Using SysML to Support Safety Certification: A Methodology and Case Study

SIMULA RESEARCH LABORATORY
AND
DET NORSKE VERITAS

December 2009

Authored by: Lionel Briand, Thierry Coq, Tonje Klykken, Shiva Nejati, Rajwinder Panesar-Walawege, Mehrdad Sabetzadeh

Preface

State of Practice

Critical systems such as those found in the avionics, automotive, maritime, and energy domains are often subject to a formal process known as certification. The goal of certification is to ensure that such systems will operate safely in the presence of known hazards, and without posing undue risks to the users, the public, or the environment.

The evidence for safe operation of a system is usually presented in the form of a safety case – a structured collection of arguments, prepared by the system supplier, providing assurance that the system is acceptably safe. A safety case is an input to the safety evaluation process, typically carried out by a certification body, to determine if all the safety measures stipulated in the relevant technical standards have been met, and that the risks have been reduced to as low as reasonably practical.

With the increasing role of software in control and monitoring of critical systems, ensuring that the software running on these systems does not compromise safety is now crucial. As such, the development of a safety case is no longer exclusively concerned with the mechanical and electrical parts of a system, but also with the software elements of the system and how these elements interact with other system components, the users, and the environment. The part of an overall safety case that deals with software and its interactions is sometimes referred to as a software safety case. It is vital for a software safety case not only to ensure that the software is sound, but also that it is used correctly in the overall system.

Maritime & Energy (M&E) are two major safety critical sectors that are becoming increasingly more reliant on software. M&E now utilize various Integrated Software-Dependent Systems (ISDSs) in such areas as fire and gas detection, drilling and production, vessel propulsion, and steering and navigation. This makes the continued dependability, and predictability of the construction of these ISDSs paramount.

There are already a number of safety standards for ISDSs, e.g. IEC 61508 and ISO 17894, that address the software safety life-cycle activities and documentation requirements for these systems. Despite the existence of these standards, the safety evidence built in support of an ISDS's software is usually far less stringent and conclusive than that built for the system's mechanical and electrical equipment. This makes ISDSs more prone to critical failures due to software.

Objective and Observations

The broad aim of the ModelME! project is to devise and promote better software engineering practices for ISDSs in the M&E. This covers, among others, topics such as visibility and trust, safety certification, management and sharing of knowledge, and integration and commissioning.

For the first stage of the project, we chose to focus on the activities surrounding certification. This decision was in part due to the availability of a suitable industrial partner, but in larger part due to the critical need to improve the safety certification practices for ISDSs in M&E. Specifically, our current objective is to develop more rigorous and cost-effective techniques for construction and assessment of software safety cases in M&E.

In line with this objective, we have started an investigation of the challenges faced by ISDS suppliers and certifiers with regards to building and evaluation of software safety cases. While our investigation is still ongoing, some general areas of difficulty can already be noted from our initial observation of the current certification processes. We highlight these areas below:

Lack of detailed guidance. The task of interpreting and operationalizing a safety standard such as IEC 61508 is a daunting one. To apply such a standard successfully, ISDS suppliers rely primarily on the certification bodies for guidance about the development methodologies, engineering best practices, and technologies that facilitate meeting the standard's expectations. Unfortunately, the available guidance for development of ISDS software is not detailed enough to establish a direct path to compliance with the relevant standards. This gap can, to a large extent, be attributed to a lack of research on how to effectively guide ISDS software developers in their tasks.

A key area in need of more rigorous guidance is how suppliers should preserve the chain of evidence that supports their claims about safety (or more generally, dependability). This is a priority that has also been advanced in a recent report on software dependability (Jackson, Thomas et al. 2007) by the US National Research Council. Building a coherent safety case is practical only if the collection of safety information is done systematically and "during" development. Doing so "after the fact" would be far too expensive or even impractical.

From reading the standards, it is not easy to determine the traceability links that need to be kept between hazards, assumptions, development activities, design and architecture decisions, test and audit records, etc., so that the chain of safety evidence can be fully preserved. We believe that developing more concrete recommendations on preserving traceability and automated tool support to assist ISDS suppliers with traceability management would lead to significant cost-reductions in the safety certification process.

Use of text-based documents for managing safety information. It is still common to use text, possibly augmented with some diagrams, as the primary means for storage and reasoning about safety evidence. This leads to two major problems:

Firstly, text is highly prone to ambiguity, incompleteness, and redundancy. This leads to the suppliers and certifiers having to invest a substantial amount of time and human resources resolving the ambiguities, identifying and addressing the areas of incompleteness, and ensuring that repeated information across multiple documents remains consistent. Using diagrammatic illustrations that are not in standardized notations (or extensions thereof) helps little to address the problem, because the semantics of these diagrams are unknown and never stated clearly. Hence, the diagrams could well become another source of ambiguity, incompleteness, and redundancy.

Secondly, text is very difficult to query and manipulate automatically. In particular, although the majority of the information in a software safety case naturally results from regular development activities (i.e., requirement, design, and verification and validation), it takes developers significant manual effort to extract safety-related information from the documents built during these activities, and put the information in an appropriate form.

Approach

A recurring theme in the solutions we put forward is the use of standardized modelling languages such as UML and SysML to express system requirements, specifications, architectures, designs, traceability data, and information about development processes and activities. This is often referred to as Model-Driven Development (MDD) (Kleppe, Warmer et al. 2003) and provides a number of advantages when a pragmatic approach is taken. In a context where safety standards (e.g., IEC 61508) and development process standards (e.g., CMMI 2) are gaining in importance, where systems are required fast approval by certification authorities (e.g., DNV), a model-driven approach provides developers, domain specialists, and assessors with more precise descriptions of systems and more standard and structured ways to represent various development artifacts, such as requirements, specifications, designs, and test cases.

Briefly, our position is that models, and not documents, should serve as the main sources of development information – documents, when needed, should be generated from models. For the purpose of safety certification, models are beneficial in many important respects. Most notably: (1) Models can be employed to clarify the expectations of safety standards and recommended practices, and develop concrete guidelines for ISDS suppliers; (2) Models expressed in standard notations avoid the ambiguity and redundancy problems associated with text-based documentation; (3) Models provide an ideal vehicle for preserving traceability and the chain of evidence between hazards, requirements, design elements, implementation, and test cases; (4) Models present opportunities for partial or full automation of many laborious safety

analysis tasks (e.g., impact analysis, completeness and consistency checking, test case generation, etc).

Contents

USING SYSML TO SUPPORT SAFETY CERTIFICATION: A METHODOLOGY AND CASE STUDY

1. Introduction

We report on our experience and lessons learned from applying model-driven development techniques for system engineering to a control system case study. Our primary purpose in this case study is to demonstrate how state-of-the-art modeling techniques can be used to support the generation of safety cases for the purpose of certification. We propose methodological guidelines for creating models that characterize our system case study along its requirements, structure, and behaviour views. We provide a traceability mechanism to specify how these views overlap and complement one another. We utilize the traceability mechanism to provide systematic evidence that the system under analysis fulfills its safety requirements. Finally, we conclude the report with a discussion of open issues and questions, and provide a plan for our future research in this direction.

In this report, we focus on state-of-the-art, model-driven development techniques for system engineering (SysML). SysML is an extension of a subset of the Unified Modeling Language (UML) using its extension (profile) mechanisms. It is intended to provide a common language for all aspects of system engineering that supports specification, analysis, design, verification, and validation. SysML is the approved standard for system modeling by the International Council on Systems Engineering (INCOSE). Though SysML is carefully defined, little experience has been publicly reported to date regarding its use, challenges, and benefits (Friednenthal, Moore et al. 2008). This is also the case in the context of system certification. Only a few books have been recently published and their methodological support is limited. To a significant extent, the way SysML should be used is driven by specific modeling objectives, for example supporting safety certification in our case. In this report, we use a case study to investigate how SysML can be used to:

- model software systems from different views (requirements, behaviour, and structure), while the consistency between these views is maintained.
- establish traceability links for explicitly relating requirements to design.
- identify relevant design information for every system safety requirement.

We evaluate how SysML can be used and extended for supporting software safety analysis using a case study introduced in Section 2. Drawing on this experience, we provide methodological guidelines for creating different SysML views capturing

requirements, behaviour and structure in Sections 3.1 and 3.2. We establish traceability links between requirements and design views (Section 3.3) and describe what additional information software engineers should provide so that relevant design information can be extracted for the system requirements along these traceability links. We discuss shortcomings of our methodology, the SysML notation, and the SysML tool-support in Section 4. We discuss open issues and directions for our future research in Section 5. We provide a partial set of SysML diagrams for the case study in the appendix (Section 7). This set includes some representative examples for every kind of SysML diagram used in our methodology. The complete set of diagrams for this case study is available in (Klykken 2009).

2. Case Study: A Production Cell System

Reactive systems, that interact with their environment continuously, represent an important class of safety critical systems widely used in domains such as avionics, automotive, and maritime and energy. The Production Cell System (PCS) (Barbey, Buchs et al. 1998) , used in this report, is a well-known exemplar for reactive systems. The system described is this exemplar is an actual production cell in Germany, and has been previously used as a benchmark to evaluate the capabilities of various specification methods for the purpose of safety analysis and verification (Lewerentz and Lindner 1995). We have chosen to use an object-oriented analysis and implementation of PCS performed based on the Fusion method (Barbey, Buchs et al. 1998) as a baseline for our study. While this choice has some rather restrictive implications on our design (see Section 3.2 for a detailed discussion), it provides us with a source of useful information for our study. Specifically, the Fusion-based analysis of PCS contains a comprehensive and precise list of (safety) requirements, a detailed object-oriented design of this system, a complete implementation of the system generated from the design, and a set of test cases exercising the system requirements. Below, we present an overview of PCS.

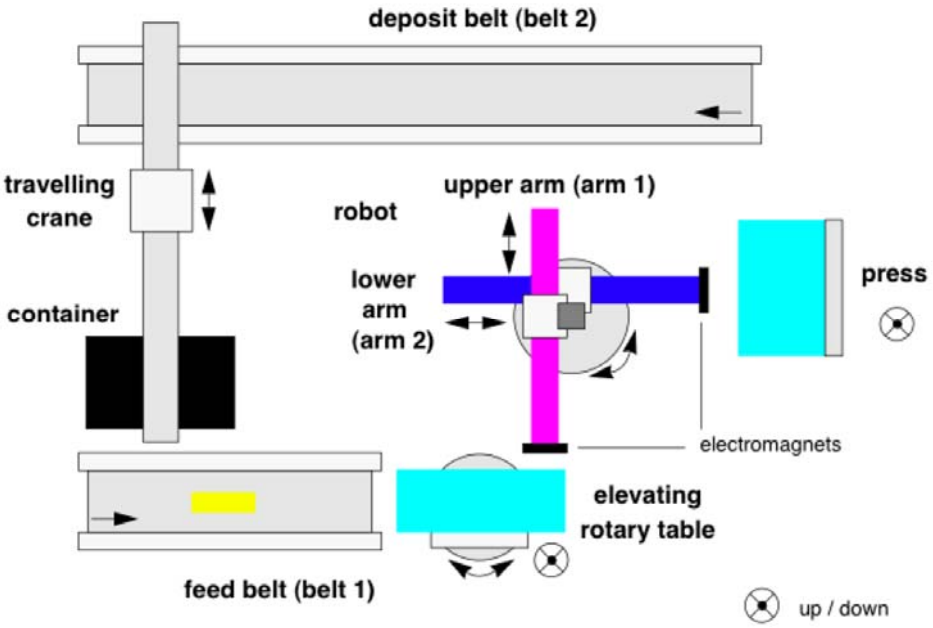

FIGURE 1. AN OVERVIEW OF PRODUCTION CELL SYSTEM (PCS).

Description of PCS. PCS is composed of six devices: two conveyors belts (feed belt and deposit belt), a travelling crane having an extendable arm equipped with electromagnet, an elevating rotary table, a press and a rotary robot having two orthogonal extendable arms equipped with electromagnet (Figure 1). The aim of the cell is the transformation of metal blanks into forged plates (by means of a press) and their transportation from the feed belt into a container.

The production cycle of each blank is the following (Figure 1):

- The feed belt conveys the blank to the table
- The table rotates and lifts to put the blank in the position where the robot is able to magnetize it,
- The first robot arm magnetizes the blank and places it into the press,
- The press forges the blank,
- The second robot arm places the resulting plate on the deposit belt,
- The crane magnetizes the plate and brings it from the deposit belt into a container.

Each of the above mechanical devices is equipped with sensors and actuators, enabling them to operate and communicate with one another. Section 7.1 in the appendix provides a detailed description of each component in PCS.

In addition to the mechanical devices shown in Figure 1, the system has an operator that is responsible for putting the blanks on the feed belt, turning on/off the system, and

performing an emergency shut down when the system is acting unsafely. Control systems often interact with other human agents such as maintainers and installers whose functions are outside the scope of this current report (see Section 5 for a discussion on future work).

3. A Methodology for Using SysML for Safety Certification

At a high level, our methodology for modeling control systems to support safety certification includes one or more iterations of the following two main phases, outlined below:

1. System requirements specification
    a. Create the *system context diagram* to determine the external entities that interact with the system, and identify the *constraints* that the external entities impose on the system.
    b. Ca*pture system-level requirements*, and specify the requirements category (Lamsweerde 2009)(e.g., safety requirement) for each individual system-level requirement.
    c. Identify top-level functionality in terms of system use cases, and relate the use cases to the system-level requirements.
2. System architecture and design interwoven with safety-related activities
    a. Create structural views of the system using *block definition diagrams* by identifying top-level software blocks and decomposing them into sub-blocks until primitive software blocks controlling sensors and actuators of the system are reached (structural)
    b. Model the use case scenarios using *sequence diagrams* to describe behavioural interactions between the top-level parts (Block instances). (behavioural)
    c. Specify *block-level requirements* and trace these requirements to the system-level requirements. Establish *traceability* links between block-level requirements and top-level blocks. (safety-related)
    d. Describe connections between the top-level system parts using *internal block diagrams*. (structural)
    e. Model sequences of activities that each top-level part performs using *activity diagrams*. (behavioural)
    f. Capture the constraints on physical properties of blocks using *parametric diagrams*. (structural)
    g. Decompose each activity in the top-level activity diagram into a sequence of *actions* performed by the primitive software blocks controlling system sensors and actuators. (behavioural)
    h. Capture the behaviour of individual top-level parts with state behaviour using *state machine diagrams*. (behavioural)
    i. Specify *pre and post-conditions* for the operations of the top-level blocks. (behavioural)
    j. Specify block-level requirements in terms of block operations, predicates involving block attributes, and interfaces between blocks using the *Object*

Constraint Language (OCL)[1] (Hans-Erik Eriksson 2003), or *activity diagrams*. (safety-related)

k. For each block-level requirement, identify *slices of design diagrams* relevant to that requirement. (safety-related)

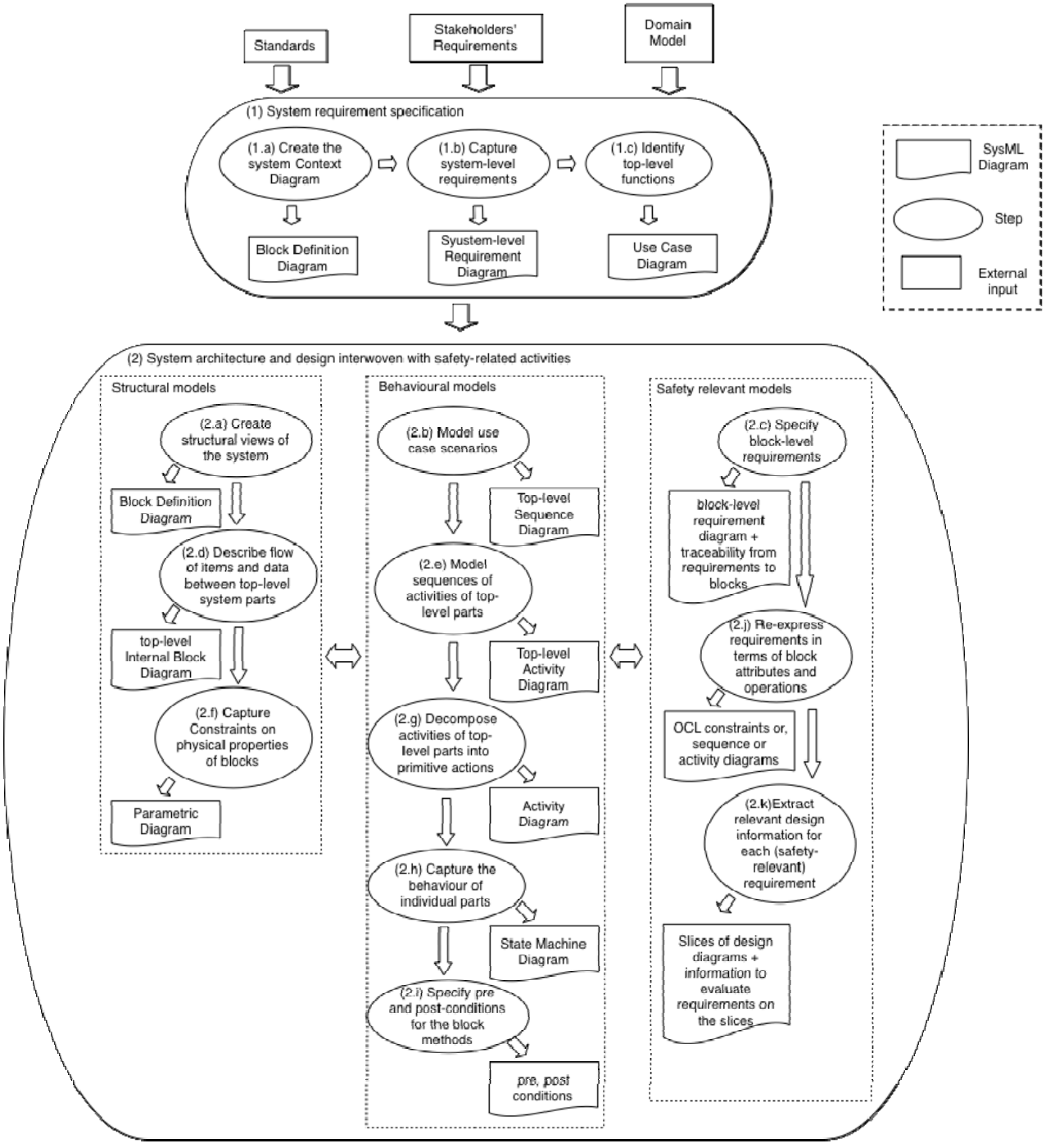

FIGURE 2. MAIN STEPS OF THE METHODOLOGY FOR USING SYSML TO SUPPORT SAFETY CERTIFICATION.

[1] OCL is a logical language for describing rules that apply to UML models developed at IBM and now part of the UML standard.

Figure 2 shows the various steps composing our methodology and the diagrams generated at each step. The methodology has two main phases: In the first phase, system required functions are identified and derived. The second phase is largely devoted to defining/synthesizing structural and behavioural aspects of the system architecture. However, since the main purpose of our methodology is to facilitate generation of safety cases, we recommend that engineers interleave certain safety-related activities with the rest of activities in this phase. For this reason, in the second phase, safety-related design activities are shown to be undertaken in parallel with the activities for creating structural and behavioural design models.

Our methodology takes as input (1) a set of standards relevant to the domain of the system under analysis, (2) stakeholders' needs and expectaions, and (3) a model capturing domain concepts and their relationships. Since in this report we focus on system engineering and domain analysis is a standard practice, we do not discuss creation of domain models. But it should be noted that they are essential inputs to our methodology and that guidelines are provided in various model-driven engineering methodologies (e.g.,(Gomaa 2000)).

Though there are many variations in the details of modeling practices, the development of structural and behavioral models (Figure 2) is common among model-based system engineering methodologies (Gomaa 2000). For software blocks, among the safety-related activities, steps (2.c) and (2.j) are typically undertaken by software engineers during which software engineers create traceability links from (critical) requirements to the design diagrams. Step (2.k), however, is the responsibility of safety engineers who use the established traceability links to extract design information relevant to each safety-related requirement, and eventually create safety cases using this information. However, traceability links from requirements to design have a more general utility regarding software reliability and quality, and hence the activities for creating traceability links in Figure 2 may be used for various purposes such as test case generation, maintenance, change management and so on. Our focus in this report is mainly on how traceability can be used for safety analysis and later on for safety case generation.

The process from requirements (phase 1) to architecture and design (phase 2) is iterative in nature and as we progressively refine the generated diagrams and augment them with more implementation-level information, we need to ensure that our refinement process is sound. We therefore need to maintain consistency between the contents of these diagrams. The arrows between steps in Figure 2 depict the consistency relations between diagrams generated at different steps. Specifically, the arrows indicate that the diagrams generated in the target step rely on the contents of the diagrams generated in the source step. These arrows suggest a sequence between steps, but it

often happens that the developers go back and forth between the diagrams generated in the source and target steps and concurrently revise them. In such cases, they must ensure that the diagrams did not become inconsistent during their revisions. Note that maintaining consistency between diagrams, in addition to ensuring the soundness of our methodology, greatly assists engineers in creating and deriving traceability links. We discuss a number of such consistency rules between design diagrams in this report.

In the rest of this section, we follow the two phases in Figure 2 successively. We discuss phase one in Section 3.1, the creation of structural and behavioural models of phase two -- in Section 3.2, and the safety-related models of phase two -- in Section 3.3. Each phase comprises several steps. We provide an ordering for following these steps such that each step relies on models generated in the previous steps. Eventually the generated models will be used either directly in the generation safety cases, or as intermediary artifacts for creation of safety case models.

3.1. System requirements specification

Understanding the problem domain and eliciting system requirements are critical initial steps in the system engineering process. Explicitly modeling problem domain concepts and system requirements provides mechanisms to better understand the requirements, reduce their ambiguity, develop a unified terminology, and validate the requirements with stakeholders to ensure the right problem is being solved.

As shown in Figure 2, this phase consists of four main steps through which we develop the following SysML diagrams: (1) A block definition diagram capturing the system context diagram in step (a), (2) A requirement diagram capturing system-level requirements in step (b), (3) A use case diagram capturing system top-level functions in step (1.c), and (4) parametric diagrams capturing environmental assumptions in step (1.d). Below, we discuss these four steps and present the diagrams generated at each step for PCS.

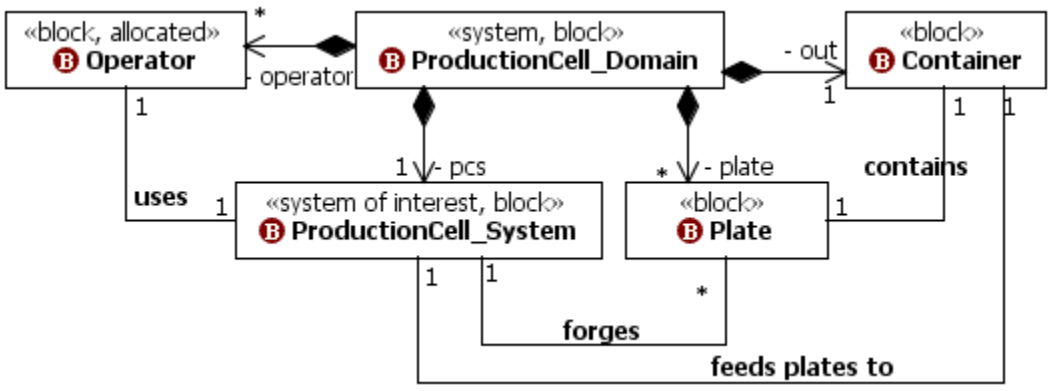

FIGURE 3. SYSTEM CONTEXT DIAGRAM FOR PCS.

Step (1.a).

Input: standards + stakeholders' needs + domain model.

Output: A SysML block definition diagram representing the boundary between the system and its environment.

General. The purpose of the system context diagram is to specify the boundary between the system and the external environment (i.e., the system context). External environment typically includes human operators, users, or hardware/software systems outside the scope of the system of interest. In UML-based MDE methodologies, context diagrams are typically represented as class/object diagrams (Gomaa 2000). In SysML, class/object diagrams are replaced by block definition diagrams. Therefore, we use block definition diagrams to capture system contexts.

Case Study. Figure 3 shows a SysML block definition diagram capturing the PCS context diagram. As shown in the figure, the diagram has a composite block specifying the PCS domain, i.e., `ProductionCell_Domain`. The domain is decomposed into the PCS system (`ProductionCell_System`), which is shown as a block with the stereotype "system of interest", and the external entities including an operator, a metal plate, and a container[2].

