UNIVERSITY OF OSLO
Department of Informatics

A Case Study using SysML for safety-critical systems

Master thesis
60 credits

Tonje Elisabeth Klykken

November 9th, 2009

© Tonje Elisabeth Klykken
2009
A Case Study using SysML for safety-critical systems
Tonje Elisabeth Klykken
http://www.duo.uio.no/
Printed by: Reprosentralen, Universitetet i Oslo

Abstract

This thesis presents a case study which is based on our experience and lessons learnt from modelling a control system using the state-of-the-art modelling language for systems engineering, Systems Modelling Language (SysML). The goals of this thesis are to (1) capture the structure and behaviour of a control system using SysML, (2) handling the development of safety requirements, (3) support generation of safety cases, a structured collection of arguments for system safety, by creating traceability links between requirements and model elements, (4) assess SysML capabilities in modelling control systems and supporting generation of safety cases.

This case study is part of the “ModelME!” project which is conducted at Simula Research Laboratory with industry partners. The aim of the “ModelME!” project is to devise better software engineering practices for Integrated Software-Dependent Systems in the Maritime & Energy sectors.

Based on the experiences of this and other simultaneous projects in the “ModelME!” project, a methodology for modelling control systems to support safety certification has been proposed. We use this methodology to present the SysML model, developed in this case study. The methodology takes a systematic approach and guides us through the process of designing a control system, from the first steps of capturing requirements, system functionality and environmental assumptions through the development of structural and behavioural diagrams and last, but not least the modelling of safety design, developing the requirements to avoid ambiguity and tracing model structures to the requirements.

In this thesis we create a comprehensive set of models to capture our case study from requirement, structure and behaviour points of view and present these models following the methodology mentioned above. We create traceability links between the requirements and design model elements/slices with the goal of assisting safety engineers in the generation of safety cases. Then we discuss the capabilities of SysML and our chosen tool regarding the creation of models for control systems and supporting safety case generation. Further we summarize lessons learned, potential improvements and directions for future work.

Acknowledgements

There are many people that have helped and inspired me, whom I wish to thank.

First of all, I want to thank Line Valbø at the Department of Informatics for all your help and support throughout the years. I wish to thank all the wonderful people at Simula Research Laboratory, especially my supervisor, Lionel Briand. I also wish to thank Shaukat Ali and Rajwinder Panesar-Walawege for all your support, and especially Shiva Nejati, for all your support and wisdom throughout the work of this thesis.

I wish to thank IBM and EmbeddedPlus for providing us with academic licences for the application Rational Software Architect and the EmbeddedPlus SysML Toolkit which we needed to conduct this case study.

To all my friends, thank you so much for always being there and supporting me. Thank you also to my mother, father and brother for all your love and support when I went back to school and throughout these years. Last, but not least, I am forever grateful for the support from my grandma, Ragnhild Klykken, who passed away last year. I cannot put into words what you mean to me. The love and support that you have always shown me and the wisdom you have shared with me both regarding school and life; I will always carry it with me. Thank you.

Tonje Elisabeth Klykken,
7th of November, 2009.

Table of contents

Table of figures

1 Introduction

The use of software is increasing in all aspects of our everyday lives, e.g. in simple home appliances and in safety-critical systems such as control systems in cars, medical instruments, nuclear power plants, military equipment and air planes. In the field of safety-critical systems, the demand of dependable software is immense. The cost of lack of dependability is often huge [1], sometimes leading to injuries or, even worse, jeopardizing human lives [2].
It is of upmost importance that the software does not compromise the safety for the system as a whole. To ensure that a system complies with a given safety standard, and hence will not cause harm to its users or its environment, the system may be subject to a certification process before it is deployed. For safety-critical systems, certification may be mandatory, depending on the system. The standards a system must comply with may be organizational or governmental, national or international, depending on the system and its use, e.g. for critical systems as in the aviation industry, there exist standards issued by the EASA[1] which *"ensure a high and uniform level of safety in civil aviation, by the adoption of common safety rules and measures"* [3]. The certification process may be conducted by third-party organizations, e.g. the international certification agency, DNV[2]. The system supplier must prepare a chain of evidence for the certification agency by providing a set of safety cases/arguments to assure the certifiers that the system meets the safety requirements as stated in the applicable standards.

According to a report from 2007 [2], there is a normal misconception in the general public that failures in software are mostly due to code errors. They state that from the number of fatal accidents caused by software, only 3 percent were due to code errors, and a larger portion are caused by poor recording and handling of requirements, and misunderstanding the user's domain. The latter may cause in an insufficient coverage of safety requirements and a lack of environmental assumptions, which may be disastrous when the system is deployed into its environment. In addition, requirements should ideally be unambiguous. However, as requirements most often are expressed textually, they can be ambiguous and unclear, which potentially can cause misunderstandings, leading to unforeseen errors in the system design.

The same report [2], also states that there is *"a widespread agreement that only a small percentage of projects deliver the required functionality, performance, and dependability within the original time and cost estimate"* and that creating safety cases after the system is created is costly and not as practical as during development.

[1] European Aviation Standard Agency, http://www.skybrary.aero/index.php/EASA
[2] Det Norske Veritas, www.dnv.no.

To be able to overcome the cost of creating dependable systems and to ensure that the systems are safe, we need to develop systems with safety in mind, and strive to trace requirements to the relevant system design rationales and decisions made during development. This facilitates a foundation that supports the generation of safety cases which are vital for a certification process.

This thesis presents a case study which is based on our experience and lessons learnt from modelling a control system using the state-of-the-art modelling language for systems engineering, Systems Modelling Language (SysML). The goals of this thesis are to (1) capture the structure and behaviour of a control system using SysML, (2) handling the development of safety requirements, (3) support generation of safety cases, a structured collection of arguments for system safety, by creating traceability links between requirements and model elements, (4) assess SysML capabilities in modelling control systems and supporting generation of safety cases.

This case study is part of the "ModelME!" project[3] which is conducted at Simula Research Laboratory with industry partners. The aim of the "ModelME!" project is to devise better software engineering practices for Integrated Software-Dependent Systems in the Maritime & Energy sectors.

Based on the experiences of this and other simultaneous projects in the "ModelME!" project, a methodology for modelling control systems to support safety certification has been proposed[4]. We present the SysML model, developed in this case study, using this methodology. The methodology takes a systematic approach and guides us through the process of designing a control system, from the first steps of capturing requirements, system functionality and environmental assumptions through the development of structural and behavioural diagrams and last, but not least the modelling of safety design, developing the requirements to avoid ambiguity and tracing model structures to the requirements.

Our design is based on an object-oriented version of the Production Cell case which was presented in a case study[5] from 1998, which focused on using a specific testing method to test the system.

SysML was adopted by OMG in 2006 [6] and is a subset and extension of the Unified Modelling Language (UML 2.0). We use SysML because it is a state-of-the-art modelling language for systems engineering, and of its capabilities for modelling requirements and relating them to elements of the designed model. We hope to show that SysML can be used to support generation of safety arguments/cases for certification purposes.

[3] http://modelme.simula.no

1.1 Contributions

The contributions of this thesis are:

- We provide a detailed, comprehensive collection of SysML diagrams for a controller of a medium-sized reactive system.
- Presentation of a methodology which has been proposed based on this case study and another simultaneous one. Each of the methodology steps are presented using examples from the case study diagrams.
- We show how SysML traceability can be used for linking safety requirements to design diagrams. We then utilize these traceability links to identify fragments of design diagrams specifically relevant to each safety requirement. Such diagrams can be used for generation of safety cases.
- We offer an evaluation of the SysML language and the tool, Rational Software Architect with the SysML Toolkit plug-in for capturing control systems with safety features.

1.2 Organization of thesis

This thesis is organized as follows:
In Chapter 2 we give a quick overview of the new or changed diagrams and concepts of SysML compared with UML 2.0, which are relevant for this thesis. For further details about SysML, see [7] and [8]. We then describe the Production Cell system in Chapter 3. In Chapter 4 we present the proposed modelling methodology by giving examples from the SysML model we have designed. In Chapter 5, we present some of the tool problems we met during development. In Chapter 6, we give a short discussion of lessons learned, conclude and give some suggestions for further work.

In Appendix A, we present all the diagrams from the case study. In Appendix B, we present a glossary of the structure blocks in the model. In Appendix C, we give an overview over the tool problems which are not described in Chapter 5.

2 System Modelling Language

System Modelling Language (SysML) was adopted by OMG in 2006, and is a subset of the Unified Modelling Language (UML 2.0) with some additional extensions, as described in Figure 1. The purpose of SysML is to have a general graphical notation for systems engineering that can be used for specification, analysis, design and verification of complex systems that contain hardware, software, personnel, information [6].

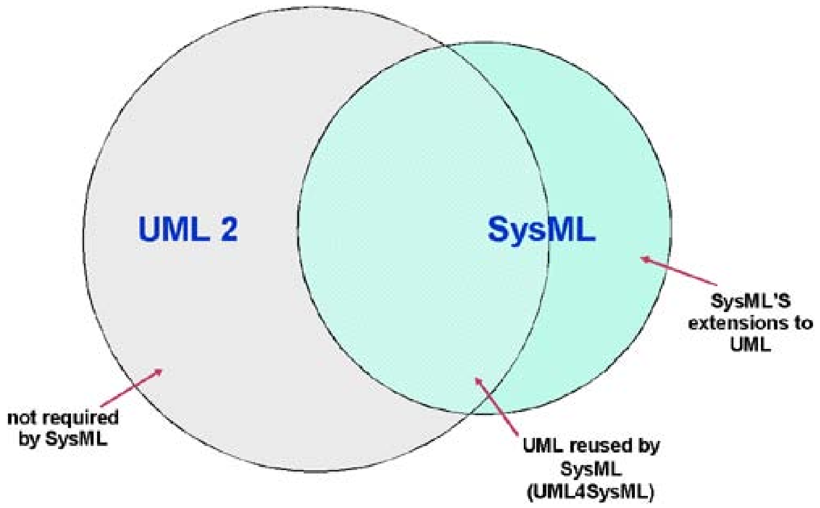


Figure 1 - SysML is a subset of UML 2 with an additional extensions. [6]

The SysML diagrams consist of some of the diagrams from UML 2.0 and some additional diagrams (see Figure 2). The new diagrams are the Requirement Diagram, where we can capture/develop the requirements and relate them to other requirements or model elements, and the Parametric diagram, where we can capture constraints for property values. The Activity Diagram, Internal Block Diagram and Block Definition Diagram are modifications of existing diagrams in UML 2.0. We will give a quick introduction to the new and changed diagrams and constructs relevant to this case study below, and some details will be further described later in this thesis, when we go though the methodology and show examples from the conducted case study.

2.1 Block Definition Diagram

In SysML, there is a concept called *block*, which is *"modular unit of system description"* [7]. Blocks are used to describe structural concepts in a system and its environment, (e.g. logical and physical elements in the system, software, hardware, humans, documents and more). A block can contain structural and behavioural features, which are described as properties and operations. The block and its features, in addition to its relationships with other blocks are defined in a **Block Definition Diagram** (BDD). A block may contain other blocks, which are called *parts*. This is described in the BDD using the *composition* relationship. In the part end of the relationship we may specify the *role* that this part has in the owning block. This role name is *"provided to be able to distinguish part properties with the same type (Block)"* [8]. In addition to modelling the structure of blocks, we can also use this diagram to show other types of blocks and entities, e.g. signals, interfaces and *ValueTypes*, which are described later in this chapter. We describe examples of a block definition diagrams in chapter 4.2.1.

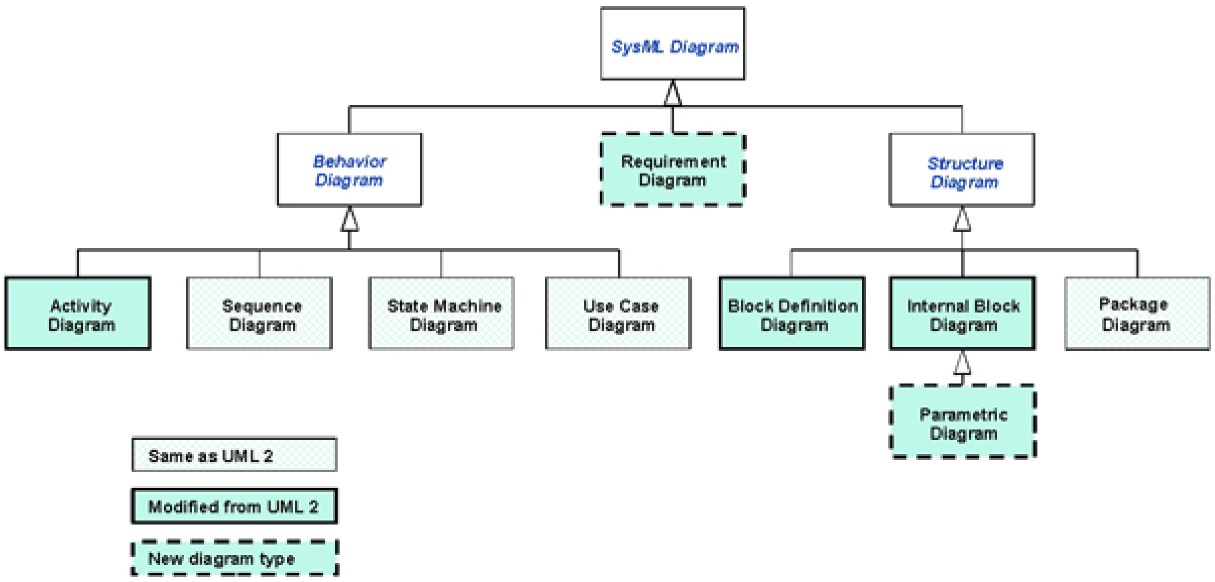


Figure 2 – The SysML diagrams and their relationship with UML diagrams. [6]

2.2 Internal Block Diagram

The internal structure of a composite block is described using an **Internal Block Diagram** (IBD), where the connections/interactions between the parts of a composite block and the usage of the parts can be defined. We can specify how the parts of a composite block are connected and specify their interaction points, by (1) describing the flow of items (e.g. physical items, data or information) using *flow ports* and *item flows*, and/or (2) describing provided/required services, as in UML 2.0, using interfaces.
There may be direct connections between the parts using a connector, which will show that the parts are connected, but not describe how (e.g. for physical parts, this can mean that they are attached physically).
To describe an interaction point used for item flows, we use *Flow Ports* and connect these using *Item Flow* connectors. If only one item is allowed to flow through the port, then it is an *atomic* flow port and can be typed by a block that defines the item. However, if several items are allowed to flow through the flow port, it can be typed by a *flow specification*, where the allowed items are specified (similar to an interface from UML 2.0). The connection between the flow ports is defined using *item flow*, which describes what actually flows through the ports and is typed accordingly.

When describing which signals a block must respond to, or which services it must provide, we use *standard ports* and *interfaces***.** This is similar to UML 2.0. We describe examples of internal block diagrams in chapter 4.2.4.

2.3 Requirement diagram

One of the new concepts in SysML is the *Requirement Block*. We use it to define conditions or functions that we want the system to satisfy. Each requirement block contains at least an id property and a textual description property. The requirements are depicted visually in a **requirement diagram** (REQ). When tracing them to other model elements, using the cross-cutting relationships described in **Table 1**, they can also be shown in other diagrams. We can capture requirements blocks by inspecting e.g. safety standards or stakeholder's needs. After requirements are captured, they may be further analysed and then partitioned, derived or refined to make them less ambiguous and more specific. In SysML we can keep track of the requirement development history by relating new requirements to the old ones using the relationships described in **Table 1** and optionally attaching a rationale, which is a stereotyped comment to justify any assumptions made during the requirement development. This way, the assumptions can be inspected by domain experts or by others to ensure that there have not been made erroneous assumptions that may lead to violation of safety. Examples of requirement diagrams are shown in chapters 4.1.2 and 4.2.3.

Relationship	Explanation
Containment	Used to break a requirement into simpler requirements. All the contained requirements should not add or remove any meaning to the original requirement.
Derive	Used to depict that a requirement is derived from another requirement. Useful for mapping the assumptions made about the system based on the requirements.
Cross-cutting relationships:	
Trace	Used to describe a general-purpose relationship, often used to relate a requirement to external documents.
Satisfy	Used to link a model element to a requirement to show that it satisfies the requirement.
Verify	Used to link a test case to a requirement to prove that a model element satisfies it.
Refine	Used to depict that a model element is refined from a requirement. Useful to map assumptions made to reduce ambiguity in a requirement.

Table 1 - Requirement relationships/dependencies.

2.4 Activity Diagram

In an **activity diagram** we are able to show the sequential and concurrent execution of actions, as in UML 2.0 [7]. An *activity* is composed of actions and describes behaviour of the system by specifying a given sequence to those actions [6]. By associating a special type of action, a *CallBehaviourAction*, to an activity, we are able to call an activity from within another. This allows for activities to be composed of each other and supports reuse. The composition of activities can be shown in a **block definition diagram** [7] as in Figure 20.
An activity can have input/output parameters, and these may flow through the activity using object flows. To impose a constraint on the sequence of the actions within the activity [8], we use control flows.
Inputs and outputs of actions are called tokens. An action may not start before there are tokens (object/control tokens) at each of the required inputs and it may not terminate before there are tokens on each of the activity's required outputs. However, if an owning activity is terminated, its contained actions/activities are terminated too.
An optional feature is allocating activities/actions to blocks. This allows us to specify which part will execute it. To do this, we can use the *AllocateActivityPartition* which resembles swim lanes in UML 2.0 or use the allocate relationship.

SysML also introduces the concept 'continuous' to describe the special case where the time between receiving tokens is zero (e.g. for information, power etc.) [8]. This means that an activity can start to execute and input tokens may still be consumed and tokens may become available at the outputs during execution of the activity. The flow of objects through the activity may in SysML also be physical objects. We show examples of activity diagrams in chapters 4.2.5 and 4.2.7.

2.5 Parametric Diagram

Constraint blocks are another of the new concepts that SysML introduces. By using the constraint blocks we are able to constrain properties of blocks in the system. We define constraints by creating constraint blocks and show the composition relationship with blocks and other constraint blocks in a **block definition diagram** as shown in Figure 24. Constraints may be composite, which allows us to reuse common equations in several constraint blocks. After defining the constraint block, we can define the parameters needed by the constraint block to be able to solve the equation that the constraint is based on. The equation can be specified in any constraint language that is suitable for the domain of the system[8].

The parametric diagram can be created either for blocks that have a constraint property (has a composition relation with a constraint block) or for composite constraint blocks. Here we can bind the parameters of the constraint block to the properties of the owning block (or constraint block) by using binding connectors, which shows an equality relationship between the ends of the association, i.e. they will have the same value. We show an example of a parametric diagram in chapter 4.2.6.

2.6 ValueType

Even though *SysML ValueType* is not a particular diagram, it is a part of SysML, and is relevant to this case study, so we will briefly explain what it is and its use.
SysML ValueType is based on *UML DataType* and is defined in a **block definition diagram.** *ValueTypes* allow us to define our own specific values types which can be used to type e.g. properties or parameters within a model.
For instance, during development, we may want a specific type to describe the extension of a robot arm. It may move between a maximum and a minimum extension point, which can be described with a real number between 0 and 1. To be specific and to separate it from other real numbers, we create a value type *Extension* and base it on the Real data type and can also add specific operations for the value type. The value type can be used to type any property or parameter in the model. Value types may be scalar values, enumerations or complex structures (i.e. a structure with value properties, e.g. *Size*, with the properties *width* and *length.*)[8]. In this case study we only use the first two, as shown in Figure 3. We can base the value types on types from already defined model libraries, e.g. the SI Unit library, or create own libraries and import them into the packages where we wish to use them.

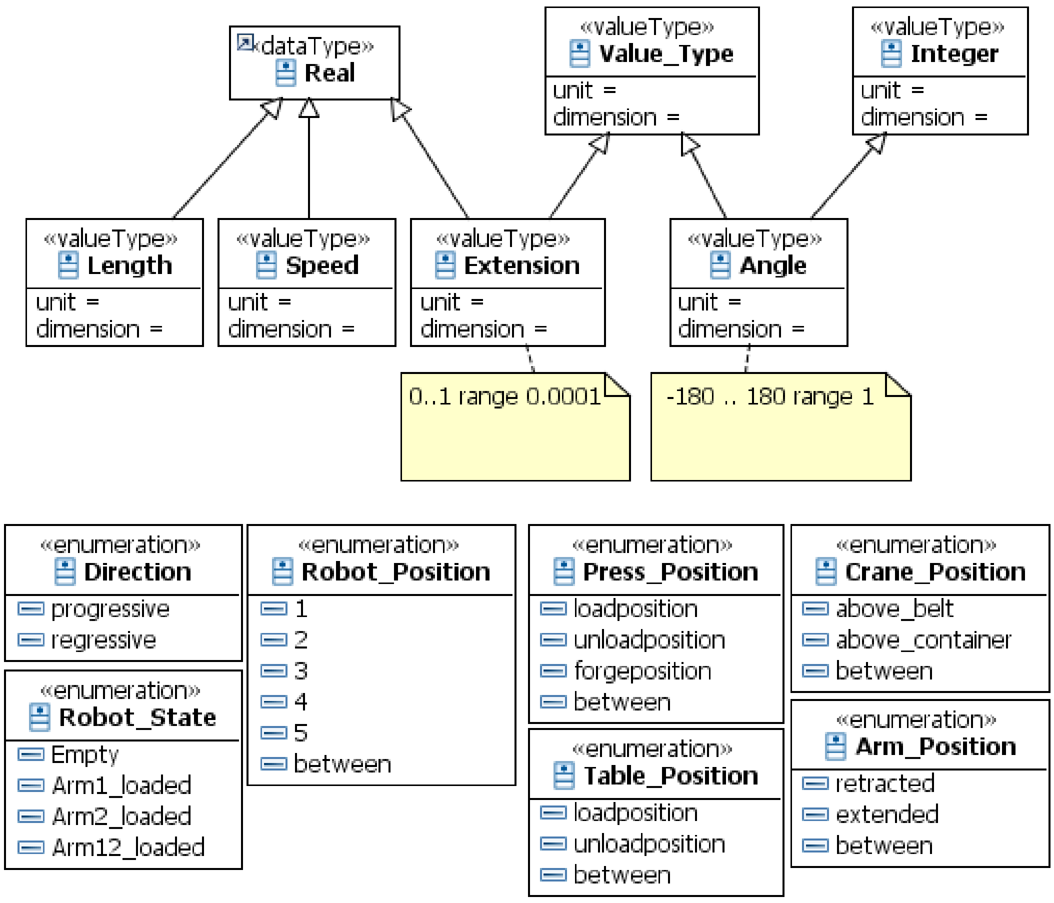


Figure 3 - bdd [Package] ValueTypes [ValueTypes and enumerations]

3 The Production Cell case study

To be able to conduct a case study modelling a safety-critical system in SysML, we need a system description that clearly describes the safety requirements and the system properties.
The case study [5] we chose as background for our case study contains a thorough description of safety requirements and the design of a production cell, whose main purpose is to forge metal plates. This case study is from 1998, and their goal was to use a specific testing method to test an object-oriented version of the production cell. It is based on the original case study [9], which was conducted in 1995, and evaluated the capability of a number of specification languages. The system described is an actual production cell in Germany, and has been the background for several papers since then.

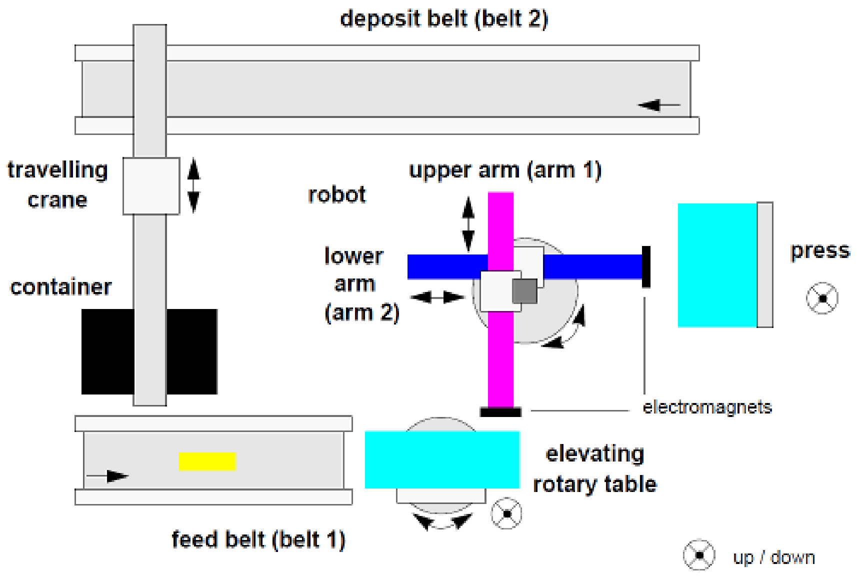


Figure 4 - Top view of the production cell.

The Production Cell consists of several physical devices which are controlled by the control system (see Figure 4), namely:

- a feed belt
- an elevating rotary table
- a rotary robot with two extendable robot arms
- a press
- a deposit belt
- a crane with an extendable robot arm

The feed belt conveys plates to *the elevating rotary table*, which moves the plate to a position for it to be picked up by one of the two extendable arms of *the robot*. *The robot* then loads the plate into *the press*, which forges the plate. The other arm of *the robot* then picks up the plate and transports it to *the deposit belt* which conveys the plate to the end of the belt where the arm of *the crane* picks it up and transports it into a container. Each of these devices will be described in more detail below.

3.1.1 Composition of the devices

Each of the devices consists of sensors and actuators which communicate with the system's controller. The sensors (photoelectric cells, switches and potentiometers) send input to the controller where they either indicate the position of the plate on a device or the rotary/vertical/horizontal position of the device or its parts. The actuators (motors and magnets) receive output from the controller, which activates them and makes them move, and thus the owning device is able to transport plates through the system.
To get a good overview of the system, we will now present describe each device and show what parts they consists of.

3.1.1.1 Feed belt

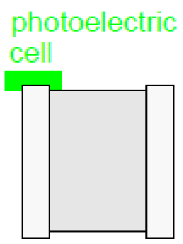


Figure 5 – Side view of feed belt.

The feed belt consists of a unidirectional motor, which makes the belt run and a photoelectric cell at the end of the belt which indicates when a plate has reached the end of the feed belt.
The feed belt gets plates from the operator and communicates with the table. It can carry one plate at a time.

Behaviour (simplified): After receiving a plate from the operator, the control system turns the motor on until the plate reaches the sensor field of the photoelectric cell, when it turns the motor off. The plate will reside at the end of the belt until the rotary table is ready to receive it.

3.1.1.2 Rotary table

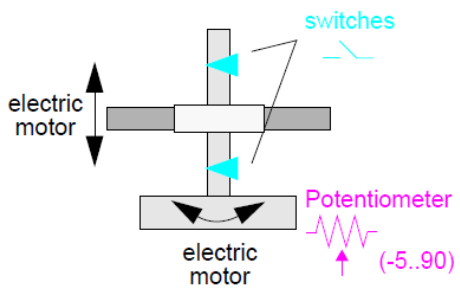


Figure 6 - Parts of rotary table.

The rotary table consists of a horizontal carrying plate, two bidirectional motors, which each take care of either the rotating or the vertical movement, two switches, which indicate the vertical position of the table (top/bottom) and a potentiometer which indicates the angle of the table.
The table communicates with the feed belt and the robot indicating whether it is ready to be loaded or unloaded. It can carry one plate at a time.

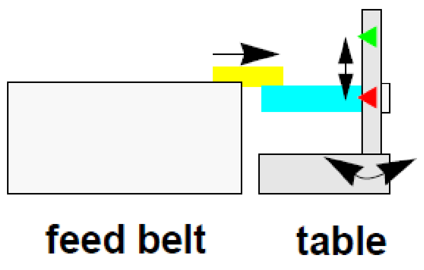


Figure 7 - Side view of feed belt and table.

Behaviour (simplified): When the table is ready to be loaded, the feed belt feeds the plate to the table, the table then moves to its unloading position and notifies the robot and awaits in this position until the robot is ready to pick up the plate and then returns to its unload position.

3.1.1.3 Rotary robot

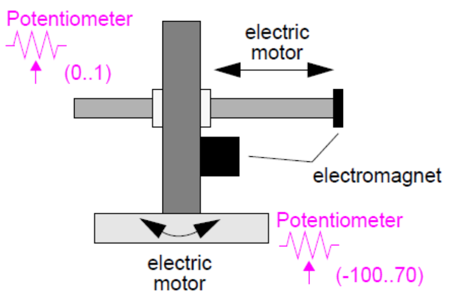


Figure 8 - Parts of rotary robot.

The robot consists of two orthogonal arms, a bidirectional motor, which takes care of the rotating movement and a potentiometer with indicates the angle of the robot. The orthogonal arms each consist of a bidirectional motor which extends or retracts the arm, a potentiometer which indicates the range of extension of the arm and a magnet which can be magnetized / demagnetized to pick/drop the plate.

The robot communicates with the table and the press and receives messages from the deposit belt. It can carry up to two plates at a time.

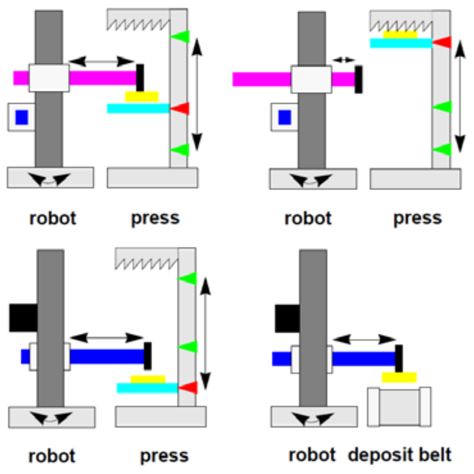


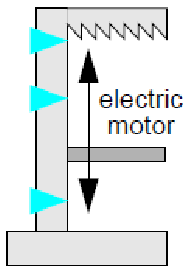
Figure 9 - Side view of robot, press and deposit belt.

Behaviour (simplified): The robot receives messages from the above mentioned machines when they are ready to be loaded or unloaded. It acts upon these messages when it is in a good state for this.

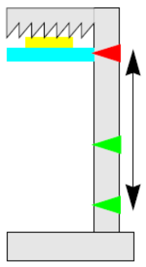
When picking up a plate, it turns on its rotating motor until the active arm points to the right machine at the same time that it turns on its extension motor to extend its arm to the right extension and then it activates its magnet to pick up the plate and retracts its arm.

When dropping a plate to a machine, it does the same as above, but deactivates its magnet once the arm is extracted to the right position.

3.1.1.4 Press

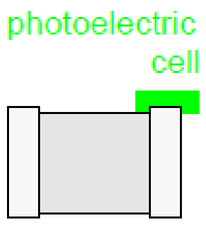


The press consists of one fixed horizontal top plate and one movable horizontal bottom plate which together forge the metal plate when moved together. The bottom plate is moved with the help of a bidirectional motor, and three switches are used to describe its vertical position (top / middle / bottom). The press communicates with the robot. It can carry one plate at a time.



Behaviour (simplified): When the robot has loaded a plate onto the press, the press starts its motor so that the movable plate moves in an upward direction until the top switch indicates that it is on top (by moving there, it forges the plate) and then it moves in the downward direction until the bottom switch indicates that it is in the bottom (unload) position. It will wait here until the robot has picked up the plate and then it will start its motor and move the movable plate in the upward direction.

3.1.1.5 Deposit belt

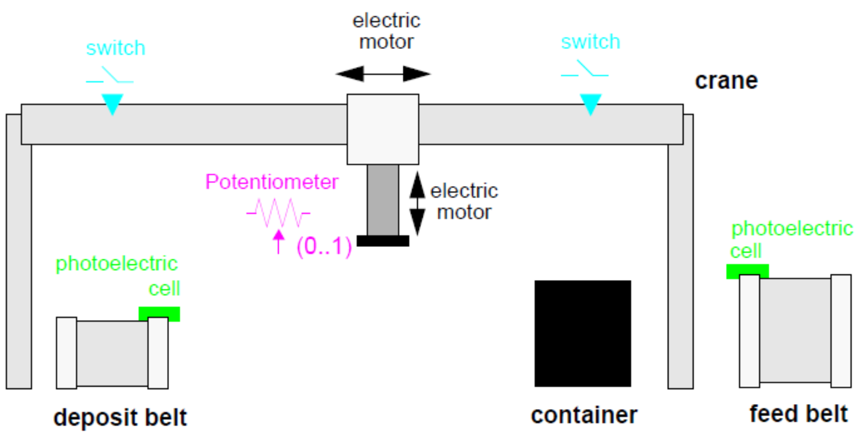


The deposit belt is nearly identical to the feed belt, the only difference is that the photoelectric cell indicates when the plate has passed the photoelectric field.
The deposit belt communicates with the crane and sends messages to the robot. It can carry up to two plates at a time.

Behaviour (simplified): When the crane has indicated that it has picked up the plate at the end of the belt, the deposit belt starts its motor, making the belt start running. When a plate passes the photoelectric cell, it stops the motor/belt and waits for the crane to pick it up and for the robot to drop a new plate at the beginning of the belt.

3.1.1.6 Crane

The crane consists of an arm, one bidirectional motor, which takes care of the horizontal movement of the arm and two switches which indicate the arm position (over deposit belt/over container). The arm is identical to the robot’s arms.
The crane communicates with the deposit belt. It can carry one plate at a time.



Behaviour (simplified): The idle position of the crane arm over the deposit belt with its arm extended. When a plate has arrived and the crane arm is in its idle position, the magnet is activated, and so the plate is picked up. The arm's extension motor retracts the arm until the potentiometer indicates that the correct position has been reached. At the same time the horizontal motor moves the arm towards the container until the switch above the container indicates that the arm is in the correct position. When the arm is in the right position the magnet will be deactivated so that the plate will be dropped into the container. The arm will then move back to its idle position over the deposit belt.

3.1.2 Asynchronous communication

An important detail, which strongly affects the design of the system, is that [5] describes that the communication between the devices is based on asynchronous events. Once a device is finished with an operation, it sends a message to one of the other devices, more specifically to let it know that the plate is ready to be picked up, or that it has been picked up from or delivered to the other device. When sending a signal, the sending block does not know whether the target is ready to act on the event, and the sender may immediately continue with its own behaviour. We model the asynchronous communication by using signals to exchange messages between the different devices. This is described further when we describe examples in the next chapter. We also assume that all signals that are sent, will arrive once and only once, and in reasonable time, as the handling of missing or duplicate signals is not described in [5].

4 Modelling methodology

Based on the experience and lessons learnt throughout this process of creating a SysML model for the Production Cell System in this case study, a methodology for modelling control systems to support safety certification has been created. In this chapter we will go through the steps of the methodology to present examples from our case study. The full case study can be found in Appendix A.

Figure 10 gives an overview over the methodology, its steps and which diagrams are produced in each step. It has two main phases, starting with the *system-level requirement specification phase* where the environmental entities and assumptions, the system-level requirements, and the main functionality of the system are captured. It is followed by the second phase, which is about s*ystem architecture and design, interwoven with safety-related activities*. This phase contains three concurrent groups of activities, namely designing and developing structure models, behavioural models, and requirement and safety design models. Some of the activities in these steps are dependent on diagrams developed in other activities. For each step, we specify the diagrams needed as input and which diagrams are produced or revisited. Further, we suggest a sequence of activities through the second phase, but during development, some of the previous steps may need to be revisited. When changes are made, it is important to update the succeeding diagrams that were produced in later steps, so that all the diagrams are consistent.

When creating a system, we have to put the environment under which the system will operate under consideration, as elements in the environment will influence the system design. Inputs to the methodology are therefore:

- **Standards:** There are standards (e.g. official, governmental, or corporate) that need to be fulfilled to be able to be certified and be allowed to operate.
- **Domain models:** The system will operate in an environment with an existing terminology that the system engineers need to be familiar with.
- **Requirements from stakeholder:** It is important to analyse the wishes and needs of the stakeholder to be able to extract the requirements. These reflect what the system will be used for, what things will be important to take care of. (E.g. timing, performance).

The methodology includes one or more iterations of the two main phases:

1) **System- level requirement specification**
    a) Create a system context diagram to determine the external entities that interact with the system.
    b) Capture system-level requirements, and specify the requirements category for each individual system-level requirements.
    c) Identify main system functions in terms of use cases, and relate these use cases to the system-level requirements.
    d) Identify the constraints that the external entities impose on the system, and include these constraints as system requirements.

2) **System architecture and design interwoven with safety-related activities**
    a) Create the structural views of the system using block definition diagrams by identifying top-level system blocks and decomposing them into sub-blocks.
    b) Model the use-case scenarios using sequence diagrams to describe behavioural interactions between the top-level parts.
    c) Specify block-level requirements and trace these requirements to the system-level requirements. Establish traceability links between the block-level requirements and top-level blocks.
    d) Describe flows of physical items and/or logical data among the top-level system parts using internal block diagrams.
    e) Model the sequences of activities for each of the top-level parts using activity diagrams.
    f) Capture the constraints on physical properties of blocks using parametric diagrams.
    g) Decompose each activity in the top-level activity diagram into a sequence of more primitive actions performed by the sub-parts.
    h) Capture the behaviour of individual top-level parts with behaviour using state machine diagrams.
    i) Specify pre- and post conditions for the methods of top-level blocks.
    j) Formalize block-level requirements in terms of block operations, predicates involving block attributes, and interfaces between blocks using the Object Constraint Language (OCL), or activity diagrams.
    k) For each block-level requirement, identify slices of design diagrams relevant to that requirement and provide information to evaluate the requirement on the diagram slices.

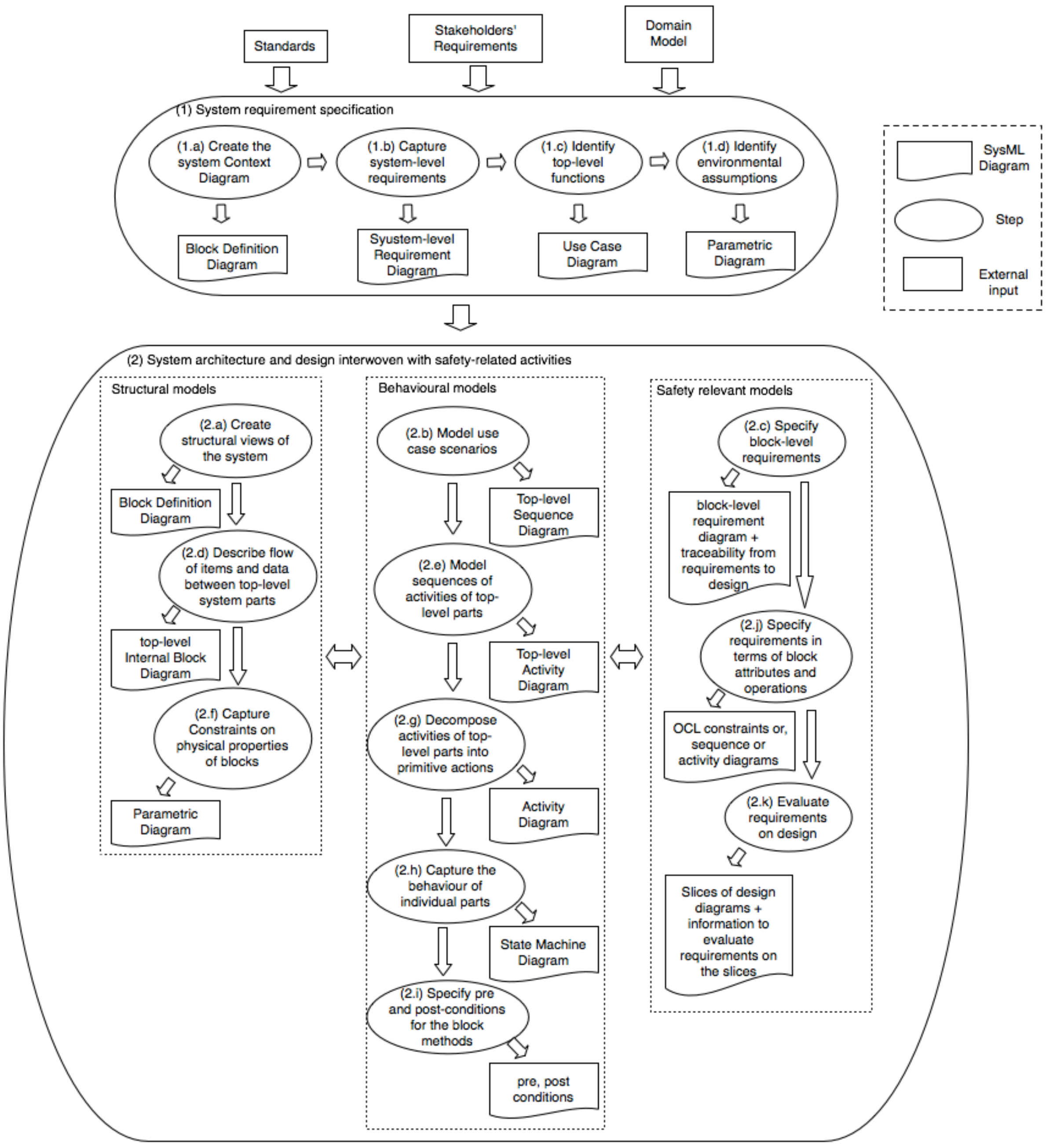

Standards
Stakeholders' Requirements
Domain Model
(1) System requirement specification
(1.a) Create the system Context Diagram
(1.b) Capture system-level requirements
(1.c) Identify top-level functions
(1.d) Identify environmental assumptions
Block Definition Diagram
Syustem-level Requirement Diagram
Use Case Diagram
Parametric Diagram
SysML Diagram
Step
External input
(2) System architecture and design interwoven with safety-related activities
Structural models
(2.a) Create structural views of the system
Block Definition Diagram
(2.d) Describe flow of items and data between top-level system parts
top-level Internal Block Diagram
(2.f) Capture Constraints on physical properties of blocks
Parametric Diagram
Behavioural models
(2.b) Model use case scenarios
Top-level Sequence Diagram
(2.e) Model sequences of activities of top-level parts
Top-level Activity Diagram
(2.g) Decompose activities of top-level parts into primitive actions
Activity Diagram
(2.h) Capture the behaviour of individual parts
State Machine Diagram
(2.i) Specify pre and post-conditions for the block methods
pre, post conditions
Safety relevant models
(2.c) Specify block-level requirements
block-level requirement diagram + traceability from requirements to design
(2.j) Specify requirements in terms of block attributes and operations
OCL constraints or, sequence or activity diagrams
(2.k) Evaluate requirements on design
Slices of design diagrams + information to evaluate requirements on the slices

Figure 10 - Overview of the methodology for modelling control systems to support safety certification.

4.1 System-level requirements specification

To be able to make well-educated decisions in the process of analysing and designing a system, it is essential to know the system and the environment in which it will operate, to know what elements will interact with the system, and to know the restrictions and requirements that the system must satisfy. In this phase, assumptions about the system and the environment it will operate in will be modelled. To ensure safety, the models created in this phase can be reviewed by domain experts to make sure that there are no misunderstandings, as suggested in [2].

4.1.1 Create the system context (Step 1a)

Input: Domain model, stakeholder's requirements.
Output: System context diagram (BDD) showing the system and the relevant environment.

The **system context diagram** gives an overview over the entities in the environment/domain that are relevant and which directly or indirectly will interact with the system of interest. The context of the system is described in a **block definition diagram** using block constructs and the **composition** relation. As mentioned earlier, a **block** construct in SysML is a general modelling concept [8] and can describe any entity, e.g. document, human, other system, software, hardware, logical or physical entities.

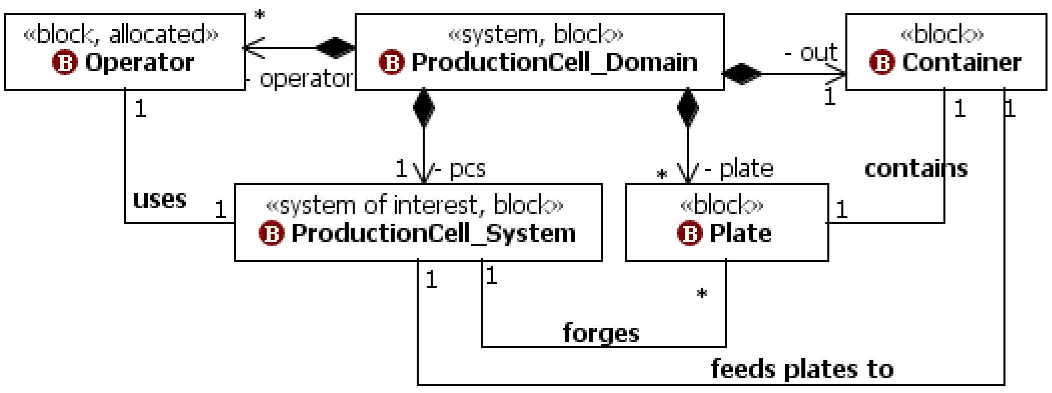


Figure 11 – bdd [Package] ProductionCellStructure [System Context]

The caption in Figure 11 shows the diagram header, which has a specific structure and is standard in SysML [8]. The header should be placed in the top left corner of the diagram frame[4]. The structure of the header is:

diagram type [model element type] model element name [usage]

[4] The tool does not support diagram frames for all diagrams, see chapter 5.2 SysML Compliance.

The header contains a lot of information about the diagram. By looking at the caption of Figure 11, we see that this is a block definition diagram**,** which represents a package named *ProductionCellStructure*, and shows the system context.

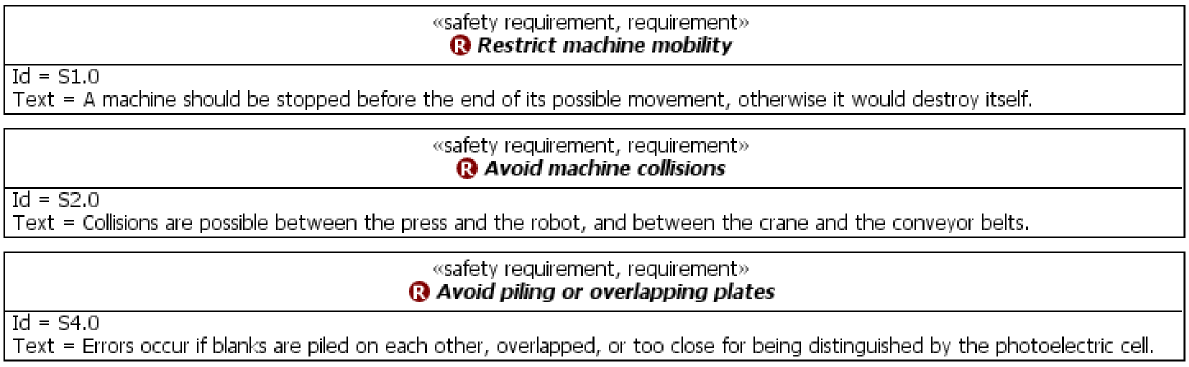
The above diagram shows the relevant entities in the environment of the production cell. To show that the focus is the *ProductionCell_System*, the block has been marked with the keywords <<*system of interest*>>. This system consists of both software and hardware, but this is not defined in our model yet. The main focus, as we will see later in this thesis, is the software part of the system, more specifically the controller of *ProductionCell_System*.
The relevant entities in the *ProductionCell_Domain*, which represents the environment where the *ProductionCell_System* will operate, are *the operators* of the system and *a container* which will contain the *metal plates* that have been forged by the system.
We also add associations between the blocks in the structure diagram, to specify how the elements interact with the system. An alternative is to create an **internal block diagram** to show how the parts are connected. Examples of internal block diagrams are described in chapter 4.2.4.

Assumptions:
The way *Plate* is fed to the system is not described in [5], and is out of the scope for this case study.

4.1.2 Capture the system-level requirements(Step 1.b)

Input: Stakeholders' requirements, standards, domain model
Output: A requirement diagram (REQ) describing the requirements that the system as a whole must conform to (the system-level requirements).

Capturing the system's **requirements** in a model and being able to keep track of their development are some of the new features in SysML, which we introduced in chapter 2.3. This assists us in keeping track of decisions/assumptions that are made about the system, by capturing the relationships between requirements in the model, using the associations described in **Table 1**.


«safety requirement, requirement»
Restrict machine mobility
Id = S1.0
Text = A machine should be stopped before the end of its possible movement, otherwise it would destroy itself.

«safety requirement, requirement»
Avoid machine collisions
Id = S2.0
Text = Collisions are possible between the press and the robot, and between the crane and the conveyor belts.

«safety requirement, requirement»
Avoid piling or overlapping plates
Id = S4.0
Text = Errors occur if blanks are piled on each other, overlapped, or too close for being distinguished by the photoelectric cell.

Figure 12 –Detail view of req [Package] ProdCell_Requirements [System-level requirements]. Whole diagram in the appendix, Figure 40.

First step is to capture the main requirements, which are the result from analysing the stakeholders' needs and wishes for the system, the standards that the system has to satisfy, and the domain model, which all are provided as inputs to the methodology. They are captured in requirement blocks, which should contain an identification property and a text property. By additionally stereotyping the requirements, we specify what kind of requirement each block represents as there can be several types of requirements (see assumption below).
In this case study, we have used the following naming convention for the identification property: All system-level safety requirements are numbered SX.0, where X is a natural number and S stands for safety. All requirements that originate from a system-level requirement will keep the first number in the reference, and the letter will describe the relationship with its original, for instance DX.1, which describes the derived requirement from the original SX.0. As requirements have a textual form, they may often be ambiguous. Further on in the development, we will strive to make the requirements more formal and unambiguous by partitioning, deriving and refining them, and keep the history of the development of the requirements to be able to trace the assumptions made. In chapter 4.2.3 the system-level requirements will be further analysed.

Assumptions:
(1) The requirements described in [5], include safety, liveness, efficiency and flexibility requirements. As this thesis is about safety and certification, we focus only on the safety requirements, but understand that there would be more requirements than these in a fully described system.

4.1.3 Identify the system-level functions (Step 1c)

Input: Domain model, system context diagram (BDD), system-level requirements (REQ).
Output: Use case diagram (UC) describing the system-level functions of the system, and use-case specifications and revisited system-level requirements (REQ).

The next step is to identify the main functions of the system, looking at the system as a black-box. The functions that are identified should fulfil the goals of the user [8], as captured in the system-level requirements. The **use case diagram** in SysML is the same as in UML 2.0, depicting functions of the system as use cases, and users of the system as actors (e.g. humans or external systems)**.**

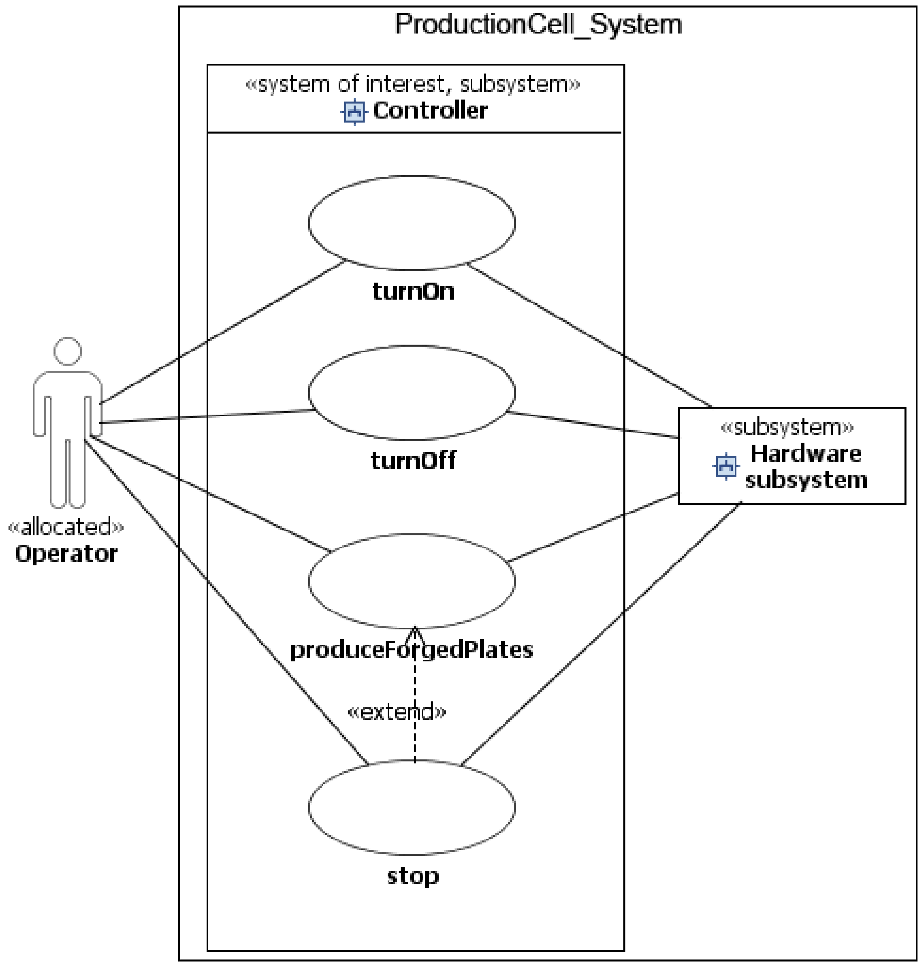


Figure 13- uc [Block] ProductionCell_System [Software main functions][5]**.**

Figure 13 shows the **use case diagram** for the *ProductionCell_System.* However, since the focus of this thesis is the software of the system, we will focus on the functions of the subsystem, *Controller*. The actor, *operator*, represents the human interacting with the system. The *Hardware subsystem* represents the set of hardware devices which interact with the software by providing input and receiving output from the software. The subsystem *Controller* has four use cases, namely *turnOn, turnOff, produceForgedPlates* and *stop* (i.e. emergency stop). These use cases can be further described and detailed textually in use case specifications or visually using activity diagrams or sequence diagrams. Examples of use case specifications for *turn_on* and *produceForgedPlates* are in the appendix, chapters A.1.4 and A.1.5.

[5] The system boundary in this diagram has been added to the image outside the tool as we could not locate the system boundary in the tool (see Chapter 5 Tool problems).

Each use case should be traced to its relevant system-level requirements, which either specifies the assumption of the creation of the use case, or it may put constraints on the allowed behaviour (e.g. in the case of safety requirements). This is modelled using the *refine* relationship as in Figure 42 which depicts the relation the use case *producedForgedPlates* has to the system-level safety requirements.

Assumptions:
(1) The use case *turnOn* is described in [5] as sending the event *TurnOn* to all the devices represented in the controller, so that the device is created and initialized. In this thesis we only describe the initialization process of the devices.
(2) The use cases *turnOff* and *stop* are not described in detail in [5], and are not described further in this thesis.

4.1.4 Identify environmental assumptions (Step 1.d)

Input: Domain model, system context diagram (BDD), system-level requirements (REQ),
Output: Added requirements to the requirement diagram (REQ), environmental constraints depicted in a parametric diagram (PAR).

Environmental assumptions are very important to consider and specify in the requirements, as not taking them into account can have serious consequences [2]. If the requirements can be specified with equations, then we can model them with *Constraint Blocks* and *Parametric diagrams*, as introduced in chapter 2.5. When the assumptions have been made, these can be discussed with domain experts, as suggested in [2], and in this way assure that the assumptions made are correct.
For instance, even though not specified in [5], the plates of the system will have a certain size and the container will only be able to room a certain amount of plates. When the container is full, the system needs to stop conveying plates to the container. This requirement may be modelled as an equation, which is dependent on the size of the container and the size of the plates. This constraint equation can be defined using a constraint block, and be made a property of the *ProductionCell_System*. By creating a parametric diagram, the constraint parameters can be bound to properties of *Plate* and *Container*. To uphold traceability between the elements of the system, the related requirement can be traced to the constraint block. The above example has not been modelled in this case study, but a detailed example of modelling a constraint is presented in chapter 4.2.6.

4.2 System architecture and design interwoven with safety-related activities

This phase contains three concurrent groups of activities, namely designing and developing structure models, behavioural models, and requirement and safety design models.
In the structure models we model the structure and interaction points of the blocks, while we model how the blocks interact in the behavioural model. In addition, we focus on the development of the requirements and the safety design models, where we formalize and trace requirements to related model elements using SysML cross-cutting constructs. On completion of this step we will have a set of diagrams that support generation of safety cases/arguments to be used in a certification process.

4.2.1 Structural views (Step 2a)

Input: System context diagram (BDD), system-level requirement diagram (REQ), domain model (containing the environment and terminology etc).
Output: Structure diagram (BDD), glossary describing the blocks.

In this step we focus on creating a structural definition of the system by decomposing the *system of interest* block from the **system context diagram** into its parts. These parts are defined using block constructs in a **block definition diagram** (as described in chapter 2.1). The input documents to the methodology and the models created in the first phase should support the assumptions made in this step.
The decomposition is an iterative process, and starts with modelling the parts of *ProductionCell_System*, namely the hardware devices and the controller, *ProductionCell_Controller*. The controller represents the software which receives inputs from and sends output to the hardware devices to control the state of the devices. As the software is based on an object-oriented solution [5], the controller consists of blocks that correspond one-to-one to the hardware devices (see assumption).
Figure 14 shows a simplified version of the top-level structure of the production cell. The parts of the controller in this diagram will later be referenced as the *top-level blocks or top-level parts*. In the next iteration of the design of the structure, the *top-level blocks* are further decomposed into its parts. This decomposition of blocks will continue until we reach the leaf blocks, namely the *sensors* and *actuators* which will interact with the relevant hardware parts of the system (see assumption). When this step is finished, the diagram will show the definition of the system structure (see Figure 44). How the parts in this structure are connected will be shown in the internal block diagrams in chapter 4.2.4. An explanation of the blocks in the structure diagram is also provided in the glossary in Appendix B.

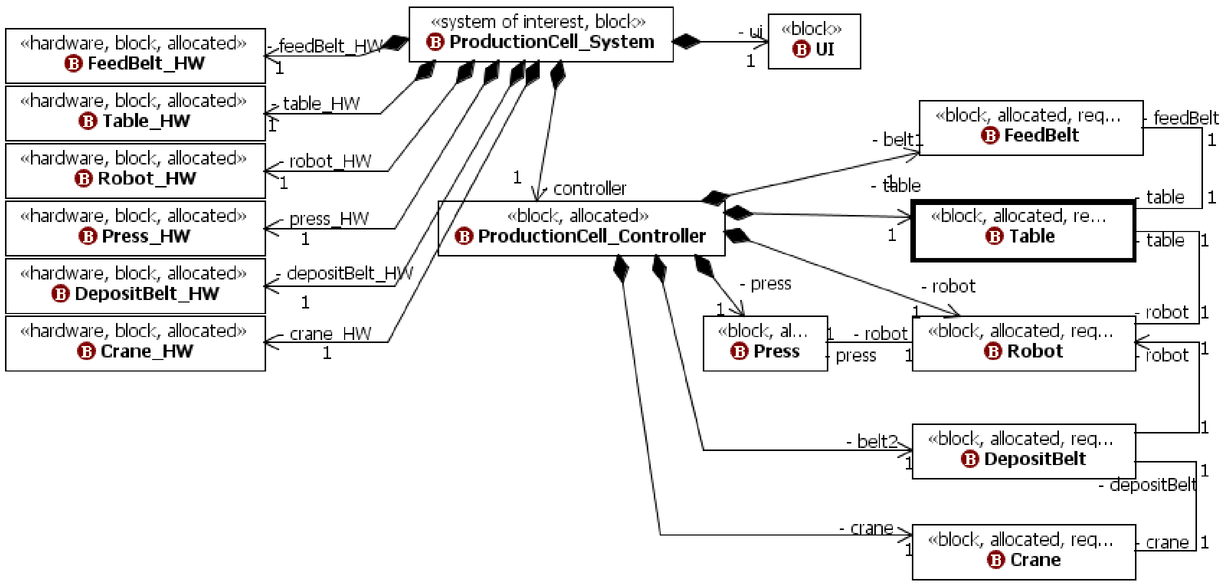

«system of interest, block» ProductionCell_System
«block» UI
- ui
«hardware, block, allocated» FeedBelt_HW
«hardware, block, allocated» Table_HW
«hardware, block, allocated» Robot_HW
«hardware, block, allocated» Press_HW
«hardware, block, allocated» DepositBelt_HW
«hardware, block, allocated» Crane_HW
feedBelt_HW
table_HW
robot_HW
press_HW
depositBelt_HW
crane_HW
controller
«block, allocated» ProductionCell_Controller
«block, allocated, req... FeedBelt
«block, allocated, re... Table
«block, al... Press
«block, allocated, req... Robot
«block, allocated, req... DepositBelt
«block, allocated, req... Crane
- belt1
- table
- press
- robot
- belt2
- crane
feedBelt
table
robot
press
depositBelt

Figure 14 - bdd [Block] ProductionCell_System [simplified top-level structural view]. Whole diagram is in the appendix, Figure 44.

Assumptions:

(1) The actual communication between the hardware and software is scoped out of this thesis. In [5] they model the communication with the hardware using servers to support their simulation. We assume that any signal that is sent between the software and hardware parts are understood by the recipient. In a real life control system there would be drivers translating the information sent between software and hardware.

(2) As the hardware is scoped out of this case study, hardware devices are not further decomposed in this step.

4.2.2 Modelling behaviour from the use case scenarios (Step 2b)

Input: Use case diagram (UC), structure diagram (BDD).

Output: Sequence diagram describing the interaction between the top-level blocks (SD).

In this step the use cases that were defined in the **use case diagram** are refined by creating a **sequence diagram** for each use case and modelling the interactions between the top-level blocks, i.e. the parts of *ProductionCell_Controller*, which we defined in the **structure diagram.** In SysML, the sequence diagram is similar to UML 2.0. Each of the parts of the controller is represented by a lifeline, so that the interaction between them can be modelled. Since the background case study [5] describes the interaction between the classes in the controller as asynchronous, the interactions in Figure 15 are modelled using asynchronous signal messages. (Signals are shown in Figure 53.)

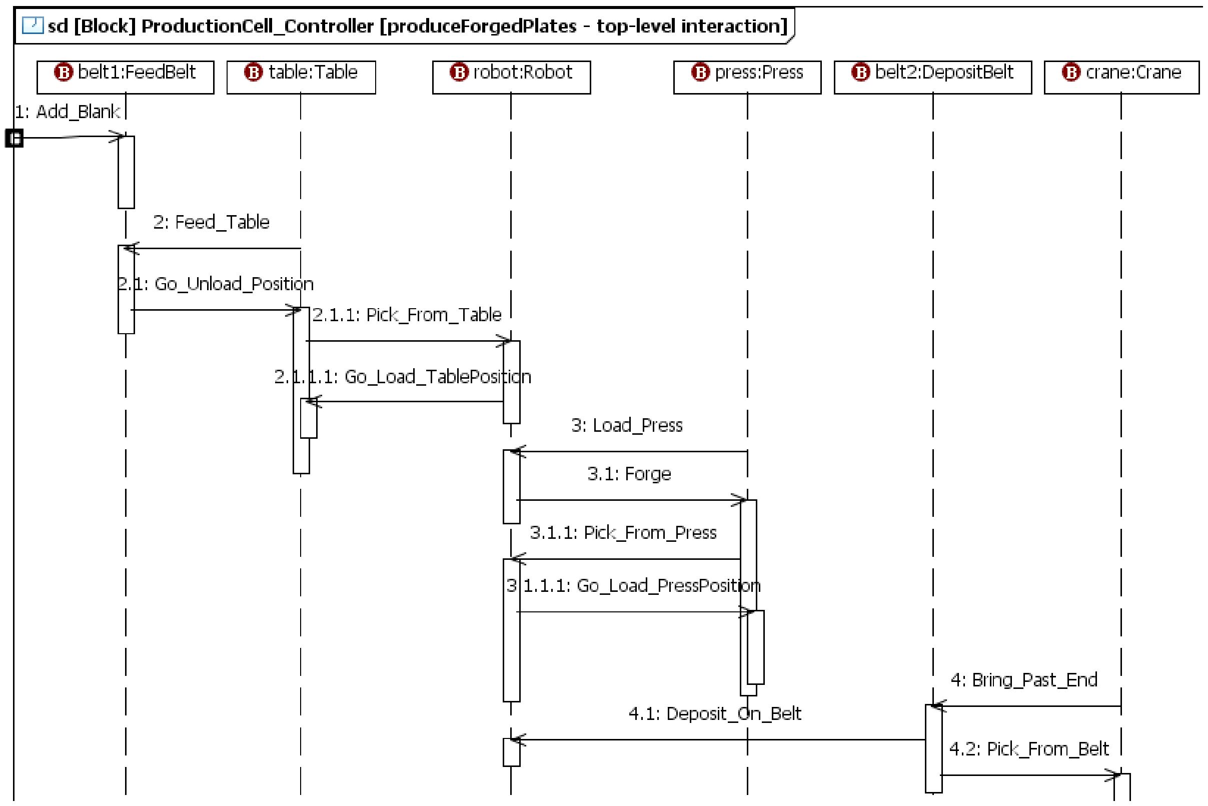


Figure 15 – sd [Block] ProductionCell_Controller [ProduceForgedBlanks]. This diagram only shows interactions between the top-level blocks. Whole diagram is in the appendix, Figure 58.

As we start modelling the behaviour of the system, the **sequence diagram** gives a good overview over the interaction between the top-level parts of the system and is helpful regarding identifying the different send/receive signal events and the related operations for each of the top-level blocks, which will be helpful in the future steps of this methodology.
For instance, when *Press* is ready, it sends the signal *Load_Press* to the *Robot*, which triggers it to start the operation *load_press(),* as is modelled in Figure 58 in the appendix. The operation *load_press()* is identified and can be added as a behavioural feature of the block *Press*. However, even though the sequence diagram is useful for identifying signals and operations, it must be noted that these diagrams are not exact descriptions of the system on this level, as it implies an ordering between all the signal events and operations (see chapter 6.1.1). The devices in the system are concurrent and may receive signal events at any time, even if they are not be ready to act on them, which is not evident from this diagram. Hence, the system may have several alternative orderings. In later steps we will model the system using activity diagrams and state machines and through these diagrams the concurrency and lack of a particular ordering is more obvious.

Assumptions:

(1) Numbers are added by the tool, and are not there by choice, but are added by the tool.
(2) We assume that the *turnOn* use case includes initialization. This use case is not described as a sequence diagram.

Consistency:

(1) Lifelines represent each of the parts of the interaction's owning block which the sequence diagram represents.

