstraction were, in descending order: tasks, activities, actions, roles, and preconditions. This is the first model developed during this project and is shown in Figure 3.2.

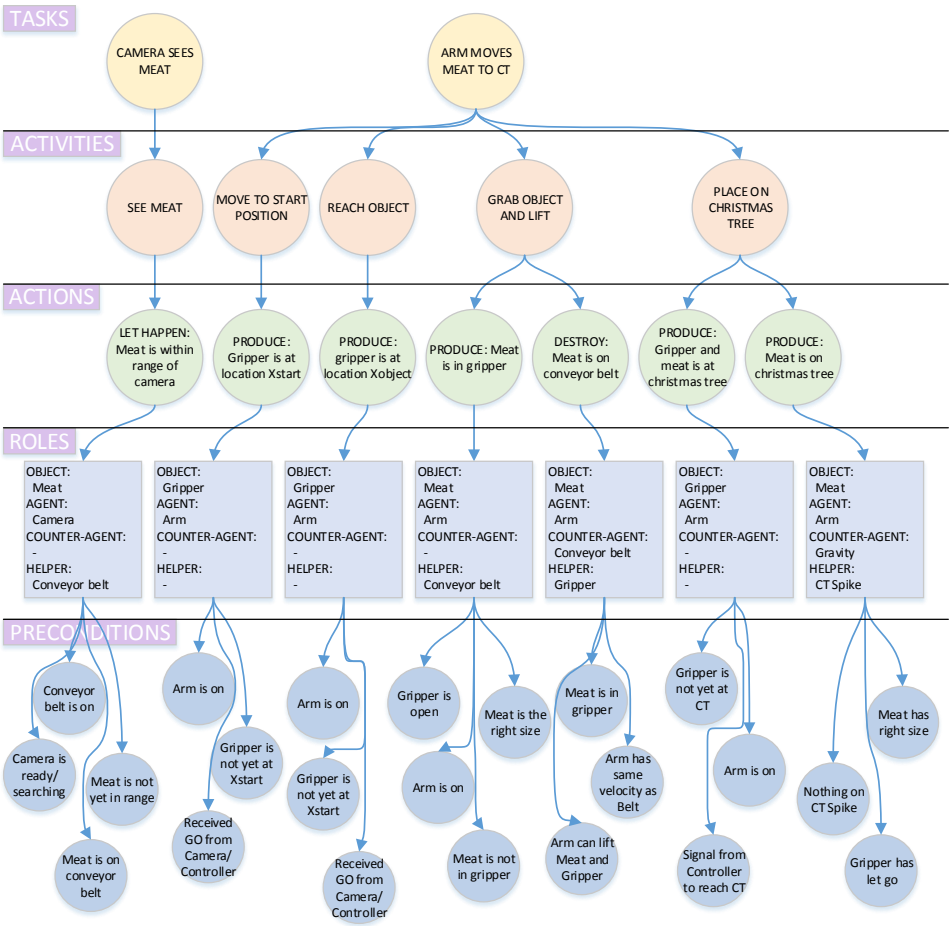


Figure 3.2: Hierarchical Means-End Model of the Robotic System

- **Tasks** describe the overall tasks that the system seeks to achieve.

- **Activities** are the activities that are needed to complete the tasks. They are separated by a significant time gap. This is the reason grab and lift are one activity, as they happen almost simultaneously. It is worth noting that the Activities level takes on the form of a sequence of events starting with the camera seeing the meat and ending with the meat on the Christmas Tree.
- **Actions** are the basic action types that make up the activities.
- **Roles** are the corresponding action roles for each action.
- **Preconditions** are the conditions that must be met for the system components to complete the action.

This model helped in identifying the separate events as well as the preconditions for these events to be successful.

A hierarchical model of a system is useful for understanding how the system works, but as discussed in the Theory chapter, it is also incomplete, as there is not much room for explaining the jumps between levels of abstraction and no way of showing communication between nodes of similar levels.

3.2.2 Action Types and Action Roles

Action Types

Using the basic action types described in section 2.2 table 2.2, it is possible to describe the robotic system events. This was already done in Section 3.2.1 with the hierarchical model.

- Let happen: Meat is seen by the camera.
- Produce: Gripper is at the start position.
- Produce: Gripper is at the position of the meat.
- Produce: Meat is in the gripper.
- Destroy: Meat is on the conveyor belt.
- Produce: Gripper and meat is at the christmas tree spike point.
- Produce: Meat is on the christmas tree spike.

Action Roles

The action role analysis looks at the components and how they act in an event.

In Figure 3.3 this is taken a step further, as it looks at how the components act in the system as a whole.

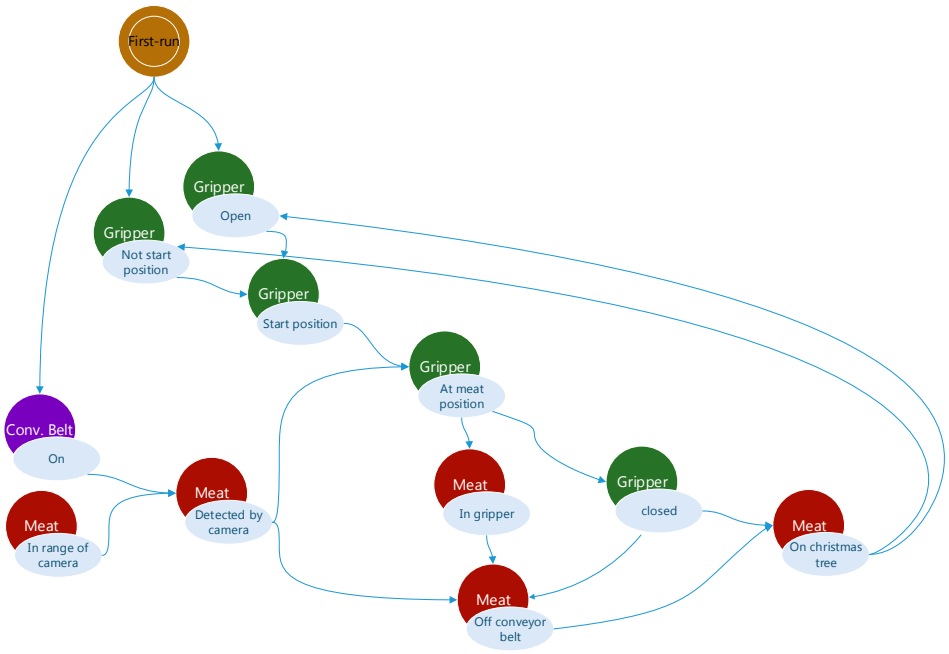


Figure 3.3: Action Types Flow Chart

The diagram is different from the other models in this chapter, as the states of the robotic system are not included. The point of this diagram is to track what the various components are doing, and what their next state will be.

The green, red, and purple circles represent the gripper, the meat and the conveyor belt, respectively. The light blue ovals show the situation/state that the component is in. The arrows going from component to component represent the change that happens, and is not named in the diagram, so at to not steal focus. The arrow points to the next logical state for a component to be in, or the next logical state for another component to be in, given the first component's state. For example, once the gripper is closed the next logical state for the meat is to be off the conveyor belt and then on the christmas tree.

This ties in with the opportunity concept of action phases, and is a good introduction to the next section, where the completion of one action is the opportunity of another. This diagram was an exercise in seeing how the

components act when time and system-states are not considered. Futhermore, this diagram brought to attention the cyclical nature of the robotic system and its dependency on the triggering action of the camera.

3.2.3 Action Phases

In order to create a model that can describe the actions of the robotic system, the concept of Action Phases was used. Since the end of one action is often the precondition for the start of another, it was considered logical to chain these actions together, so that the completion of, for example, the gripper opening, would satisfy the opportunity criteria of the reach-meat action of the gripper.

Figure 3.4 is the first attempt at an analysis of the robotic system with a focus on Action Phases. The actions were chosen from the Action Types analysis in Section 3.3.2. In order to reduce the visual clutter, the content of the action phase boxes have been reduced to only a few words. This is to limit complexity and give an overview of the thought process involved. A full analysis was undertaken, and the result of this is shown in the diagram on page 26. The red arrows connecting the action phase boxes indicate how the completion of one action is the opportunity of another action.

The application of the action phases theory revealed a great deal of preconditions that had not been previously considered.

In the hierarchical model preconditions are included. However, these preconditions are few and vague compared to the preconditions outlined in this section. This is due to the distinction between preconditions of opportunity and potentiality. Having to think about an action in terms of the opportunity to perform that action and the potentiality for that action to be performed, reveals more preconditions than are revealed when simply thinking about "what needs to be in place for this action to take place?" This revealing of preconditions is also part of the reason why a structured way to think about actions is valuable.

Documenting actions as separate events does not leave room for the intended outcome of the system. A sequence of successful events does not automatically result in the intended outcome. For instance, if the meat is the wrong size, all the actions that the robot system perform might be successful, but the meat would still end up on the floor before, during, or after the robot attempts to lift it off the conveyor belt and place it on the meat hook. This can be avoided by analysing the potentiality and opportunity of each action phase while designing the robot system, as this could motivate the designers to implement monitoring systems for the power, liability, capability and reachability of each action phase.

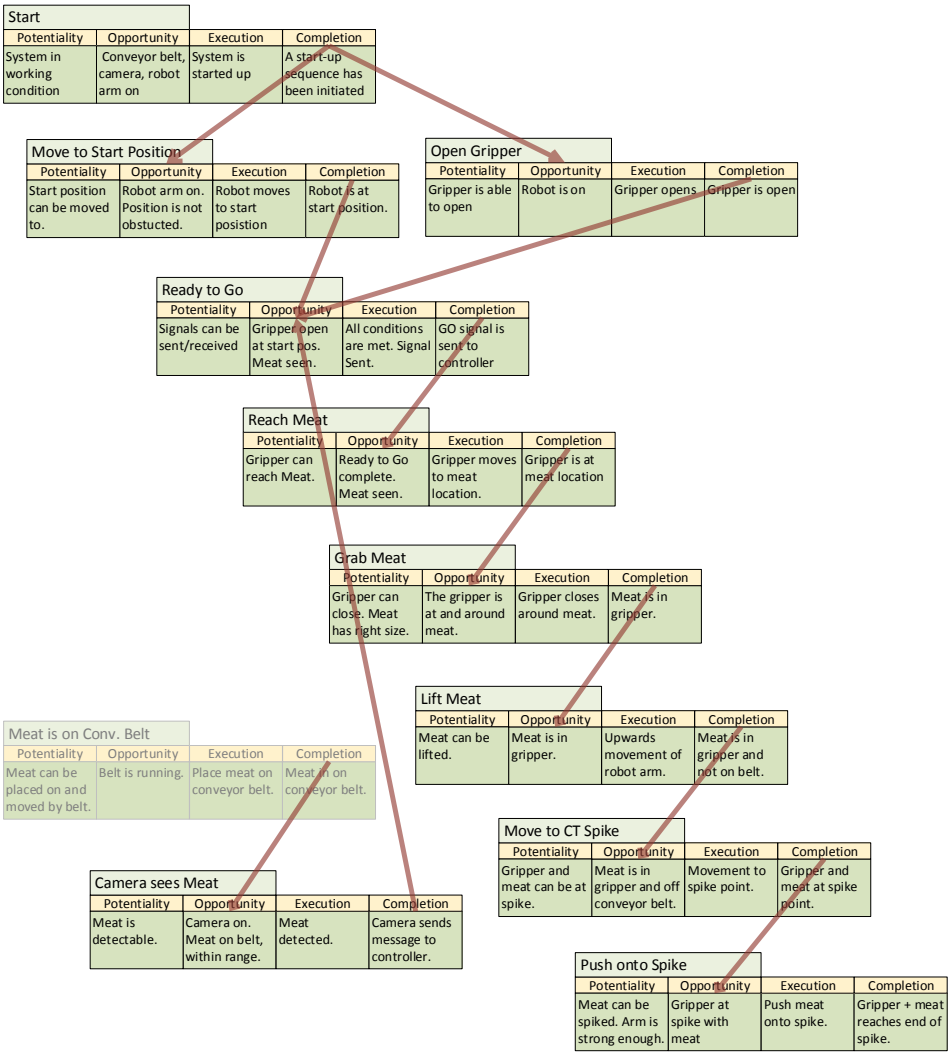


Figure 3.4: Cascaded Action Phase Diagram

3.3 Developing the Model

By bringing together the models and theories in section 3.2 the model on page 26 was developed. This has the circular nature of the action roles diagram, Figure 3.3, linked together like the cascaded action phases of the action phase diagram, Figure 3.4. Figure 3.5 shows a simplified version of the final model for easy reference.