The PCS system block in Figure 3 includes not just the software of PCS, i.e., controller software for mechanical devices, but also its hardware parts, i.e., mechanical devices such as `FeedBelt`, `Table`, and `Robot`. The context of the system is composed of all the entities that directly interact with the system, either with hardware devices (such as metal plate and container) or with the software embedded in the devices (such as operator).

General. Of great importance to us is to explicitly specify those requirements that can be tracked back to assumptions about the system environment. This is because incorrect environmental assumptions and incorrect transition from these assumptions to system requirements may cause catastrophic system failures (Jackson, Thomas et al. 2007).

Environmental assumptions are properties of the system context (external entities) that entail additional requirements for the system. For example, plate and container are two external entities in PCS (Figure 3). Their properties, such as the weight or size of the plate or the capacity of the container, may lead to requirements on PCS. Specifically, the PCS mechanical devices should be sufficiently large and powerful to carry and transform

[2]Some elements in the SysML diagrams included in this report are annotated with the stereotype "allocate" (e.g., see block "operator" in Figure 3). Note that this stereotype is not part of our methodology. It is automatically generated by the SysML tool we use after creation of traceability links between model elements.

the plate. Also, the PCS should stop operating once it produces enough forged blanks to fill the container.

In our methodology, we express environmental assumptions using parametric constraints associated with the system context diagram (Step 1.a). This is because these assumptions are often expressed as equations over physical properties and can be best formalized using parametric constraints.

Case Study. For example, Figure 4 represents a parametric constraint describing how the size of `Plate` and `FeedBelt` must be related so that the plate can be put on the feed belt. We then need to include the constraints on the size of `FeedBelt` and other mechanical devices as system-level requirements. In this example, these constraints are about the hardware aspects of this system, and hence, imply system-level hardware requirements that are outside the scope of this report.

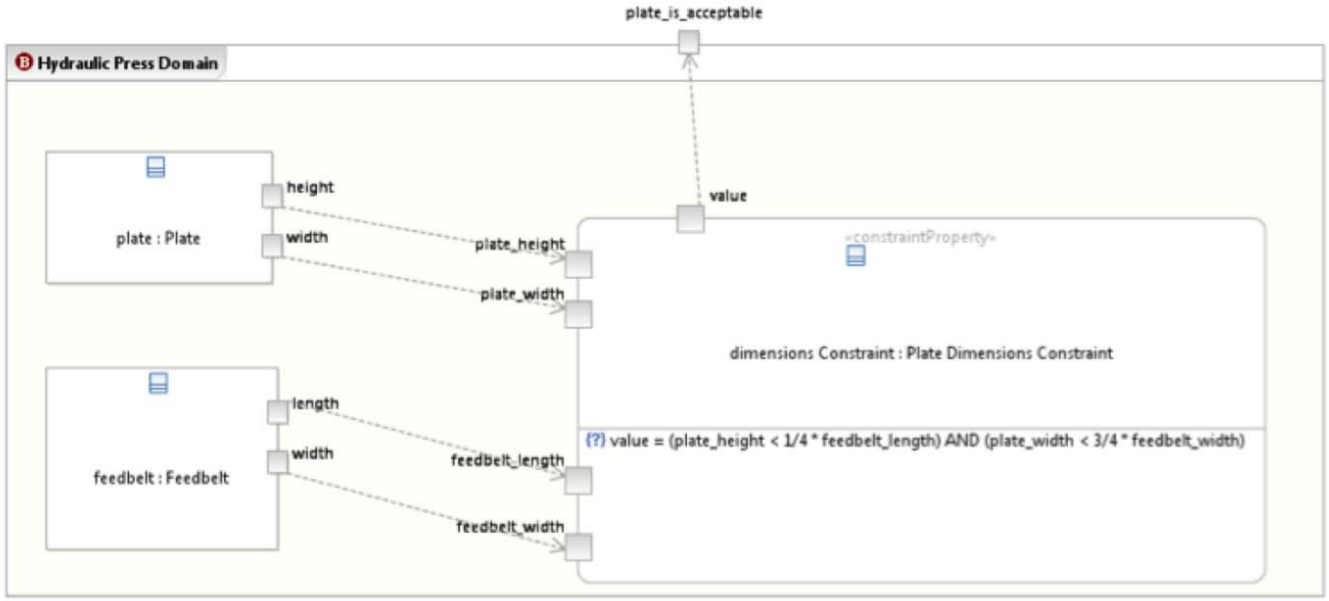

FIGURE 4. A SYSML PARAMETRIC DIAGRAM DESCRIBING AN ASSUMPTION.

Step (1.b).

Input: standards + stakeholders' requirements + domain model.

Output: A SysML requirement diagram containing system-level requirements.

General. In this step, we identify the system requirements, including the safety-relevant ones, and specify the requirements category for each individual requirement. It is very difficult to collect a complete set of system requirements. One way to decrease chances of overlooking requirements is to use standard requirement specification templates for documenting the requirements (e.g., (Lamsweerde 2009)). These

templates contain a comprehensive list of requirement categories such as functional safety, functional liveness, performance, interface, reliability, availability, design constraint, and maintainability. Each system has several requirements falling under each of these categories. A category without any corresponding requirement can be a sign of missing system requirements.

SysML allows the use of packages to organize requirements into different categories. In our work, we have used stereotypes instead because it would let us specify cases where a requirement falls under several categories.

In requirements analysis, software engineers usually perform an initial assessment of the system requirements to determine whether the requirement is feasible. If a requirement is deemed economically or technically infeasible, it will be left out of the rest of the process. The feasible requirements, then, should be analysed for testability criteria and augmented with appropriate test cases. Extending our methodology to include testing activities is left for future work.

Case Study. Figure 5 shows a (partial) SysML requirement diagram representing the initial requirements for the production cell system. Requirements (S1.0-S4.0) describe functional safety, i.e., the erroneous behaviours that the system must not exhibit. P1.0 describes a non-functional, performance requirement, and finally L1.0 is a functional liveness requirement describing the main function of the system.

The set of requirements in Figure 5 is partial as it does not cover may typical requirements categories such as maintainability, usability, availability. Due to our focus on safety certification, in this report we only consider the functional safety requirements. We will elaborate on other requirement categories in future reports.

All of the requirements in Figure 5 address the entire system, hardware or software. Hence, we refer to them as *system-level* requirements. These requirements may come from a variety of sources including, external standards, customers, future users, domain experts, environmental assumptions, existing system design, existing implementation guidelines, and frameworks.

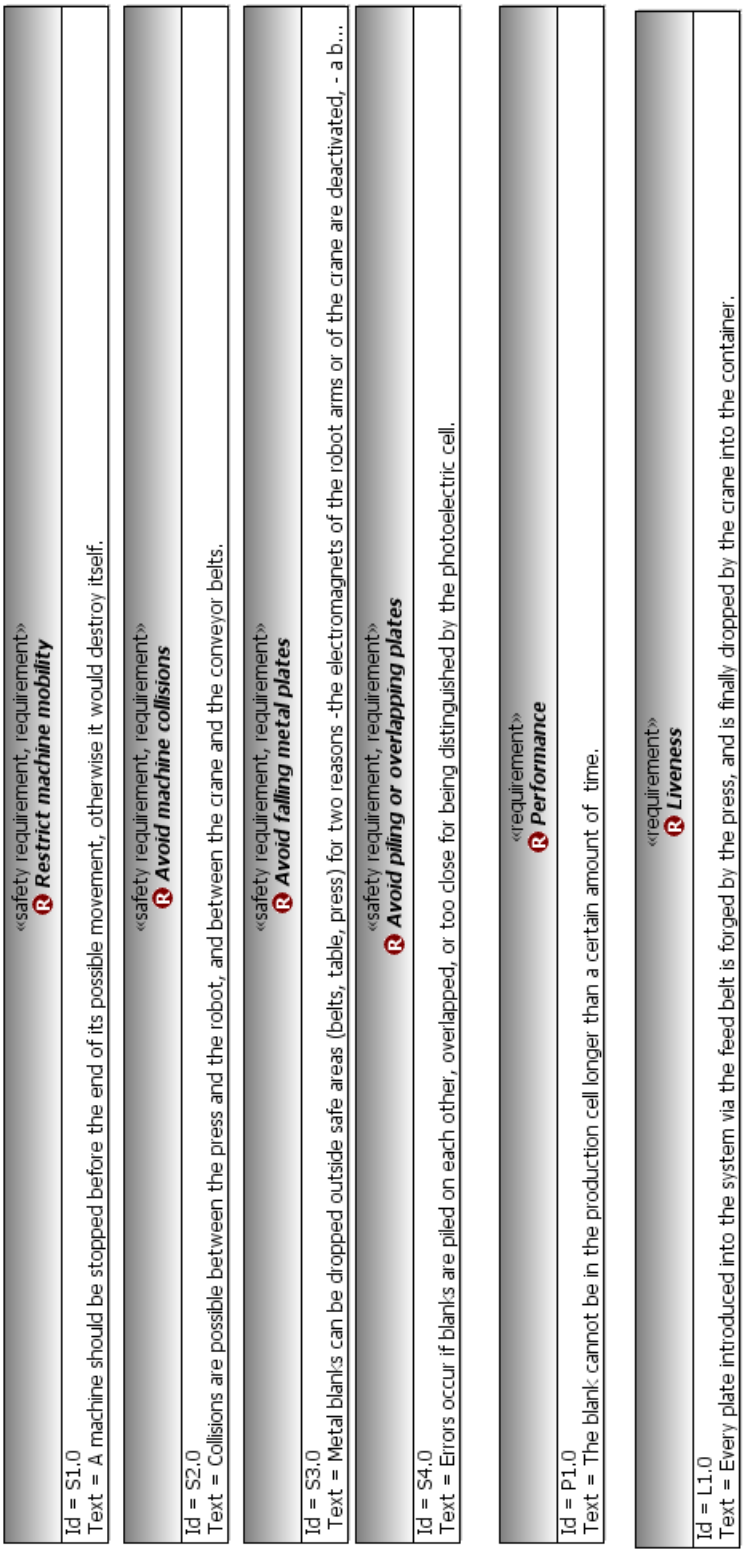

FIGURE 5. SYSTEM-LEVEL REQUIREMENT DIAGRAM FOR PCS.

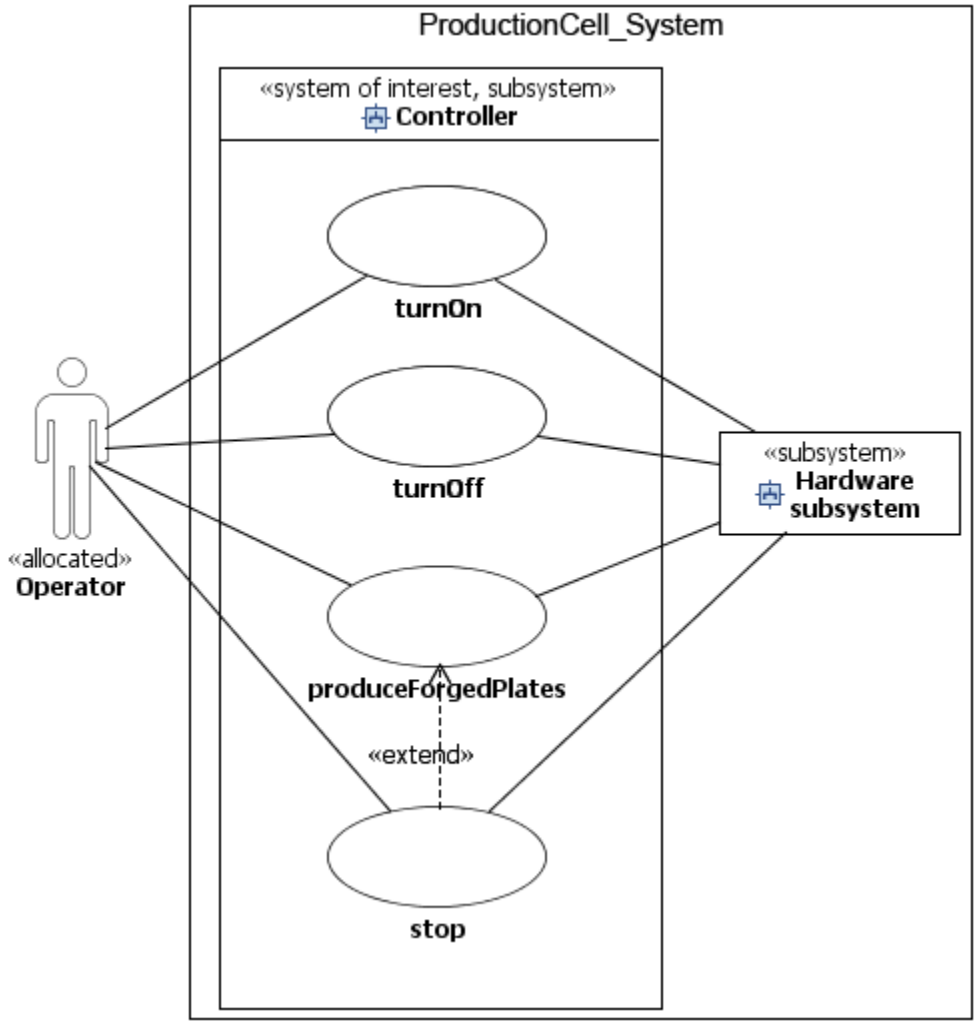

FIGURE 6. USE CASE DIAGRAM FOR THE PCS CONTROLLER.

Step (1.c).

Input: system context diagram + standards + system-level requirements + domain model.

Output: A SysML use case diagram representing top-level system functions + use case descriptions.

General. Use cases represent the functionality of the software in a system from an external point of view.

Case Study. Figure 6 shows a use case diagram for the PCS software controller. The "Hardware subsystem" in the figure represents the set of all mechanical devices of PCS. As shown in the figure, PCS has one major use case, `produceForgedBlanks`, to describe the main function of the system, and three other use cases to capture the turn on, turn off and emergency shutdown functions of PCS. Guidelines for creating use case

diagrams in SysML are similar to those in UML. For a detailed description of these guidelines see (Gomaa 2000).

Each use case has a scenario describing the use case behaviour in a step-wise manner. Use case descriptions for PCS are available in Section 7.4 of the appendix.

General. Use cases are defined to satisfy system functions. They can be directly traced to the system functional requirements that they are expected to satisfy. Non-functional requirements are also related to use cases because they constrain the behaviour of the use cases by stating the way they must be implemented.

Case Study. Figure 7 describes the relationships between the requirements in Figure 5 and the use cases in Figure 6. We annotate these relations by the stereotype “refine” because the use cases provide a more precise and operational description about the requirements. As shown in Figure 7, in PCS, all the requirements we listed are related to the main system use case: `produceForgedBlanks`. Note that control systems typically have an additional actor to represent the global system clock and additional use cases traced to system availability, maintainability, and security requirements, among others. Such actors and use cases, however, are outside the scope of this current report.

FIGURE 7 TRACEABILITY RELATIONS BETWEEN THE USE CASE PRODUCEFORGEDBLANK IN FIGURE 6 AND THE REQUIREMENTS IN FIGURE 5

3.2. Creating design and architecture models

Software design typically involves the creation of two main complementary views: structural and behavioural. Structural views describe the organization of the system in terms its constituent blocks and their interaction points, and the behavioural view describes how the blocks work together and communicate with one another to deliver functionality. Steps (2.a), (2.d), and (2.f) in Figure 2 focus on structural diagrams and steps (2.b), (2.e), (2.g), and (2.h) on behavioural diagrams. In addition to generating structural and behavioural views, we create safety-related models in steps (2.c), (2.k), and (2.j). We discuss step (2.c) in this section and the other two steps in Section 3.3. Below, we discuss these steps in the order suggested by our methodology in Figure 2.

Step (2.a).

Input: system context diagram + domain model.

Output: A SysML block definition diagram representing the system structure + system glossary.

General. 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 constituents (sub-blocks). Figure 8 represents the decomposition of the PCS into its hardware and software subsystems, and a partial decomposition of these subsystems. In our methodology, we continue the block decomposition, until we generate primitive software blocks controlling system actuators and sensors.

Generally speaking, the design of an embedded system consists of three major sub-systems: (1) hardware components, (2) control software, and (3) drivers that enable the communication between hardware and software. In our baseline study (Barbey, Buchs et al. 1998), drivers were replaced by a simulator. Following our methodology we have not show the drivers either.

In addition to identifying blocks and the hierarchical block decomposition, we need to identify relationships between blocks, and block attributes (see Figure 39 in the appendix). In object-oriented methodologies such relationships include generalizations, associations, and dependencies. For this purpose, we use the same guidelines provided for identifying relationships and attributes as in UML class diagrams (Gomaa 2000).

Finally, block definition diagrams are accompanied by a document describing the blocks and the relationships between the blocks. This document is referred to as system *glossary*.

Case Study. Figure 8 shows the hardware subsystem (`ProductionCell_HW`), and the software controller subsystem (`ProductionCell_Controller`). The diagram would have included an additional subsystem for drivers, should we have chosen to model them as well.

In this study, we focus on software, and therefore, the design and analysis of the PCS controller, that is the `ProductionCell_Controller` block in Figure 8. To avoid clutter, in this figure, we have shown the decomposition hierarchy of software parts for the `FeedBelt` and `Table` blocks only. The complete block decomposition for the production cell controller is available in Figure 39, in the appendix.

We refer to the blocks that are not at the *leaf-level* as composite blocks, and to those blocks that are at the first level of decomposition, i.e., `FeedBelt, Table, Robot, Press, DepositBelt`, and `Crane`, as *top-level* system blocks.

In SysML, blocks are defined very generally as structural units such as a system, hardware, software, data elements or other conceptual entities (Friednenthal, Moore et al. 2008). The blocks in Figure 8 correspond to software or hardware units. In particular, each hardware block represents one mechanical device, and each software block can be mapped to a software class in the code. Note that this choice is carried over from our baseline study, and hence, is specific to this relatively simplistic case. If we had followed common object-oriented design practices in a more realistic system, software blocks at the design level would correspond to a cluster of software classes/units in the code. In practice, common design methodologies suggest to create separate controller, interface, and entity classes corresponding to entities in the domain (Gomaa 2000). Whereas, in the design shown in Figure 8, where we follow faithfully our baseline design, there is a one-to-one correspondence between software classes (blocks) and mechanical devices and all the activities related to each device (including control and interface activities) are bundled together in one class. Note that though this design choice, while not entirely following modern object oriented practices, did not interfere with the main goals of our study.

A glossary for PCS, describing the blocks and the relationships in Figure 8, is provided in Section 7.5 in the appendix.

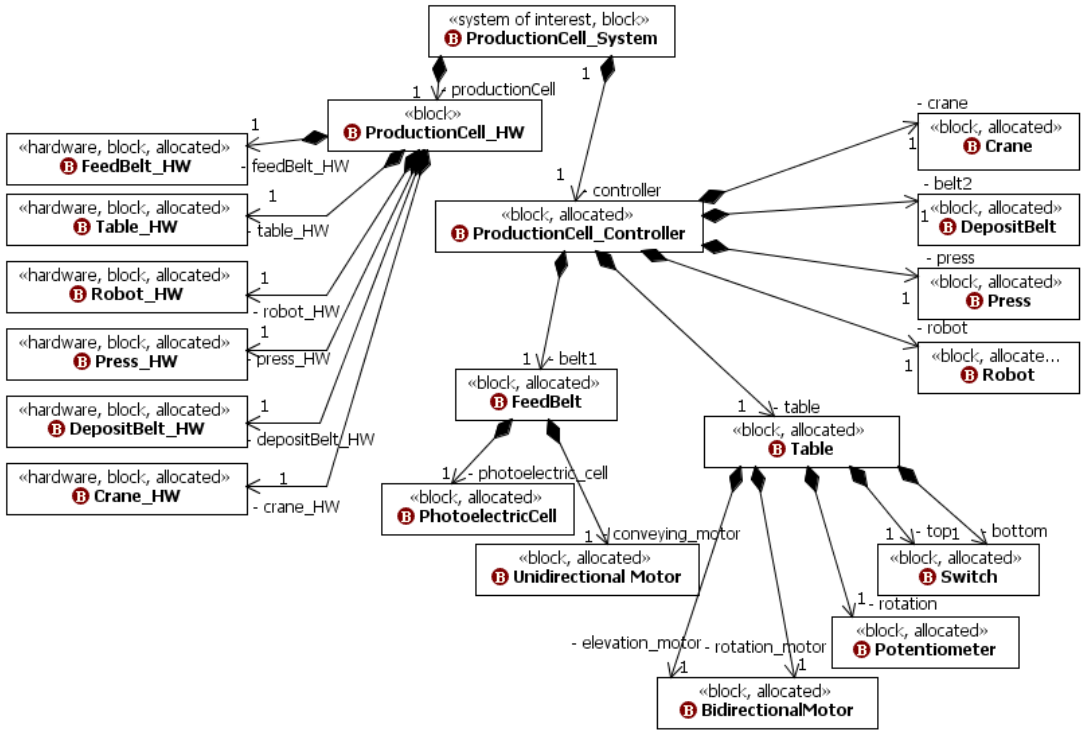

FIGURE 8. AN EXCERPT OF THE BLOCK DEFINITION DIAGRAM REPRESENTING THE STRUCTURE OF PCS.

Step (2.b).

Input: use case diagram + use case descriptions + system structure diagram

Output: Sequence diagrams describing interactions between top-level system parts (Block instances).

General. The syntax and semantics of SysML sequence diagrams are generally the same those of UML sequence diagrams (Gomaa 2000). The only difference is that in UML sequence diagrams describe interactions between objects, whereas SysML sequence diagrams describe interactions between system parts, i.e., block instances. In our methodology, we use sequence diagrams to capture interactions between top-level system parts only. Each interaction in a sequence diagram triggers some operation of a top-level part. Since the top-level parts are often composite, they need to delegate some tasks to their sub-parts or receive some information from them to perform their operations. In our work, we use activity diagrams to describe the operations of each top-level part and the decomposition of these operations into sequences of actions of sub-parts. We will discuss the activity diagrams in steps (2.g) and (2.h).

The reason that we use sequence diagrams only for top-level parts is that they would tend to become cluttered and unmanageably wide if all interactions between parts and

their sub-parts were shown in one diagram. This is more due to a tool problem rather than the problem with the sequence diagram formalism. Ideally, the tool should allow use to create sequence diagrams at different levels of granularity with one diagram describing the top-level interactions and several other diagrams describing the break-down of the top-level interactions. Unfortunately, our modeling tool, Rational Software Architect (RSA), does not support the operation break-down for sequence diagrams. This is the main reason as to why we decided to use activity diagrams, instead of sequence diagrams, for describing detailed interactions between parts.

Case Study. Figure 9 represents a fragment of a sequence diagram describing interactions between top-level system parts of PCS. A complete version of this diagram is given in Figure 29, in the appendix. Top-level parts of PCS are instances of the top-level PCS blocks. Typically, each use case has one *normal* sequence diagram describing the main function of that use case and several *alternative* sequence diagrams describing exceptional scenarios of that use case. The diagram in Figure 9 is the normal sequence diagram for the `produceForgedBlank` use case in Figure 6.

In this case study, top-level software blocks control independent and distributed mechanical devices. For such systems the most common way of communication is the asynchronous mode of communication. In this mode of communication, the event will be sent even though the receiving block may not be ready to accept the event. To model this mode of communication we need to show the signals (Hans-Erik Eriksson 2003) communicated between the system blocks. On receipt of a signal, if the receiving part is not ready to react immediately, the signal is buffered. Once the part is ready, it responds to the signal by invoking its appropriate operation.

Once we create sequence diagrams, we can identify operations for each part and allocate these operations to the corresponding blocks in the block definition diagram. For example, the `Add_Blank` signal in Figure 9 triggers `add_blank`() operation of the block `FeedBelt`, and `Go_Unload_Position` signal in Figure 9 triggers `go_unload_Position`() operation of the `Table` block. As shown in the complete block definition diagram of the production cell system in Figure 39, in the appendix, the `add_blank`() and `go_unload_position`() operations are allocated to `FeedBelt` and `Table` blocks, respectively.

Consistency. We conclude this step by outlining the consistency rules between sequence diagrams and block definition diagrams or internal block diagrams:

- Identified operations on a lifeline should be added as operations to the block that the part instantiates.
- Received events should be included in the block's provided interfaces.
- Sent events should be included in the block's required interfaces.