(2) Identified operations on a lifeline should be added as operations to the block the part represents.

(3) Receive events should be a part of the block's provided interfaces.

(4) Send events should be part of the block's required interfaces.

4.2.3 Specify block-level requirements. (Step 2c)

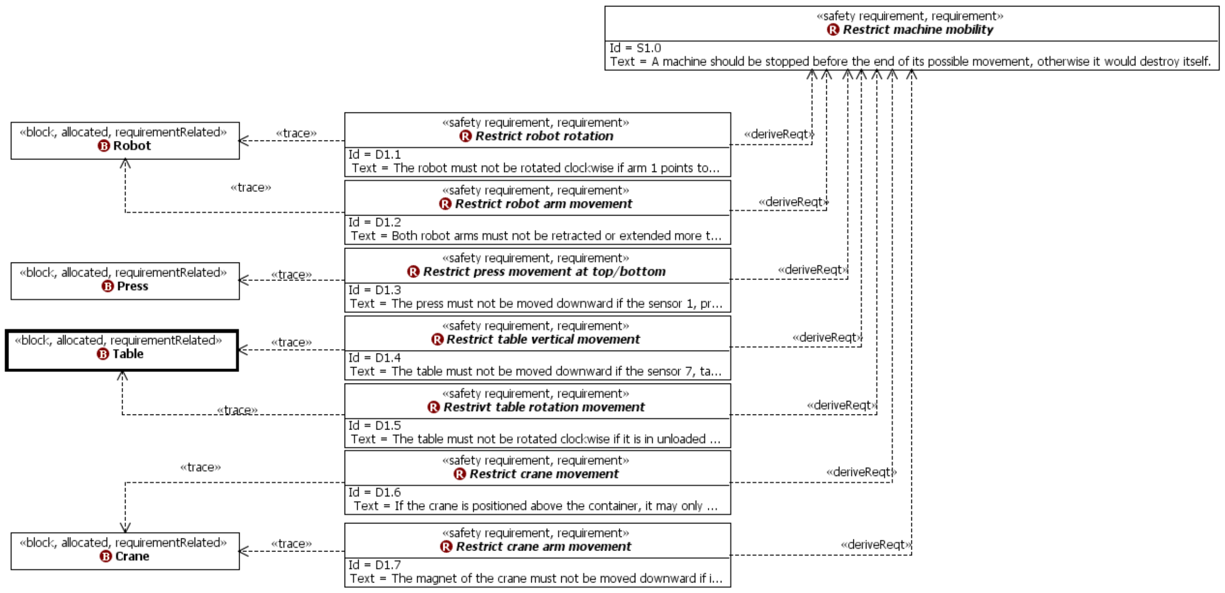
Input: System-level requirements (REQ), structure diagrams (BDD).

Output: Requirement diagram describing the partitioned and derived requirements with focus on the structure blocks (REQ).

In an earlier step of the methodology, the **system-level requirements** were captured by inspecting the information given from stakeholders, standards and the domain model. Each of these requirements might affect one or more of the *top-level parts* of the system, which were captured in the **structure diagram**.

As mentioned in chapter 2.3, textual requirements may be ambiguous, so the requirements will at any time be subject for partitioning, deriving or refining. This way we are able to create requirements that may be easier to fulfil and easier to show that they are satisfied. However, in this step, all the **system-level requirements** will be inspected to see if and how they relate to each of the top-level blocks of the system. This may result in new requirements, called *block-level* requirements, which are derived from the original system-level requirements. To trace these and keep track of the development, we use the *deriveReqt* relationship (see **Table 1**). The relationship originates from the derived requirement to the original system-level requirement. By capturing a rationale of the assumptions made and connecting it to the relationship connector, the requirements and assumptions may be inspected for validity at a later time. In addition, the derived requirement is traced to the related block or set of blocks using the *trace* relationship (see **Table 1**).

Figure 16 shows the system-level safety requirement S1.0 from Figure 12, which concerns restricting machine mobility to avoid destroying machinery. This affects the software, which controls the hardware. So, for each of the top-level software blocks, we inspect whether S1.0 affects it and/or its parts and if so, describe it in a new requirement, relate the derived requirement to the original requirement using the *deriveReqt* relationship and trace the derived requirement to the related top-level block. The requirements derived from S1.0 are all traced to the software blocks of the devices that only have a certain range of movement. In other words, S1.0 is traced to all of the devices, except for the belts, through its derived requirements. Though not in this example, some requirements may be traced to more than one block, if their satisfaction requires a collaboration of several blocks.


«safety requirement, requirement»
Restrict machine mobility
Id = S1.0
Text = A machine should be stopped before the end of its possible movement, otherwise it would destroy itself.
«block, allocated, requirementRelated»
Robot
«trace»
«safety requirement, requirement»
Restrict robot rotation
Id = D1.1
Text = The robot must not be rotated clockwise if arm 1 points to...
«deriveReqt»
«trace»
«safety requirement, requirement»
Restrict robot arm movement
Id = D1.2
Text = Both robot arms must not be retracted or extended more t...
«deriveReqt»
«block, allocated, requirementRelated»
Press
«trace»
«safety requirement, requirement»
Restrict press movement at top/bottom
Id = D1.3
Text = The press must not be moved downward if the sensor 1, pr...
«deriveReqt»
«block, allocated, requirementRelated»
Table
«trace»
«safety requirement, requirement»
Restrict table vertical movement
Id = D1.4
Text = The table must not be moved downward if the sensor 7, ta...
«deriveReqt»
«trace»
«safety requirement, requirement»
Restrivt table rotation movement
Id = D1.5
Text = The table must not be rotated clockwise if it is in unloaded ...
«deriveReqt»
«trace»
«safety requirement, requirement»
Restrict crane movement
Id = D1.6
Text = If the crane is positioned above the container, it may only ...
«deriveReqt»
«block, allocated, requirementRelated»
Crane
«trace»
«safety requirement, requirement»
Restrict crane arm movement
Id = D1.7
Text = The magnet of the crane must not be moved downward if i...
«deriveReqt»
Figure 16 – Part of req [Package] ProdCell_Requirements, showing the derivation of one of the system-level requirements into block-level requirements. The full diagram and text can be viewed in the appendix A.3.1.

Assumptions:
(1) Since the main focus is the software of the system, we are only relating requirements to the software blocks.
(2) Please note that in some of the requirements from [5], the notion of "blank" appears. This refers to a plate which is not forged.

4.2.4 Communication and flow of items between top-level parts (Step 2d)

Input: Structure diagrams (BDD), interaction diagrams (SD), block-level requirements (REQ)
Output: Internal block diagrams (IBD)

In this step we will connect the parts of the system and describe the interaction between them. There are three types of diagrams we wish to specify, which we will show through the following three examples; (1) the item flow through the hardware devices of the system, (2) the communication between the hardware and software, and (3) defining the interaction points between the top-level software blocks and describing their interaction by defining the interfaces. This will result in a set of **internal block diagrams** for the *ProductionCell_System*. Internal block diagrams were introduced in chapter 2.2.

(1) First we will start with the *item flow* of the physical *Plate* through the hardware parts of the system. Even though the focus of the thesis is on the software of the system, it is

important to understand how the hardware parts will interact to be able to have a correct understanding of the system.

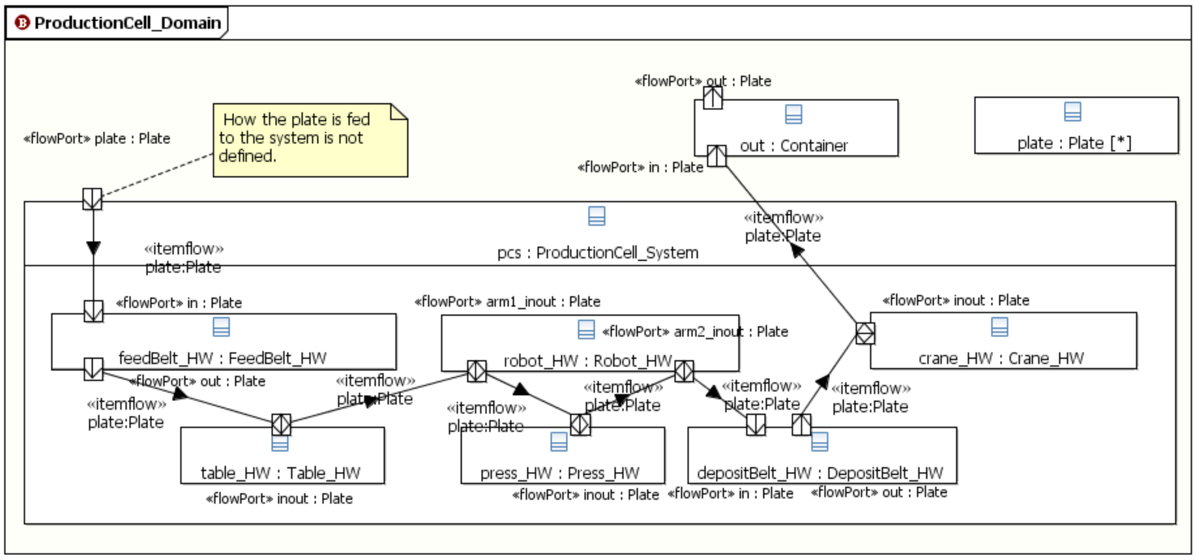


Figure 17 – ibd [Block] ProductionCell_Domain [Item flow HW devices]

In Figure 17 we have specified the item flow of the *Plate* through the hardware parts. The interaction points are specified using *atomic flow ports*, typed by *Plate* to describe that this is the only allowed item that may flow through the hardware devices. The port also describes the direction an item may flow, more specifically using the directions *in*, *out* or *inout*. This is visually described in the diagram with arrows on the ports. The choice of port design shows how the plate flows through each device. For instance, the feed belt receives the plate in one end, and sends it at the other end. In Figure 17, *FeedBelt_HW* has one flow port with direction *in* and another with direction *out* to show that the *Plate* flows through the feed belt. The robot, however, picks a plate with one of the arms and it stays there until it is dropped at the receiving device. Consequently, we represent this with a flow port with the direction *inout* to show that the plate flows in and out of *Robot_HW* at the same spot. The actual route and direction for the flow of a *Plate* is described using the item flow connector between the flow ports.

Assumptions:

(1) Even though the diagram in Figure 17 is created for the block ProductionCell_Domain[6], our focus is on the ProductionCell_System.

(2) How the plates are introduced to the system is not defined. We assume that a plate arrives at the system as soon as the operator tells the system to add_blank.

[6] The purpose of this internal block diagram (ibd) in Figure 17 is to show the interaction in ProductionCell_System. See Chapter 5 regarding the tool problem with boundary ports.

Consistency:

The parts of *ProductionCell_System* shown in the internal block diagram correspond with the parts of the *ProductionCell_System* which were defined in the structure diagram. The names of the parts are consistent with the role names of the composition association.

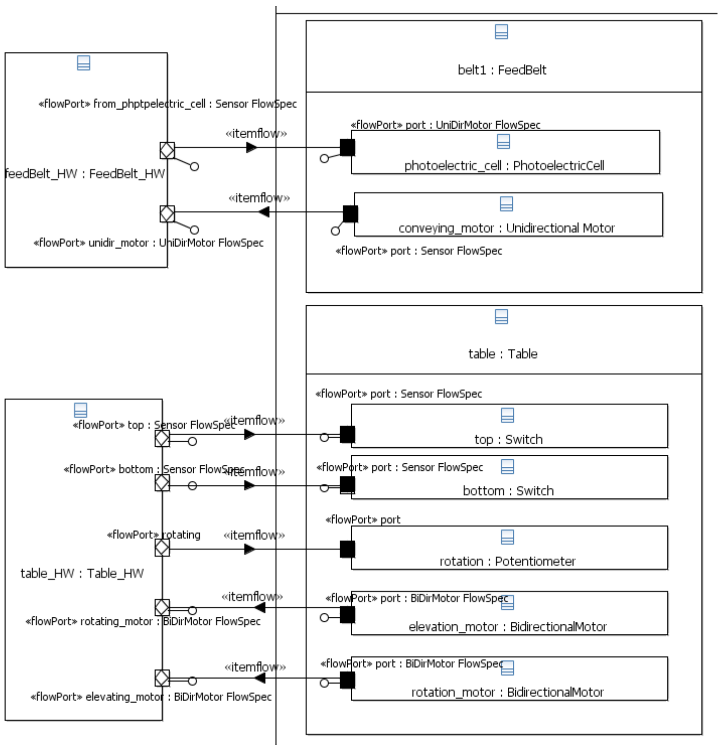

belt1 : FeedBelt
«flowPort» from_phptpelectric_cell : Sensor FlowSpec
«itemflow»
«flowPort» port : UniDirMotor FlowSpec
photoelectric_cell : PhotoelectricCell
feedBelt_HW : FeedBelt_HW
«itemflow»
conveying_motor : Unidirectional Motor
«flowPort» unidir_motor : UniDirMotor FlowSpec
«flowPort» port : Sensor FlowSpec
table : Table
«flowPort» port : Sensor FlowSpec
«flowPort» top : Sensor FlowSpec
«itemflow»
top : Switch
«flowPort» bottom : Sensor FlowSpec
«itemflow»
«flowPort» port : Sensor FlowSpec
bottom : Switch
«flowPort» port
«flowPort» rotating
«itemflow»
rotation : Potentiometer
table_HW : Table_HW
«itemflow»
«flowPort» port : BiDirMotor FlowSpec
«flowPort» rotating_motor : BiDirMotor FlowSpec
elevation_motor : BidirectionalMotor
«itemflow»
«flowPort» port : BiDirMotor FlowSpec
rotation_motor : BidirectionalMotor
«flowPort» elevating_motor : BiDirMotor FlowSpec

Figure 18 – Detail view of the ibd [Block] ProductionCell_Domain [HW/SW Communication]. Full diagram in the appendix, Figure 49.

(2) The **internal block diagram** in Figure 18 shows the interaction points between hardware and software. This is diagram however strongly simplified as the details of the actual communication with the hardware are scoped out of this case study. We include this diagram only to show how the flow of communication is assumed to be (see assumption). Each of the flow ports are typed with a *flow specification* containing the signals we assume would be sent to the controller. The conjugated port which is marked black, is also typed with the flow specification, but the directions of signals or items is opposite from what is specified in the flow specification.

Assumptions:

(1) We assume that the software and hardware understands the signals sent between them. In a real life system, these signals would be information that would need to be interpreted and converted into a signal that can be understood by controller.

(2) Power flow and other mechanical/electrical flows are out of the scope for this thesis.

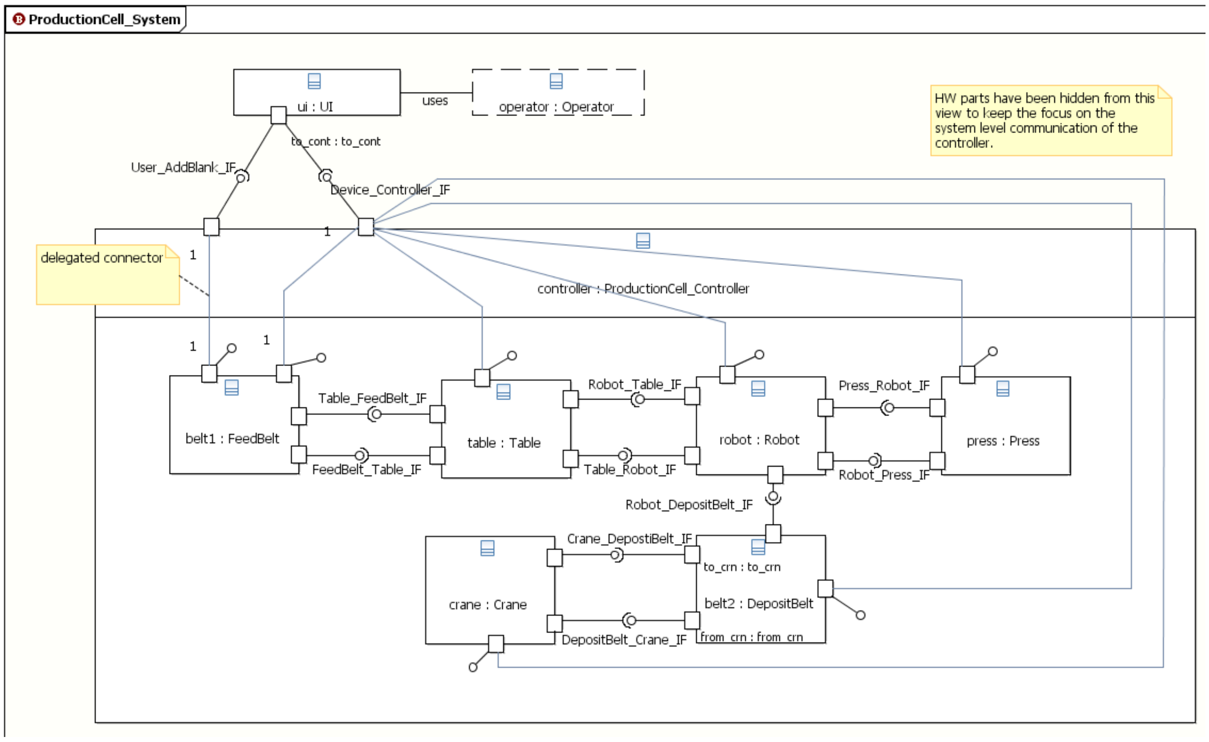


Figure 19 - ibd [Block] ProductionCell_System [Communication between top-level blocks]

(3) The interaction between the software parts of the system were first identified in the **sequence diagram** in a previous step. This information is very useful to be able to specify interaction points and define interfaces for the top-level parts of the *ProductionCell_Controller* (see consistency comment below). The **internal block diagram** in Figure 19 shows the interaction points and communication between the parts of *ProductionCell_Controller*, in addition to showing how the operator communicates with the system.

The interaction points are described using *standard ports*, which are similar to ports in UML 2.0. Each of the ports in Figure 19 either provides or requires the interfaces which are defined in Figure 52. They are based on the information about the send/receive events from the sequence diagram (see consistency comment below). The SysML specification [7] states: “*a block must be able to react to all signals that are specified in its behavioural port's provided interfaces*”.

Even though not specified in [5], we have added a user interface block, *UI*, to show the possible interaction an operator would have with the system. The boundary ports on the block *ProductionCell_System* are delegation ports which delegates received signals to relevant parts.

Assumptions:

We have assumed that the operator uses an user interface to communicate with the system, even though not defined in [5].

Consistency:
(1) All send events on a lifeline in the **sequence diagram**, which represents a part of the system, should be represented in one of the part's required interfaces.
(2) All receive events on a lifeline in the **sequence diagram,** which represents a part of the system, should be part of one of the part's provided interfaces.
(3) All parts in the **internal block diagram** for a block should be consistent with the parts defined for that block in the **structure diagram**.

4.2.5 Identify top-level part activities and their sequences. (Step 2e)

Input: Structure diagram (BDD), sequence diagram (SD)
Output: Block diagram (BDD) describing activity composition, activity diagram (ACT) describing the ordering of the activities of the top-level parts.

After completing this step we will have an **activity diagram** with an overview over the sequential and concurrent behaviour of, and the asynchronous interaction between the top-level parts of the *ProductionCell_Controller*. In addition we will have a **block definition diagram** describing the composition of the activities of the system. Modelling at this level allows us to keep an overview over the system as a whole and the communication between the parts of the system throughout the duration of the activity.
The main functionality in this case study, *produceForgedPlates*, which was captured in the **use case diagram** and refined in the **sequence diagram**, is the main activity in the example below. However, in this example, as shown in Figure 60, we have also included the *initialize* activities of each of the parts, and are therefore able to see the whole life span of the system between the receptions of the *TurnOn* and *TurnOff* signals.

As described in chapter 2.4, an activity is composed of other activities or actions. While creating the **sequence diagram** in Figure 58, the operations of the top-level blocks were identified based on the interaction between the top-level parts of the system. These operations can now be used to identify the activities which the main activity is composed of, and which parts are responsible of executing them. The composition of the activities can be defined in a **block definition diagram**, as shown in Figure 20.

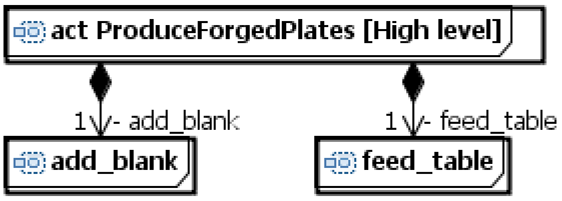
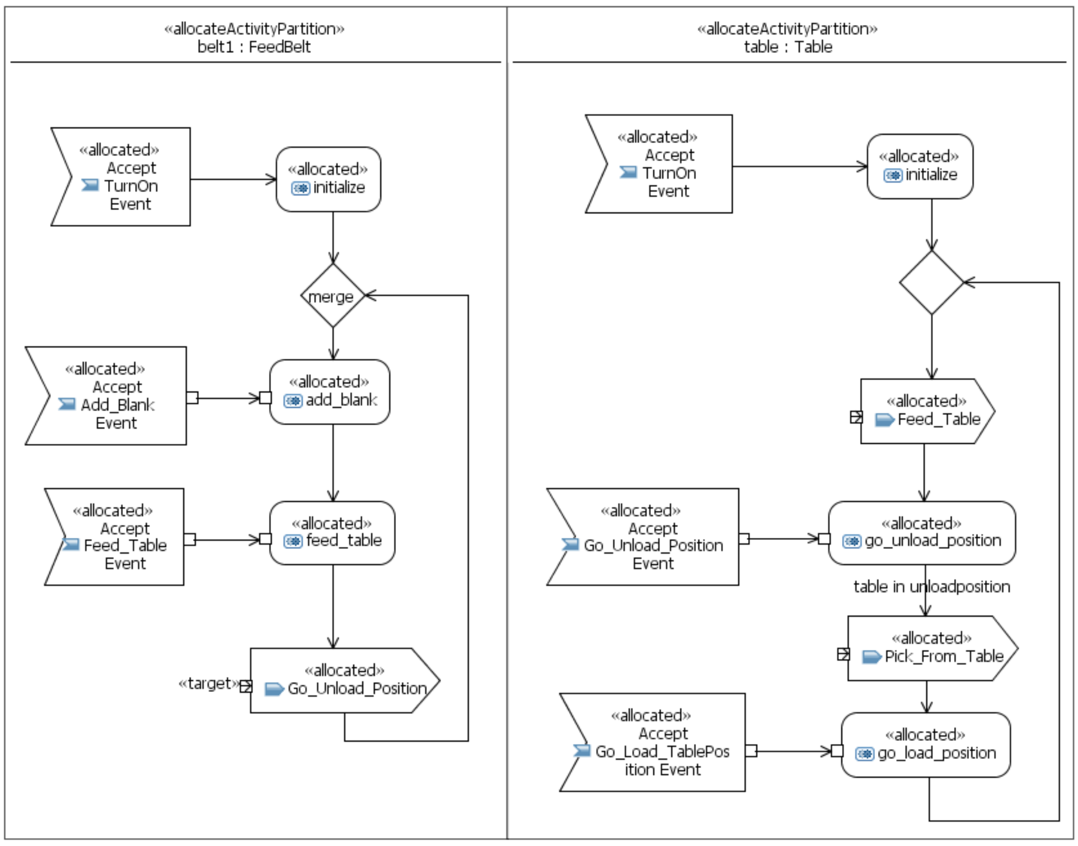


Figure 20 – Detail view of bdd [Package] Activity [Composition of produceForgedBlanks] showing two of the contained activities. Whole diagram in the appendix, Figure 59.

There are also other approaches for identification of activities. In [8], they suggest identifying the main functionality for each block involved, or identifying activities using the effects/activities of state machines in addition to the above approach.

To reference the contained activity in the diagram, we use a *CallBehaviourAction*[7] which has an association to the activity we wish to invoke, as described in chapter 2.4. One of the advantages, in addition of giving us possibilities to reuse the activity, is that we do not have to specify its details yet. To show the asynchronous interaction between the parts of the system, we represent the send and receive events, which were identified in the **sequence diagram**, as *sendSignalActions* and *acceptEventActions*. Each of the actions is then allocated to the part of the system responsible of executing it, using *AllocateActivityPartitions*, which resembles swim lanes in UML 2.0. This way we add traceability from each activity to the top-level part of the system that is responsible of its execution.



«allocateActivityPartition» belt1 : FeedBelt
«allocateActivityPartition» table : Table
«allocated» Accept TurnOn Event
«allocated» initialize
merge
«allocated» Accept Add_Blank Event
«allocated» add_blank
«allocated» Accept Feed_Table Event
«allocated» feed_table
«target»
«allocated» Go_Unload_Position
«allocated» Accept TurnOn Event
«allocated» initialize
«allocated» Feed_Table
«allocated» Accept Go_Unload_Position Event
«allocated» go_unload_position
table in unloadposition
«allocated» Pick_From_Table
«allocated» Accept Go_Load_TablePosition Event
«allocated» go_load_position

Figure 21 – Detail view of the activity diagram, act [Activity] produceForgedBlanks, showing the activity partitions for the FeedBelt and Table. Full version is in appendix, Figure 60.

Once the activities and actions are defined and allocated, we start to specify the order in which the actions in the diagram can execute, using control flows, as shown in Figure 21, where we see the *FeedBelt* and *Table* activity partitions. Defining the order in which the activities can be executed, adds constraints to the behaviour of the system. An activity will

[7] A CallBehaviourAction may also represent other types of SysML behaviour.

not execute until it has tokens on all its required inputs. The sequence diagram can be helpful for us to identify the internal order of the activities of a part, by looking at the order of the operation calls and send events in each of the local execution occurrences.
As soon as the owning activity starts executing, all the *acceptEventActions* in Figure 21 are ready to accept signal events at any time. By connecting these actions to their related activities using object flows, we add an extra constraint to the execution of the activity, by specifying that it cannot execute until it has received the signal, e.g. the connection between the actions *'Accept Feed_Table Event'* and *'feed_table'* in Figure 21. However, as the signal event may arrive at any time, it may also arrive before the control token, and in that case the received signal is stored in the input pin of the activity until the activity is ready to consume it.
In contrast to the sequence diagram, the particular ordering in the activity diagram is defined within each partition (i.e. part of the system), and so, the asynchronous behaviour of the system is maintained. By following the sending and accepting of events between the partitions, we see the sequential behaviour through the system, and by following the control flow in the different activity partitions, we see the concurrent behaviour of the system.

In Figure 60, when the controller receives the *TurnOn* signal, it broadcasts the same signal to all the top-level parts. As soon as the different parts accept the signal events, the control flows start in the different partitions of the diagram. When the controller receives the *turnOff* signal, the *produceForgedPlates* activity and its contained activities terminates, regardless of whether any of the contained activities are currently executing or not (see assumptions).

Assumptions:
(1) In [5] they describe that some signals are sent at the end of the *initialize* operation of some of the classes. We have chosen to model these signals outside the initialize activity.
(2) We assume that there exists an activity after *act produceForgedPlates* that will take care of bringing the different devices and plates to safe positions.

Consistency
(1) Operations in top-level blocks should be consistent with the activities in the partition that represents that block. (i.e. the activity which the *callbehaviouraction* is associated with)
(2) *SendSignalActions* in a partition should be part of a signal reception described in the owning block/part's required interface and be a send event in the sequence diagram on the lifeline of the related part.
(3) *AcceptEventActions* in a partition should be part of a signal reception in the owning block/part's provided interface and be a receive event in the sequence diagram on the lifeline of the related part.

4.2.6 Constraints on physical properties of blocks (Step 2f)

Input: Requirements for blocks (REQ), low-level structure diagram (BDD)
Output: Constraint definition diagram (BDD), Parametric diagram (PAR) with physical constraints of the system.

So far, the constraints we have added to the system have been related to environmental assumptions and behaviour. After inspecting and deriving requirements to block-level requirements, we may find that some physical properties of the hardware (or other) blocks also needs to be constrained. In this example, we will look at the requirement D2.3, "*The magnet of the crane is not allowed to knock against the deposit belt or the container laterally*". According to the description of the crane, as described in chapter 3.1.1.6, the horizontal and vertical movement of the crane will happen simultaneously. We need to make sure that the crane is in a safe position vertically (i.e. the crane arm being at the container extension) before it reaches a position where it could possibly hit the container laterally (see Figure 23). We could have solved this by designing the behaviour so that the crane retracts its arm to the safe position before it moves towards the container, but since we also have to satisfy a performance requirement, and wish to keep the design as true to [5] as possible, we will satisfy the requirement by adding a constraint to the hardware blocks representing the crane and its parts and assume that this constraint will be fulfilled in the hardware design. Since there are no sensors at the last safe horizontal position, we need to constrain relationship between the speeds of the motors in *Crane_HW*,

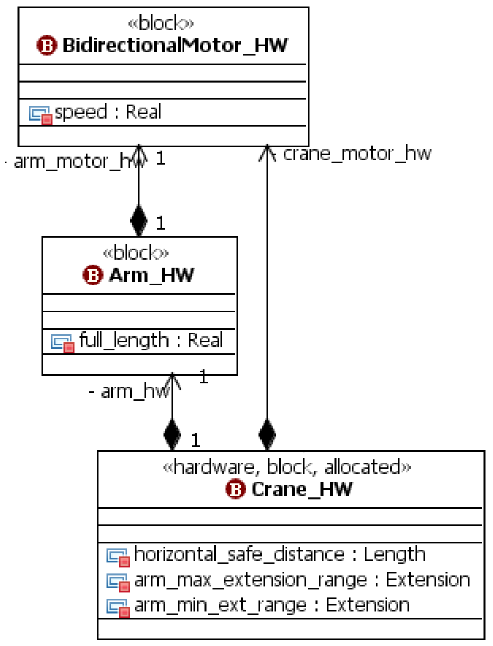

«block»
BidirectionalMotor_HW
speed : Real
- arm_motor_hw
1
crane_motor_hw
1
«block»
Arm_HW
full_length : Real
- arm_hw
1
1
«hardware, block, allocated»
Crane_HW
horizontal_safe_distance : Length
arm_max_extension_range : Extension
arm_min_ext_range : Extension

Figure 22- Part of the block definition diagram that concerns the Crane_HW.

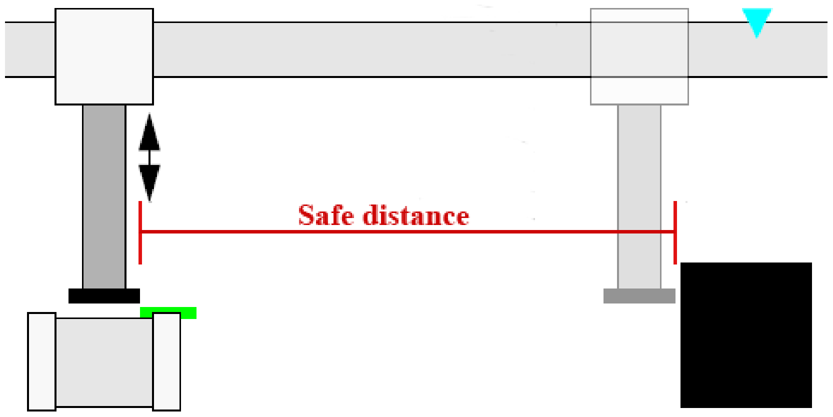


Figure 23 – Graphical description of the constraint parameter, *safe_distance*.

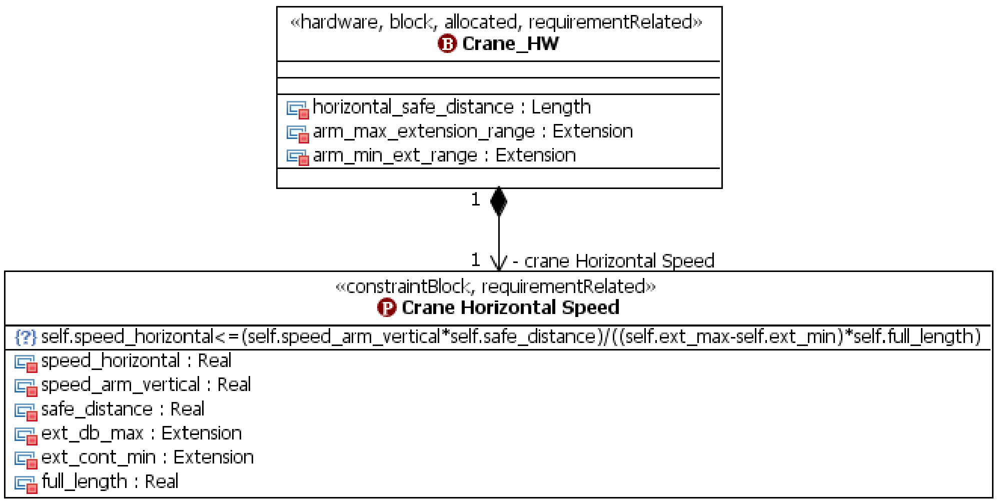

«hardware, block, allocated, requirementRelated»
Crane_HW
horizontal_safe_distance : Length
arm_max_extension_range : Extension
arm_min_ext_range : Extension
1
1
- crane Horizontal Speed
«constraintBlock, requirementRelated»
Crane Horizontal Speed
{?} self.speed_horizontal<=(self.speed_arm_vertical*self.safe_distance)/((self.ext_max-self.ext_min)*self.full_length)
speed_horizontal : Real
speed_arm_vertical : Real
safe_distance : Real
ext_db_max : Extension
ext_cont_min : Extension
full_length : Real

Figure 24 - bdd [Package] Analysis [Crane_HW Constraint]

We have added parts to the block of *Crane_HW*, as shown in Figure 22, to be able to show this example. As generally described in chapter 2.5, and depicted in Figure 24, we create the constraint block *"Crane Horizontal Speed"*, which constrains the relationship between the speeds of the two motors involved. This way we want to ensure that the time needed to move the arm to the retracted vertical position should be less than or the same as the time it takes to move to the last possible safe horizontal position.

To be able to fulfil the requirement, we need to define some parameters in the constraint block that will be bound to properties in the hardware blocks, as shown in Figure 25:

- ***safe_distance**: describes the distance the extended arm can move from the deposit belt towards the container without colliding with the side of the container (see Figure 23).*
- ***speed_horizontal, speed_arm_vertical:** the speed of the motors of the crane and the crane arm, respectively.*
- ***ext_db_max, ext_cont_min:** the arm extensions at deposit belt and container, respectively. (Range: 0-1).*

- ***full_length:*** *the full length of the arm, which we need to be able to calculate the full moving distance.*

The equation below is based on the parameters above and well-known laws of physics[8]:

self.speed_horizontal<=(self.speed_arm_vertical*self.safe_distance)/
((self.ext_max-self.ext_min)*self.full_length)

Since the constraint block, *Crane Horizontal Speed*, is defined to be a constraint property of *Crane_HW* in Figure 24, we can create **the parametric diagram** for *Crane_HW*. In the parametric diagram in Figure 25 we bind the parameters of the constraint property to the block properties[9] and hereby constrain them with the equation defined in the constraint block. The binding is done with a binding connector that ensures equality at both ends of the association.

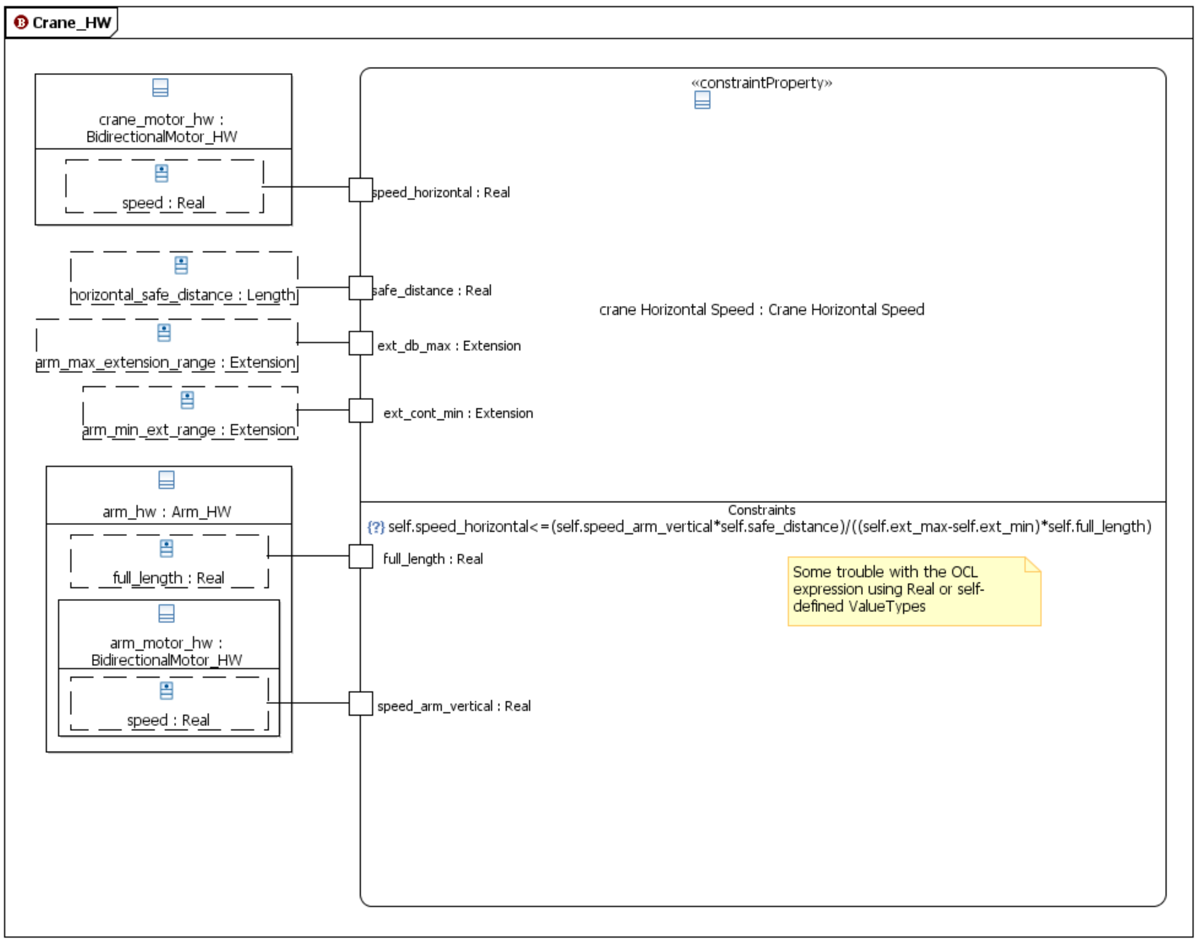

Crane_HW
crane_motor_hw : BidirectionalMotor_HW
speed : Real
horizontal_safe_distance : Length
arm_max_extension_range : Extension
arm_min_ext_range : Extension
arm_hw : Arm_HW
full_length : Real
arm_motor_hw : BidirectionalMotor_HW
speed : Real
«constraintProperty»
speed_horizontal : Real
safe_distance : Real
ext_db_max : Extension
ext_cont_min : Extension
crane Horizontal Speed : Crane Horizontal Speed
Constraints
{?} self.speed_horizontal<=(self.speed_arm_vertical*self.safe_distance)/((self.ext_max-self.ext_min)*self.full_length)
full_length : Real
Some trouble with the OCL expression using Real or self-defined ValueTypes
speed_arm_vertical : Real

Figure 25 – Parametric diagram par [Block] Crane_HW [Horizontal Speed]

[8] Using the physics law where distance = speed * time to come to this equation.
[9] Parametric diagrams can also be created for constraint blocks which are composed by other constraint blocks.

Assumptions:
(1) Requirement D2.3 is also applicable regarding lateral collisions when moving towards the deposit belt. However, we have not considered this as it will be fulfilled if the requirement where arm should not be extended more than necessary is fulfilled.
(2) The tool did not allow creating parametric diagrams for composite constraints (see chapter 5.2), so the equation is more complex then intended.
(3) We assume that by setting constraints on the hardware properties, these will be taken care of in the hardware design.

4.2.7 Decompose activities of top-level parts into primitive actions (Step 2g)

Input: Top-level activity diagram (ACT), structure diagram (BDD)
Output: Block diagram describing the composition of activities (BDD) and activity diagrams describing the lower-level activities/actions (ACT).

In chapter 4.2.5 we described the activities for each of the top-level parts of the system. In this step of the methodology, these activities will be decomposed into the activities of the lower-level parts. This is an iterative process that ends when we reach the actions of the leaf blocks, which in our case study are actions of the type *sendSignalActions* or *acceptEventActions,* which we will use to send signals to the actuators or receive signals from the sensors.
To identify the lower-level activities/actions of a composed activity, we must investigate the parts of the block responsible for the composite activity, and find the parts that have to be involved to fulfil the activity's purpose**.** The methods of identifying the activities/actions can be the same as in 4.2.5. Some activities may represent operations with parameters. In that case, we create *activity parameter nodes* to reflect the parameters, as illustrated in Figure 26 (see chapter 5.3 reg. tool problems).

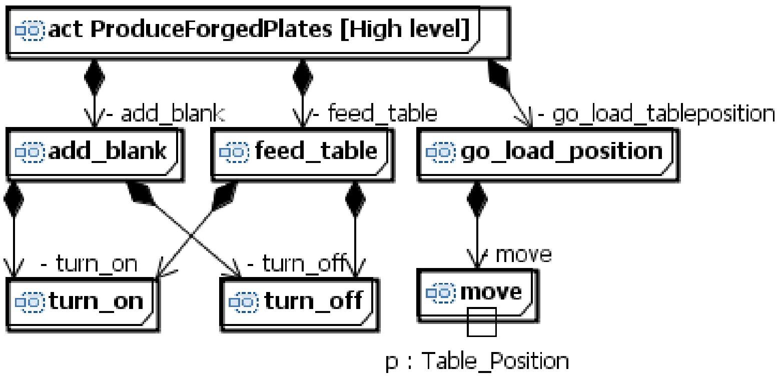


Figure 26 – Detail view of bdd [Package] Activity [Composition of activity].

Looking at the example from chapter 4.2.5 and the description of the system from [5], we see that the activities of the top-level blocks of *ProductionCell_Controller* are all composed of lower-level activities. The decomposition of some of these activities is shown in Figure 26, where we show some of the lower-level activities for the feed belt and the table. After the activities/actions have been identified, we can allocate them to the parts of the top-level blocks and order them in sequence of execution as we did in chapter 4.2.5.

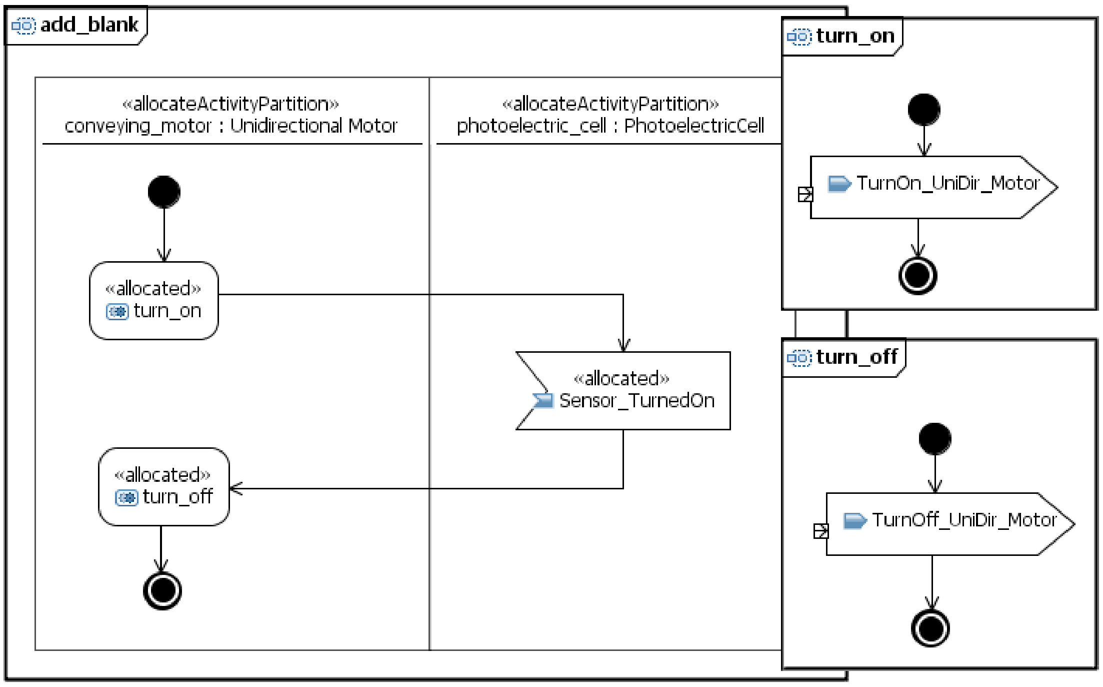

add_blank
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
«allocated»
Sensor_TurnedOn
«allocated»
turn_off
turn_on
TurnOn_UniDir_Motor
turn_off
TurnOff_UniDir_Motor

Figure 27 – The activity diagrams, act add_blank, describing the operation add_blank() for the block FeedBelt from Figure 21. Also showing the lower level activity diagrams for turn_on and turn_off.

Looking closer at the activity *add_blank*, in relation to the parts of the block *FeedBelt*, we see that it is composed of two activities, *turn_on* and *turn_off*. These are both activities of the block *UniDirectionalMotor*. In Figure 27 we allocate and put these activities in order. For the system to know when the plate is in the right position to call the *turn_off* activity, we add an *acceptEventAction* to illustrate that we have to wait for the signal *Sensor_TurnedOn* from the *photoelectric_cell*, i.e from the sensor at the end of the feed belt. We then decompose the activities *turn_on* and *turn_off* and use *sendSignalActions* to represent the sending of the signals *TurnOn_UniDir_Motor* and *TurnOff_UniDir_Motor*. We assume that these signals are sent to and understood by the hardware.
Now we have fully decomposed the activity *add_blank*. To summarize it, we can “read” the activity by iterating bottom first through the set of activity diagrams. When *FeedBelt* receives the signal *Add_Blank*, the activity *add_blank* starts. First it executes *turn_on*, which sends a signal to the *UniDirectionalMotor*, then it waits for a signal from the sensor before it executes *turn_off* which sends a signal to the same motor.

Consistency:
The consistency is mostly the same as in chapter 4.2.5, only on a lower level.

(1) Operations in the lower-level blocks should be consistent with the activities in partitions that represent that block. (i.e. the activity which the *callBehaviourAction* is associated with)
(2) Signals that are sent or received using *sendSignalActions* or *acceptEventActions* should be part of the flow specification/interface of the related port of the block those actions are allocated to.
(3) The blocks which the partitions of an owning activity represent, should be parts of the block that is responsible for executing the owning activity.

4.2.8 Capture the behaviour of individual parts. (Step 2h)

Input: HW/SW communication (IBD), top-level internal structure (IBD), lower-level structure diagrams (BDD), activity diagram (ACT)
Output: State machine (SM) for all behavioural blocks.

Using the **structure** and **activity diagrams** we created earlier, we will in this step identify the blocks with behaviour and capture that behaviour by creating state machines for each of them. The state machine diagram in SysML is similar as the one in UML 2.0.

For every software block that has been allocated from an activity, we create **a state machine diagram**. Following the methodology, we will create simple state machines for each of the leaf software blocks and reuse these in the orthogonal state machines we create for each of the composite software blocks. This will be described in detail below.

In reactive systems, the sensor and actuator hardware blocks communicate with the software by feeding it with input or receiving output, respectively. The leaf software blocks, i.e. sensor and actuator blocks, show the state of the related hardware blocks through their properties. Since each of the leaf software parts of the same type (block) have the same behaviour in the system, we can create one state machine for each leaf block and reuse it in all the orthogonal state machines of the composite blocks where that block is a part. The state machines for the composite blocks will then behave regarding to the combination of the states of all their parts.

The structure we use to create orthogonal state machines for the composite blocks (i.e. all blocks which are not leaf blocks) is depicted in Figure 28. Each layer represents a region or a set of regions in an orthogonal state machine. For each part the block is composed of, we create a region which will contain the state machine representing that part. In addition, we also create one more region which will contain the state machine for the composed block, in which transitions will occur based on the combined state of all its parts represented by the properties set in the other concurrent regions.

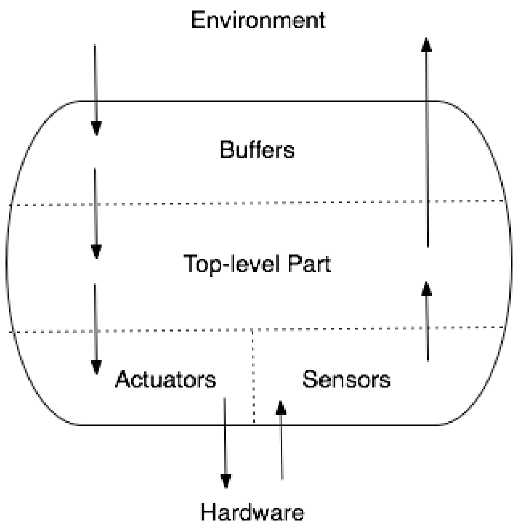


Figure 28 – Orthogonal model explaining the layer of regions for the state machines of the top-level blocks.

As the system uses asynchronous communication, we have to take into account that the target block may not be ready to respond to an incoming signal. In the activity diagram, this was solved using the available notation to buffer the signal at the input pin, waiting for the control token to arrive to so that the activity could execute. In the orthogonal state machine, however, we create buffers by adding, for every expected incoming signal, one concurrent region containing a state machine, so that the signal can be consumed whenever it is received and put the state machine into a state of *"Signal X received"*. As soon as the block is in the right state to process the signal, the block method corresponding to that signal is invoked in the buffer. If there are several buffers in the state machine, we must make sure that maximum one of the buffers are allowed to make a transition at any time. In other words, the guards of the transition in the buffer have to be disjoint (also see chapter 6.1.1). An alternative solution would be to defer the incoming signal events in all the states where they should not be consumed. However, in [10], they discuss that deferrable events should be used with care and may be modelled more directly using concurrent states, as the methodology suggests.

We start by creating state machines for each of the leaf blocks from **the structure diagram** with behaviour, and then we create state machines for each of the higher-level, composite blocks with behaviour, until we reach the root block, *ProductionCell_Controller*. Below we show examples of a sensor state machine, an actuator state machine, a composite block state machine and finally the root block state machine.

Leaf blocks

As we can see in the **internal block diagram** for hardware/software communication[10], a sensor software block will receive signals from the related hardware sensor. We must ensure

[10] Since the hardware is not decomposed into parts, we only see which of the top-level hardware blocks the signal originates from.

that the state machine for that software block has triggers to match those signals. In Figure 29 we show that the *Switch* state machine will make a transition to another state when receiving one of the signals, *Sensor_TurnedOn* or *Sensor_TurnedOff*. This means that the hardware sensor has moved to another state, which will then be reflected in the properties of the software. Notice that in addition to the initial state, there are also two different entry points[11]. This gives the opportunity for two different sensor parts in the system to start in different states, e.g. the two switches in *Table* (see Figure 32).

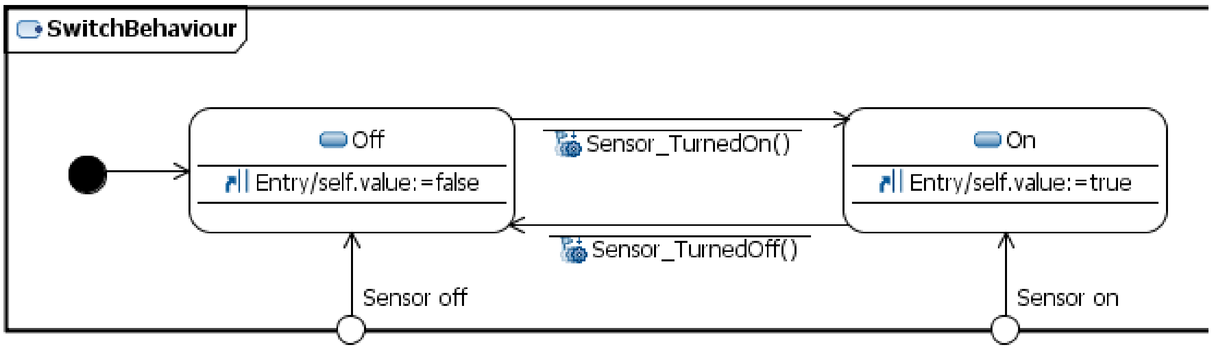


Figure 29 - stm [Block] Switch [Behaviour]

As we can see from one of the **activity diagrams** which contains activities that are allocated to an actuator, (e.g. Figure 27), an actuator software block may receive a call event. In the **activity diagram** this is modelled using *callBehaviourActions*. Inside the related activity, we see it communicates with the hardware by sending a signal to the related hardware actuator. To keep this consistent in the state machine, we must therefore ensure that the state machine has triggers that can respond to the call events and that signals are sent to the related hardware. We partly show this in Figure 30.

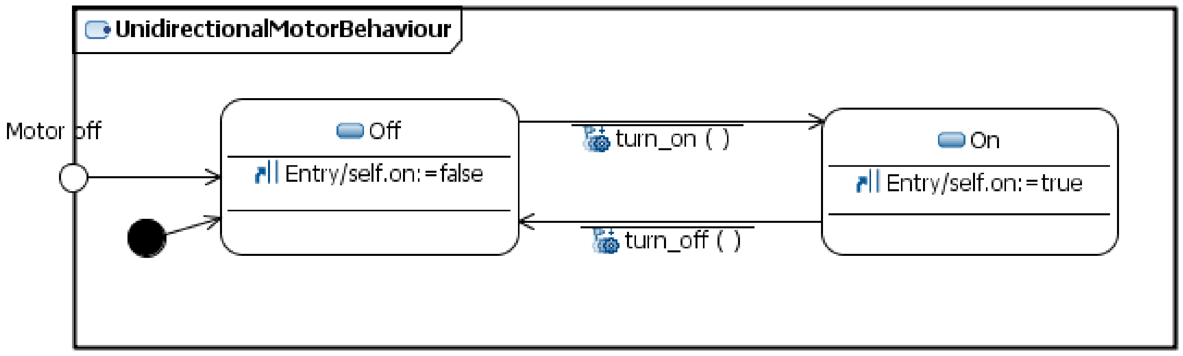


Figure 30 - stm [Block] UniDirectionalMotor [Behaviour]

[11] There is no need for both entry points, as the initial state will lead to same state as the "Sensor off" entry point. We have still added both entry points to specify the difference in the composite diagram (see Figure 32).

However, as we have not modelled the details for the target hardware, we do not show the sending of the signals specifically in the state machines for the actuator software blocks. We assume that by calling "*turn_on*" or other related call events, the related method ensures that the correct signal is sent to the correct hardware part.

As we see from the two above diagrams, we update properties of the leaf blocks upon entry to the new state. These properties represent the state of the hardware and will eventually allow higher-level state machines to react to the combined states of all its parts.

Composite blocks

We now have one state machine for every leaf block in the **structure diagram**. For each composite block, we will now create an orthogonal state machine which will contain one region for each part of the block. In addition, there will be one region for each of the expected incoming signals, and one region which will contain the state machine that describes the behaviour of the composite block, based on the states of its parts and buffers. Below we will use parts of the state machines for *Table* and *FeedBelt* to describe state machines for composite blocks.

Part regions

As we can see from the excerpt of the structure diagram in Figure 31, the top-level block for *Table* consists of five parts, more precisely, two *BiDirectionalMotors*, two *Switches* and one *Potentiometer*. For each of these parts, we add the related state machine to a new region in the orthogonal state machine for *Table*, as shown in Figure 32. To differentiate between the different state machines of the same type, we also add the role names from the structure diagram to the state machines.

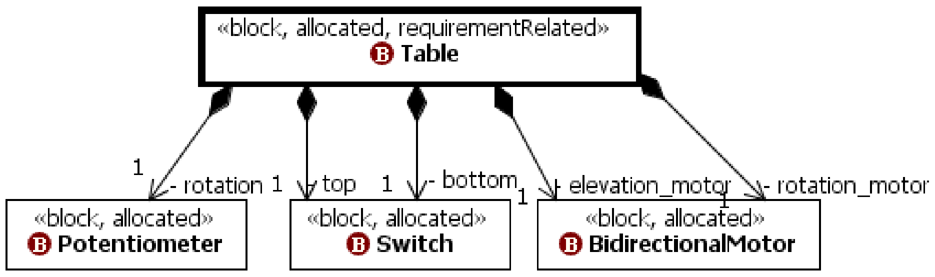


Figure 31 - Excerpt of structure block definition diagram, whole diagram can be seen in Figure 44.

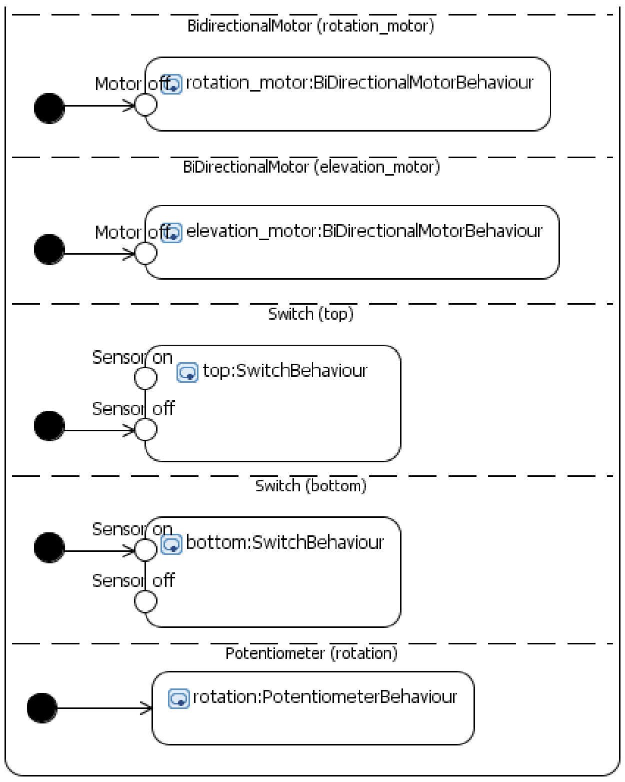

BidirectionalMotor (rotation_motor)
Motor off
rotation_motor:BiDirectionalMotorBehaviour
BiDirectionalMotor (elevation_motor)
Motor off
elevation_motor:BiDirectionalMotorBehaviour
Switch (top)
Sensor on
Sensor off
top:SwitchBehaviour
Switch (bottom)
Sensor on
Sensor off
bottom:SwitchBehaviour
Potentiometer (rotation)
rotation:PotentiometerBehaviour

Figure 32 - Excerpt of orthogonal state machine for Table, showing only the regions for the parts of Table. Whole diagram in Figure 91.

Buffer regions

There are two different buffer designs in the model of this case study. We will show examples of both using the buffer regions of *Table* and *FeedBelt*.

For the *FeedBelt* state machine, we add one region for each expected incoming signal as described earlier. This way we consume the signals when they arrive, which may be in any order and at any time. E.g. when the *Add_Blank* signal is received, the transition between the *AddBlankBuffer Empty* state and the *AddBlankBuffer Full* state is triggered. The state machine for the *AddBlankBuffer* will reside in the *Full* state until the *FeedBelt's* parts reach the state where both the *photoelectric_cell* and the *conveying_motor* are off. It will then transition back to the empty state and call *add_blank()*. In the full state machine diagram in Figure 90, we can see that this call event will trigger the transition between states *Belt empty* and *Wait for Plate to reach end*. We also have to ensure that all the transition guards from the *full* state to the *empty* state used in the buffers are disjoint, to avoid indeterminism and potential dead locks (see chapter 6.1.1). From the *FeedBelt* example, we see that the two guards will never be true at the same time.

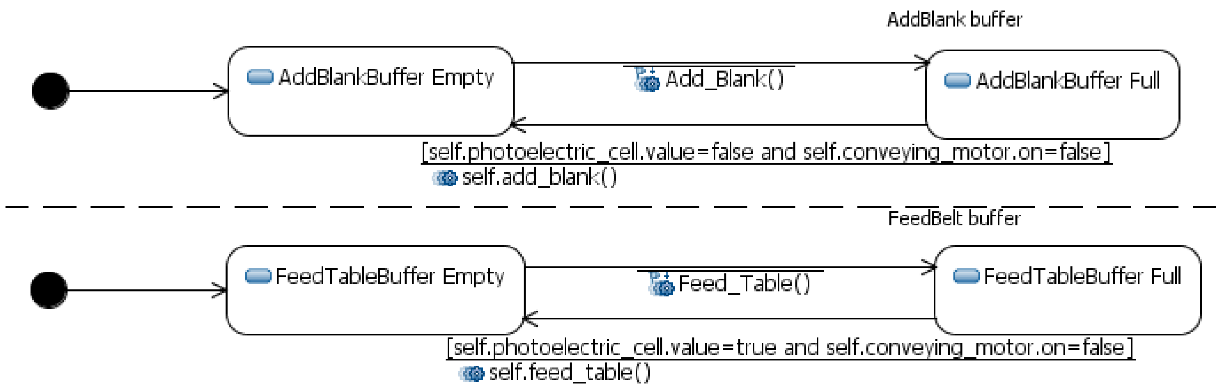


Figure 33 - Buffer regions from state machine for FeedBelt. For whole diagram, see Figure 90.

Even though the events sent between the top-level blocks are described as asynchronous in [5], we can see by studying the activity diagram that when either *Table*, *Press* or *DepositBelt* send a signal, they will wait for the other device to respond before continuing. To emphasize this, we could have modelled the state machines without buffers and let the incoming signals trigger transitions in the main region. However, to keep consistency between the different diagrams, we decided to model these state machines with buffers too. These buffers do, however, have a slightly different design than the other ones, which is shown in Figure 34. There is only one buffer region regardless of number of expected incoming signals. The state machine in this region contains a number of states and transitions where each of the expected incoming signals will trigger a transition. We order them to show that there is an expected sequence in the reception of the signals. In Figure 34 the expected sequence of the incoming signals for *Table* is modelled in the buffer. We add guards to explicitly state when the transition may be triggered, even though it would not be necessary in this case, since the guards describe the only state where the call events may trigger a transition.

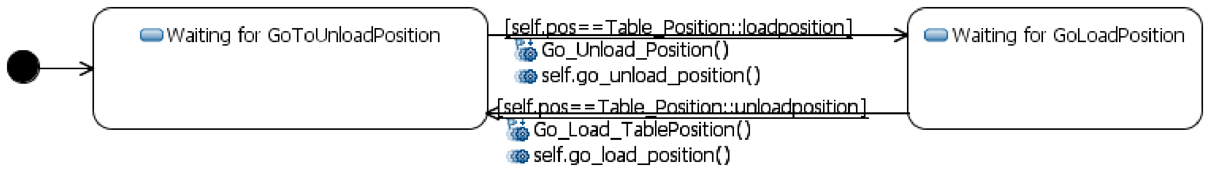

Waiting for GoToUnloadPosition
[self.pos==Table_Position::loadposition]
Go_Unload_Position()
self.go_unload_position()
[self.pos==Table_Position::unloadposition]
Go_Load_TablePosition()
self.go_load_position()
Waiting for GoLoadPosition

Figure 34 – Detail view of buffer region from Table state machine. For whole diagram, see Figure 91.

Main region

In the main region we bring together all the parts of the composite block. The transitions made between the states are according to call events from the buffers or based on the properties of the sensors. Through the effects of the transitions in the main region, we also control the actuators.

In Figure 35 we see the main region of the *Table* state machine, where we can see the different states *Table* can be in. These states can all be described with the values of the

properties set by the state machines of the leaf blocks, as described earlier. Combined the properties form the state invariant for each of the main region states.
In the state machine of the main region of *Table*, we see that there are four states distributed in two composite states. In the composite *Moving* state, we use orthogonal states to describe the states *Moving Up(...)* and *Moving Down(...)*, since a movement in *Table* contains concurrent activity in the two actuator parts, namely *rotating_motor* and *elevation_motor*. Once the related call event from the buffer has been received (e.g. *go_unload_position()*), both motors will start moving and will be stopped individually as soon as their related sensors let the system know that the right rotation or elevation position has been reached. When both regions have reached their final states, indicating that the position of the *Table* has been reached, the state machine makes a transition to one of the sub-states of the *Not Moving* composite state and sends a signal to one of the other top-level blocks, e.g. *^feedbelt.Feed_Table()* (see assumption).

Root block
After creating state machines for all the parts of the controller block, we are now able to arrange them in a state machine for the *ProductionCell_Controller,* where we describe the life span of the controller and the concurrency between the devices.
Figure 89 shows the controller state machine. We focus on the operating state of the system and abstract from the details of the other states. When the controller receives the signal *TurnOn()*, the system starts initializing (see assumptions), and the devices will move to their starting positions.

When all the parts are done with initialization, the state machine will make a transition to the operating orthogonal state, where all the state machines that were created for the composite top-level blocks are arranged into regions to be able to show the concurrency between the parts of the controller.

The state machines for the parts of the system are non-terminating. However, when the signal *TurnOff* is received, the state machine exits the operating state after finishing executing any possible transitions effects or entry/exit activities (see assumptions). After shutting down the system and bringing the system to a safe state, the system goes back to the idle state. Emergency stop is not modelled in this case study.