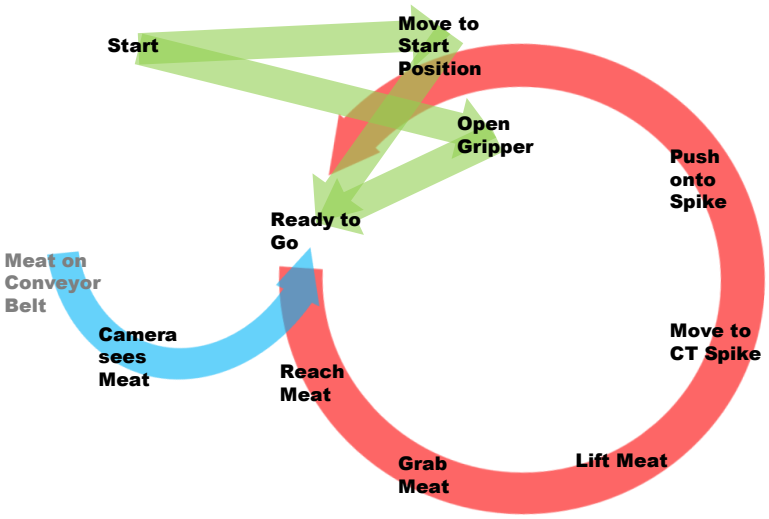


Figure 3.5: Robotic System Model, simplified

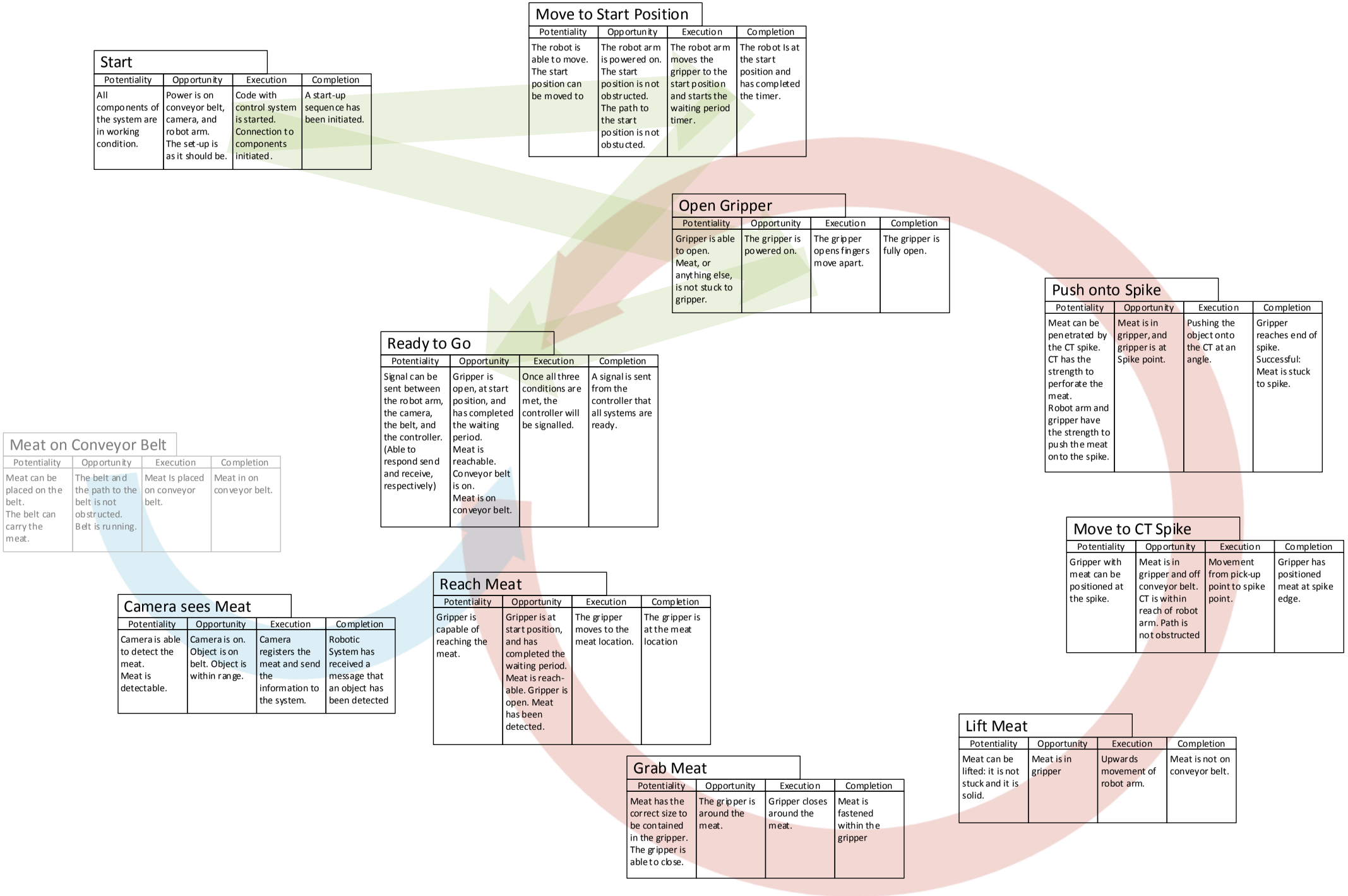
What became apparent when doing this model was the three parts of the model that have been highlighted in the colours blue, green, and red.

The green part is the initialization that happens only the one time when the system is started up. The two states Move to Start Position and Open Gripper are both part of the initialization and the red circle.

The red circle is the cyclical part of the robotic system. This circle is followed and completed in this way every time a new object is detected. It is a circle, since the state that the system starts in is the same state that it end in when the sequence of actions appearing in the red circle is finished.

The blue part represents the triggering of the meat detection by the camera. This is independent of the initialization and the circular process, as it will register an object regardless of whether the system is at the Ready to Go state or not.

The Ready to Go state is in this way a node point for the system, as this is where the system waits for a trigger in order to resume action, and this is also where the initialization process terminates.


Start
Potentiality
All components of the system are in working condition.
Opportunity
Power is on conveyor belt, camera, and robot arm. The set-up is as it should be.
Execution
Code with control system is started. Connection to components initiated.
Completion
A start-up sequence has been initiated.
Move to Start Position
Potentiality
The robot is able to move. The start position can be moved to
Opportunity
The robot arm is powered on. The start position is not obstructed. The path to the start position is not obstucted.
Execution
The robot arm moves the gripper to the start position and starts the waiting period timer.
Completion
The robot Is at the start position and has completed the timer.
Open Gripper
Potentiality
Gripper is able to open. Meat, or anything else, is not stuck to gripper.
Opportunity
The gripper is powered on.
Execution
The gripper opens fingers move apart.
Completion
The gripper is fully open.
Push onto Spike
Potentiality
Meat can be penetrated by the CT spike. CT has the strength to perforate the meat. Robot arm and gripper have the strength to push the meat onto the spike.
Opportunity
Meat is in gripper, and gripper is at Spike point.
Execution
Pushing the object onto the CT at an angle.
Completion
Gripper reaches end of spike. Successful: Meat is stuck to spike.
Ready to Go
Potentiality
Signal can be sent between the robot arm, the camera, the belt, and the controller. (Able to respond send and receive, respectively)
Opportunity
Gripper is open, at start position, and has completed the waiting period. Meat is reachable. Conveyor belt is on. Meat is on conveyor belt.
Execution
Once all three conditions are met, the controller will be signalled.
Completion
A signal is sent from the controller that all systems are ready.
Meat on Conveyor Belt
Potentiality
Meat can be placed on the belt. The belt can carry the meat.
Opportunity
The belt and the path to the belt is not obstructed. Belt is running.
Execution
Meat Is placed on conveyor belt.
Completion
Meat in on conveyor belt.
Move to CT Spike
Potentiality
Gripper with meat can be positioned at the spike.
Opportunity
Meat is in gripper and off conveyor belt. CT is within reach of robot arm. Path is not obstructed
Execution
Movement from pick-up point to spike point.
Completion
Gripper has positioned meat at spike edge.
Camera sees Meat
Potentiality
Camera is able to detect the meat. Meat is detectable.
Opportunity
Camera is on. Object is on belt. Object is within range.
Execution
Camera registers the meat and send the information to the system.
Completion
Robotic System has received a message that an object has been detected
Reach Meat
Potentiality
Gripper is capable of reaching the meat.
Opportunity
Gripper is at start position, and has completed the waiting period. Meat is reach-able. Gripper is open. Meat has been detected.
Execution
The gripper moves to the meat location.
Completion
The gripper is at the meat location
Lift Meat
Potentiality
Meat can be lifted: it is not stuck and it is solid.
Opportunity
Meat is in gripper
Execution
Upwards movement of robot arm.
Completion
Meat is not on conveyor belt.
Grab Meat
Potentiality
Meat has the correct size to be contained in the gripper. The gripper is able to close.
Opportunity
The gripper is around the meat.
Execution
Gripper closes around the meat.
Completion
Meat is fastened within the gripper

3.3.1 Model Methodology

Since this project is about developing a methodology, the concept developed with this model is the idea of three distinct processes: Circular process, Initialization process, and Triggering process, and the node points that would connect a triggering process to a circular process.

Potentially, this concept could work for a robotic system with multiple circular processes and triggers. Consider multiple robots in an assembly line, each handling a separate part of the larger system. They would each have a main circular process, which could involve either one or multiple node points, as they receive the objects to be worked on, and as they hand them off to the next part of the process. Exploring the actions in each individual circular process with the action phases concept could reveal potential problems of the system that had not been previously considered. This would be useful in the design phase, where many potential issues could be identified and solved.

As it is a robotic system and not a singular unit it can be occupying multiple states at the same time. For instance, moving to start position while opening the gripper. In that sense, the model is not so much a state-diagram as a flow-diagram.

3.3.2 Initialization Process States

These are the states that are part of the initialization process, and are shown in the model as the green lines. The states Open Gripper and Move to Start Position are part of both the initialization process and the circular process. The initialization process simply triggers these to start, so that the circular process is initialized to wait at the Ready to Go node point.

Start

This state is the initializer for the whole system. In the Start state the code that runs the controller for the robotic system is activated.

Open Gripper

This state opens the gripper and checks that it has been opened. This state is both activated at the start of the system, and when the meat has been placed on a spike at the end of the circular process.

Move to Start Position

Just as the Open Gripper state, this state is also both a part of the initialization process and the circular process. This is the state that ensures the movement of the robot arm from an arbitrary position to the start position.

3.3.3 Trigger Process States

These are the states that are part of the triggering process, and are shown in the model as part of the blue line.

Meat on Conveyor Belt

This state is the action of the meat being placed on the conveyor belt. In its current form, this is done by a person, but in the future this would potentially be done by another part of the larger meat processing system that the robotic system is a part of.

This state is currently grey, as it is only possible to check if the meat has been placed on the conveyor belt by the camera in the next state. It is still included, though, as it would be relevant in an assembly line, where an earlier part of the entire system would inform the robotic system that the meat has been placed. This would be useful in case the camera was malfunctioning.

Camera Sees Meat

Once the camera registers the meat, this state is completed. Technically, this state is always active as it checks every image it takes for the object, but only completes and fires an activation to the system once an object is registered.

3.3.4 Circular Process States

These are the states that are part of the circular process, and are shown in the model as part of the red circle.

Ready to Go

The Ready to Go state is more of a node point, and a state that is relevant to have even if it is not an explicit action. It is important to include, because it is the state that checks that all the components are ready when the triggering event happens, and the state where the circular process begins and ends.

This state checks that the gripper is both at the right position, open, and that the waiting period has been completed. When an object is then detected by the camera, a signal is sent for the robot arm to reach for the object, which is the next state, and which causes the Ready to Go state to have completed/terminate.

Reach Meat

This state starts the robot arm moving towards the predicted pick-up point of the meat, and completes once the robot arm has moved the open gripper to the position of the meat on the conveyor belt.

Grab Meat

This state is the action of closing the gripper around the meat. Since there is a sensor inside the gripper, it is possible to check whether the object has been successfully reached and contained in the gripper.