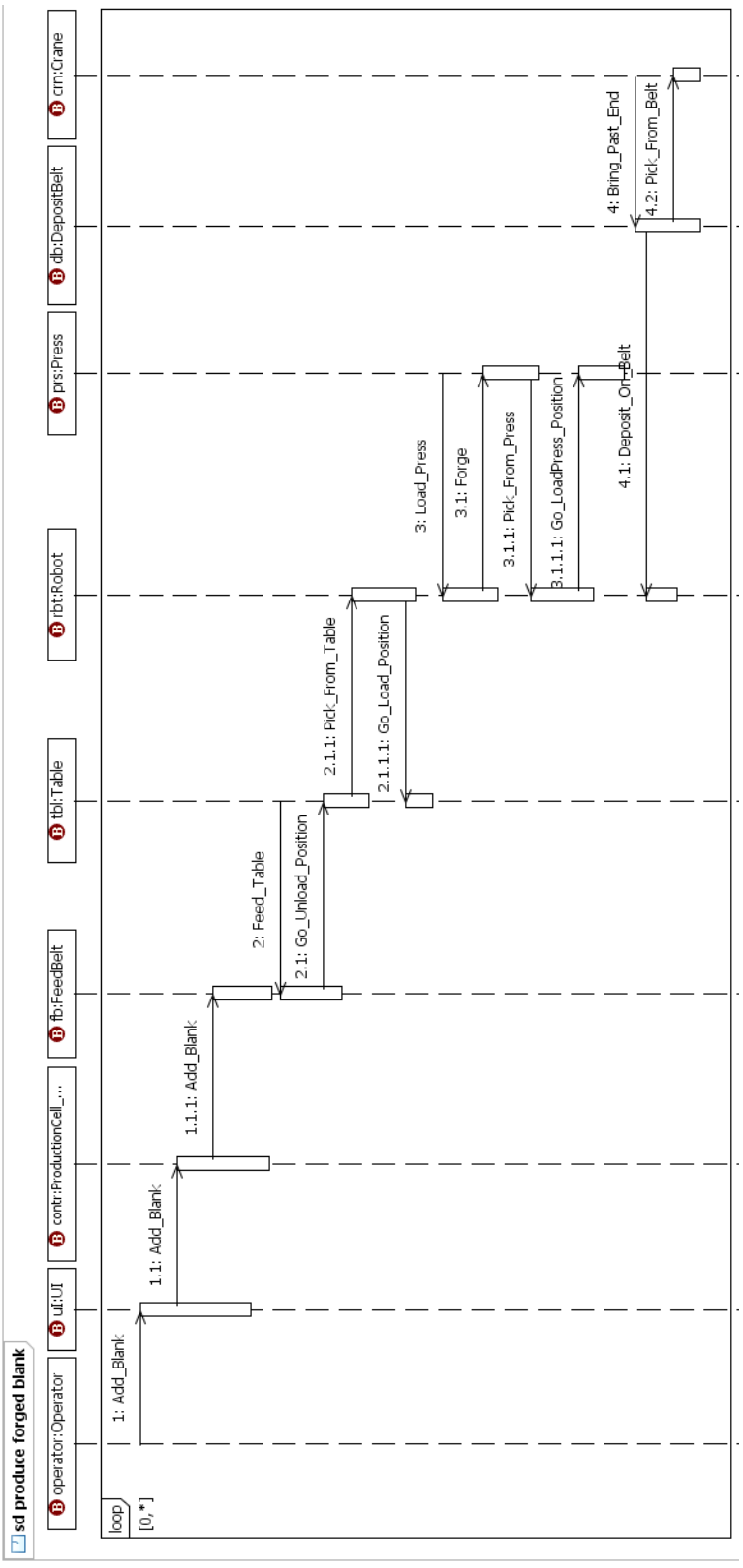

FIGURE 9. EXCERPT OF THE SEQUENCE DIAGRAM REPRESENTING INTERACTIONS BETWEEN TOP-LEVEL PARTS OF PCS FOR THE USE CASE "PRODUCEFORGEDBLANKS" IN FIGURE 6.

Step (2.c).

Input: system-level requirements diagram + system structure diagram.

Output: A requirement diagram containing block-level requirements, and traceability links from these requirements to blocks.

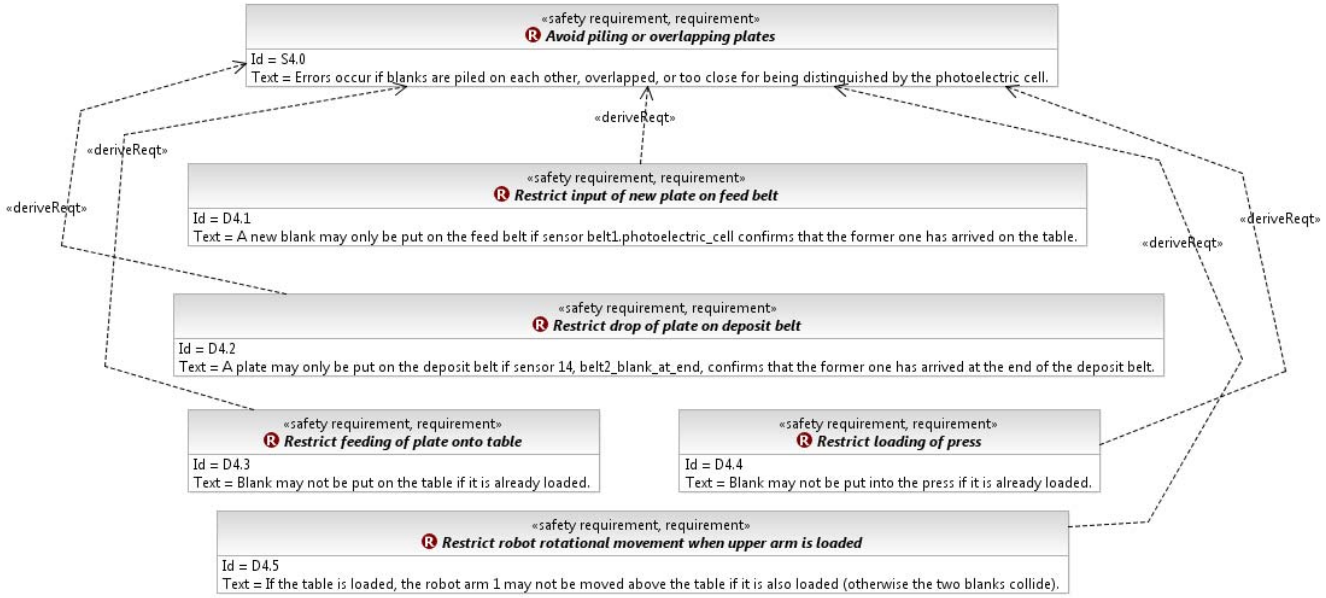

FIGURE 10. DECOMPOSING SYSTEM-LEVEL REQUIREMENT S4.0 IN FIGURE 5 INTO BLOCK-LEVEL REQUIREMENTS.

General. Having decomposed the PCS system into its constituent blocks, we try to decompose the system-level requirements in a similar manner. The goal of this step is to decompose system-level requirements into lower-level requirements that can be traced to a single or to a small set of blocks. General guidelines for requirements decomposition often do not exist, as this task requires domain-specific knowledge. One way to assist engineers in identifying blocks potentially relevant to a system-level requirement is to trace down the requirement to use cases, and then further down to sequence diagrams related to those use cases. The blocks corresponding to the parts appearing in the sequence diagrams are potentially responsible for fulfilling the system-level requirement of interest.

Case Study. Figure 10 represents decomposition of the system-level requirement, S4.0, into block-level ones: The system-level requirement concerns avoidance of piling or overlapping plates during the production of metal plates, and the block-level ones describe what specifically each top-level system block should perform to prevent piling or overlapping of plates. This requirement is related to the `producedForgeBlank` use case

in Figure 6 whose corresponding sequence diagram in Figure 29 includes all system parts including those that can be traced to the block-level requirements in Figure 10.

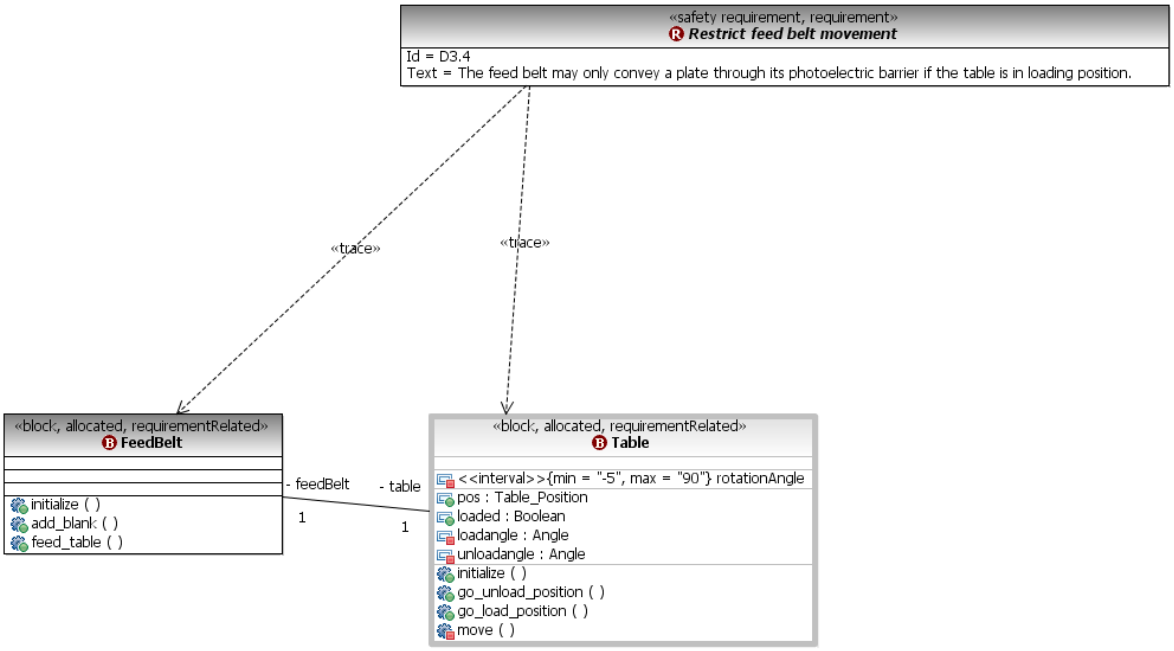

FIGURE 11. TRACEABILITY BETWEEN THE BLOCK-LEVEL REQUIREMENT D3.4 AND THE BLOCKS TABLE AND FEEDBELT.

As mentioned above, the block-level requirements are derived such that they refer to a single or a hand-full of system blocks. Therefore, it is rather straightforward to make links between the requirements and the blocks explicit using SysML traceability links. Figure 11 shows how a block-level requirement is traced to a system block. This requirement explicitly refers to `FeedBelt` and `Table` blocks, which are top-level blocks in PCS. We use the SysML general-purpose traceability links to connect the requirements and blocks.

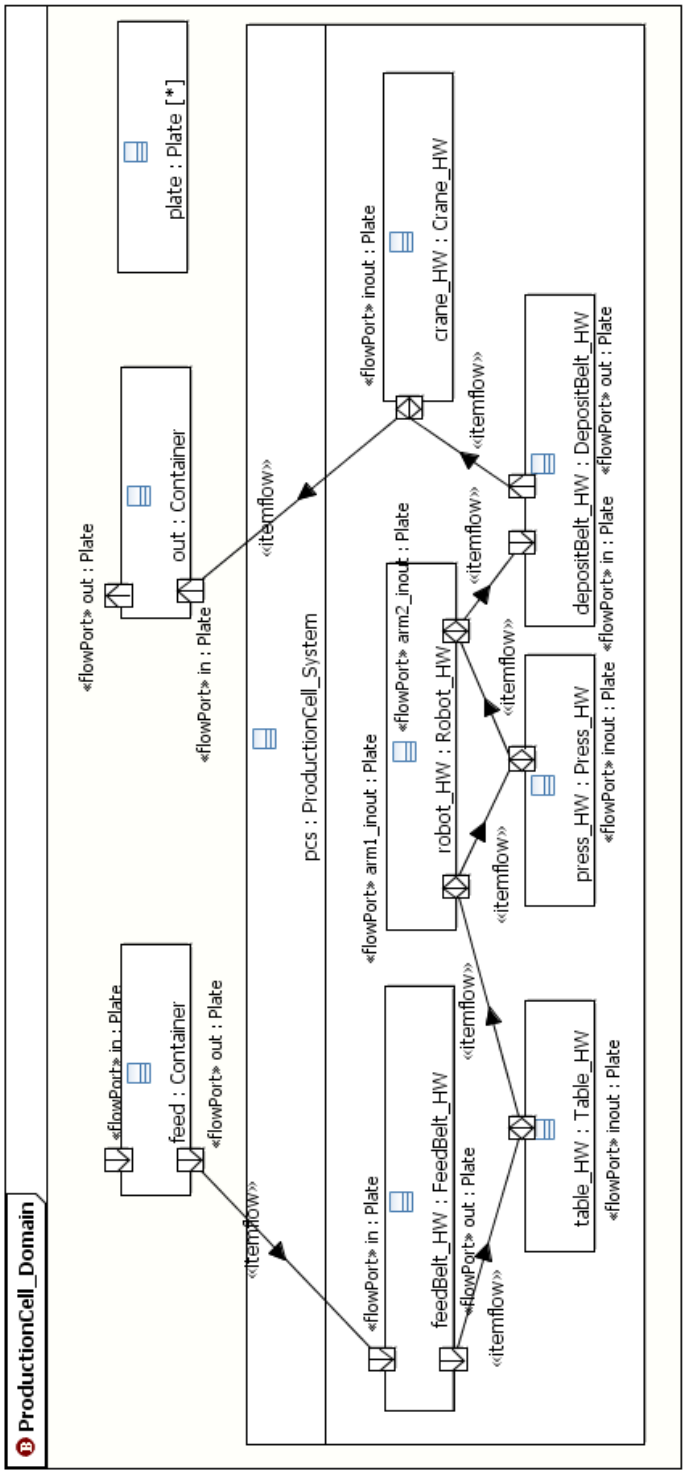

FIGURE 12. INTERNAL BLOCK DIAGRAM DESCRIBING FLOW OF THE BLANK BETWEEN HARDWARE PARTS OF THE PRODUCTION CELL SYSTEM.

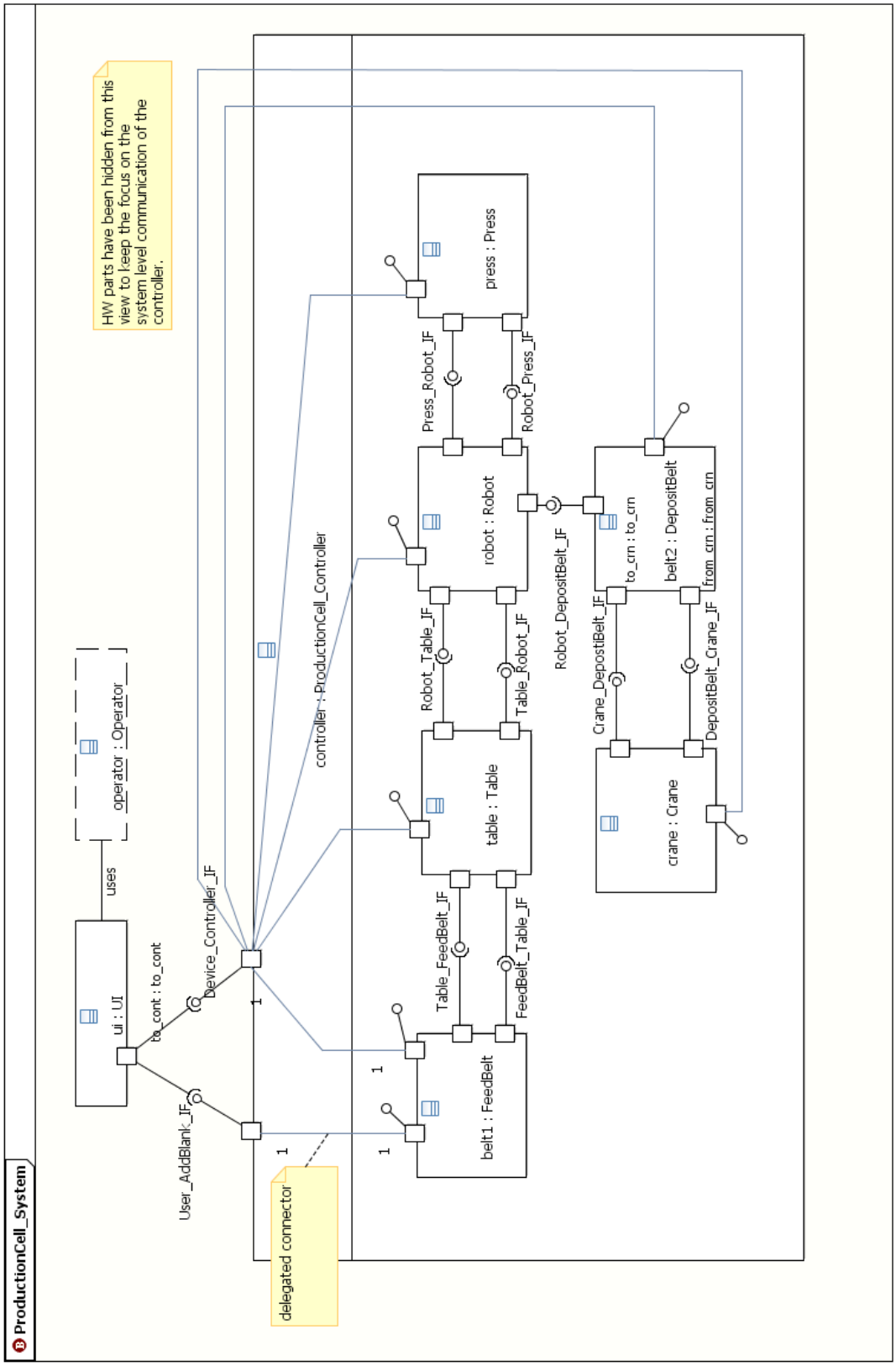

FIGURE 13. INTERNAL BLOCK DIAGRAM DESCRIBING INTERFACES BETWEEN SOFTWARE PARTS OF THE PRODUCTION CELL CONTROLLER.

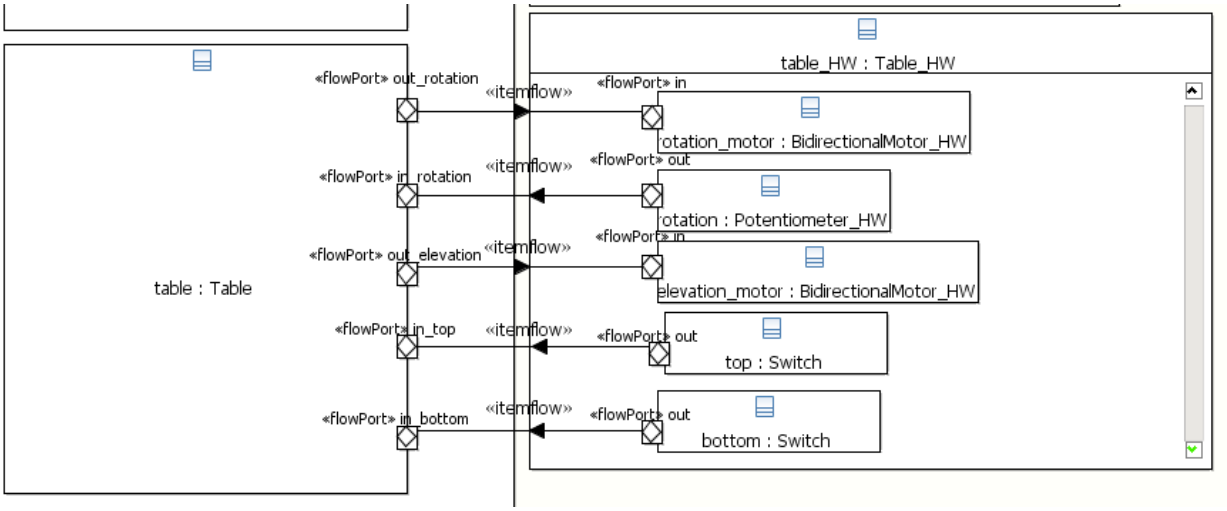

FIGURE 14. INTERNAL BLOCK DIAGRAM DESCRIBING INTERFACE BETWEEN SOFTWARE AND HARDWARE BLOCKS OF PCS.

Step (2.d).

Input: system structure diagram + sequence diagrams.

Output: Internal block diagrams describing flow of physical items or data between parts of system blocks.

General. In the system structure diagram developed in step (2.a), we defined the hierarchical decomposition of Production Cell System (PCS) in to its basic blocks. We now specify the internal structure of a composite block by describing the connections/interactions between the parts of a composite block and the usage of the parts. The diagrams that can capture such information are internal block diagrams. These diagrams describe how the parts of a composite block are connected for each usage of that block. The connections between system parts are defined through two kinds of connectors: (1) physical connections: the flow of physical items or data using flow ports and item flows, and (2) logical connections: the connections between software parts captured by provided/required services, as in UML 2.0, using interfaces.

The main role of the internal block diagrams is to provide a structural view of the system parts and their interactions. The need for such diagrams becomes more explicit in cases where several parts are instances of the same system block. These parts may exhibit different interfaces, or may be connected to other parts through different ports, or may play different roles in the system.

Lack of knowledge about system interfaces is one of the main sources of problems in integration of control systems. To avoid such problems, safety certification standards such as IEC 61508 require suppliers to explicitly articulate interfaces of the sub-systems of their products and dependencies between these sub-systems. Internal block diagrams are meant to capture such information, and hence, are often needed to be included in the certification documents.

In addition, in system engineering, software components may be deployed in different design configurations, resulting in different product applications. It is important to individually model each individual deployment as different deployments exhibit different functions. Internal block diagrams can be used to model different system deployments.

Case Study. Figure 12 represents an internal block diagram describing the connections between the hardware parts in PCS, described using itemflows and flow ports. The diagram shows the flow of blank, which is a physical object, between the system parts. In contrast, Figure 13 represents the communication between software parts of PCS, described using provided/required interfaces and standard ports. In Figure 13, the interfaces attached to the communication ports specify the operations involved at each interaction point and the direction of the interaction.

In addition to the diagrams in Figure 12 and Figure 13, which respectively describe interactions between hardware parts and between software parts, Figure 14 shows the connections between hardware and software parts of the system. The figure, specifically, focuses on the connections between hardware and software parts of the table. Such diagrams are particularly useful to show what hardware parts are controlled by the software, i.e., those that act as actuators, and what hardware parts provide input to the software, i.e., those that act as sensors. For example, in Figure 14, the `top` and `bottom` switches, and the `rotation` potentiometer are sensors, and the `elevation` and `rotary` motors are actuators. Note that as mentioned in step (2.a), we have decided to abstract away from drivers, and hence in our diagram hardware and software parts are directly connected. Otherwise, we would have needed to include drivers between hardware and software parts.

In our methodology, we create at least three internal block diagrams: one describing communications between hardware parts (Figure 12), one describing communications between software parts (Figure 13), and the last one describing communications between hardware and software parts (Figure 14). Each internal block diagram is attached to a system block and embodies parts corresponding to sub-parts of that block. For example, the diagrams in Figure 12, Figure 13 and Figure 14 are related to the blocks `ProductionCell_HW`, `ProductionCell_Controller`, and `ProductionCell_System`, respectively. These diagrams should account for all the use cases relevant to that block,

and all the scenarios within those use cases including normal and alternative ones. The connectors between the parts in each internal block diagram are determined by the message (signal) communicated in sequence/activity diagrams in which those parts participate.

Consistency.

- The parts shown in an internal block diagram of a block are instances of the sub-blocks of that block in the block definition diagram. Furthermore, in the block definition diagram, each block is related to its sub-blocks via composition relations annotated with role names on the side of the sub-blocks. The name of each part in an internal block diagram is the same as its role name in the block definition diagram.
- 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.
- 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.

Steps (2.e) and (2.g).

Input: sequence diagrams + system structure diagrams + block-level requirements diagrams.

Output: Activity diagrams describing interactions between system parts.

General. In these two steps, we use activity diagrams to capture the decomposition of the operations of the top-level parts into sequences of operations of their sub-parts. The number of decomposition levels between activity diagrams is the same as the number of block decomposition levels in the structure diagram (block definition diagram).

We could use sequence diagrams for describing decomposition of sequence operations as well because sequence diagrams and activity diagrams are equivalent formalisms for expressing interactions between system parts. But for reasons mentioned in step (2.b), we decided to use sequence diagrams for describing top-level interactions and activity diagrams for capturing decomposition of top-level interactions to sequences of low-level operations.