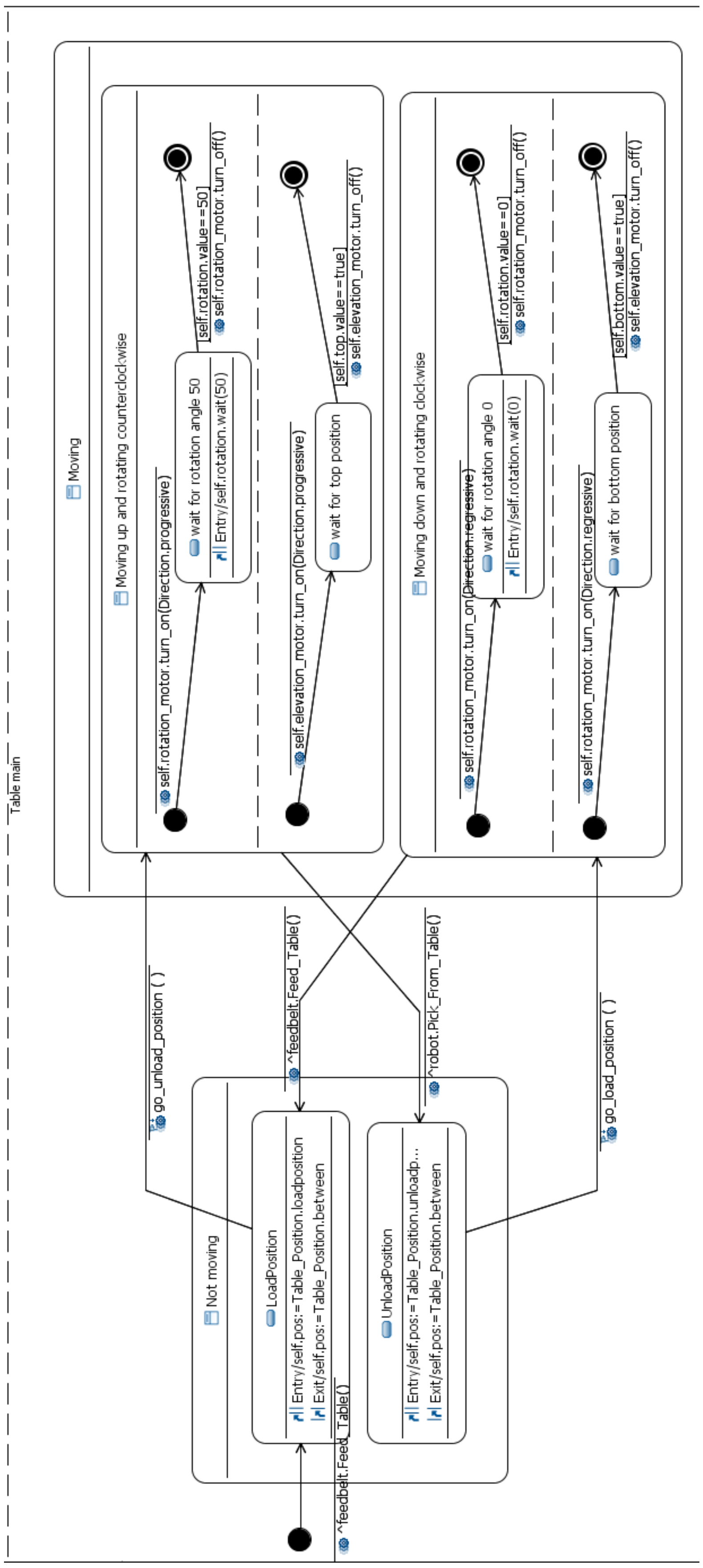


Table main
Moving
Moving up and rotating counterclockwise
self.rotation_motor.turn_on(Direction.progressive)
wait for rotation angle 50
Entry/self.rotation.wait(50)
[self.rotation.value==50]
self.rotation_motor.turn_off()
self.elevation_motor.turn_on(Direction.progressive)
wait for top position
[self.top.value==true]
self.elevation_motor.turn_off()
Moving down and rotating clockwise
self.rotation_motor.turn_on(Direction.regressive)
wait for rotation angle 0
Entry/self.rotation.wait(0)
[self.rotation.value==0]
self.rotation_motor.turn_off()
self.rotation_motor.turn_on(Direction.regressive)
wait for bottom position
[self.bottom.value==true]
self.elevation_motor.turn_off()
go_unload_position ( )
^feedbelt.Feed_Table()
^robot.Pick_From_Table()
go_load_position ( )
Not moving
LoadPosition
Entry/self.pos:=Table_Position.loadposition
Exit/self.pos:=Table_Position.between
UnloadPosition
Entry/self.pos:=Table_Position.unloadp...
Exit/self.pos:=Table_Position.between
^feedbelt.Feed_Table()

Figure 35 – Detail view of main region of the Table state machine. For whole diagram, see appendix, Figure 91.

Assumptions:
(1) There are no plates in the system at start-up of the system.
(2) When the operator turns off the system, we assume that the shutdown state will contain measures to bring all the devices and plates to safe positions.
(3) We use OCL to describe guards.
(4) The syntax *":="* is used for assigning values, to be able to differentiate it from the OCL syntax of the guards.
(5) We use the semantics *"^target.Signalname(args)"* for send clauses, as suggested in [11].
(6) There is no system for resending lost signals, or handling duplicate signals in [5]. We therefore assume that all signals that are sent, will be received once by the target within a acceptable amount of time.
(7) When triggering a transition using a call event for operation *"turn_on"/"turn_off"* or other similar operations in the leaf blocks, we assume that a related signal will sent to the related hardware block, even though this is not modelled in this case study.

Consistency:
(1) There should be a state machine for each block with behaviour in the block definition diagram.
(2) For each operation in a block in the block definition diagram and thus for each activity allocated to that block through a *callBehaviourAction* in the activity diagram, there should be a related call event in the state machine for that block.
(3) For each *acceptEventAction* in the activity diagram which is allocated to a block, and thus for each receive event on a lifeline representing that block in a sequence diagram, there should be a transition which triggers for the same event in the state machine for that block.
(4) For each *acceptEventAction* in the activity diagram which is allocated to a top-level block, and thus for each receive event on a lifeline representing that block in a sequence diagram, there should be a buffer region that contains a trigger event for the same event in the state machine for that block.

4.2.9 Pre- and post conditions (Step 2i)

Input: Structure diagrams (BDD), activity diagrams (ACT)
Output: Pre- and post conditions for the top-level blocks.

In this step, we will specify the pre- and post conditions for the operations in the top-level blocks. These conditions will be useful in the safety design later in the methodology. Pre/post conditions set constraints to what state the owning block and its environment can be in before/after the execution of a method.
The activities which were identified in the **activity diagrams** of this case study are consistent with the operations of the blocks, so we can investigate them to find which pre/post conditions exist for each of the methods in the top-level blocks of the system. By inspecting the **activity diagrams** we can find the state of the system before and after execution of the relevant activity. To find the pre-condition, we investigate all required inputs to an activity, trace the control flow back to the previous activity/activities and check the state of the system. For the post-conditions, we check the state of the system after execution of the activity. In SysML any constraint language can be used. In this case study we use OCL.

For instance, by looking at Figure 21 and Figure 27, we see *FeedBelt* activity *add_blank* and its lower-level activity diagram. For a control token to arrive at the activity *add_blank*, it will come from either *feed_table* or *initialize*. By inspection of these activities (Figure 66 in the appendix), we can see that both the photoelectric cell and the motor of feed belt are turned off when the activities are finished executing (see assumption below).

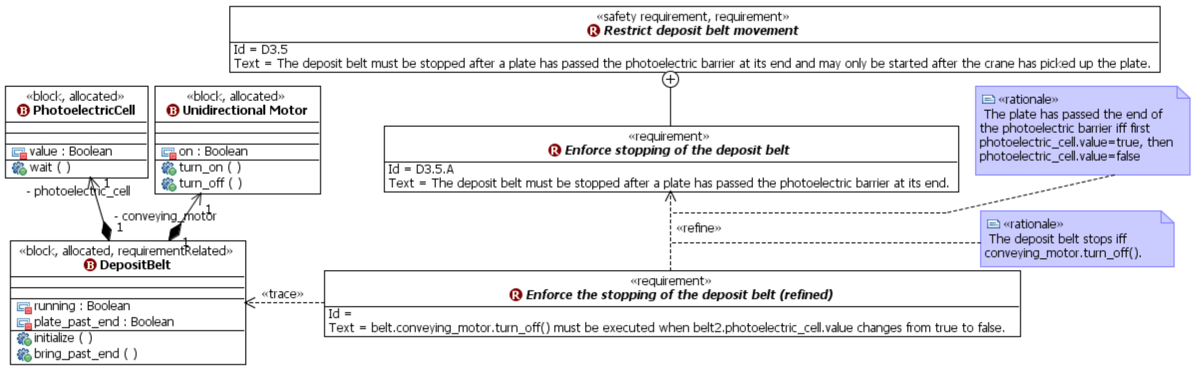
The result of the investigation for the operation *add_blank()* is[12]:
context FeedBelt::add_blank()
pre: self.photoelectric.value=false and self.conveying_motor.on=false
post: self.photoelectric.value=true and self.conveying_motor.on=false

[12] In the tool the context is set. We add it here to specify the context.

4.2.10 Formalizing requirements. (Step 2j)

Input: Requirement diagrams (REQ), structure diagrams (BDD)
Output: Cross-cutting diagram with formalized requirements and related blocks.

In chapter 4.2.3 we derived the system-level requirements into block-level requirements to describe the effect the system-level requirements have on each of the top-level parts of the system. This process made the requirements more specific. However, as they are textual, they may still be ambiguous and be subjects for interpreting.
In this step, we will therefore formalize the requirements using the context of the related blocks and parts, and base it on their attributes and operations. The result of this will be new refined requirements which are related to the originals using the *refine* relationship. This will ease the process when we in the next step will look for design elements that satisfy the requirement.



«safety requirement, requirement»
Restrict deposit belt movement
Id = D3.5
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.
«block, allocated»
PhotoelectricCell
value : Boolean
wait ( )
«block, allocated»
Unidirectional Motor
on : Boolean
turn_on ( )
turn_off ( )
«rationale»
The plate has passed the end of the photoelectric barrier iff first photoelectric_cell.value=true, then photoelectric_cell.value=false
«requirement»
Enforce stopping of the deposit belt
Id = D3.5.A
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end.
- photoelectric_cell
- conveying_motor
1
1
1
1
«refine»
«rationale»
The deposit belt stops iff conveying_motor.turn_off().
«block, allocated, requirementRelated»
DepositBelt
running : Boolean
plate_past_end : Boolean
initialize ( )
bring_past_end ( )
«trace»
«requirement»
Enforce the stopping of the deposit belt (refined)
Id =
Text = belt.conveying_motor.turn_off() must be executed when belt2.photoelectric_cell.value changes from true to false.

Figure 36 - req [Package] Requirements [Formalizing req D3.5]

Requirement D3.5 was originally derived from S3.0 (Figure 114), which has been partitioned into D3.5A and D3.5B. By partitioning the requirement into two smaller ones, they became more specific, though the two partitioned requirements combined should neither add or remove anything from the containing requirement [8]. We will use D3.5A as an example for this step, as shown in Figure 36.
The text of D3.5A is *"The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end"*. Looking at the text of the requirement, we need to ask ourselves two questions: How is the deposit belt stopped? How to register that the plate has passed the end of the photoelectric barrier? We will need to look at the block attributes and methods of *DepositBelt*, which we traced the requirement to in an earlier step. From this we see that the deposit belt is stopped by calling the part *conveying_motor*'s method *turn_off().* We support the formalization of the first part of the requirement with the rationale:

"The deposit belt is stopped iff belt2.conveying_motor.turn_off()".

We use the full name from the perspective of the controller as there are several *conveying_motors* in the system. For the second part of the requirement, we know that if a plate is going to pass the photoelectric cell, the cell’s value has to be true first, then false. The rationale we use for this is:

“The plate has passed the end of the photoelectric barrier iff
first belt2.photoelectric_cell.value=true, then belt2.photoelectric_cell.value=false”

After we have formalized these parts of D3.5A, we can create a new refined requirement with the rationales connected, so that we provide the history and reasoning of the refinement. The formalized text will be:

“belt2.conveying_motor.turn_off() must be executed
when belt2.photoelectric_cell.value changes from true to false.”

This process is done for every requirement. Using the tool, we create one diagram per main requirement. We will continue working on these in the next step. Because of time constraints, we have formalized only a small set of requirements.

Side notes:
(1) Some of the requirements in this step may give pre- or post conditions to block methods. These pre/post conditions for the methods can be compared to those that were found in the previous step. This way we may also uncover faults in the design or discover requirements which are not precise enough.
(2) SysML also allows the creation of test cases for each requirement like we show an example of in Figure 37. As in regular test case generation, test cases should be made independent from the design process.

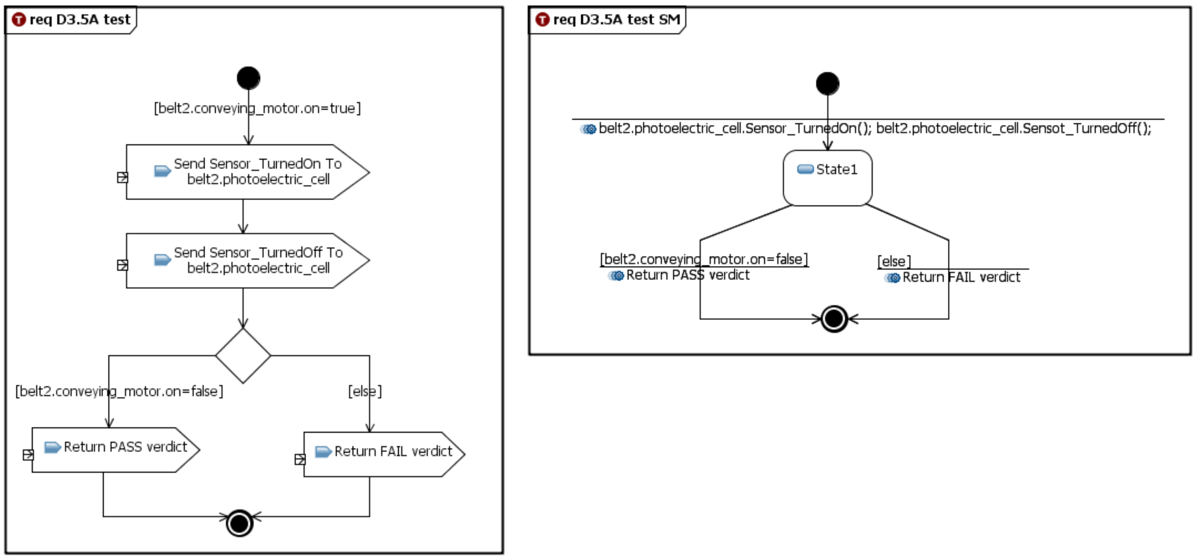

req D3.5A test
[belt2.conveying_motor.on=true]
Send Sensor_TurnedOn To belt2.photoelectric_cell
Send Sensor_TurnedOff To belt2.photoelectric_cell
[belt2.conveying_motor.on=false]
[else]
Return PASS verdict
Return FAIL verdict
req D3.5A test SM
belt2.photoelectric_cell.Sensor_TurnedOn(); belt2.photoelectric_cell.Sensot_TurnedOff();
State1
[belt2.conveying_motor.on=false]
Return PASS verdict
[else]
Return FAIL verdict

Figure 37 – Test cases to verify the requirement D3.5A

4.2.11 Mapping requirements to the design. (Step 2k)

Input: Set of formalized cross-cutting diagrams with formalized requirements (REQ), all design diagrams.
Output: Safety analysis cross-cutting diagrams containing requirements and related slices of diagrams.

In this step we will map the requirements we formalized in the previous step to the relevant slices of the design diagrams, so that we can be able to support safety arguments/cases for the certification process. A *slice* is a relevant part of a diagram which shows that a requirement is satisfied. For each formalized requirement, we will investigate the diagrams of the related blocks to find all the relevant slices to prove that the requirement is satisfied in the design. The requirement D3.5A, which we use as an example in this step, is related to the behaviour of the system, so to find the slices needed, the subjects of our investigation will be the behavioural diagrams of the model.

The text of the formalized requirement of D3.5A is *"belt2.conveying_motor.turn_off() must be executed when belt2.photoelectric_cell.value changes from true to false"*. By looking at the block traces defined for the requirement earlier in Figure 114, and the structure diagram in Figure 44, we see that we have to check the behavioural design models created for *DepositBelt* and its parts, namely *photoelectric_cell* and *conveying_motor.* To check this, we want to find all situations in the model where the *photoelectric_cell* of *DepositBelt* goes from true to false, to verify that the *turn_off()* method of *conveying_motor* of *DepositBelt* is always executed right after.

In our model, we have to check the **activity diagrams** and the **state machines** to find all the relevant slices[13]. This means investigating all activities that are allocated to *DepositBelt* or any of its parts, in addition to investigating the composite state machine for *DepositBelt.*
In Figure 38, which is an extension of Figure 36 from the previous step, we show all the relevant slices from our diagrams that relate to the situation where the sensor goes from true to false. When checking the **state machines** for *DepositBelt* and its parts, we find the situation described in the main region of the state machine. More precisely, we find that when the property *belt2.photoelectric_cell.value* changes from *true* to *false,* a transition is made, so that the exit activity *conveying_motor.turn_off()* of the *Belt running* state is executed.
In the **activity diagram**, we check the activities *initialize* and *bring_past_end,* which both are allocated to *DepositBelt* in the diagram *act produceForgedPlates* and find that the related situation is described in the activity *bring_past_end,* where it receives the signals

[13] We have not modelled this scenario using sequence diagrams in this case study, as we have used sequence diagrams only for modelling interaction between the top-level blocks.

Sensor_TurnedOn and *Sensor_TurnedOff* in sequence and then executes the activity *turn_off*, which is allocated to the *conveying_motor*.

All slices that are identified are then connected to the related requirement using the **satisfy** relationship, meaning that we in the role of safety engineers see that the slice satisfies/fulfils the targeted requirement. When it comes to other requirements, we might only need small parts of state machines regions or activities, or we might need sets of several slices to satisfy a requirement. However, to describe this specifically in a diagram requires more tool support.

Even though this is not a focus of this case study, we wish to mention that any related test cases that have been created may also be mapped to the requirement using the **verify** relationship, as shown in Figure 118 in the appendix.

At this point, it is impossible to give an exact recipe how to map any requirement to the model, as the complexity depends on the requirement in question. But the big lines can be extracted from the steps in the requirement management in this methodology;

- Capture, partition, derive and refine the requirements.
- Formalize the requirements to meet the design terminology; attributes, operations etc.
- Map the requirement to related slices of the structure or behaviour
- Verify the requirements using test cases.

We need to keep a close eye to the development of the requirements to ensure that no wrong assumptions are made about the system or its environment that may propagate into the design of the system. By partitioning, deriving and refining the requirements it may be easier to show that the requirement is fulfilled or not. Also, by keeping track of all the assumptions made, domain experts may investigate these to ensure that no wrong assumptions have been made.

Also see chapter 5.1 for description of tool problems, and chapter 6.1.3 for lessons learned regarding traceability. For more examples of traceability mappings for safety design, see appendix, A.10.

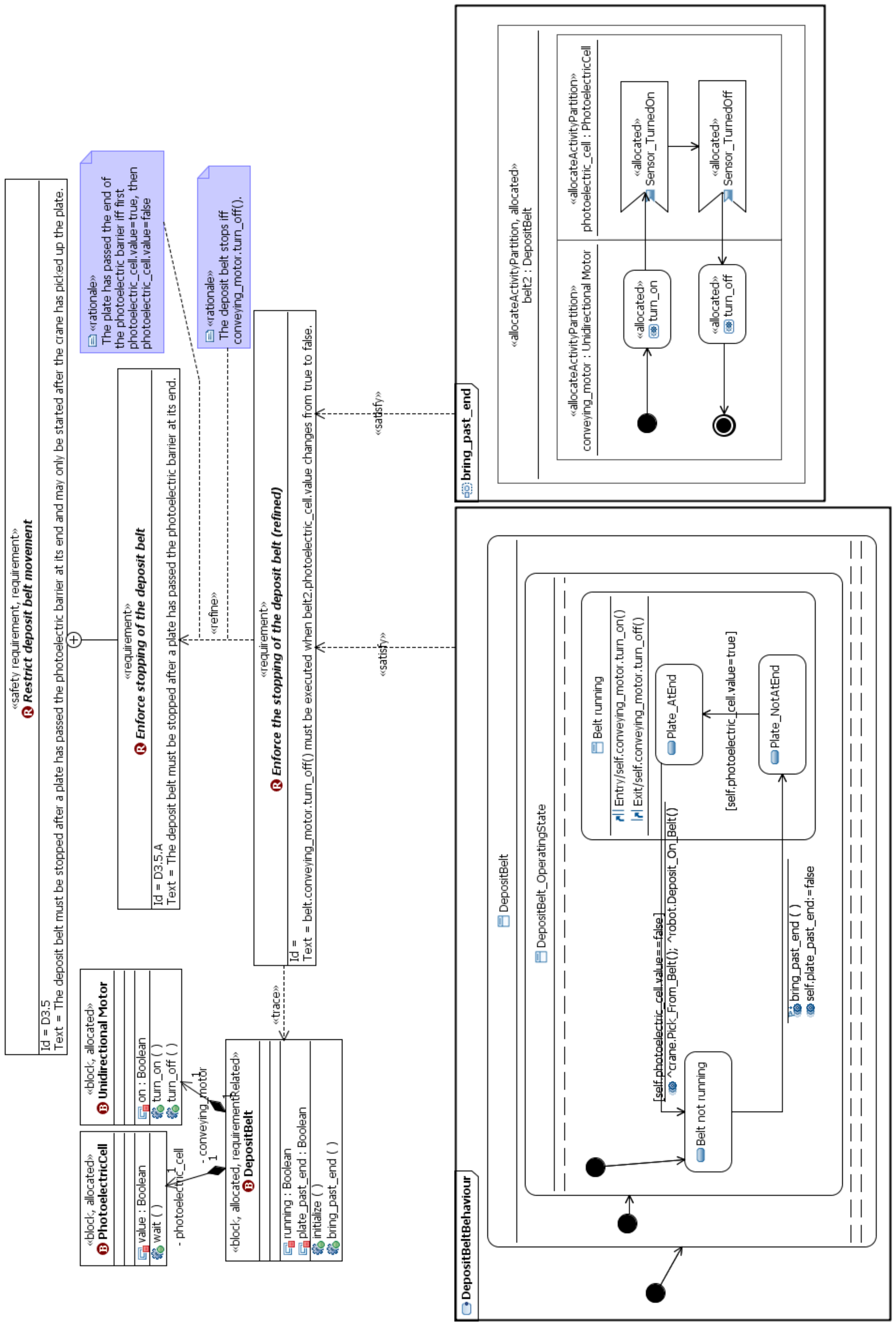

«safety requirement, requirement»
Restrict deposit belt movement
Id = D3.5
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.
«rationale»
The plate has passed the end of the photoelectric barrier iff first photoelectric_cell.value=true, then photoelectric_cell.value=false
«requirement»
Enforce stopping of the deposit belt
Id = D3.5.A
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end.
«rationale»
The deposit belt stops iff conveying_motor.turn_off().
«refine»
«requirement»
Enforce the stopping of the deposit belt (refined)
Id =
Text = belt.conveying_motor.turn_off() must be executed when belt2.photoelectric_cell.value changes from true to false.
«satisfy»
«satisfy»
«trace»
«block, allocated»
PhotoelectricCell
value : Boolean
wait ( )
«block, allocated»
Unidirectional Motor
on : Boolean
turn_on ( )
turn_off ( )
- photoelectric_cell
- conveying_motor
«block, allocated, requirementRelated»
DepositBelt
running : Boolean
plate_past_end : Boolean
initialize ( )
bring_past_end ( )
bring_past_end
«allocateActivityPartition, allocated»
belt2 : DepositBelt
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
«allocated»
turn_off
«allocated»
Sensor_TurnedOn
«allocated»
Sensor_TurnedOff
DepositBeltBehaviour
DepositBelt
DepositBelt_OperatingState
Belt running
Entry/self.conveying_motor.turn_on()
Exit/self.conveying_motor.turn_off()
Plate_AtEnd
Plate_NotAtEnd
[self.photoelectric_cell.value=true]
Belt not running
[self.photoelectric_cell.value==false]
^crane.Pick_From_Belt(); ^robot.Deposit_On_Belt()
bring_past_end ( )
self.plate_past_end:=false

Figure 38 - Cross-cutting diagram bringing together the structure, behaviour and requirement diagram for requirement D3.5A.

5 Tool problems

We used IBM Rational® Software Architect™ Standard Edition, Version: 7.5.0, [12] extended with the plug-in EmbeddedPlus SysML Toolkit 2.5.1.1, [13]. In this chapter we will describe some of the problems we met during the development using these above-mentioned tools. The problems mentioned have been investigated without finding a solution. However, we do recognize that some problems may be user problems, where the solution may exist in the tool, but is not found. In Appendix C there is an additional list over tool problems we have not mentioned here.

5.1 Graphical

In our view, the biggest challenge of the tool is **graphical.** To be able to quickly get an overview, ensure that a model is correct and avoid misunderstandings when reading a diagram, it should be built up as logical as possible and the connectors should not be tangled.

Tangled connectors
When opening a previously stored diagram, it tends to open with tangled connectors, even if they were arranged neatly upon saving and closing the diagram. We also experienced that moving one element in a diagram could cause connectors to be rearranged illogically. For instance, if we rearranged the position of an action which was connected to a fork node while working in an activity diagram, all outgoing connection lines were rearranged illogically. This problem mostly occurred when the diagrams grew large. Hence, making small changes to and getting new screenshots of a diagram was very time consuming, as the connectors needed to be rearranged. We discovered that by choosing the *"arrange"* function in the menu of the tool followed by "*undo*", the diagram was brought back to the last saved diagram, with neatly arranged connectors intact. However, this did not always work.

Requirement blocks
Graphically, it was a problem to get a good overview of the requirements. The text in requirement blocks does not wrap, so the blocks had to be very wide for us to be able to read the whole text. This made it difficult to keep the requirement diagrams nice, especially when we started to partition and derive the requirements. We think it would be helpful to be able to export the diagrams into tables, as mentioned in [8].

Slices
When modelling traceability, we wanted to set focus on specific parts of a diagram. To do this, we dragged existing diagrams into the safety analysis diagram. However, the layout of the original diagram was not preserved in the copy, and the nodes and connectors were in a

jumble. Our solution was to either rearrange the diagram copy, which was time consuming at best, or use screenshots from the original diagram. The latter is not a solution when we in the future wish to automate the diagram, since the screenshot contains nothing but visual information. A solution, if made possible in the tool, could be to import a slave of the original diagram, where no changes of elements are allowed, but where we could filter out the uninteresting parts of the diagram and thereby highlight the parts of the diagram we are interested in, or by preserving the original layout of the diagram and then closing all compartments, except the ones with the information we wish to highlight.

Internal block diagrams - Boundary ports

When creating internal block diagrams, we define the structure of the parts of a block. This block may have *ports*, which are called boundary ports when describing them in the diagram of the owning block. However, we were not able to show these boundary ports. For instance, when describing the physical item flow (as in Figure 48), we originally created the diagram for the block *ProductionCell_System.* We were able to create boundary ports in the diagram at first, but when opening the diagram after closing it, both ports and any item flow connectors connected to it, were gone. The ports were still in the model. We solved this by creating the internal block diagram for higher level block, *ProductionCell_Domain,* instead. This made the tool treat the flow port of *ProductionCell_System* as a normal flow port.

Internal block diagrams - Item flows

We had a similar, but opposite problem with *item flows*. When deleting an item flow from the internal block diagram and model, it did not disappear from the diagram until we closed and reopened it.

5.2 SysML Compliance

According to a report on the SysML Forum[14], no tools are complete regarding compliance to the SysML standard. However, if we were to follow this further, we would need information regarding how they have measured the different tools, and find out what was lacking in the different tools.

Missing frames, regions and boundaries

In the SysML standard [7], there is a requirement regarding diagram frames. Frames are missing in the block definition, requirement and use case diagrams of the tool. According to email correspondence with EmbeddedPlus, this is an error in the specification, and they are working to correct this in the next version of SysML. Two similar problems are that system boundary is missing for use cases, and interruptible regions are missing for activity diagrams.

[14] http://www.sysmlforum.com/tools.htm

Parametric diagram

Using these tools gives us only a limited possibility of creating parametric diagrams. According to the SysML standard [7], it should be possible to create parametric diagrams for both regular blocks with constraint properties and for composite constraint blocks. However, the tool only allows parametric diagrams for the regular blocks.

In chapter 4.2.6, regarding the creation of constraints on physical blocks, we originally wanted to make a parametric diagram for a composite constraint block. The purpose of this was to make the diagram more readable, try out the composite constraint blocks and show an example of reusing constraint blocks. Even though the tool allowed us to create composite constraint blocks, it was not possible to create parametric diagrams for them.

Send/receive signals in state machines

The SysML standard also describes the usage of a graphical notation for *sendSignal* and *receiveSignal actions* in state machines. This is not available in the tool. Describing the sending and receiving of signals graphically may be better, as to be more specific, preserve consistency, make diagrams easily readable and may also be easier to automate. Text is error prone.

5.3 Usability problems

Internal block diagrams

When creating internal block diagrams, all properties, references, ports and parts are shown as default upon creation of the diagram (though none of the connections). With several internal block diagrams, it is easy to accidentally delete a port or part which is in use in another diagram. When this happens, we get a warning, but with no possibility to cancel, the element is then deleted from the model anyway. It is possible to undo after deleting, but a cancel option might be a better feature.

Error messages

The error messages in the tool are sometimes very subtle and give little explanation to where the problem is. This goes for error messages that occur during validation of the model and normal error messages.

Activity diagram/User problem

In the activity diagram we defined parameters for some of the activities, e.g. the Table activity *act move* in Figure 68. However, when calling the activity, as in Figure 67, we were not able to specify a value for that parameter. There is an action called createObjectAction, which we can type with the ValueType we want, but we did not manage to find out how to specify, for instance, the *loadposition* literal value of the *Table_Position* valuetype.

6 Discussion

6.1 Lessons learned

During the development of the model, several issues needed to be closely inspected.

6.1.1 Asynchronous communication

The asynchronous, concurrent behaviour in the system created some challenges during modelling. These situations had a tendency to surface while modelling the robot. The robot was designed receive several concurrent incoming signals from three different sources and act on them according to its state and was the most complex of the devices in the system.

(Lack of) Ordering of events
When we modelled the interactions of the top-level blocks, using the sequence diagram (see Figure 58), an ordering[15] was imposed on the send and receive events. However, as there are many possible orderings of events that may occur, this sequence diagram only describes one of the possible orderings of events in the system.
By using *combined fragments* we tried to show more of the possible orderings, but at the top-level this wasn't possible as either the combined fragments would have had to been allowed to overlap, or the sequence diagram would be very complex which would make it prone to errors and thus work against its purpose. In the high-level activity diagram (see Figure 60), however, we were able to show the interaction and concurrency without imposing an ordering.

Deadlocks
When we modelled activities, we used the existing notation in the activity diagram to model that each device may accept the expected incoming signals at any time by using *AcceptEventActions*. When a signal is accepted, it is stored in the input pin of the related activity until all object/control tokens are present at the inputs of the activity and the signal is then consumed. In this sense, the asynchronous communication and ordering are taken care of. However, we met a different challenge in the activity diagrams when designing the robot behaviour. In the partition of the robot, we added a decision node (see Figure 94) and guards so that the control flow can choose its path based on the state of the robot. Since the robot has two arms, there is always the choice of two activities, which can be executed if the related signal has been accepted. By inspecting the diagram closer, we discovered that the way we modelled the behaviour of the *Robot* may cause dead locks (see Figure 62). For instance, if

[15] An ordering of events occurrences is also called a *trace* [14] S. S. Alhir, *UML in a nutshell: a desktop quick reference*: O'Reilly & Associates, Inc., 1998..

the robot is in a state where arm1 is loaded, and the press is still loaded, the control token may still choose the path to the *load_press* activity. Hence, we will experience a dead lock, since the *Robot* will be waiting for the *Press* to send the *Load_Press* signal. Whether the related signal has been accepted or not is not a condition for the guard.

We did not have time to fix this issue in this thesis, but we have the following suggestions for solutions: (1) Add an extra condition to the guards, either by checking whether the received signal has arrived, or checking the press property *pos*, which describes the position of the *Press*, or (2) impose an ordering to the activities, which may cause us to reuse the same activity several times in the same partition. In that case, we may create central buffers for each of the incoming signals, where the signals will be stored when they are accepted. These buffers can be shared by the activities waiting for the same signal.

Race conditions

Yet another situation surfaced when we were modelling the state machine for the *Robot*. To be able to receive incoming signals at any time, we had two design choices, either (1)defer each signal event in all states except in the state where it is expected to trigger a transition, or (2)create signal buffers. Since [10] advised against deferring signal events, we decided to create signal buffers, as described in 4.2.8.

In state machines, where we have several concurrent signal buffers, it is important that the guards of the buffers are disjoint, so that we avoid race conditions. E.g. if the two of the buffer guards of *Robot* are not disjoint, then they may both trigger a transition at the same time. The effect on both of the transitions will each call an event in the main region state machine, and the outcome is undetermined. One of the call events will be lost, and since the signal is not repeated, we will eventually end up in a dead lock. An example is if *Press* and *Table* send signals to *Robot*, and the state of *Robot* triggers a transition in both buffers. If robot picks the plate from the *Table*, the *Robot* will never pick the plate from *Press,* and we have a dead lock.

To avoid this, we investigated the possible signals for every state in the composite idle state of *Robot* and discovered that there are two possible signal buffers that may transition for each of the states (see Table 2). In the state *Arm1_loaded*, the two signals are sent from the same source and will never be present in the buffers at the same time, so we could merge the two buffers into one, as shown in Figure 94.

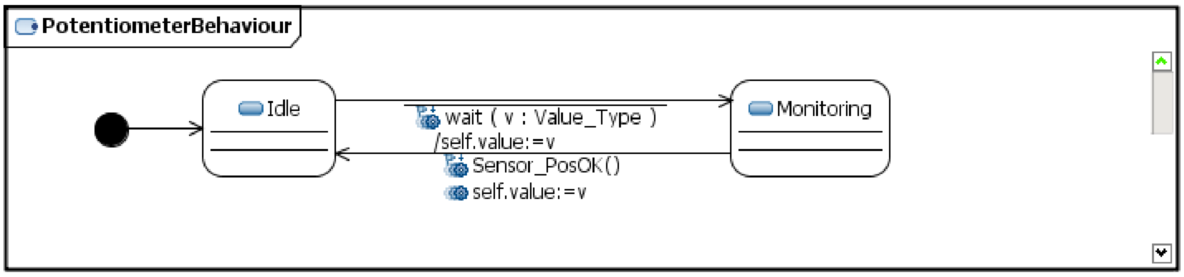
ARM 1 LOADED	**ARM 2 LOADED**	**ROBOT_STATE**	**Signal buffers that may trigger a transition**	
FALSE	**FALSE**	**Unloaded**	**PFT**	**PFP**
FALSE	**TRUE**	**Arm2_loaded**	**PFT**	**DOB**
TRUE	FALSE	Arm1_loaded	LP	PFP
TRUE	**TRUE**	**Arm12_loaded**	**LP**	**DOB**

Table 2 – Describing in which signal buffers each state of the Robot may trigger a transition.
(PFT – Pick from table, PFP – Pick From Press, LP – Load Press, DOB – Deposit On Belt)

To avoid the race condition problem in the other possible states, we added a priority between sets of signal buffers, by adding a property that is set when the signal is received. If two buffers are in danger of transitioning at the same time, we add a priority to one of the transitions that has to wait. We gave priority to the signal events that were sent by devices that were the last in the production chain of the two. This experience taught us the importance of disjoint buffer guards between the regions, in other words, there should be an XOR relation between them.

6.1.2 Hardware devices and communication with software

While working with this thesis we have learnt more on how hardware and software may communicate and how different types of sensors work. Even though the hardware has been mostly scoped out of this thesis, we needed information as to how e.g. the hardware potentiometer may work to be able to create a realistic state machine. Switches and photoelectric cells are straightforward, as they have two states, on or off, while potentiometers may be designed in different ways. In our design, we assume that we in the call event *wait(v:Value_Type)* send a signal to the hardware potentiometer so that it will let us know when the related device is in the right position by sending us a signal *Sensor_PosOK* when this happens. This way we are able to abstract from the details about how the actual potentiometer works.



While working with this case study, we have understood that the object-oriented solution we base our design on is not typical for control systems. In a full solution there would for instance be a driver between the hardware and software that transforms the signal from the hardware, into a signal that the software can interpret, but as this is not described in [5], we have abstracted away from the driver and assume that the controller understands the signals from the hardware.

6.1.3 Traceability and mapping of requirements

Based on our experience in this thesis, we see that creating traceability links to support the generation of safety cases can be a lot of work and requires specific effort throughout the development process. It demands that the safety engineers have a lot of knowledge about the system, its usage and properties to be able to trace the requirements to the appropriate blocks, operations and diagrams. By adding the traceability during the development, it seems to ease the process, though this would need to be investigated further. Even though we had firsthand knowledge about the system, the modelling and extracting of the correct information and slices from the diagrams proved to be difficult due to tool problems as described in chapter 5 regarding slices. This will demand more tool support.

One aspect that came to mind during the work with this thesis, is that the safety analysis design may become outdated, if parts of the system change, e.g. during development or in later versions. In future work, we also need to make sure that changes made in the design are updated in the safety analysis design to keep the diagrams consistent, and that changes are marked in case the changes require further inspection regarding the safety analysis.

Another issue is that tracing requirements from both the activity diagram and the state machines may result in some redundancy. From the experience with this case study, we learnt that one composite state machine diagram describes the whole behaviour of a block and the states of its parts, while we need to inspect several activity diagrams to discover the behaviour or state of the same block. More specifically, the main region and buffers of a composite state machine, combined, contain the most specific and compact information in the model as all the information about each part of a block and its behaviour can be shown in one diagram. However, the diagrams may easily become complex, but by using composite states, we are able to view the design of the device at different levels. In addition, the state machines also contain state invariants and properties, which provide a lot of useful information. To find the state of the parts of the system in an activity diagram, it requires more effort, as we need to inspect several diagrams. For instance, by looking at the safety analysis diagram for requirement 3.4 in the appendix A.10.3, we see that to be able see the state of the *Table* before the activity *feed_table* is executed by *FeedBelt,* we need to inspect the diagram *act move* which is called from *act go_loadposition,* which is the last activity before it sends the signal *Feed_Table*. On the other hand, the activity diagrams use the sequences of actions to describe block behaviour in a very intuitive manner. Specifically, when it came to the safety analysis conducted in this thesis, activity diagrams gave a visually good overview to show sequences of actions between the environment and system, e.g. *bring_past_*end in D3.5A. However, which diagram or combination of diagrams is most useful in which contexts will depend on the requirement and needs more research.

6.2 Conclusion

6.2.1 SysML Evaluation

A modelling language will give positive effects, by gaining a common and unambiguous understanding of a system by looking at the system at a higher abstraction. In this case study

we use SysML to bring together system, software and safety engineers and this may bring a common understanding of the system.
As a novice to modelling systems, there are a lot of new concepts that may be difficult to comprehend at first. There are several similar concepts that describe the same parts of the model in different contexts, e.g. block/part or action/activity. In any language, there is a threshold for learning a new modelling language, but since SysML is a subset of the standardized and much used modelling language in both industry and education, UML, and the extensions of SysML are built upon similar constructs that we are used to see in UML; the learning threshold of the language may be lower. However, the similarities between the languages may also cause some confusion. Tool problems or tools that do not comply with the standard may also cause confusion about the language.
As for safety design, the cross-cutting links, which we use between the requirements and model elements, help us bring the different parts of the system together and we think this is helpful on the way to generating safety cases. At the point we reached in our research in this case study, the SysML constructs proved to be able to describe what we set out to do, namely to trace which parts of the design diagrams each requirement was satisfied. However, we think that there are some challenges that may need to be met. In addition to the tool problems regarding slicing, one challenge was to specifically show when a sets of slices originating from diagrams from separate top-level blocks were collaborating to satisfy a requirement. It might be useful to look into expanding the profile to create more specific constructs to be able to handle sets of slices, smaller slices or specific sequences.

6.2.2 Tool evaluation

The application Rational Software Architect from IBM[16] is based on Eclipse, and is an extendible and quite user-friendly tool when it comes to the user interface. It is easy to install, and easy to extend with plug-ins like the SysML ToolKit from EmbeddedPlus[17]. As described in chapter 5, some of the problems of the tools caused us to use a lot of time on tidying up the models. As we are new to this tool, it took some extra time to investigate if some of the problems actually were problems with the tool or if we just hadn’t found the solution yet. As mentioned, confusion about SysML constructs may also be caused by problems with the tool, e.g. properties that need to be set may be hidden in the user interface. With more experience with the tool, however, one will know where to look for solutions. If new users are familiar with Eclipse, we think the threshold for learning this tool is lower. The price of the tools are quite high, however, which may be a high threshold for purchasing the tools. All in all, for the work performed in this thesis, the tool left us with mostly positive experiences to be able to fulfil our task, even though some of the problems described in chapter 5 and appendix C, made it challenging at times, and should be fixed.

[16] http://www.ibm.com
[17] http://www.embeddedplus.com

6.3 Future work

Fix shortcomings

As mentioned earlier, there are some shortcomings to this model that needs to be corrected and extended. The error in the Robot partition of the activity diagram as described in 6.1.1 needs to be fixed, and depending on the decisions made, some changes may be needed in the state machine for Robot as well to keep the models consistent.

More experience in traceability and analysis

In this thesis we have traced six requirements to the model elements. To be able to make well-educated decisions, more experience is needed in this field, e.g. to be able to realize which diagrams are most convenient to use in certain contexts. Performance also plays an immense role in safety, e.g. showing that the feed belt motor must stop within a certain time after the photoelectric cell has turned on is important for the safety requirement to be fully satisfied. More examples regarding performance analysis might therefore be interesting to look into.

Tool and language support

From the experiences with the tools used in this case study we see that we may need more tool support to be able to effectively mark slices of a diagram. Also, we need to be sure that slices are kept up to date when changes are made to the system. In addition, since the trace construct we use to trace requirements to related blocks is a very general and weak relationship, it might be interesting to look into extending the profile, to create more specific constructs to fulfil the purpose of safety analysis.

Automation

The final goal is to be able to automate the generation of safety cases. How to automate and extract the traced parts of the model to be able to support the generation of safety cases is an important part of future work.

7 References

[1] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda, "Mars Polar Lander fault identification using model-based testing," in *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard*, 2001, pp. 128-135.

[2] D. Jackson, M. Thomas, and L. I. Millett, *Software for Dependable Systems - Sufficient Evidence?*: The National Academies Press, 2007.

[3] European Aviation Safety Agency (EASA), "Annual Safety Review," 2008, http://www.skybrary.aero/bookshelf/books/825.pdf

[4] L. Briand, T. Klykken, S. Nejati, R. Panesar-Walawege, and M. Sabetzadeh, "Using SysML to Support Safety Certification: A Methodology and Case Study," Simula Research Laboratory, 2009.

[5] S. Barbey, D. Buchs, and C. Péraire, "A Case Study for Testing Object-Oriented Software: A Production Cell," 1998.

[6] Object Management Group (OMG), "OMG Systems Modeling Language, The Official OMG SysML site," http://www.omgsysml.org/: Object Management Group, 2009.

[7] Object Management Group (OMG), "OMG Systems Modeling Language (OMG SysML), version 1.1," 2008, http://www.omg.org/docs/formal/08-11-02.pdf

[8] S. Friednenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*, 1 ed.: Morgan Kaufmann OMG Press, 2008.

[9] C. Lewerentz and T. Lindner, "Formal Development of Reactive System: Case Study Production Cell," in *Lecture Notes in Computer Science*. vol. 891, 1995.

[10] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual Second Edition*: Addison-Wesley, 2005.

[11] L. Briand, "UML State Machines," in *Course slides of SYSC 3100 System Analysis and Design, Carleton University, Ottawa, Canada*.

[12] IBM, "Rational Software Architect Standard Edition," 2009, http://www-01.ibm.com/software/awdtools/swarchitect/standard/?S_TACT=105AGX15&S_CMP=LP

[13] EmbeddedPlus, "SysML Toolkit for the IBM Rational Software Development Platform (RSDP)," 2009, http://embeddedplus.com/downloads/SysML%20Toolkit%20for%20RSDP.pdf

[14] S. S. Alhir, *UML in a nutshell: a desktop quick reference*: O'Reilly & Associates, Inc., 1998.

Appendix A Production Cell Case Study

A.1 System-level diagrams

A.1.1 Context diagram

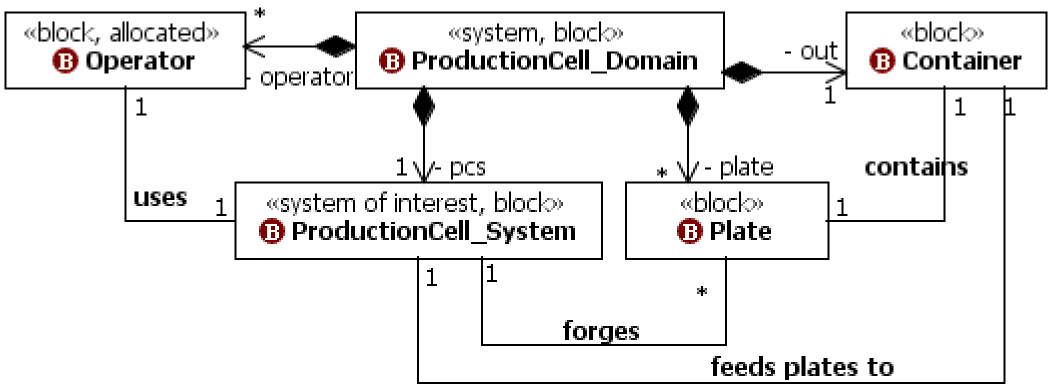
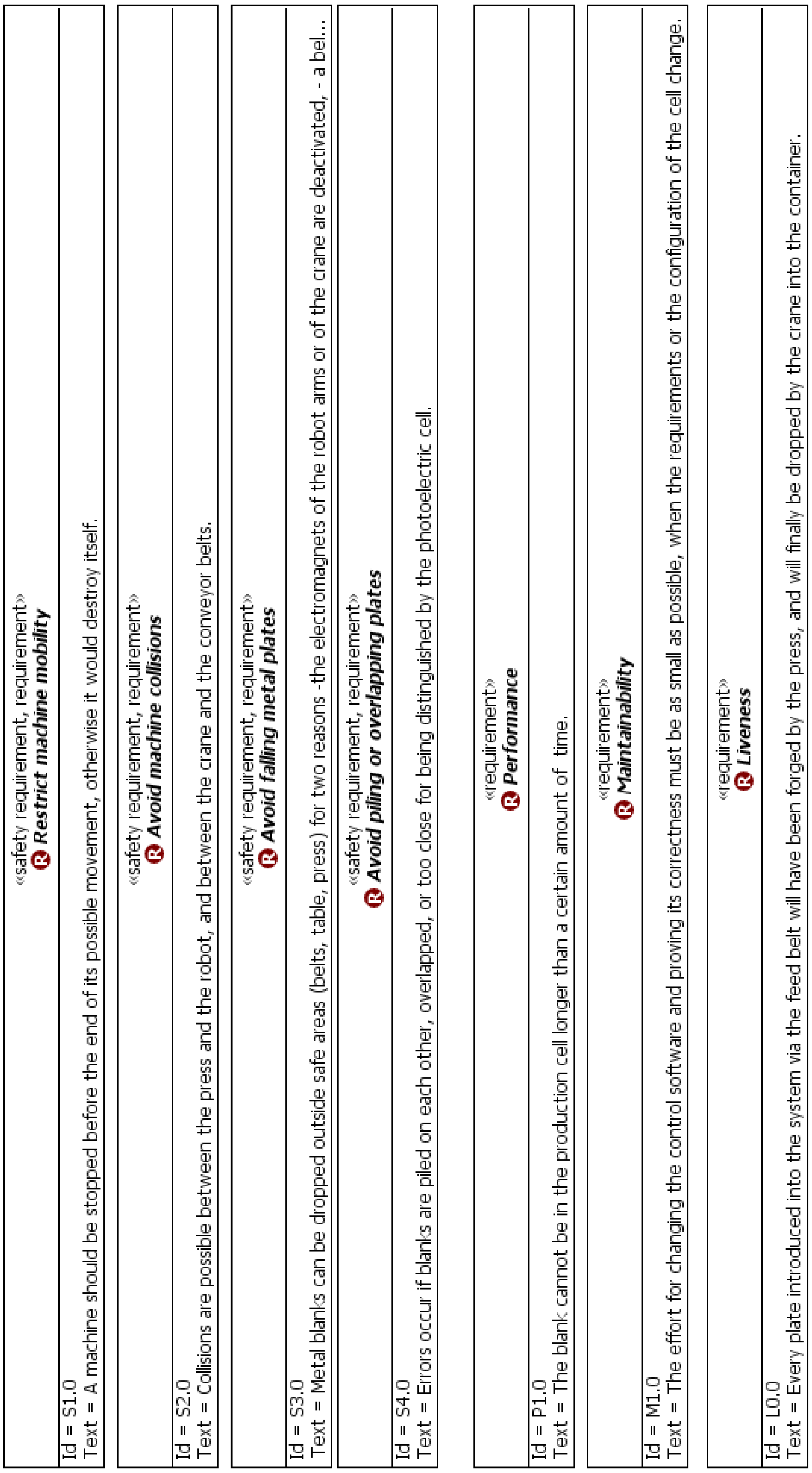


Figure 39 - System context diagram which shows the elements in the environment that may interact with the system.

A.1.2 High-level requirement Diagrams


«safety requirement, requirement»
Restrict machine mobility
Id = S1.0
Text = A machine should be stopped before the end of its possible movement, otherwise it would destroy itself.

«safety requirement, requirement»
Avoid machine collisions
Id = S2.0
Text = Collisions are possible between the press and the robot, and between the crane and the conveyor belts.

«safety requirement, requirement»
Avoid falling metal plates
Id = S3.0
Text = Metal blanks can be dropped outside safe areas (belts, table, press) for two reasons -the electromagnets of the robot arms or of the crane are deactivated, - a bel...

«safety requirement, requirement»
Avoid piling or overlapping plates
Id = S4.0
Text = Errors occur if blanks are piled on each other, overlapped, or too close for being distinguished by the photoelectric cell.

«requirement»
Performance
Id = P1.0
Text = The blank cannot be in the production cell longer than a certain amount of time.

«requirement»
Maintainability
Id = M1.0
Text = The effort for changing the control software and proving its correctness must be as small as possible, when the requirements or the configuration of the cell change.

«requirement»
Liveness
Id = L0.0
Text = Every plate introduced into the system via the feed belt will have been forged by the press, and will finally be dropped by the crane into the container.

Figure 40 - System level requirements, high-level requirements which all will be further partitioned and derived.

A.1.3 Use case diagram

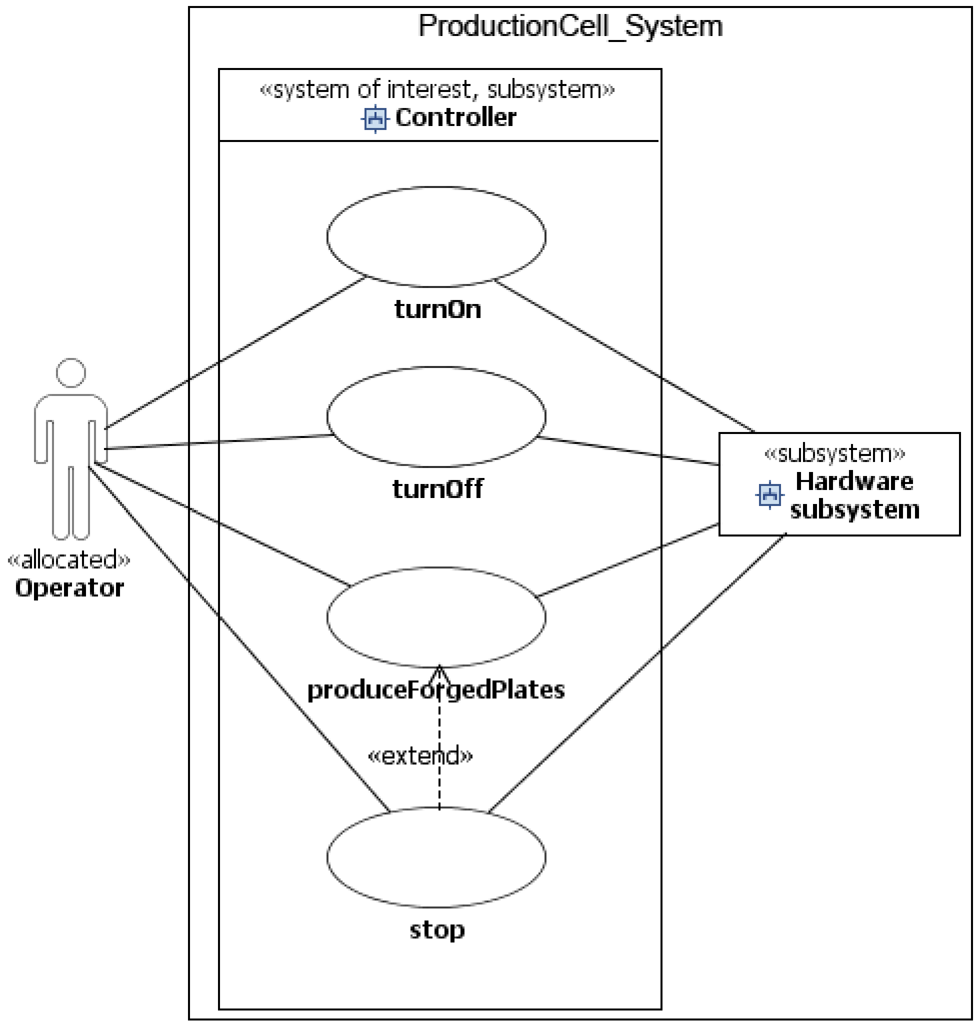


Figure 41 - Use Case Diagram for the Production Cell System, describing the system's main functions and actors

The Hardware subsystem represents the set of hardware devices which will provide input to and receive the output from the controller.

Reg. Use case diagram: The boundary in this diagram has been added to the image outside the tool as we could not locate the system boundary in the tool (see Chapter 5 Tool problems) The allocated stereotype has been removed to ensure that the image is consistent with this stage in the methodology,

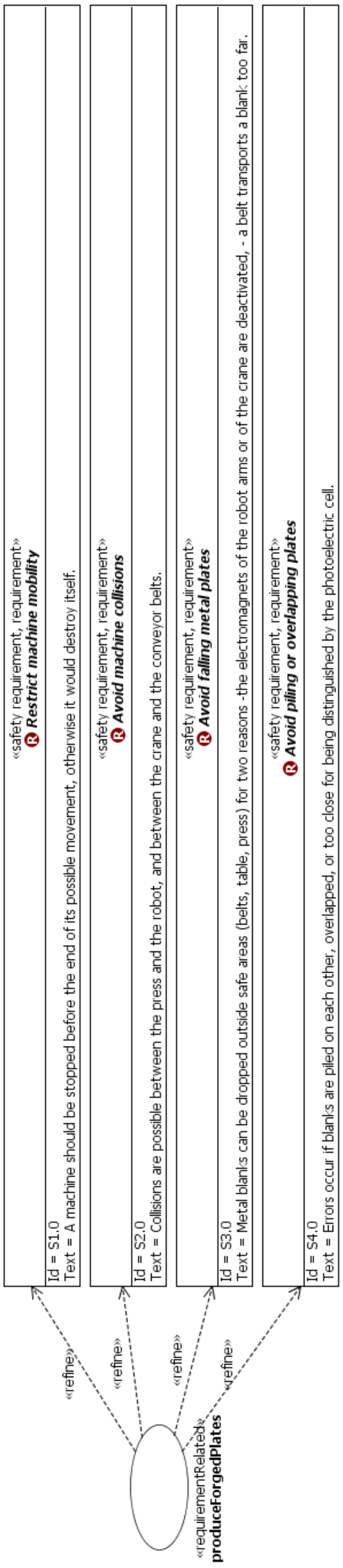
A.1.4 Use case specification for ‘Turn On’

Use Case	***Turn on***
Actor	▪ Operator, FeedBelt, Table, Robot, Press, DepositBelt, Crane
Pre-condition	▪ System is turned off ▪ No plates are present in the system
Description	1. The operator turns on the system. 2. *The system* tells the machines in the system to turn on and go to their initial positions. a. *The system* commands ***the feed belt*** to turn off its motor. b. ***The feed belt*** turns off its motor. c. *The system* commands ***the table*** to move into its load position. d. *The table* lowers and rotates until it reaches its load position. e. *The system* stores a request that ***the table*** is ready for a new plate until ***the feed belt*** is ready to act on it. f. *The system* commands ***the robot*** to move to the table pick up position. g. ***The robot*** retracts both its arms and rotates until it reaches the table pick up position (position 2). h. *The system* commands ***the press*** to move to its load position. i. ***The press*** moves its movable plate upwards until in the top position, and then moves its movable plate downwards until it is in middle (load) position. j. *The system* stores a request that ***the press*** is ready to receive a plate until ***the robot*** is ready to act on it. k. *The system* stores a request that ***the deposit belt*** is ready to receive a plate until ***the robot*** is ready to act on it. l. *The system* commands ***the crane*** to move to its pickup position over the deposit belt. m. ***The crane*** *moves* towards ***the deposit belt*** and extends its arm until it is in the pickup position over the deposit belt. n. *The system* stores a request for ***the deposit belt*** to bring a plate to the end of the belt. The system stores this request until ***the deposit belt*** is ready to act on it.
Post-condition	▪ All the physical machines in the system are turned on and in their initial position and are ready to receive a plate. ▪ FeedBelt’s belt is not running. ▪ Table is in load position. ▪ Robot is in position 2, arms are retracted. ▪ Press is in load position. ▪ Crane is positioned over the deposit belt, arm extended to the deposit belt extension. ▪ No plates are in the system.

A.1.5 Use case description for ‘Produce forged plates’

Use Case	***Produce forged plates***
Actor	▪ Operator ▪ FeedBelt_HW ▪ Table_HW ▪ Robot_HW ▪ Press_HW ▪ DepositBelt_HW ▪ Crane_HW
Pre-condition	▪ System and all machines are turned on ▪ FeedBelt is not running. ▪ There is no plate at the end of the feed belt.
Description	*1. The operator adds a new plate to the feed belt.* *2. The system commands the feed belt to move the plate to the end of the belt.* *3. The feed belt moves the plate to the end of the belt.* *4. When the table is in load position, the system commands the feed belt to feed the plate onto the table.* *5. The feed table feeds the plate onto the table.* *6. The system commands the table to go to its unload position.* *7. The table rotates and elevates to its unload position.* *8. When the robot’s upper arm is unloaded and the robot is ready, the system commands the robot to pick up the plate from the table.* *9. The robot rotates until its upper arm points to the table, and then it extends its upper arm until it is positioned over the table, and activates its magnet to pick up the plate, before it retracts its upper arm to its retracted position.* *10. When the plate is picked from the table, the system commands the table to go to its load position.* *11. The table rotates and lowers itself to its load position.* *12. When the press is in load position and the robot is ready, the system commands the robot to load the plate to the press.* *13. The robot rotates until its upper arm points to the press, then it extends its upper arm until its magnet is positioned over the press, and deactivates its magnet to drop the plate, before it retracts its upper arm until it is in its retracted position.* *14. The system commands the press to forge the plate and to bring the movable plate to its unload (bottom) position.* *15. The press moves its movable plate upwards until it arrives to the top position, where the plate is forged, then it moves to its bottom (unload) position.* *16. When the robot’s lower arm is unloaded and the robot is ready, the*

	system commands the robot to pick up the plate from the press. 17. *The robot rotates until its lower arm points to the press, and then it extends its lower arm until its magnet is positioned over the press and activates its magnet to pick up the plate and then retracts its lower arm.* 18. *When the plate is picked up from the press, the system commands the press to go to its load position.* 19. *The press moves its movable plate to its load (middle) position.* 20. *When the deposit belt is ready to receive a new plate and the robot's lower arm is loaded and the robot is ready, the system commands the robot to drop the plate on the deposit belt.* 21. *The robot rotates so that its lower arm points to the deposit belt, then it extend its lower arm until the magnet is over the deposit belt. There it deactivates its magnet to drop the plate onto the deposit belt and retracts the lower arm to its retracted position.* 22. *When there is no plate at the end of the deposit belt, the system commands the deposit belt to move a new plate to the end of the belt.* 23. *The deposit belt brings the plate to the end of the belt.* 24. *When there is a new plate at the end, the system commands the crane to pick up the plate and bring the plate to the container.* 25. *When the crane is in its initial position over the deposit belt, it activates its magnet so that it picks up the plate and moves horizontally towards the container at the same time as it retracts its arm until it reaches its container position. It deactivates its magnet, so that the plate is dropped into the container and moves back to its initial position.*
Post-conditions	▪ The plate that was added is forged and placed into the container ▪ Other plates may be anywhere in the system


«requirementRelated»
produceForgedPlates
«refine»
«refine»
«refine»
«refine»
«safety requirement, requirement»
Restrict machine mobility
«safety requirement, requirement»
Avoid machine collisions
«safety requirement, requirement»
Avoid falling metal plates
«safety requirement, requirement»
Avoid piling or overlapping plates

Id = S1.0
Text = A machine should be stopped before the end of its possible movement, otherwise it would destroy itself.

Id = S2.0
Text = Collisions are possible between the press and the robot, and between the crane and the conveyor belts.

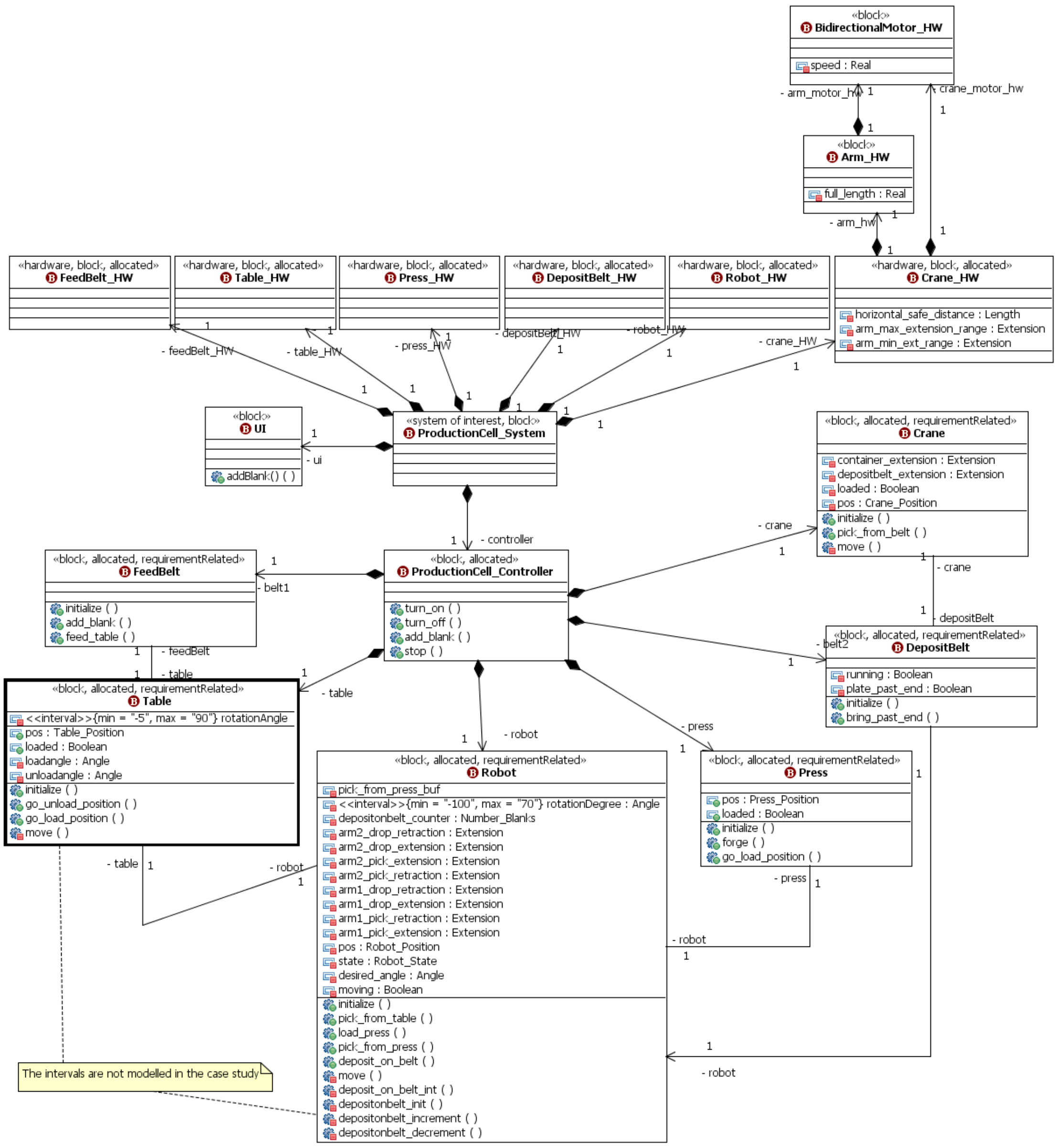
Id = S3.0
Text = Metal blanks can be dropped outside safe areas (belts, table, press) for two reasons -the electromagnets of the robot arms or of the crane are deactivated, - a belt transports a blank too far.

Id = S4.0
Text = Errors occur if blanks are piled on each other, overlapped, or too close for being distinguished by the photoelectric cell.

Figure 42 - System-level requirements refine the use case produceForgedPlates

A.2 Structure diagrams

A.2.1 Structure diagram with top-level blocks



«block» BidirectionalMotor_HW
speed : Real
- arm_motor_hw 1
crane_motor_hw 1
«block» Arm_HW
full_length : Real
- arm_hw 1
«hardware, block, allocated» FeedBelt_HW
«hardware, block, allocated» Table_HW
«hardware, block, allocated» Press_HW
«hardware, block, allocated» DepositBelt_HW
«hardware, block, allocated» Robot_HW
«hardware, block, allocated» Crane_HW
horizontal_safe_distance : Length
arm_max_extension_range : Extension
arm_min_ext_range : Extension
- feedBelt_HW
- table_HW
- press_HW
- depositBelt_HW
- robot_HW
- crane_HW
«block» UI
addBlank() ( )
- ui
«system of interest, block» ProductionCell_System
- controller
«block, allocated» ProductionCell_Controller
turn_on ( )
turn_off ( )
add_blank ( )
stop ( )
«block, allocated, requirementRelated» Crane
container_extension : Extension
depositbelt_extension : Extension
loaded : Boolean
pos : Crane_Position
initialize ( )
pick_from_belt ( )
move ( )
- crane
- depositBelt
«block, allocated, requirementRelated» FeedBelt
initialize ( )
add_blank ( )
feed_table ( )
- belt1
- feedBelt
- table
«block, allocated, requirementRelated» DepositBelt
running : Boolean
plate_past_end : Boolean
initialize ( )
bring_past_end ( )
- belt2
«block, allocated, requirementRelated» Table
<<interval>>{min = "-5", max = "90"} rotationAngle
pos : Table_Position
loaded : Boolean
loadangle : Angle
unloadangle : Angle
initialize ( )
go_unload_position ( )
go_load_position ( )
move ( )
- robot
- press
«block, allocated, requirementRelated» Robot
pick_from_press_buf
<<interval>>{min = "-100", max = "70"} rotationDegree : Angle
depositonbelt_counter : Number_Blanks
arm2_drop_retraction : Extension
arm2_drop_extension : Extension
arm2_pick_extension : Extension
arm2_pick_retraction : Extension
arm1_drop_retraction : Extension
arm1_drop_extension : Extension
arm1_pick_retraction : Extension
arm1_pick_extension : Extension
pos : Robot_Position
state : Robot_State
desired_angle : Angle
moving : Boolean
initialize ( )
pick_from_table ( )
load_press ( )
pick_from_press ( )
deposit_on_belt ( )
move ( )
deposit_on_belt_int ( )
depositonbelt_init ( )
depositonbelt_increment ( )
depositonbelt_decrement ( )
«block, allocated, requirementRelated» Press
pos : Press_Position
loaded : Boolean
initialize ( )
forge ( )
go_load_position ( )
The intervals are not modelled in the case study

Figure 43 - bdd [Package] ProductionCell_Structure [top-level blocks]. For simplified diagram, see Figure 45.