Lift Object

Once the object has been registered as in the gripper, and the gripper has finished closing around the meat, the robot arm lifts the gripper and the meat off of the Conveyor Belt. This state is completed as soon as the meat is no longer in contact with the conveyor belt.

Move to CT Spike

This state represents the movement of the gripper to the end of one of the christmas tree spikes. There should be a method for checking if the CT is full, but currently the system is just programmed to fill one tree and then terminate, without checking if there is anything on the spikes.

Push onto Spike

This state represents the movement of the gripper from the CT spike end to the base of the spike. This movement causes the spike to penetrate the meat.

The end of this action is the start of the states Move to Start Position and Open Gripper. After those states have completed as well, the system's circular process has completed one full run, and is back to start, waiting for a new trigger.

3.3.5 Exclusion of Information in Model

In order for the model to be useful, some information has to be discarded or simplified. This is a topic worthy of some consideration, as a cluttered model with large chunks of text taking up most of the space would be just as useless as an oversimplified model.

For this reason the preconditions of each state have been selected so that only the preconditions special/exclusive to that particular state are included

in the model. General preconditions such as power supply and working parts are omitted, as they would take up a lot of space. They should obviously still be included in the implementation, as the monitoring system should be able to detect a power failure or a damaged part.

Chapter 4

Functional Model Based Monitoring System

A model developed using the methodology outlined in chapter 3 can be useful in many phases of system development and utilization, such as design, monitoring, diagnosis and planning.

A system can be designed based on a functional analysis of the desired tasks to be performed. In a finished system, the model can be used for monitoring the system and registering if anything goes wrong by checking preconditions to actions. This in turn could be useful for diagnosing why an error occurred, and even planning how to move forward and either bypass the error, correct the error, or stop the system.

For this project it was decided to build a monitoring system for the robotic system based on the model developed in section 3.3. Since the model was built from a functional perspective, this gives us a thorough understanding of how the system works, and makes it possible to build a functional monitoring system. This means that the monitoring system can provide more detailed information for the error messages, which could eventually lead to expanding the monitoring system into a diagnosis and planning system.

The monitoring program only receives signals from the system. It will throw error messages and warnings, but these will not have any effect on the robotic system.

The model has been implemented, and is described in detail in the next chapter. However, due to time limitations, the system is only capable of detecting three distinct errors.

A simulation environment was developed, in order to simulate the data

that the robotic system would have passed to the monitoring system. It was not possible to test the monitoring system on the robotic system, as it had to be handed back to the slaughterhouse.

4.1 Monitoring System Layout

The monitoring system tracks the robotic system, in order to detect deviations. The monitoring system is based on the model developed in Chapter 3.

The model in Figure 4.1 shows the way the monitoring system is implemented.

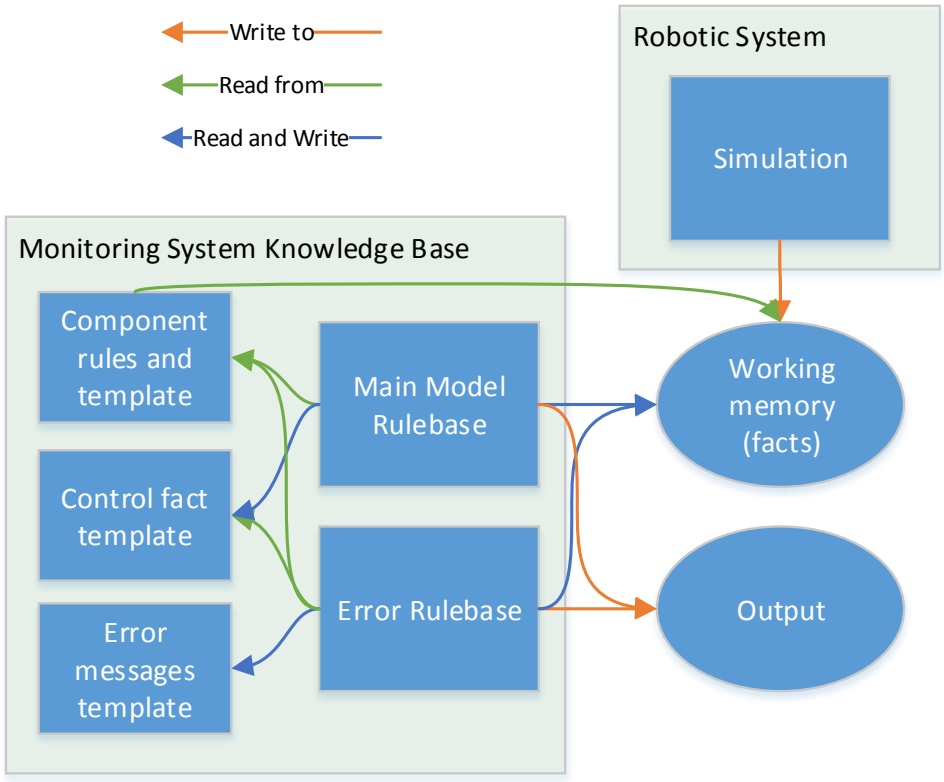


Figure 4.1: Layout of the Implemented Monitoring System

The ovals represents the parts of the monitoring system that change during operation. The rectangles represent the parts that do not change.

The knowledge base[10] is made up of the templates, the main model rulebase, and the error rulebase. The knowledge base is the monitoring system implementation that will be explained and discussed in chapter 5. The

programming language CLIPS is used for the implementation, which will also be explained in chapter 5. Output is the output from the knowledge base written to the screen and two .txt files, one containing all messages and one containing the error messages.

The component rules and template is updated by the simulation which serves to log the component states. Technically, the simulation asserts the updated component values as facts to the working memory, which triggers the component rules to update the component templates.

The controlfact template tracks the state of the robotic system. Every time a rule in the main model rulebase is activated, it sets a control fact asserting what state the system is in. For example, when the Ready to Go rule is triggered, the control fact is updated to say "ready-to-go", which tells the rest of the monitoring system that the robot system is in the Ready to Go state.

The error message template is used to set whether errors have been detected or not. These are set to 0 when the system starts up, to indicate that no errors have been registered. If an error is registered, the corresponding error message is set to 1. This is done in order to register the error, but also to ensure the rule does not continue throwing the same error message. If the main system moves on, the error message is reset to 0, so the rules can detect if the error occurs again.

The templates are not changed, but the values of the variables they contain are updated.

The main model rulebase contains the rules that check if the robotic system states have completed. This rulebase reads from the templates and the facts that have been asserted in the working memory, and asserts new facts accordingly. Also, it writes to the screen and to a text file about which states have been completed, and updates the control fact about which state(s) the monitoring system believes the robotic system is in.

The error rulebase contains the rules made to check for errors. These read the component values, the working memory, and the control facts set by the main system, and determined is an error has occurred from this information. The error rules do not influence the main system monitoring, but simply throw an alert by writing to the screen and a text file.

Chapter 5

Implementation of Monitoring System

This chapter documents the implementation of the monitoring system for the robotic system. The monitoring system is built using the model developed in Chapter 3, and aims to show the applicability of this model.

The monitoring system is built using the programming language CLIPS. The monitoring system is tested using a simulation of the robotic system. All the code developed for this chapter can be seen in Appendix A.

In this chapter the meat is referred to as either "meat" or "object".

5.1 CLIPS

CLIPS (C Language Integrated Production System) is a programming language used for the development of expert systems[6]. Expert systems are a branch of Artificial Intelligence that attempts to emulate decision making based on a set of rules and facts[10].

The meaning of this is closely linked to their semantic meanings. Just as most people know that if it is raining and if you go outside, then you will get wet, an expert system would be able to deduce the same information by receiving the weather conditions at a location and the person location in the form of a fact. A rule would then be activated saying that the person will get wet.

Facts can be asserted either at the start of the program or by rules during the run of the program. Also, facts can be asserted using either the `assert` function or the `deffacts` function. Facts can be removed using the `retract` function.

Rules can be equated with if-then logic. Templates are comparable to C++ classes, as they can contain their own sets of rules and variables.

Rules can establish facts, and facts can make other rules become active, which might in turn establish further facts.

When run, a CLIPS program checks all rules simultaneously, unlike most programming languages where the code is run sequentially, and sections are only run when they are called.

CLIPS syntax of templates, rules, and functions for creating facts:

```
(deftemplate weather
    (slot location)
    (slot temperature)
    (slot humidity)
    )

(defrule in_bad_weather
    (weather raining)
    (position outside)
    =>
    (assert (clothes wet))
    )

(assert (fact-by-itself))  ;this creates one fact

(deffacts scenario         ;this creates three facts
    (weather sunny)
    (wind strong)
    (position at-home)
    )
```

It should be mentioned that if the above example was run, the rule would not fire as the facts asserted say that the weather is sunny and the position is inside. Also, the template is not used.

5.2 Robotic System Monitoring Availability

During the action phase analysis of the robotic system, many requirements for the completion of actions became apparent, and while it is possible to monitor large parts of the system, many areas are simply not possible to measure. This includes both parts of the system that should in the future be possible to monitor, and also areas where monitoring would be inconveniently difficult or expensive.

5.2.1 Signals Currently Available

The available signals for the components are the following:

- For the robot arm, it is possible to check the position of the gripper, the distance between the gripper fingers, and whether an object is between the gripper finger.
- The conveyor belt can be either on or off.
- From the camera, an image is sent every 30 ms.

These signals are used to create the simulated signals in Section 5.3.1, which can be used for checking many of the conditions in the robotic system.

5.2.2 Unavailable Signals

The robot is implemented in such a way that the sensor that checks whether there is an object between the gripper fingers, is only updated when the gripper moves. Therefore, this signal is currently not included in the implementation.

Other situations that are not possible to monitor:

- Whether the power is on, is not possible to check unless the monitoring system is implemented on a separate computer from the controller.
- If the meat is not detected, it cannot be checked whether this is due to a camera malfunction or simply that the meat was not put on the conveyor belt.
- It is not possible to check if the meat is staying on the CT spike after the gripper moves away.

More could possibly be identified, but is not relevant for the rest of this project. It would be relevant if a full implementation was to be done, or in case a new system was to be designed.

5.3 Simulation Environment: Python and MatLab

The simulation environment is a simple MatLab program which generates a .mat file containing numbers. These symbolize the robot systems signals. These are saved, and are thus not updated regularly.

The MatLab files are interpreted by a Python program which loads all the signals into the CLIPS program.

The Python code uses a for loop to send the simulated component values, one value at a time, to the CLIPS file. A wait function is implemented in

the for loop, which limits the update rate to once every 0.1 seconds, in order to simulate time for the CLIPS program.

The CLIPS system is developed to update all the information on clock ticks, so that the system is not only updated every time there is a change in the robotic system (which would not be predictable if the system was running in real time.)

5.3.1 Simulated Signals

Using the available signals from the robotic system, it is possible to create some state-signals in the robotic system controller. For instance, if the conveyor belt is running, it is possible to set a variable called convbelt = 1, and if it was off convbelt would be = 0.

The state-signals that the simulation environment generates are shown in Table 5.1.

`RobotArmState`	0: Waiting	1: Moving	2: Lifting
	3: Pushing	4: NotReady	
`GripperPosition`	0: NotStart	1: Start	2: Moving
	3: Object	4: CTbegin	5: CTend
`GripperState`	0: open	1: closed	
`ConveyorBelt`	0: off	1: on	
`ObjectDetection`	0: object-not-seen	1: object-seen	

Table 5.1: Simulated Signals

For the signals GripperState, ConveyorBelt, and ObjectDetection, their respective binary states are self-explanatory. RobotArmState and GripperPosition need a short explanation.

For RobotArmState, Moving is whenever the arm is moving but neither lifting nor pushing. Lifting is the state right after the meat has been grabbed and Pushing is the state during the movement from the beginning of the CT spike to the end of the CT spike. NotStart is the waiting period after having moved to the start position before the Ready to Go signal is sent. After the 1.2 second wait, the state is switched to Waiting.

For GripperPosition, NotStart is the state when the arm should be moving the gripper to the start position. CTbegin and CTend are active when

the gripper is at the spike point and the spike base, respectively. Start is any time the Gripper is at the start position, Object is when the gripper is around the meat, while the meat is still on the conveyor belt, and Moving is any other time.