Case Study. Figure 15 shows an activity diagram representing sequences of activities that `Table` and `FeedBelt` perform. Figure 53 to Figure 56 in the appendix represents the sequences of activities for the rest of top-level PCS parts.

For example, the sequence diagram in Figure 9 shows that `FeedBelt` and Table communicate through signals `Feed_Table` and `Go_Unload_Position`. It further shows that Table performs certain operations in response to `Go_Unload_position` and `Go_Load_Position`, and `FeedBelt` performs some operations in response to `Feed_Table`

and `Add_Blank` signals. All this information is explicitly represented in the activity diagram in Figure 15. Note that to avoid clutter the `initialize` activities of `FeedBelt` and `Table` in Figure 15 are not shown in the sequence diagram in Figure 9 (see Figure 30 in the appendix for the complete version of the sequence diagram).

Each activity in Figure 15 can be decomposed into a sequence of operations of the low-level parts. For example, Figure 16 shows the sequence of the operations that the sub-parts of `FeedBelt`, i.e., `conveying_motor` and `photoelectric_cell`, should perform to fulfill the `add_blank` activity of `FeedBelt` in Figure 15.

In activity diagrams, the actions of each part must be added as operations to the block corresponding to that part. For example, the `FeedBelt` block has an operation `add_blank()` corresponding to the activity `add_blank` (Figure 15). Similarly, the `UniDirectionalMotor` block has an operation `turn_on()` because the `conveying_motor` part (in Figure 16) has a `turn_on` activity.

Some of the signals in activity diagrams are indeed triggers for the activities, e.g., the `Feed_Table` signal is a trigger for the `feed_table()` operation of `FeedBelt` in Figure 15. These signals are mainly added to our design to enable the asynchronous mode of communication between the top-level system parts. In a synchronous mode of communication, we would not need such signals – Table could simply call the operation `feed_table()` of `FeedBelt` directly. We do not add these signals to the blocks because their corresponding activities are already added as block operations.

The signals that do not act as activity triggers exist in our activity diagrams as well. These signals are often used to model the communication between software sensor blocks and input devices. For example, the signal `Sensor_TurnedOn` in Figure 16 is sent by a sensor to the `photoelectric_cell` part of `FeedBelt`. In our methodology, we assume that these signals set block attributes to a specific value. For example, the `Sensor_TurnedOn` signal causes the `value` attribute of photoelectric_cell to be set to `true`. Similarly, `photoelectric_cell` can receive a `Sensor_TurnedOff` signal which causes its `value` attribute to become `false`.

Consistency.

- Every operation of a top-level block should appear as an activity for that block and vice versa.
- Send Signal actions in a partition should be part of a signal reception described in the owning block/part's required interface.
- Accept Event actions in a partition should be part of a signal reception in the owning block/part's provided interface.

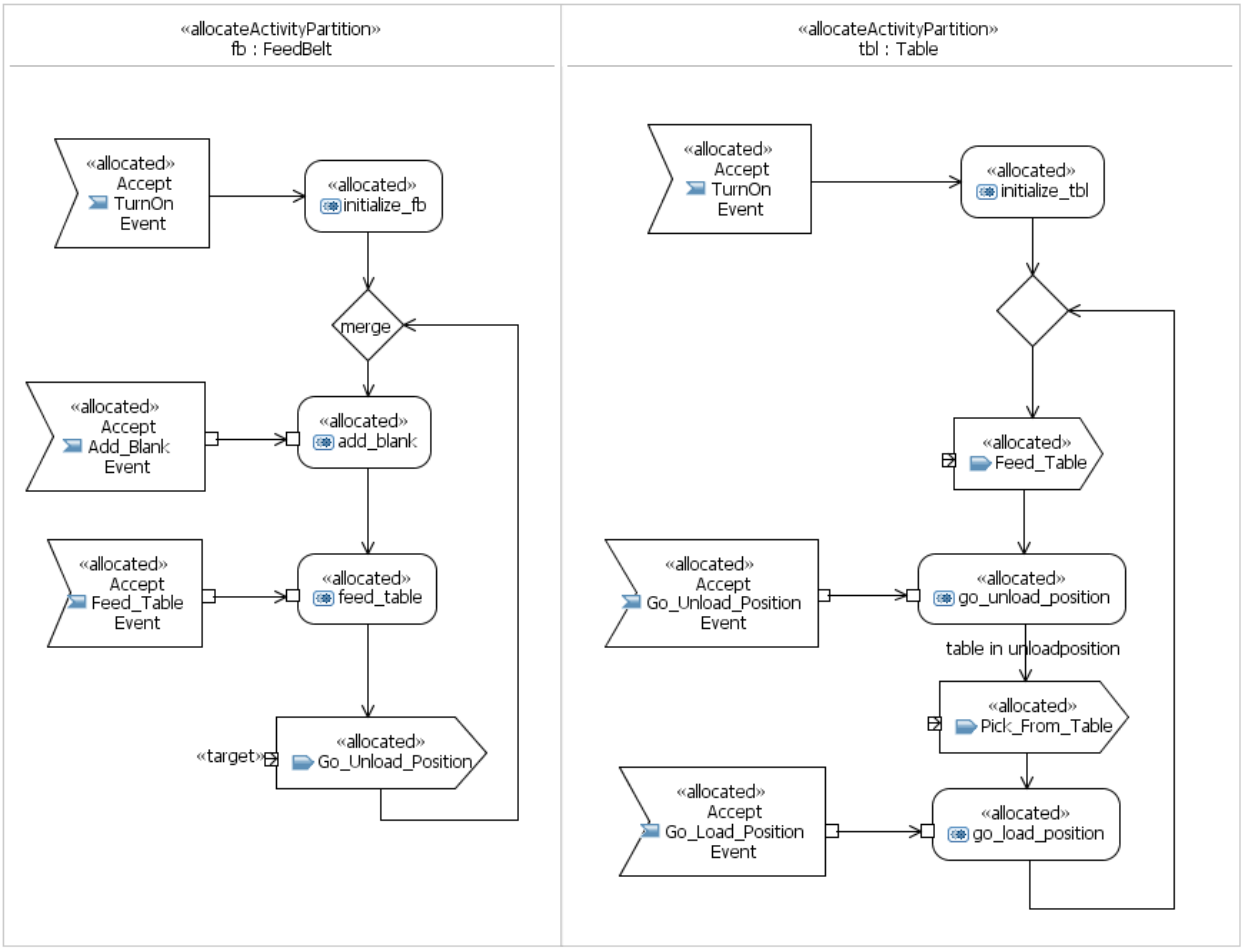

FIGURE 15. TOP-LEVEL ACTIVITY DIAGRAM REPRESENTING INTERACTIONS BETWEEN FEEDBELT AND TABLE PARTS OF PCS.

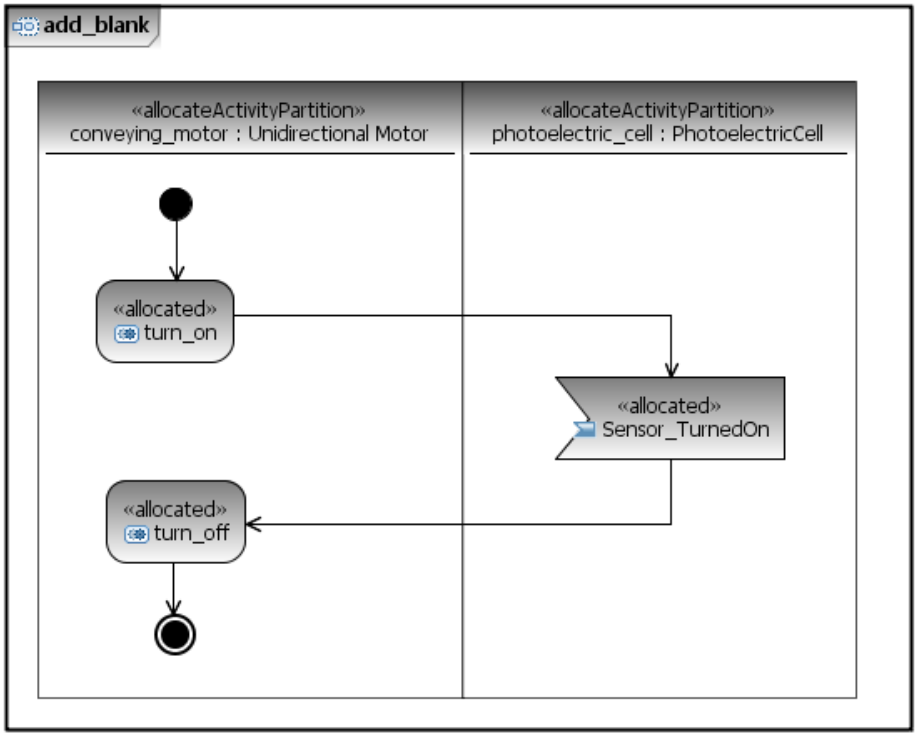

FIGURE 16. AN ACTIVITY DIAGRAM REPRESENTING HOW SUB-PARTS OF THE FEEDBELT INTERACT WITH ONE ANOTHER TO FULFILL THE ADD_BLANK ACTIVITY OF FEEDBELT SHOWN IN FIGURE 15.

Step (2.f).

Input: block-level requirements + internal block diagrams.

Output: Parametric SysML diagrams describing constraints on physical properties of the blocks

General. Activity and sequence diagrams constrain the behaviour of software blocks, but sometimes we need to put constraints on the values of the block attributes.

Case Study. One of the PCS requirements is "The magnet of the crane is not allowed to knock against the deposit belt or the container laterally". When the crane is on top of the deposit belt, it extends its arm to pick the plate. Then it starts moving toward the container, and in parallel it starts retracting its arm. To avoid collision, the crane's arm must be completely retracted before the crane reaches the container edge. This implies a constraint over the crane speed (`sp_h`), the crane's arm speed (`sp_arm_v`), the distance from deposit belt to the container's edge (`safe_dist`), and the length of the crane's arm (`ext_max-ext_min`). We use the SysML parametric diagram shown in Figure 17 to express this constraint. In our design, the crane's speed and the crane's arm speed are properties of the hardware parts of the crane. Therefore, we chose to associate this diagram to the crane hardware block (`Crane_HW`) as opposed to the crane software block.

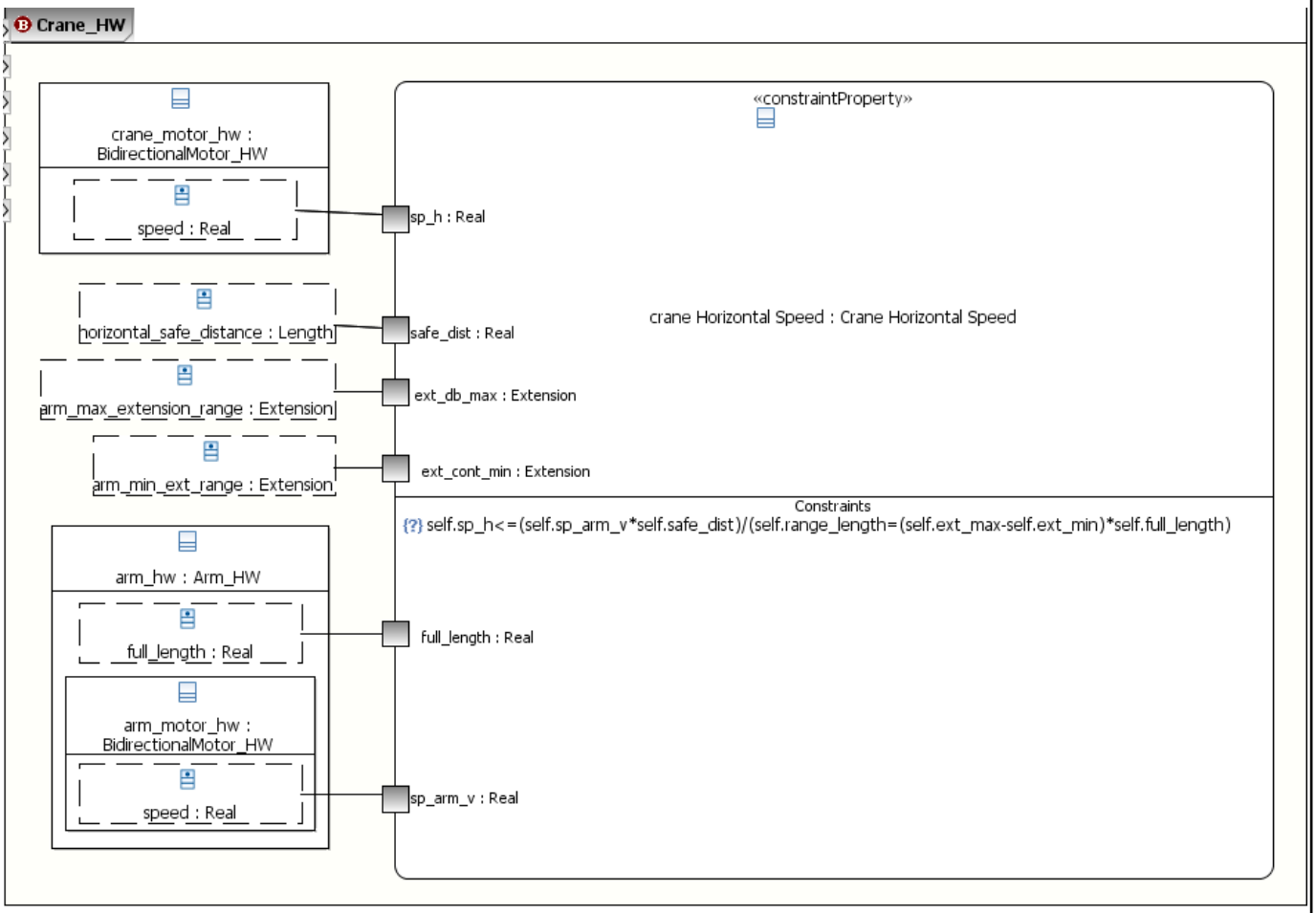

FIGURE 17. A PARAMETRIC DIAGRAM DESCRIBING THE RELATIONSHIP BETWEEN THE SPEED OF THE CRANE AND THE SPEED OF EXTENTION/RETRACTION OF ITS ARM.

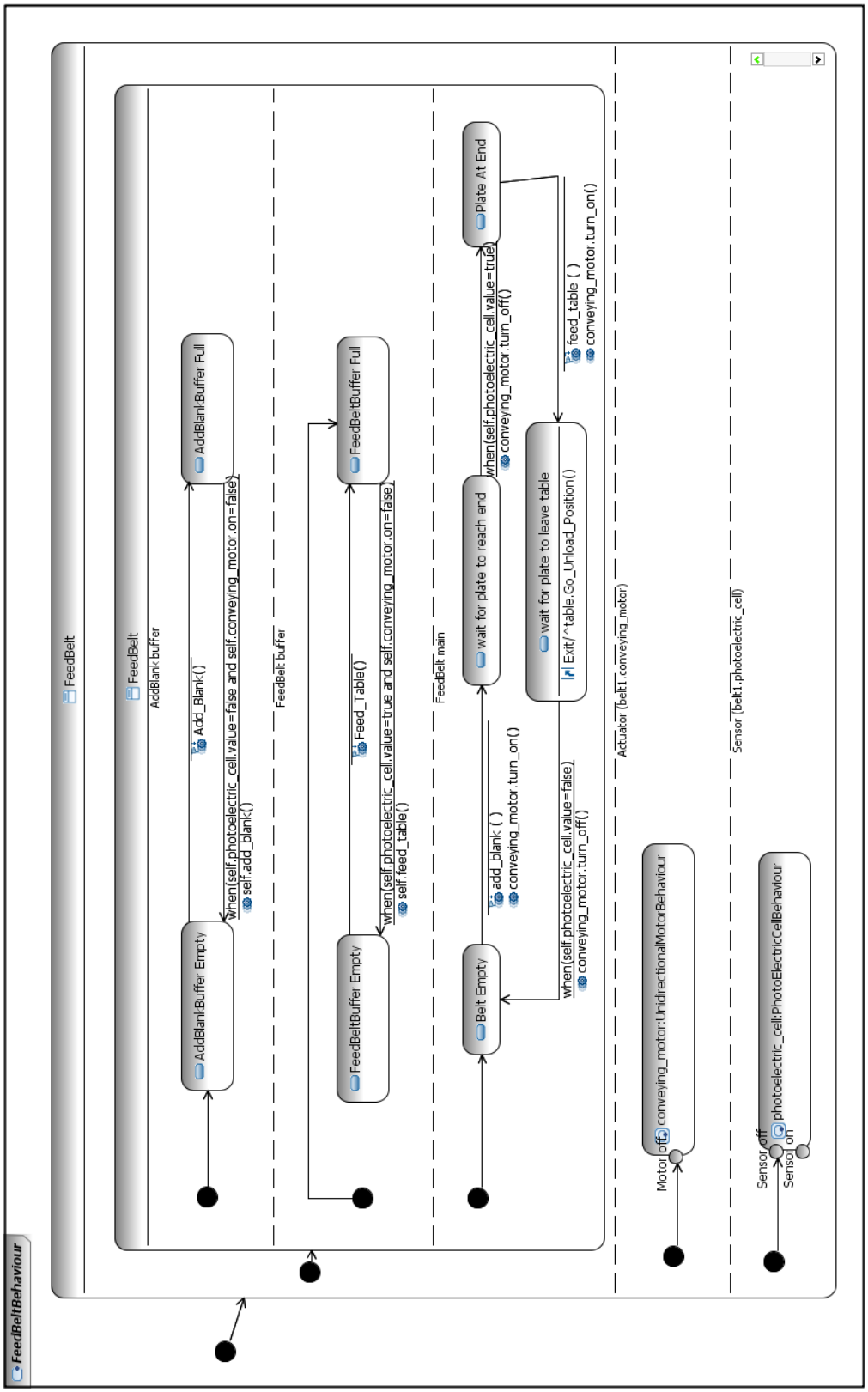

FIGURE 18. A STATE MACHINE DIAGRAM FOR FEEDBELT.

Step (2.h).

Input: activity diagrams + internal block diagrams.

Output: A state machine diagram for each system block with state behaviour.

General. In our methodology, we create one state machine diagram for each block with control behaviour. In our case study, every block controls a mechanical device, either a composite device such as a table or a sensor/actuator of a composite device. Therefore, we need to create a state machine diagram for each of these blocks.

Case Study. For the leaf-level blocks, i.e., actuators and sensors, the state machines are flat. For example, Figure 19 shows a state machine for the `Switch` block. In contrast, the state machines for the top-level blocks are composite. For example, Figure 18 shows a state machine diagram for the `FeedBelt` block. As shown in the figure, this state machine is a composition of several sub-state machines explained below.

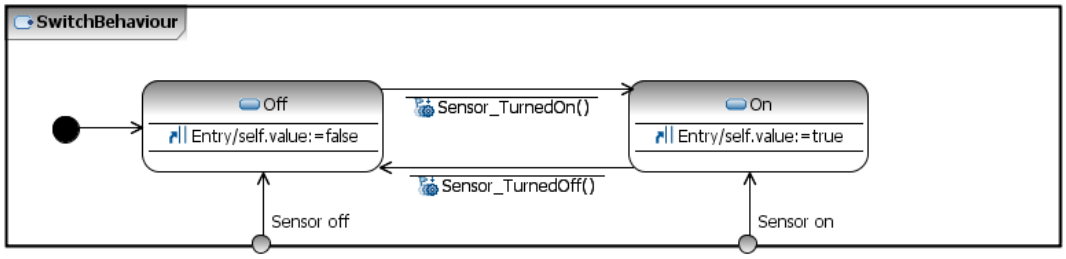

FIGURE 19. STATE MACHINE FOR THE SWITCH SENSOR.

First, `FeedBelt` has one sensor sub-block (`photoelectric_cell`) and one actuator sub-block (`conveying_motor`). The feed belt communicates with its sensor and actuator hardware devices through these two sub-blocks. There is one sub-state machine for each of these sub-blocks in Figure 18. We only show a reference to these state machines here because they have already been fully specified for the `Switch` block and the `Unidirectional_Motor` block (e.g., see Figure 19 for the Switch state machine). Second, `FeedBelt` communicates with its environment, the operator and the `Table` block, in an asynchronous way. To properly model this mode of communication, we assume that each event coming from the environment is kept in a buffer of size one, and will be sent to `FeedBelt` only when it is ready to process it. The state machine in Figure 18 has two buffer state machines: one for the `Add_Blank`() signal coming from the operator, and one for the `Feed_Table`() signal coming from the table. Each buffer state machine has two states: one for buffer empty and one for buffer full. By receiving the signal from the environment the buffer moves from its empty state to its full state (see for example the transition with trigger `Add_Blank`() of the `AddBlank` buffer state machine). It remains in its full state until a change event indicating that the system is ready to process the signal becomes true (e.g., the `FeedBelt` is ready to process Add_Blank() when the guard "`not photoelectric_cell.value and not conveying_motor_on`" holds). Then the buffer

sends the signal to the main system state machine (e.g., using the effect `self.add_blank()` of the `AddBlank` buffer), and moves back to its empty state. Finally, there is one state machine that describes the behaviour of the `FeedBelt` block (`FeedBelt main` state machine in Figure 18). This state machine receives messages from the buffer or the sensor state machines (e.g., `photoelectric_cell`) and sends messages to the actuator state machines (e.g., `conveying_motor`) or to its environment, i.e., other top-level blocks. For example the environment for `FeedBelt` is the `Table` block, and for `Table` is the `FeedBelt` and `Robot` blocks.

An abstract view of the typical state machine structure of PCS top-level blocks is shown in Figure 20. As shown in the figure, each state machine diagram is composed of four kinds of sub-state machines: buffers, top-level part, actuators, and sensors. In Figure 18, the buffers are `AddBlank` and `FeedTable` bufers, the top-level part is `FeedBelt main`, the actuator is `conveying_motor`, and the sensor is `photoelectric_cell`. The direction of communication between these sub-state machines, the environment, and the hardware devices are shown using arrows. Note that the communication between buffers and top-level part, and between top-level part and actuator/sensors is synchronous and is represented as procedure call in Figure 18. However, the communication between each top-level part and its environment, i.e., other top-level parts, is synchronous.

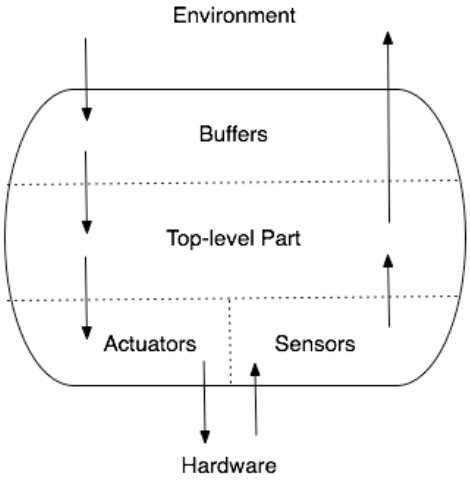

FIGURE 20. A GENERIC VIEW OF STATE MACHINE DIAGRAMS FOR TOP-LEVEL BLOCKS.

It is often useful to explicitly specify the assertions that hold at the state machine states. These assertions are referred to as *state invariants*. State invariants are in fact necessary to help ensure states are clearly and precisely defined. For example, in Figure 18, the state invariants can specify that the feed belt is not running in states "`Bell Empty`" and "`Plate At End`", but is running at states "`wait for plate to reach end`" and "`wait for plat to leave table`". Specifically, the state invariant for the former states

is `self.running`, and for the latter states `not self.running`, where `running` is an attribute of the block `FeedBelt`.

Consistency.

- Signals received by a part in activity diagrams should trigger exactly one transition of the state machine of that part.
- Signals sent by a part in activity diagrams should appear as the effect of at least one transition in the state machine of that part.

Step (2.i).

Input: activity diagrams + system structure diagrams.

Output: Pre- and post-conditions for the operations of top-level blocks.