A.2.2 Full structure diagram

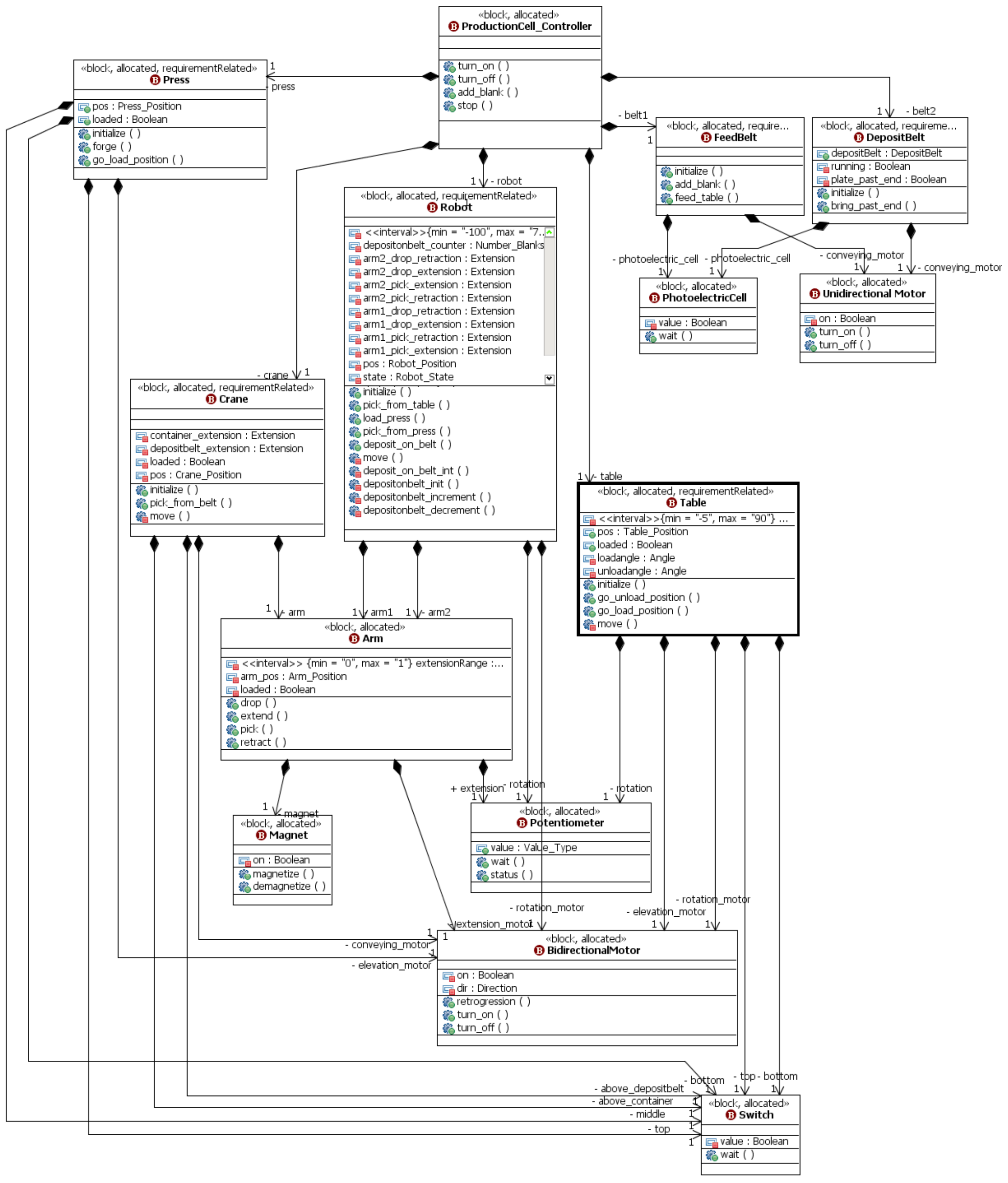


Figure 44 - bdd [Package] ProductionCell_Structure [ProductionCell_Controller and parts]. Full structure diagram of ProductionCell_Controller and its parts (without the superclasses Actuator/Sensor to avoid more cluttered diagram).

A.2.3 Simplified structure diagrams

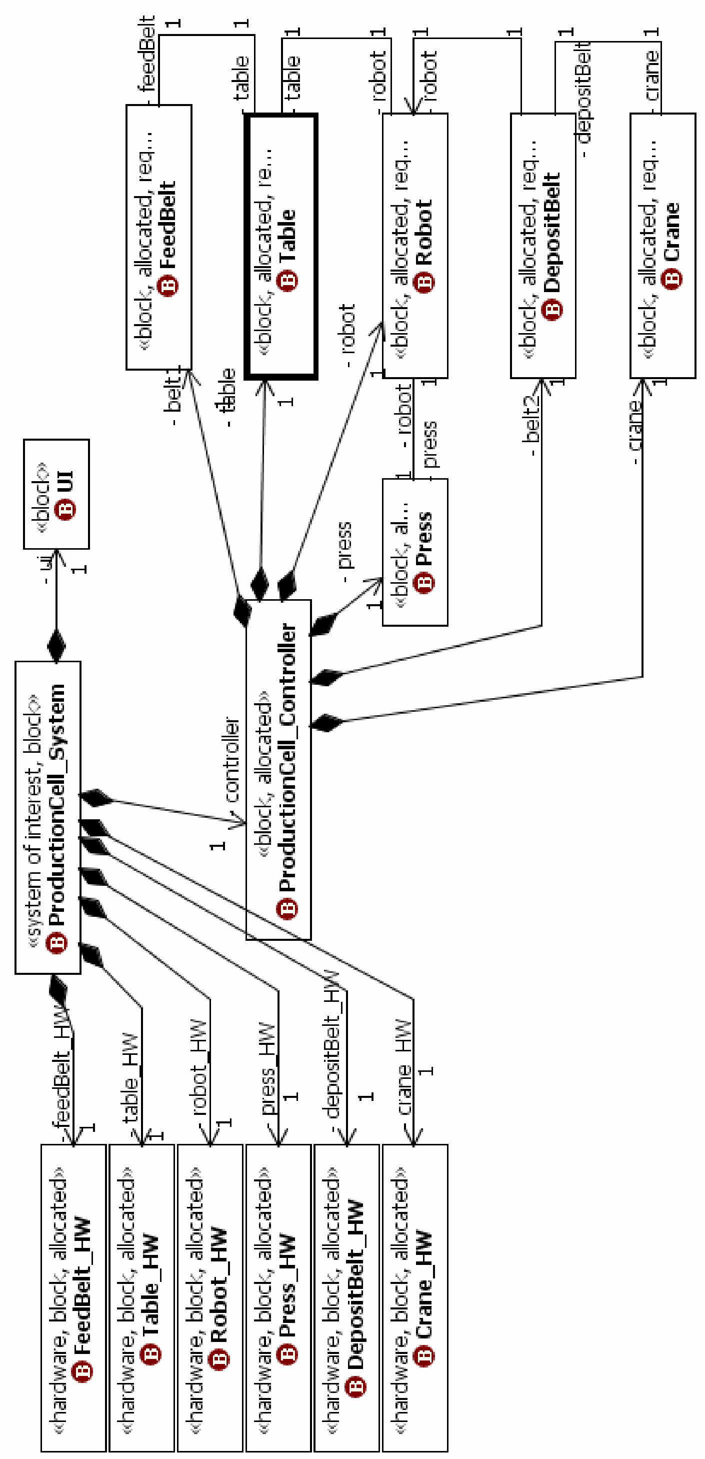
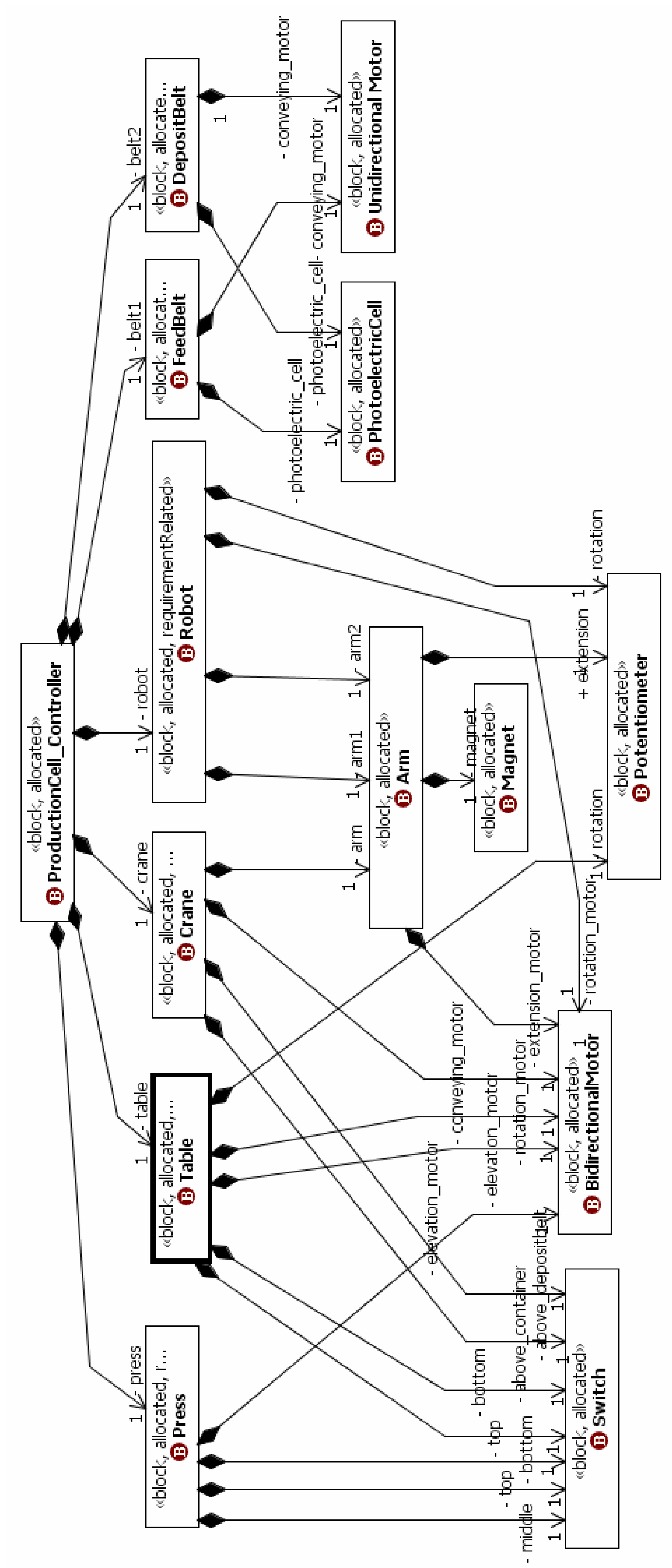

«block» UI
«system of interest, block» ProductionCell_System
«block, allocated» ProductionCell_Controller
«block, allocated, req... FeedBelt
«block, allocated, re... Table
«block, allocated, req... Robot
«block, al... Press
«block, allocated, req... DepositBelt
«block, allocated, req... Crane
«hardware, block, allocated» FeedBelt_HW
«hardware, block, allocated» Table_HW
«hardware, block, allocated» Robot_HW
«hardware, block, allocated» Press_HW
«hardware, block, allocated» DepositBelt_HW
«hardware, block, allocated» Crane_HW
- ui
- controller
- belt
- table
- robot
- press
- belt2
- crane
- feedBelt
- depositBelt
- feedBelt_HW
- table_HW
- robot_HW
- press_HW
- depositBelt_HW
- crane_HW
1

Figure 45 - Simplified structure diagram over the top-level blocks. Full diagram in Figure 43.



«block, allocated» ProductionCell_Controller
«block, allocate... DepositBelt
«block, allocat... FeedBelt
«block, allocated, requirementRelated» Robot
«block, allocated, ... Crane
«block, allocated,... Table
«block, allocated, r... Press
«block, allocated» Unidirectional Motor
«block, allocated» PhotoelectricCell
«block, allocated» Arm
«block, allocated» Magnet
«block, allocated» Potentiometer
«block, allocated» BidirectionalMotor
«block, allocated» Switch
- belt2
- belt1
- robot
- crane
- table
- press
- conveying_motor
- photoelectric_cell
- arm
- arm1
- arm2
- magnet
+ extension
- rotation
- rotation_motor
- extension_motor
- elevation_motor
- above_container
- above_depositbelt
- bottom
- top
- middle

Figure 46 - Simplified stucture diagram over the system of interest and lower level blocks. Full diagram in Figure 44.

A.2.4 Internal Block diagrams

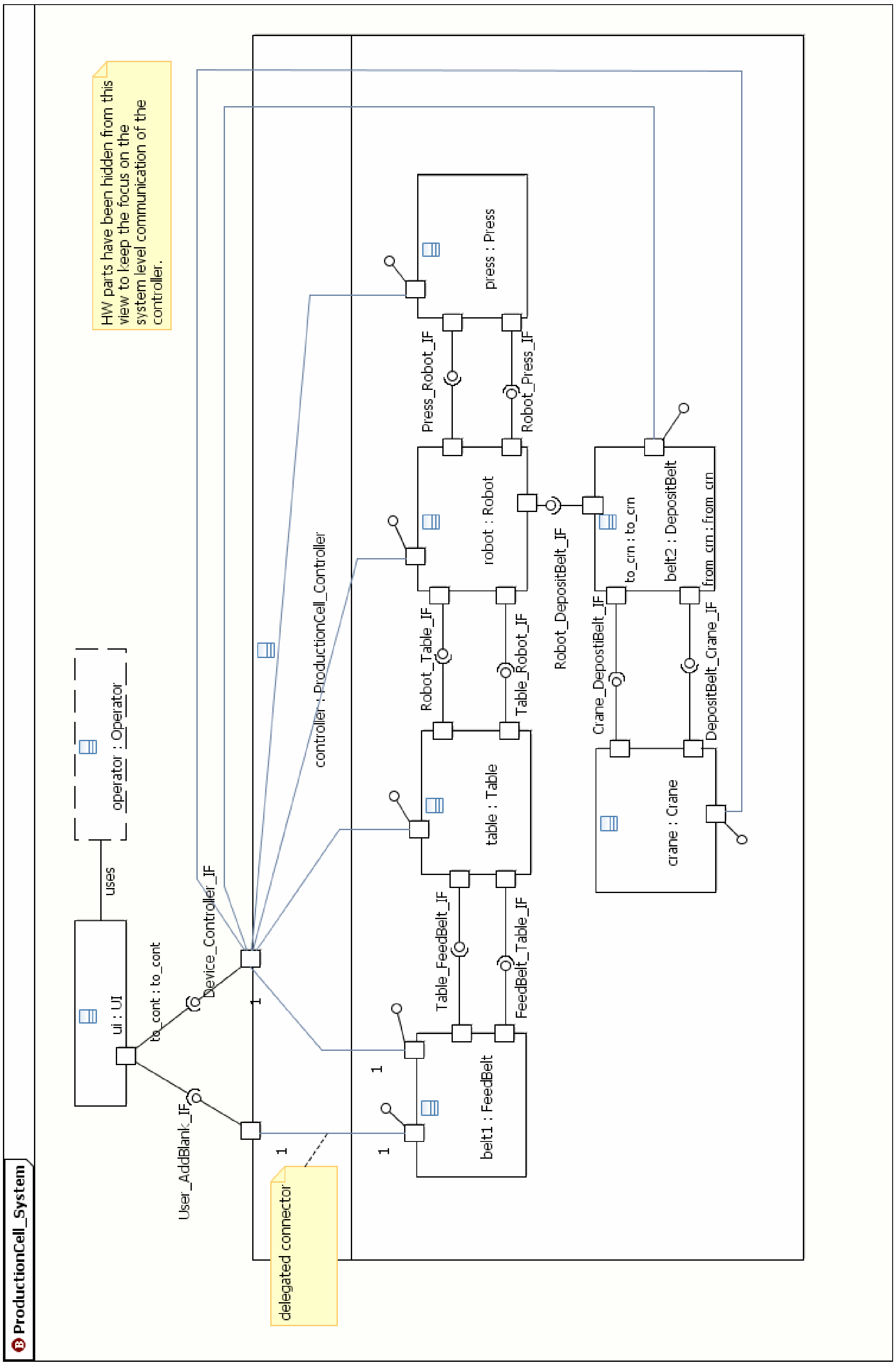


Figure 47 – ibd [Block] ProductionCell_System [Ports and interfaces between top-level blocks]. Interfaces shown in Figure 52.

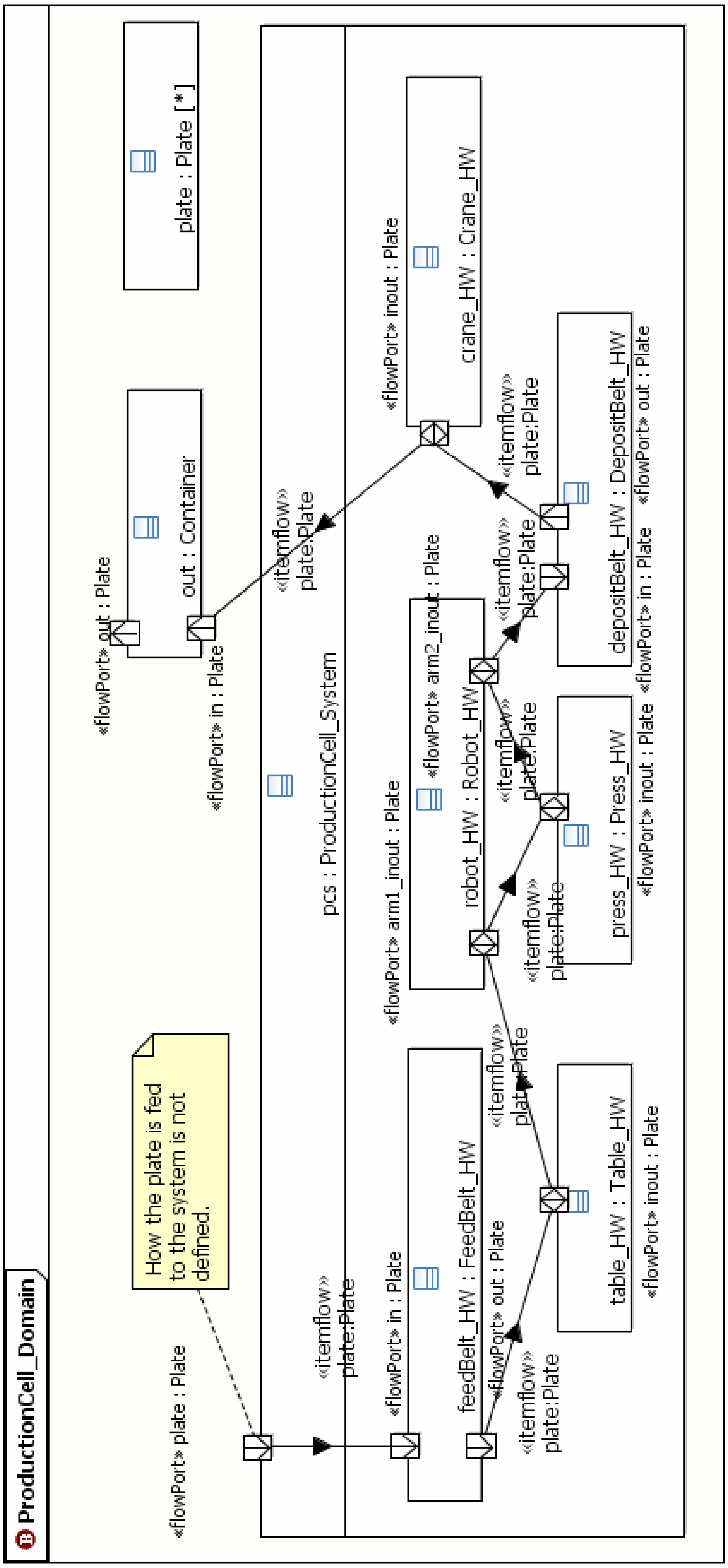


Figure 48 – ibd [Block] ProductionCell_Domain [Item Flow - Plate]

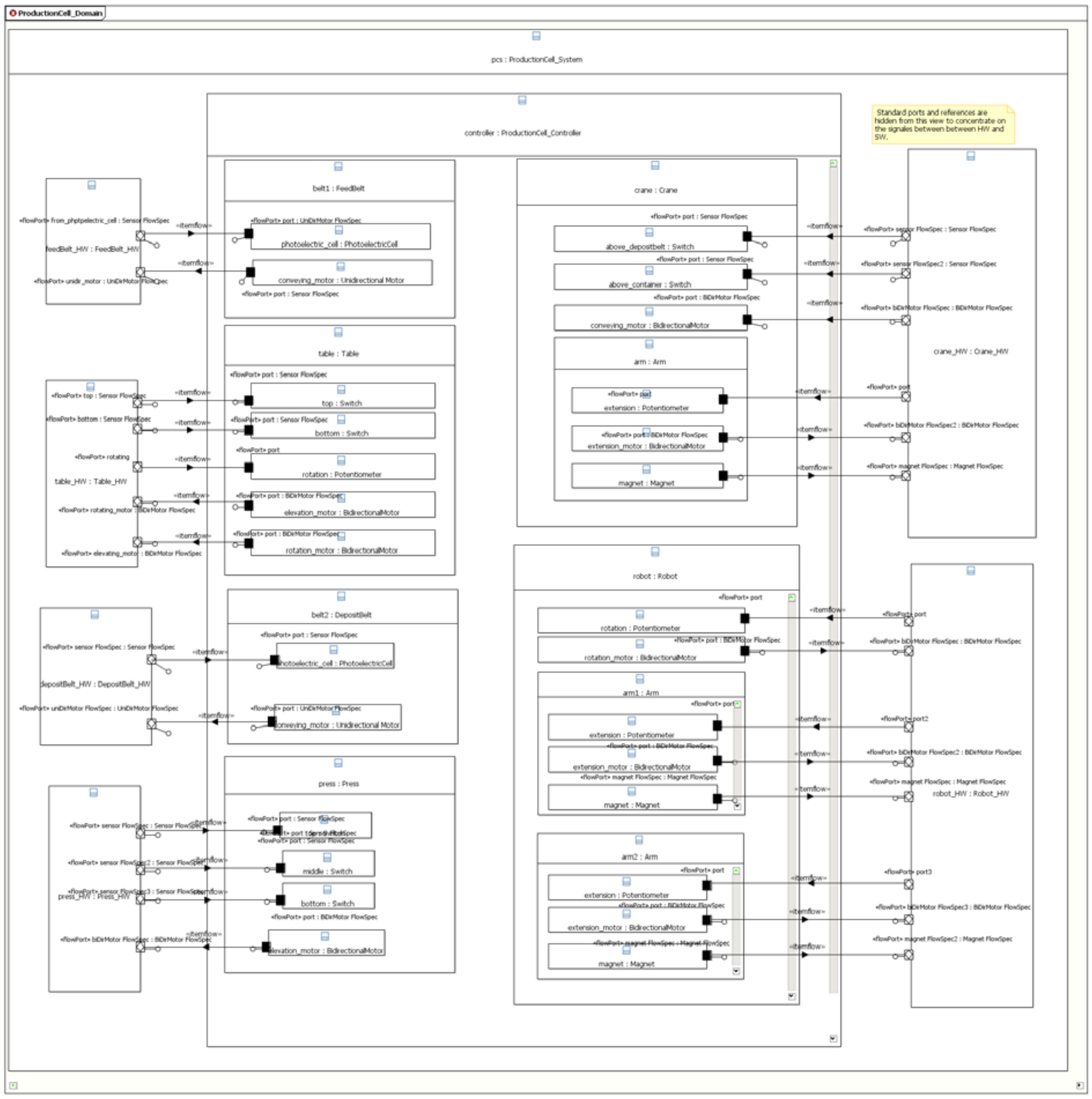

ProductionCell_Domain
pcs : ProductionCell_System
controller : ProductionCell_Controller
Standard ports and references are hidden from this view to concentrate on the signales between between HW and SW.
belt1 : FeedBelt
photoelectric_cell : PhotoelectricCell
conveying_motor : Unidirectional Motor
feedBelt_HW : FeedBelt_HW
table : Table
top : Switch
bottom : Switch
rotation : Potentiometer
elevation_motor : BidirectionalMotor
rotation_motor : BidirectionalMotor
table_HW : Table_HW
belt2 : DepositBelt
depositBelt_HW : DepositBelt_HW
press : Press
middle : Switch
bottom : Switch
elevation_motor : BidirectionalMotor
press_HW : Press_HW
crane : Crane
above_depositbelt : Switch
above_container : Switch
conveying_motor : BidirectionalMotor
arm : Arm
extension : Potentiometer
extension_motor : BidirectionalMotor
magnet : Magnet
crane_HW : Crane_HW
robot : Robot
rotation : Potentiometer
rotation_motor : BidirectionalMotor
arm1 : Arm
arm2 : Arm
robot_HW : Robot_HW
«itemflow»
«flowPort» port : Sensor FlowSpec
«flowPort» port : BiDirMotor FlowSpec
«flowPort» port : UniDirMotor FlowSpec
«flowPort» magnet FlowSpec : Magnet FlowSpec

Figure 49 - ibd [Block] ProductionCell_Domain [HW/SW Communiaction], see close ups in Figure 50 and Figure 51.

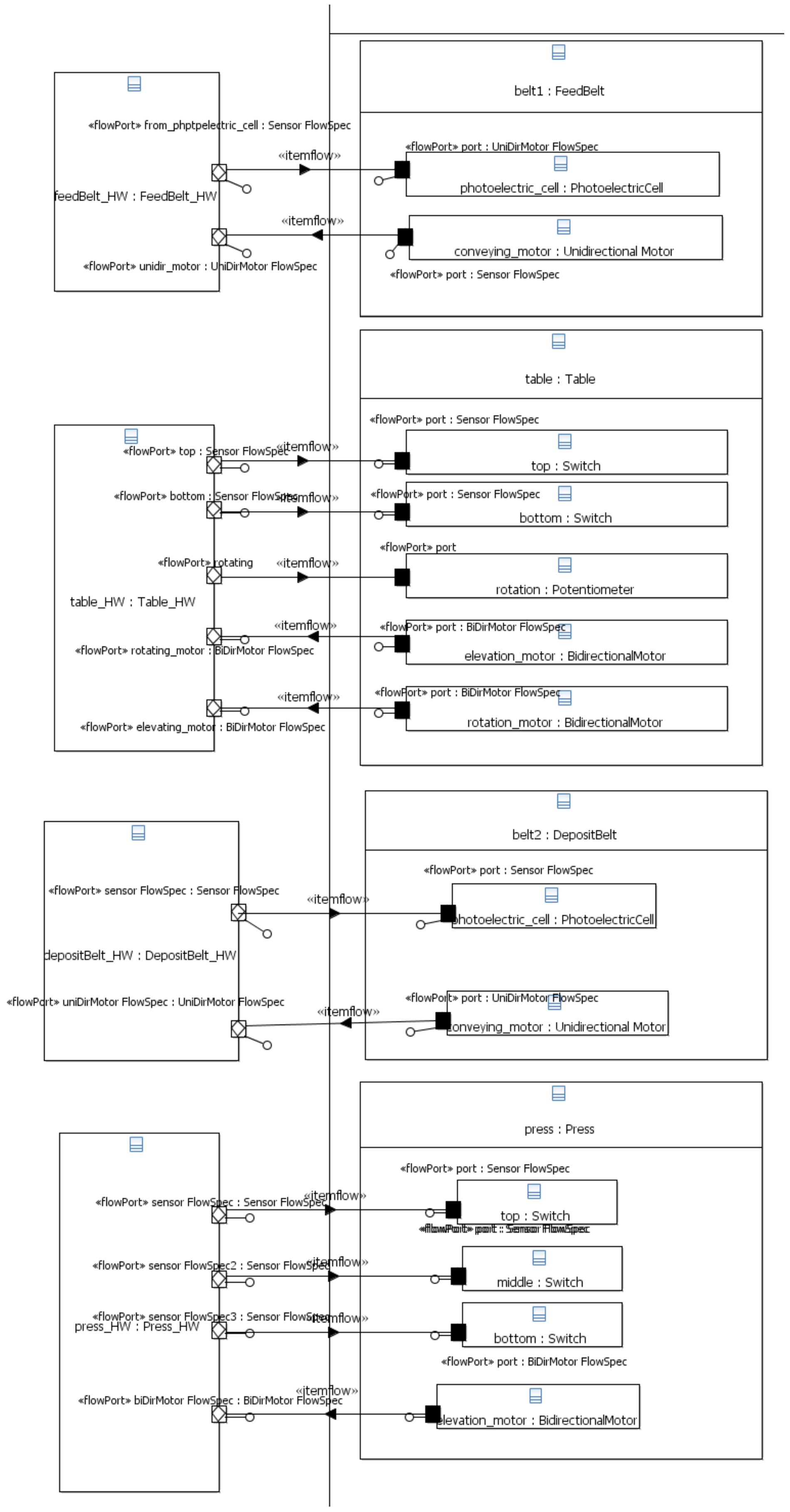

belt1 : FeedBelt
«flowPort» from_phptpelectric_cell : Sensor FlowSpec
«itemflow»
«flowPort» port : UniDirMotor FlowSpec
photoelectric_cell : PhotoelectricCell
feedBelt_HW : FeedBelt_HW
«itemflow»
conveying_motor : Unidirectional Motor
«flowPort» unidir_motor : UniDirMotor FlowSpec
«flowPort» port : Sensor FlowSpec
table : Table
«flowPort» port : Sensor FlowSpec
«flowPort» top : Sensor FlowSpec
«itemflow»
top : Switch
«flowPort» bottom : Sensor FlowSpec
«itemflow»
«flowPort» port : Sensor FlowSpec
bottom : Switch
«flowPort» port
«flowPort» rotating
«itemflow»
rotation : Potentiometer
table_HW : Table_HW
«itemflow»
«flowPort» port : BiDirMotor FlowSpec
«flowPort» rotating_motor : BiDirMotor FlowSpec
elevation_motor : BidirectionalMotor
«itemflow»
«flowPort» port : BiDirMotor FlowSpec
rotation_motor : BidirectionalMotor
«flowPort» elevating_motor : BiDirMotor FlowSpec
belt2 : DepositBelt
«flowPort» port : Sensor FlowSpec
«flowPort» sensor FlowSpec : Sensor FlowSpec
«itemflow»
photoelectric_cell : PhotoelectricCell
depositBelt_HW : DepositBelt_HW
«flowPort» uniDirMotor FlowSpec : UniDirMotor FlowSpec
«flowPort» port : UniDirMotor FlowSpec
«itemflow»
conveying_motor : Unidirectional Motor
press : Press
«flowPort» port : Sensor FlowSpec
«flowPort» sensor FlowSpec : Sensor FlowSpec
«itemflow»
top : Switch
«flowPort» port : Sensor FlowSpec
«flowPort» sensor FlowSpec2 : Sensor FlowSpec
«itemflow»
middle : Switch
«flowPort» sensor FlowSpec3 : Sensor FlowSpec
press_HW : Press_HW
«itemflow»
bottom : Switch
«flowPort» port : BiDirMotor FlowSpec
«itemflow»
«flowPort» biDirMotor FlowSpec : BiDirMotor FlowSpec
elevation_motor : BidirectionalMotor

Figure 50 - Part of the ibd [Block] ProductionCell_Domain [HW/SW Communiaction]. Whole diagram in Figure 49.

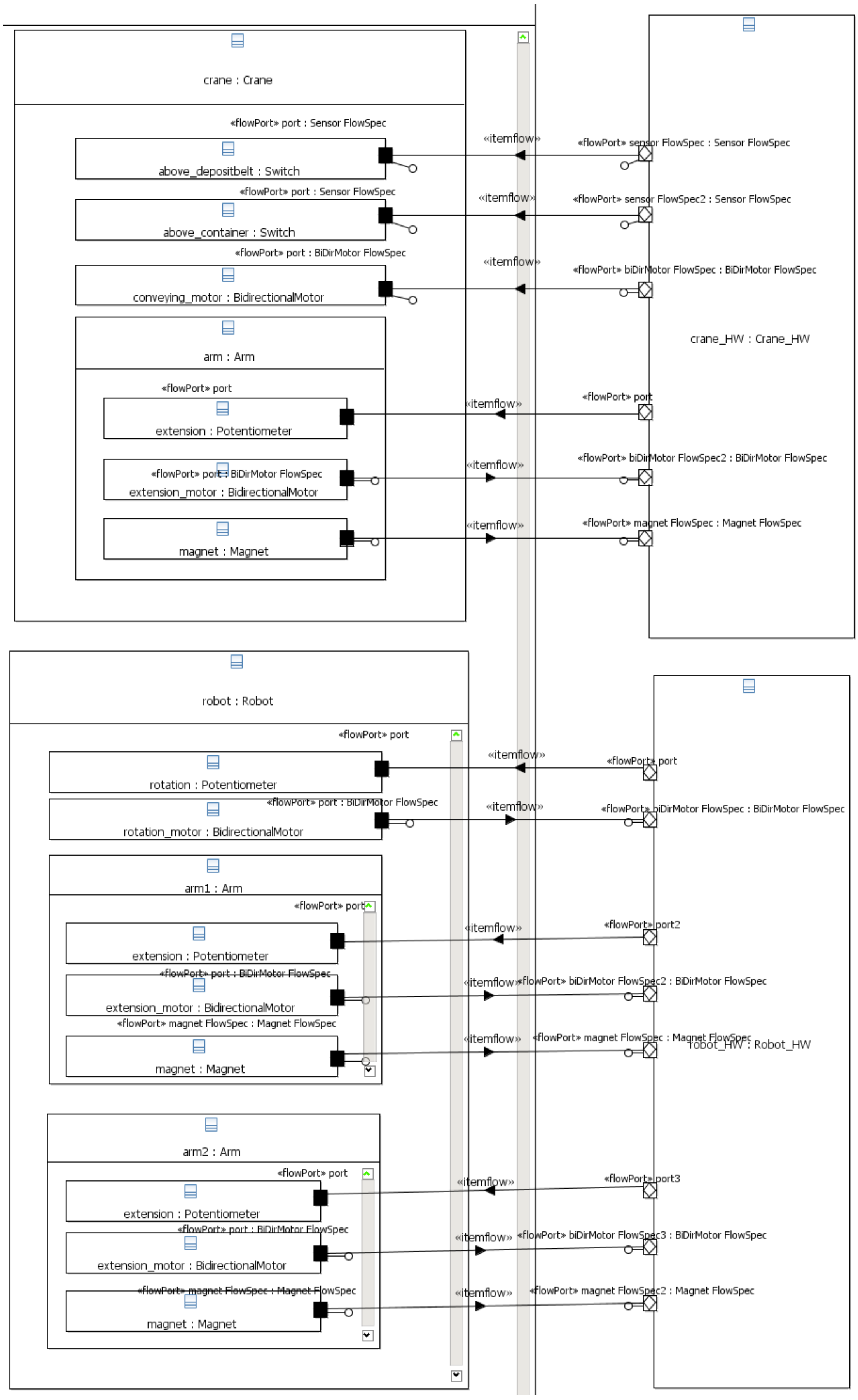

crane : Crane
«flowPort» port : Sensor FlowSpec
above_depositbelt : Switch
«flowPort» port : Sensor FlowSpec
above_container : Switch
«flowPort» port : BiDirMotor FlowSpec
conveying_motor : BidirectionalMotor
arm : Arm
«flowPort» port
extension : Potentiometer
«flowPort» port : BiDirMotor FlowSpec
extension_motor : BidirectionalMotor
magnet : Magnet
«itemflow»
«flowPort» sensor FlowSpec : Sensor FlowSpec
«flowPort» sensor FlowSpec2 : Sensor FlowSpec
«flowPort» biDirMotor FlowSpec : BiDirMotor FlowSpec
crane_HW : Crane_HW
«flowPort» port
«flowPort» biDirMotor FlowSpec2 : BiDirMotor FlowSpec
«flowPort» magnet FlowSpec : Magnet FlowSpec
robot : Robot
«flowPort» port
rotation : Potentiometer
«flowPort» port : BiDirMotor FlowSpec
rotation_motor : BidirectionalMotor
arm1 : Arm
«flowPort» port
extension : Potentiometer
«flowPort» port : BiDirMotor FlowSpec
extension_motor : BidirectionalMotor
«flowPort» magnet FlowSpec : Magnet FlowSpec
magnet : Magnet
arm2 : Arm
«flowPort» port
extension : Potentiometer
«flowPort» port : BiDirMotor FlowSpec
extension_motor : BidirectionalMotor
«flowPort» magnet FlowSpec : Magnet FlowSpec
magnet : Magnet
«flowPort» port
«flowPort» biDirMotor FlowSpec : BiDirMotor FlowSpec
«flowPort» port2
«flowPort» biDirMotor FlowSpec2 : BiDirMotor FlowSpec
«flowPort» magnet FlowSpec : Magnet FlowSpec
robot_HW : Robot_HW
«flowPort» port3
«flowPort» biDirMotor FlowSpec3 : BiDirMotor FlowSpec
«flowPort» magnet FlowSpec2 : Magnet FlowSpec

Figure 51 - Part of ibd [Block] ProductionCell_Domain [HW/SW Communiaction]. Whole diagram in Figure 49.

A.2.5 Interfaces for top-level blocks

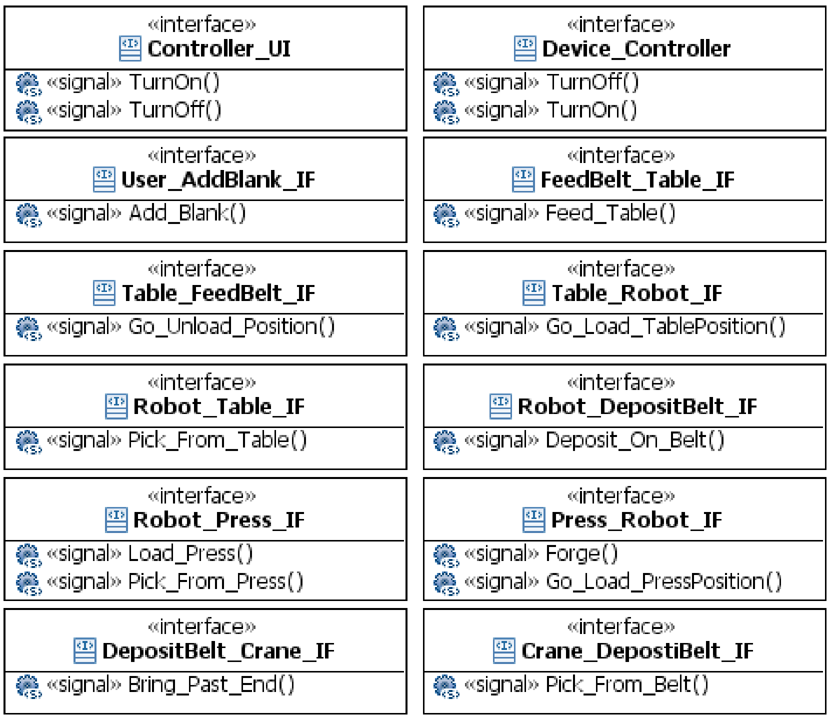

«interface» Controller_UI
«signal» TurnOn()
«signal» TurnOff()
«interface» Device_Controller
«signal» TurnOff()
«signal» TurnOn()
«interface» User_AddBlank_IF
«signal» Add_Blank()
«interface» FeedBelt_Table_IF
«signal» Feed_Table()
«interface» Table_FeedBelt_IF
«signal» Go_Unload_Position()
«interface» Table_Robot_IF
«signal» Go_Load_TablePosition()
«interface» Robot_Table_IF
«signal» Pick_From_Table()
«interface» Robot_DepositBelt_IF
«signal» Deposit_On_Belt()
«interface» Robot_Press_IF
«signal» Load_Press()
«signal» Pick_From_Press()
«interface» Press_Robot_IF
«signal» Forge()
«signal» Go_Load_PressPosition()
«interface» DepositBelt_Crane_IF
«signal» Bring_Past_End()
«interface» Crane_DepostiBelt_IF
«signal» Pick_From_Belt()

Figure 52 - Interfaces used on the standard ports of the top-level blocks. Show signal receptions of the signals in Figure 53.

A.2.6 Signals for top-level blocks

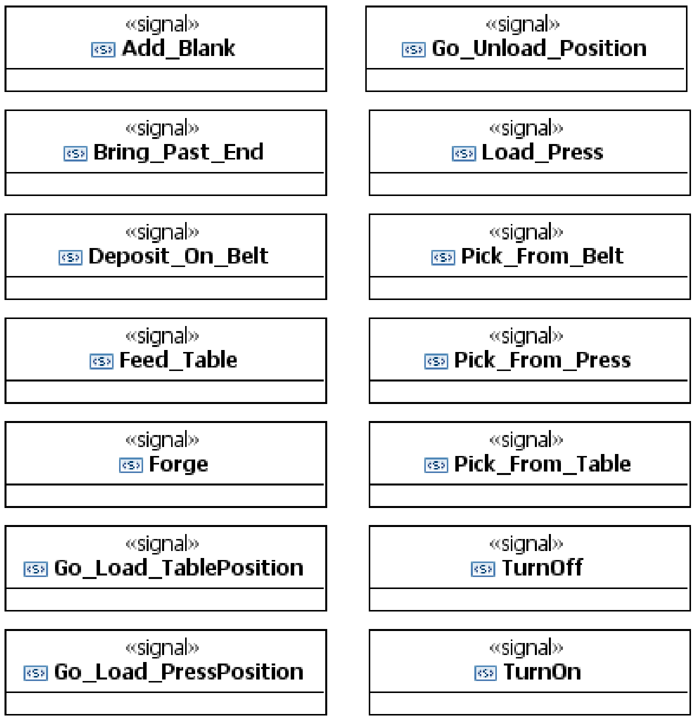
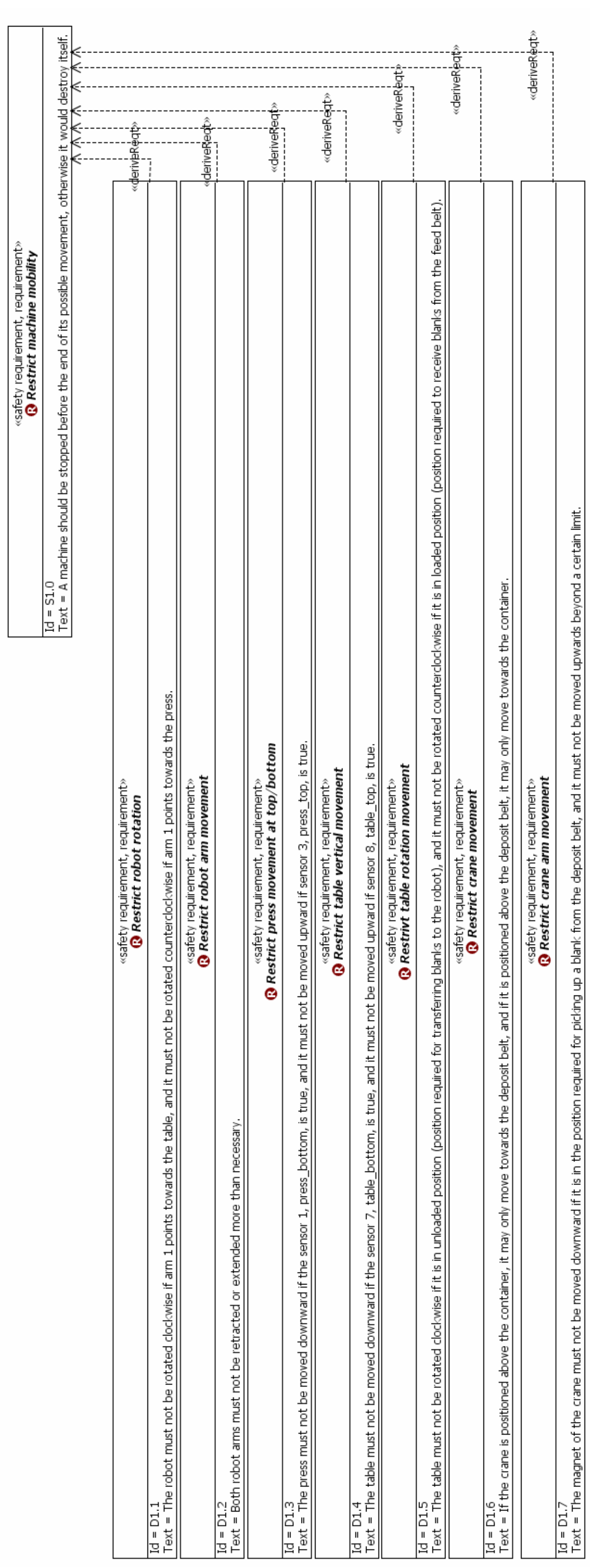


Figure 53 - High Level Signals, which are shown as receptions in the interfaces in Figure 52.

A.3 Derived block-level requirement diagrams

A.3.1 Req S1.0



«safety requirement, requirement»
Restrict machine mobility
Id = S1.0
Text = A machine should be stopped before the end of its possible movement, otherwise it would destroy itself.

«safety requirement, requirement»
Restrict robot rotation
Id = D1.1
Text = The robot must not be rotated clockwise if arm 1 points towards the table, and it must not be rotated counterclockwise if arm 1 points towards the press.

«safety requirement, requirement»
Restrict robot arm movement
Id = D1.2
Text = Both robot arms must not be retracted or extended more than necessary.

«safety requirement, requirement»
Restrict press movement at top/bottom
Id = D1.3
Text = The press must not be moved downward if the sensor 1, press_bottom, is true, and it must not be moved upward if sensor 3, press_top, is true.

«safety requirement, requirement»
Restrict table vertical movement
Id = D1.4
Text = The table must not be moved downward if the sensor 7, table_bottom, is true, and it must not be moved upward if sensor 8, table_top, is true.

«safety requirement, requirement»
Restrivt table rotation movement
Id = D1.5
Text = The table must not be rotated clockwise if it is in unloaded position (position required for transferring blanks to the robot), and it must not be rotated counterclockwise if it is in loaded position (position required to receive blanks from the feed belt).

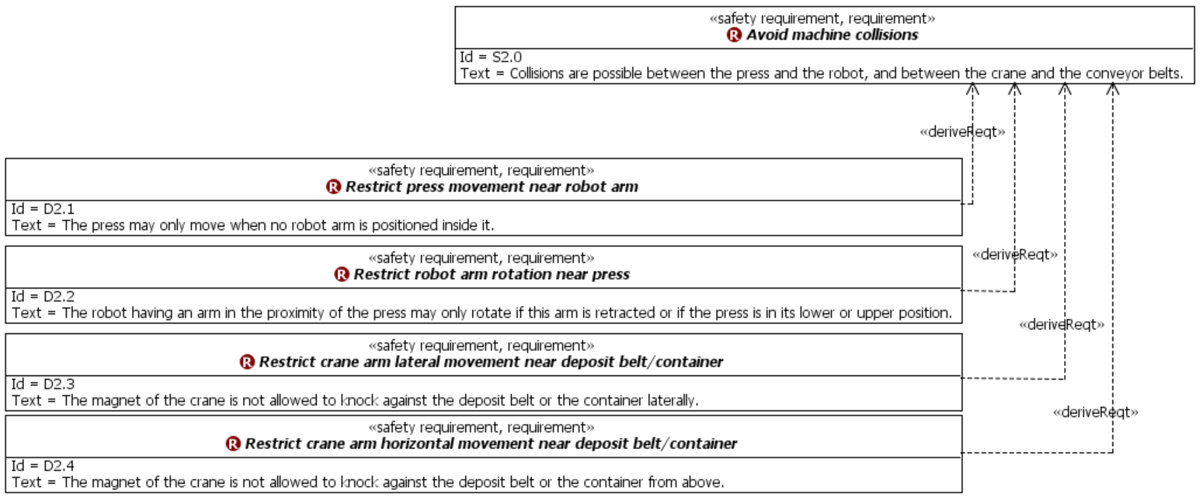
«safety requirement, requirement»
Restrict crane movement
Id = D1.6
Text = If the crane is positioned above the container, it may only move towards the deposit belt, and if it is positioned above the deposit belt, it may only move towards the container.

«safety requirement, requirement»
Restrict crane arm movement
Id = D1.7
Text = The magnet of the crane must not be moved downward if it is in the position required for picking up a blank from the deposit belt, and it must not be moved upwards beyond a certain limit.

Figure 54 - req [Package] Requirements [req s1.0 derived]

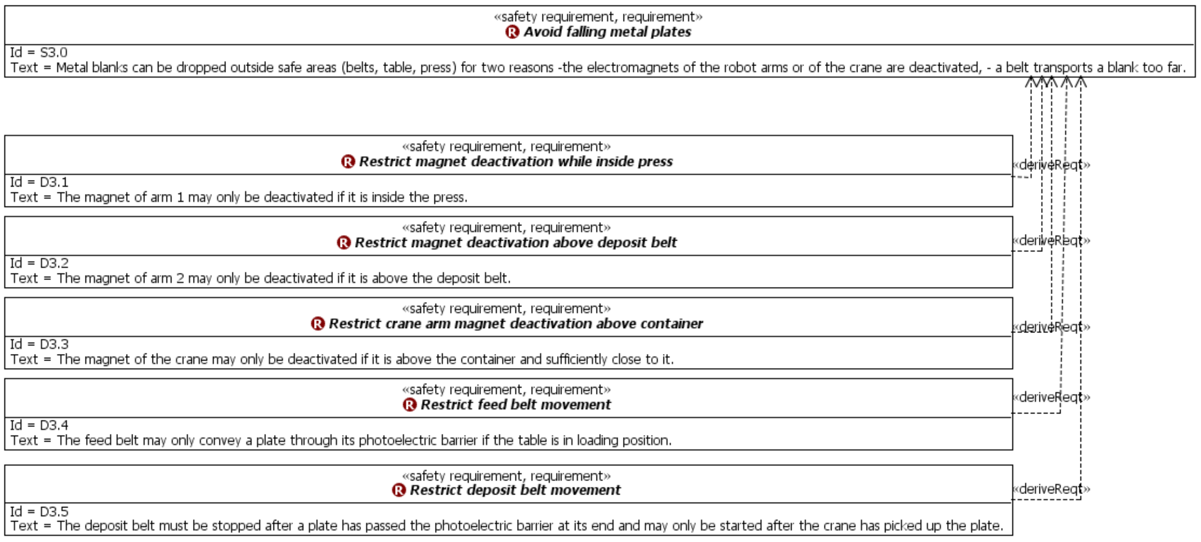
S1.0	**A machine should be stopped before the end of its possible movement, otherwise it would destroy itself.**
D1.1	The robot must not be rotated clockwise if arm 1 points towards the table, and it must not be rotated counterclockwise if arm 1 points towards the press.
D1.2	Both robot arms must not be retracted or extended more than necessary.
D1.3	The press must not be moved downward if the sensor 1, press_bottom, is true, and it must not be moved upward if sensor 3, press_top, is true.
D1.4	The table must not be moved downward if the sensor 7, table_bottom, is true, and it must not be moved upward if sensor 8, table_top, is true.
D1.5	The table must not be rotated clockwise if it is in unloaded position (position required for transferring plates to the robot), and it must not be rotated counterclockwise if it is in loaded position (position required to receive plates from the feed belt).
D1.6	If the crane is positioned above the container, it may only move towards the deposit belt, and if it is positioned above the deposit belt, it may only move towards the container.
D1.7	The magnet of the crane must not be moved downward if it is in the position required for picking up a plate from the deposit belt, and it must not be moved upwards beyond a certain limit.

A.3.2 Req S2.0


«safety requirement, requirement»
Avoid machine collisions
Id = S2.0
Text = Collisions are possible between the press and the robot, and between the crane and the conveyor belts.
«deriveReqt»
«safety requirement, requirement»
Restrict press movement near robot arm
Id = D2.1
Text = The press may only move when no robot arm is positioned inside it.
«deriveReqt»
«safety requirement, requirement»
Restrict robot arm rotation near press
Id = D2.2
Text = The robot having an arm in the proximity of the press may only rotate if this arm is retracted or if the press is in its lower or upper position.
«deriveReqt»
«safety requirement, requirement»
Restrict crane arm lateral movement near deposit belt/container
Id = D2.3
Text = The magnet of the crane is not allowed to knock against the deposit belt or the container laterally.
«deriveReqt»
«safety requirement, requirement»
Restrict crane arm horizontal movement near deposit belt/container
Id = D2.4
Text = The magnet of the crane is not allowed to knock against the deposit belt or the container from above.

S2.0	**Collisions are possible between the press and the robot, and between the crane and the conveyor belts.**
D2.1	The press may only move when no robot arm is positioned inside it.
D2.2	The robot having an arm in the proximity of the press may only rotate if this arm is retracted or if the press is in its lower or upper position.
D2.3	The magnet of the crane is not allowed to knock against the deposit belt or the container laterally.
D2.4	The magnet of the crane is not allowed to knock against the deposit belt or the container from above.

A.3.3 Req S3.0


«safety requirement, requirement»
Avoid falling metal plates
Id = S3.0
Text = Metal blanks can be dropped outside safe areas (belts, table, press) for two reasons -the electromagnets of the robot arms or of the crane are deactivated, - a belt transports a blank too far.

«safety requirement, requirement»
Restrict magnet deactivation while inside press
Id = D3.1
Text = The magnet of arm 1 may only be deactivated if it is inside the press.

«safety requirement, requirement»
Restrict magnet deactivation above deposit belt
Id = D3.2
Text = The magnet of arm 2 may only be deactivated if it is above the deposit belt.

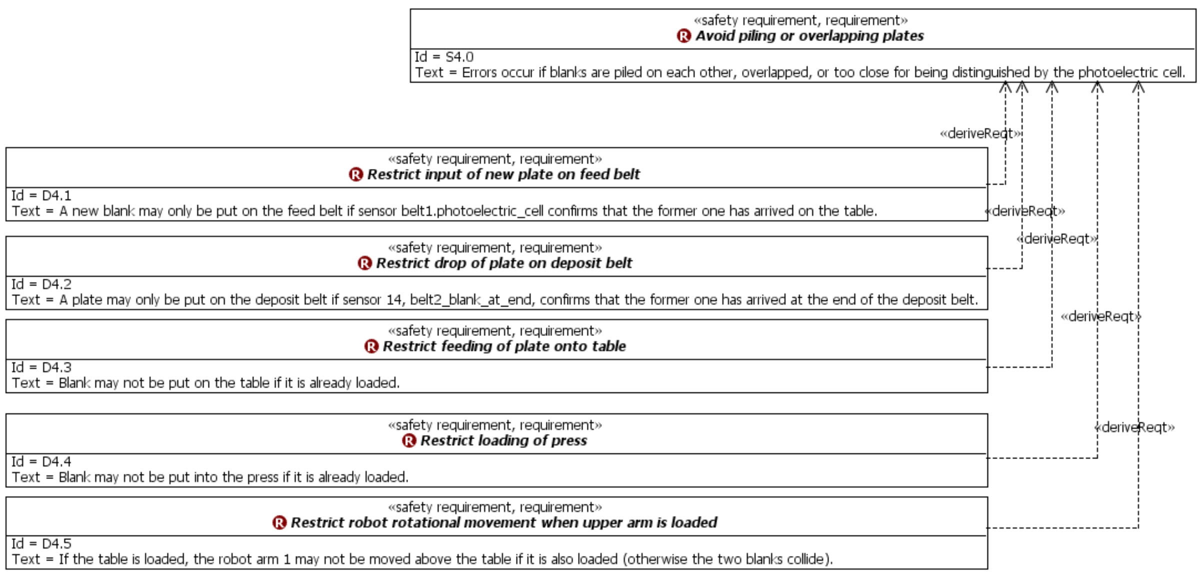
«safety requirement, requirement»
Restrict crane arm magnet deactivation above container
Id = D3.3
Text = The magnet of the crane may only be deactivated if it is above the container and sufficiently close to it.

«safety requirement, requirement»
Restrict feed belt movement
Id = D3.4
Text = The feed belt may only convey a plate through its photoelectric barrier if the table is in loading position.

«safety requirement, requirement»
Restrict deposit belt movement
Id = D3.5
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.

S3.0	**Metal blanks can be dropped outside safe areas (belts, table, press) for two reasons -the electromagnets of the robot arms or of the crane are deactivated, - a belt transports a blank too far.**
D3.1	The magnet of arm 1 may only be deactivated if it is inside the press.
D3.2	The magnet of arm 2 may only be deactivated if it is above the deposit belt.
D3.3	The magnet of the crane may only be deactivated if it is above the container and sufficiently close to it.
D3.4	The feed belt may only convey a plate through its photoelectric barrier if the table is in loading position.
D3.5	The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.

A.3.4 Req S4.0


«safety requirement, requirement»
Avoid piling or overlapping plates
Id = S4.0
Text = Errors occur if blanks are piled on each other, overlapped, or too close for being distinguished by the photoelectric cell.

«safety requirement, requirement»
Restrict input of new plate on feed belt
Id = D4.1
Text = A new blank may only be put on the feed belt if sensor belt1.photoelectric_cell confirms that the former one has arrived on the table.

«safety requirement, requirement»
Restrict drop of plate on deposit belt
Id = D4.2
Text = A plate may only be put on the deposit belt if sensor 14, belt2_blank_at_end, confirms that the former one has arrived at the end of the deposit belt.

«safety requirement, requirement»
Restrict feeding of plate onto table
Id = D4.3
Text = Blank may not be put on the table if it is already loaded.

«safety requirement, requirement»
Restrict loading of press
Id = D4.4
Text = Blank may not be put into the press if it is already loaded.

«safety requirement, requirement»
Restrict robot rotational movement when upper arm is loaded
Id = D4.5
Text = If the table is loaded, the robot arm 1 may not be moved above the table if it is also loaded (otherwise the two blanks collide).

S4.0	**Errors occur if blanks are piled on each other, overlapped, or too close for being distinguished by the photoelectric cell.**
D4.1	A new blank may only be put on the feed belt if sensor belt1.photoelectric_cell confirms that the former one has arrived on the table.
D4.2	A plate may only be put on the deposit belt if sensor 14, belt2_blank_at_end, confirms that the former one has arrived at the end of the deposit belt.
D4.3	Blank may not be put on the table if it is already loaded.
D4.4	Blank may not be put into the press if it is already loaded.
D4.5	If the table is loaded, the robot arm 1 may not be moved above the table if it is also loaded (otherwise the two blanks collide).

A.4 ValueType diagram

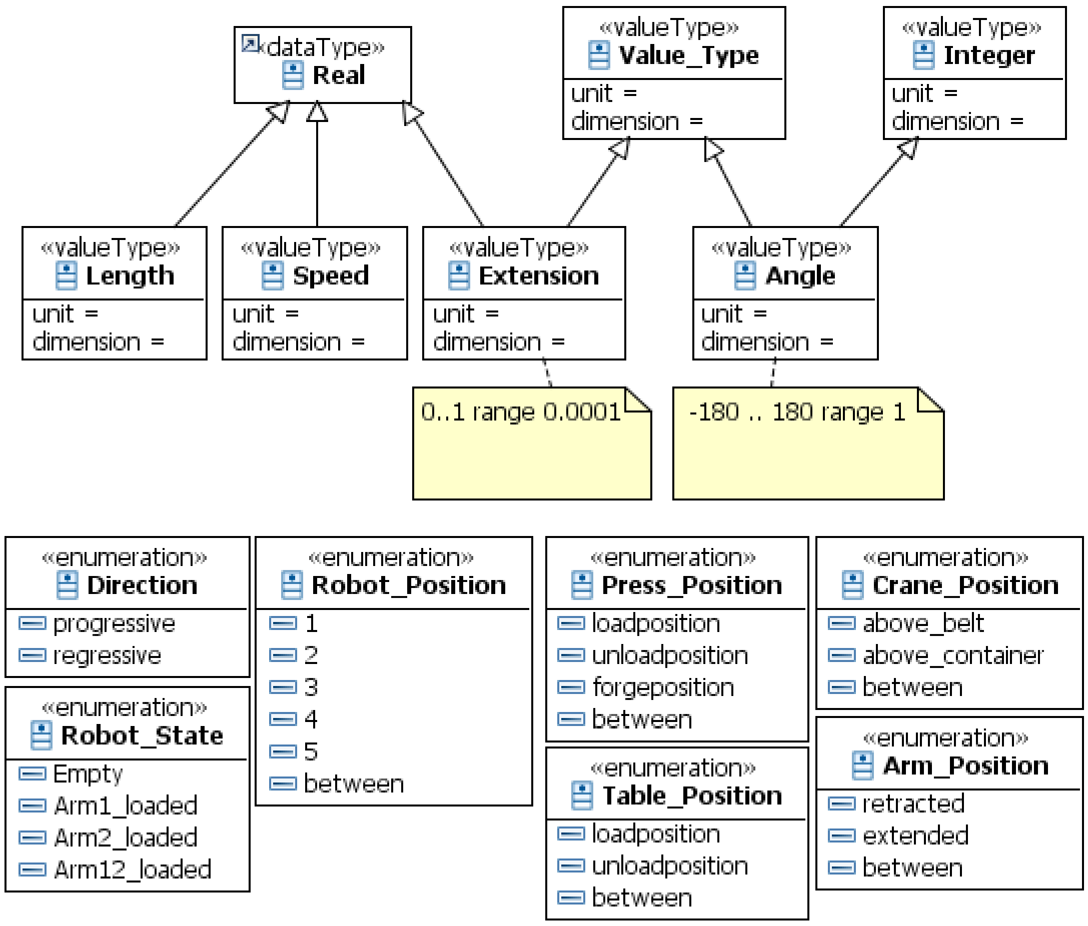


Figure 55 - bdd [Package] ValueTypes

A.5 Sequence diagrams

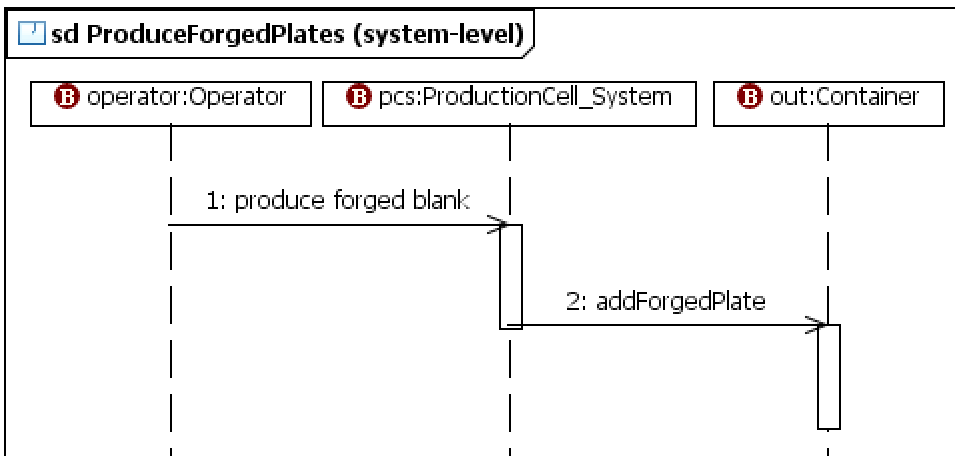


Figure 56 - sd produceForgedPlates [system-level interaction]

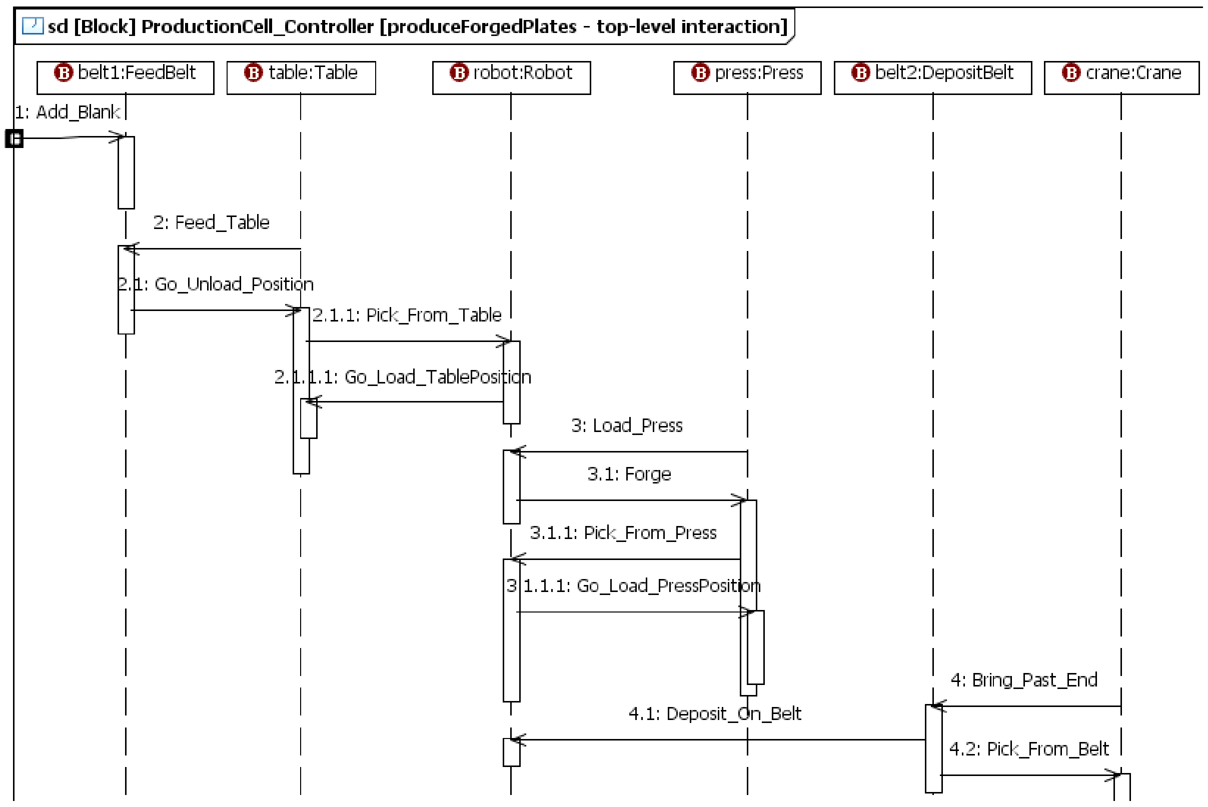


Figure 57 – sd [Block] ProductionCell_Controller [produceForgedPlates – top-level interaction]

This diagram only shows the interaction between the top-level blocks.