5.4 Implementation in CLIPS

A monitoring system for the robotic system is implemented in CLIPS using the model developed in Chapter 3.

Templates are used for each of the simulated signals from the simulated robotic system Table 5.1. These are updated for each clock-tick of the simulation system, which is defined in the Python code as one tenth of a second.

Table 5.2 shows the templates that are created for the system. Table 5.3 shows the templates being used to create components, error messages and an initialization control fact, once the system is run for the first time. Also, the text files are created and opened.

templates
(deftemplate component (slot name))
(deftemplate state (slot component) (slot value))
(deftemplate controlfact (slot message))
(deftemplate errormessage (slot message) (slot value))

Table 5.2: templates created for the monitoring system

files opened
(open "all_messages.txt" all_messages "w") (open "error_messages.txt" error_messages "w")
templates utilized
(assert (controlfact (message first-run))) (assert (component (name RobotArm))) (assert (component (name GripperPosition))) (assert (component (name GripperState))) (assert (component (name ConveyorBelt))) (assert (component (name ObjectDetection))) (assert (errormessage (message conv-belt-off) (value 0))) (assert (errormessage (message timeout-ready2go) (value 0))) (assert (errormessage (message obj-det-sys-not-ready) (value 0)))

Table 5.3: Initialization of monitoring system

Below is the layout of what the rules look like in the implementation. This is provided to give the reader a better understanding of the following sections, where the implemented rules are shows.

```
(defrule generic_rule
    Conditions: facts checked
    =>
    Outcomes: facts asserted/modified
              facts retracted
              messages printed
    )
```

In the following lists of rules, *facts checked* means that these are the conditions that must be met for the rule to become active and carry out fact assertions, printing messages or retracting facts.

When the conditions are met, *facts asserted/modified* and *facts retracted* are the operations that take place within the rule.

When messages are printed they are also saved to the *all_messages.txt* file. When an error message is printed it is saved to both the *all_messages.txt* and *error_messages.txt* file.

5.4.1 Rules for Main System Monitoring

The main system monitoring rules have been divided into their respective processes for easier overview.

Each rule sets one or more control facts which are used to activate other rules.

Figure 5.1 shows the interconnection of both the main system rules and the error rules. The main rules are following the same pattern as the model from chapter 3. The error rules are connected by the purple arrows, and as can be seen, they are affected at different states of the main system. The conveyor belt error is outside the processes, as it will trigger any time the conveyor belt is not on.

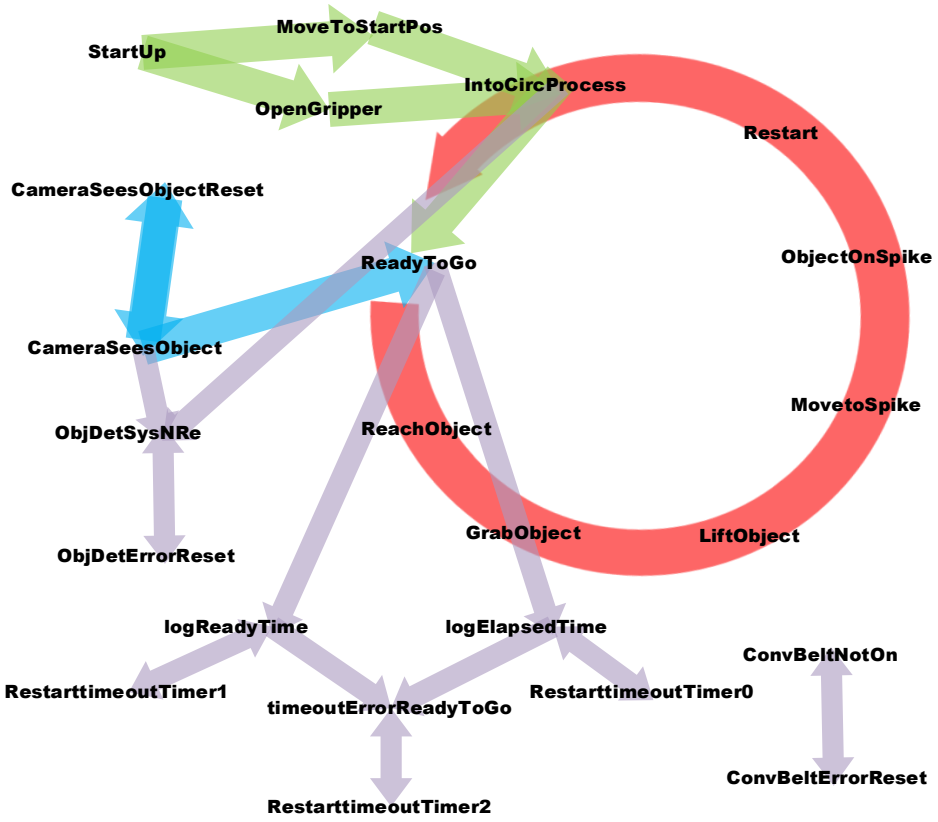


Figure 5.1: Main Rulebase and Error Rulebase Interconnection

Initialization Process Rules

These are the rules that make up the initialization process. The first rule, StartUp, is activated at the start of the program, and this in turn activates the two next rules.

There is a difference from the model developed in chapter 3, which is that the rule IntoCircProcess has been included. This is used for error checking

and signifies that the gripper is open and at the start position, but the go signal from the controller, which takes 1.2 second, has not yet been received.

StartUp

facts checked:
controlfact: first-run

facts asserted/modified:
component values are set to 0 {initialized}
controlfact: sysready-pos
controlfact: sysready-grip

messages printed:
STARTUP Complete

facts retracted:
controlfact: first-run

OpenGripper

facts checked:
controlfact: sysready-grip
GripperState = 0 {open}

facts asserted:
controlfact: gripper-open

facts retracted:
controlfact: sysready-grip

messages printed:
OPENGRIPPER Complete

MoveToStartPos

facts checked:
controlfact: sysready-pos
GripperPosition = 1 {Start}
controlfact: gripper-open

facts asserted:
controlfact: start-pos

facts retracted:
controlfact: sysready-pos

messages printed:
MOVETOSTARTPOS Complete

IntoCircProcess

facts checked:
controlfact: gripper-open
controlfact: start-pos

facts asserted:
controlfact: intoReady2Go

facts retracted:
controlfact: gripper-open
controlfact: start-pos

messages printed:
waiting: 1.2 s

ReadyToGo

facts checked:
controlfact: intoReady2Go
RobotArm = 0 {Waiting}
GripperPosition = 1 {Start}
GripperState = 0 {open}
ConveyorBelt = 1 {on}
controlfact: gripper-open
controlfact: start-pos

facts asserted:
controlfact: ready2go

facts retracted:
controlfact: intoReady2Go

messages printed:
System is READY

Triggering Process Rules

These are the rules that are activated when an object is registered by the camera.

The second rule resets the (object-seen) fact, when the ObjectDetected signal goes to 0. This fact is to ensure that, while ObjectDetected = 1, the monitoring system doesn't keep writing to the screen or the output file that a new object has been detected.

The controlfact: object-seen fact is set in the triggering process, but retracted in the circular process once the ReachObject rule has been activated. This is done so that the system won't reach for an object that was detected before the system was ready for it. The triggering process will still detect

an object, however the circular process will not respond to it.

CameraSeesObject

facts checked:
ConveyorBelt = 1 {on}
ObjectDetection = 1 {object-seen}
The fact (object-seen) has not been asserted.

facts asserted:
(object-seen)
controlfact: object-seen

messages printed:
OBJECT SEEN

CameraSeesObjectReset

facts checked:
(object-seen)
ObjectDetection = 0 {object-not-seen}

facts retracted:
(object-seen)

Circular Process Rules

These are the rules for checking that the circular process is active.

The rules are all set by both a control fact which checks if the previous rules have been triggered, and a check for the correct component values, to check if the actions have happened in the simulation.

The circular process also has slight differences from the model developed. First off, the action Push-onto-Spike has been replaced with ObjectOnSpike. This is because the rule checks if the task has been completed, as opposed to checking if it is ongoing. Secondly, the last rule, Restart, has been introduced as the rule that resets the circular process back to intoReady2Go.

Once the system is ready to go and an object is registered the first rule in the circular process is triggered.

ReachObject

facts checked:
controlfact: ready2go
controlfact: object-seen
RobotArm = 1 {moving}
GripperPosition = 3 {object}
GripperState = 0 {open}
ConveyorBelt = 1 {on}

facts asserted:
controlfact: reach-object

facts retracted:
controlfact: ready2go
controlfact: object-seen

messages printed:
OBJECT REACHED

GrabObject

facts checked:
controlfact: reach-object
RobotArm = 1 {moving}
GripperPosition = 3 {object}
GripperState = 1 {closed}
ConveyorBelt = 1 {on}

facts asserted:
controlfact: grab-object

facts retracted:
controlfact: reach-object

messages printed:
OBJECT GRABBED

LiftObject

facts checked:
controlfact: grab-object
RobotArm = 2 {lifting}
GripperPosition = 2 {moving}
GripperState = 1 {closed}

facts asserted:
controlfact: lift-object

facts retracted:
controlfact: grab-object

messages printed:
OBJECT LIFTED

MovetoSpike

facts checked:
controlfact: lift-object
RobotArm = 3 {pushing}
GripperPosition = 4 {CTbegin}
GripperState = 1 {closed}

facts asserted:
controlfact: push-onto-spike

facts retracted:
controlfact: lift-object

messages printed:
ROBOT ARM PUSHING

ObjectOnSpike

facts checked:
controlfact: push-onto-spike
RobotArm = 0 {waiting}
GripperPosition = 5 {CTend}
GripperState = 1 {closed}

facts asserted:
controlfact: object-on-spike

facts retracted:
controlfact: push-onto-spike

messages printed:
OBJECT ON SPIKE

Restart

facts checked:
controlfact: object-on-spike
RobotArm = 0 {waiting}
GripperPosition = 1 {start}
GripperState = 0 {open}
ConveyorBelt = 1 {on}

facts asserted:
controlfact: intoReady2Go
controlfact: sysready-pos
controlfact: sysready-grip

facts retracted:
controlfact: object-on-spike

messages printed:
BACK TO START

5.4.2 Rules for Error Monitoring

Rules were also used to check for errors in the system. There are the main error rules that check if the errors are present, and then there are rules for resetting the error once the error has been resolved. This is to make sure that the error could fire again, if it re-emerged. For instance, if there was an object not detected due to the system not being ready, the system should still be able to catch a second object on the conveyor belt, and throw a second error if a third object showed up too soon.

An error message template was created so it would be easy to set and reset error messages. Three error messages are thus asserted with initial value = 0 at the start of the system.

In the timeout error case there are also supporting rules that check the time since the system entered the Ready to Go state.

In a full implementation, there would be timer rules similar to the Ready to Go timeout rule for every main system state. This would be to check whether the robotic system is progressing as planned, or possibly stuck somewhere.

Rules for Error: Conveyor Belt Off

These are the rules for detecting if the conveyor belt is off. This rule only throws a warning message if both the RobotArmState and the GripperPosition values are equal to zero, as this means the system is either just starting up, or waiting at the start position for the Ready to Go timer to run out.

This is so that the system can start up with the conveyor belt off without it triggering an error message. However, if either component changes value, i.e the robot arm moves or the Ready to Go timer has finished, an error is thrown.

If an error has been thrown and the conveyor belt is turned on, the error message is reset to zero, so that it can register if the error occurs again.

ConvBeltNotOn

facts checked:
errormessage: conv-belt-off 0 {error not detected}
ConveyorBelt = 0 {off}
RobotArm = ?ra (the value of RobotArm is saved to the local variable ra)
GripperPosition = ?gp (the value of GripperPosition is saved to the local variable gp)

This rule checks if gp + ra is different from 0. If so, the following happens:

facts modified:
errormessage: conv-belt-off 1 {error detected}

messages printed:
!ERROR: Conveyor Belt is Off

if gp + ra = 0 the following warning message is asserted:
!WARNING: Conveyor Belt is Off

ConvBeltErrorReset

facts checked:
errormessage: conv-belt-off 1 {error detected}
ConveyorBelt = 1 {on}

facts modified:
errormessage: conv-belt-off 0 {error not detected}

Rules for Error: Object Detected, System Not Ready

These are the rules for throwing an error if an object is detected on the conveyor belt before the system is ready.