General. The last step in creating behavioural models in phase 2 is to specify the pre- and post-conditions for the block operations. Pre-condition is an assertion attached to the operation that must be guaranteed prior to any call to that operation, and post-condition is an assertion that must be guaranteed to hold by the operation body on return from any call to the operation in which the precondition was satisfied on entry. In our methodology, we recommend writing pre- and post-conditions for the top-level blocks because these pre- and post-conditions will be used in the safety analysis steps when we want to connect the safety requirements to design

Case Study. It is often straightforward to write pre-conditions for the operations. For example, the pre-condition for the `add_blank` operation of the `FeedBelt` block is that the `FeedBelt` should not be running and there should not be any blank at the end of the `FeedBelt`. The following OCL expression formalizes the pre-condition for the `add_blank`() operation:

```
not self.photoelectric_cell.value and not self.conveying_motor.on
```

To write post-conditions, we first recall that the sequences of steps in each operation of the top-level blocks are already specified using activity diagrams (step (2.g)). For example, the activity diagram in Figure 16 shows the sequence of steps in the `add_blank`() operation. A post-condition for `add_blank`() is, then, a statement that holds after executing the steps in Figure 16. Note that there is no need to write complete post-conditions as it is often very difficult and time-consuming to write such post conditions. Our recommendation is that post-conditions are only need to describe what signals have been received from the sensors and what system attributes have been changed by these signals. Such post-conditions give sufficient information about the position of the mechanical devices and of the blank in the production cell after execution of each operation. For example, as shown in Figure 16, after executing `add_blank`(), a signal

from its `photoelectric_cell` is received indicating that there is a blank at the end of the feed belt. The OCL expression formalizing this post-condition is:

```
self.photoelectric_cell.value
```

Where `value` can true or false depending on if the `photoelectric` switch is on or off, respectively. We will use the pre- and post-conditions in phase 3 to create proper traceability links between requirements and design diagrams.

3.3. Safety-Related Steps

The last two steps of our methodology, steps (2.j) and (2.k) concern the generation of safety arguments (cases) from the diagrams created in the earlier steps. These steps are usually carried out by software and safety engineers when a first version of the design has been devised through steps (2.a) to (2.i) based on core functional requirements of the system. Sometimes, however, engineers prefer to interleave the activities in steps (2.j) and (2.k) with the earlier design steps. This would enable them to take into account the system safety concerns early in the design process, and hence generate design diagrams that capture safety requirements more accurately. In any case, these activities are generally expensive. Therefore, we expect the engineers to undertake them only for selected safety-critical requirements.

To create a safety argument, the safety engineer needs to first identify the relationships between safety requirements and design diagrams. Such relationships can be better identified when the requirements are described in a structured form. The safety requirements described in steps (1.b) and (2.c) are already described in a precise textual form, but they can be made more structured if we rewrite them in a logical or behavioural language. Therefore, we expect the software engineer to first formulate the requirements in terms of logical or behavioural constraints. The safety engineer is then in a position to use these constraints to identify relationships between requirements and design diagrams, and then, extract slices, i.e., sub-graphs, of design diagrams relevant to each requirement. Specifically, in step (2.j) of our methodology, the software engineer translates the requirements into design constraints containing block operations and block attributes. In this report, we consider two different kinds of notations for expressing design constraints: a logical notation based on the Object Constraint Language (OCL), and a behavioural notation based on activity diagrams. Then, in step (2.k), the safety engineer uses the resulting formalized requirements to identify slices of the design diagrams relevant to the given safety-related requirements. In future, we will show how these slices can then be analyzed to determine whether a requirement has been properly addressed by the design, and further how slices and the traceability links between requirements and slices can be utilized for generation of safety cases.

Step (2.j).

Input: block-level requirements diagrams + traceability links from block-level requirements to top-level system blocks.

Output: logical constraints (OCL) formalizing the block-level requirements in terms of the block operations or predicates involving block attributes + behavioural constraints (activity diagrams) formalizing the block-level requirements in terms of sequential constraints on block operations.

In step (2.c), we identified block-level requirements and made links between these requirements and top-level blocks explicit. Such links are rather too coarse-grained to be solely relied on for extracting model slices. This is because requirements often do not concern entire blocks, rather they refer to particular operations or attributes of the blocks. In our methodology, we make the traceability links to blocks more specific by augmenting them with mappings from the requirement terms to block operations or to (boolean) predicates over block attributes. By boolean predicates, we mean logical expressions containing boolean operators and operands. For example, the following expression is a boolean predicate or predicate for short:

```
self.table.bottom.value and self.table.rotation.value = 0
```

where `self.table.bottom.value` is boolean and `self.table.rotation.value` is integer.

For example, the requirement in Figure 11 refers to the `FeedBelt` operation for "conveying the plate from feed belt to table" and to the "loading position" of table. The term "conveying the plate from feed belt to table" matches the `feed_table`() operation of `FeedBelt`, as this operation is responsible for passing the plate to the table. The term "table being in loading position" corresponds to a situation where the `pos` attribute of table is set to `loadposition`: `table.pos = loadposition`. Alternatively, we can be more precise and map "table being in its load position" to the table's sensors' attributes. Specifically, the table is in load position when its `top` switch is "on" (`table.top.value`) and its `rotation` potentiometer is positioned at the `0` angle (`table.rotation.value = 0`). We attach SysML comment boxes to the requirement to represent the mappings from the requirements terms to the block operations and to predicates over block attributes. Each mapping attached to a requirement is a textual sentence following this template:

"phrase from requirements" iff "OCL expression"

where the context of the OCL expression is the block related to the requirement, and the navigation expression is a block operation or is a predicate over the block attributes. In functional safety requirements, one typical case is when the phrase is a statement about the state of the system and the OCL expression formalizes that statement as a predicate over block attributes. A second typical case is when the phrase refers to an action and the OCL expression models a sequence of operation calls.

For example, the comment boxes in Figure 21 attached to the requirement represent the mappings we discussed above. Note that in this case study, we focused only on functional safety requirements and for such requirements, it is sufficient to have

mappings from the requirement terms to block operations, and to predicates over block attributes. We may require different forms of mappings for other categories of requirements and this will be investigated in the future.

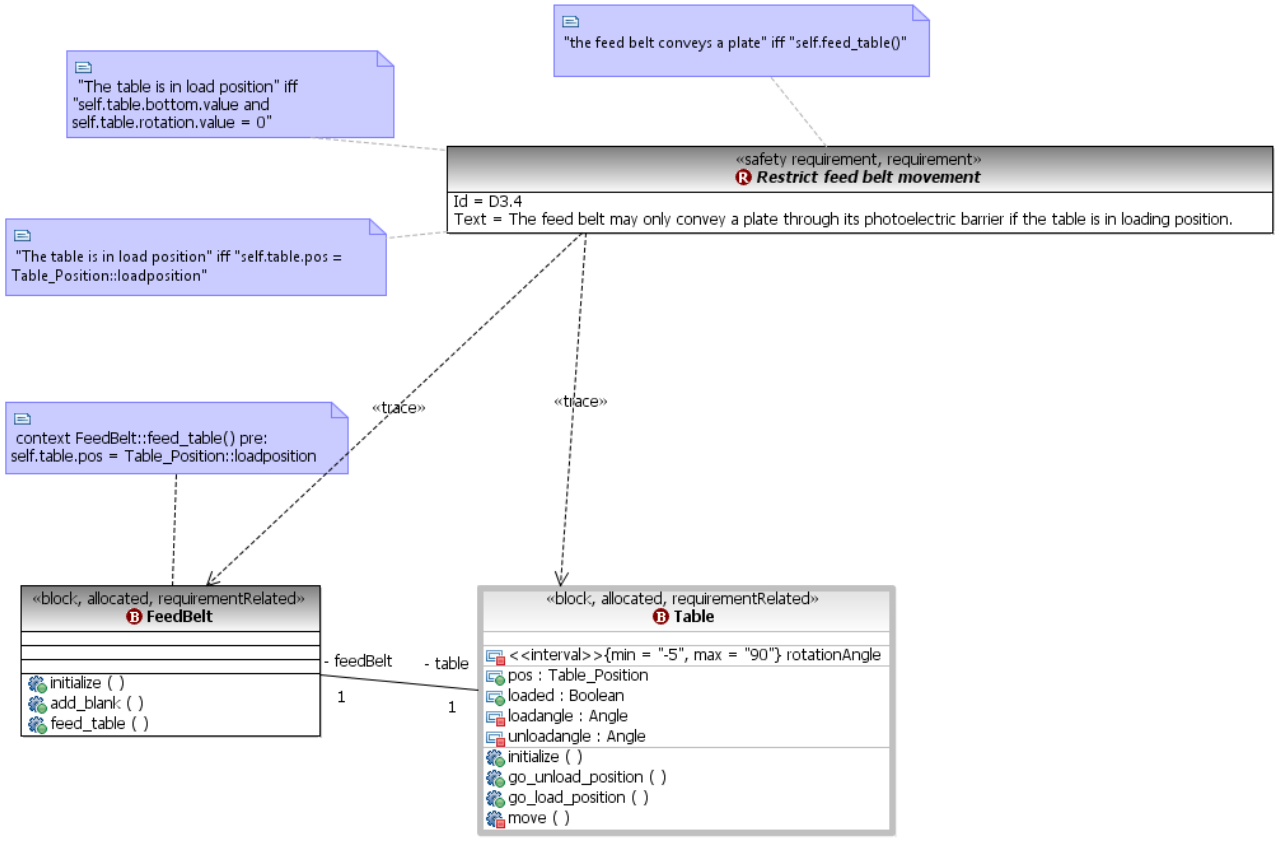

FIGURE 21. THE DIAGRAM IN FIGURE 11 AUGMENTED WITH (1) MAPPINGS BETWEEN THE REQUIREMENTTERMS AND THE DESIGN ELEMENTS, AND (2) AN OCL CONSTRAINT FORMALIZING THE REQUIREMENT IN TERMS OF THE BLOCK OPERATIONS AND PREDICATES OVER BLOCK ATTRIBUTES.

Having specified the mappings between the requirements and block elements, we now transform the requirements into design constraints. Functional safety requirements are typically formalized using either logical languages such as OCL, or using behavioral notations such as activity/sequence/state machine diagrams. Below, we show the formalization of the requirement in Figure 21 in OCL and activity diagrams.

The OCL constraint formalizing the requirement in Figure 21 is a pre-condition constraint stating that the assertion, `table.pos = loadposition`, should hold prior to the invocation of the `feed_table()` operation. The box attached to the `FeedBelt` block in Figure 21 represents this OCL constraint. Alternatively, we could be more precise and use the assertion, `table.top.value and table.rotation.value = 0`, as the pre-

condition for `feed_table()` in the OCL constraint. Both of these OCL constraints are valid.

(1) Functional safety requirements often constrain the system behaviour by describing the system *unsafe scenarios*, i.e., sequences of operations that the system must not exhibit to operate safely. For instance the unsafe scenario that the requirement in Figure 21 describes is that "the table is not in loading position and the feed belt transfers the plate to the table". We rewrite this requirement as sequence(s) of system operations that should be prevented to occur. To do so, we identify the system operations that cause table to move to a position different from "load position" by checking the operations' post-conditions. In our example, `go_unload_position()` moves the table to `unload_position`, while `go_load_position()` and `initialize()` move the table to `load_position`. Note that we convert the requirements to sequences of *public* system operations, and hence, we do not consider `move()` which is a private operation of table.
Figure 22 shows the sequences of system operations that are unsafe according to the requirement in Figure 21. Informally, the activity diagram in the figure specifies the unsafe sequences of the systems which are those in which `go_unload_position()` occurs before `feed_table()` without any occurrences of `go_load_position()` or `initialize()` in between. Note that we use the notation "not {act1, act2, ..., actn}" to indicate occurrence of any activity that is not in the set {act1, act2, ..., actn}.

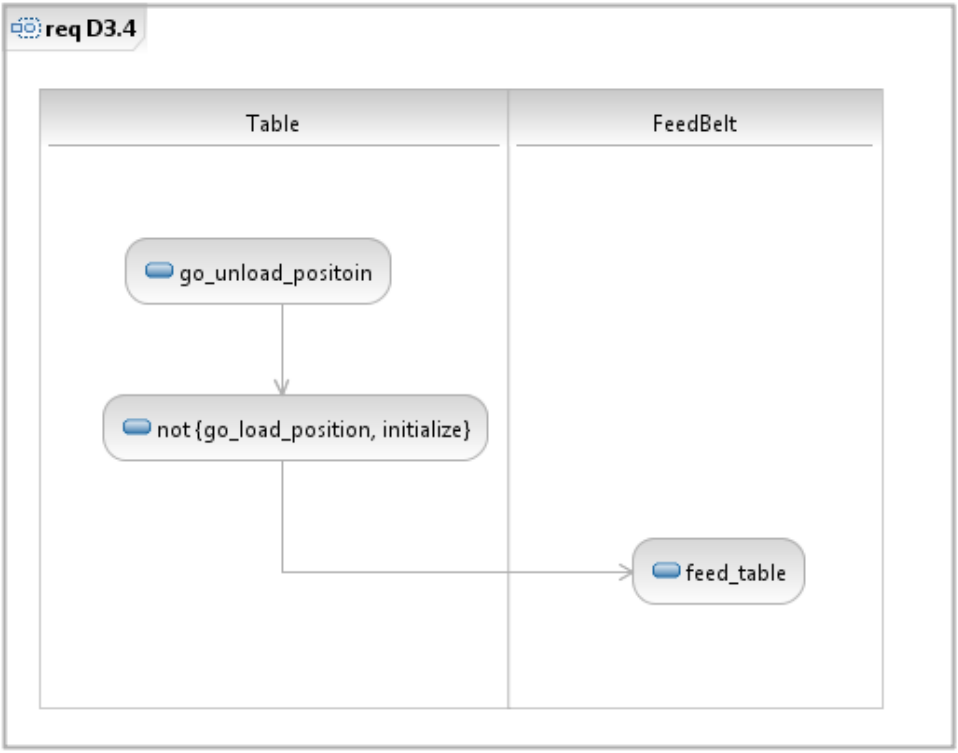

FIGURE 22. UNSAFE SCENARIOS OF PCS ACCORDING TO THE REQUIREMENT IN FIGURE 22.

In the next step, we show how the logical and behavioural representations of requirements can be used by safety engineers for identification of design diagram slices.

Step (2.k).

Input: Design constraints described as OCL constraints or behavioural diagrams describing safety-related requirements.

Output: Slices of design diagrams supporting the safety-related requirements.

Given a design constraint describing a system requirement, our ultimate goal is to extract all slices of different design diagrams relevant to that requirement. Functional safety requirements constrain system behaviours, and hence, are related to behavioural diagrams, namely, sequence diagrams, activity diagrams, and state machine diagrams. In our work, we show how the slices of activity and state machine diagrams can be extracted for a safety requirement. The process for extracting sequence diagram slices is very similar to that of extracting activity diagram slices. We describe the process for identification of slices for the two alternative requirement formalizations (pre-conditions, and unsafe activity diagram scenarios):

1. OCL pre-condition constraints: Consider the two alternative OCL constraints for the requirement in Figure 21 repeated below:

   (I)
   ```
   context FeedBelt::feed_table pre:
       self.table.bottom.value and self.table.rotation.value = 0
   ```

   (II)
   ```
   context FeedBelt::feed_table pre:
       self.table.pos = Table_Position::loadposition
   ```

   An OCL pre-condition constraint consists of top-level block operations and predicates over block attributes. We first present the assumptions that we use to extract diagram slices:

   a. For every block operation related to a safety requirement, there is a transition with a trigger corresponding to that operation in the state machine diagram of that block. This is because safety requirements refer to block behaviours. Therefore, every block operation that is relevant to safety requirement should cause that block to change its state. For example, the operation `feed_table()` is related to the requirement in Figure 21 and it appears as a transition trigger in the state machine in Figure 18 (see the `FeedBelt main` state machine).
   b. For every operation of a top-level block, there is an activity corresponding to that operation in at least one activity diagram partition related to that block. See the consistency rules for steps (2.e) and (2.g). For example, the operation `feed_table()` of block `FeedBelt` appears in the `FeedBelt` partition of the activity diagram in Figure 15.

c. Predicates related to a safety requirement appear in the state machine diagram of that block as state invariants, or transition guard conditions, or an attribute assignment in a transitions' effect. Note that such predicates constrain the behaviour of a block. Hence, they should appear as constraining guard conditions or state invariants, or as variable assignments in the state machines related to the block. For example, as will see later in this step, the predicate, `self.table.pos = Table_Position::loadposition`, appears in the state machine of `Table` as a variable assignment in a transition effect.
d. For every predicate defined over attributes of sensor blocks, there is a corresponding signal in at least one activity diagram partition related to that sensor block. See the discussion on signals of the sensor blocks in steps (2.e) and (2.g). For example, the predicate `self.table.bottom.value` is related to the `Table_BottomSensorOn` signal in Figure 24.

To extract activity diagram slices related to an OCL pre-condition, it is more convenient to write the pre-condition part in terms of sensors attributes. This is possible when the pre-conditions are about the position of mechanical devices or about the position of blank on mechanical devices. Such pre-conditions can be easily described in terms of sensors attributes. The pre-conditions in our safety requirements are of this kind. This allows us to use the relationships between sensors attributes and signals (see assumption (d) above), and extract the activity diagram slices. For instance, we use the OCL pre-condition (II) described above to extract activity diagram slices through the steps described below:

1. Go through all top-level activity diagrams, and extract those partitions of `FeedBelt` and `Table` in which the `feed_table()` operation appears. By assumption (b), such partitions exist. The reason that we extract `Table` partition in addition to `FeedBelt` partition is that the pre-condition is defined over table's attributes. Figure 23 represents the partition in which the `feed_table()` operation of `FeedBelt` appears from together with the partition related to table from the same diagram. Note that in our case study there is only one activity diagram in which `feed_table()` appears, but of course this is not always the case.
2. Go through all activity diagrams of the table sensors and extract the partition in which the signals related to the pre-condition `self.table.bottom.value and self.table.rotation.value = 0` appears. Figure 24 shows an activity diagram in which the `Table_BottomSensorOn` and `RotationPosOK` signals appear. The `Table_BottomSensorOn` signal makes the predicate "`self.table.rotation.value = 0`" true.
3. Show that the slice related to the pre-condition (in Figure 24) occurs before the `feed_table()` operation (in Figure 23). The diagram in Figure 24 describes the sub-activities of the `go_load_position()` operation in Figure 23. This operation is immediately followed by the `Feed_Table` signal which is a trigger for the `feed_table()` operation of `FeedBelt`.

Hence the slices in Figure 23 and Figure 24 provide a complete argument for the requirement in Figure 21.

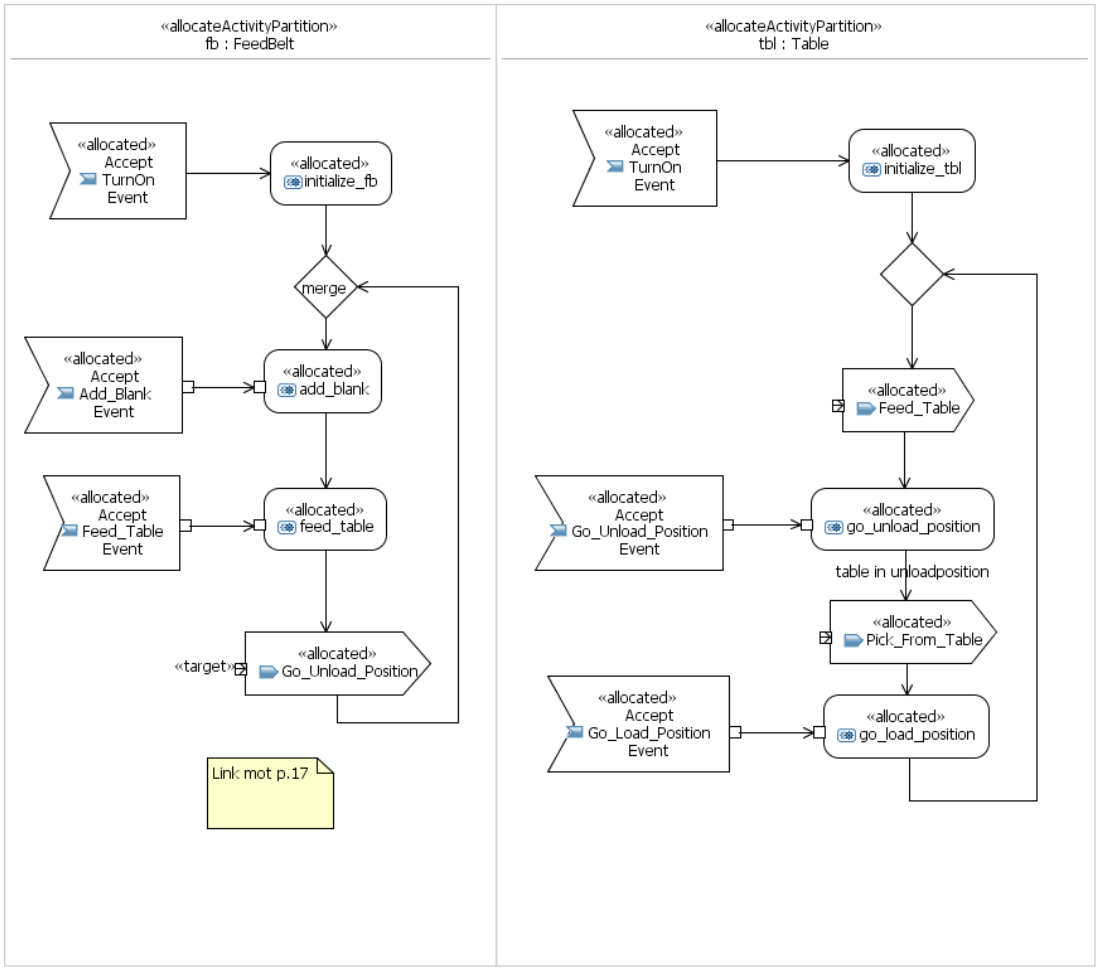

FIGURE 23. THE FEEDBELT AND TABLE PARTITIONS RELATED TO THE REQUIREMENT IN FIGURE 21.

Having identified the activity diagram slices, we now discuss how state machine diagram slices relevant to a pre-condition constraint can be identified. For the state machine diagram, we choose the alternative formulation of the pre-condition in terms of top-level block attributes, i.e., the OCL constraint (II), and we follow the steps below:

1. Identify those transitions in the block state machine (`FeedBelt` state machine in our example) for which the block operation in the OCL constraint (`feed_table()` in our example) is a trigger. By assumption (b), such transition exists. Furthermore, identify any transition in the table state machine for which the block operation (`feed_table()`in our example) is an effect. Figure 25 illustrates the transition of the feet belt state machine with the `feed_table()` trigger and the transition of the table

state machine with the `feed_table()` effect (the transition labels are highlighted in red boxes in the figure).

2. Show that the precondition of the block operation (`feed_table()` in our example) holds either at the source state (based on its state invariant) of the transitions with `feed_table()` trigger, or it is a guard condition on the transitions with `feed_table()` trigger, or the pre-condition holds in the target state of the transitions with `feed_table()` effect. As shown in the figure, the entry action of the `LoadPosition` state corresponds to the pre-condition of our OCL constraint, i.e., `self.table.pos = Table_Position::loadposition`. This implies that this precondition holds in the target state of the transition from the table state machine, which in turn implies that the pre-condition holds prior to the `feed_table()` operation.