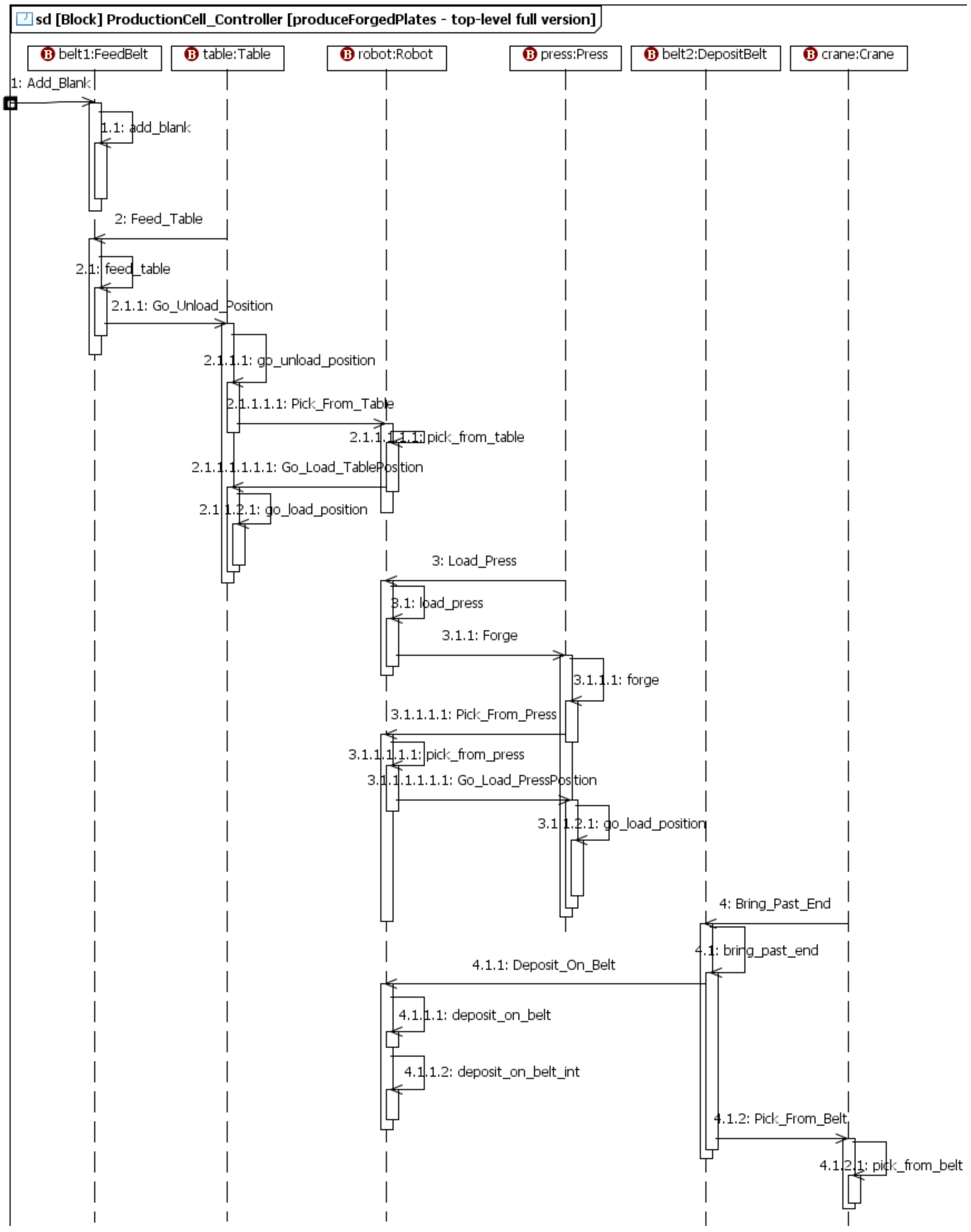

sd [Block] ProductionCell_Controller [produceForgedPlates - top-level full version]
belt1:FeedBelt
table:Table
robot:Robot
press:Press
belt2:DepositBelt
crane:Crane
1: Add_Blank
1.1: add_blank
2: Feed_Table
2.1: feed_table
2.1.1: Go_Unload_Position
2.1.1.1: go_unload_position
2.1.1.1.1: Pick_From_Table
2.1.1.1.1.1: pick_from_table
2.1.1.1.1.1.1: Go_Load_TablePosition
2.1.1.2.1: go_load_position
3: Load_Press
3.1: load_press
3.1.1: Forge
3.1.1.1: forge
3.1.1.1.1: Pick_From_Press
3.1.1.1.1.1: pick_from_press
3.1.1.1.1.1.1: Go_Load_PressPosition
3.1.1.2.1: go_load_position
4: Bring_Past_End
4.1: bring_past_end
4.1.1: Deposit_On_Belt
4.1.1.1: deposit_on_belt
4.1.1.2: deposit_on_belt_int
4.1.2: Pick_From_Belt
4.1.2.1: pick_from_belt

Figure 58 - sd [Block] ProductionCell_Controller [produceForgedPlates – top-level full version]

A.6 Activity diagrams

A.6.1 Block definition diagram

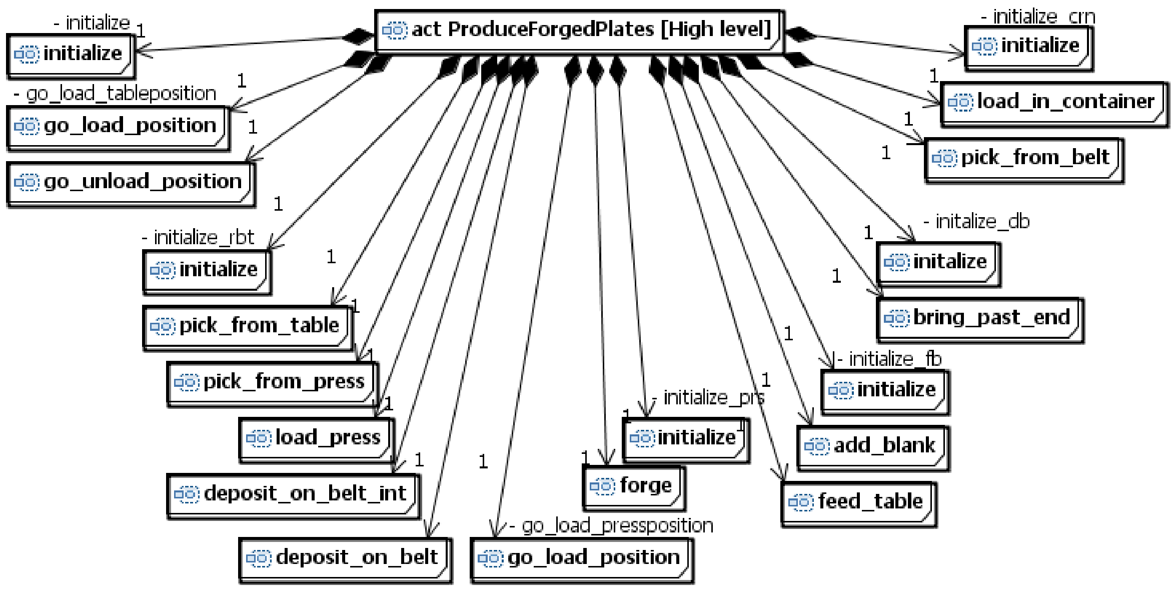


Figure 59 - bdd [Package] Activity [Composition of act produceForgedPlates]

A.6.2 High level

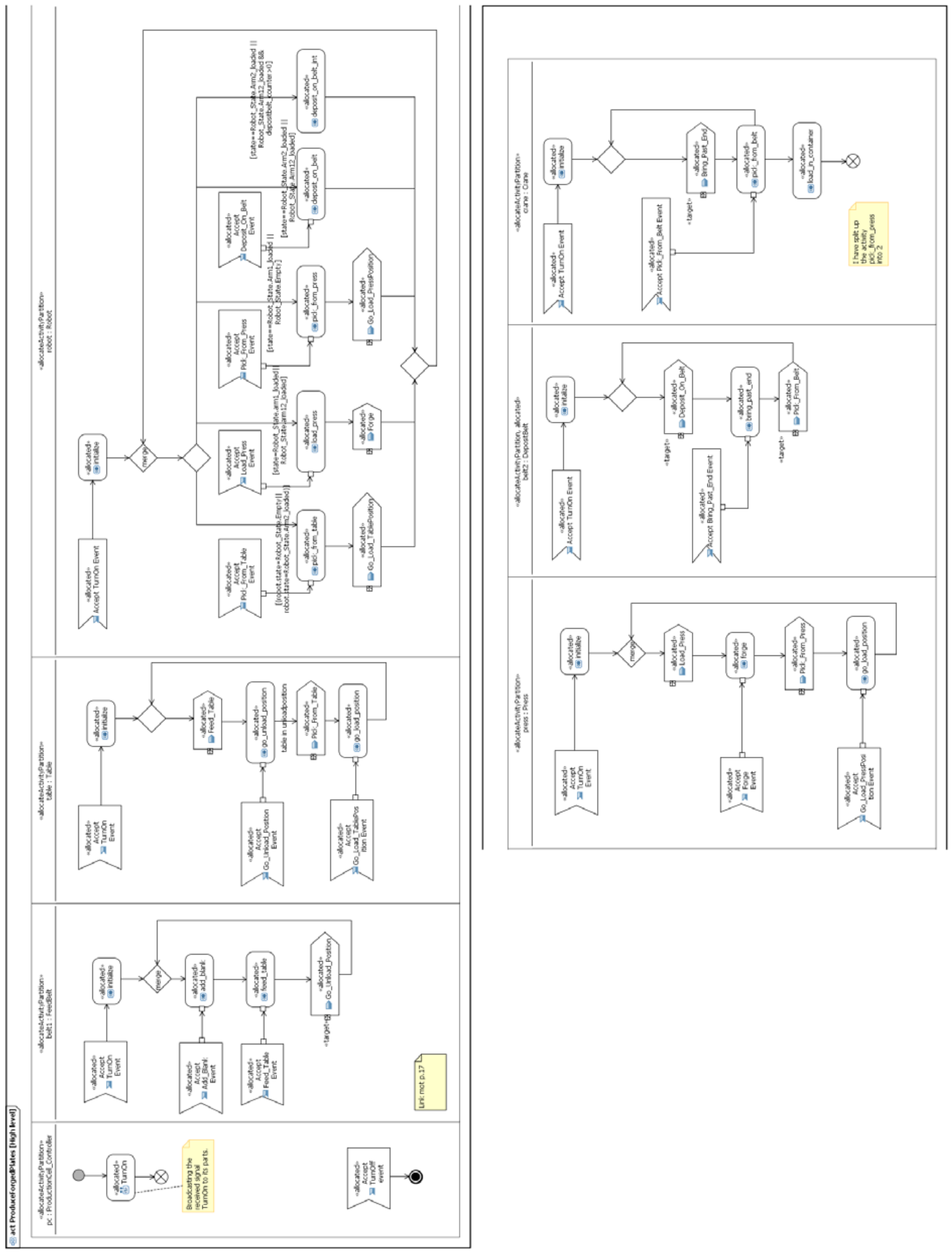


Figure 60 – The full activity diagram, act ProduceForgedPlates. For a closer look, see Figure 61, Figure 62 and Figure 63.

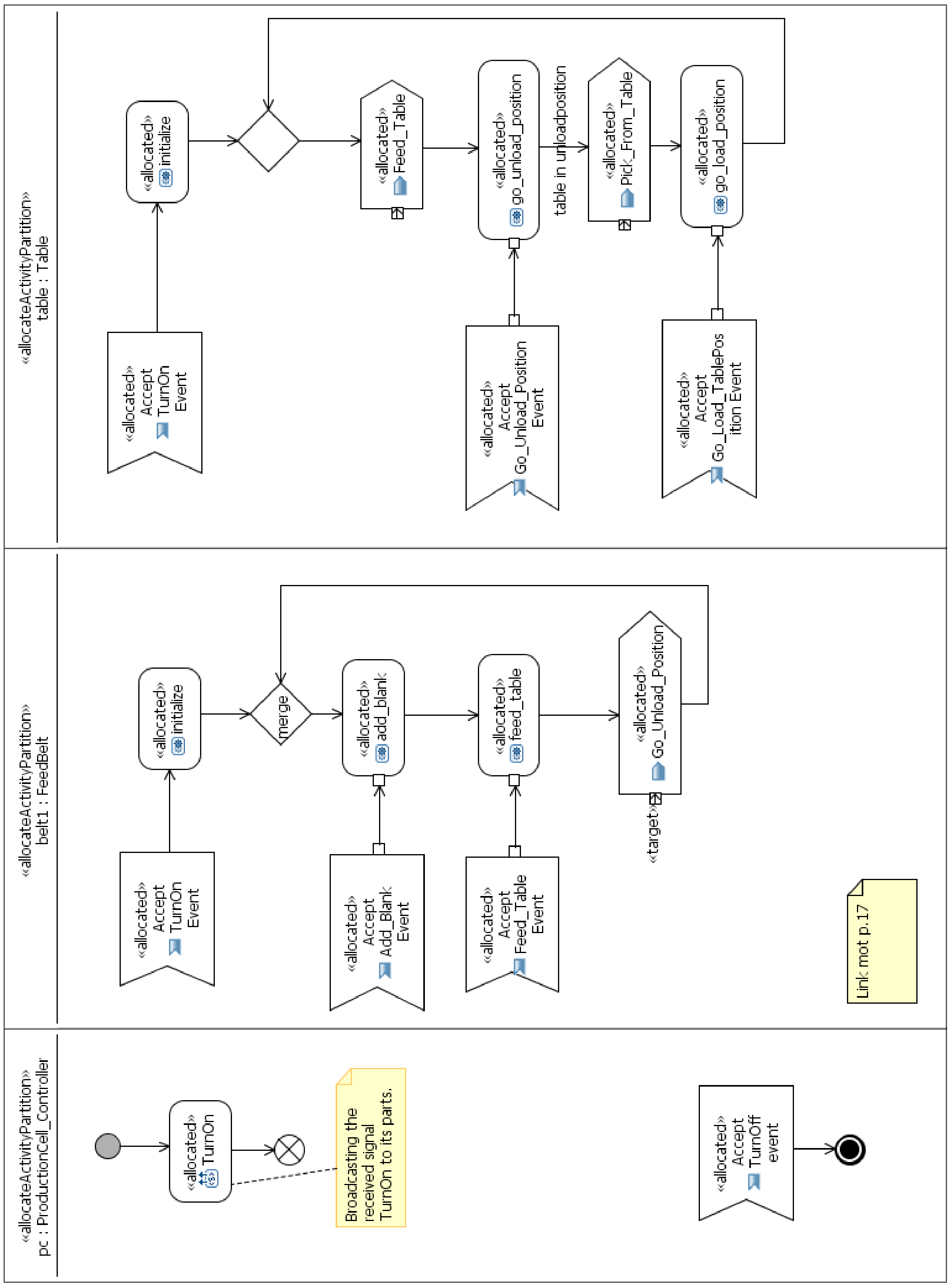

«allocateActivityPartition» table : Table
«allocated» Accept TurnOn Event
«allocated» initialize
«allocated» Feed_Table
«allocated» go_unload_position
table in unloadposition
«allocated» Pick_From_Table
«allocated» go_load_position
«allocated» Accept Go_Unload_Position Event
«allocated» Accept Go_Load_TablePosition Event
«allocateActivityPartition» belt1 : FeedBelt
«allocated» initialize
merge
«allocated» add_blank
«allocated» feed_table
«allocated» Go_Unload_Position
«target»
«allocated» Accept TurnOn Event
«allocated» Accept Add_Blank Event
«allocated» Accept Feed_Table Event
Link mot p.17
«allocateActivityPartition» pc : ProductionCell_Controller
«allocated» TurnOn
Broadcasting the received signal TurnOn to its parts.
«allocated» Accept TurnOff event

Figure 61 - Detail view of partitions for Controller, FeedBelt and Table from the diagram act ProduceForgedPlates in Figure 60

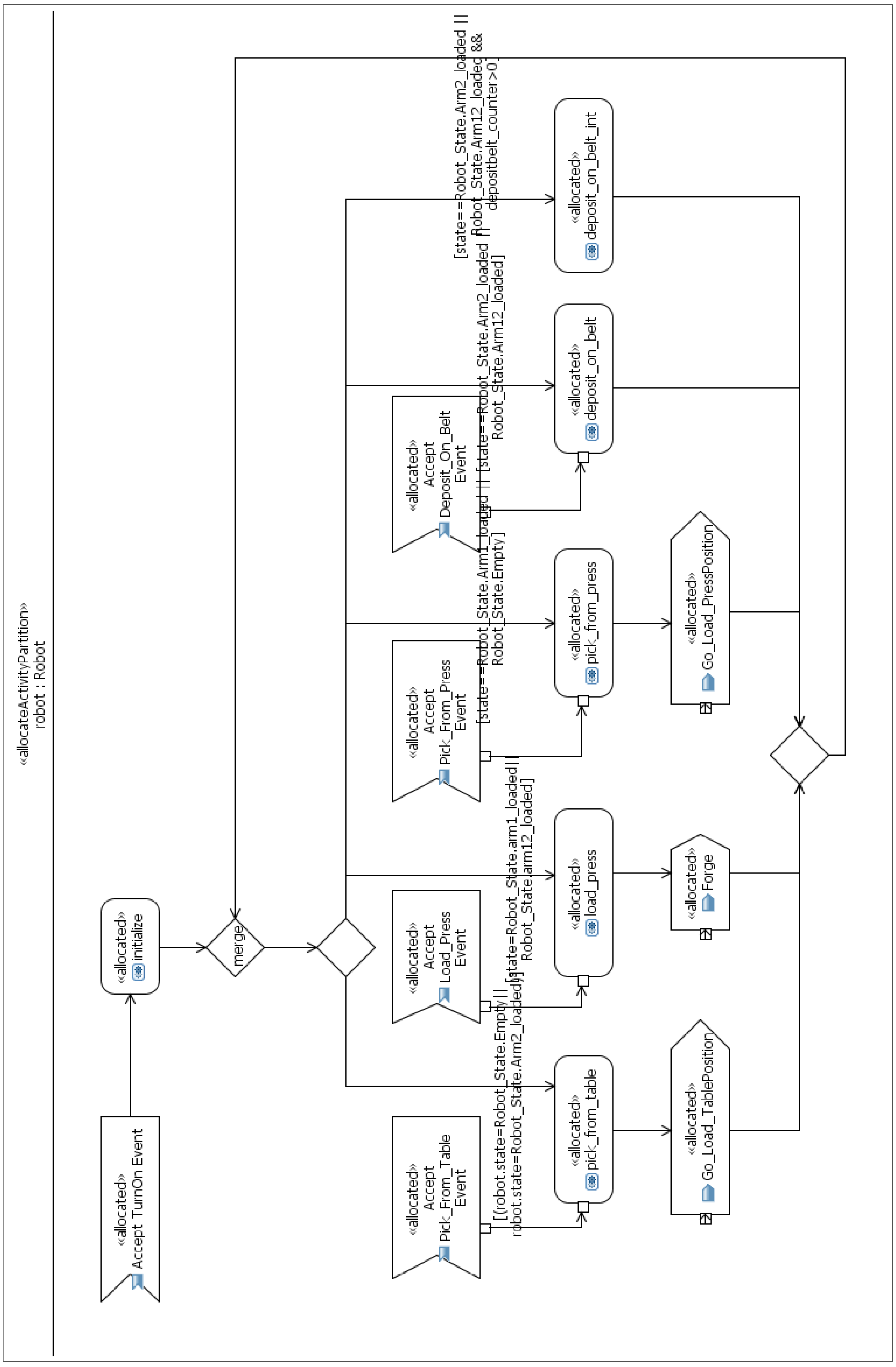

«allocateActivityPartition»
robot : Robot
«allocated» Accept TurnOn Event
«allocated» initialize
merge
«allocated» Accept Pick_From_Table Event
«allocated» Accept Load_Press Event
«allocated» Accept Pick_From_Press Event
«allocated» Accept Deposit_On_Belt Event
[(robot.state=Robot_State.Empty || robot.state=Robot_State.Arm2_loaded)]
[state=Robot_State.arm1_loaded|| Robot_State.arm12_loaded]
[state==Robot_State.Arm1_loaded || Robot_State.Empty]
[state==Robot_State.Arm2_loaded || Robot_State.Arm12_loaded]
[state==Robot_State.Arm2_loaded || Robot_State.Arm12_loaded && depositbelt_counter>0]
«allocated» pick_from_table
«allocated» load_press
«allocated» pick_from_press
«allocated» deposit_on_belt
«allocated» deposit_on_belt_int
«allocated» Go_Load_TablePosition
«allocated» Forge
«allocated» Go_Load_PressPosition

Figure 62 - Detail view of partitions for Robot from the diagram act ProduceForgedPlates in Figure 60. Note that there is an error in this diagram, see 6.1.1 for details.

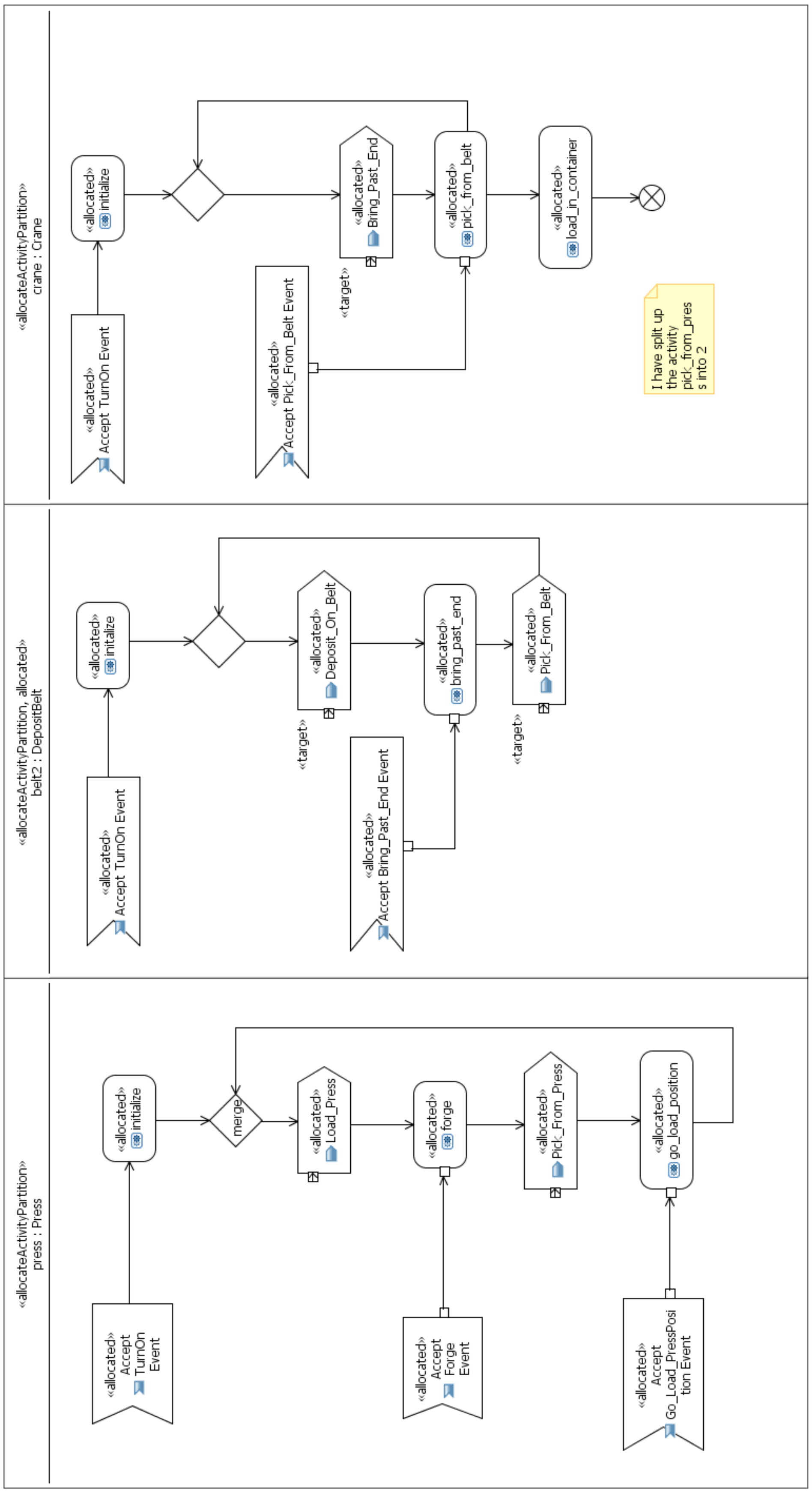

«allocateActivityPartition» crane : Crane
«allocated» Accept TurnOn Event
«allocated» initialize
«allocated» Accept Pick_From_Belt Event
«target»
«allocated» Bring_Past_End
«allocated» pick_from_belt
«allocated» load_in_container
I have split up the activity pick_from_press into 2
«allocateActivityPartition, allocated» belt2 : DepositBelt
«allocated» Accept TurnOn Event
«allocated» initalize
«target»
«allocated» Deposit_On_Belt
«allocated» Accept Bring_Past_End Event
«allocated» bring_past_end
«target»
«allocated» Pick_From_Belt
«allocateActivityPartition» press : Press
«allocated» Accept TurnOn Event
«allocated» initialize
merge
«allocated» Load_Press
«allocated» Accept Forge Event
«allocated» forge
«allocated» Pick_From_Press
«allocated» Accept Go_Load_PressPosition Event
«allocated» go_load_position

Figure 63 - Detail view of partitions for Press, DepositBelt and Crane from the diagram act ProduceForgedPlates in Figure 60.

A.6.3 Low level activity diagrams

A.6.3.1 FeedBelt

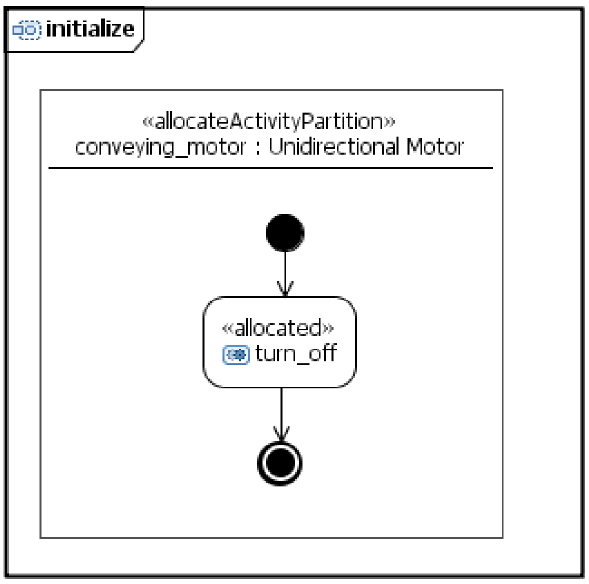

initialize
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocated»
turn_off

Figure 64 - Activity diagram for FeedBelt, act initialize, which is called from diagram in Figure 60.

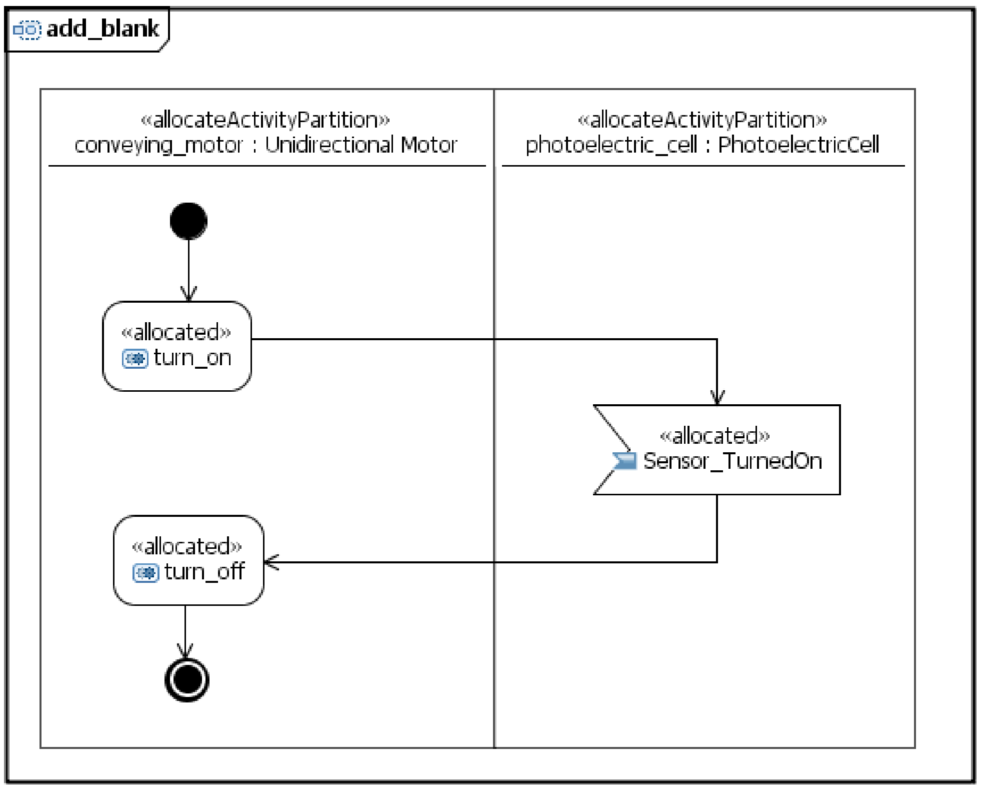

add_blank
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
«allocated»
Sensor_TurnedOn
«allocated»
turn_off

Figure 65 - Activity diagram for FeedBelt, act add_blank. which are called from diagram in Figure 60.

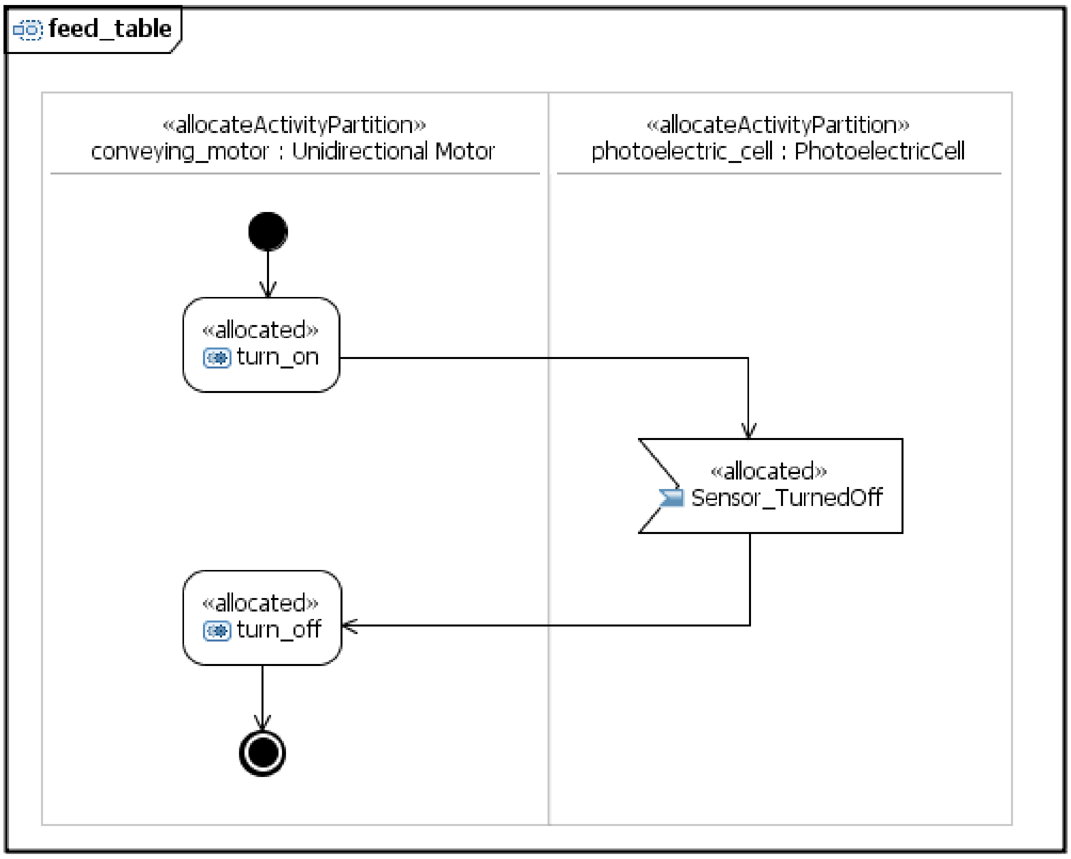

feed_table
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
«allocated»
Sensor_TurnedOff
«allocated»
turn_off

Figure 66 - Activity diagram for FeedBelt, act feed_table. which are called from diagram in Figure 60.

A.6.3.2 Table

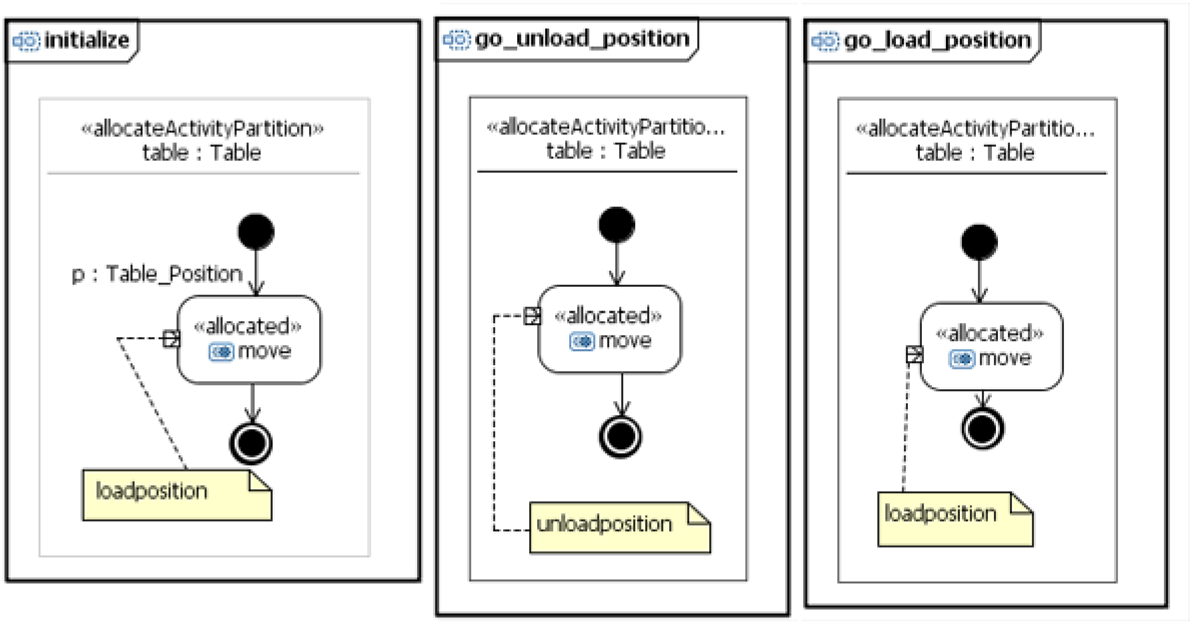


Figure 67 - Activity diagrams for Table, act initialize, act go_unload_position and act go_load_position. which are called from diagram in Figure 60.

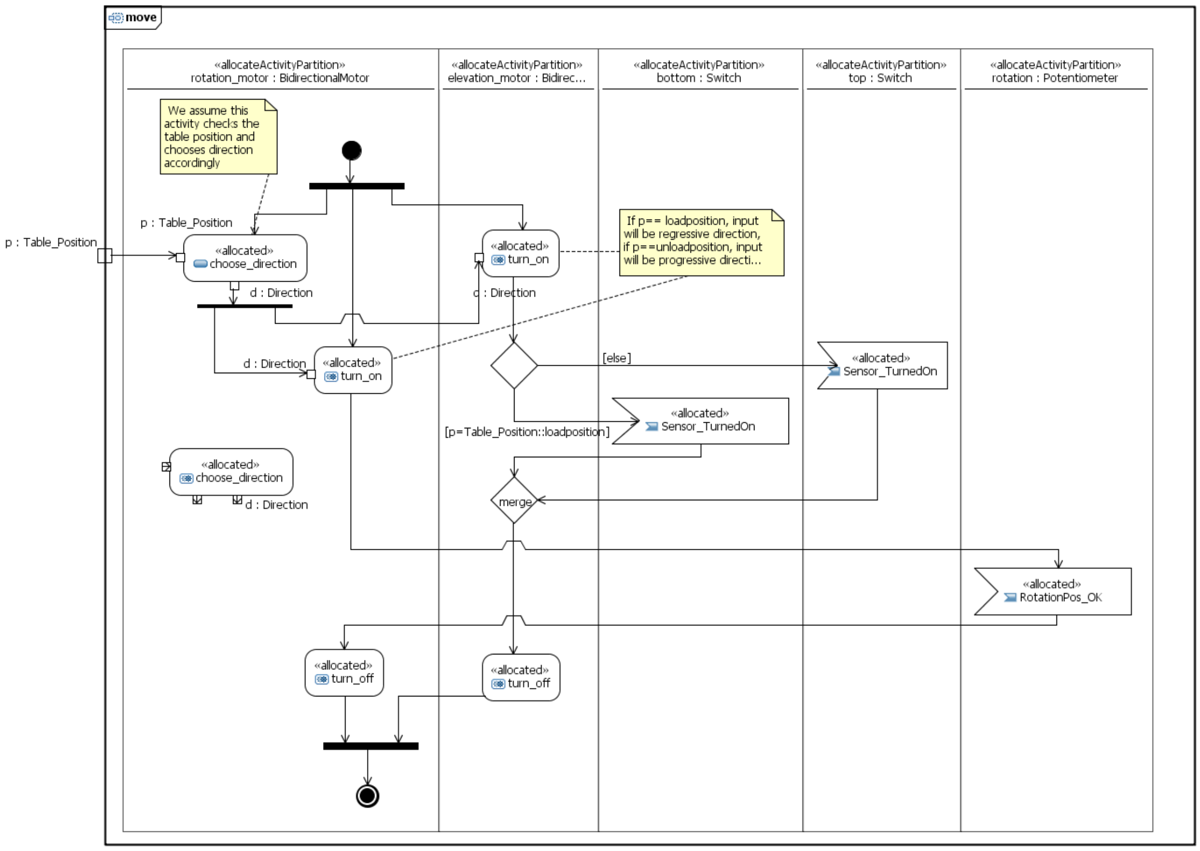

move
«allocateActivityPartition» rotation_motor : BidirectionalMotor
«allocateActivityPartition» elevation_motor : Bidirec...
«allocateActivityPartition» bottom : Switch
«allocateActivityPartition» top : Switch
«allocateActivityPartition» rotation : Potentiometer
We assume this activity checks the table position and chooses direction accordingly
p : Table_Position
p : Table_Position
«allocated» choose_direction
d : Direction
d : Direction
«allocated» turn_on
«allocated» turn_on
c : Direction
If p== loadposition, input will be regressive direction, if p==unloadposition, input will be progressive directi...
[else]
«allocated» Sensor_TurnedOn
[p=Table_Position::loadposition]
«allocated» Sensor_TurnedOn
«allocated» choose_direction
d : Direction
merge
«allocated» RotationPos_OK
«allocated» turn_off
«allocated» turn_off

Figure 68 - Activity diagram for Table, act move, which is called from Figure 67.

A.6.3.3 Robot

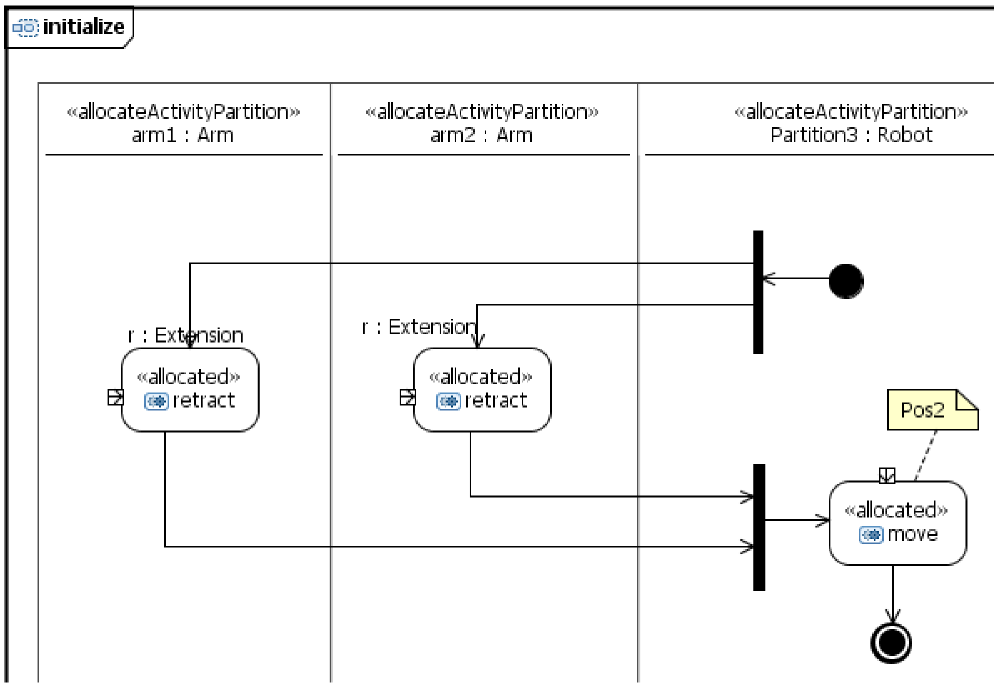

initialize
«allocateActivityPartition» arm1 : Arm
«allocateActivityPartition» arm2 : Arm
«allocateActivityPartition» Partition3 : Robot
r : Extension
«allocated» retract
r : Extension
«allocated» retract
Pos2
«allocated» move

Figure 69 - Activity diagram for Robot, act initialize, which is called from Figure 60.

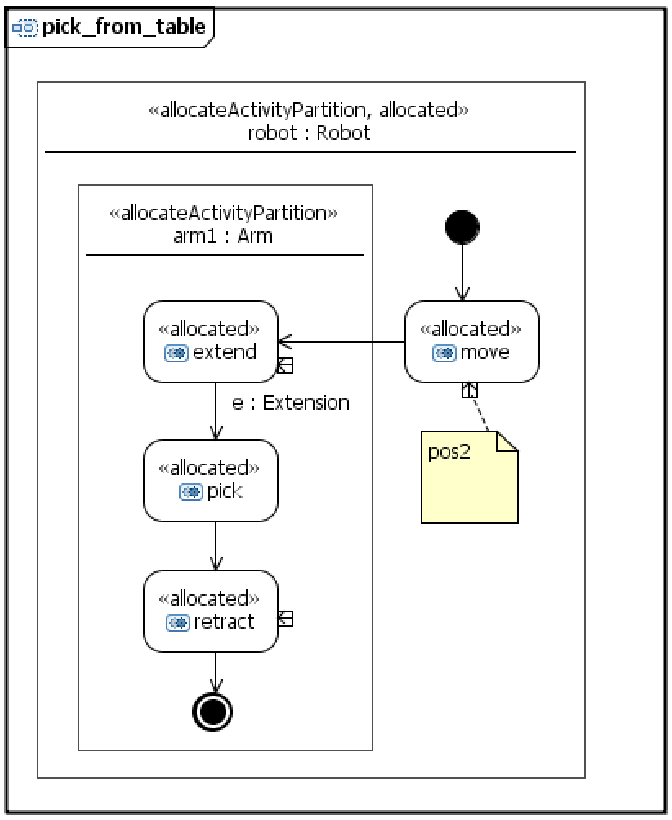


Figure 70 - Activity diagram for Robot, act pick_from_table, which is called from the diagram in Figure 60.

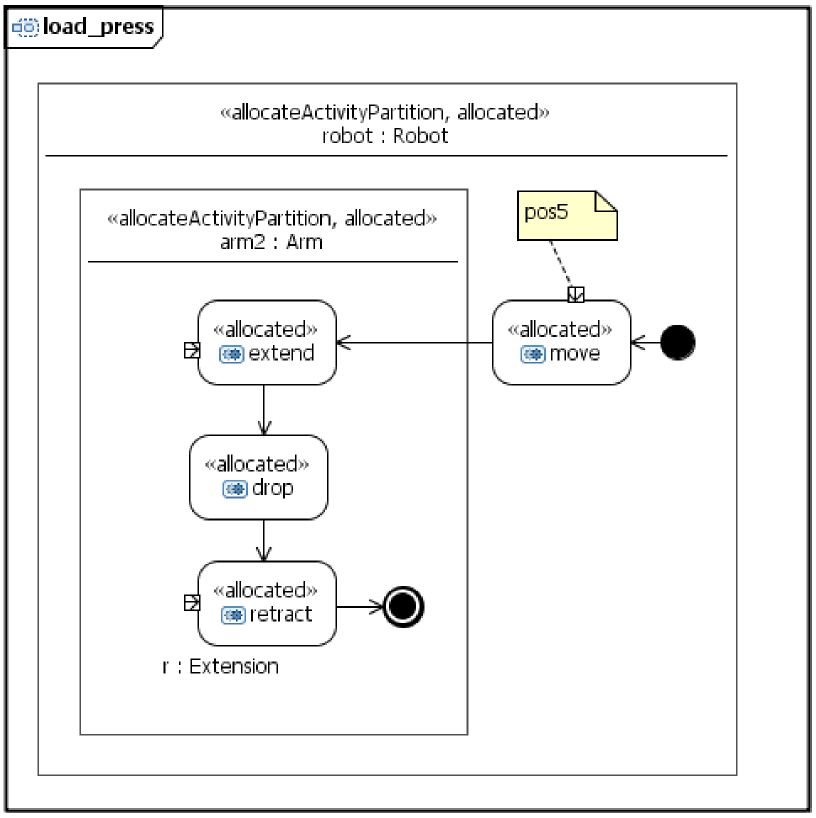


Figure 71 – Activity diagram for Robot, act load_press, which is called from the diagram in Figure 60.

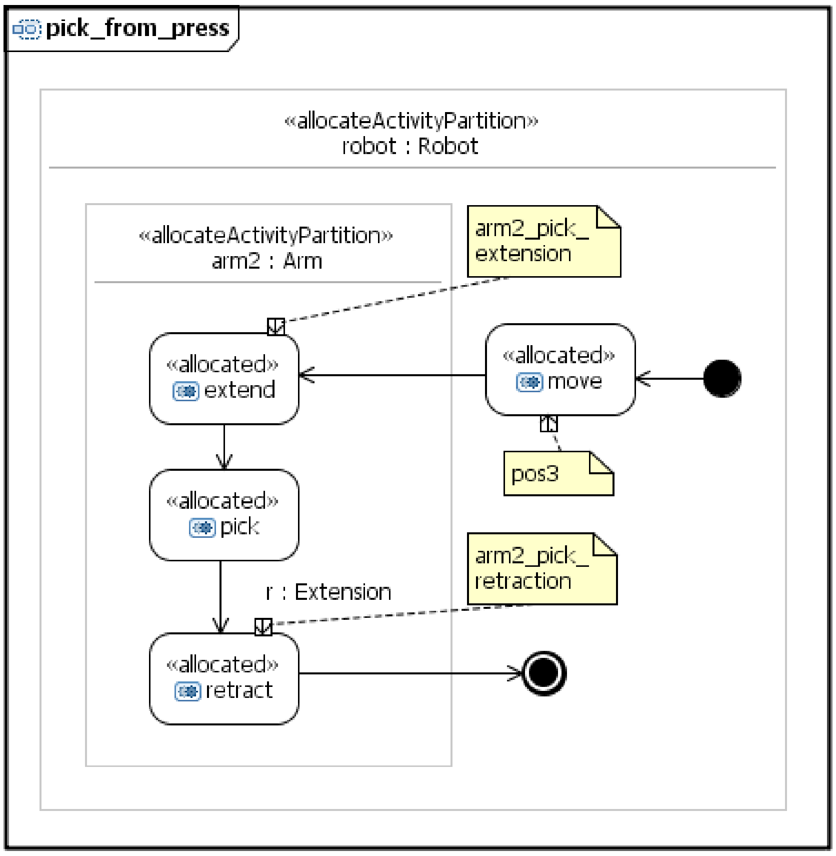


Figure 72 – Activity diagram for Robot, act pick_from_press, which is called from the diagram in Figure 60.

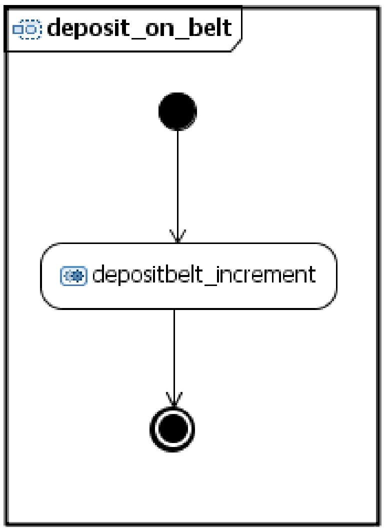


Figure 73 – Activity diagram for Robot, act deposit_on_belt, which is called from the diagram in Figure 60.

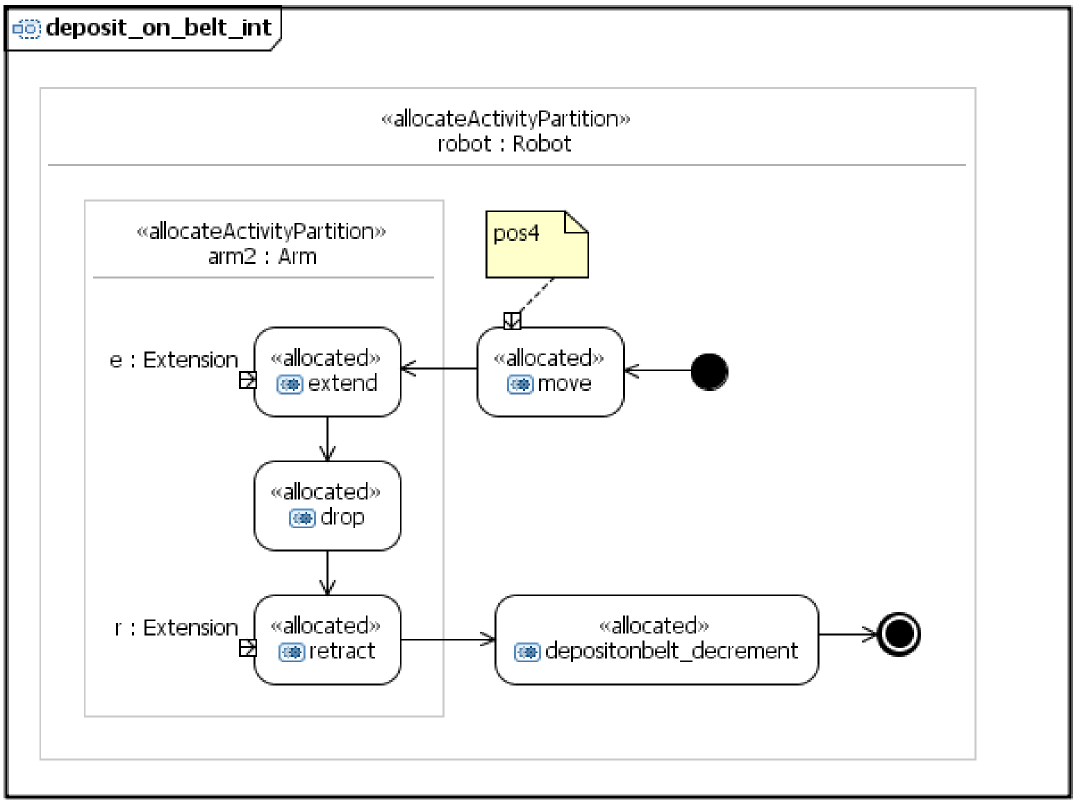


Figure 74 - Activity diagram for Robot, act deposit_on_belt_int, which is called from the diagram in Figure 60.

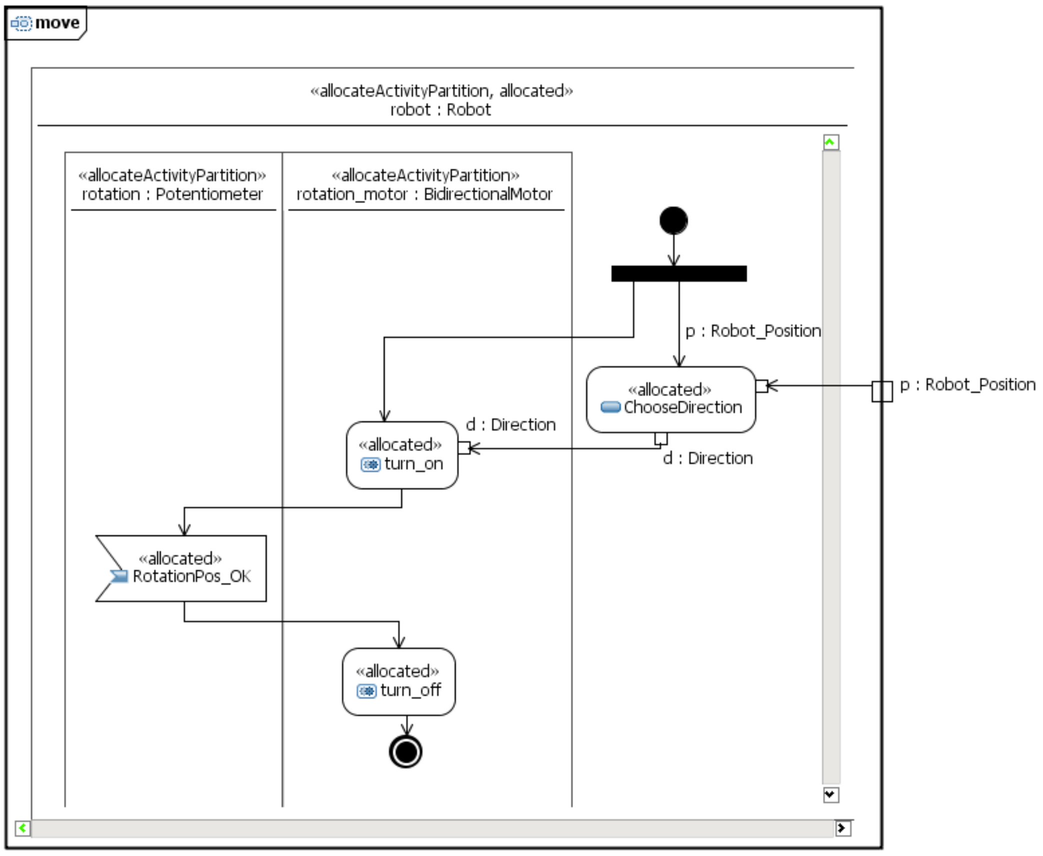


Figure 75 – Activity diagram for Robot, act move, which is called from most of the above Robot diagrams.

A.6.3.4 Press

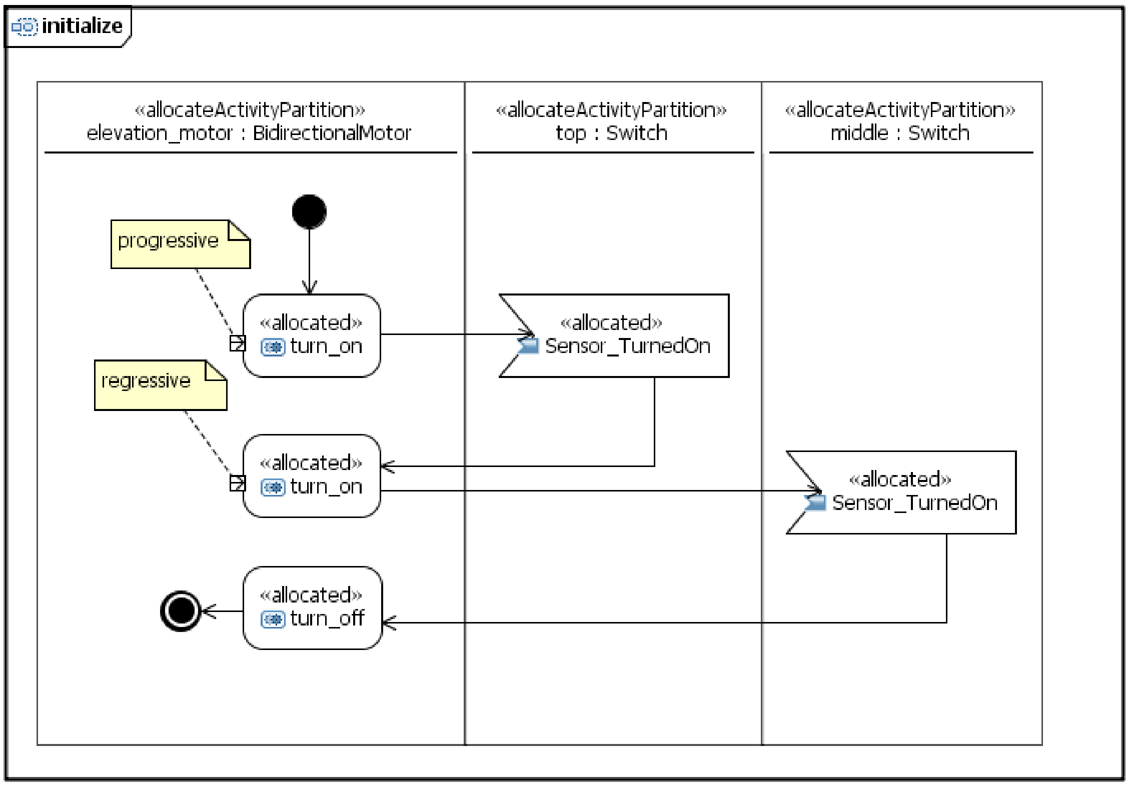


Figure 76 - Activity diagram for Press, act initialize, which is called from Figure 60.

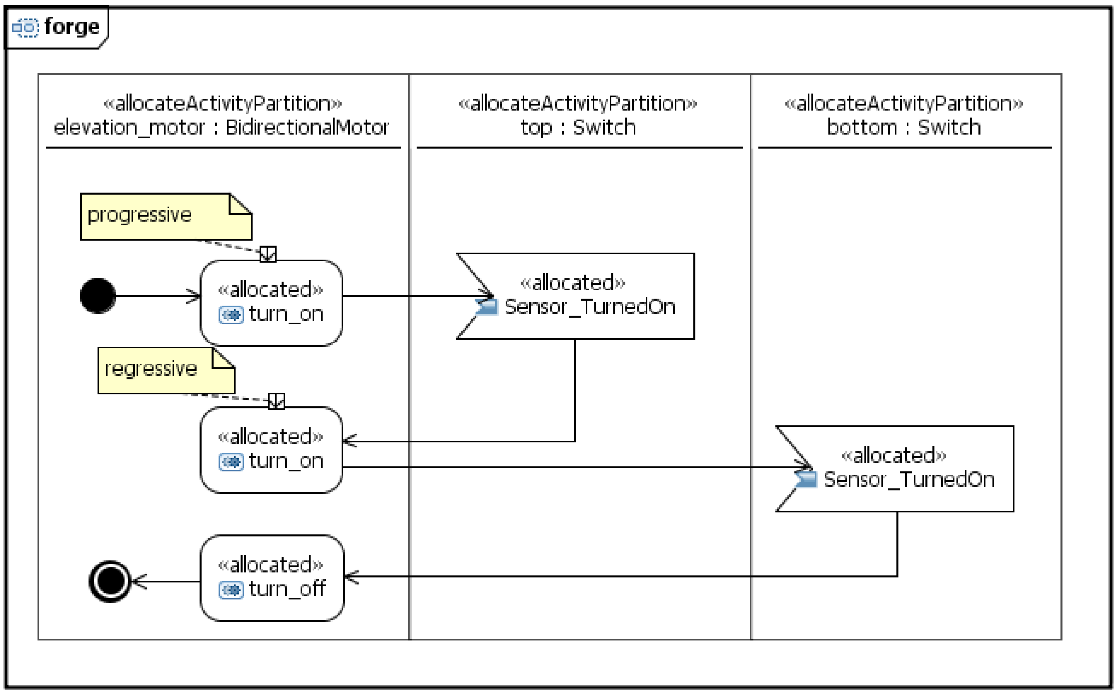


Figure 77 - Activity diagram for Press, act forge, which is called from the diagram in Figure 60.

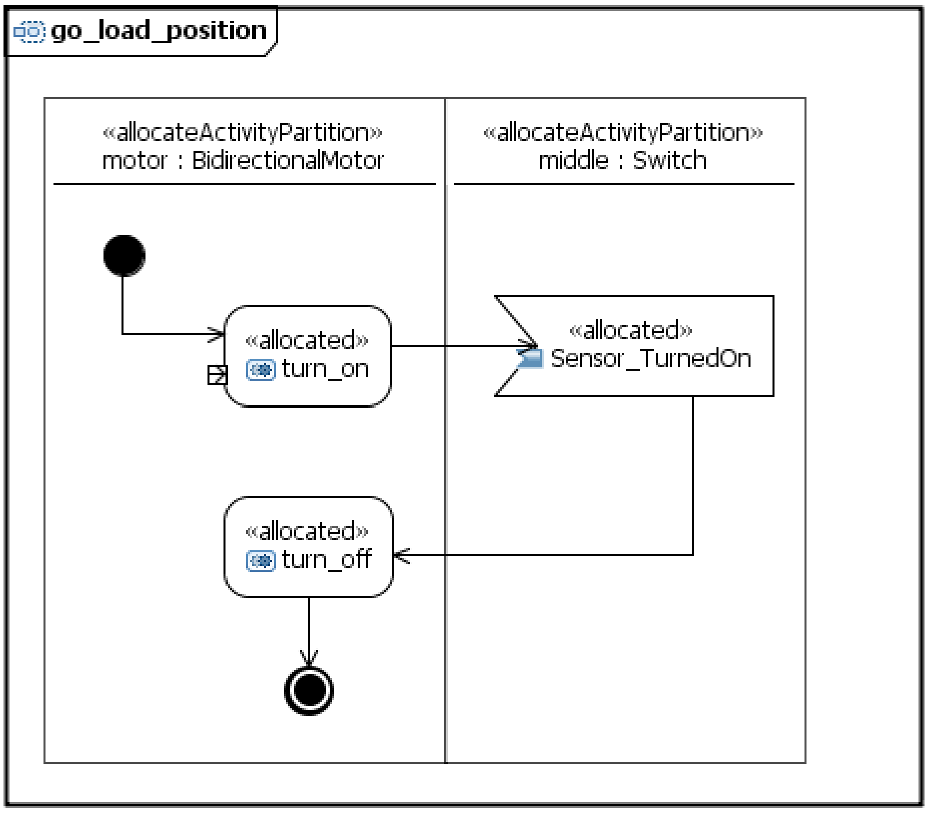


Figure 78 - Activity diagram for Press, act go_load_position, which is called from the diagram in Figure 60.

A.6.3.5 DepostiBelt

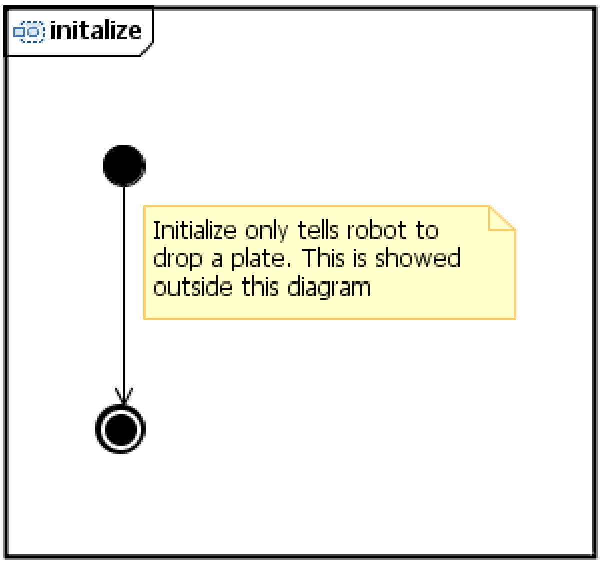


Figure 79 - Activity diagram for DepositBelt, act initialize, which is called from the diagram in Figure 60.