If the RobotArm value is any other value than "Waiting" and an object is detected, the first rule throws an error. The second rule resets the errormessage if the circular process is started (controlfact reach-object is set).

ObjDetSysNRe

facts checked:
errormessage: obj-det-sys-not-ready 0 {error not detected}
RobotArm = ?f1
ObjectDetection = 1 {object-seen}
The rule tests whether ?f1 $\neq$ 0

facts modified:
errormessage: obj-det-sys-not-ready 1 {error detected}

messages printed:
!ERROR: Object Detected before System Ready

ObjDetErrorReset

facts checked:
errormessage: obj-det-sys-not-ready 1 {error detected}
controlfact: reach-object

facts modified:
errormessage: obj-det-sys-not-ready 0 {error not detected}

Rules for Error: Timeout in Ready-to-Go State

These are the rules for detecting a timeout error in the Ready to Go state. Some of these rules use the CLIPS function `time`. This function saves the current run-time to a variable.

The first rule logs the time at which the robotic system is at the ready to go state, and the second rule updates the system time continuously. This enables the main error rule to check if more that 10 seconds has passed since the system went into the Ready to Go state without continuing to the reach-object state (because no object has been detected). In the main rule, the elapsed time variable is constantly retracted so that it can be reset, as long as the timeout error hasn't been detected.

The last three rules are used to reset the error message and the two timers in case the timeout error was detected.

logreadytime

facts checked:
controlfact: ready2go

facts asserted/modified:
(time-ready (time))

logElapsedTime

facts checked:
controlfact: ready2go
RobotArm = 1 {moving}

facts asserted/modified:
(time-elapsed (time))

timeoutErrorReadyToGo

facts checked:
errormessage: timeout-ready2go 0 {error not detected}
(time-elapsed ?tti)
(time-ready ?tri)

This rule checks if 1.0 < (?tti - ?tri). This is equal to 10 seconds. If so, the following happens:

facts modified:
errormessage: timeout-ready2go 1 {error detected}

messages printed:
!ERROR : Timeout (Ready2go): (?tti - ?tri) seconds

if 1.0 < (?tti - ?tri) is not true:

facts retracted:
(time-elapsed ?tti)

RestarttimeoutTimer0

facts checked:
controlfact: reach-object
The fact (time-elapsed ?) exists, where "?" is an arbitrary number.

facts retracted:
(time-elapsed ?)

RestarttimeoutTimer1

facts checked:
controlfact: reach-object
The fact (time-ready ?) exists, where "?" is an arbitrary number.

facts retracted:
(time-ready ?)

RestarttimeoutTimer2

facts checked:

controlfact: reach-object
errormessage: timeout-ready2go 1 {error detected}

facts modified:

errormessage: timeout-ready2go 0 {error not detected}

Chapter 6

Results and Tests

This chapter shows the findings of the project. The methodology developed in chapter 3 is explained. Then, the tests used to test the implementation are introduced, as well as their outcomes.

6.1 Methodology Developed

This methodology is mainly a summary of what was done in chapter 3, but without having to make multiple separate models. This was done by identifying the steps that would accomplish the same goal as all the theories applied and models developed, without having to do it all over again.

The steps for creating a model of a robotic system like the one in chapter 3 are identified as:

Identify tasks

Identify the main task the are to be achieved in the system. This is equivalent to the top level of the hierarchical diagram in Figure 3.2

Identify actions

The actions necessary to complete the tasks can be identified by exploring what the basic action types of the desired system are.

Identify action phases

Once the actions of the system have been identified, they should each be analysed in terms of potentiality, opportunity, execution and completion.

Cascade action phases

The actions should be chained together by identifying which actions need to be completed in order to trigger the opportunity requirements of other actions.

Identify processes

Finally, by explicitly identifying the initialization, triggering and circular processes of the system, and adding the accompanying node points, the model should resemble the model developed in section 3.3.

6.2 Testing the Monitoring System

These are the four scenarios that have been created to test the CLIPS implementation.

In the following sections showing the scenarios, a plot of the data from the corresponding MatLab file is shown first, and then the output asserted by CLIPS to the screen is shown. The simulation values for the components are repeated in Table 6.1 for easy reference, as these are the five signals modelled in the MatLab files.

In the error scenarios, the states that are monitored by the error detection system is plotted in magenta, with the point at which the error occurs shown in red.

The scenarios were created with the possibility to be run alone or one after another. However, when the program is run, the conveyor belt is always set to 0 at first, which will trigger a warning from the monitoring system. This is because the monitoring system initializes all values as 0 at the beginning of the program.

The "finish!" message at the end of the screenshots is to show when a scenario has terminated.

`RobotArmState`	0: Waiting	1: Moving	2: Lifting
	3: Pushing	4: NotReady	
`GripperPosition`	0: NotStart	1: Start	2: Moving
	3: Object	4: CTbegin	5: CTend
`GripperState`	0: open	1: closed	
`ConveyorBelt`	0: off	1: on	
`ObjectDetection`	0: object-not-seen	1: object-seen	

Table 6.1: Simulated Signals

6.2.1 One Run Scenario

This scenario models one full run-cycle. The monitoring system should simply register that the robotic system does what it was made to do. This scenario should not trigger any errors, except for the conveyor belt warning at the beginning, which merely signifies that the program initialized correctly.

The simulation starts up in the initialization process, which terminates in the Read to Go node point. Then a piece of meat is registered by the camera, which triggers the robot to grab, lift and place the meat on the christmas tree. The simulation ends with the robotic system back at the start position, waiting for another object-seen signal

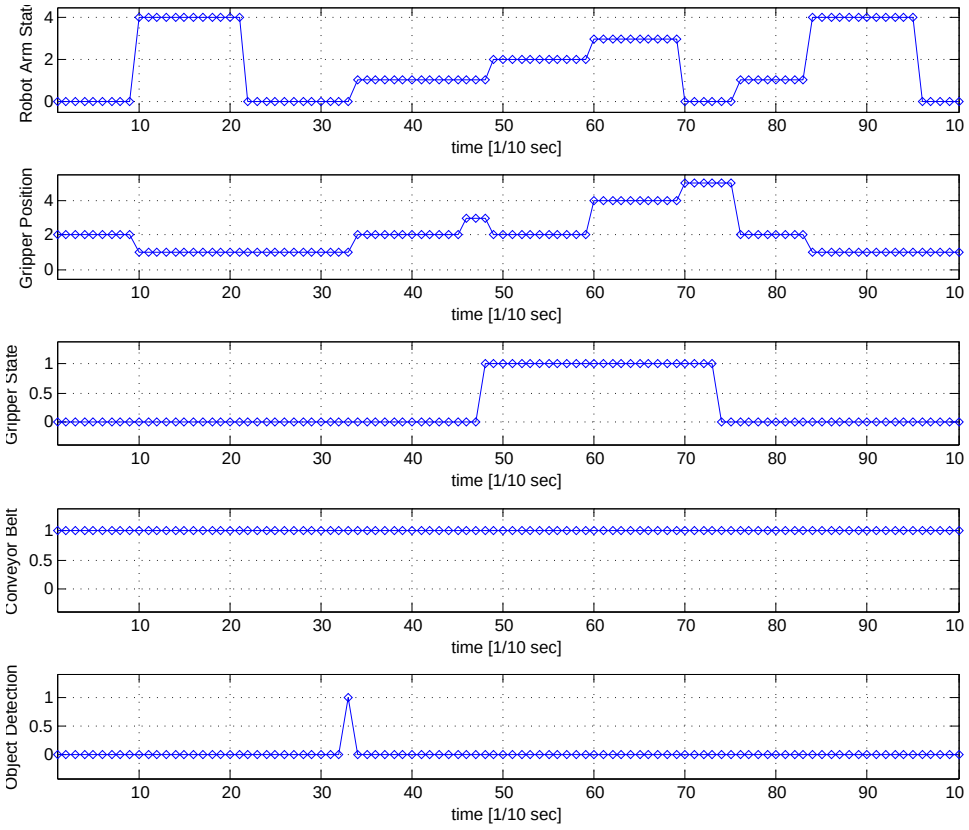


Figure 6.1: one_run_scenario() MatLab Simulation

```
STARTUP Complete
OPENGRIPPER Complete
!WARNING: Conveyor Belt is Off

MOVETOSTARTPOS Complete
(waiting: 1.2 s))
System is READY

OBJECT SEEN

OBJECT REACHED

OBJECT GRABBED

OBJECT LIFTED

ROBOT ARM PUSHING

OBJECT ON SPIKE

BACK TO START
OPENGRIPPER Complete
MOVETOSTARTPOS Complete
(waiting: 1.2 s))
System is READY


finish!
```

Figure 6.2: CLIPS output for one_run_scenario()

6.2.2 Error: Conveyor Belt Off

This is a scenario where the conveyor belt is not turned on. As can be seen in the MatLab simulation, the conveyor belt is off from the start. However, the system initially just throws a warning, and it is first when the system has begun moving that an error is asserted.

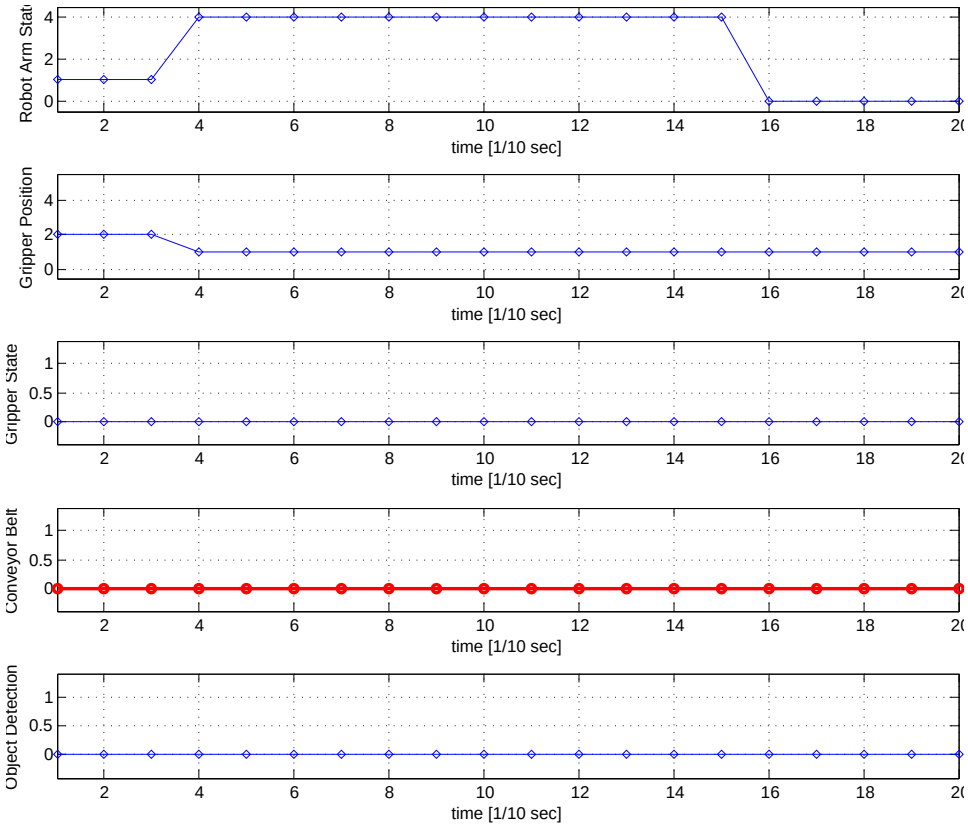


Figure 6.3: conv_belt_off() MatLab Simulation

```
STARTUP Complete
OPENGRIPPER Complete
!WARNING: Conveyor Belt is Off
!WARNING: Conveyor Belt is Off
!ERROR: Conveyor Belt is Off

MOVETOSTARTPOS Complete
(waiting: 1.2 s))


finish!
```

Figure 6.4: CLIPS output for conv_belt_off()

6.2.3 Error: Object Detected, System Not Ready

In this scenario the system is supposed to start up the initialization process, but the camera detects the object before the system has finished the 1.2 s waiting period. Therefore, the system is not in the Ready to Go state when the object is registered by the camera, and the robot will not reach for the object.

This scenario shows how the triggering process works unaffected by the circular and initialization process states, and all the processes are unaffected by the error monitoring system.

The Initialization state sends the message "(waiting: 1.2 s)", and then the triggering process rules detects an object on the conveyor belt. The error rule sees both of these situations, and immediately throws an error. After this, the initialization timer has run out, and the Ready to Go state of the circular process sends the message "System is READY", unaware that an object has been missed.