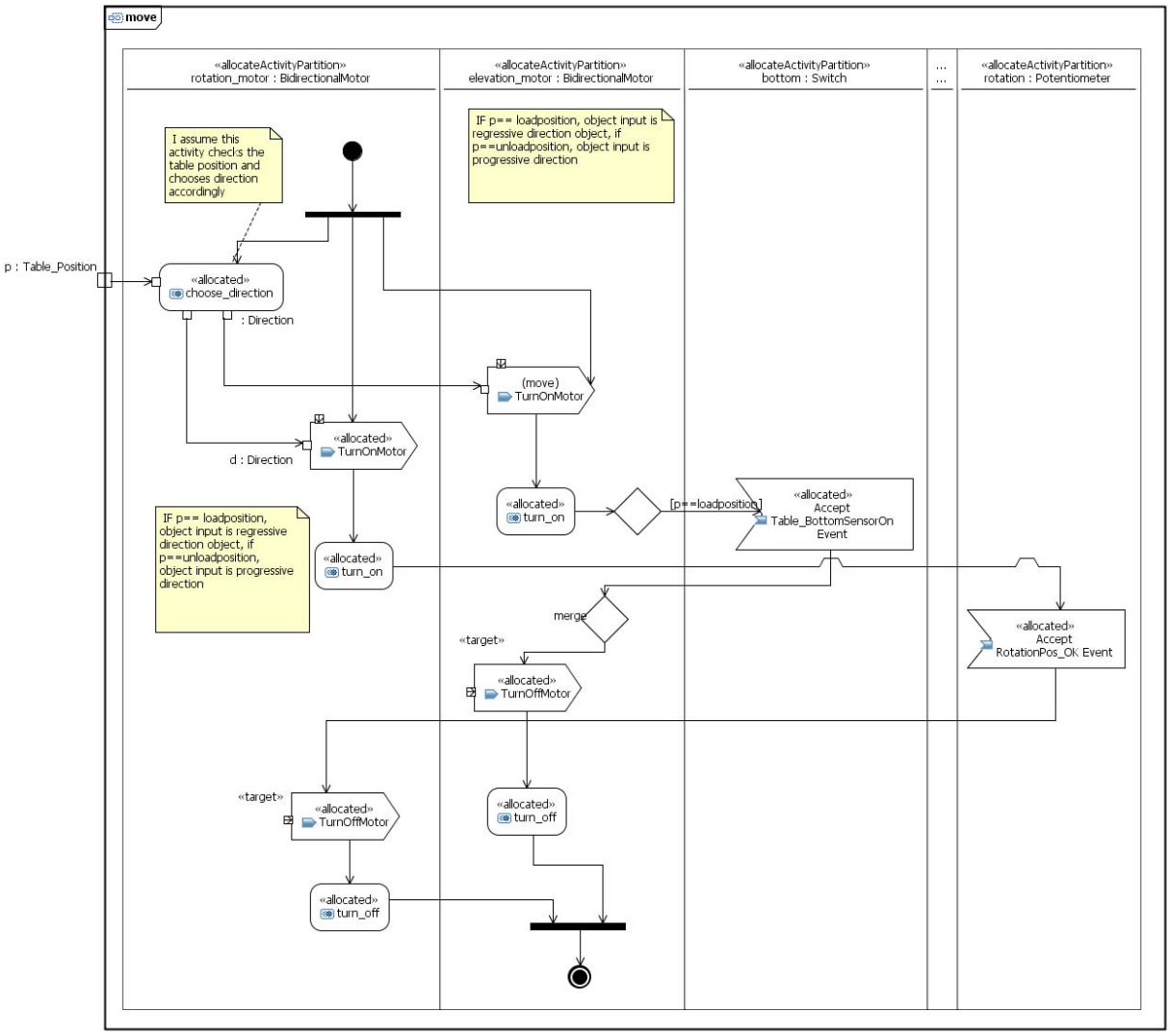

FIGURE 24. THE ACTIVITY DIAGRAM SLICE IN RELATED TO THE PRE-CONDITION OF THE REQUIREMENT IN FIGURE 21.

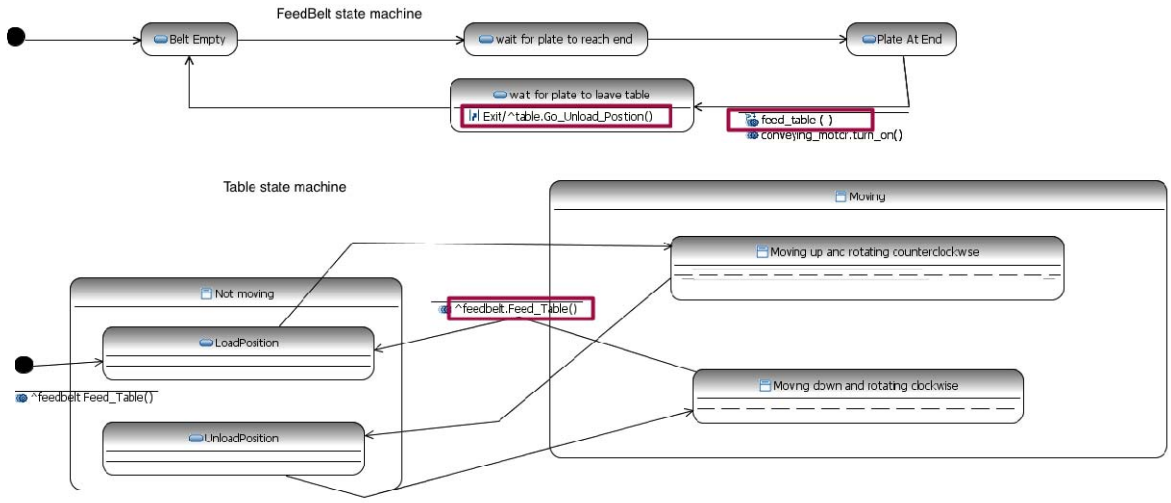

FIGURE 25. STATE MACHINE SLICES RELEVANT TO THE PROPERTY IN FIGURE 21.

2. Unsafe scenarios represented as activity diagrams: Given an unsafe scenario, we explain how activity and state machine diagram slices related to this scenario can be identified. Consider the scenario in Figure 22 stating that the system is unsafe if in between `go_unload_position()` of `Table` and `feed_table()` of `FeedBelt` no occurrences of `go_load_position()` or `initialize()` occur.

   Extraction of Activity diagram Slices: To extract activity diagram slices, we extract all activity partitions that contain any of the activity/actions referred to in the given unsafe scenario. In our example, it would mean that we go through all top-level activity diagrams and extract those partitions of `FeedBelt` in which operation `feed_table()` appears, and those partitions of `Table` in which `go_unload_position()` appears. These partitions for our example are shown in Figure 23.We then need to show that in these slices between any occurrences of `go_unload_position()` and `feed_table()` there has to be at least one occurrence of `go_load_position()` or `initialize()`. This can be easily checked on the diagram in Figure 23: The `feed_table()` can only be reached if signal `Feed_Table` is sent from the table partition. This signal is sent after the completion of `go_load_position()` or of `initialize()`. Thus, `feed_table()` can only occur after one of these two operations.

   Extraction of State machine diagram Slices: To extract state machine diagram slices, we hide all transitions that do not contain in their label the activities/actions in the given unsafe scenario. In our example, it would mean that we go through the state machine diagrams for `FeedBelt` and `Table`. We then hide every trigger and effect in these state machines except for the operations

that are used in the scenario in Figure 22: `feed_table()`, `go_unload_position()`, `go_load_position()`, and `initialize()`. The resulting slices are shown in shown in Figure 26. These two slices when put together never generate the unsafe scenario in Figure 22 because in the table state machine slice, the path from the transition labeled `go_unload_position()` always goes through a transition labeled "go_load_position()" before reaching the transition with effect `feed_table()`. Thus, it never happens that "feed_table()" follows `go_unload_position()` without a `go_load_position()` in between.

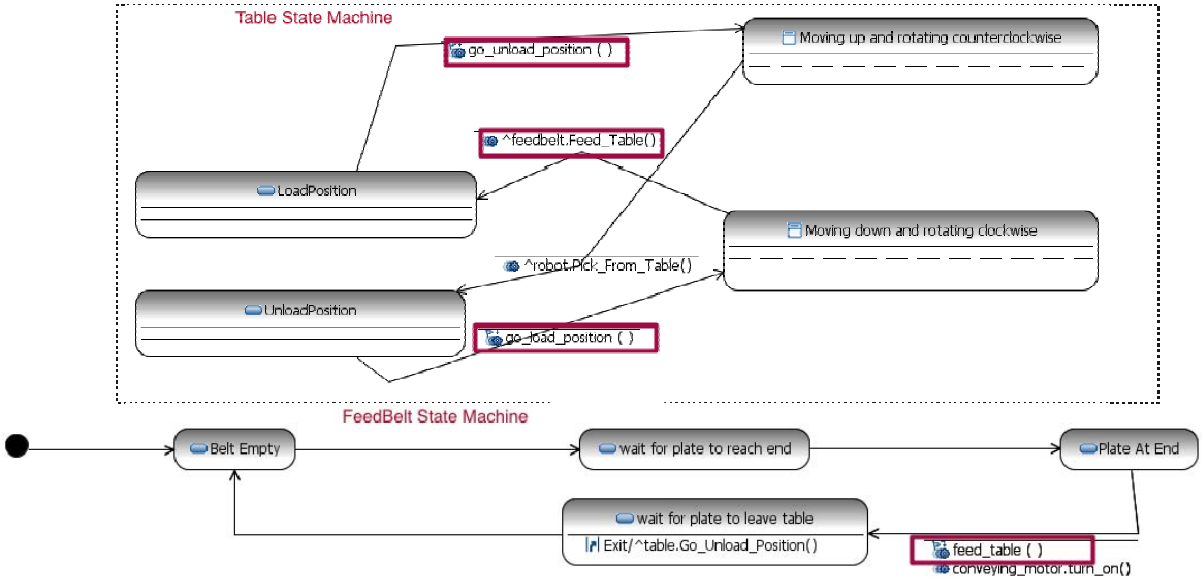

FIGURE 26. STATE MACHINE DIAGRAM SLICES RELATED TO THE SCENARIO IN FIGURE 24.

4. Discussion

In this section, we outline the most important open questions or issues discovered in this report. We will attempt to address them in subsequent steps of the ModelME! Project.

Establishing Mappings between Domain and Design Concepts. Requirements, both the system-level ones (Step (1.b)) and the block-level ones (Step (2.c)), are to be agreed by all the parties concerned with the system. To make this possible, these requirements are formulated in terms of problem domain concepts and the vocabulary used by such parties and not in terms of software or hardware design concepts. For example, the requirement that "The table must not be rotated clockwise when it is in the loadedposition" does not refer to any design concept such as the software table controller or the table actuators or sensors because otherwise it cannot be understood and agreed upon by the involved parties. To bridge the requirements and

design language gap, we need to provide a systematic way for expressing and representing the mappings between the requirements terms and design elements.

In step (2.j), we introduced some textual templates to represent mappings between functional safety requirements and design elements. This, however, needs to be extended in several ways (1) we need to devise appropriate templates for describing mappings for other kinds of requirements, (2) we need to provide assistance for retrieving/deriving the mappings, and (3) we need to extend SysML with proper language constructs for capturing the mappings.

Extending SysML traceability capabilities. Our case study shows that SysML traceability mechanism is rather primitive. Specifically, SysML provides the following categories of links: (1) from requirements to entire diagrams, (2) from requirements to diagram elements, e.g., blocks, actions, states, decision nodes, and (3) from requirements to requirements. These links are described as simple binary relations labeled with stereotypes. Our study shows the need for extending SysML traceability links in the following ways:

- More complex link structures: We often need to augment the links with some additional information that cannot be captured using stereotypes. For example, we need to explicitly describe mappings between requirements and design elements. Furthermore, we often need to describe situations where a collection of diagrams satisfies a single requirement, or some diagrams provide alternative ways for satisfaction of a requirement. To capture such cases, we need to be able to describe conjunction/disjunction operators between the links.
- Better link semantics: SysML allows links to be labeled by stereotypes such as “trace”, “satisfy”, “verify”, “drivedby”, and “refine”. The semantics and usage of these stereotypes are not properly described in the existing SysML standards and books. Moreover, the links between requirements are currently either labeled by “drivedby” or “refine”. But, requirements may interact in many other ways (Lamsweerde 2009): they may contradict one another, they might be inconsistent with each other, or a requirement may positively contribute to the satisfaction of another requirement. Such links between requirements are important as they help us make informed trade-offs between requirements, in particular between the nonfunctional ones.
- More kinds of links: As mentioned above, SysML traceability links are defined between requirements and diagrams or elements in the diagrams. In this report, we often needed to extract a slice of a diagram, i.e., a sub-graph of a diagram, and create links to those sub-graphs.

An outline of different types of links needed by our methodology is given in the appendix (Section 7.2).

Designing a Language for describing unsafe scenarios. Unsafe/undesirable scenarios have been long used in the literature for formalizing safety/security threats, anti-goals, exceptional system behaviours, etc. In this report, we noticed that such

scenarios can further be used in safety arguments, potentially resulting in more succinct/intuitive arguments. Moreover, use of such scenarios in a safety argument helps automate checking compliance between a requirement and its accompanying evidence. Neither SysML nor UML currently have explicit construct for capturing such scenarios. To be able to effectively use these scenario representations, we need to extend SysML behavioural diagrams so that they can capture negative behaviours.

Providing support for extracting and visualizing model slices. An important part of our methodology is identification of design diagram slices relevant for a functional safety requirement. To be able to generalize our work to non-functional, safety-related requirements we need to see if slicing can be defined in a systematic way for these types of requirements as well. Moreover, slicing is a manually intensive task and needs automated support. Finally, proper visualization for representing slices related to a requirement and proper support for navigation between the slices will greatly influence the success and adoption of our methodology in practice. We need to support, to the maximum extent, the inspection and analysis of slices.

5. Conclusion and Future Work

In this report, we developed detailed SysML diagrams for the Production Cell System. We drew on this experience and proposed methodological guidelines for creating SysML diagrams with a primary focus on providing support for the certification process. In particular, we described how system safety requirements can be traced to the design diagrams, and how safety engineers can utilize the traceability information to create safety arguments. We further identified some limitations of our methodology, the SysML modelling approach, and the SysML tool support, and discussed some potential solutions for addressing these limitations.

In future, we plan:

- to extend the application of our methodology to industrial systems (this was already started with K-Safe at Kongsberg) and refine the methodology to accommodate new, arising needs.
- to extend and refine our traceability mechanism to address categories of requirements other than functional safety.
- to adapt the methodology so that it can be used for generating safety cases that precisely conform to the conceptual model being developed in our previous work in (Lionel Briand, Thierry Coq et al. 2009).
- to provide automated scalable techniques for diagram slicing to be used for extraction of safety information from SysML design diagrams.
- to provide tool-supported techniques for visualizing safety case information and for checking compliance of this information with the safety requirements.

In addition, the above directions, we plan to improve our work by addressing the issues discussed in Section 4 as well as other detailed points discussed throughout this report.

References

Barbey, S., D. Buchs, et al. (1998). A Case Study for Testing Object-Oriented Software: A Production Cell.

Friednenthal, S., A. Moore, et al. (2008). A Practical Guide to SysML: The Systems Modeling Language, Morgan Kaufmann OMG Press.

Gomaa, H. (2000). Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley.

Hans-Erik Eriksson, M. P., Brian Lyons, David Fado (2003). UML 2 Toolkit (OMG), John Wiley & Sons.

Jackson, D., M. Thomas, et al. (2007). Software for Dependable Systems - Sufficient Evidence?, The National Academies Press.

Kleppe, A., A. Warmer, et al. (2003). MDA Explained - The Model Driven Architecture: Practice and promise, Addison-Wesley.

Klykken, T. (2009). A Case Study Using SysML for Safety-Critical Systems Department of Informatics. Oslo, University of Oslo. **Master thesis**.

Lamsweerde, A. v. (2009). Requirements Engineering, Wiley, John & Sons.

Lewerentz, C. and T. Lindner, Eds. (1995). Formal Development of Reactive System: Case Study Production Cell. Lecture Notes in Computer Science.

Lionel Briand, Thierry Coq, et al. (2009). Characterizing the Chain of Evidence for Software Safety Cases: A Conceptual Model Based on the IEC 61508 Standard. ModelMe! Technical Report 0901, http://modelme.simula.no/.

6. Appendix

6.1. Description of the Production Cell mechanical parts

The Production Cell control system communicates with each of the machine's sensors and actuators and this makes up the communication flow between hardware and software blocks of the Production Cell. The sensors (photoelectric cells, switches, magnets) either indicate the position of the plate on a machine or the rotary/vertical/horizontal position of the parts of each machine, and the actuators (motors and magnets) are activated to make the parts move or to carry a forged/not-forged plate. Below we describe each mechanical part in the production cell in more detail.

FeedBelt. 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.

Rotray 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.

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.

Robot. The robot consists of two orthogonal arms, a bidirectional motor, which take 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 activated/deactivated 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.

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.

Press. The press consists of one fixed horizontal plate and one movable horizontal plate which forge the metal plate when these are moved together. It also consists of and a bidirectional motor, which makes the movable horizontal plate move, and three switches, which indicating the movable plate's 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 to the press, it 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.

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.

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.

6.2. Characteristics of Traceability Links Required by Our Methodology

There are three main types of links that users need to create to apply our methodology for safety analysis:

- Requirement-to-Requirement links. These can describe:
  - if a functional or non-functional requirement positively/negatively contributes to a non-functional requirement.
  - if a functional requirement contradicts or is inconsistent with another functional requirement.
  - the break-down of a system-level functional requirement into a set of finer-grained block-level functional requirements.
    - See step (2.c) for examples of such traceability links.
- Requirement-to-slice links. These can describe:
  - if a requirement is traced to a slice of a single or multiple diagrams (potentially, a slice can be a whole diagram or set of diagrams).
    - For example in Figure 21, a requirement is traced to a couple of blocks.
  - if a slice satisfies a requirement.
    - For example, the slices in Figure 23 to Figure 26 satisfy the requirement in Figure 21, and we need to put traceability links between the requirement and those slices.
  - relationships between a requirement and a use case realizing that requirement.
    - See Figure 7 for an example.
- Applicable consistency rules:
  - The diagram satisfying a requirement should be included in the diagrams traced to that requirement.
    - For example, the requirement in Figure 21 is traced to two blocks (table and feed belt), and is satisfied by the slices in Figure 23 to Figure 26. These slices are all fragments of the state machine and activity diagrams belonging to those to two blocks.
  - We sometimes may want to say that a requirement is traced to (or is satisfied by) two or more alternative models or two more alternative sets of models.
    - For example, the requirement in Figure 21 is satisfied by the slice in Figure 23 and the one in Figure 26 independently. These two slices are two alternative representations (one is an activity

diagram, and the other one is a state-machine diagram) for satisfying that requirement.

- Requirement-to-constraint links. These describe if a requirement is derived from an environmental constraint. By a constraint, I mean an OCL constraint or a SysML parametric diagram.
    - For example, the parametric diagram in Figure 4 may introduce new system-level requirements, and in such case, there should be traceability links between this requirement and the constraint.

6.3. SysML Diagrams for the Production Cell System

Use case diagram

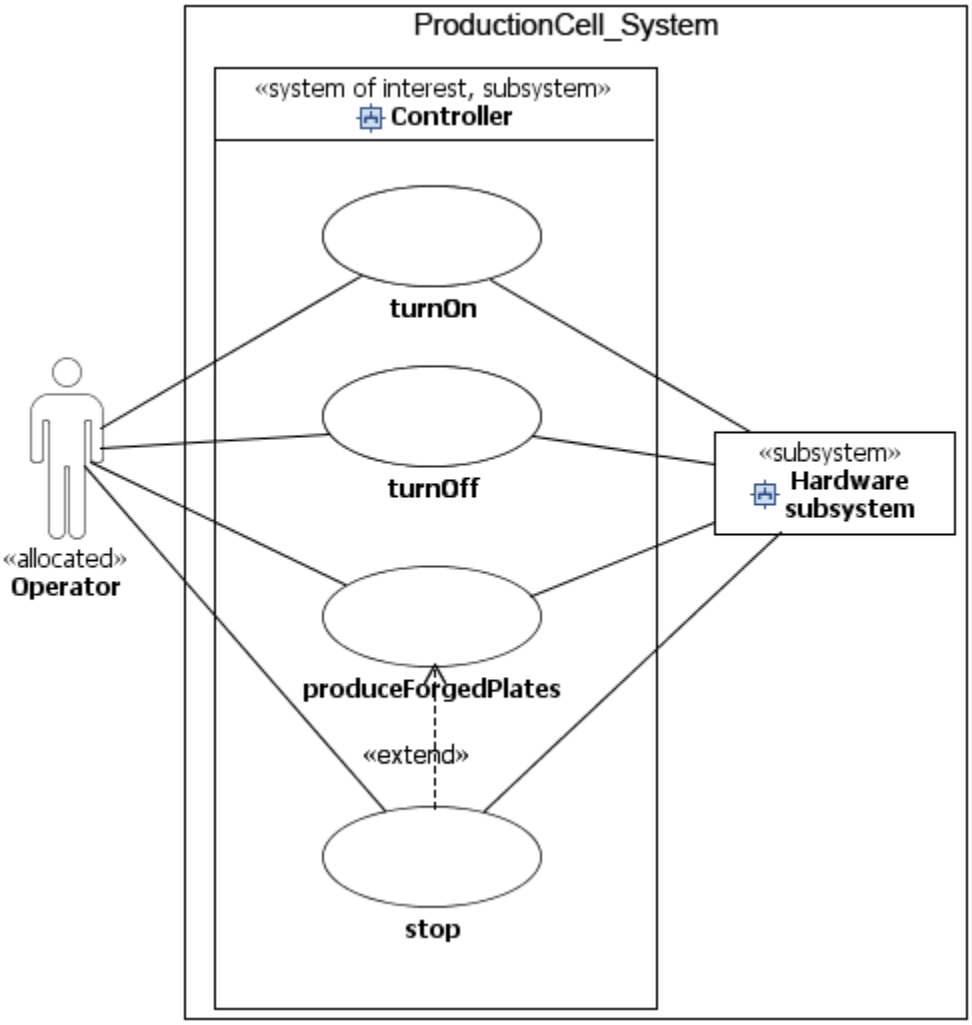

FIGURE 27 – USE CASE DIAGRAM FOR THE PRODUCTION CELL SYSTEM, DESCRIBING THE SYSTEM'S MAIN FUNCTIONS AND ACTORS.

Sequence diagrams

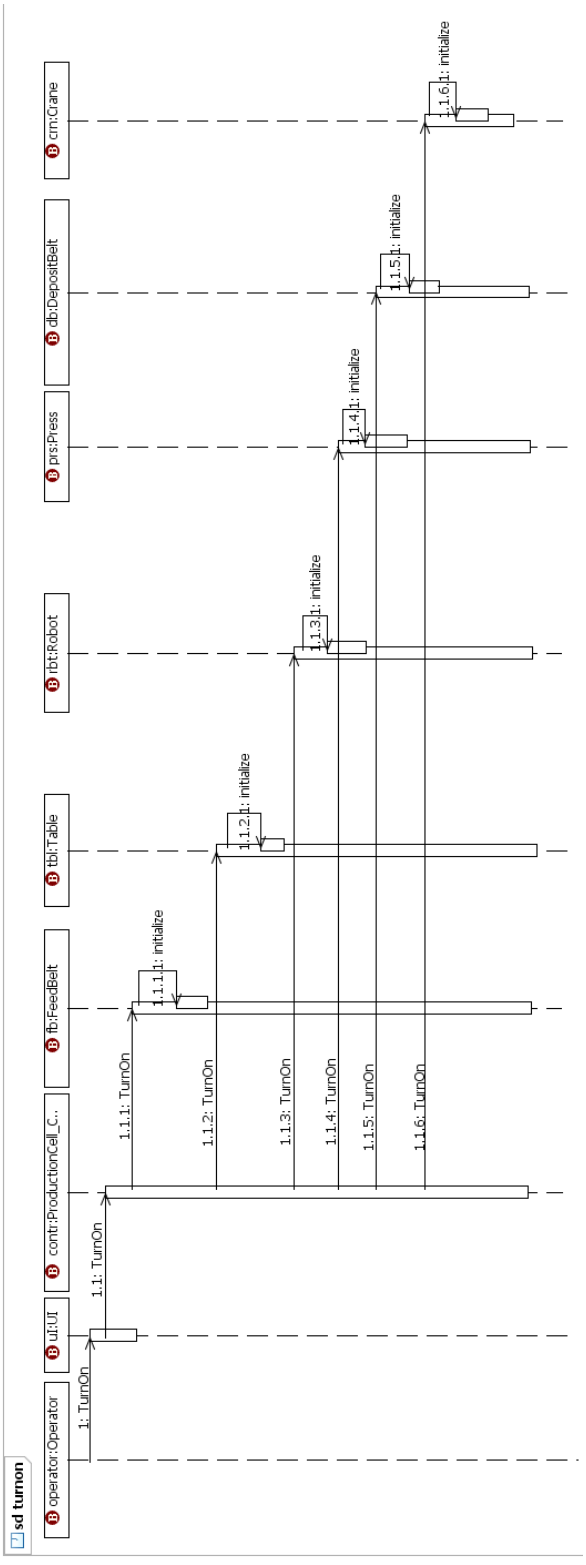

FIGURE 28 - SEQUENCE DIAGRAM FOR THE USE CASE TURN ON.