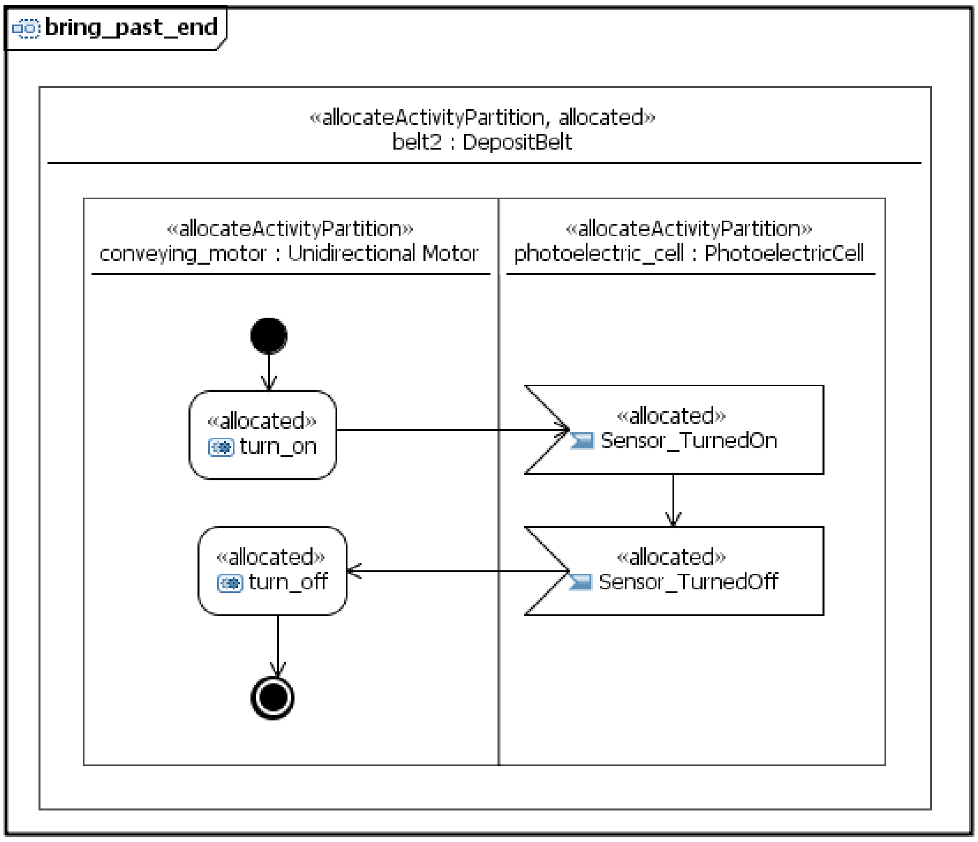

bring_past_end
«allocateActivityPartition, allocated»
belt2 : DepositBelt
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
«allocated»
Sensor_TurnedOn
«allocated»
turn_off
«allocated»
Sensor_TurnedOff

Figure 80 - Activity diagram for DepostiBelt, act bring_past_end, which is called from the diagram in Figure 60.

A.6.3.6 Crane

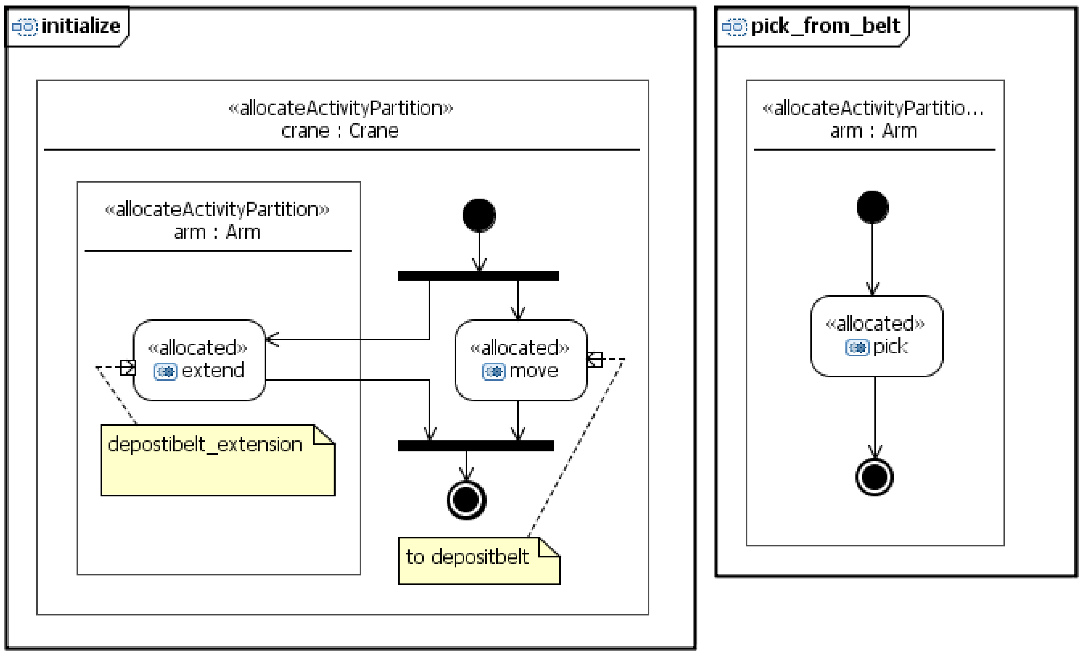


Figure 81 - Activity diagram for Crane, act initialize and act pick_from_belt, which are called from the diagram in Figure 60.

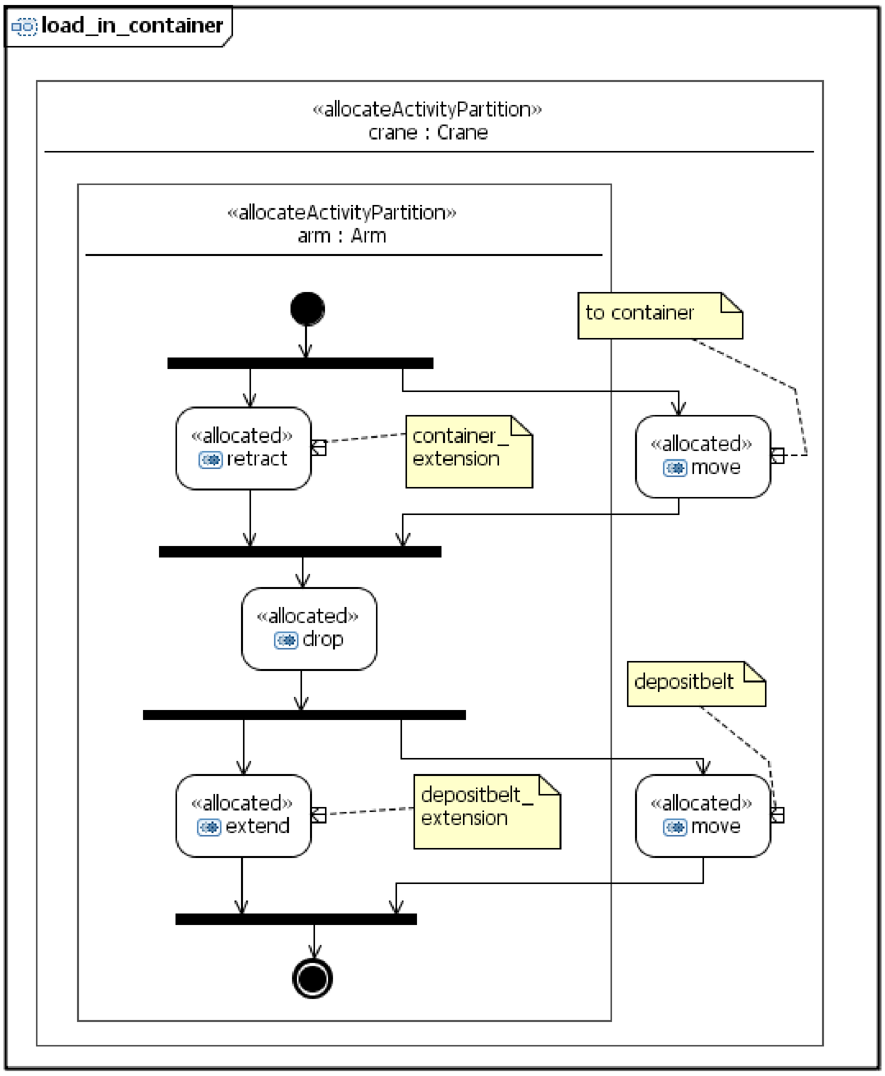


Figure 82 - Activity diagram for Crane, act load_in_container, which is called from the diagram in Figure 60. Note that this activity diagram describes the second part of the method pick_from_belt as it is described in [5], first part is described act pick_from_belt in Figure 81.

A.6.3.7 Arm

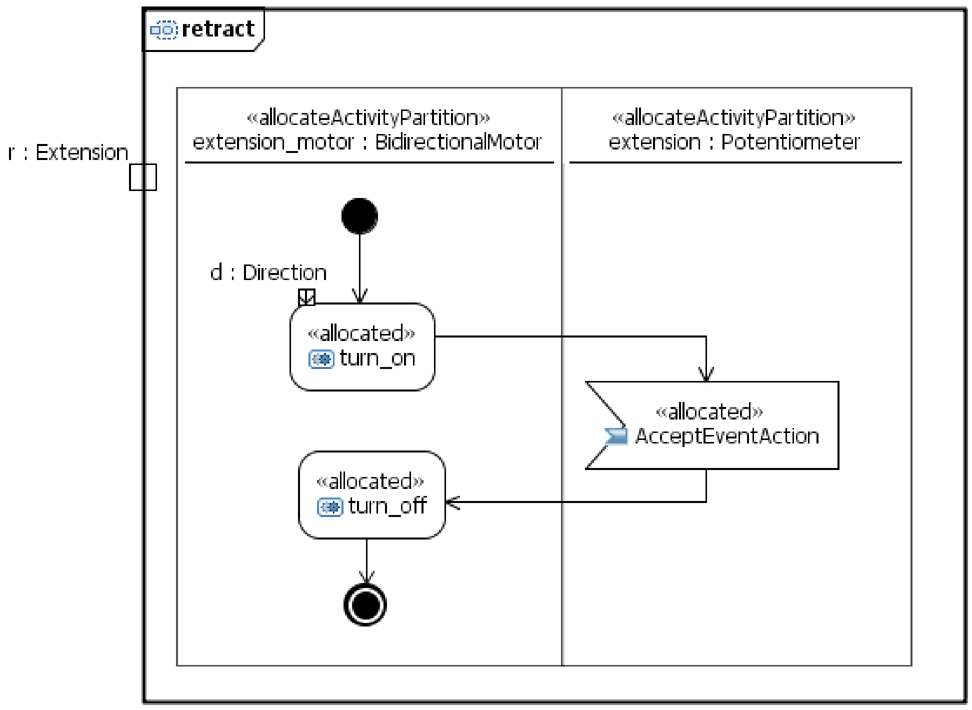

retract
r : Extension
«allocateActivityPartition»
extension_motor : BidirectionalMotor
«allocateActivityPartition»
extension : Potentiometer
d : Direction
«allocated»
turn_on
«allocated»
AcceptEventAction
«allocated»
turn_off

Figure 83 - Activity diagram for Arm, act retract, which is called from Robot and Crane activities.

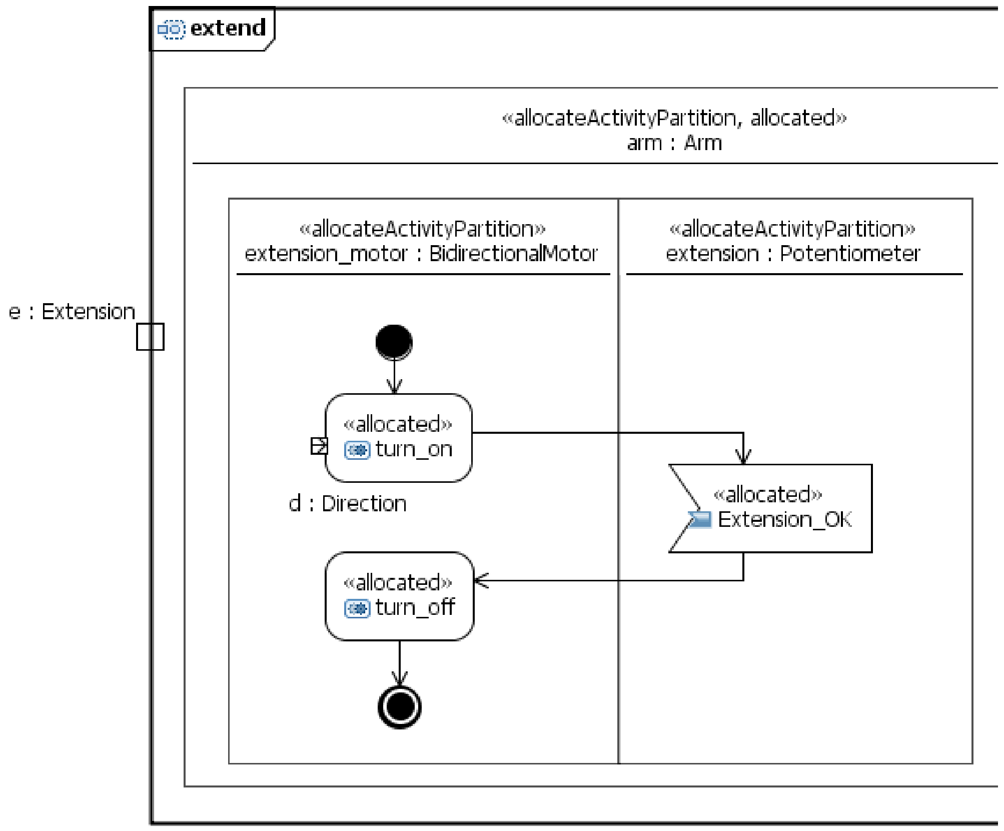

extend
«allocateActivityPartition, allocated»
arm : Arm
«allocateActivityPartition»
extension_motor : BidirectionalMotor
«allocateActivityPartition»
extension : Potentiometer
e : Extension
«allocated»
turn_on
d : Direction
«allocated»
Extension_OK
«allocated»
turn_off

Figure 84 - Activity diagram for Arm, act extend, which is called from Robot and Crane activities.

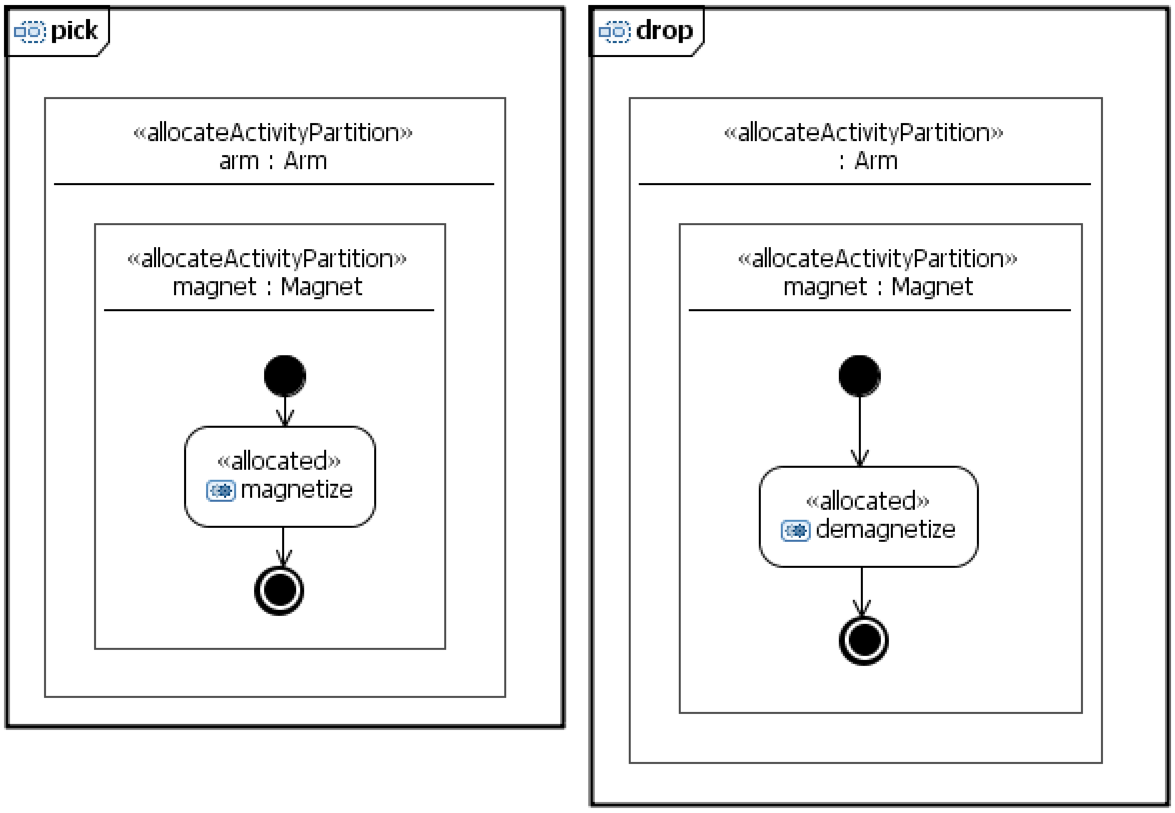

pick
«allocateActivityPartition»
arm : Arm
«allocateActivityPartition»
magnet : Magnet
«allocated»
magnetize
drop
«allocateActivityPartition»
: Arm
«allocateActivityPartition»
magnet : Magnet
«allocated»
demagnetize

Figure 85 – Activity diagrams for Arm, act pick and act drop, which are called from Robot and Crane activities.

A.6.3.8 UniDirectionalMotor

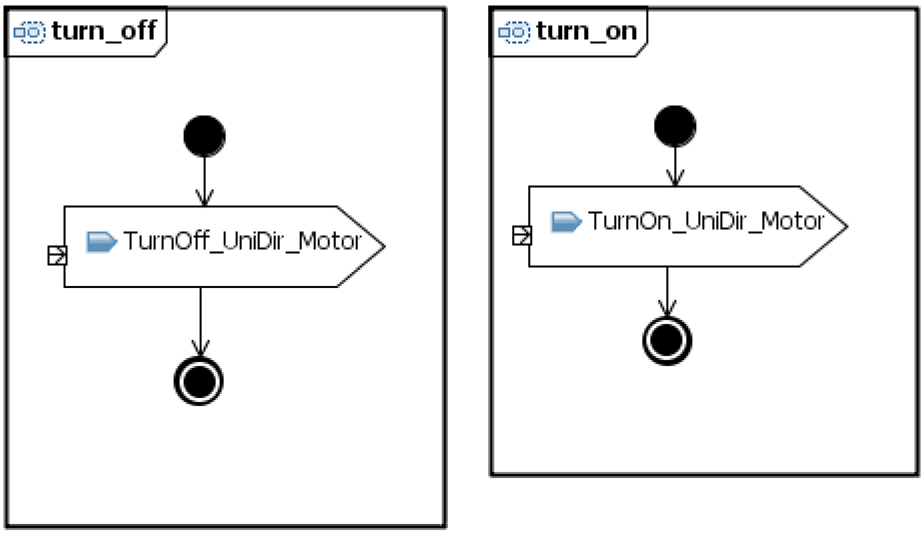
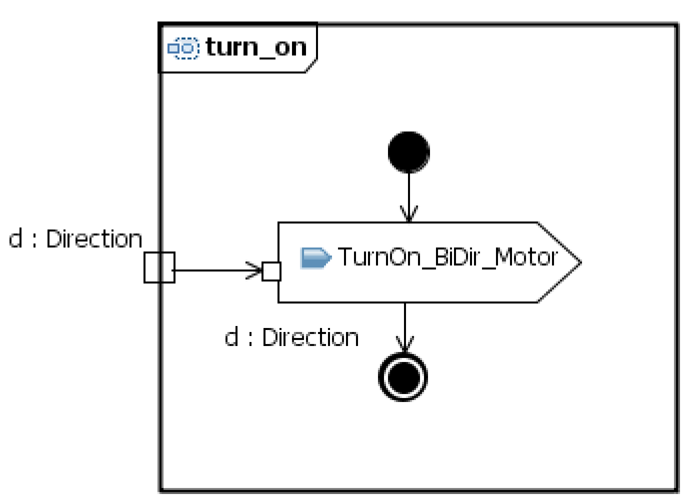


Figure 86 – Activity diagrams for UniDirectionalMotor, act turn_off and act turn_on, which are called from FeedBelt and DepositBelt activities.

A.6.3.9 BiDirectionalMotor



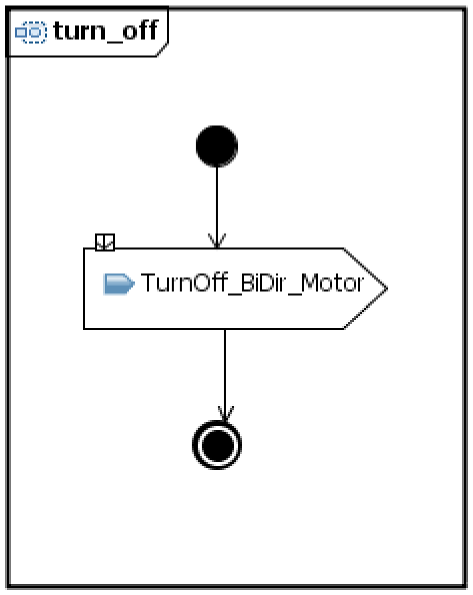
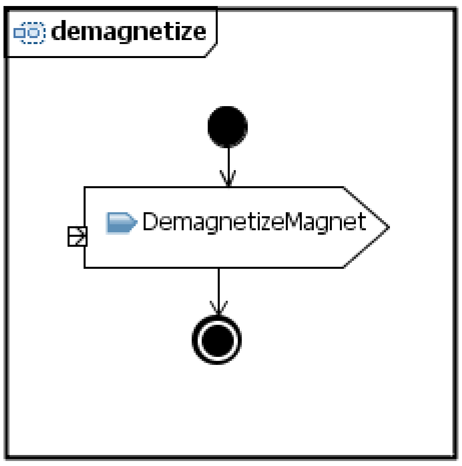


Figure 87 - Activity diagrams for BiDirectionalMotor, act turn_on and act turn_off, which are called from Table, Robot, Press and Crane activities.

A.6.3.10 Magnet



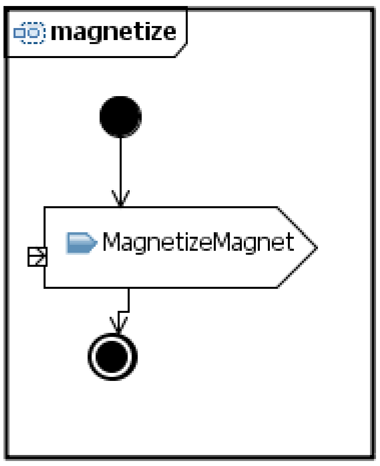


Figure 88 - Activity diagrams for Magnet, act demagnetize and act magnetize, which are called from Arm activities in Figure 85.

A.7 State machines

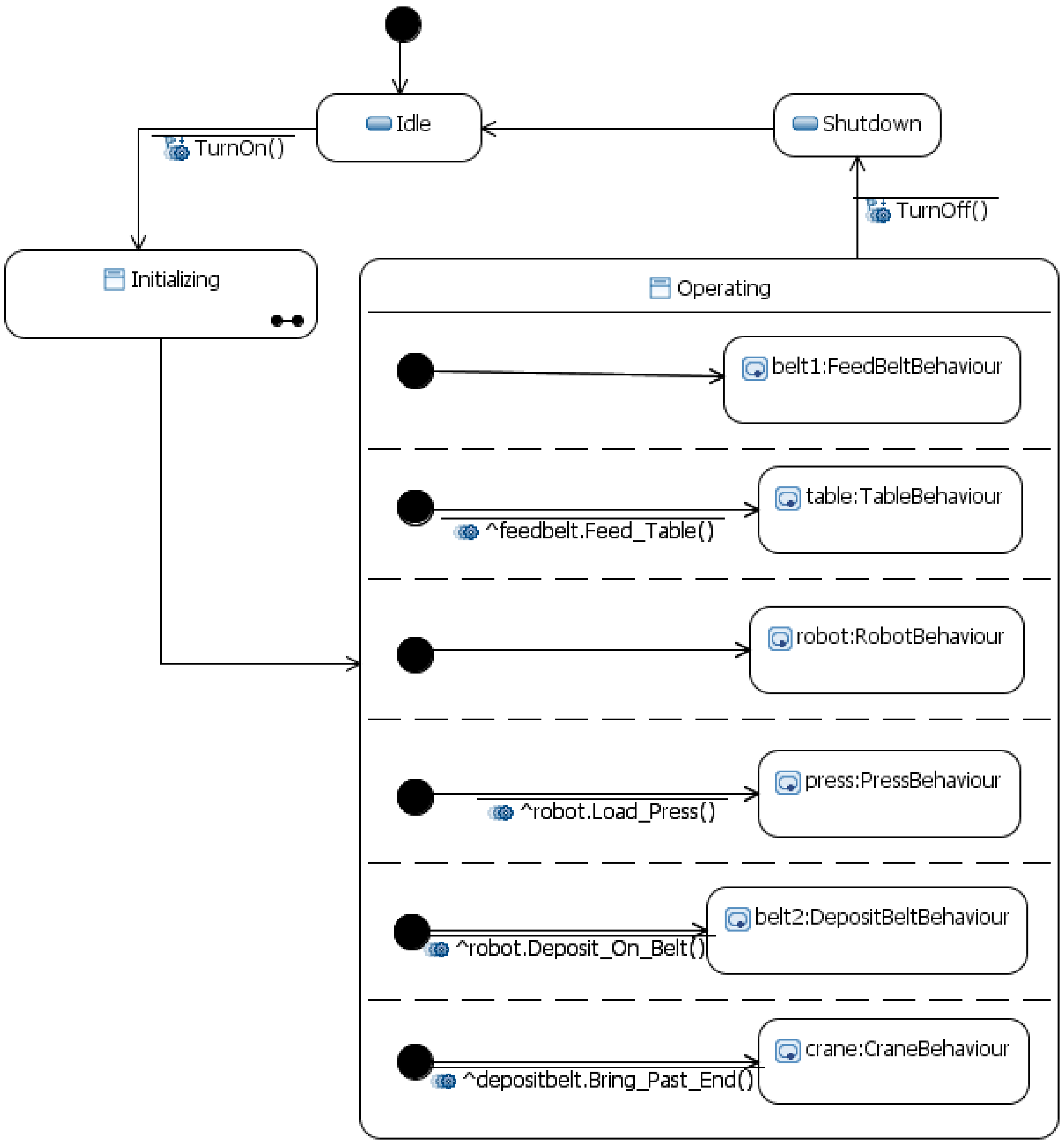


Figure 89 stm [Block] ProductionCell_Controller, with focus on the state Operating.

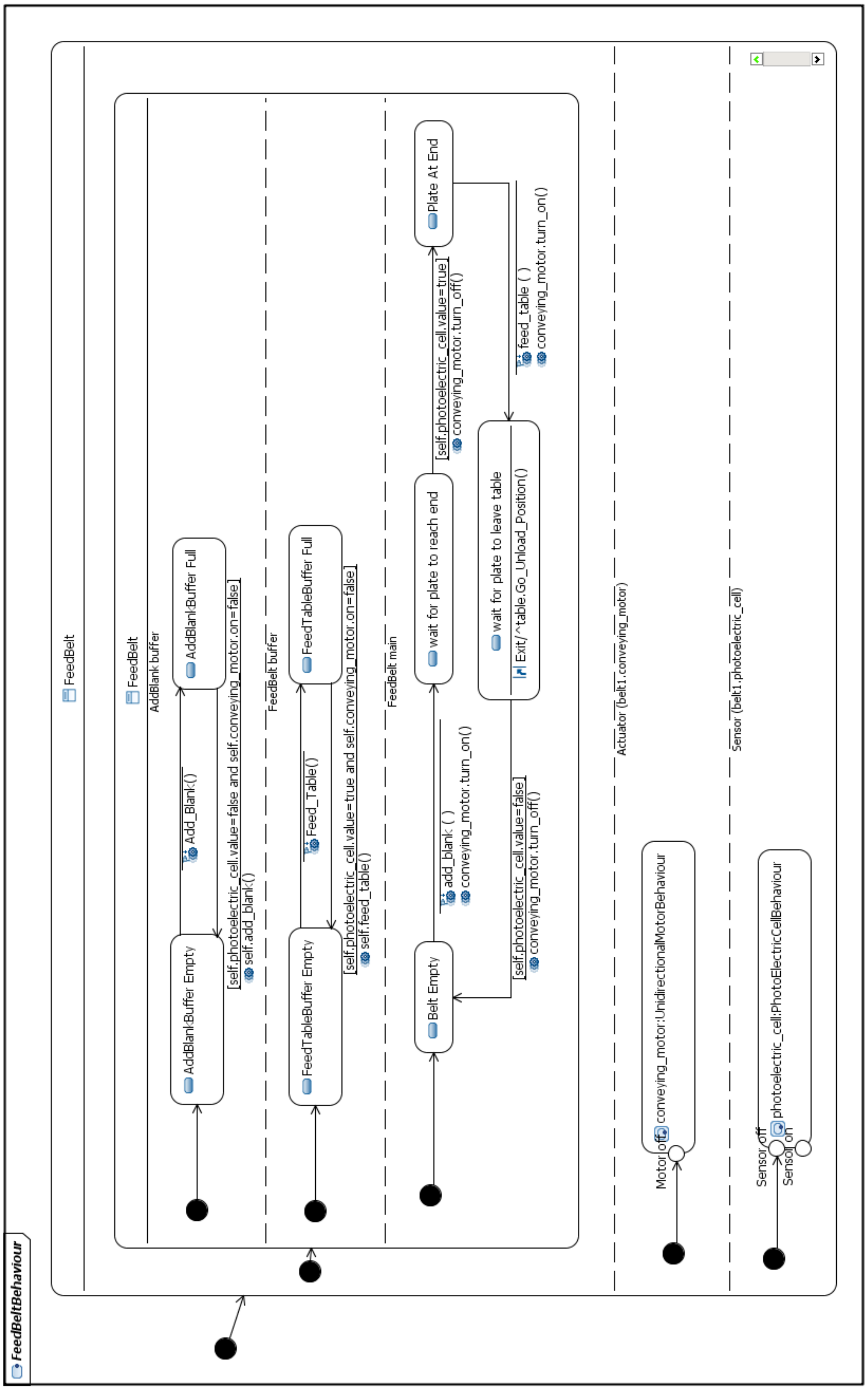

FeedBeltBehaviour
FeedBelt
FeedBelt
AddBlank buffer
AddBlank-Buffer Full
AddBlank-Buffer Empty
Add_Blank()
[self.photoelectric_cell.value=false and self.conveying_motor.on=false]
self.add_blank()
FeedBelt buffer
FeedTableBuffer Full
FeedTableBuffer Empty
Feed_Table()
[self.photoelectric_cell.value=true and self.conveying_motor.on=false]
self.feed_table()
FeedBelt main
Plate At End
wait for plate to reach end
Belt Empty
wait for plate to leave table
Exit/^table.Go_Unload_Position()
[self.photoelectric_cell.value=true]
conveying_motor.turn_off()
feed_table ( )
conveying_motor.turn_on()
add_blank ( )
conveying_motor.turn_on()
[self.photoelectric_cell.value=false]
conveying_motor.turn_off()
Actuator (belt1.conveying_motor)
conveying_motor:UnidirectionalMotorBehaviour
Motor off
Sensor (belt1.photoelectric_cell)
photoelectric_cell:PhotoElectricCellBehaviour
Sensor off
Sensor on

Figure 90 - stm [Block] FeedBelt [Operating].

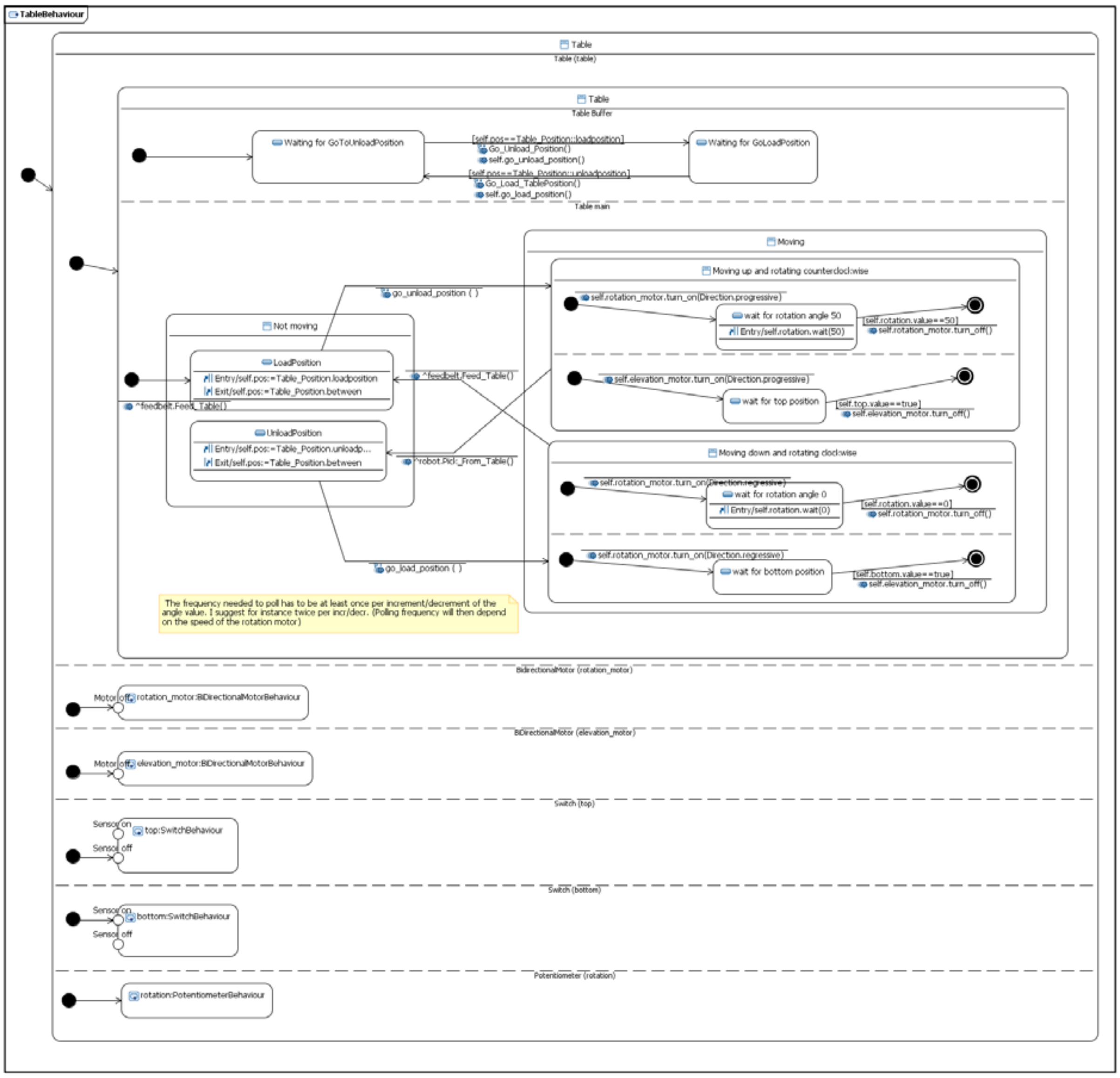

TableBehaviour
Table
Table (table)
Table
Table Buffer
Waiting for GoToUnloadPosition
[self.pos==Table_Position::loadposition]
Go_Unload_Position()
self.go_unload_position()
Waiting for GoLoadPosition
[self.pos==Table_Position::unloadposition]
Go_Load_TablePosition()
self.go_load_position()
Table main
Moving
Moving up and rotating counterclockwise
go_unload_position ( )
self.rotation_motor.turn_on(Direction.progressive)
wait for rotation angle 50
Entry/self.rotation.wait(50)
[self.rotation.value==50]
self.rotation_motor.turn_off()
self.elevation_motor.turn_on(Direction.progressive)
wait for top position
[self.top.value==true]
self.elevation_motor.turn_off()
Not moving
LoadPosition
Entry/self.pos:=Table_Position.loadposition
Exit/self.pos:=Table_Position.between
^feedbelt.Feed_Table()
^feedbelt.Feed_Table()
UnloadPosition
Entry/self.pos:=Table_Position.unloadp...
Exit/self.pos:=Table_Position.between
^robot.Pick_From_Table()
Moving down and rotating clockwise
self.rotation_motor.turn_on(Direction.regressive)
wait for rotation angle 0
Entry/self.rotation.wait(0)
[self.rotation.value==0]
self.rotation_motor.turn_off()
go_load_position ( )
self.rotation_motor.turn_on(Direction.regressive)
wait for bottom position
[self.bottom.value==true]
self.elevation_motor.turn_off()
The frequency needed to poll has to be at least once per increment/decrement of the angle value. I suggest for instance twice per incr/decr. (Polling frequency will then depend on the speed of the rotation motor)
BidirectionalMotor (rotation_motor)
Motor off
rotation_motor:BiDirectionalMotorBehaviour
BiDirectionalMotor (elevation_motor)
Motor off
elevation_motor:BiDirectionalMotorBehaviour
Switch (top)
Sensor on
Sensor off
top:SwitchBehaviour
Switch (bottom)
Sensor on
Sensor off
bottom:SwitchBehaviour
Potentiometer (rotation)
rotation:PotentiometerBehaviour

Figure 91 - stm [Block] Table [Operating]. For details, see Figure 92 and Figure 93.

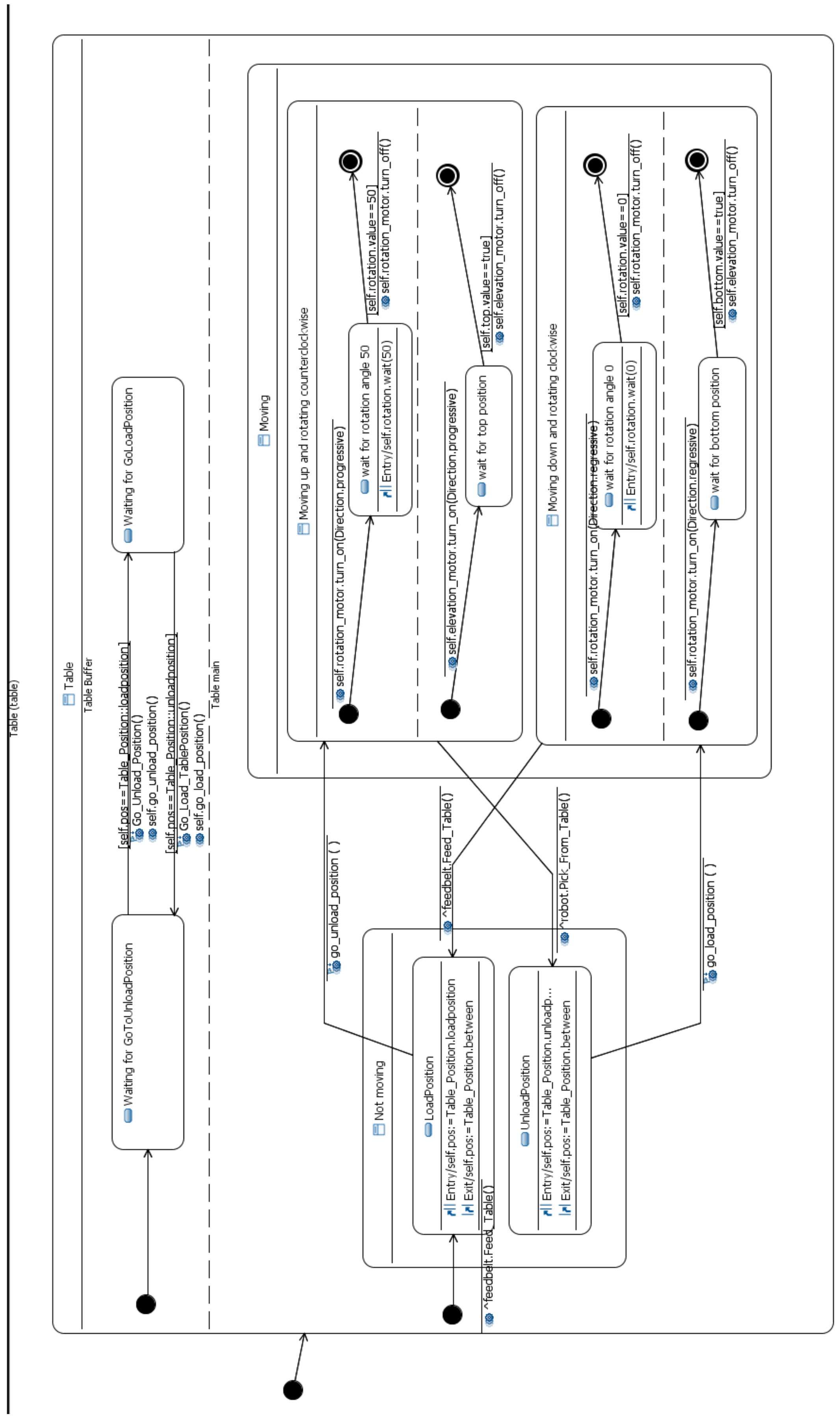


Table (table)
Table
Table Buffer
Table main
Waiting for GoToUnloadPosition
Waiting for GoLoadPosition
[self.pos==Table_Position::loadposition]
Go_Unload_Position()
self.go_unload_position()
[self.pos==Table_Position::unloadposition]
Go_Load_TablePosition()
self.go_load_position()
Moving
Moving up and rotating counterclockwise
self.rotation_motor.turn_on(Direction.progressive)
wait for rotation angle 50
Entry/self.rotation.wait(50)
[self.rotation.value==50]
self.rotation_motor.turn_off()
self.elevation_motor.turn_on(Direction.progressive)
wait for top position
[self.top.value==true]
self.elevation_motor.turn_off()
Moving down and rotating clockwise
self.rotation_motor.turn_on(Direction.regressive)
wait for rotation angle 0
Entry/self.rotation.wait(0)
[self.rotation.value==0]
self.rotation_motor.turn_off()
self.rotation_motor.turn_on(Direction.regressive)
wait for bottom position
[self.bottom.value==true]
self.elevation_motor.turn_off()
Not moving
LoadPosition
Entry/self.pos:=Table_Position.loadposition
Exit/self.pos:=Table_Position.between
UnloadPosition
Entry/self.pos:=Table_Position.unloadp...
Exit/self.pos:=Table_Position.between
go_unload_position ( )
^feedbelt.Feed_Table()
^robot.Pick_From_Table()
go_load_position ( )
^feedbelt.Feed_Table()

Figure 92 – Part of state machine for Table, showing only buffer and main regions. For whole diagram, see Figure 91.

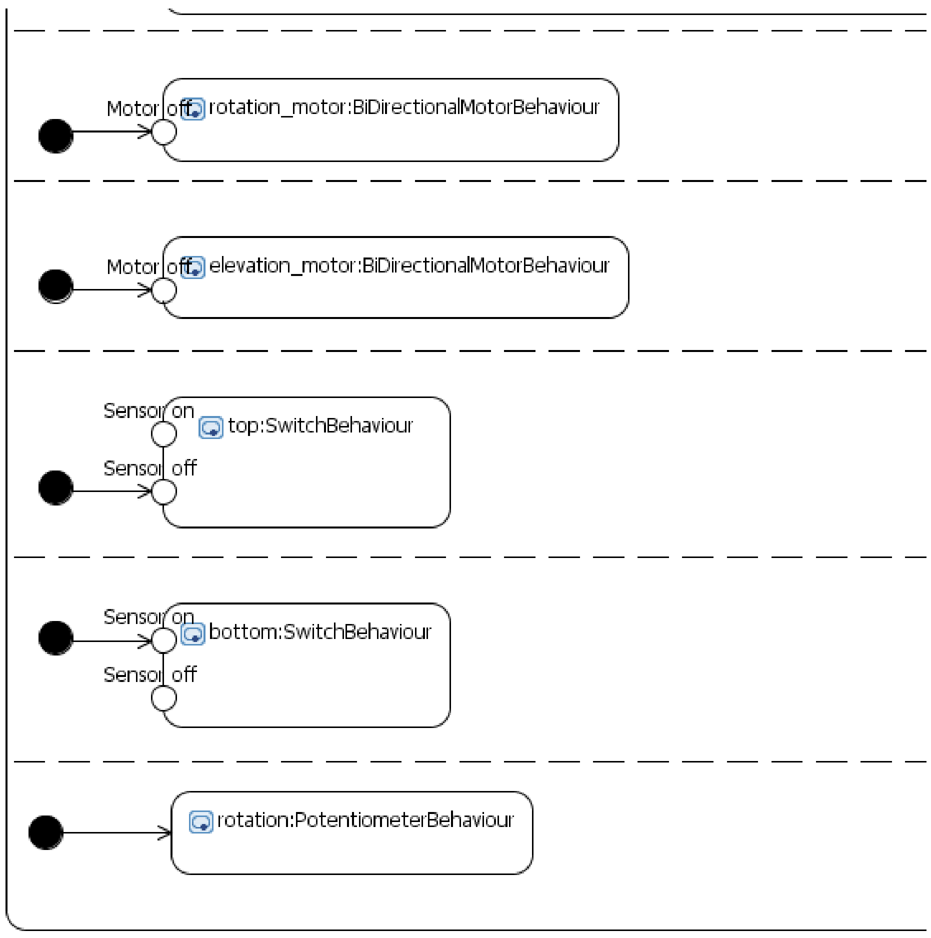

Motor off
rotation_motor:BiDirectionalMotorBehaviour
Motor off
elevation_motor:BiDirectionalMotorBehaviour
Sensor on
top:SwitchBehaviour
Sensor off
Sensor on
bottom:SwitchBehaviour
Sensor off
rotation:PotentiometerBehaviour

Figure 93 – Part of state machine for Table, showing only the part regions. For whole diagram see Figure 91.

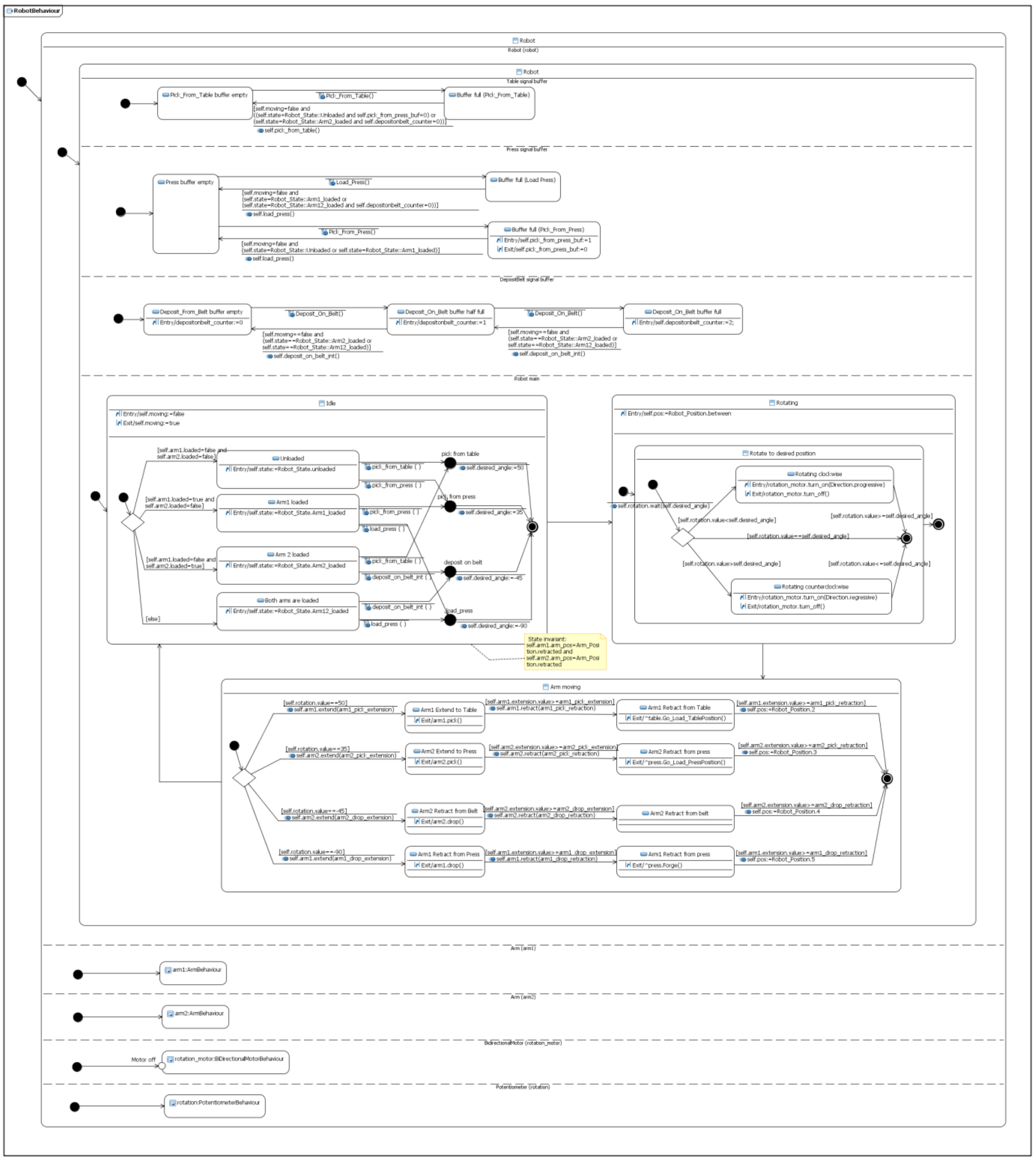


Figure 94 - stm [Block] Robot [Operating], for detail view, see Figure 95, Figure 96 and Figure 97.

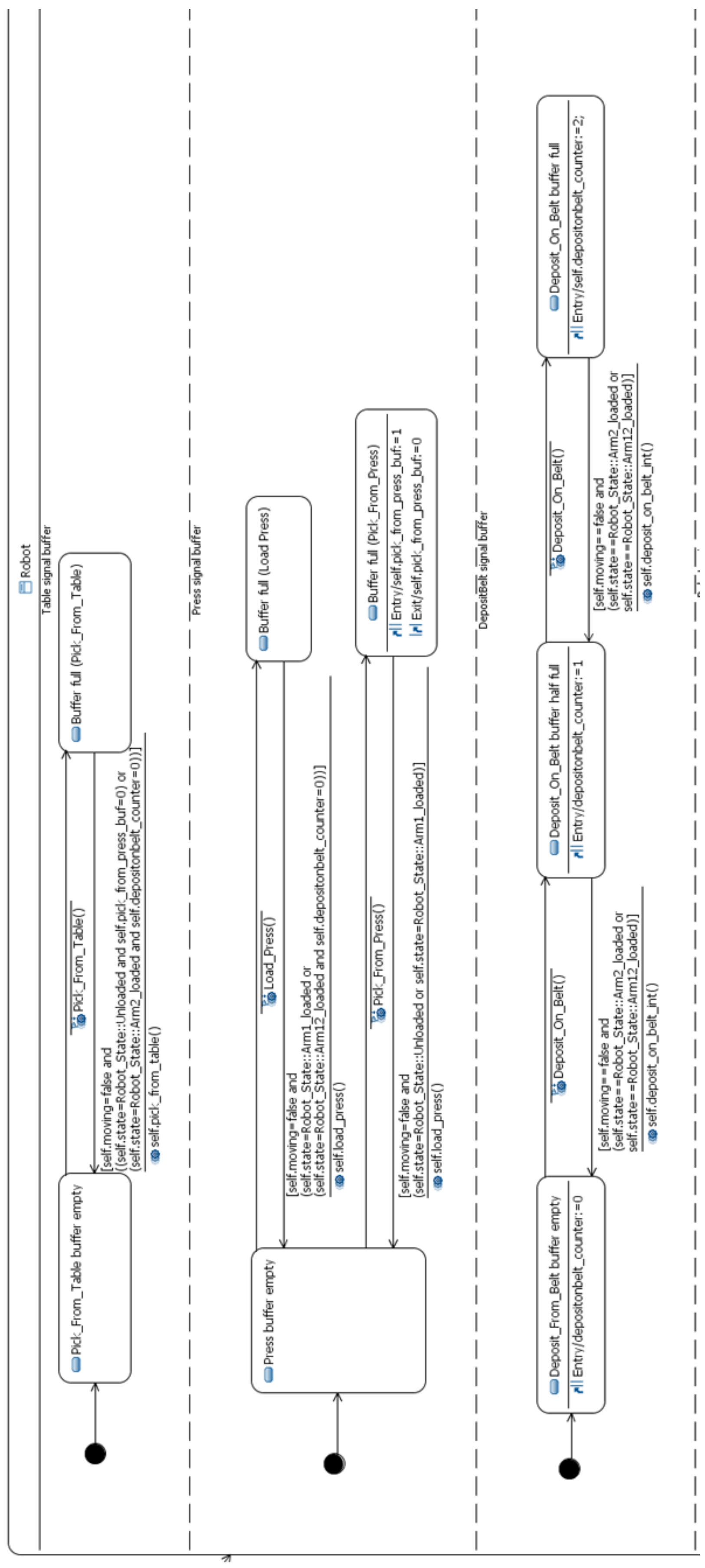
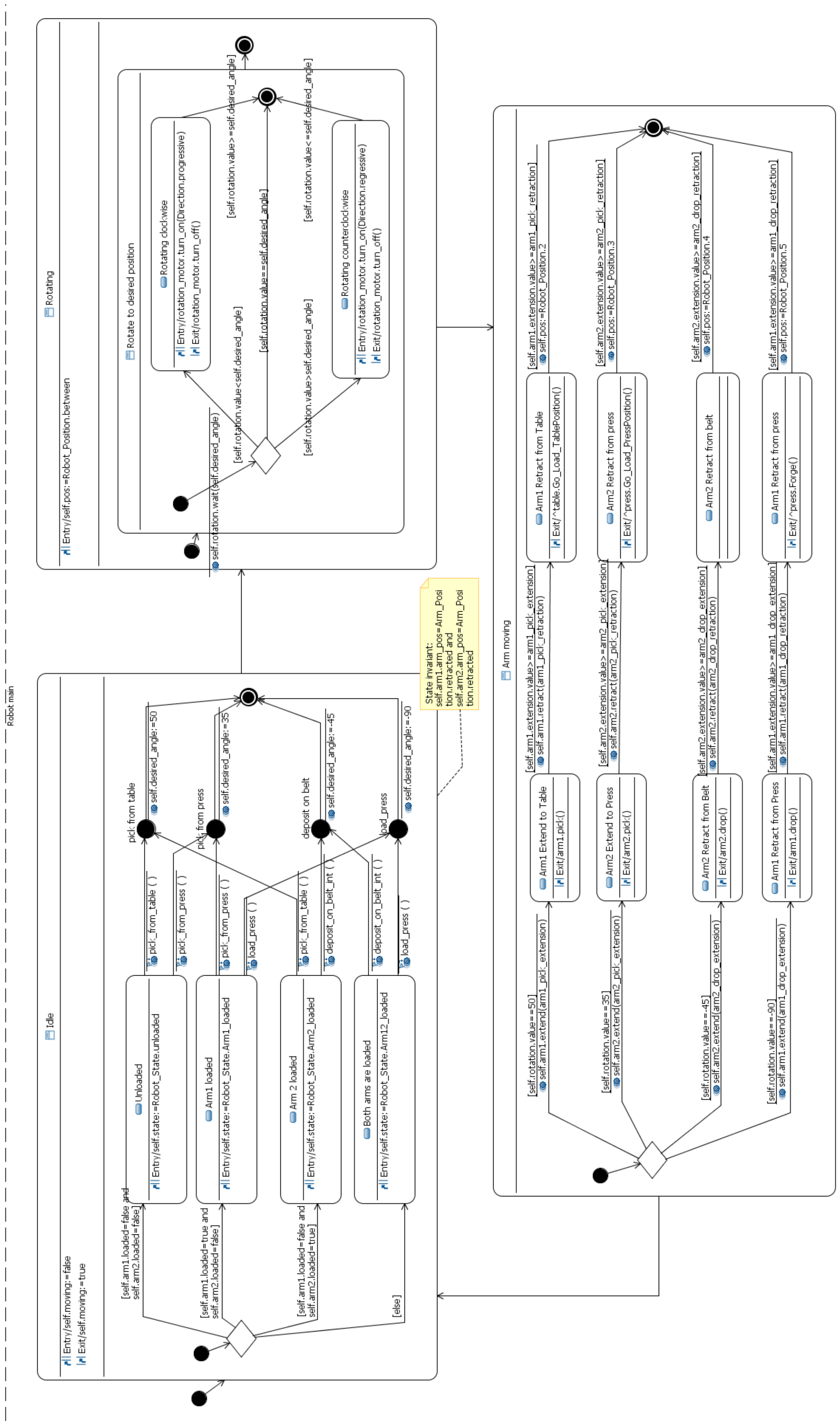

Robot
Table signal buffer
Pick_From_Table buffer empty
Buffer full (Pick_From_Table)
Pick_From_Table()
[self.moving=false and (self.state=Robot_State::Unloaded and self.pick_from_press_buf=0) or (self.state=Robot_State::Arm2_loaded and self.depositonbelt_counter=0))]
self.pick_from_table()
Press signal buffer
Press buffer empty
Buffer full (Load Press)
Load_Press()
[self.moving=false and (self.state=Robot_State::Arm1_loaded or (self.state=Robot_State::Arm12_loaded and self.depositonbelt_counter=0))]
self.load_press()
Buffer full (Pick_From_Press)
Entry/self.pick_from_press_buf:=1
Exit/self.pick_from_press_buf:=0
Pick_From_Press()
[self.moving=false and (self.state=Robot_State::Unloaded or self.state=Robot_State::Arm1_loaded)]
self.load_press()
DepositBelt signal buffer
Deposit_From_Belt buffer empty
Entry/depositonbelt_counter:=0
Deposit_On_Belt buffer half full
Entry/depositonbelt_counter:=1
Deposit_On_Belt buffer full
Entry/self.depositonbelt_counter:=2;
Deposit_On_Belt()
[self.moving==false and (self.state==Robot_State::Arm2_loaded or self.state==Robot_State::Arm12_loaded)]
self.deposit_on_belt_int()
Deposit_On_Belt()
[self.moving==false and (self.state==Robot_State::Arm2_loaded or self.state==Robot_State::Arm12_loaded)]
self.deposit_on_belt_int()

Figure 95 – Detailed view of buffer regions from stm [Block] Robot [Operating]. Whole diagram in Figure 94.



Robot main
Idle
Entry/self.moving:=false
Exit/self.moving:=true
Unloaded
Entry/self.state:=Robot_State.unloaded
Arm1 loaded
Entry/self.state:=Robot_State.Arm1_loaded
Arm 2 loaded
Entry/self.state:=Robot_State.Arm2_loaded
Both arms are loaded
Entry/self.state:=Robot_State.Arm12_loaded
[self.arm1.loaded=false and self.arm2.loaded=false]
[self.arm1.loaded=true and self.arm2.loaded=false]
[self.arm1.loaded=false and self.arm2.loaded=true]
[else]
pick_from_table ( )
pick_from_press ( )
load_press ( )
deposit_on_belt_int ( )
pick from table
pick from press
deposit on belt
load_press
self.desired_angle:=50
self.desired_angle:=35
self.desired_angle:=-45
self.desired_angle:=-90
State invariant: self.arm1.arm_pos=Arm_Position.retracted and self.arm2.arm_pos=Arm_Position.retracted
Rotating
Entry/self.pos:=Robot_Position.between
Rotate to desired position
Rotating clockwise
Entry/rotation_motor.turn_on(Direction.progressive)
Exit/rotation_motor.turn_off()
Rotating counterclockwise
Entry/rotation_motor.turn_on(Direction.regressive)
Exit/rotation_motor.turn_off()
[self.rotation.value<self.desired_angle]
[self.rotation.value==self.desired_angle]
[self.rotation.value>self.desired_angle]
[self.rotation.value>=self.desired_angle]
[self.rotation.value<=self.desired_angle]
self.rotation.wait(self.desired_angle)
Arm moving
Arm1 Extend to Table
Exit/arm1.pick()
Arm2 Extend to Press
Exit/arm2.pick()
Arm2 Retract from Belt
Exit/arm2.drop()
Arm1 Retract from Press
Exit/arm1.drop()
Arm1 Retract from Table
Exit/^table.Go_Load_TablePosition()
Arm2 Retract from press
Exit/^press.Go_Load_PressPosition()
Arm2 Retract from belt
Arm1 Retract from press
Exit/^press.Forge()
[self.rotation.value==50]
self.arm1.extend(arm1_pick_extension)
[self.rotation.value==35]
self.arm2.extend(arm2_pick_extension)
[self.rotation.value==-45]
self.arm2.extend(arm2_drop_extension)
[self.rotation.value==-90]
self.arm1.extend(arm1_drop_extension)
[self.arm1.extension.value>=arm1_pick_extension]
self.arm1.retract(arm1_pick_retraction)
[self.arm2.extension.value>=arm2_pick_extension]
self.arm2.retract(arm2_pick_retraction)
[self.arm2.extension.value>=arm2_drop_extension]
self.arm2.retract(arm2_drop_retraction)
[self.arm1.extension.value>=arm1_drop_extension]
self.arm1.retract(arm1_drop_retraction)
[self.arm1.extension.value>=arm1_pick_retraction]
self.pos:=Robot_Position.2
[self.arm2.extension.value>=arm2_pick_retraction]
self.pos:=Robot_Position.3
[self.arm2.extension.value>=arm2_drop_retraction]
self.pos:=Robot_Position.4
[self.arm1.extension.value>=arm1_drop_retraction]
self.pos:=Robot_Position.5

Figure 96 – Detailed view of main region from stm [Block] Robot [Operating]. Whole diagram in Figure 94.

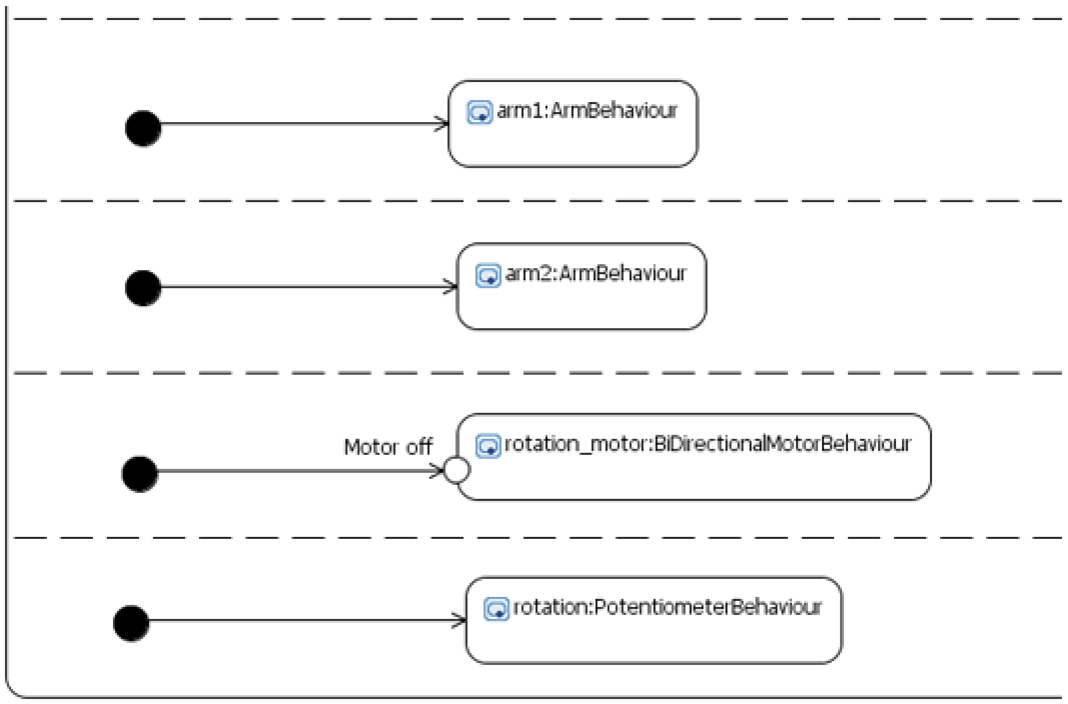

arm1:ArmBehaviour
arm2:ArmBehaviour
Motor off
rotation_motor:BiDirectionalMotorBehaviour
rotation:PotentiometerBehaviour

Figure 97 - Detailed view of part regions from stm [Block] Robot [Operating]. Whole diagram in Figure 91.