In this scenario the simulation terminates, but if the system had continued running without another object being detected, the Ready to Go timeout error would also have fired.

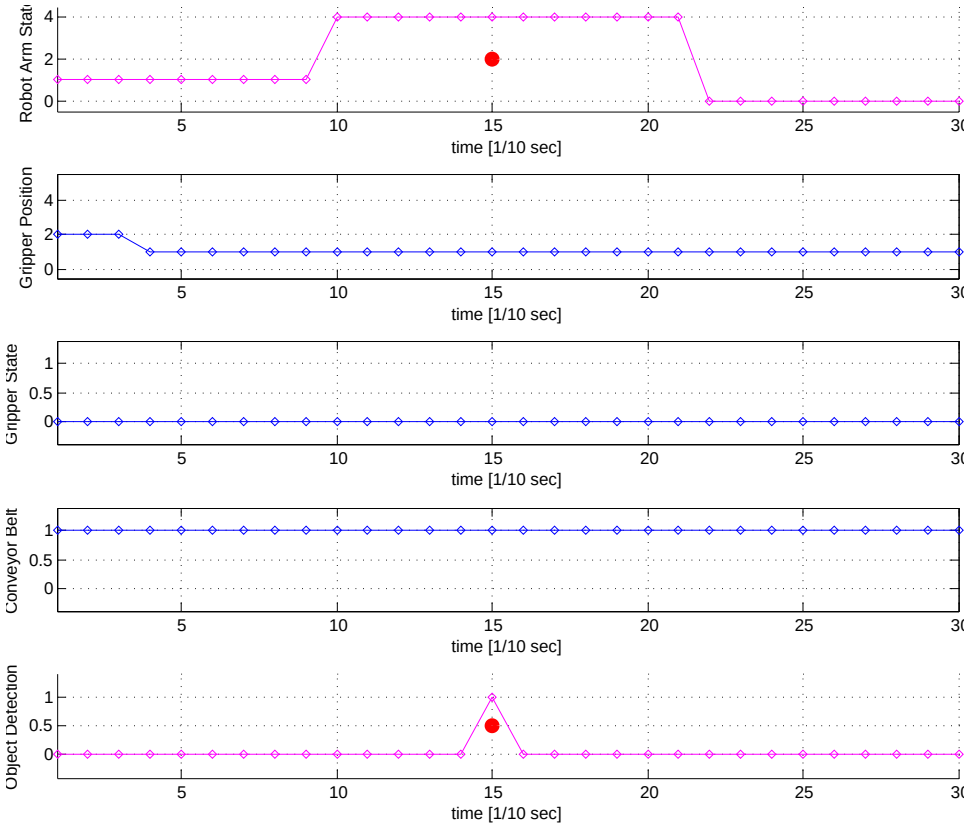


Figure 6.5: obj_det_sys_not_ready() MatLab Simulation

```
STARTUP Complete
OPENGRIPPER Complete
!WARNING: Conveyor Belt is Off

MOVETOSTARTPOS Complete
(waiting: 1.2 s))

OBJECT SEEN
!ERROR: Object Detected before System Ready

System is READY


finish!
```

Figure 6.6: CLIPS output for obj_det_sys_not_ready()

6.2.4 Error: Timeout in Ready-to-Go State

If the system has entered the Ready to Go state, but no object is detected within a certain amount of time, an error should be thrown. This scenario simulates such a situation.

The monitoring system interprets that the system is in the Ready to Go state when GripperPosition = 1 (Start) and RobotArmState = 0 (Waiting). The error monitoring system sees that the main monitoring system is in the Ready to Go state, and starts checking the time elapsed.

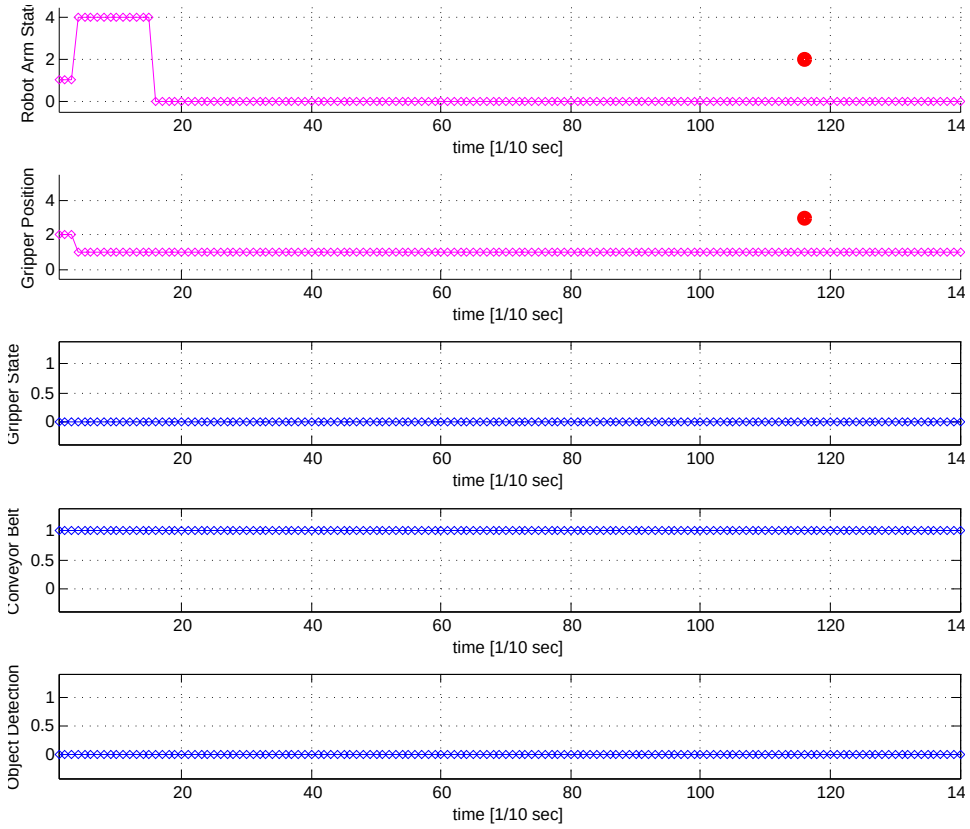


Figure 6.7: timeout_ready2go() MatLab Simulation

```
STARTUP Complete
OPENGRIPPER Complete
!WARNING: Conveyor Belt is Off

MOVETOSTARTPOS Complete
(waiting: 1.2 s))

System is READY

!ERROR : Timeout (Ready2go): 10.08997 seconds


finish!
```

Figure 6.8: CLIPS output for timeout_ready2go()

6.2.5 All Scenarios Run Sequentially

All the scenarios were chained together in order to test that the system could detect multiple errors, and that the scenarios could be run one after another.

The order is consistent with a scenario where the meat is placed on the conveyor belt before the timer has run out, which causes the robotic system to miss the meat. Another piece of meat is put on the conveyor belt, which the system catches. Then, the system is left on for more than ten seconds, and finally the conveyor belt is turned off before the system is turned off.

The order of the scenarios were: obj_det_sys_not_ready(), one_run _scenario(), timeout_ready2go(), and conv_belt_off(). The point at which one scenario finishes and another starts can be identified in the screenshot by the printed "finish!" message.

```
  STARTUP Complete
  OPENGRIPPER Complete
  !WARNING: Conveyor Belt is Off

  MOVETOSTARTPOS Complete
  (waiting: 1.2 s))

  OBJECT SEEN
  !ERROR: Object Detected before System Ready

  System is READY


finish!
  OBJECT SEEN

  OBJECT REACHED

  OBJECT GRABBED

  OBJECT LIFTED

  ROBOT ARM PUSHING

  OBJECT ON SPIKE

  BACK TO START
  OPENGRIPPER Complete
  MOVETOSTARTPOS Complete
  (waiting: 1.2 s))
  System is READY


finish!
  !ERROR : Timeout (Ready2go): 10.09151 seconds


finish!
  !ERROR: Conveyor Belt is Off


finish!
```

Figure 6.9: CLIPS output with all scenarios run

Chapter 7

Conclusion

The purpose of this project was to contribute to the development of a methodology for the functional modelling of robotic systems, by modelling a known robotic system. Furthermore, it was intended to test the developed model by creating a monitoring system based on it.

In order to model the robotic system, a number of theories were explored. The theories selected had contributed to the creation of MFM, which models continuous systems, and it was reasoned that these theories could be useful for creating a methodology for developing discrete models.

Once the model had been developed, it was implemented in the expert system CLIPS, and tested to see if it could detect what a simulation of the robotic system was doing, as well as any errors that may occur.

The implementation of the model covered all the model states, but only a limited amount of error detection was implemented. This was due to time limitations, as the implementation was done at the end of the project.

The main findings of this project was that it was possible to model the robotic system using the theories selected, and develop a monitoring system based on this model. The implemented monitoring system was able to follow the system states and detect errors in a simulation of the robotic system.

It was found that the theory of action phases was quite useful for identifying preconditions, as well as identifying where errors might occur in the robotic system. This means that the analysis of a system's action phases can be helpful in designing a monitoring system.

7.1 Future Work

The logical next step for this project would be to test the implementation on a physical system. This would most likely reveal weaknesses in the model and the implementation that had previously not been considered.

It could also be interesting to expand the error detection system, in order to develop a diagnosis system, and eventually a planning system that would be able to guide the robotic system back to working smoothly in the event of an error.

Appendix A

Monitoring System Program

For the code, Python 2.7 and pyclips 1.0.7 versions were used.

In order to run the code in Linux, both the monitoring.py and monitoring.clp files have to be in the same folder as the terminal is opened to. First, open and run the MatLab file, in order to create the simulation environment. Then enter `pyhon2.7 monitoring.py` into the terminal to run the code.