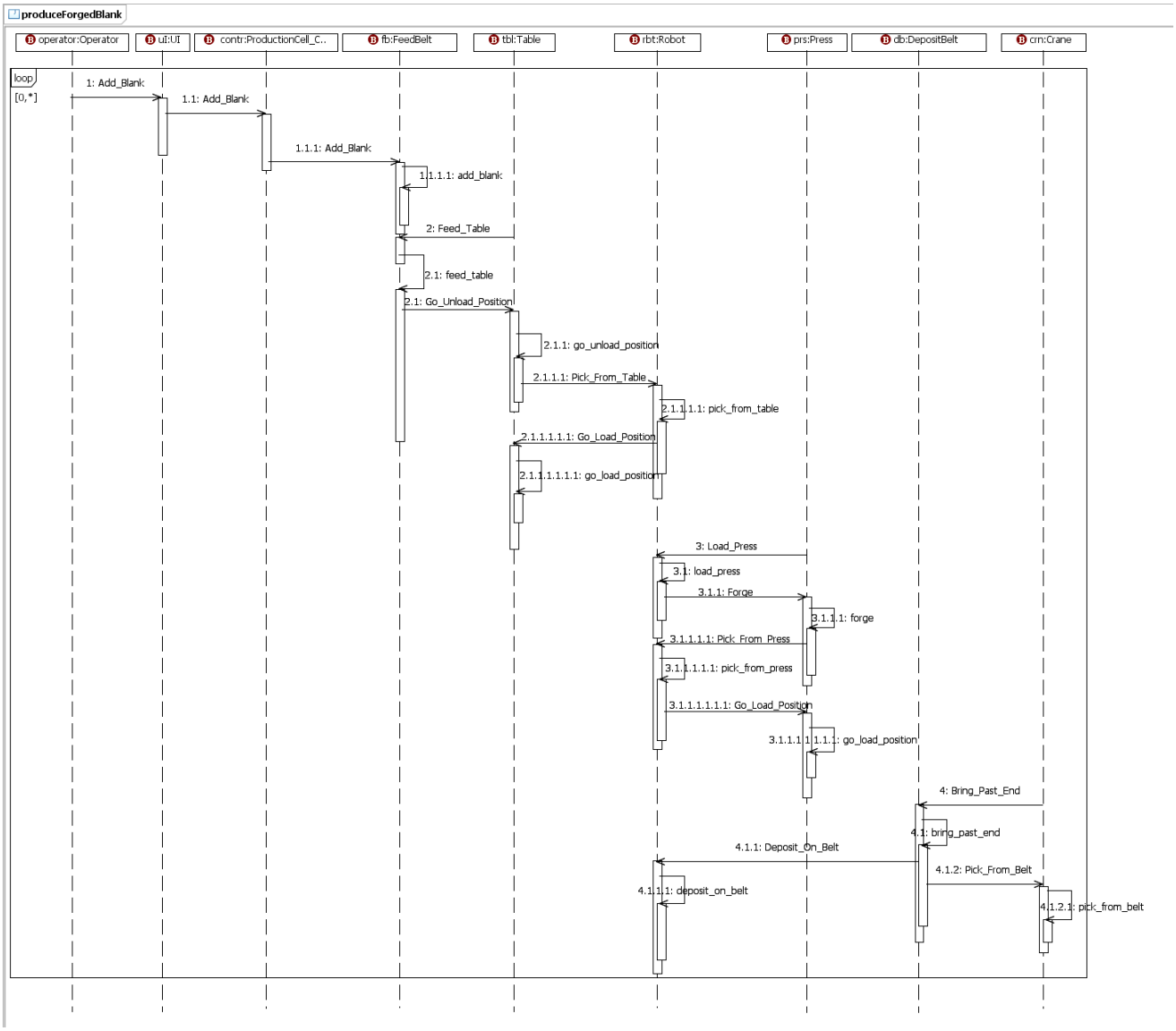

FIGURE 29 - SEQUENCE DIAGRAM FOR USE CASE PRODUCE FORGED BLANK.

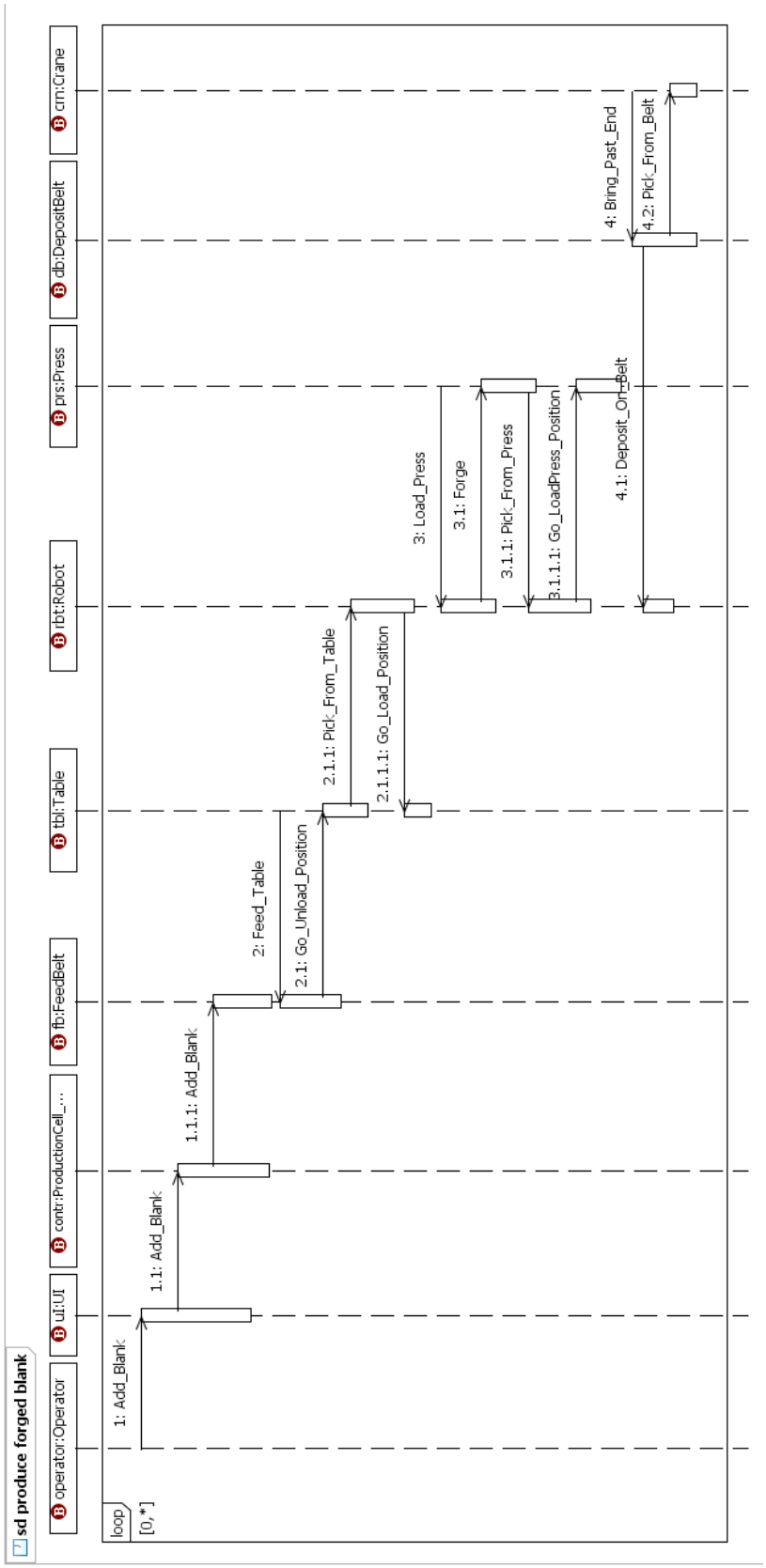

FIGURE 30 - SEQUENCE DIAGRAM (SIMPLIFIED) FOR USE CASE PRODUCE FORGED BLANK.

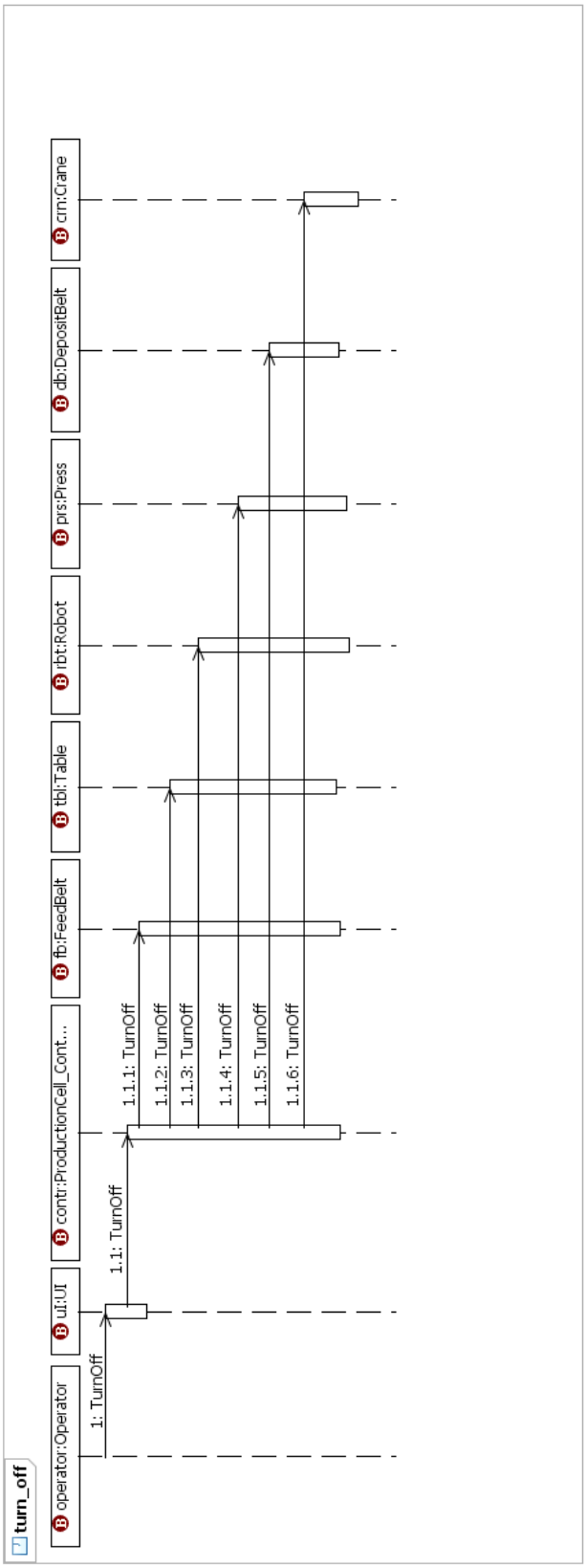

FIGURE 31 - SEQUENCE DIAGRAM FOR THE USE CASE TURN OFF.

Requirements

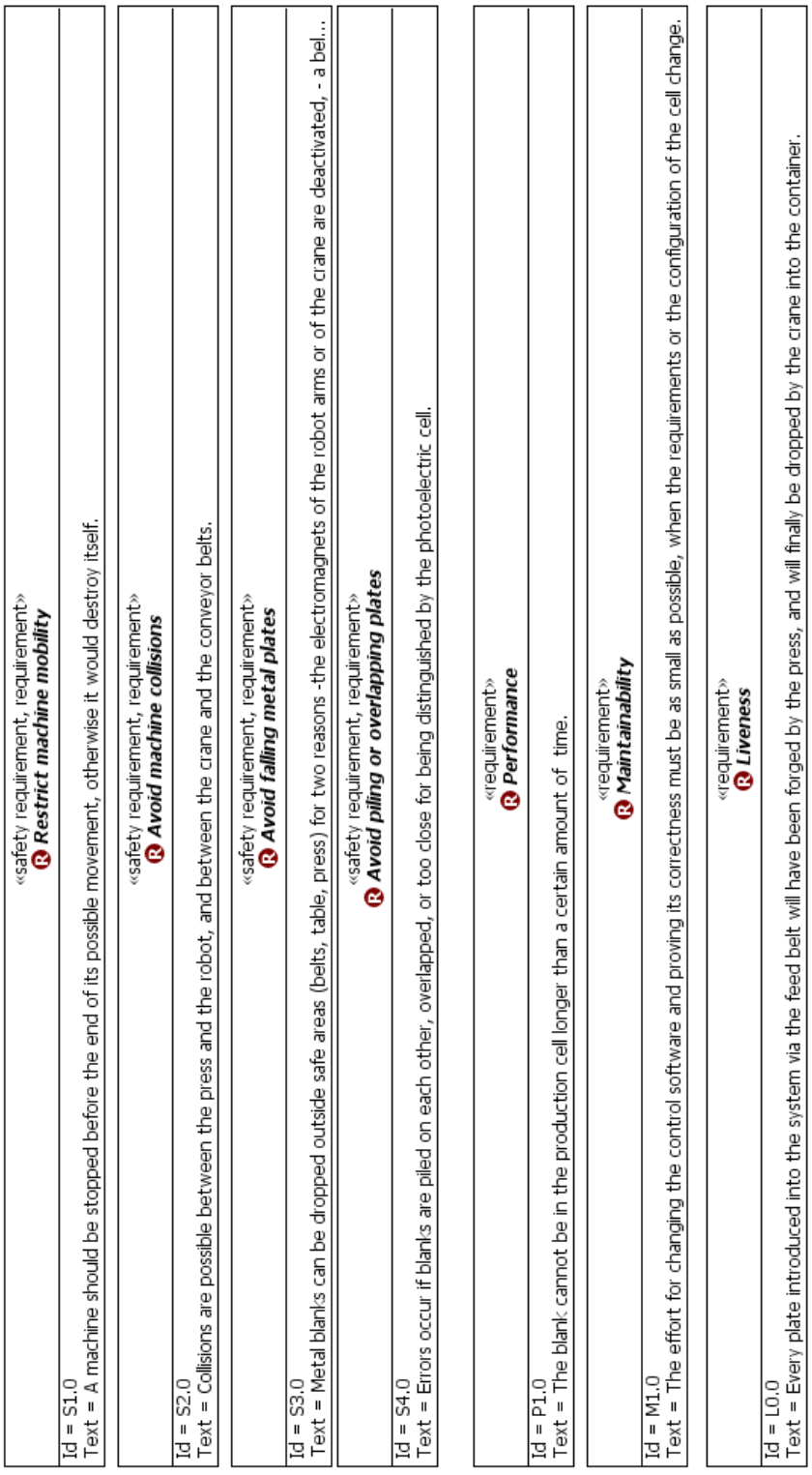

FIGURE 32- SYSTEM LEVEL REQUIREMENTS

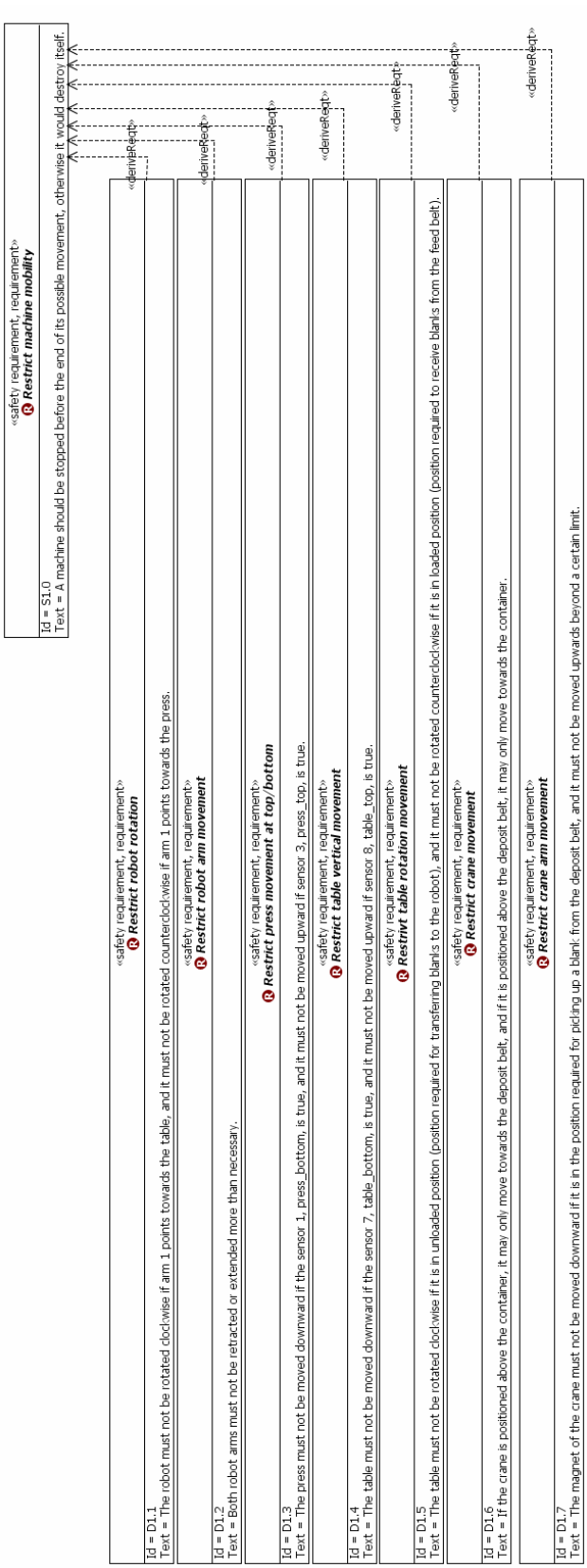

FIGURE 33 – DERIVED SAFETY REQUIREMENTS REGARDING RESTRICTING MACHINE MOBILITY.

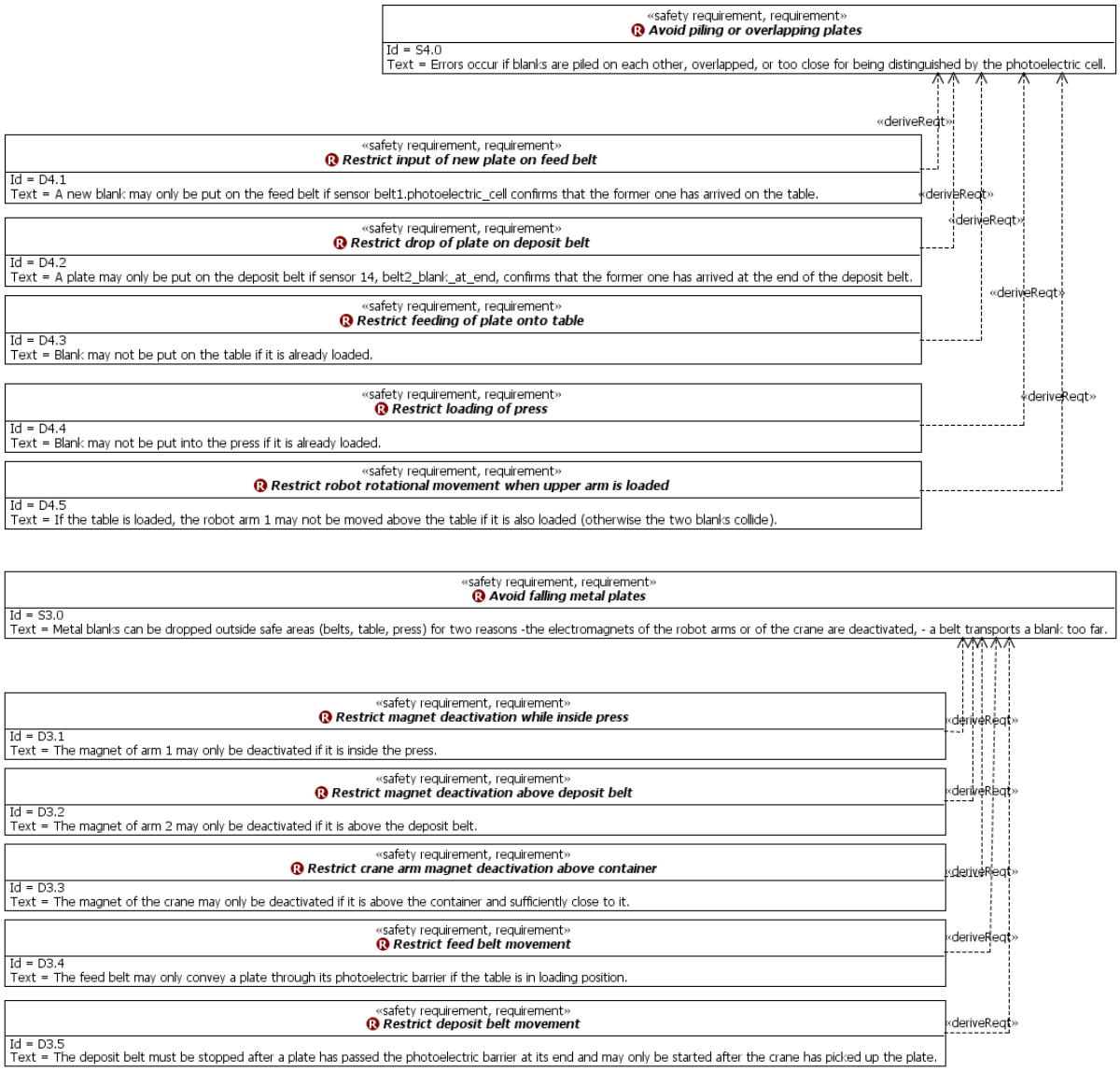

FIGURE 34 – DERIVED SAFETY REQUIREMENTS REGARDING AVOIDING FALLING OR PILING OF PLATES.

FIGURE 35 - OTHER REQUIREMENTS.

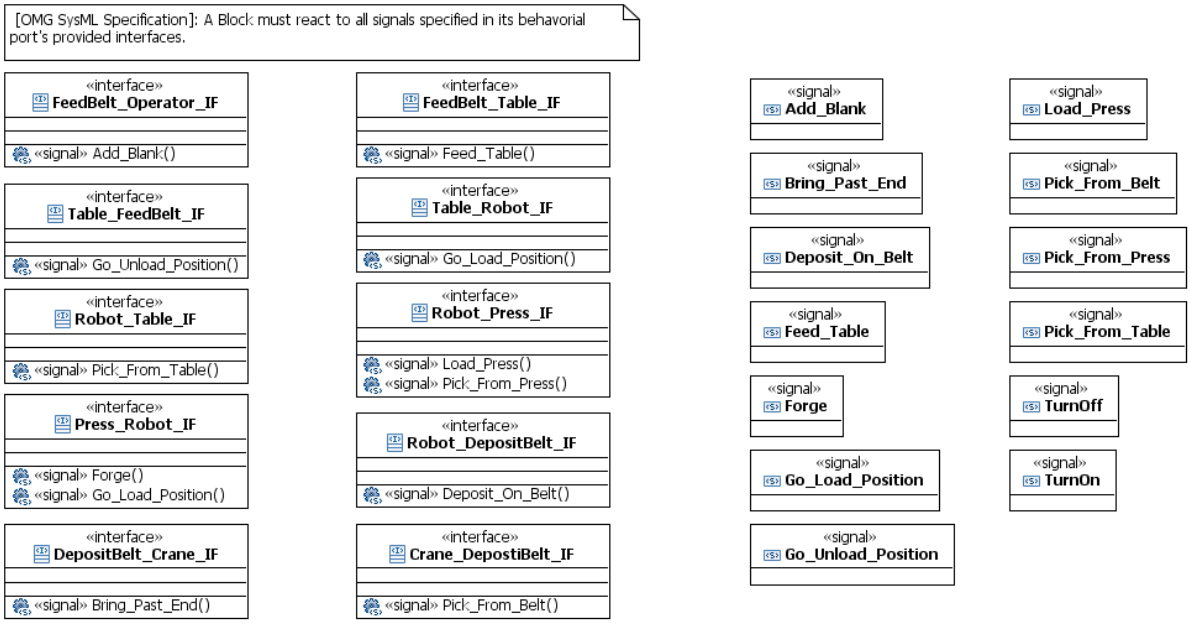

FIGURE 36 - OVERVIEW INTERFACES AND HIGH LEVEL SIGNALS.