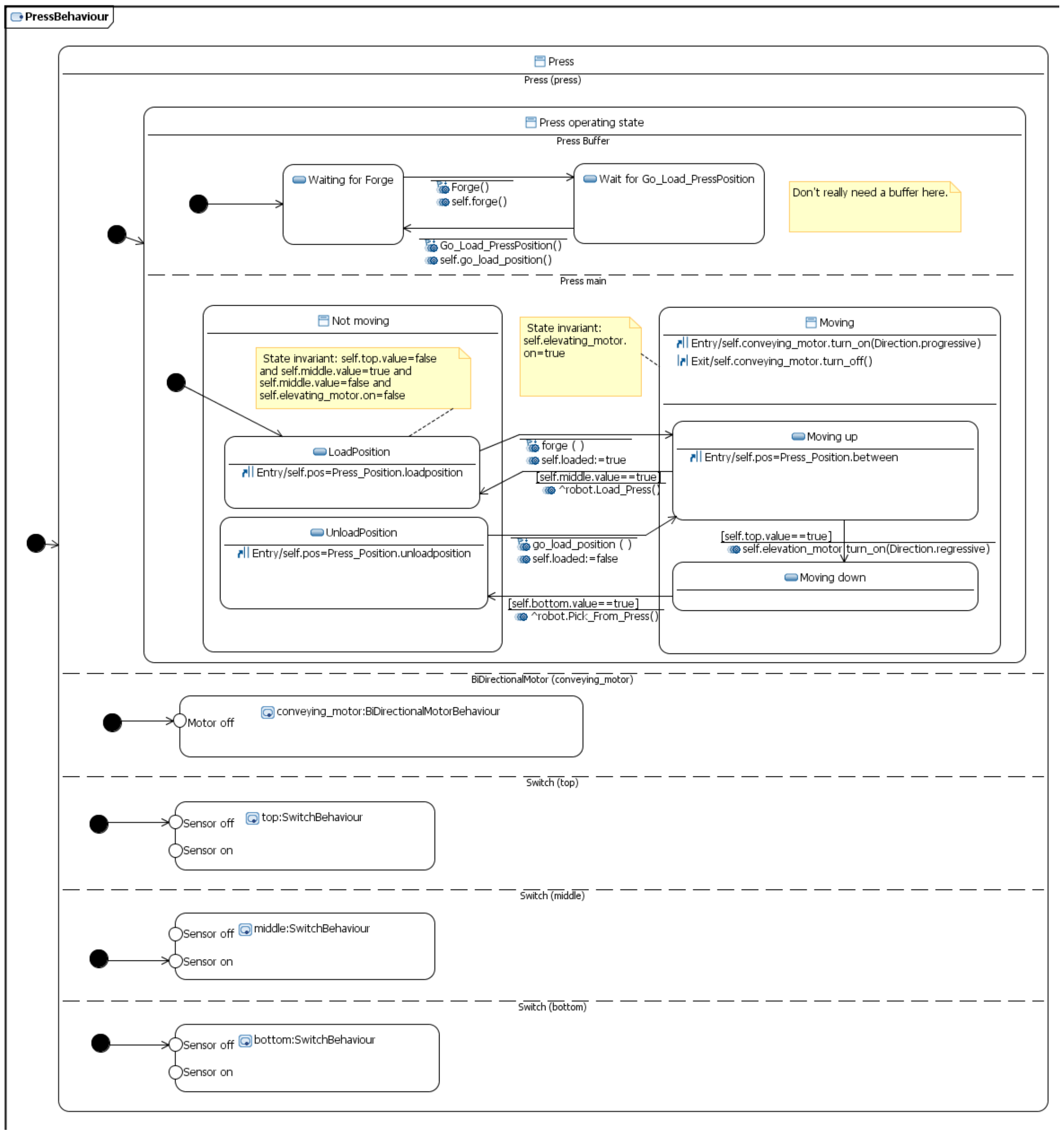

PressBehaviour
Press
Press (press)
Press operating state
Press Buffer
Waiting for Forge
Forge()
self.forge()
Wait for Go_Load_PressPosition
Don't really need a buffer here.
Go_Load_PressPosition()
self.go_load_position()
Press main
Not moving
State invariant: self.top.value=false and self.middle.value=true and self.middle.value=false and self.elevating_motor.on=false
State invariant: self.elevating_motor.on=true
Moving
Entry/self.conveying_motor.turn_on(Direction.progressive)
Exit/self.conveying_motor.turn_off()
LoadPosition
Entry/self.pos=Press_Position.loadposition
forge ( )
self.loaded:=true
[self.middle.value==true]
^robot.Load_Press()
Moving up
Entry/self.pos=Press_Position.between
UnloadPosition
Entry/self.pos=Press_Position.unloadposition
go_load_position ( )
self.loaded:=false
[self.top.value==true]
self.elevation_motor.turn_on(Direction.regressive)
Moving down
[self.bottom.value==true]
^robot.Pick_From_Press()
BiDirectionalMotor (conveying_motor)
Motor off
conveying_motor:BiDirectionalMotorBehaviour
Switch (top)
Sensor off
Sensor on
top:SwitchBehaviour
Switch (middle)
Sensor off
Sensor on
middle:SwitchBehaviour
Switch (bottom)
Sensor off
Sensor on
bottom:SwitchBehaviour

Figure 98 - stm [Block] Press [Operating]

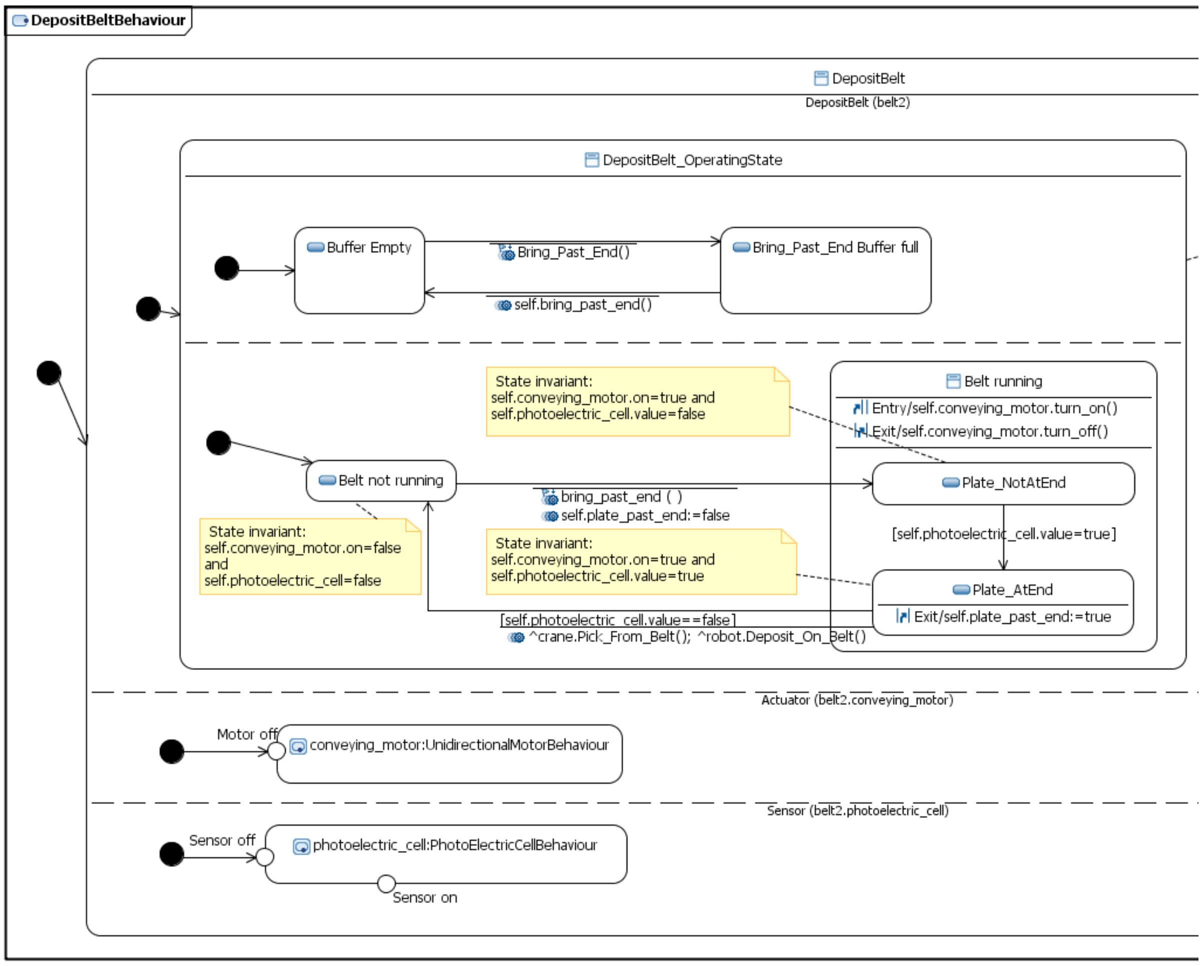

DepositBeltBehaviour
DepositBelt
DepositBelt (belt2)
DepositBelt_OperatingState
Buffer Empty
Bring_Past_End()
Bring_Past_End Buffer full
self.bring_past_end()
State invariant:
self.conveying_motor.on=true and
self.photoelectric_cell.value=false
Belt running
Entry/self.conveying_motor.turn_on()
Exit/self.conveying_motor.turn_off()
Belt not running
bring_past_end ( )
self.plate_past_end:=false
Plate_NotAtEnd
[self.photoelectric_cell.value=true]
State invariant:
self.conveying_motor.on=false
and
self.photoelectric_cell=false
State invariant:
self.conveying_motor.on=true and
self.photoelectric_cell.value=true
Plate_AtEnd
Exit/self.plate_past_end:=true
[self.photoelectric_cell.value==false]
^crane.Pick_From_Belt(); ^robot.Deposit_On_Belt()
Actuator (belt2.conveying_motor)
Motor off
conveying_motor:UnidirectionalMotorBehaviour
Sensor (belt2.photoelectric_cell)
Sensor off
photoelectric_cell:PhotoElectricCellBehaviour
Sensor on

Figure 99 - stm [Block] DepositBelt [Operating]

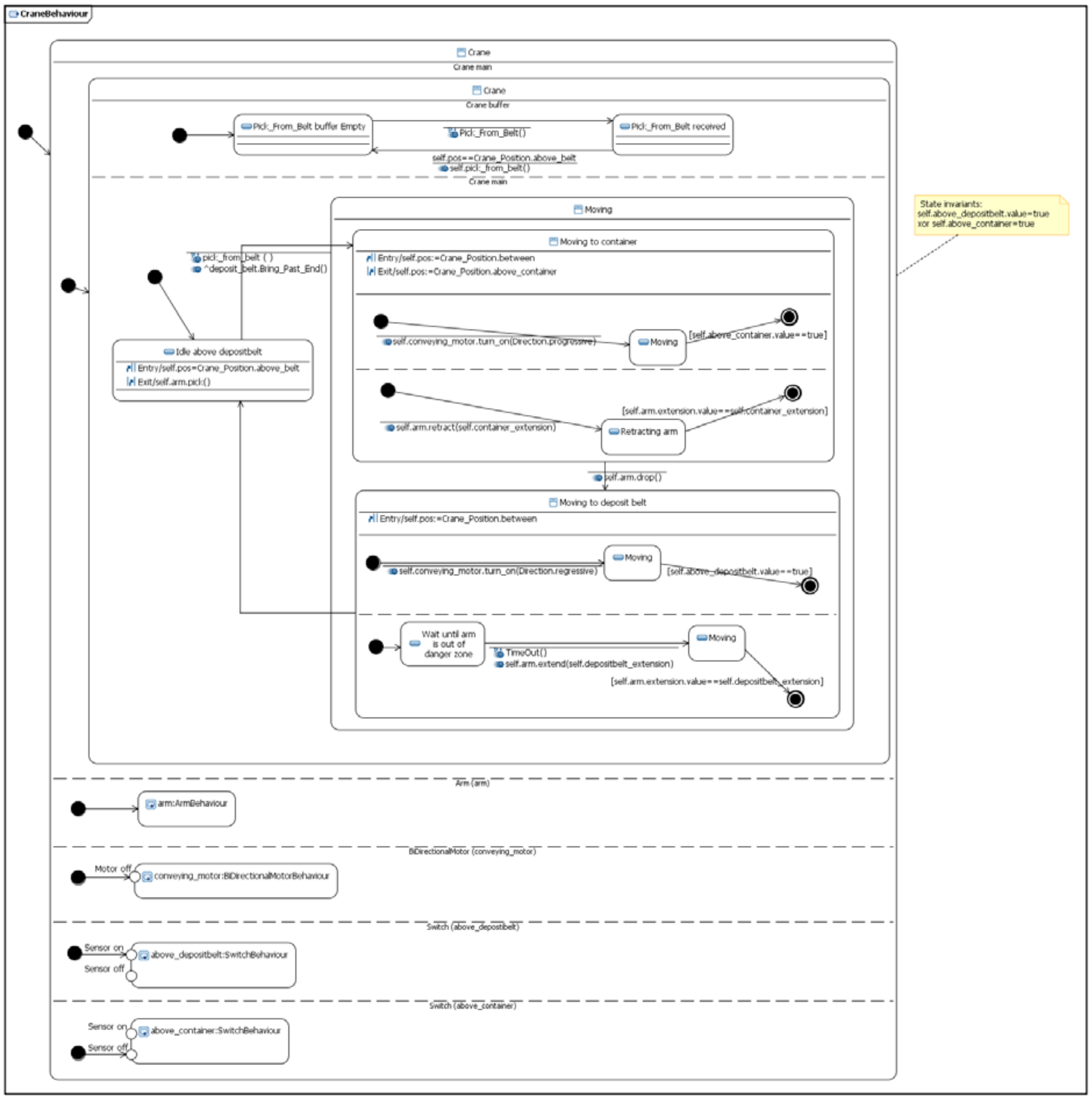

CraneBehaviour
Crane
Crane main
Crane
Crane buffer
Pick_From_Belt buffer Empty
Pick_From_Belt()
Pick_From_Belt received
self.pos==Crane_Position.above_belt
self.pick_from_belt()
Crane main
Moving
Moving to container
Entry/self.pos:=Crane_Position.between
Exit/self.pos:=Crane_Position.above_container
pick_from_belt ( )
^deposit_belt.Bring_Past_End()
Idle above depositbelt
Entry/self.pos=Crane_Position.above_belt
Exit/self.arm.pick()
self.conveying_motor.turn_on(Direction.progressive)
Moving
[self.above_container.value==true]
[self.arm.extension.value==self.container_extension]
self.arm.retract(self.container_extension)
Retracting arm
self.arm.drop()
Moving to deposit belt
Entry/self.pos:=Crane_Position.between
Moving
self.conveying_motor.turn_on(Direction.regressive)
[self.above_depositbelt.value==true]
Wait until arm is out of danger zone
Moving
TimeOut()
self.arm.extend(self.depositbelt_extension)
[self.arm.extension.value==self.depositbelt_extension]
Arm (arm)
arm:ArmBehaviour
BiDirectionalMotor (conveying_motor)
Motor off
conveying_motor:BiDirectionalMotorBehaviour
Switch (above_depostbelt)
Sensor on
Sensor off
above_depositbelt:SwitchBehaviour
Switch (above_container)
Sensor on
Sensor off
above_container:SwitchBehaviour
State invariants:
self.above_depositbelt.value=true
xor self.above_container=true

Figure 100 - stm [Block] Crane [Operating], for details, see Figure 101 and Figure 102.

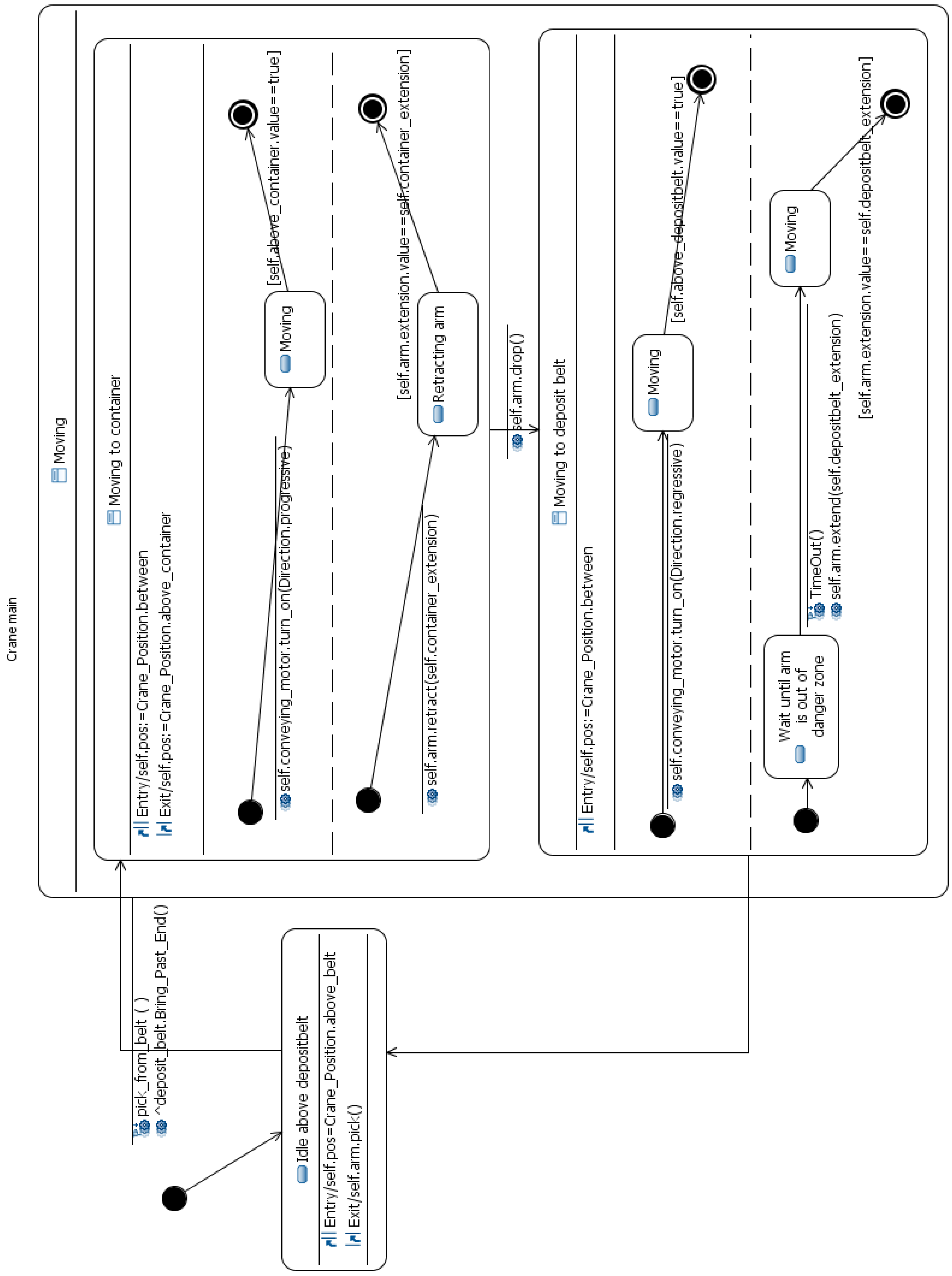

Crane main
Moving
Moving to container
Entry/self.pos:=Crane_Position.between
Exit/self.pos:=Crane_Position.above_container
self.conveying_motor.turn_on(Direction.progressive)
Moving
[self.above_container.value==true]
self.arm.retract(self.container_extension)
Retracting arm
[self.arm.extension.value==self.container_extension]
self.arm.drop()
Moving to deposit belt
Entry/self.pos:=Crane_Position.between
self.conveying_motor.turn_on(Direction.regressive)
Moving
[self.above_depositbelt.value==true]
Wait until arm is out of danger zone
TimeOut()
self.arm.extend(self.depositbelt_extension)
Moving
[self.arm.extension.value==self.depositbelt_extension]
pick_from_belt ( )
^deposit_belt.Bring_Past_End()
Idle above depositbelt
Entry/self.pos=Crane_Position.above_belt
Exit/self.arm.pick()

Figure 101 – Detailed view of Crane buffer and main regions from state machine diagram in Figure 100.

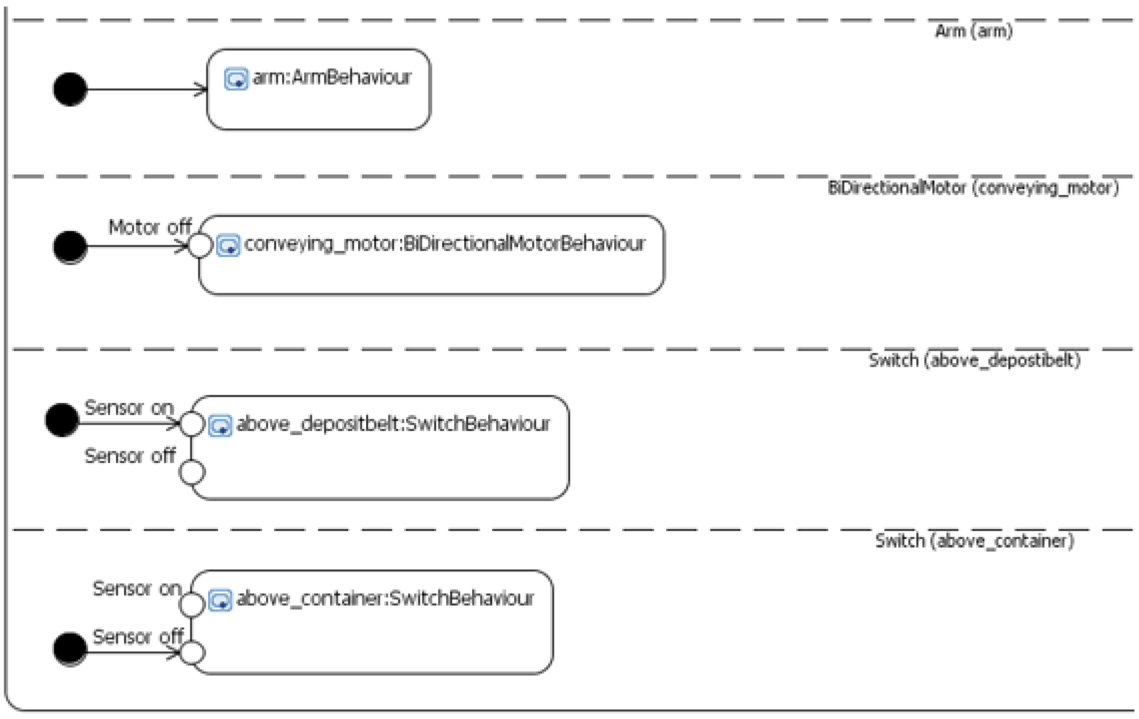

Arm (arm)
arm:ArmBehaviour
BiDirectionalMotor (conveying_motor)
Motor off
conveying_motor:BiDirectionalMotorBehaviour
Switch (above_depostibelt)
Sensor on
Sensor off
above_depositbelt:SwitchBehaviour
Switch (above_container)
Sensor on
Sensor off
above_container:SwitchBehaviour

Figure 102 - Detailed view of Crane part regions from state machine diagram in Figure 100.

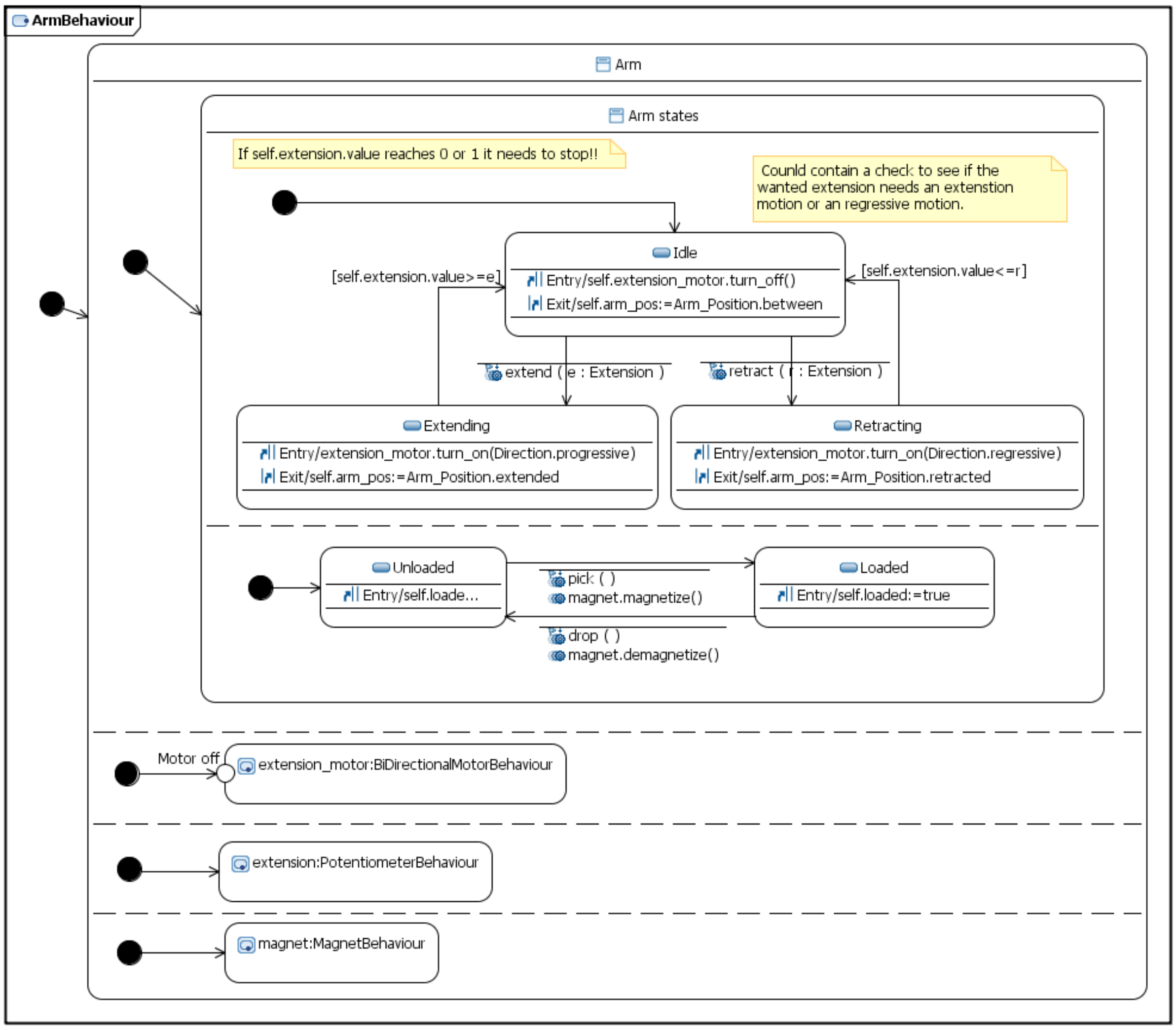

ArmBehaviour
Arm
Arm states
If self.extension.value reaches 0 or 1 it needs to stop!!
Counld contain a check to see if the wanted extension needs an extenstion motion or an regressive motion.
Idle
Entry/self.extension_motor.turn_off()
Exit/self.arm_pos:=Arm_Position.between
[self.extension.value>=e]
[self.extension.value<=r]
extend (e : Extension)
retract (r : Extension)
Extending
Entry/extension_motor.turn_on(Direction.progressive)
Exit/self.arm_pos:=Arm_Position.extended
Retracting
Entry/extension_motor.turn_on(Direction.regressive)
Exit/self.arm_pos:=Arm_Position.retracted
Unloaded
Entry/self.loade...
pick ( )
magnet.magnetize()
Loaded
Entry/self.loaded:=true
drop ( )
magnet.demagnetize()
Motor off
extension_motor:BiDirectionalMotorBehaviour
extension:PotentiometerBehaviour
magnet:MagnetBehaviour

Figure 103 - stm [Block] Arm [Operating]

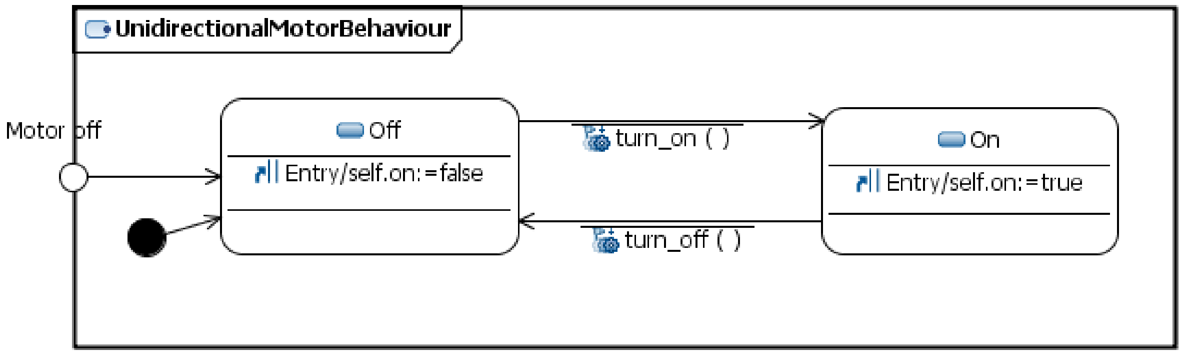


Figure 104 - stm [Block] UniDirectionalMotor [Behaviour]

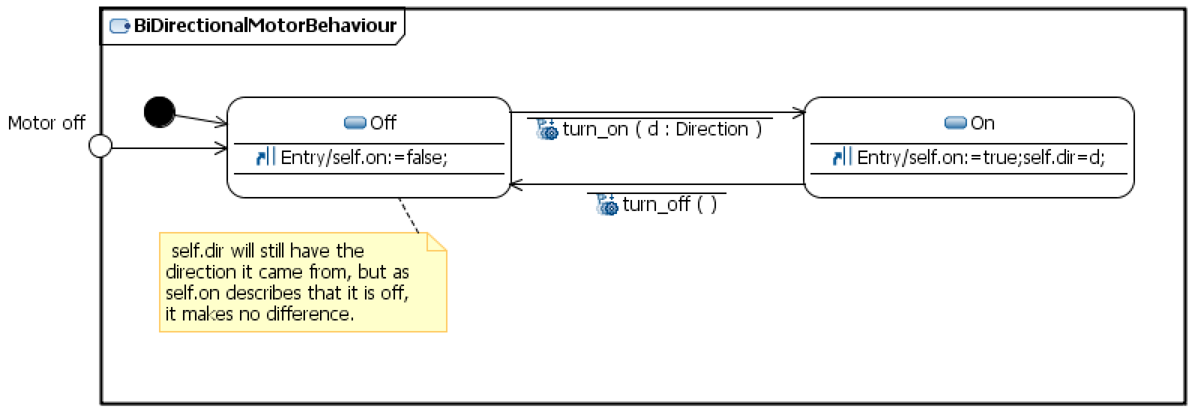

BiDirectionalMotorBehaviour
Motor off
Off
Entry/self.on:=false;
turn_on (d : Direction)
turn_off ( )
On
Entry/self.on:=true;self.dir=d;
self.dir will still have the direction it came from, but as self.on describes that it is off, it makes no difference.

Figure 105 - stm [Block] BiDirectionalMotor [Behaviour]

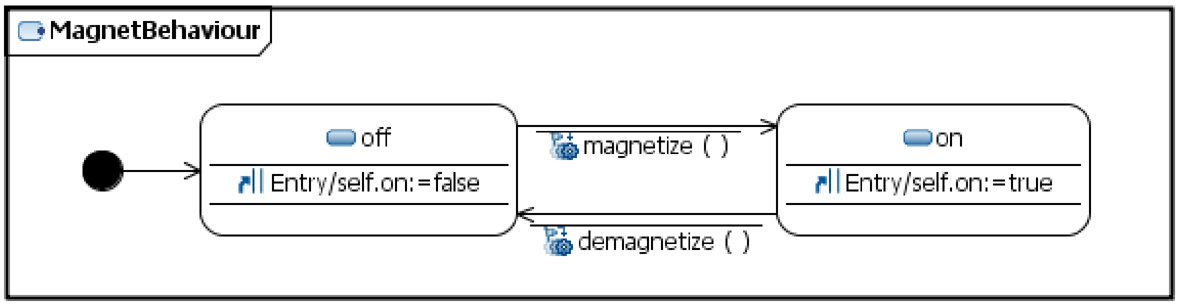


Figure 106 - stm [Block] Magnet [Behaviour]

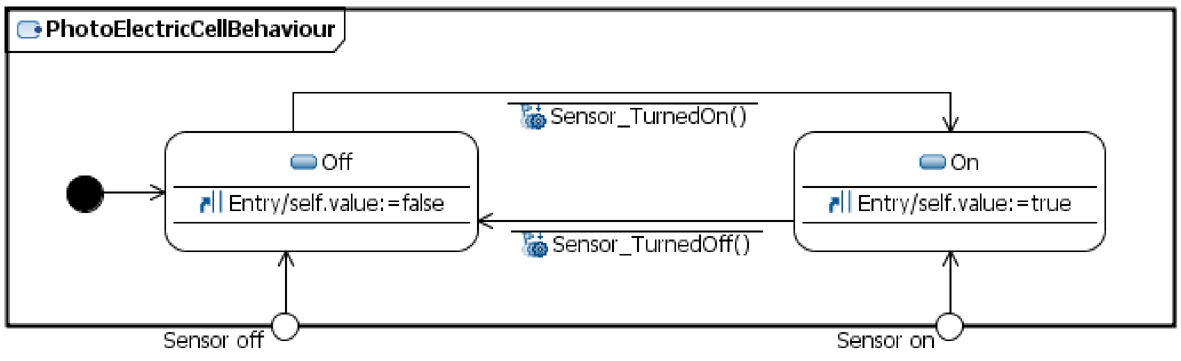


Figure 107 - stm [Block] PhotoElectricCell [Behaviour]

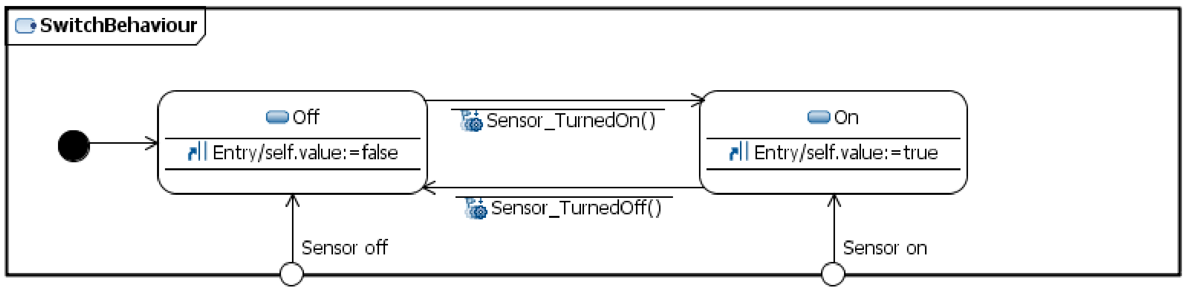


Figure 108 - stm [Block] Switch [Behaviour]

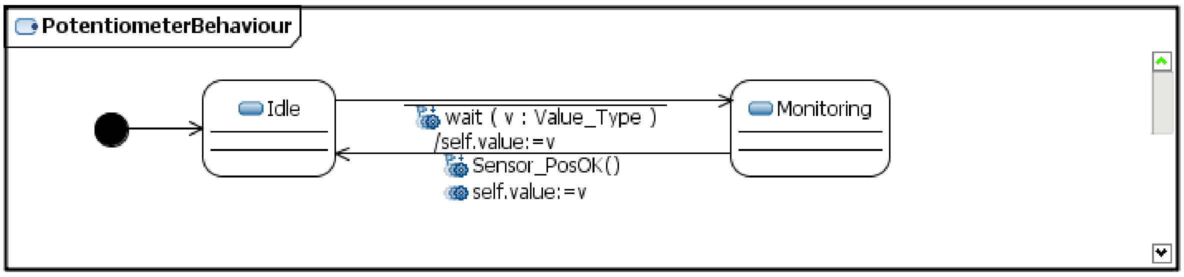
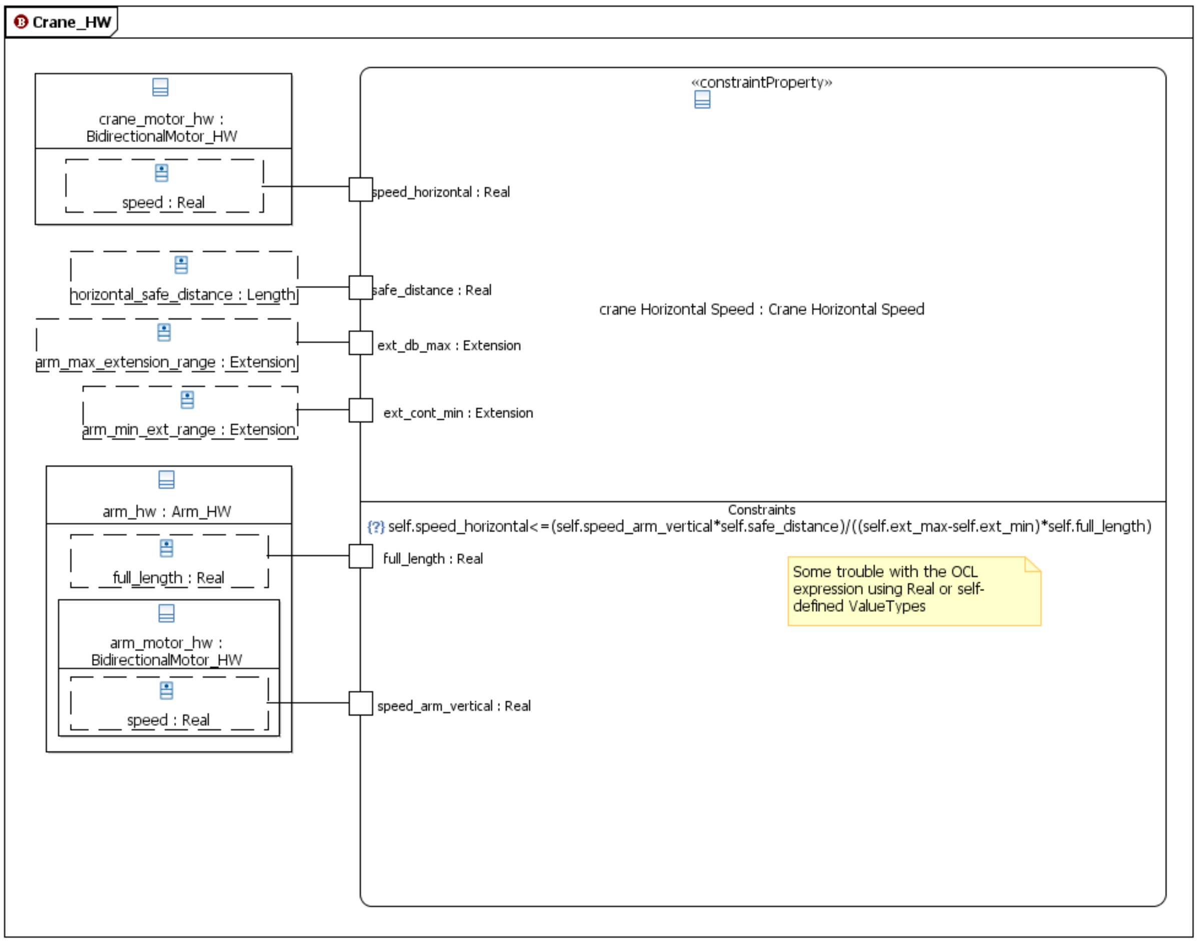


Figure 109 - stm [Block] Potentiometer [Behaviour]

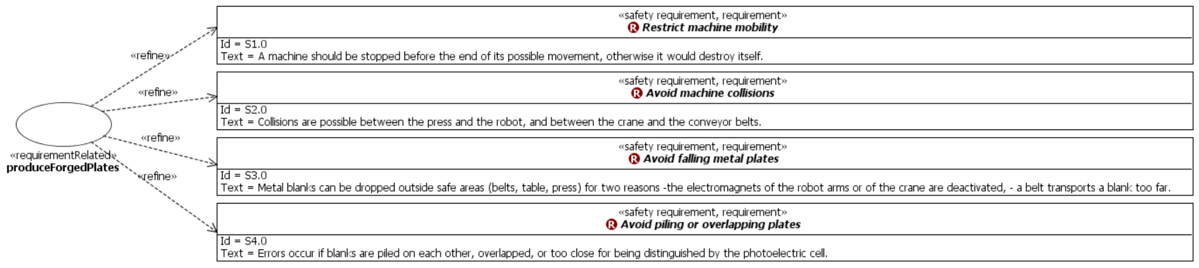
A.8 Parametric diagram



Crane_HW
crane_motor_hw : BidirectionalMotor_HW
speed : Real
horizontal_safe_distance : Length
arm_max_extension_range : Extension
arm_min_ext_range : Extension
arm_hw : Arm_HW
full_length : Real
arm_motor_hw : BidirectionalMotor_HW
speed : Real
«constraintProperty»
crane Horizontal Speed : Crane Horizontal Speed
speed_horizontal : Real
safe_distance : Real
ext_db_max : Extension
ext_cont_min : Extension
Constraints
{?} self.speed_horizontal<=(self.speed_arm_vertical*self.safe_distance)/((self.ext_max-self.ext_min)*self.full_length)
full_length : Real
speed_arm_vertical : Real
Some trouble with the OCL expression using Real or self-defined ValueTypes

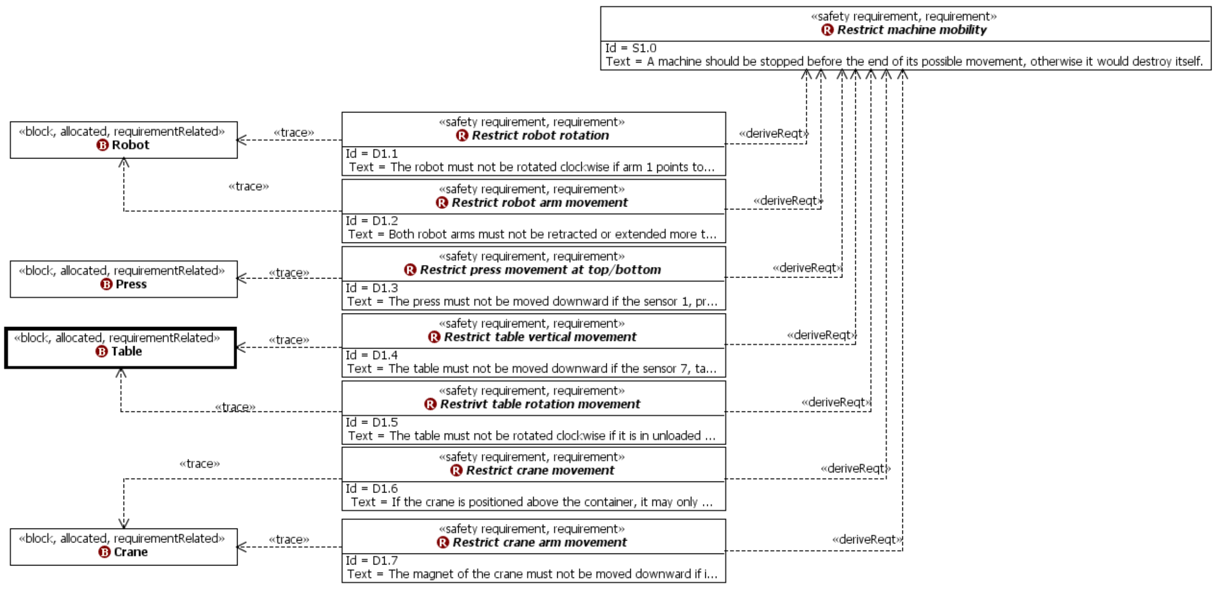
Figure 110 - par [Block] Crane_HW [Crane Horizontal Speed]

A.9 Cross-cutting diagrams



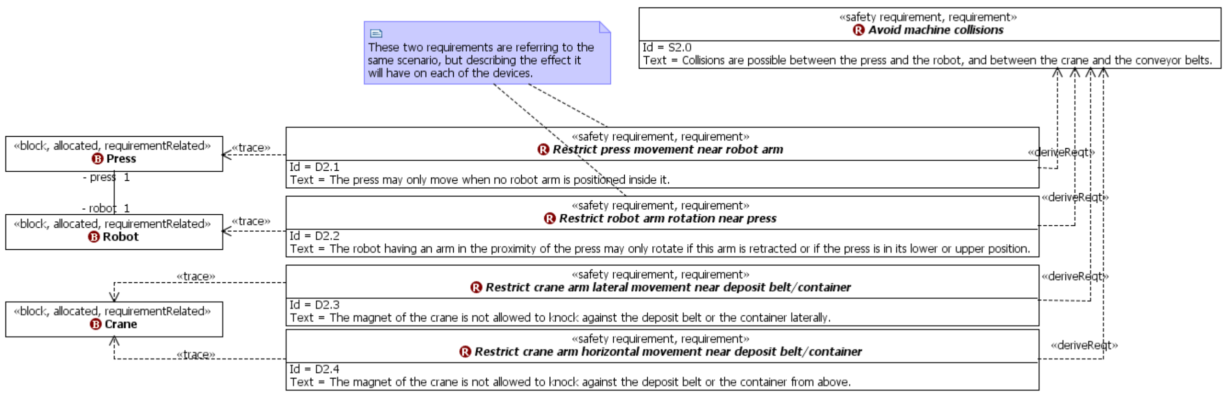
«safety requirement, requirement»
Restrict machine mobility
Id = S1.0
Text = A machine should be stopped before the end of its possible movement, otherwise it would destroy itself.
«safety requirement, requirement»
Avoid machine collisions
Id = S2.0
Text = Collisions are possible between the press and the robot, and between the crane and the conveyor belts.
«safety requirement, requirement»
Avoid falling metal plates
Id = S3.0
Text = Metal blanks can be dropped outside safe areas (belts, table, press) for two reasons -the electromagnets of the robot arms or of the crane are deactivated, - a belt transports a blank too far.
«safety requirement, requirement»
Avoid piling or overlapping plates
Id = S4.0
Text = Errors occur if blanks are piled on each other, overlapped, or too close for being distinguished by the photoelectric cell.
«refine»
«refine»
«refine»
«refine»
«requirementRelated»
produceForgedPlates

Figure 111 - High level requirements refined from use case.



«safety requirement, requirement»
Restrict machine mobility
Id = S1.0
Text = A machine should be stopped before the end of its possible movement, otherwise it would destroy itself.
«block, allocated, requirementRelated»
Robot
«trace»
«safety requirement, requirement»
Restrict robot rotation
Id = D1.1
Text = The robot must not be rotated clockwise if arm 1 points to...
«deriveReqt»
«trace»
«safety requirement, requirement»
Restrict robot arm movement
Id = D1.2
Text = Both robot arms must not be retracted or extended more t...
«deriveReqt»
«block, allocated, requirementRelated»
Press
«trace»
«safety requirement, requirement»
Restrict press movement at top/bottom
Id = D1.3
Text = The press must not be moved downward if the sensor 1, pr...
«deriveReqt»
«block, allocated, requirementRelated»
Table
«trace»
«safety requirement, requirement»
Restrict table vertical movement
Id = D1.4
Text = The table must not be moved downward if the sensor 7, ta...
«deriveReqt»
«trace»
«safety requirement, requirement»
Restrivt table rotation movement
Id = D1.5
Text = The table must not be rotated clockwise if it is in unloaded ...
«deriveReqt»
«trace»
«safety requirement, requirement»
Restrict crane movement
Id = D1.6
Text = If the crane is positioned above the container, it may only ...
«deriveReqt»
«block, allocated, requirementRelated»
Crane
«trace»
«safety requirement, requirement»
Restrict crane arm movement
Id = D1.7
Text = The magnet of the crane must not be moved downward if i...
«deriveReqt»

Figure 112 - Requirement S1.0 with derived requirements and traces to top-level blocks.



These two requirements are referring to the same scenario, but describing the effect it will have on each of the devices.
«safety requirement, requirement»
Avoid machine collisions
Id = S2.0
Text = Collisions are possible between the press and the robot, and between the crane and the conveyor belts.
«block, allocated, requirementRelated»
Press
«trace»
«safety requirement, requirement»
Restrict press movement near robot arm
Id = D2.1
Text = The press may only move when no robot arm is positioned inside it.
«deriveReqt»
- press 1
- robot 1
«block, allocated, requirementRelated»
Robot
«trace»
«safety requirement, requirement»
Restrict robot arm rotation near press
Id = D2.2
Text = The robot having an arm in the proximity of the press may only rotate if this arm is retracted or if the press is in its lower or upper position.
«deriveReqt»
«trace»
«safety requirement, requirement»
Restrict crane arm lateral movement near deposit belt/container
Id = D2.3
Text = The magnet of the crane is not allowed to knock against the deposit belt or the container laterally.
«deriveReqt»
«block, allocated, requirementRelated»
Crane
«trace»
«safety requirement, requirement»
Restrict crane arm horizontal movement near deposit belt/container
Id = D2.4
Text = The magnet of the crane is not allowed to knock against the deposit belt or the container from above.
«deriveReqt»

Figure 113 - Requirement S2.0 with derived requirements and traces to top-level blocks.

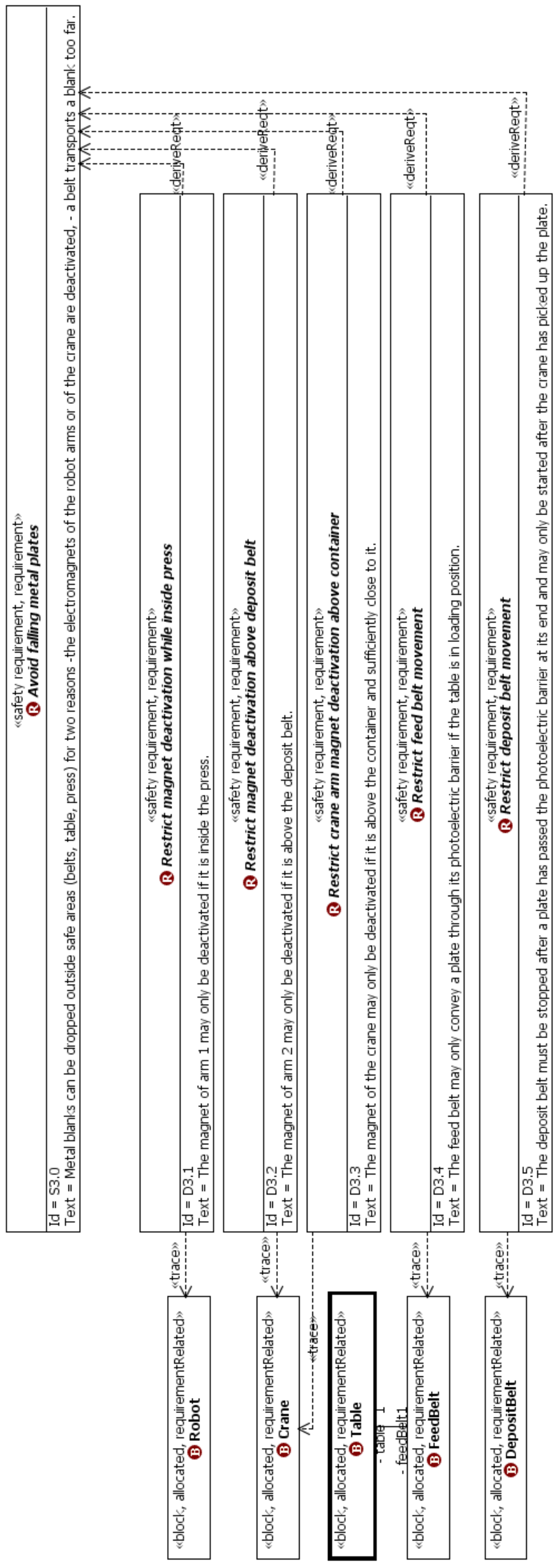

«safety requirement, requirement»
Avoid falling metal plates
Id = S3.0
Text = Metal blanks can be dropped outside safe areas (belts, table, press) for two reasons -the electromagnets of the robot arms or of the crane are deactivated, - a belt transports a blank too far.
«safety requirement, requirement»
Restrict magnet deactivation while inside press
Id = D3.1
Text = The magnet of arm 1 may only be deactivated if it is inside the press.
«safety requirement, requirement»
Restrict magnet deactivation above deposit belt
Id = D3.2
Text = The magnet of arm 2 may only be deactivated if it is above the deposit belt.
«safety requirement, requirement»
Restrict crane arm magnet deactivation above container
Id = D3.3
Text = The magnet of the crane may only be deactivated if it is above the container and sufficiently close to it.
«safety requirement, requirement»
Restrict feed belt movement
Id = D3.4
Text = The feed belt may only convey a plate through its photoelectric barrier if the table is in loading position.
«safety requirement, requirement»
Restrict deposit belt movement
Id = D3.5
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.
«deriveReqt»
«deriveReqt»
«deriveReqt»
«deriveReqt»
«deriveReqt»
«trace»
«trace»
«trace»
«trace»
«trace»
«block, allocated, requirementRelated»
Robot
«block, allocated, requirementRelated»
Crane
«block, allocated, requirementRelated»
Table
- table 1
- feedBelt1
«block, allocated, requirementRelated»
FeedBelt
«block, allocated, requirementRelated»
DepositBelt

Figure 114 - Requirement S3.0 with derived requirements and traces to top-level blocks.

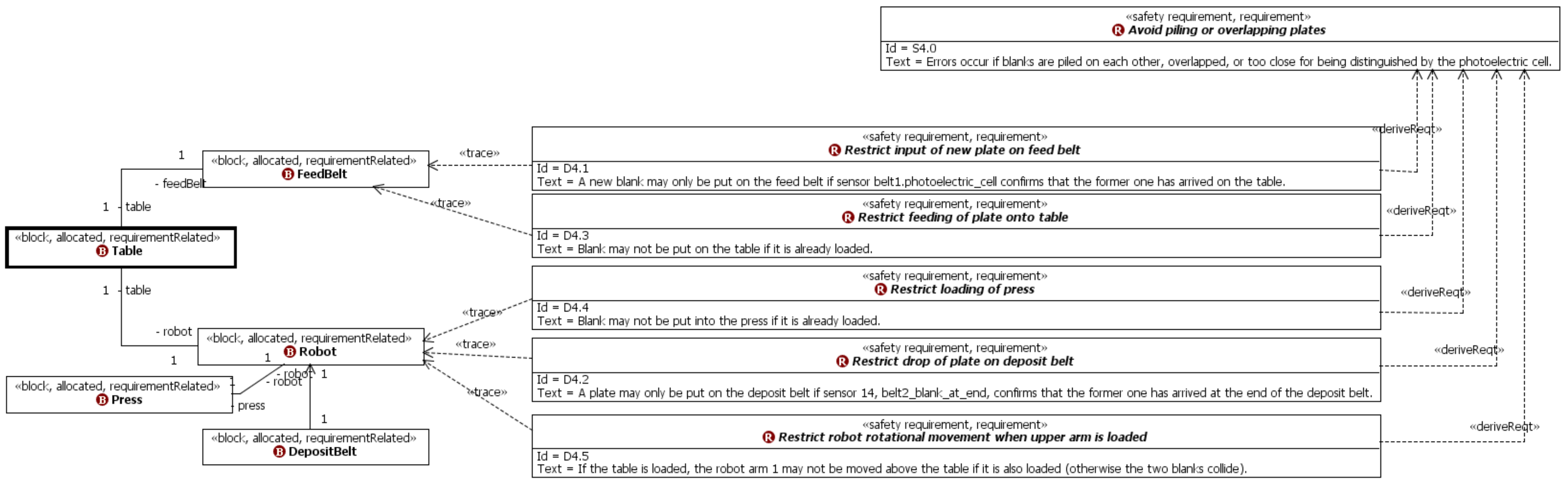

«safety requirement, requirement»
Avoid piling or overlapping plates
Id = S4.0
Text = Errors occur if blanks are piled on each other, overlapped, or too close for being distinguished by the photoelectric cell.
«safety requirement, requirement»
Restrict input of new plate on feed belt
Id = D4.1
Text = A new blank may only be put on the feed belt if sensor belt1.photoelectric_cell confirms that the former one has arrived on the table.
«safety requirement, requirement»
Restrict feeding of plate onto table
Id = D4.3
Text = Blank may not be put on the table if it is already loaded.
«safety requirement, requirement»
Restrict loading of press
Id = D4.4
Text = Blank may not be put into the press if it is already loaded.
«safety requirement, requirement»
Restrict drop of plate on deposit belt
Id = D4.2
Text = A plate may only be put on the deposit belt if sensor 14, belt2_blank_at_end, confirms that the former one has arrived at the end of the deposit belt.
«safety requirement, requirement»
Restrict robot rotational movement when upper arm is loaded
Id = D4.5
Text = If the table is loaded, the robot arm 1 may not be moved above the table if it is also loaded (otherwise the two blanks collide).
«deriveReqt»
«deriveReqt»
«deriveReqt»
«deriveReqt»
«deriveReqt»
«trace»
«trace»
«trace»
«trace»
«trace»
«block, allocated, requirementRelated»
FeedBelt
«block, allocated, requirementRelated»
Table
«block, allocated, requirementRelated»
Robot
«block, allocated, requirementRelated»
Press
«block, allocated, requirementRelated»
DepositBelt
1
- feedBelt
1
- table
1
- table
- robot
1
1
- robot
- robot
1
press
1

Figure 115 - Requirement S4.0 with derived requirements and traces to top-level blocks.

A.10 Safety analysis for selected requirements

The safety diagrams are very large in size, so we have chosen to show the diagrams in parts. For each example we will first show the diagram as a whole, and then we show a close-up of a main safety analysis part of the diagram, where the text is readable.
In each diagram, we show slices of relevant design diagrams. But we also give you links to the whole diagrams and explain which slices are extracted.

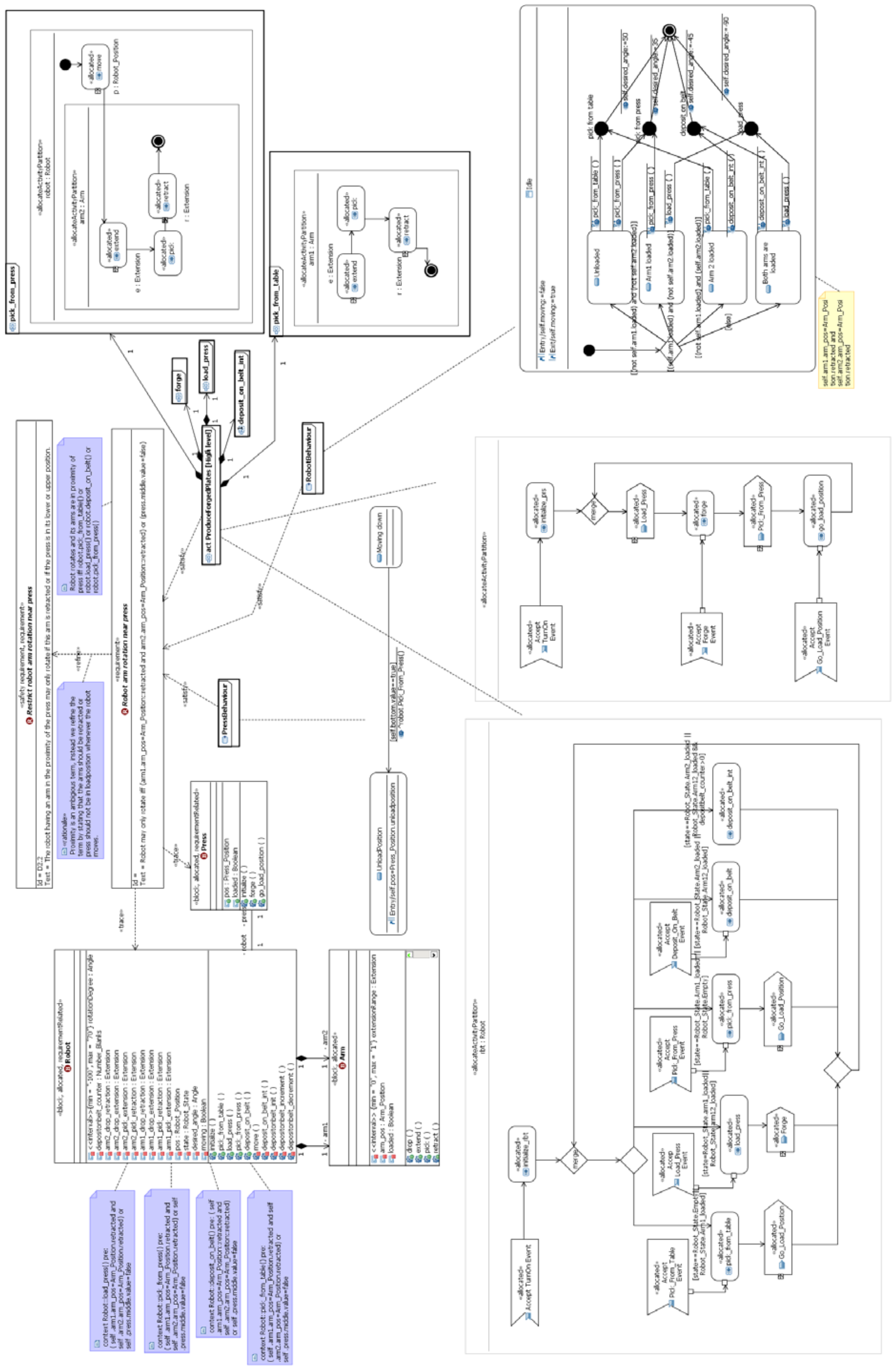
A.10.1 Safety analysis diagram for req 2.2.

The next two pages show the safety analysis diagram for requirement 2.2. The first image is the full image, the second shows a close-up view for the analysis.

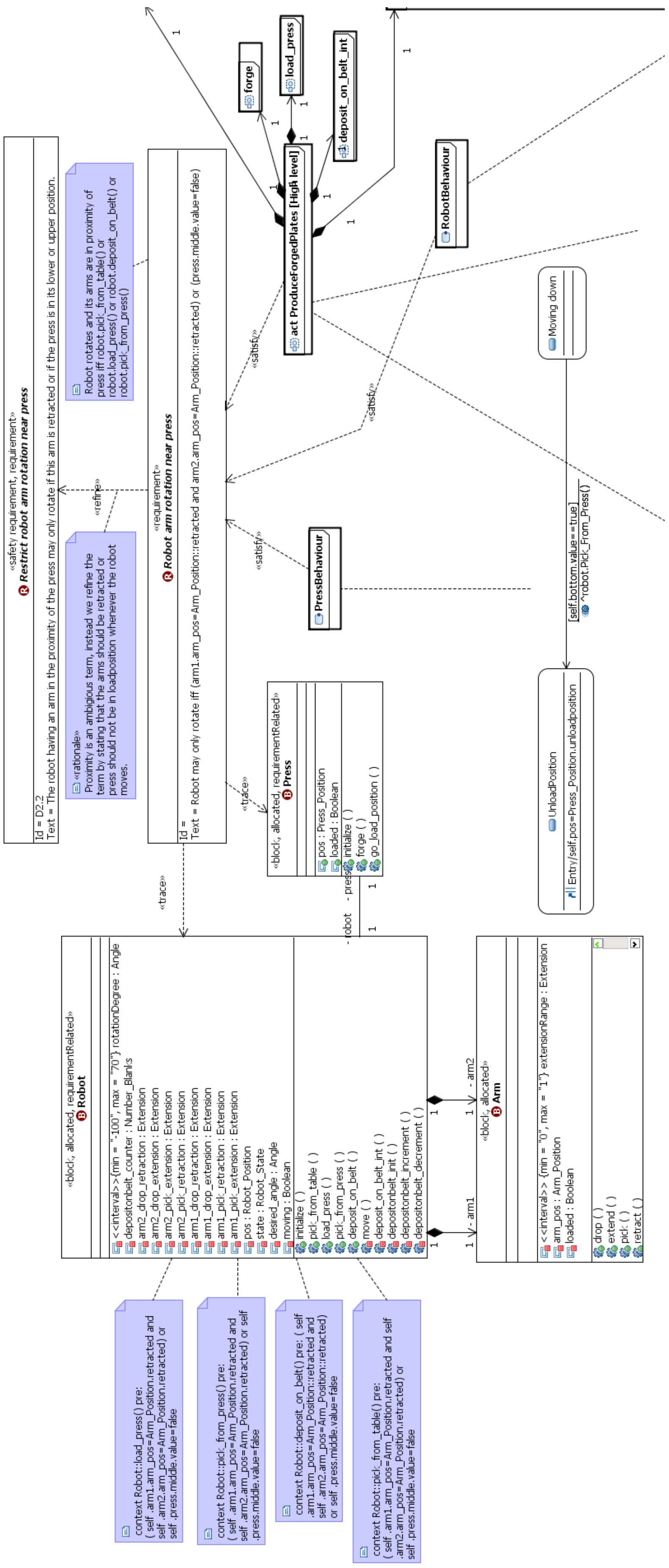
For detailed views of the activity diagrams:
Robot partition in Figure 62
Press partition in Figure 63.
Robot activity diagram, act pick_from_press, in Figure 72.
Robot activity diagram, act pick_from_table in Figure 70.

We do not show examples from the following diagrams, but they are part of the analysis.
Robot activity diagram, act deposit_on_belt_in in Figure 74.
Robot activity diagram, act load_press in Figure 71.
Press activity diagram, act forge in Figure 77.

For detailed views of the state machine diagrams:
Robot in Figure 96, slice: the idle state
Press in Figure 98, slice from main region: Moving Down and Unload Position states and the transition between them.