A.1 CLIPS Code: monitoring.clp

```
;;; ======================================================
;;;
;;; example: diagnosis for pick up and place robotic system
;;;
;;; ======================================================
(watch all)


;-----------------------------------------------------------------
;-----------------------------------------------------------------
; ----  TEMPLATES declared and RULES asserted
;-----------------------------------------------------------------
;-----------------------------------------------------------------
(deftemplate component (slot name))
(deftemplate state (slot component) (slot value))
(deftemplate controlfact (slot message))
(open "all_messages.txt" all_messages "w")
(open "error_messages.txt" error_messages "w")

(assert (controlfact (message first-run)))
(assert (component (name RobotArm)))
(assert (component (name GripperPosition)))
(assert (component (name GripperState)))
(assert (component (name ConveyorBelt)))
(assert (component (name ObjectDetection)))
```

```
(deftemplate system_state (slot name) (slot value))
(assert (system_state (name InitProcess) (value 0)))
(assert (system_state (name TrigProcess) (value 0)))


;---------------------------------------------------------------
;---------------------------------------------------------------
; ----  COMPONENT STATES updated from robot simulation
;---------------------------------------------------------------
;---------------------------------------------------------------
(defrule RobotArmState
    ?a <-(RAS ?x)
    ?s<-(state (component RobotArm) (value ?))
    =>
    (modify ?s (value ?x))
    (retract ?a)
    )
(defrule GripperPositionState
    ?a <-(GP ?x)
    ?s<-(state (component GripperPosition) (value ?))
    =>
    (modify ?s (value ?x))
    (retract ?a)
    )
(defrule GripperStateState
    ?a <-(GS ?x)
    ?s<-(state (component GripperState) (value ?))
    =>
    (modify ?s (value ?x))
    (retract ?a)
    )
(defrule ConveyorBeltState
    ?a <-(CB ?x)
    ?s<-(state (component ConveyorBelt) (value ?))
    =>
    (modify ?s (value ?x))
    (retract ?a)
    )
(defrule ObjectDetectionState
    ?a <-(OD ?x)
    ?s<-(state (component ObjectDetection) (value ?))
    =>
    (modify ?s (value ?x))
    (retract ?a)
    )
;---------------------------------------------------------------
;---------------------------------------------------------------
; ----  ROBOT SYSTEM STATES updated based on component states
;---------------------------------------------------------------
;---------------------------------------------------------------

;---------------------------------------------------------------
;  STARTUP process
```

```
;---------------------------------------------------------------
(defrule StartUp
    ?c<-(controlfact (message first-run))
    ?comp0 <-(component (name RobotArm))
    ?comp1 <-(component (name GripperPosition))
    ?comp3 <-(component (name GripperState))
    ?comp4 <-(component (name ConveyorBelt))
    ?comp5 <-(component (name ObjectDetection))
    =>
    (assert (state (component RobotArm) (value 0)))
    (assert (state (component GripperPosition) (value 0)))
    (assert (state (component GripperState) (value 0)))
    (assert (state (component ConveyorBelt) (value 0)))
    (assert (state (component ObjectDetection) (value 0)))
    (retract ?c)
    (assert (controlfact (message sysready-pos)))
    (assert (controlfact (message sysready-grip)))
    (printout t "  STARTUP Complete" crlf)
    (printout all_messages (time) "  STARTUP Complete" crlf)
    )

(defrule OpenGripper
    ?c<-(controlfact (message sysready-grip))
    (state (component GripperState) (value 0))
    =>
    (retract ?c)
    (assert (controlfact (message gripper-open)))
    (printout t "  OPENGRIPPER Complete" crlf)
    (printout all_messages (time) "  OPENGRIPPER Complete" crlf)
    )

(defrule MoveToStartPos
    ?c<-(controlfact (message sysready-pos))
    (state (component GripperPosition) (value 1))
    (controlfact (message gripper-open))
    =>
    (retract ?c)
    (assert (controlfact (message start-pos)))
    (printout t "  MOVETOSTARTPOS Complete" crlf)
    (printout all_messages (time) "  MOVETOSTARTPOS Complete" crlf)
    )

(defrule IntoCircProcess
    ?c0<-(controlfact (message gripper-open))
    ?c1<-(controlfact (message start-pos))
    =>
    (assert (controlfact (message intoReady2Go)))
    (printout t "  (waiting: 1.2 s))" crlf)
    (printout all_messages (time) "  (waiting: 1.2 s)" crlf)
    (retract ?c0)
    (retract ?c1)
    )
(defrule ReadyToGo
    ?c<-(controlfact (message intoReady2Go))
```

```
    (state (component RobotArm) (value 0))
    (state (component GripperPosition) (value 1))
    (state (component GripperState) (value 0))
    (state (component ConveyorBelt) (value 1))
    =>
    (assert (controlfact (message ready2go)))
    (printout t "  System is READY" crlf)
    (printout all_messages (time) "  System is READY" crlf)
    (retract ?c)
    )

;-----------------------------------------------------------------
;  TRIGGERING process
;-----------------------------------------------------------------

(defrule CameraSeesObject
    (state (component ConveyorBelt) (value 1))
    (state (component ObjectDetection) (value 1))
    (not (object-seen))
    =>
    (assert (controlfact (message object-seen)))
    (assert (object-seen))
    (printout t "  OBJECT SEEN" crlf)
    (printout all_messages (time) "  OBJECT SEEN" crlf)
    )

(defrule CameraSeesObjectReset
    ?cr <- (object-seen)
    (state (component ObjectDetection) (value 0))
    =>
    (retract ?cr)
    )


;-----------------------------------------------------------------
;  CIRCULAR/MAIN process
;-----------------------------------------------------------------

(defrule ReachObject
    ?c0<-(controlfact (message ready2go))
    ?c1<-(controlfact (message object-seen))
    (state (component RobotArm) (value 1))
    (state (component GripperPosition) (value 3))
    (state (component GripperState) (value 0))
    (state (component ConveyorBelt) (value 1))
    =>
    (assert (controlfact (message reach-object)))
    (printout t "  OBJECT REACHED" crlf)
    (printout all_messages (time) "  OBJECT REACHED" crlf)
    (retract ?c0)
    (retract ?c1)
    )
(defrule GrabObject
    ?c0<-(controlfact (message reach-object))
```

```
    (state (component RobotArm) (value 1))
    (state (component GripperPosition) (value 3))
    (state (component GripperState) (value 1))
    (state (component ConveyorBelt) (value 1))
    =>
    (assert (controlfact (message grab-object)))
    (printout t "  OBJECT GRABBED" crlf)
    (printout all_messages (time) "  OBJECT GRABBED" crlf)
    (retract ?c0)
    )

(defrule LiftObject
    ?c0<-(controlfact (message grab-object))
    (state (component RobotArm) (value 2))
    (state (component GripperPosition) (value 2))
    (state (component GripperState) (value 1))
    =>
    (assert (controlfact (message lift-object)))
    (printout t "  OBJECT LIFTED" crlf)
    (printout all_messages (time) "  OBJECT LIFTED" crlf)
    (retract ?c0)
    )

(defrule MovetoSpike
    ?c0<-(controlfact (message lift-object))
    (state (component RobotArm) (value 3))
    (state (component GripperPosition) (value 4))
    (state (component GripperState) (value 1))
    =>
    (assert (controlfact (message push-onto-spike)))
    (printout t "  ROBOT ARM PUSHING" crlf)
    (printout all_messages (time) "  ROBOT ARM PUSHING" crlf)
    (retract ?c0)
    )

(defrule ObjectOnSpike
    ?c0<-(controlfact (message push-onto-spike))
    (state (component RobotArm) (value 0))
    (state (component GripperPosition) (value 5))
    (state (component GripperState) (value 1))
    =>
    (assert (controlfact (message object-on-spike)))
    (printout t "  OBJECT ON SPIKE" crlf)
    (printout all_messages (time) "  OBJECT ON SPIKE" crlf)
    (retract ?c0)
    )

(defrule Restart
    ?c0<-(controlfact (message object-on-spike))
    (state (component RobotArm) (value 0))
    (state (component GripperPosition) (value 1))
    (state (component GripperState) (value 0))
    (state (component ConveyorBelt) (value 1))
    =>
```

```
    (printout t "  BACK TO START" crlf)
    (printout all_messages (time) "  BACK TO START" crlf)
;    (assert (controlfact (message intoReady2Go)))
    (assert (controlfact (message sysready-pos)))
    (assert (controlfact (message sysready-grip)))
    (retract ?c0)
    )
;---------------------------------------------------------------------
;---------------------------------------------------------------------
; ----  ERROR MESSAGES
;---------------------------------------------------------------------
;---------------------------------------------------------------------
(deftemplate errormessage (slot message) (slot value))

(assert (errormessage (message conv-belt-off) (value 0)))
(assert (errormessage (message timeout-ready2go) (value 0)))
(assert (errormessage (message obj-det-sys-not-ready) (value 0)))


;---------------------------------------------------------------------
;  conv belt not on
;---------------------------------------------------------------------

(defrule ConvBeltNotOn
    ?e<-(errormessage (message conv-belt-off) (value 0))
    (state (component ConveyorBelt) (value 0))
    (state (component RobotArm) (value ?ra))
    (state (component GripperPosition) (value ?gp))
    =>
    ( if
        (neq (+ ?ra ?gp) 0)
      then
        (printout t "  !ERROR: Conveyor Belt is Off" crlf)
        (printout all_messages (time)
            "  !ERROR: Conveyor Belt is Off" crlf)
        (printout error_messages (time)
            "  !ERROR: Conveyor Belt is Off" crlf)
        (modify ?e (value 1))
      else
        (printout t "  !WARNING: Conveyor Belt is Off" crlf)
        (printout all_messages (time)
            "  !WARNING: Conveyor Belt is Off" crlf)
    )
)

(defrule ConvBeltErrorReset
    ?e <- (errormessage (message conv-belt-off) (value 1))
    (state (component ConveyorBelt) (value 1))
    =>
    (modify ?e (value 0))
)
;---------------------------------------------------------------------
;  timeout ready2go
;---------------------------------------------------------------------
```

```
(defrule logreadytime
    (controlfact (message ready2go))
    =>
    (assert (time-ready (time)))
)
(defrule RestarttimeoutTimer1
    (controlfact (message reach-object))
    ?tr <- (time-ready ?)
    =>
    (retract ?tr)
)

(defrule logElapsedTime
    (controlfact (message ready2go))
    (state (component RobotArm) (value 0))
    =>
    (assert (time-elapsed (time)))
)

(defrule RestarttimeoutTimer0
    (controlfact (message reach-object))
    ?tt <- (time-elapsed ?)
    =>
    (retract ?tt)
)

(defrule timeoutErrorReadyToGo
    ?e <- (errormessage (message timeout-ready2go) (value 0))
    ?tt <- (time-elapsed ?tti)
    ?tr <- (time-ready ?tri)
    =>
    ( if (< 1 (- ?tti ?tri))
      then  (printout t "  !ERROR : Timeout (Ready2go): " (* (- ?tti ?tri) 10)
              " seconds" crlf )
        (printout all_messages (time) "  !ERROR : Timeout (Ready2go):
            " (* (- ?tti ?tri) 10) " seconds" crlf )
        (printout error_messages (time) "  !ERROR : Timeout (Ready2go):
            " (* (- ?tti ?tri) 10) " seconds" crlf )
        (modify ?e (value 1))
    )
    (retract ?tt)
)

(defrule RestarttimeoutTimer2
    (controlfact (message reach-object))
    ?e  <- (errormessage (message timeout-ready2go) (value 1))
    =>
    (modify ?e (value 0))
)


;---------------------------------------------------------------
;  Obj det sys not ready
```

```
;-------------------------------------------------------------

(defrule ObjDetSysNRe
    ?e <- (errormessage (message obj-det-sys-not-ready) (value 0))
    (state (component GripperPosition) (value ?f2))
    (state (component RobotArm) (value ?f1))
    (state (component ObjectDetection) (value 1))
    (test (neq ?f1 0))
    =>
    (printout t "  !ERROR: Object Detected before System Ready" crlf)
    (printout all_messages (time)
        "  !ERROR: Object Detected before System Ready" crlf)
    (printout error_messages (time)
        "  !ERROR: Object Detected before System Ready" crlf)
    (modify ?e (value 1))
)

(defrule ObjDetErrorReset
    ?e <- (errormessage (message obj-det-sys-not-ready) (value 1))
    (controlfact (message reach-object))
    =>
    (modify ?e (value 0))
)
```

A.2 Python Code: monitoring.py

```
import clips
import time
import scipy.io
import numpy
from pylab import *
from matplotlib import *
# RobotArmState = 0.0
#   0:Idle  1:Moving 2:Lifting 3:Pushing 4:NotReady
# GripperPosition = 0.0
#   0:NotStart 1:Start  2:Moving  3:Object   4:CTbegin 5:CTend
# ObjectInGripper = 0.0
#   0:empty    1:full
# GripperState = 0.0
#   0:open     1:closed
# ConveyorBelt = 0.0
#   0:off 1:on
# ObjectDetection = 0.0
#   0:object-not-seen 1:object-seen

def main():
  stime = 0.01 #set sleeptimer
  clips.Reset()
  clips.BatchStar("monitoring.clp")

  x = scipy.io.loadmat('obj_det_sys_not_ready.mat')
  run_scenario(x,stime)
  print '\nfinish!'
```

```
    x = scipy.io.loadmat('one_run_scenario.mat')
    run_scenario(x,stime)
    print '\nfinish!'

    x = scipy.io.loadmat('timeout_ready2go.mat')
    run_scenario(x,stime)
    print '\nfinish!'

    x = scipy.io.loadmat('conv_belt_off.mat')
    run_scenario(x,stime)
    print '\nfinish!'

  def run_scenario(x,stime):
    pyRAS = x['robot_arm_state']
    pyGP  = x['gripper_position']
    pyGS  = x['gripper_state']
    pyCB  = x['conveyor_belt']
    pyOD  = x['object_detection']
    n = pyOD.shape[1]
    for i in range(n):
      a0 = "(RAS %d)" % pyRAS[0,i]
      clips.Assert(a0)
      a1 = "(GP %d)"  % pyGP[0,i]
      clips.Assert(a1)
      a3 = "(GS %d)"  % pyGS[0,i]
      clips.Assert(a3)
      a4 = "(CB %d)"  % pyCB[0,i]
      clips.Assert(a4)
      a5 = "(OD %d)"  % pyOD[0,i]
      clips.Assert(a5)
      time.sleep(stime)
      clips.Run()
      s = clips.StdoutStream.Read()
      if (s != None):
        print s
  #---------------------------------------------------------------
  #  main()
  #---------------------------------------------------------------
  main()
```

A.3 Output

In the code, the timeout error checks for 10 seconds, which is 1.0 second in the system time. This can be seen by comparing the timing of the "System is READY" to the "!ERROR: Timeout" message.

A.3.1 error_messages.txt

```
0.344033  !ERROR: Object Detected before System Ready
1.908485  !ERROR : Timeout (Ready2go): 10.01375 seconds
2.6874  !ERROR: Conveyor Belt is Off
```

A.3.2 all_messages.txt

```
0.3305  STARTUP Complete
0.330563  OPENGRIPPER Complete
0.330619  !WARNING: Conveyor Belt is Off
0.332675  MOVETOSTARTPOS Complete
0.332723  (waiting: 1.2 s)
0.343962  OBJECT SEEN
0.344015  !ERROR: Object Detected before System Ready
0.357439  System is READY
0.501843  OBJECT SEEN
0.568583  OBJECT REACHED
0.580131  OBJECT GRABBED
0.587847  OBJECT LIFTED
0.647548  ROBOT ARM PUSHING
0.711642  OBJECT ON SPIKE
0.901395  BACK TO START
0.902995  OPENGRIPPER Complete
0.903767  MOVETOSTARTPOS Complete
0.904394  (waiting: 1.2 s)
0.905566  System is READY
1.908438  !ERROR : Timeout (Ready2go): 10.01375 seconds
2.687364  !ERROR: Conveyor Belt is Off
```

A.4 MatLab Code: simulated_robotic_system.m

This code has to be run in MatLab in order to create the five .mat files that make up the simulation environment.

```
clc
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ------ One Run Scenario ---------------------%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
st = 1;
en = 100;
t = linspace(1,100,en);

robot_arm_state = zeros(1,en-st+1);
robot_arm_state(1, 1:end) = 0;
robot_arm_state(1, 10:end) = 4;
robot_arm_state(1, 22:end) = 0;
robot_arm_state(1, 34:end) = 1;
robot_arm_state(1, 49:end) = 2;
robot_arm_state(1, 60:end) = 3;
robot_arm_state(1, 70:end) = 0;
robot_arm_state(1, 76:end) = 1;
robot_arm_state(1, 84:end) = 4;
robot_arm_state(1, 96:end) = 0;

gripper_position = zeros(1,en-st+1);
gripper_position(1, 1:end) = 2;
```

```
gripper_position(1, 10:end) = 1;
gripper_position(1, 34:end) = 2;
gripper_position(1, 46:end) = 3;
gripper_position(1, 49:end) = 2;
gripper_position(1, 60:end) = 4;
gripper_position(1, 70:end) = 5;
gripper_position(1, 76:end) = 2;
gripper_position(1, 84:end) = 1;

object_in_gripper = zeros(1,en-st+1);
object_in_gripper(1, 1:end) = 0;
object_in_gripper(1, 46:end) = 1;
object_in_gripper(1, 76:end) = 0;

gripper_state = zeros(1,en-st+1);
gripper_state(1, 1:end) = 0;
gripper_state(1, 48:end) = 1;
gripper_state(1, 74:end) = 0;

conveyor_belt = zeros(1,en-st+1);
conveyor_belt(1, 1:end) = 1;
conveyor_belt(1, 2:end) = 1;

object_detection = zeros(1,en-st+1);
object_detection(1, 1:end) = 0;
object_detection(1, 33:end) = 1;
object_detection(1, 34:end) = 0;

figure(1)
n = 5;
subplot(n,1,1)
plot(t, robot_arm_state,'b-d')
grid on
axis([1 100 -0.5 4.5])
xlabel('time [1/10 sec]')
ylabel('Robot Arm State')

subplot(n,1,2)
plot(t, gripper_position,'b-d')
xlabel('time [1/10 sec]')
ylabel('Gripper Position')
grid on
axis([1 100 -0.5 5.5])

subplot(n,1,3)
plot(t, gripper_state,'b-d')
xlabel('time [1/10 sec]')
ylabel('Gripper State')
grid on
axis([1 100 -0.4 1.4])

subplot(n,1,4)
plot(t, conveyor_belt,'b-d')
xlabel('time [1/10 sec]')
```

```
ylabel('Conveyor Belt')
grid on
axis([1 100 -0.4 1.4])

subplot(n,1,5)
plot(t, object_detection,'b-d')
xlabel('time [1/10 sec]')
ylabel('Object Detection')
grid on
axis([1 100 -0.4 1.4])


save('one_run_scenario',...
    'robot_arm_state',...
    'gripper_position',...
    'object_in_gripper',...
    'gripper_state',...
    'conveyor_belt',...
    'object_detection');


%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ------ Timeout Ready to Go -----------%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

st = 1;
en = 140;
t = linspace(1,en,en);

robot_arm_state = zeros(1,en-st+1);
robot_arm_state(1, 1:end) = 1;
robot_arm_state(1, 4:end) = 4;
robot_arm_state(1, 16:end) = 0;

gripper_position = zeros(1,en-st+1);
gripper_position(1, 1:end) = 2;
gripper_position(1, 4:end) = 1;

object_in_gripper = zeros(1,en-st+1);
object_in_gripper(1, 1:end) = 0;
subplot(n,1,3)

gripper_state = zeros(1,en-st+1);
gripper_state(1, 1:end) = 0;

conveyor_belt = zeros(1,en-st+1);
conveyor_belt(1, 1:end) = 1;

object_detection = zeros(1,en-st+1);
object_detection(1, 1:end) = 0;

save('timeout_ready2go',...
    'robot_arm_state',...
```

```
    'gripper_position',...
    'object_in_gripper',...
    'gripper_state',...
    'conveyor_belt',...
    'object_detection');

figure(2)
n = 5;
subplot(n,1,1), hold on
plot(t, robot_arm_state,'m-d')
plot(116,2,'ro','LineWidth',4)
grid on, axis([1 140 -0.5 4.5])
xlabel('time [1/10 sec]'), ylabel('Robot Arm State')

subplot(n,1,2),hold on
plot(t, gripper_position,'m-d')
plot(116,3,'ro','LineWidth',4)
xlabel('time [1/10 sec]')
ylabel('Gripper Position')
grid on, axis([1 140 -0.5 5.5])

subplot(n,1,3)
plot(t, gripper_state,'b-d')
xlabel('time [1/10 sec]')
ylabel('Gripper State')
grid on, axis([1 140 -0.4 1.4])

subplot(n,1,4)
plot(t, conveyor_belt,'b-d')
xlabel('time [1/10 sec]')
ylabel('Conveyor Belt')
grid on, axis([1 140 -0.4 1.4])

subplot(n,1,5)
plot(t, object_detection,'b-d')
xlabel('time [1/10 sec]')
ylabel('Object Detection')
grid on, axis([1 140 -0.4 1.4])
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ------ Object Detected System Not Ready ----%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

st = 1;
en = 30;
t = linspace(1,en,en);

figure(3)
robot_arm_state = zeros(1,en-st+1);
robot_arm_state(1, 1:end) = 1;
robot_arm_state(1, 10:end) = 4;
robot_arm_state(1, 22:end) = 0;

gripper_position = zeros(1,en-st+1);
```

```
gripper_position(1, 1:end) = 2;
gripper_position(1, 4:end) = 1;

object_in_gripper = zeros(1,en-st+1);
object_in_gripper(1, 1:end) = 0;

gripper_state = zeros(1,en-st+1);
gripper_state(1, 1:end) = 0;

conveyor_belt = zeros(1,en-st+1);
conveyor_belt(1, 1:end) = 1;

object_detection = zeros(1,en-st+1);
object_detection(1, 1:end) = 0;
object_detection(1, 15:end) = 1;
object_detection(1, 16:end) = 0;

save('obj_det_sys_not_ready',...
    'robot_arm_state',...
    'gripper_position',...
    'object_in_gripper',...
    'gripper_state',...
    'conveyor_belt',...
    'object_detection');

n = 5;
subplot(n,1,1), hold on
plot(t, robot_arm_state,'m-d')
plot(15,2,'ro','LineWidth',4)
grid on
axis([1 30 -0.5 4.5])
xlabel('time [1/10 sec]')
ylabel('Robot Arm State')

subplot(n,1,2)
plot(t, gripper_position,'b-d')
xlabel('time [1/10 sec]')
ylabel('Gripper Position')
grid on
axis([1 30 -0.5 5.5])

subplot(n,1,3)
plot(t, gripper_state,'b-d')
xlabel('time [1/10 sec]')
ylabel('Gripper State')
grid on
axis([1 30 -0.4 1.4])

subplot(n,1,4)
plot(t, conveyor_belt,'b-d')
xlabel('time [1/10 sec]')
ylabel('Conveyor Belt')
grid on
axis([1 30 -0.4 1.4])
```

```
subplot(n,1,5),hold on
plot(t, object_detection,'m-d')
plot(15,0.5,'ro','LineWidth',4)
xlabel('time [1/10 sec]')
ylabel('Object Detection')
grid on
axis([1 30 -0.4 1.4])
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ------ conv_belt_off ----%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

st = 1;
en = 20;
t = linspace(1,en,en);

figure(4)
robot_arm_state = zeros(1,en-st+1);
robot_arm_state(1, 1:end) = 1;
robot_arm_state(1, 4:end) = 4;
robot_arm_state(1, 16:end) = 0;

gripper_position = zeros(1,en-st+1);
gripper_position(1, 1:end) = 2;
gripper_position(1, 4:end) = 1;

object_in_gripper = zeros(1,en-st+1);
object_in_gripper(1, 1:end) = 0;

gripper_state = zeros(1,en-st+1);
gripper_state(1, 1:end) = 0;

conveyor_belt = zeros(1,en-st+1);
conveyor_belt(1, 2:end) = 0;

object_detection = zeros(1,en-st+1);
object_detection(1, 1:end) = 0;

save('conv_belt_off',...
    'robot_arm_state',...
    'gripper_position',...
    'object_in_gripper',...
    'gripper_state',...
    'conveyor_belt',...
    'object_detection');

n = 5;
subplot(n,1,1)
plot(t, robot_arm_state,'b-d')
grid on
axis([1 20 -0.5 4.5])
xlabel('time [1/10 sec]')
ylabel('Robot Arm State')
```

```
subplot(n,1,2)
plot(t, gripper_position,'b-d')
xlabel('time [1/10 sec]')
ylabel('Gripper Position')
grid on
axis([1 20 -0.5 5.5])

subplot(n,1,3)
plot(t, gripper_state,'b-d')
xlabel('time [1/10 sec]')
ylabel('Gripper State')
grid on
axis([1 20 -0.4 1.4])

subplot(n,1,4),
plot(t, conveyor_belt,'r-o','LineWidth',2)
xlabel('time [1/10 sec]')
ylabel('Conveyor Belt')
grid on
axis([1 20 -0.4 1.4])

subplot(n,1,5)
plot(t, object_detection,'b-d')
xlabel('time [1/10 sec]')
ylabel('Object Detection')
grid on
axis([1 20 -0.4 1.4])
```

Bibliography

[1] M. Lind, "An introduction to multilevel flow modeling," *Nuclear Safety and Simulation, International Electronic Journal of Nuclear Safety and Simulation, 2(1), 22–32*, 2011.

[2] M. Lind, "Control functions in MFM," *Nuclear Safety and Simulation, International Electronic Journal of Nuclear Safety and Simulation, 2(2), 132–140*, 2011.

[3] M. Lind and K. Heussen, "Representing causality and reasoning about controllability of multi-level flow-systems," *Ieee International Conference on Systems, Man and Cybernetics, Ieee Sys M, Ieee Sys Man Cybern, Ieee International Conference on Systems Man and Cybernetics Conference Procee, Ieee International Conference on Systems, Man, and Cybernetics.*, 2010.

[4] M. Lind, "Promoting and opposing: A semantic analysis of von wright's action types." Unpublished lecture note, December 2012.

[5] M. Lind, "Foundations of functional modelling for engineering design concepts of means and ends." Unpublished lecture note, April 2010.

[6] J. Giarratano, *CLIPS User's Guide.* clipsrules.sourceforge.net, 6.3 ed.

[7] M. Lind, "Notes on hierarchy, heterarchy and distribution." Unpublished lecture note for course 31372, February 2015.

[8] M. Lind, "Levels of execution." Unpublished lecture note for course 31372, May 2009.

[9] R. Harré, *Causal Powers: Theory of Natural Necessity.* Rowman and Littlefield, first ed., 1975.

[10] J. Giarratano and G. Riley, *Expert Systems: Principles and Programming.* PWS, third ed., 1998.