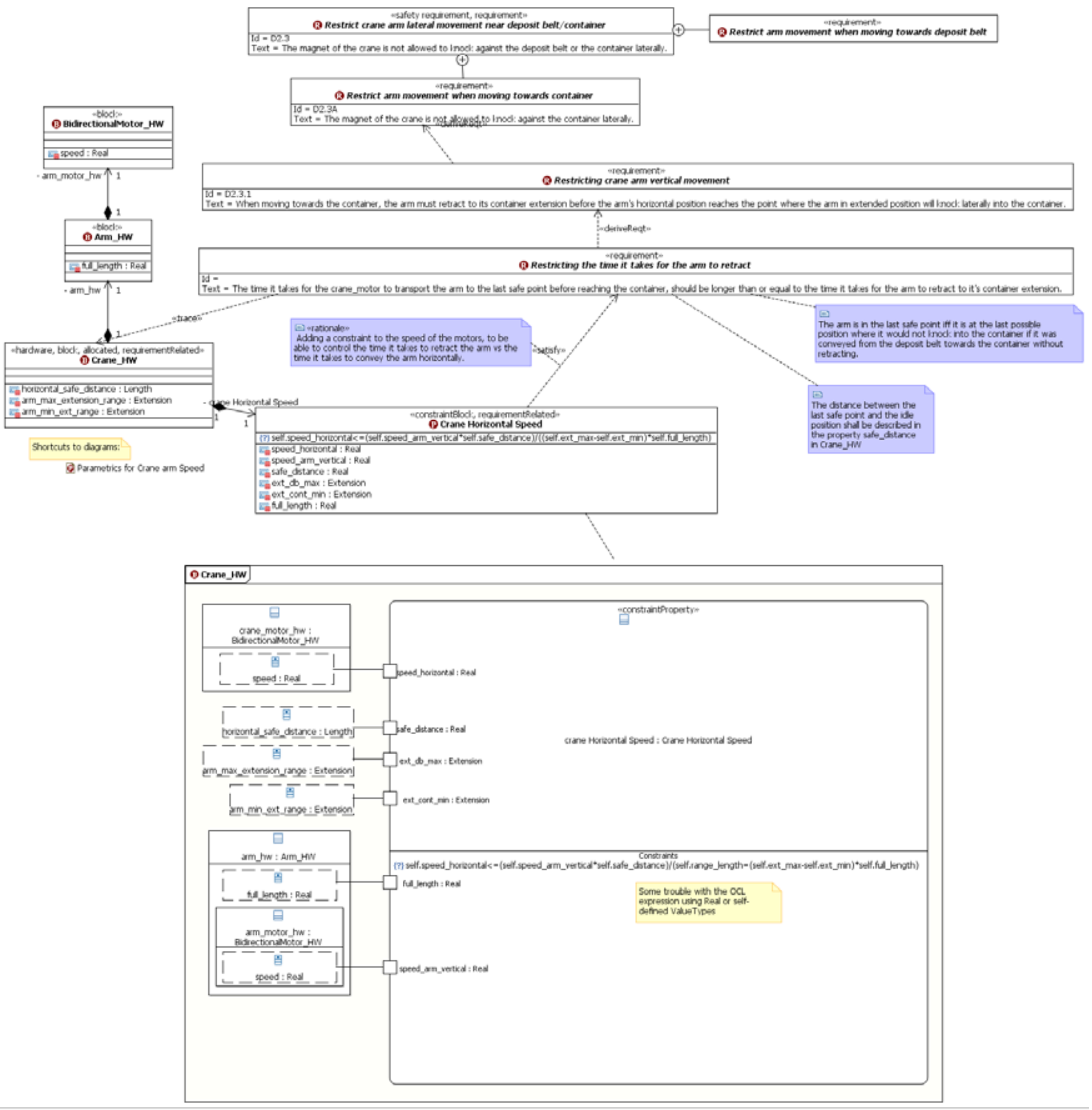
pick_from_press
pick_from_table
forge
load_press
deposit_on_belt_int
act ProduceForgedPlates [High level]
RobotBehaviour
PressBehaviour
«safety requirement, requirement» Restrict robot arm rotation near press
«requirement» Robot arm rotation near press
«rationale»
«refine»
«satisfy»
«trace»
«block, allocated, requirementRelated» Robot
«block, allocated, requirementRelated» Press
«block, allocated» Arm
Moving down
UnloadPosition
Idle
Unloaded
Arm1 loaded
Arm 2 loaded
Both arms are loaded
«allocateActivityPartition»
«allocateActivityPartition» rbt : Robot
Accept TurnOn Event
merge


«safety requirement, requirement»
Restrict robot arm rotation near press
Id = D2.2
Text = The robot having an arm in the proximity of the press may only rotate if this arm is retracted or if the press is in its lower or upper position.
«rationale»
Proximity is an ambigious term, instead we refine the term by stating that the arms should be retracted or press should not be in loadposition whenever the robot moves.
Robot rotates and its arms are in proximity of press iff robot.pick_from_table() or robot.load_press() or robot.deposit_on_belt() or robot.pick_from_press()
«refine»
«requirement»
Robot arm rotation near press
Id =
Text = Robot may only rotate iff (arm1.arm_pos=Arm_Position::retracted and arm2.arm_pos=Arm_Position::retracted) or (press.middle.value=false)
«trace»
«satisfy»
act ProduceForgedPlates [High level]
forge
load_press
deposit_on_belt_int
RobotBehaviour
PressBehaviour
Moving down
UnloadPosition
Entry/self.pos=Press_Position.unloadposition
[self.bottom.value==true]
^robot.Pick_From_Press()
«block, allocated, requirementRelated»
Press
pos : Press_Position
loaded : Boolean
initialize ( )
forge ( )
go_load_position ( )
- robot
- press
«block, allocated, requirementRelated»
Robot
<<interval>>{min = "-100", max = "70"} rotationDegree : Angle
depositonbelt_counter : Number_Blanks
arm2_drop_retraction : Extension
arm2_drop_extension : Extension
arm2_pick_extension : Extension
arm2_pick_retraction : Extension
arm1_drop_retraction : Extension
arm1_drop_extension : Extension
arm1_pick_retraction : Extension
arm1_pick_extension : Extension
pos : Robot_Position
state : Robot_State
desired_angle : Angle
moving : Boolean
initialize ( )
pick_from_table ( )
load_press ( )
pick_from_press ( )
deposit_on_belt ( )
move ( )
deposit_on_belt_int ( )
depositonbelt_init ( )
depositonbelt_increment ( )
depositonbelt_decrement ( )
- arm1
- arm2
«block, allocated»
Arm
<<interval>> {min = "0", max = "1"} extensionRange : Extension
arm_pos : Arm_Position
loaded : Boolean
drop ( )
extend ( )
pick ( )
retract ( )
context Robot::load_press() pre: (self .arm1.arm_pos=Arm_Position.retracted and self .arm2.arm_pos=Arm_Position.retracted) or self .press.middle.value=false
context Robot::pick_from_press() pre: (self .arm1.arm_pos=Arm_Position.retracted and self .arm2.arm_pos=Arm_Position.retracted) or self .press.middle.value=false
context Robot::deposit_on_belt() pre: (self .arm1.arm_pos=Arm_Position::retracted and self .arm2.arm_pos=Arm_Position::retracted) or self .press.middle.value=false
context Robot::pick_from_table() pre: (self .arm1.arm_pos=Arm_Position.retracted and self .arm2.arm_pos=Arm_Position.retracted) or self .press.middle.value=false

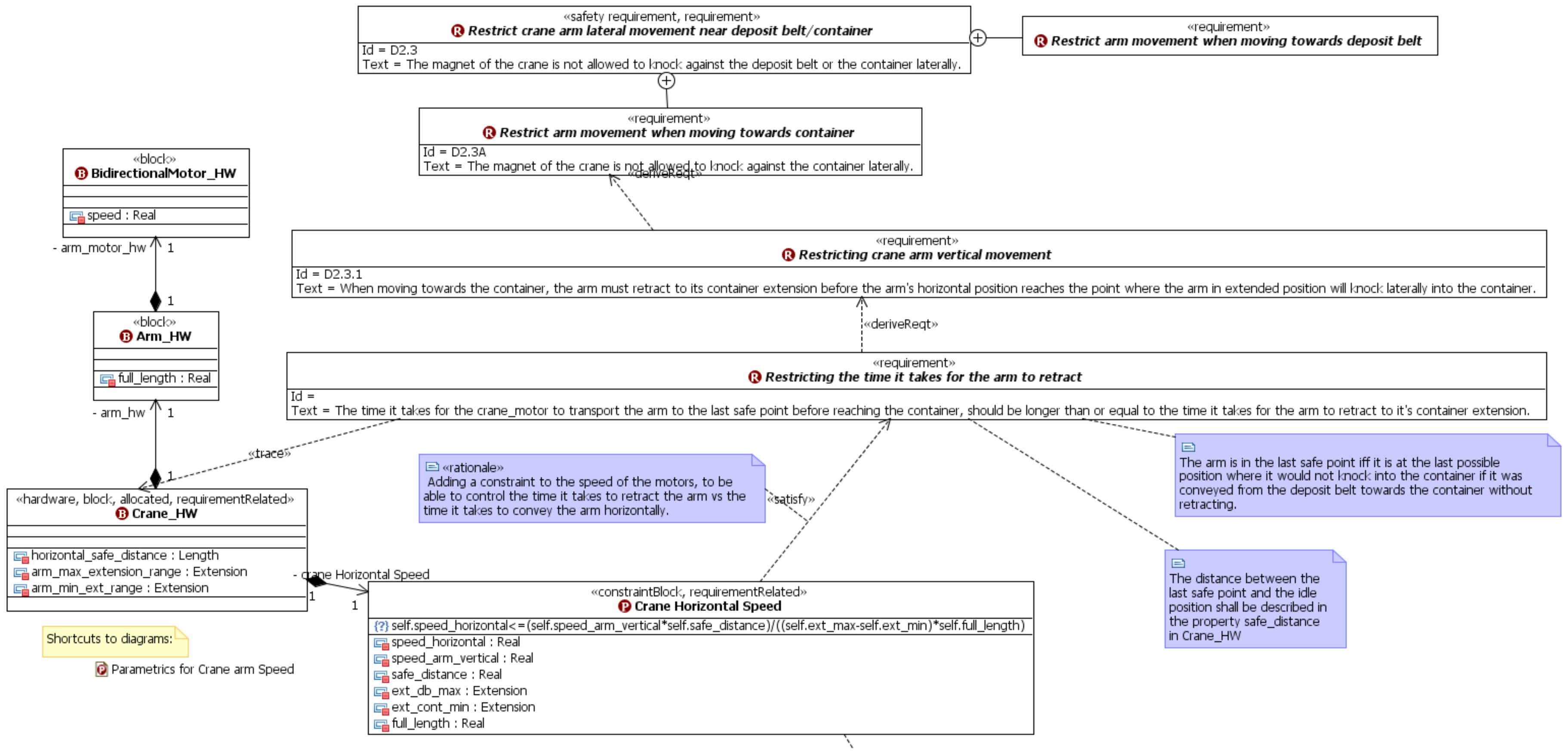
A.10.2 Safety analysis for requirement 2.3A

The next two pages show the safety analysis diagram for requirement 2.3A. The first image is the full image, the second shows a close-up view for the analysis.

The parametric diagram for the constraint property of Crane_HW, can be seen in Figure 110.



«safety requirement, requirement»
Restrict crane arm lateral movement near deposit belt/container
Id = D2.3
Text = The magnet of the crane is not allowed to knock against the deposit belt or the container laterally.
«requirement»
Restrict arm movement when moving towards deposit belt
«requirement»
Restrict arm movement when moving towards container
Id = D2.3A
Text = The magnet of the crane is not allowed to knock against the container laterally.
«deriveReqt»
«block»
BidirectionalMotor_HW
speed : Real
- arm_motor_hw
1
1
«block»
Arm_HW
full_length : Real
- arm_hw
1
1
«requirement»
Restricting crane arm vertical movement
Id = D2.3.1
Text = When moving towards the container, the arm must retract to its container extension before the arm's horizontal position reaches the point where the arm in extended position will knock laterally into the container.
«deriveReqt»
«requirement»
Restricting the time it takes for the arm to retract
Id =
Text = The time it takes for the crane_motor to transport the arm to the last safe point before reaching the container, should be longer than or equal to the time it takes for the arm to retract to it's container extension.
«trace»
«rationale»
Adding a constraint to the speed of the motors, to be able to control the time it takes to retract the arm vs the time it takes to convey the arm horizontally.
«satisfy»
The arm is in the last safe point iff it is at the last possible position where it would not knock into the container if it was conveyed from the deposit belt towards the container without retracting.
«hardware, block, allocated, requirementRelated»
Crane_HW
horizontal_safe_distance : Length
arm_max_extension_range : Extension
arm_min_ext_range : Extension
- crane Horizontal Speed
1
1
Shortcuts to diagrams:
Parametrics for Crane arm Speed
«constraintBlock, requirementRelated»
Crane Horizontal Speed
{?} self.speed_horizontal<=(self.speed_arm_vertical*self.safe_distance)/((self.ext_max-self.ext_min)*self.full_length)
speed_horizontal : Real
speed_arm_vertical : Real
safe_distance : Real
ext_db_max : Extension
ext_cont_min : Extension
full_length : Real
The distance between the last safe point and the idle position shall be described in the property safe_distance in Crane_HW
Crane_HW
crane_motor_hw : BidirectionalMotor_HW
speed : Real
horizontal_safe_distance : Length
arm_max_extension_range : Extension
arm_min_ext_range : Extension
arm_hw : Arm_HW
full_length : Real
arm_motor_hw : BidirectionalMotor_HW
speed : Real
«constraintProperty»
speed_horizontal : Real
safe_distance : Real
ext_db_max : Extension
ext_cont_min : Extension
crane Horizontal Speed : Crane Horizontal Speed
Constraints
{?} self.speed_horizontal<=(self.speed_arm_vertical*self.safe_distance)/(self.range_length=(self.ext_max-self.ext_min)*self.full_length)
full_length : Real
Some trouble with the OCL expression using Real or self-defined ValueTypes
speed_arm_vertical : Real

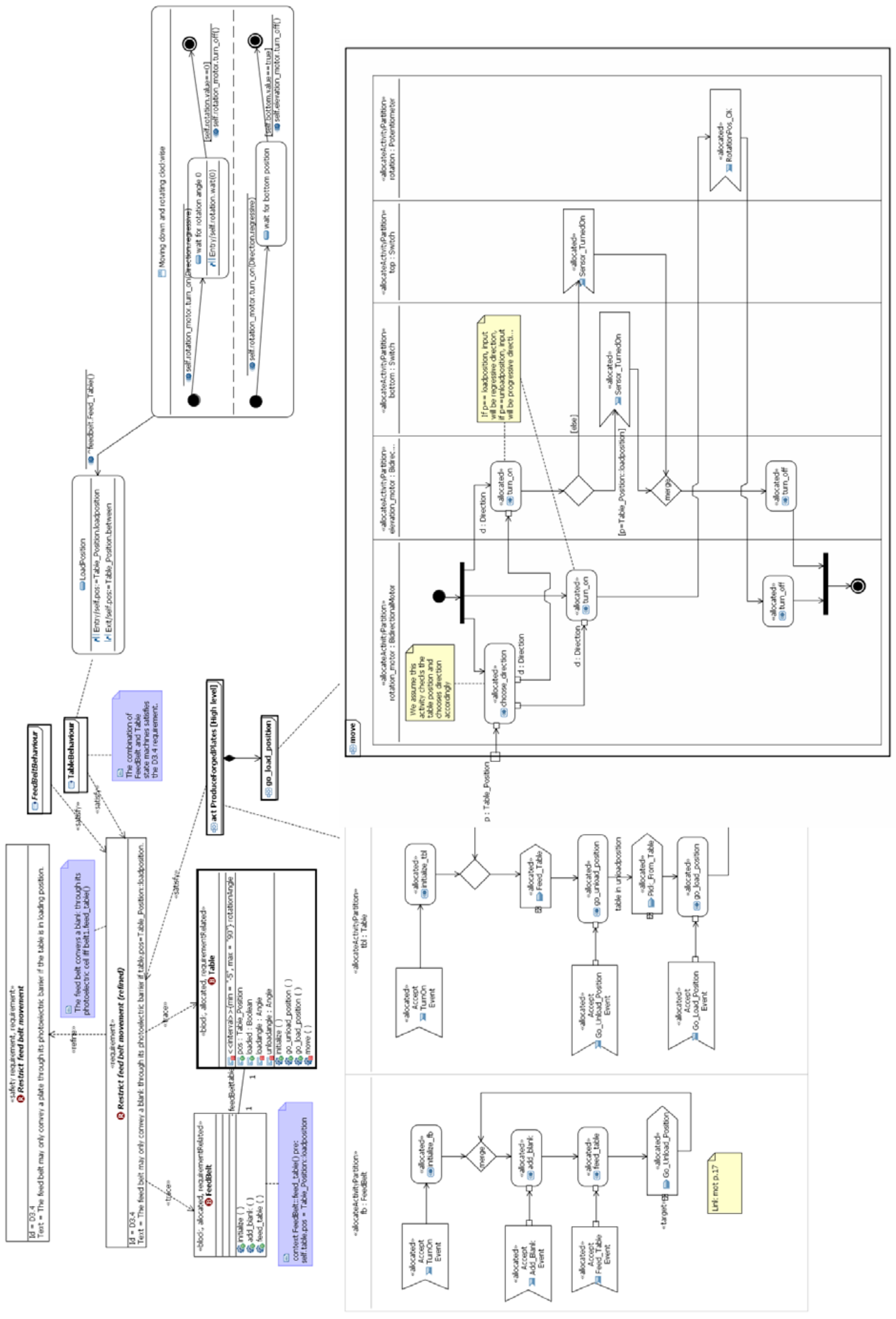

«safety requirement, requirement»
Restrict crane arm lateral movement near deposit belt/container
Id = D2.3
Text = The magnet of the crane is not allowed to knock against the deposit belt or the container laterally.
«requirement»
Restrict arm movement when moving towards deposit belt
«requirement»
Restrict arm movement when moving towards container
Id = D2.3A
Text = The magnet of the crane is not allowed to knock against the container laterally.
«deriveReqt»
«requirement»
Restricting crane arm vertical movement
Id = D2.3.1
Text = When moving towards the container, the arm must retract to its container extension before the arm's horizontal position reaches the point where the arm in extended position will knock laterally into the container.
«deriveReqt»
«requirement»
Restricting the time it takes for the arm to retract
Id =
Text = The time it takes for the crane_motor to transport the arm to the last safe point before reaching the container, should be longer than or equal to the time it takes for the arm to retract to it's container extension.
The arm is in the last safe point iff it is at the last possible position where it would not knock into the container if it was conveyed from the deposit belt towards the container without retracting.
«rationale»
Adding a constraint to the speed of the motors, to be able to control the time it takes to retract the arm vs the time it takes to convey the arm horizontally.
«satisfy»
«trace»
The distance between the last safe point and the idle position shall be described in the property safe_distance in Crane_HW
«block»
BidirectionalMotor_HW
speed : Real
- arm_motor_hw
1
1
«block»
Arm_HW
full_length : Real
- arm_hw
1
1
«hardware, block, allocated, requirementRelated»
Crane_HW
horizontal_safe_distance : Length
arm_max_extension_range : Extension
arm_min_ext_range : Extension
- crane Horizontal Speed
1
1
«constraintBlock, requirementRelated»
Crane Horizontal Speed
{?} self.speed_horizontal<=(self.speed_arm_vertical*self.safe_distance)/((self.ext_max-self.ext_min)*self.full_length)
speed_horizontal : Real
speed_arm_vertical : Real
safe_distance : Real
ext_db_max : Extension
ext_cont_min : Extension
full_length : Real
Shortcuts to diagrams:
Parametrics for Crane arm Speed

A.10.3 Safety analysis diagram for requirement 3.4

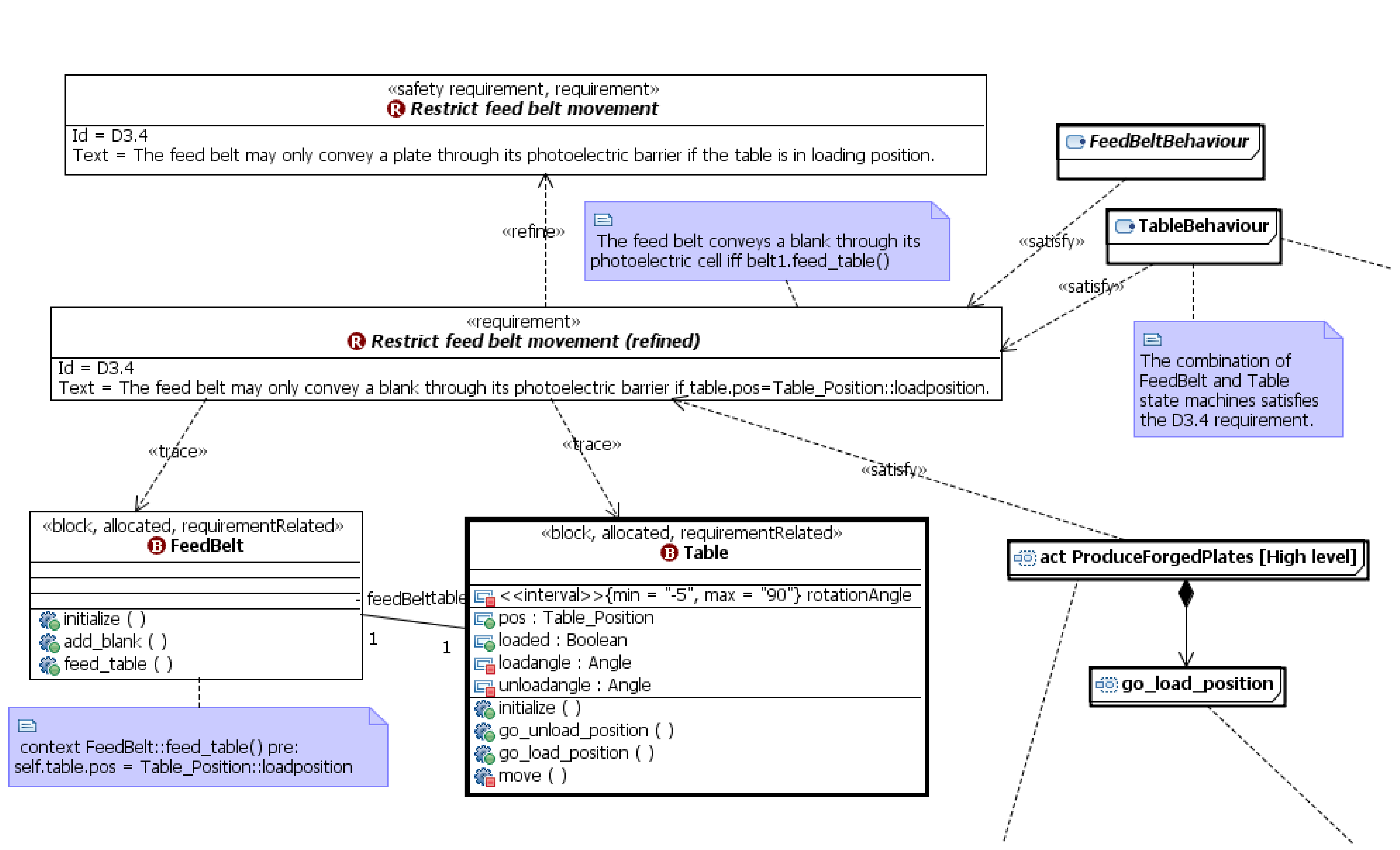
The next two pages show the safety analysis diagram for requirement 3.4. The first image is the full image, the second shows a close-up view for the analysis.

For detailed views of the activity diagrams:
FeedBelt and Table partition in Figure 61.
Table activity diagram, act move in Figure 68.

For detailed views of the state machine diagrams:
Table in Figure 92, slice from main region: “Moving Down and Rotating Clockwise” and “LoadPosition” states and the transition between them.
FeedBelt in Figure 90.



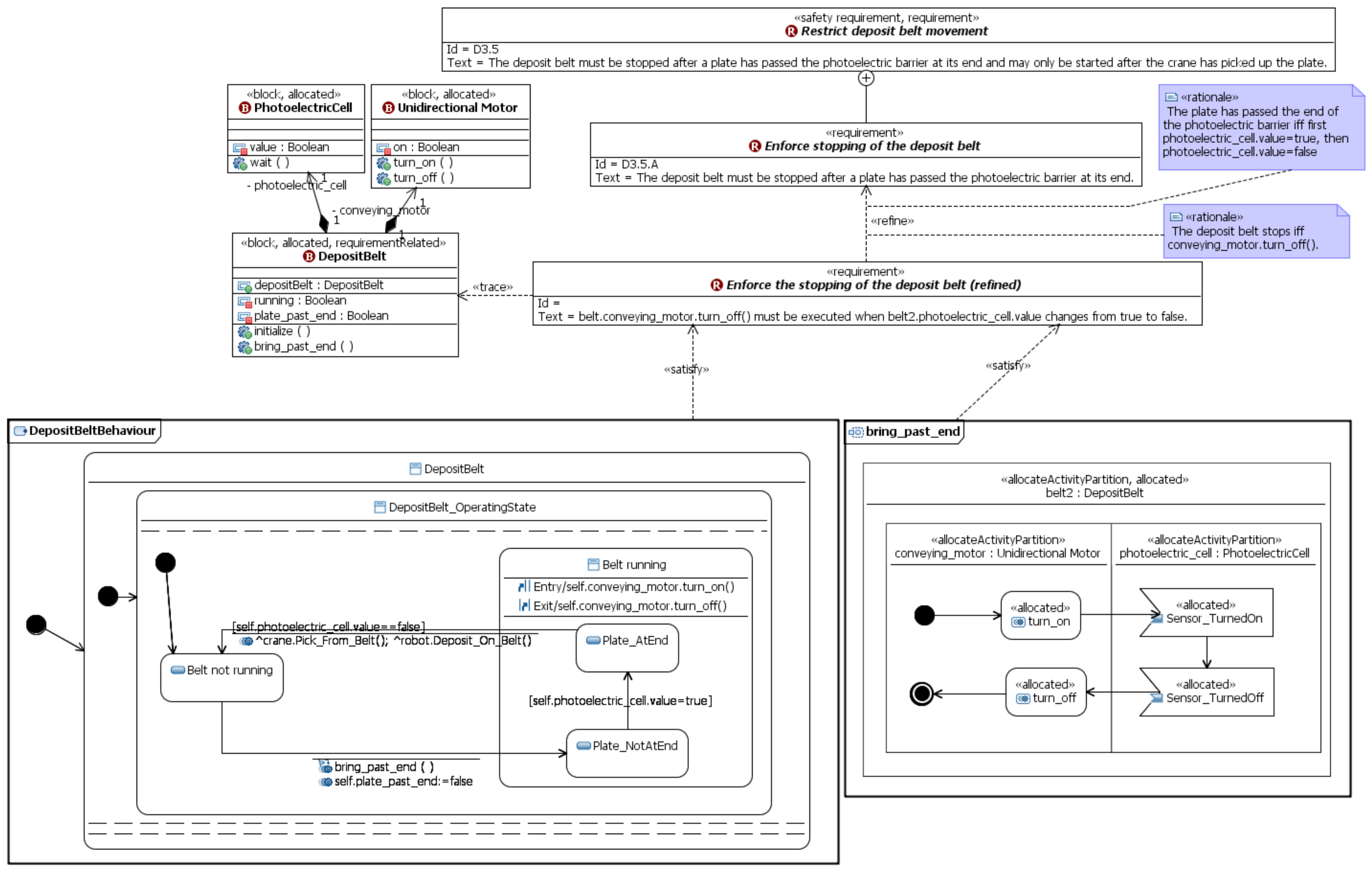
«safety requirement, requirement» Restrict feed belt movement
Id = D3.4
Text = The feed belt may only convey a plate through its photoelectric barrier if the table is in loading position.
«refine»
The feed belt conveys a blank through its photoelectric cell iff belt1.feed_table()
«requirement» Restrict feed belt movement (refined)
Id = D3.4
Text = The feed belt may only convey a blank through its photoelectric barrier if table.pos=Table_Position::loadposition.
«trace»
«satisfy»
FeedBeltBehaviour
TableBehaviour
The combination of FeedBelt and Table state machines satisfies the D3.4 requirement.
act ProduceForgedPlates [High level]
go_load_position
«block, allocated, requirementRelated» FeedBelt
initialize ( )
add_blank ( )
feed_table ( )
feedBelt
table
«block, allocated, requirementRelated» Table
<<interval>> {min = "-5", max = "90"} rotationAngle
pos : Table_Position
loaded : Boolean
loadangle : Angle
unloadangle : Angle
initialize ( )
go_unload_position ( )
go_load_position ( )
move ( )
context FeedBelt::feed_table() pre: self.table.pos = Table_Position::loadposition
LoadPosition
Entry/self.pos:=Table_Position.loadposition
Exit/self.pos:=Table_Position.between
^feedbelt.Feed_Table()
Moving down and rotating clockwise
self.rotation_motor.turn_on(Direction::regressive)
wait for rotation angle 0
Entry/self.rotation.wait(0)
[self.rotation.value==0]
self.rotation_motor.turn_off()
self.rotation_motor.turn_on(Direction::regressive)
wait for bottom position
[self.bottom.value==true]
self.elevation_motor.turn_off()
move
«allocateActivityPartition» rotation_motor : BidirectionalMotor
«allocateActivityPartition» elevation_motor : Bidirec...
«allocateActivityPartition» bottom : Switch
«allocateActivityPartition» top : Switch
«allocateActivityPartition» rotation : Potentiometer
We assume this activity checks the table position and chooses direction accordingly
If p== loadposition, input will be regressive direction, if p==unloadposition, input will be progressive directi...
«allocated» choose_direction
d : Direction
«allocated» turn_on
«allocated» turn_off
[else]
[p=Table_Position::loadposition]
merge
«allocated» Sensor_TurnedOn
«allocated» RotationPos_Ok
p : Table_Position
«allocateActivityPartition» fb : FeedBelt
«allocateActivityPartition» tbl : Table
«allocated» Accept TurnOn Event
«allocated» initialize_fb
«allocated» Accept Add_Blank Event
«allocated» add_blank
«allocated» Accept Feed_Table Event
«allocated» feed_table
«allocated» Go_Unload_Position
«target»
Link: mct p.17
«allocated» initialize_tbl
«allocated» Feed_Table
«allocated» Accept Go_Unload_Position Event
«allocated» go_unload_position
table in unloadposition
«allocated» Pkd_From_Table
«allocated» Accept Go_Load_Position Event
«allocated» go_load_position


«safety requirement, requirement»
Restrict feed belt movement
Id = D3.4
Text = The feed belt may only convey a plate through its photoelectric barrier if the table is in loading position.
FeedBeltBehaviour
TableBehaviour
«refine»
«satisfy»
«satisfy»
The feed belt conveys a blank through its photoelectric cell iff belt1.feed_table()
«requirement»
Restrict feed belt movement (refined)
Id = D3.4
Text = The feed belt may only convey a blank through its photoelectric barrier if table.pos=Table_Position::loadposition.
The combination of FeedBelt and Table state machines satisfies the D3.4 requirement.
«trace»
«trace»
«satisfy»
«block, allocated, requirementRelated»
FeedBelt
initialize ( )
add_blank ( )
feed_table ( )
- feedBelttable
1
1
«block, allocated, requirementRelated»
Table
<<interval>>{min = "-5", max = "90"} rotationAngle
pos : Table_Position
loaded : Boolean
loadangle : Angle
unloadangle : Angle
initialize ( )
go_unload_position ( )
go_load_position ( )
move ( )
act ProduceForgedPlates [High level]
go_load_position
context FeedBelt::feed_table() pre:
self.table.pos = Table_Position::loadposition

A.10.4 Safety requirement for requirement 3.5A

The next page shows the safety analysis diagram for requirement 3.4.

To see the DepositBelt activity diagram for act bring_past_end, see Figure 80.
To see the whole state machine diagram for DepositBelt, see Figure 99.

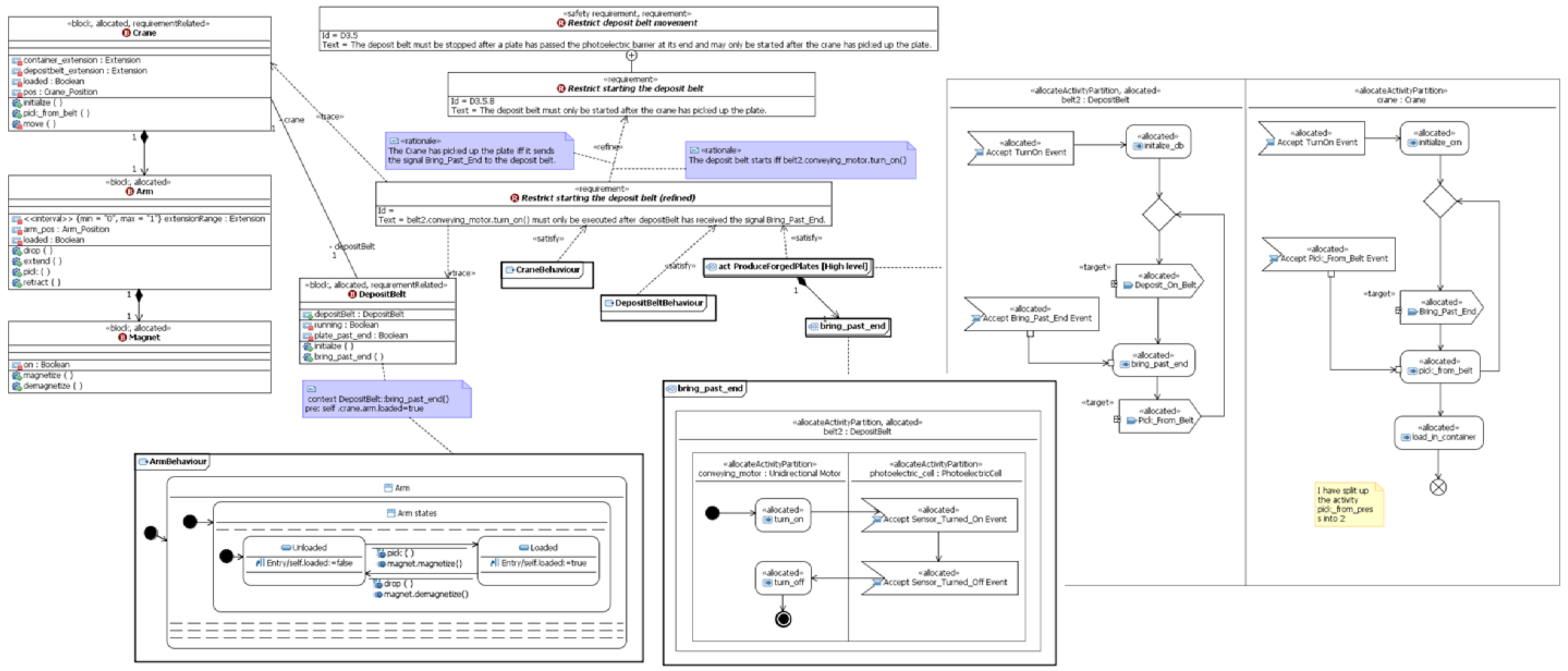

«safety requirement, requirement»
Restrict deposit belt movement
Id = D3.5
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.
«rationale»
The plate has passed the end of the photoelectric barrier iff first photoelectric_cell.value=true, then photoelectric_cell.value=false
«requirement»
Enforce stopping of the deposit belt
Id = D3.5.A
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end.
«refine»
«rationale»
The deposit belt stops iff conveying_motor.turn_off().
«requirement»
Enforce the stopping of the deposit belt (refined)
Id =
Text = belt.conveying_motor.turn_off() must be executed when belt2.photoelectric_cell.value changes from true to false.
«trace»
«satisfy»
«satisfy»
«block, allocated»
PhotoelectricCell
value : Boolean
wait ( )
«block, allocated»
Unidirectional Motor
on : Boolean
turn_on ( )
turn_off ( )
- photoelectric_cell
- conveying_motor
1
1
1
1
«block, allocated, requirementRelated»
DepositBelt
depositBelt : DepositBelt
running : Boolean
plate_past_end : Boolean
initialize ( )
bring_past_end ( )
DepositBeltBehaviour
DepositBelt
DepositBelt_OperatingState
Belt running
Entry/self.conveying_motor.turn_on()
Exit/self.conveying_motor.turn_off()
Plate_AtEnd
Plate_NotAtEnd
[self.photoelectric_cell.value=true]
[self.photoelectric_cell.value==false]
^crane.Pick_From_Belt(); ^robot.Deposit_On_Belt()
Belt not running
bring_past_end ( )
self.plate_past_end:=false
bring_past_end
«allocateActivityPartition, allocated»
belt2 : DepositBelt
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
«allocated»
Sensor_TurnedOn
«allocated»
Sensor_TurnedOff
«allocated»
turn_off

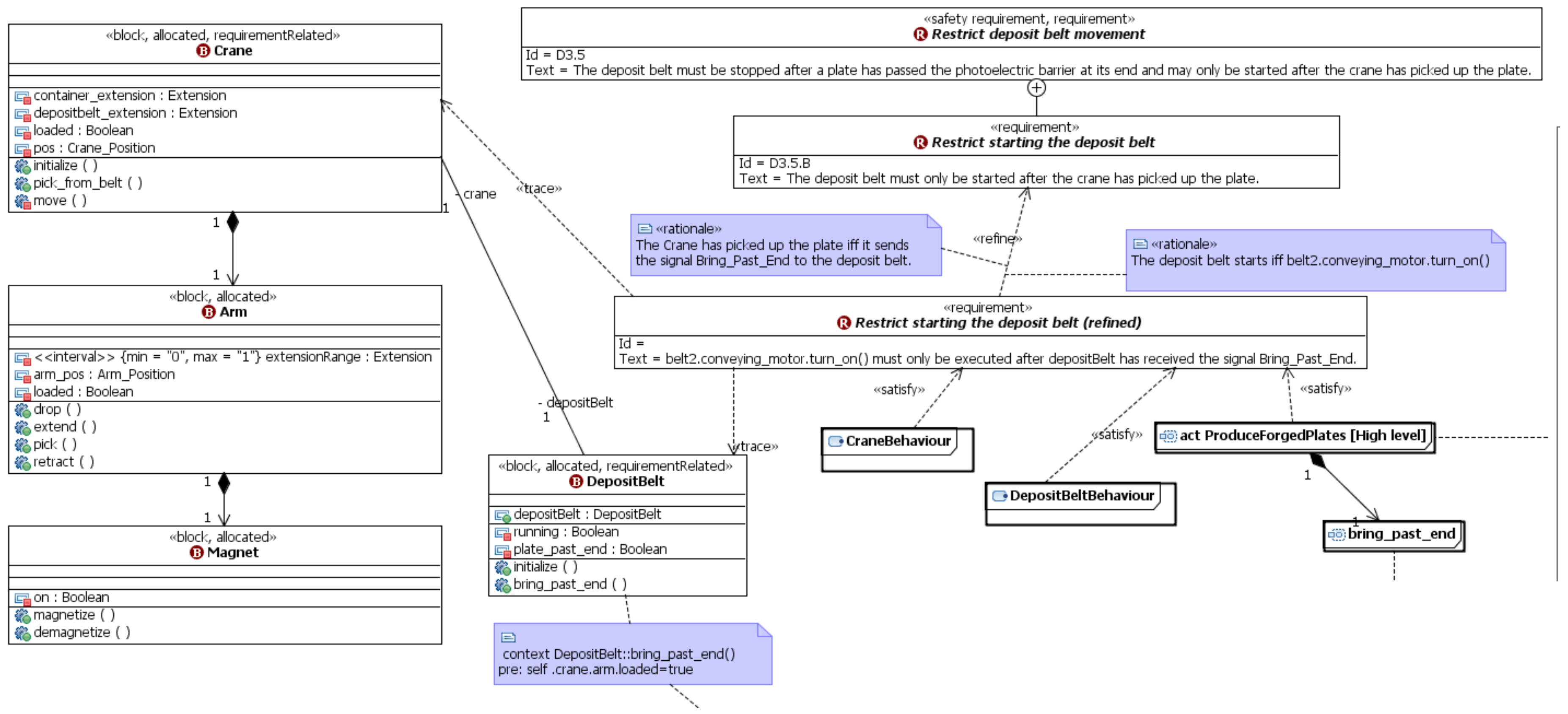
A.10.5 Safety analysis for requirement 3.5B

The next two pages show the safety analysis diagram for requirement 3.5B. The first image is the full image, the second shows a close-up view for the analysis.

For detailed views of the activity diagrams:
DepositBelt and Crane partion in Figure 63.
DepositBelt activity diagram, act bring_past_end, in Figure 80.

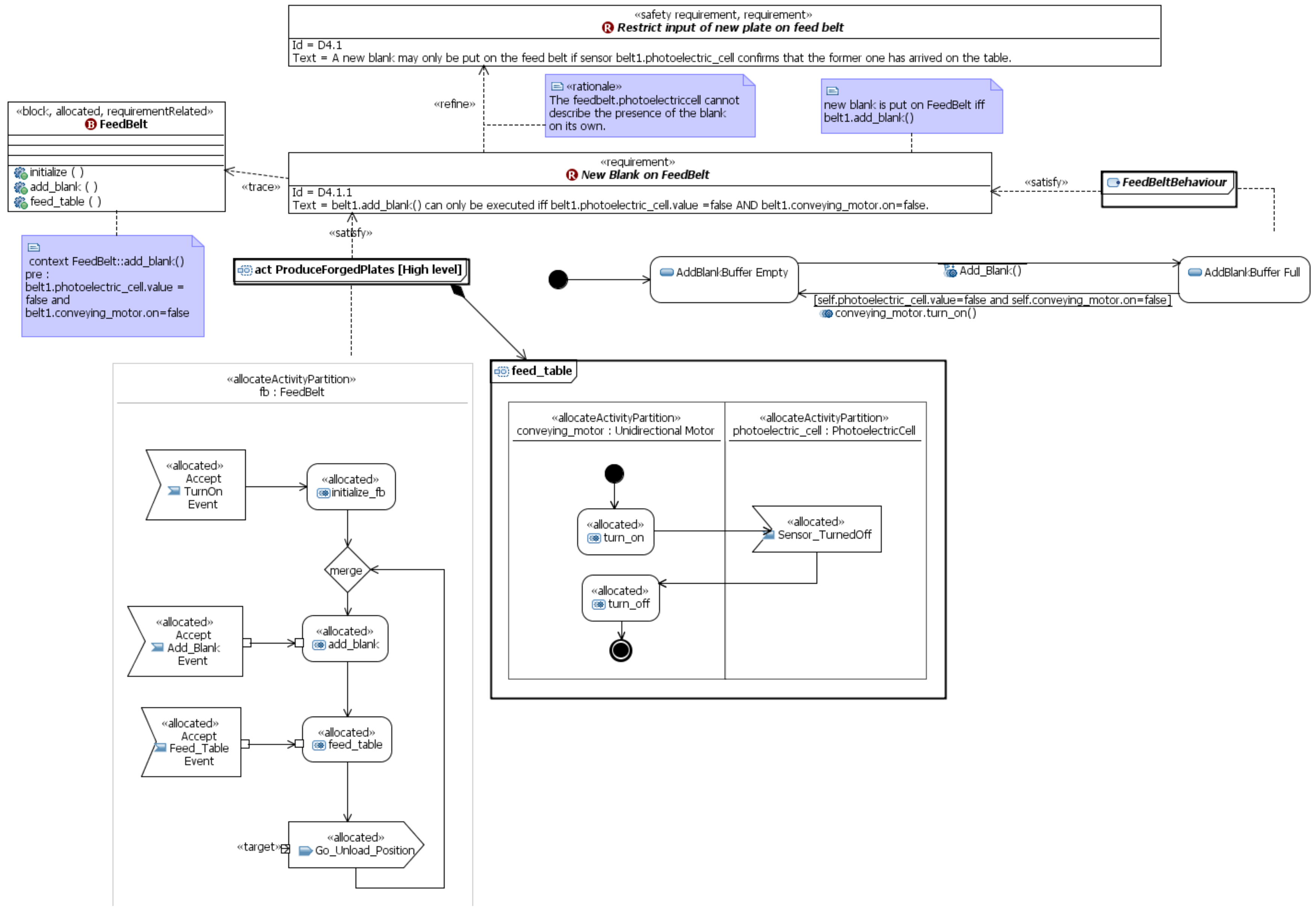
For detailed views of the state machine diagrams:
ArmBehaviour in Figure 103, slice: arm main region.
DepositBeltBehaviour in Figure 99. (have not sliced this)
CraneBehaviour in Figure 100. (have not sliced this)


«block, allocated, requirementRelated»
Crane
container_extension : Extension
depositbelt_extension : Extension
loaded : Boolean
pos : Crane_Position
initialize ( )
pick_from_belt ( )
move ( )
«block, allocated»
Arm
<<interval>> {min = "0", max = "1"} extensionRange : Extension
arm_pos : Arm_Position
loaded : Boolean
drop ( )
extend ( )
pick ( )
retract ( )
«block, allocated»
Magnet
on : Boolean
magnetize ( )
demagnetize ( )
«safety requirement, requirement»
Restrict deposit belt movement
Id = D3.5
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.
«requirement»
Restrict starting the deposit belt
Id = D3.5.B
Text = The deposit belt must only be started after the crane has picked up the plate.
«rationale»
The Crane has picked up the plate iff it sends the signal Bring_Past_End to the deposit belt.
«refine»
«rationale»
The deposit belt starts iff belt2.conveying_motor.turn_on()
«requirement»
Restrict starting the deposit belt (refined)
Id =
Text = belt2.conveying_motor.turn_on() must only be executed after depositBelt has received the signal Bring_Past_End.
crane
«trace»
- depositBelt
«satisfy»
CraneBehaviour
DepositBeltBehaviour
act ProduceForgedPlates [High level]
bring_past_end
«block, allocated, requirementRelated»
DepositBelt
depositBelt : DepositBelt
running : Boolean
plate_past_end : Boolean
initialize ( )
bring_past_end ( )
context DepositBelt::bring_past_end()
pre: self .crane.arm.loaded=true
ArmBehaviour
Arm
Arm states
Unloaded
Entry/self.loaded:=false
Loaded
Entry/self.loaded:=true
pick ( )
magnet.magnetize()
drop ( )
magnet.demagnetize()
bring_past_end
«allocateActivityPartition, allocated»
belt2 : DepositBelt
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
turn_off
Accept Sensor_Turned_On Event
Accept Sensor_Turned_Off Event
«allocateActivityPartition, allocated»
belt2 : DepositBelt
Accept TurnOn Event
initialize_db
«target»
Deposit_On_Belt
Accept Bring_Past_End Event
bring_past_end
Pick_From_Belt
«allocateActivityPartition»
crane : Crane
initialize_cm
Accept Pick_From_Belt Event
Bring_Past_End
pick_from_belt
load_in_container
I have split up the activity pick_from_pres s into 2


«block, allocated, requirementRelated»
Crane
container_extension : Extension
depositbelt_extension : Extension
loaded : Boolean
pos : Crane_Position
initialize ( )
pick_from_belt ( )
move ( )
«block, allocated»
Arm
<<interval>> {min = "0", max = "1"} extensionRange : Extension
arm_pos : Arm_Position
loaded : Boolean
drop ( )
extend ( )
pick ( )
retract ( )
«block, allocated»
Magnet
on : Boolean
magnetize ( )
demagnetize ( )
1
1
1
1
- crane
1
- depositBelt
1
«trace»
«block, allocated, requirementRelated»
DepositBelt
depositBelt : DepositBelt
running : Boolean
plate_past_end : Boolean
initialize ( )
bring_past_end ( )
context DepositBelt::bring_past_end()
pre: self .crane.arm.loaded=true
«safety requirement, requirement»
Restrict deposit belt movement
Id = D3.5
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.
«requirement»
Restrict starting the deposit belt
Id = D3.5.B
Text = The deposit belt must only be started after the crane has picked up the plate.
«rationale»
The Crane has picked up the plate iff it sends the signal Bring_Past_End to the deposit belt.
«refine»
«rationale»
The deposit belt starts iff belt2.conveying_motor.turn_on()
«requirement»
Restrict starting the deposit belt (refined)
Id =
Text = belt2.conveying_motor.turn_on() must only be executed after depositBelt has received the signal Bring_Past_End.
«trace»
«satisfy»
«satisfy»
«satisfy»
CraneBehaviour
DepositBeltBehaviour
act ProduceForgedPlates [High level]
1
1
bring_past_end

A.10.6 Safety analysis diagram for requirement 4.1

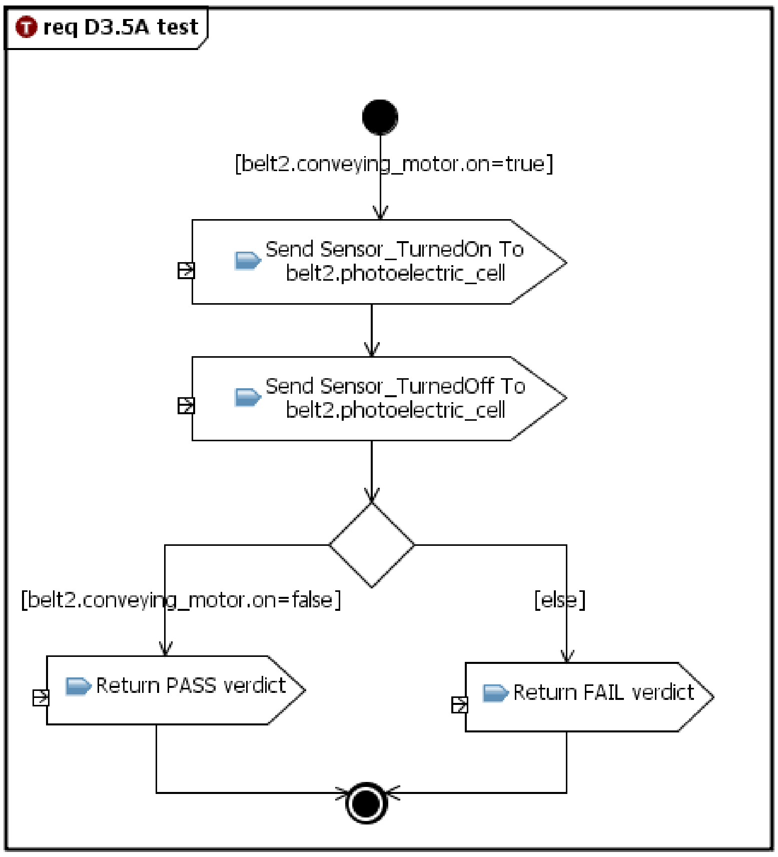
The following diagram shows the safety analysis diagram for requirement 3.5B.



«safety requirement, requirement»
Restrict input of new plate on feed belt
Id = D4.1
Text = A new blank may only be put on the feed belt if sensor belt1.photoelectric_cell confirms that the former one has arrived on the table.
«refine»
«rationale»
The feedbelt.photoelectriccell cannot describe the presence of the blank on its own.
new blank is put on FeedBelt iff belt1.add_blank()
«block, allocated, requirementRelated»
FeedBelt
initialize ( )
add_blank ( )
feed_table ( )
«trace»
«requirement»
New Blank on FeedBelt
Id = D4.1.1
Text = belt1.add_blank() can only be executed iff belt1.photoelectric_cell.value =false AND belt1.conveying_motor.on=false.
«satisfy»
FeedBeltBehaviour
«satisfy»
context FeedBelt::add_blank()
pre :
belt1.photoelectric_cell.value =
false and
belt1.conveying_motor.on=false
act ProduceForgedPlates [High level]
AddBlank:Buffer Empty
Add_Blank()
AddBlank:Buffer Full
[self.photoelectric_cell.value=false and self.conveying_motor.on=false]
conveying_motor.turn_on()
«allocateActivityPartition»
fb : FeedBelt
«allocated»
Accept
TurnOn
Event
«allocated»
initialize_fb
merge
«allocated»
Accept
Add_Blank
Event
«allocated»
add_blank
«allocated»
Accept
Feed_Table
Event
«allocated»
feed_table
«target»
«allocated»
Go_Unload_Position
feed_table
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
«allocated»
Sensor_TurnedOff
«allocated»
turn_off

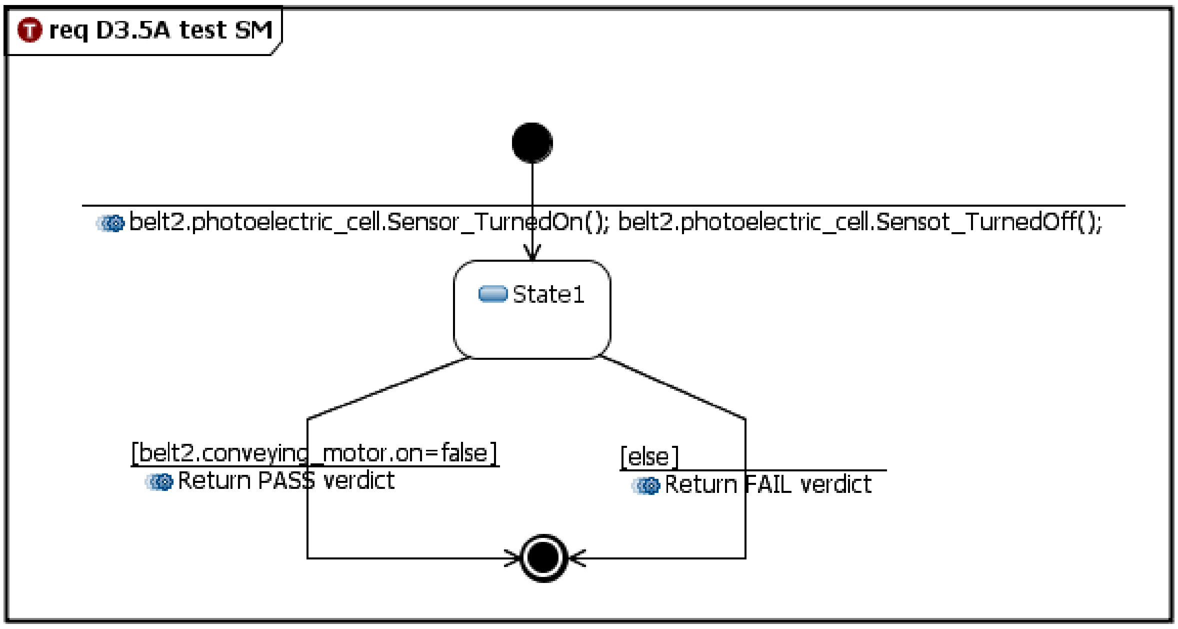
A.11 Test diagrams

A.11.1 Req D3.5A tests



req D3.5A test
[belt2.conveying_motor.on=true]
Send Sensor_TurnedOn To belt2.photoelectric_cell
Send Sensor_TurnedOff To belt2.photoelectric_cell
[belt2.conveying_motor.on=false]
[else]
Return PASS verdict
Return FAIL verdict

Figure 116 - act [Package] Analysis [Test case for req D3.5A]



req D3.5A test SM
belt2.photoelectric_cell.Sensor_TurnedOn(); belt2.photoelectric_cell.Sensot_TurnedOff();
State1
[belt2.conveying_motor.on=false]
Return PASS verdict
[else]
Return FAIL verdict

Figure 117 - sm [Package] Analysis [Test case for req D3.5A]

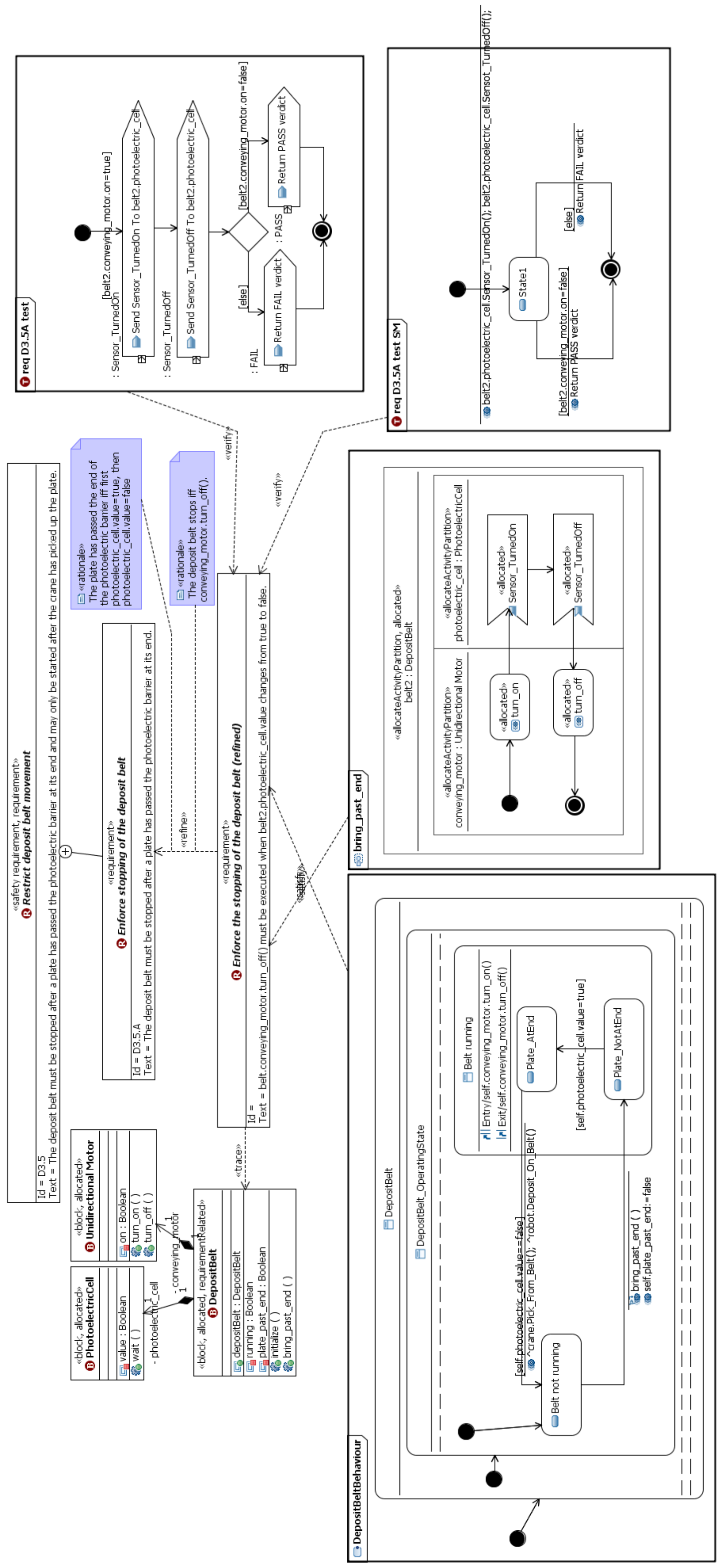

«safety requirement, requirement»
Restrict deposit belt movement
Id = D3.5
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end and may only be started after the crane has picked up the plate.
«requirement»
Enforce stopping of the deposit belt
Id = D3.5.A
Text = The deposit belt must be stopped after a plate has passed the photoelectric barrier at its end.
«rationale»
The plate has passed the end of the photoelectric barrier iff first photoelectric_cell.value=true, then photoelectric_cell.value=false
«rationale»
The deposit belt stops iff conveying_motor.turn_off().
«refine»
«requirement»
Enforce the stopping of the deposit belt (refined)
Id =
Text = belt.conveying_motor.turn_off() must be executed when belt2.photoelectric_cell.value changes from true to false.
«verify»
«verify»
«satisfy»
«trace»
«block, allocated»
PhotoelectricCell
value : Boolean
wait ( )
«block, allocated»
Unidirectional Motor
on : Boolean
turn_on ( )
turn_off ( )
- photoelectric_cell
- conveying_motor
«block, allocated, requirementRelated»
DepositBelt
depositBelt : DepositBelt
running : Boolean
plate_past_end : Boolean
initialize ( )
bring_past_end ( )
req D3.5A test
: Sensor_TurnedOn
[belt2.conveying_motor.on=true]
Send Sensor_TurnedOn To belt2.photoelectric_cell
: Sensor_TurnedOff
Send Sensor_TurnedOff To belt2.photoelectric_cell
[belt2.conveying_motor.on=false]
[else]
: FAIL
Return FAIL verdict
: PASS
Return PASS verdict
req D3.5A test SM
belt2.photoelectric_cell.Sensor_TurnedOn(); belt2.photoelectric_cell.Sensot_TurnedOff();
State1
[belt2.conveying_motor.on=false]
Return PASS verdict
[else]
Return FAIL verdict
bring_past_end
«allocateActivityPartition, allocated»
belt2 : DepositBelt
«allocateActivityPartition»
conveying_motor : Unidirectional Motor
«allocateActivityPartition»
photoelectric_cell : PhotoelectricCell
«allocated»
turn_on
«allocated»
turn_off
«allocated»
Sensor_TurnedOn
«allocated»
Sensor_TurnedOff
DepositBeltBehaviour
DepositBelt
DepositBelt_OperatingState
Belt running
Entry/self.conveying_motor.turn_on()
Exit/self.conveying_motor.turn_off()
Plate_AtEnd
Plate_NotAtEnd
[self.photoelectric_cell.value=true]
[self.photoelectric_cell.value=false]
^crane.Pick_From_Belt(); ^robot.Deposit_On_Belt()
Belt not running
bring_past_end ( )
self.plate_past_end:=false

Figure 118 - Cross-cutting diagram with traces and tests for req D3.5A

Appendix B Glossary

Software block	Software blocks in this model represent the physical part of the same name and contains the control logic needed to control the that part.			
Hardware block	Hardware block represent the physical part. Only the high level hardware blocks are represented in this model.			

Block	Description	Part of	Specialization of	Contains
Actuator	***Superclass** generalizing the software blocks which represent the hardware parts which can be activated/deactivated by the control system.*	----	----	
Arm	**Software block** in the control system which contains the logic to control the physical extendable arm of the containing machine and enables the machine to reach nearby machines at different distances to pick up/drop metal plate.	Robot, Crane	----	Potentiometer, Magnet, BidirectionalMotor
BidirectionalMotor	**Software block** in the control system which contains the logic needed to activate/deactivate the physical motor (actuator) which can move in both progressive or regressive direction.	Table, Robot, Press, Crane, Arm	ElectricMotor	
Container	**Software block** representing the final destination for the forged metal plates. No control logic.	----	----	
Crane	**Software block** in the control system which contains the logic needed to control the physical crane and communicate with the surrounding machines..	ProductionCell_Controller	----	BidirectionalMotor, 2 Switch, Arm
Crane_HW	**Hardware block** representing the physical crane.	ProductionCell_HW	----	
DepositBelt	**Software block** in the control system which contains the logic needed to control the physical deposit belt and communicate with the surrounding machines.	ProductionCell_Controller	----	UnidirectionalMotor, PhotoelectricCell
DepositBelt_HW	**Hardware block** representing the physical deposit belt.	ProductionCell_HW	----	

ElectricMotor	***Superclass*** *generalizing the software blocks which represent two types of physical motors (actuators) which can be activated/deactivated by the control system.*	----	Actuator	
FeedBelt	**Software block** in the control system which contains the logic needed to control the physical feed belt and communicate with the surrounding machines.	ProductionCell_Controller	----	UnidirectionalMotor, PhotoelectricCell
FeedBelt_HW	**Hardware block** representing the physical feed belt.	ProductionCell_HW	----	
Magnet	**Software block** in the control system which contains the logic needed to activate/deactivate the physical magnet (actuator) which is placed at the end of each arm.	Arm	Actuator	
Operator	**Block** used to describe a human in a block diagram. In this model it is included to be able to describe the human in the domain's internal block diagram. No control logic.	ProductionCell_Domain	----	
PhotoelectricCell	**Software block** in the control system which contains the logic needed to react to the input from the physical photoelectric cell (sensor).	FeedBelt, DepositBelt	Sensor	
Potentiometer	**Software block** in the control system which contains the logic needed to react to the input from the physical potentiometer (sensor) which describes either the extension range or the angle of a machine or part.	Robot, Arm	Sensor	
Press	**Software block** in the control system which contains the logic needed to control the physical press and communicate with the surrounding machines.	ProductionCell_Controller	----	
Press_HW	**Hardware block** representing the physical press.	ProductionCell_HW	----	
ProductionCell_Controller	The software top level block.	ProductionCell_Domain	----	
ProductionCell_Domain	This is the top level block in this model. It describes the whole domain of the system, which contains the system of interest (both software and hardware), the operator, the plates that will be/are forged and the container.	----	----	ProductionCell_System, Operator, Plate, Container
ProductionCell_HW	The top level hardware block this represents the hardware as a whole.	ProductionCell_Domain	----	
ProductionCell_System	Block that represents the system of interest, both hardware and softare blocks.	ProductionCell_Domain	----	ProductionCell_Controller, ProductionCell_HW

Robot	**Software block** in the control system which contains the logic needed to control the physical robot and communicate with the surrounding machines.	ProductionCell_Controller	----	
Robot_HW	**Hardware block** representing the physical robot.	ProductionCell_HW	----	
Sensor	***Superclass** generalizing the software blocks that represent the hardware parts which register change in the environment and provide input to the control system.*	----	----	
Switch	**Software block** in the control system which contains the logic needed to react to the input from the physical switch (sensor) which describes the position of parts of a machine.	Table, Press, Crane	Sensor	
Table	**Software block** in the control system which contains the logic needed to control the physical rotary table and communicate with the surrounding machines.	ProductionCell_Controller	----	2 BidirectionalMotor, 2 switches, Potentiometer
Table_HW	**Hardware block** representing the physical rotary table.	ProductionCell_HW	----	
UI	**Block** representing the user interface. This is not designed into further detail.	----	----	
UnidirectionalMotor	**Software block** in the control system which contains the logic needed to activate/deactivate the physical motor (actuator) which can move only in progressive direction.	FeedBelt, DepositBelt	ElectricMotor	

Appendix C Tool problems

Problem	*Description*	*Answer/(temporary) solution*
Signals (send in state machine)	**Conformance (optional):** Graphical notation for SendSignal is not available in state machines.	It is not required, but could contribute to make a simpler and more consistent diagram.
Sequence diagram	**Conformance:** Not possible to decompose lifelines, or use *extra global* combined *fragments* [14] .	A combined fragment can't go outside the diagram frame to include an incoming signal to a part. (e.g. include the first Add_Blank signal in a combined fragment in
Error messages	**General:** Some of the error messages during validation are very subtle and give less than little explanation for where the problematic elements are. Often occurs with association errors.	
Association problems	**General:** It may seem that some associations are left in the model after an associated block or node has been deleted from the model. Unsure when this happens.	Errors often have occurred on associations which are not connected to anything in the model.
Using existing activities in state machines	**Tool:** If we wish to use a previously modelled activity in a state machine, the whole activity is moved to/inside the state machine, without updating references in the activity's containing activity diagram where it is/may be used.	If the activity is referenced in another activity diagram, this will make the containing activity diagram incomplete.
Signals (send activity)	**Tool:** When adding a *SendSignalAction* in an activity diagram, the tool asks us to either create a Signal or select an existing element and then it automatically inserts the chosen signal in the parameters *signal* and the *type* of the automatically created *target* input pin.	Use the SendSignal action's properties tab to set all properties correct.
Messy diagrams	**Graphical:** Long qualified names and stereotypes are default in the program, and tends to clutter the diagram.	Haven't looked much into how to change the default settings, but I have tried, without finding other solutions than to select each of the elements I want to change and changing their appearance.
Control/Object flow	**Graphical:** Unable to change the default look of the	This is not a requirement, but would be a nice feature to be able to

	control and object flow to distinguish them from each other in an activity diagram.	distinguish flows in a large diagram. Solution: Select manually all connectors of one type and change colour of the lines. Not possible to make them dashed.
Activity parameter node	**Graphical:** The node is visually the same as a normal port in the diagram while the specification examples use a large rectangle node. Can make the instant understanding of the diagram more difficult.	Not an important flaw. Most for people without experience?
Activity	**Graphical:** Only way of making an activity is by making an activity diagram in the model.	The specification describes an activity as a normal block, but it is not possible to create the activity block without creating the diagram.
Item flow	**Graphical:** Deleting an item flow in an IBD diagram does not give an immediate effect visually even though they are deleted from the model.	Close and open the diagram and the item flows are gone.
Flow port	**Graphical:** Flow ports that are placed on the boundary of the containing block in an IBD will disappear when closing and opening the diagram. They are created in the model, however.	By creating an IBD for the containing block (one level above) of the block that needs the boundary port, the port will stick to the boundary.
Activity diagram and Accept Event.	**Graphical (minor detail):** When choosing a trigger event, the name isn’t set.	By changing the name as in the state machine trigger events, the name would be sure to be in synch with the event. This could avoid human errors during design if just changing the name and not the triggered event. (which will make it look right, but be an error in the design of the model.)
Flow ports	**Graphical:** When typing a flow port with a flow specification, this is registered by the tool as an interface and therefore the both the port and the conjugated port is marked with a lollipop symbol (if the notation is correct, this is correct). See Figure 49 for an example.	SysML specification uses lollipop notation only for standard ports.
Activity partition	**Graphical:** The “swimlanes”	

	AllocateActivityPartitions can’t be rearranged among themselves.	
IBD	**Graphical:** Troublesome to make nice graphical diagrams when opening the part compartments of parts of the main block.	When moving the bottom-most part in the ibd, this movement causes the whole containing part and its contained parts to move inside the diagram. By moving it far enough down in the diagram, the diagram keeps still until the bottom part is moved again..
Activity/Action parameter – specifying values	**User/tool:** Possible to type the parameter node, but not to specify the value.	May be user problem, not finding out how to type an input for an activity parameter node.
Activity diagram – Interruptible region	**User/Tool:** Lack of interruptible region in the activity diagram.	There are some menu choices regarding interruptible region, but we were unable to find where to find the element to draw the region.
Different configurations	**User/Tool:** [8] describes an initial values compartment to be able to define different values to different block configurations. We have not found this feature.	Tried to look for this regarding making different configurations for e.g. the different configurations for the arm.
IBD	**User/Tool:** When creating several IBDs to cover different aspects of the model, it may be easy to delete a part or port that is in use in another diagram. We get a warning, but no change to cancel. Some of the times this has happened, a delete has been impossible to undo.	