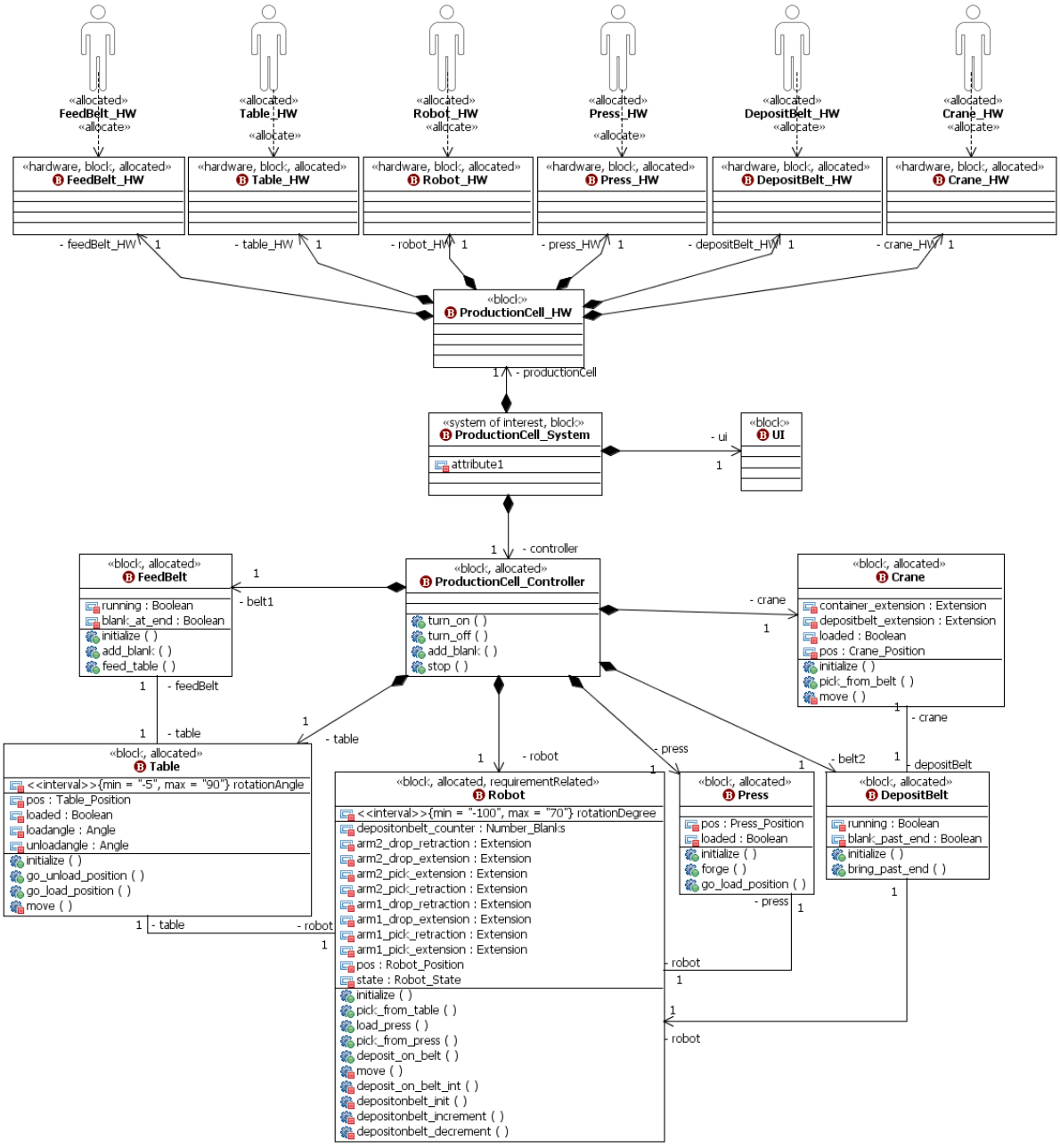

FIGURE 37 - SYSTEM DIAGRAM

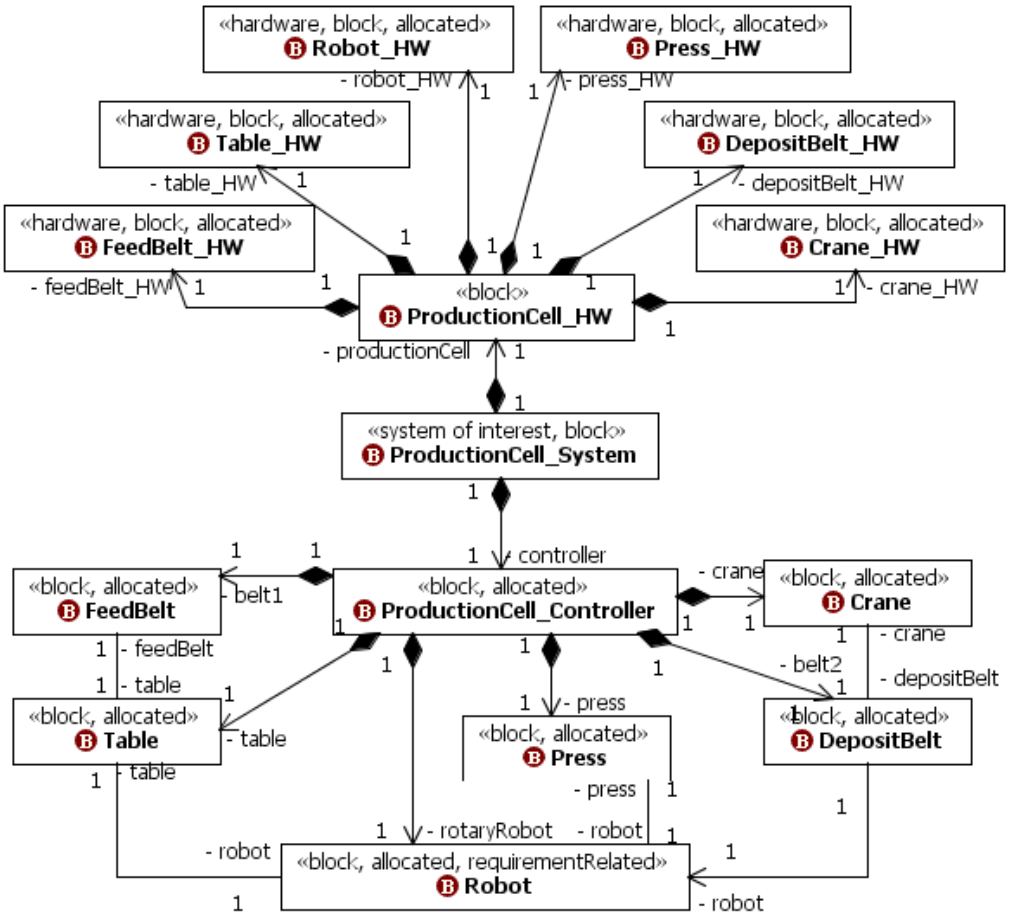

FIGURE 38 - CONTEXT DIAGRAM (SIMPLIFIED)

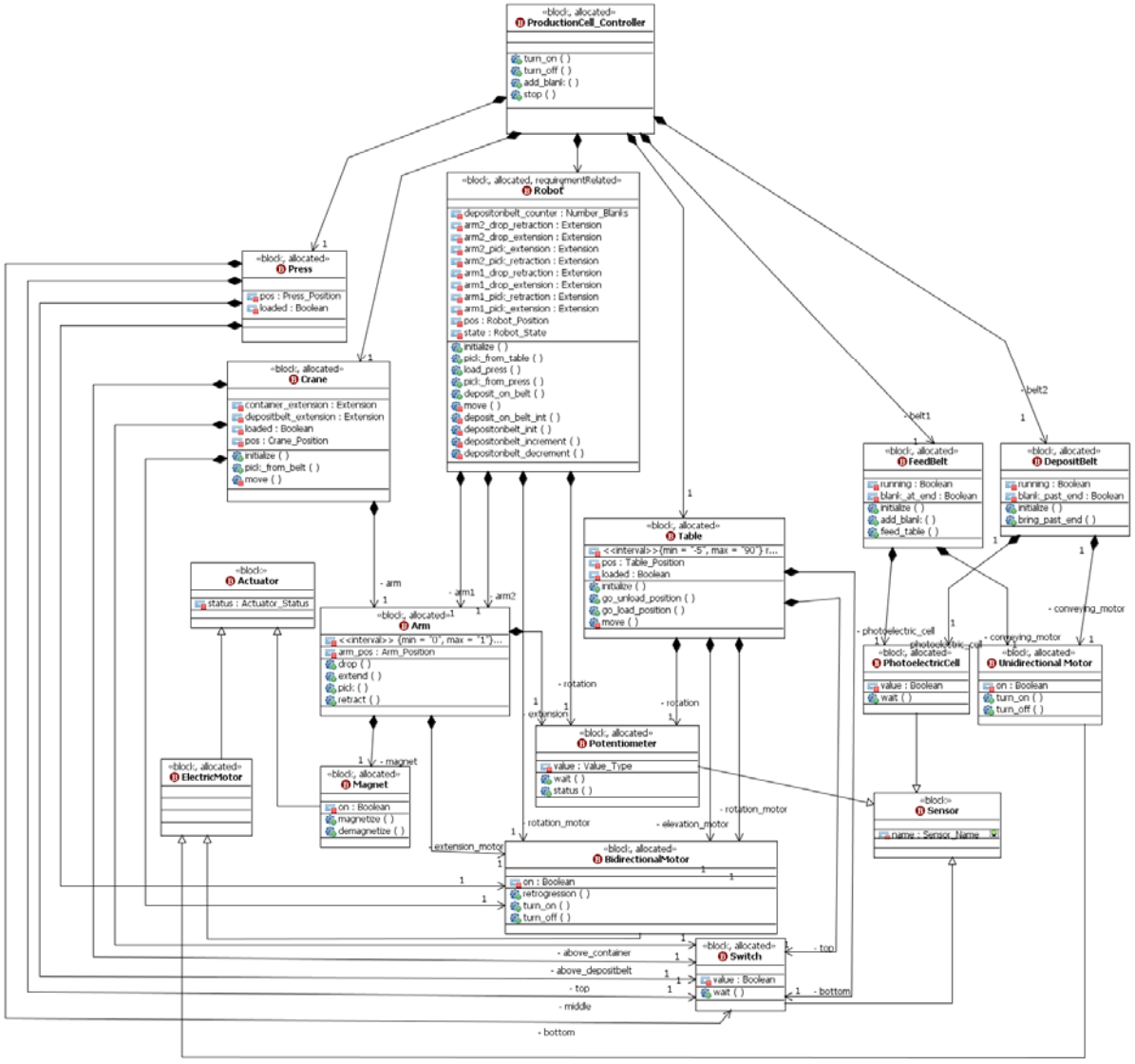

FIGURE 39 - FULL SOFTWARE STRUCTURE DIAGRAM (FOCUS: PRODUCTIONCELL_CONTROLLER)

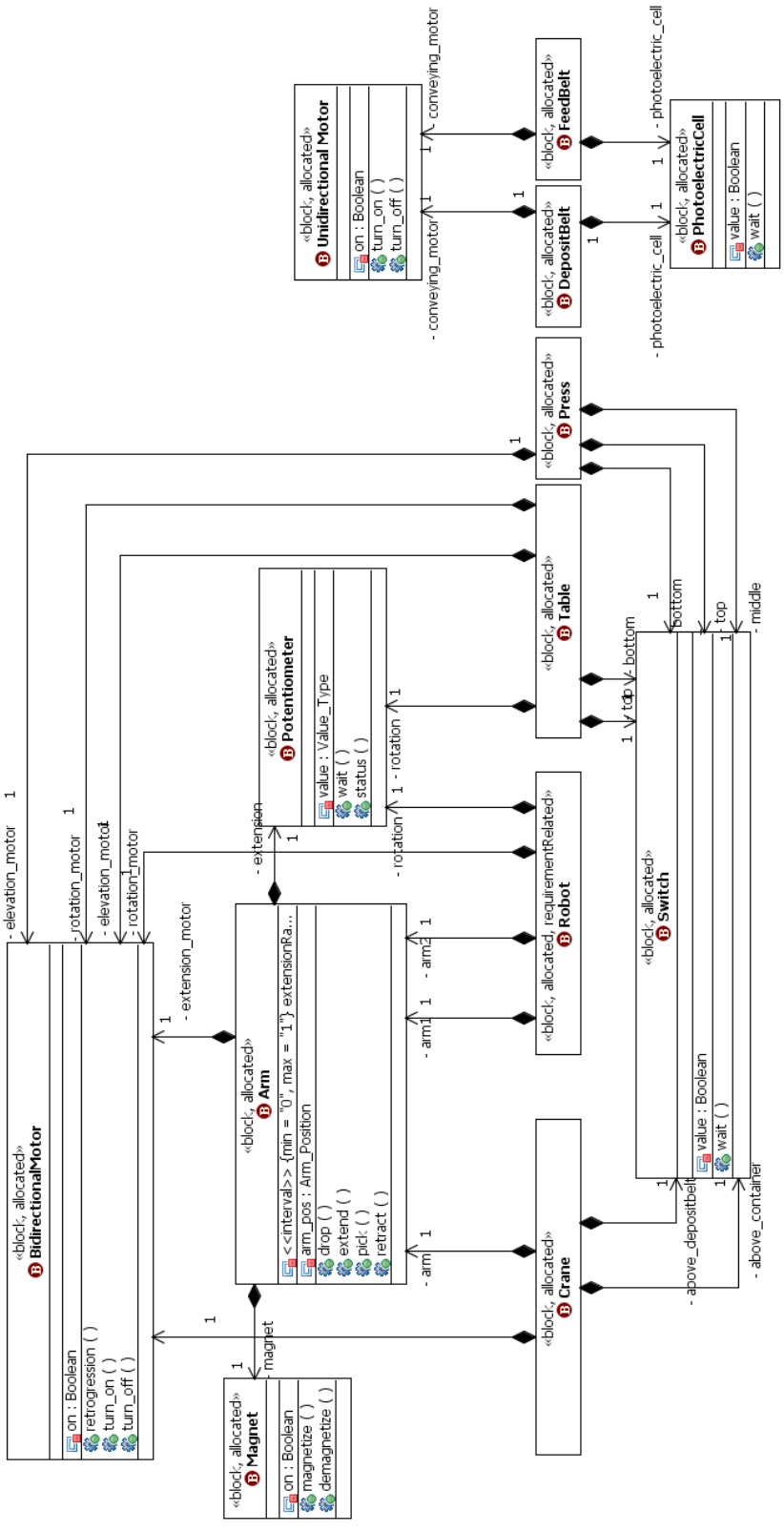

FIGURE 40 – PART OF STRUCTURE DIAGRAM (FOCUS: MACHINE SOFTWARE PARTS)

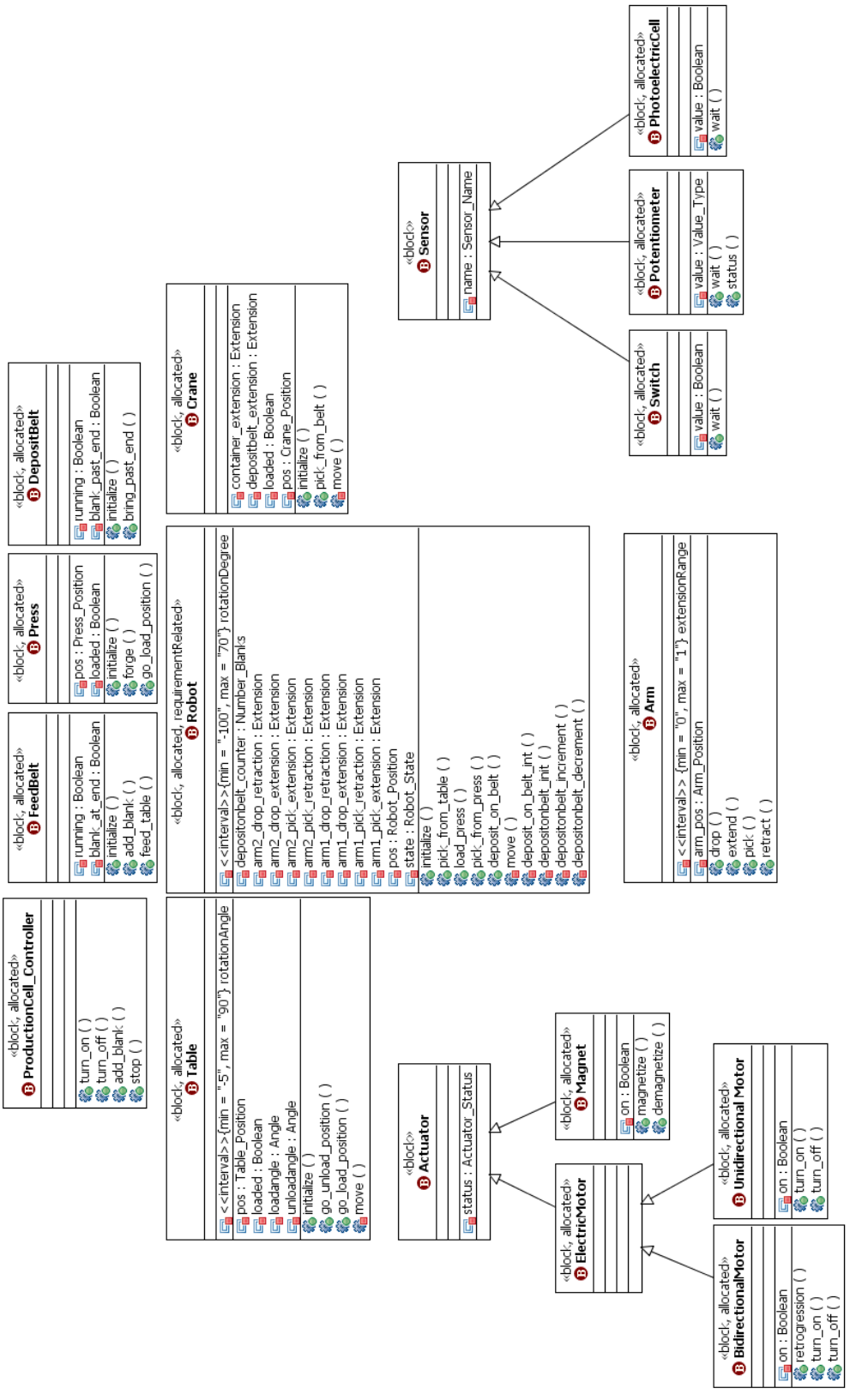

FIGURE 41 – VIEW OF THE ATTRIBUTES/OPERATIONS OF THE SOFTWARE BLOCKS OF THE MACHINES.

State machines

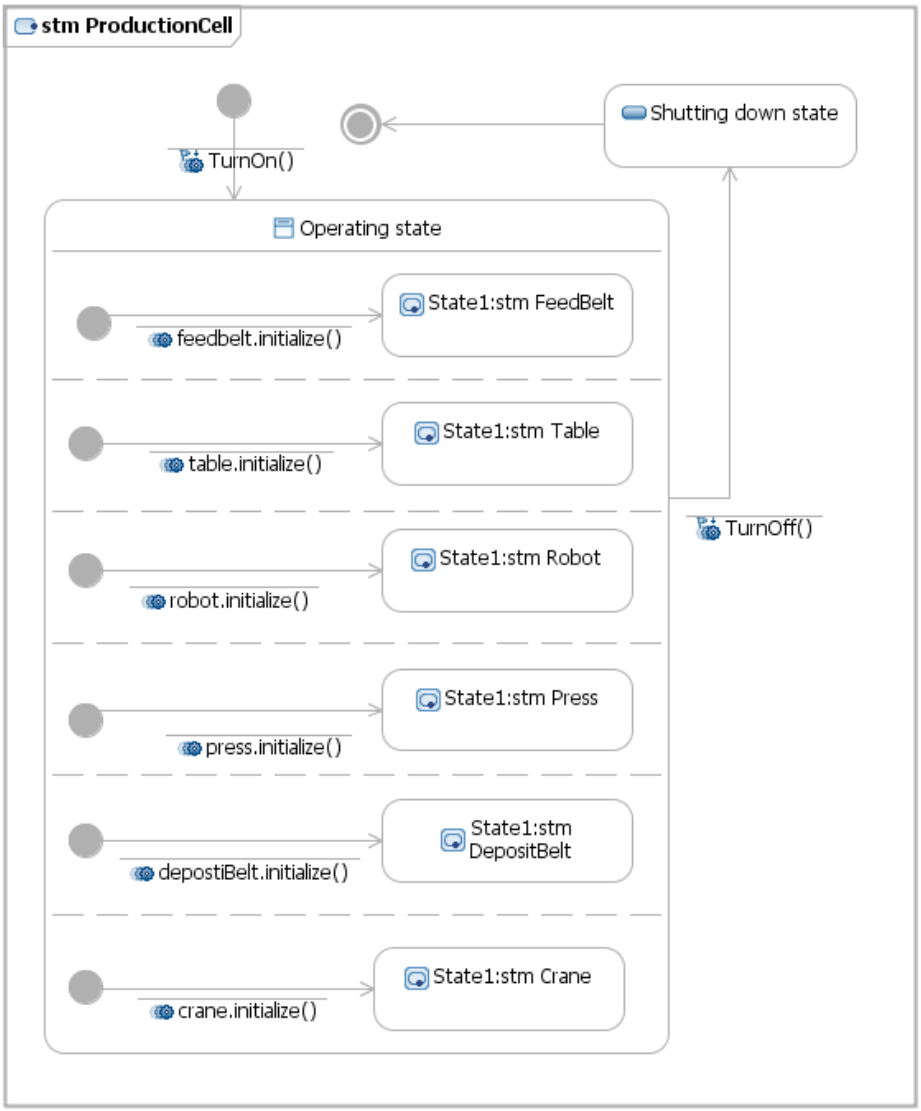

FIGURE 42 - STATE MACHINE FOR BLACK PRODUCTIONCELL_CONTROLLER.

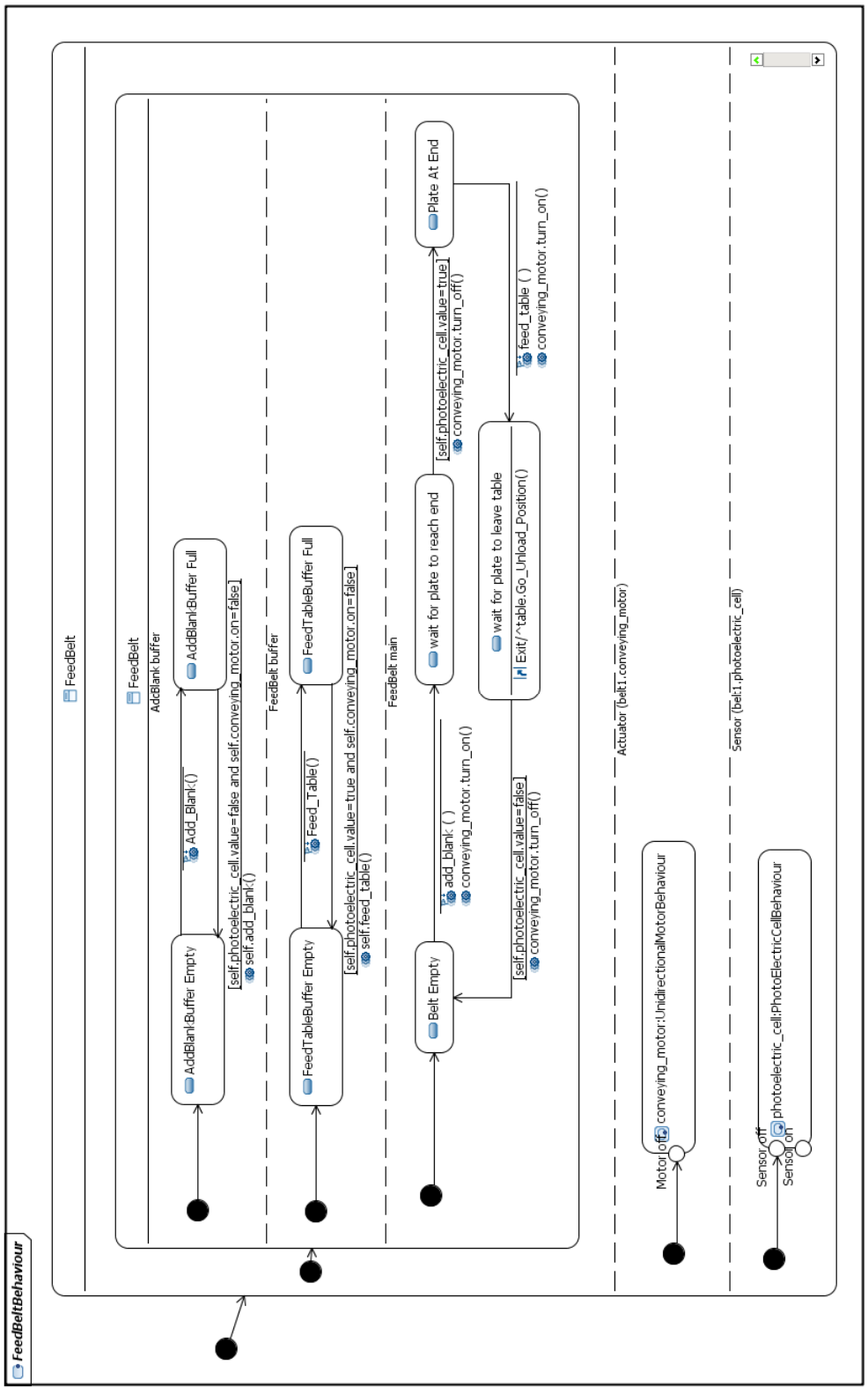

FIGURE 43 - STATE MACHINE FOR BLACK FEEDBELT.

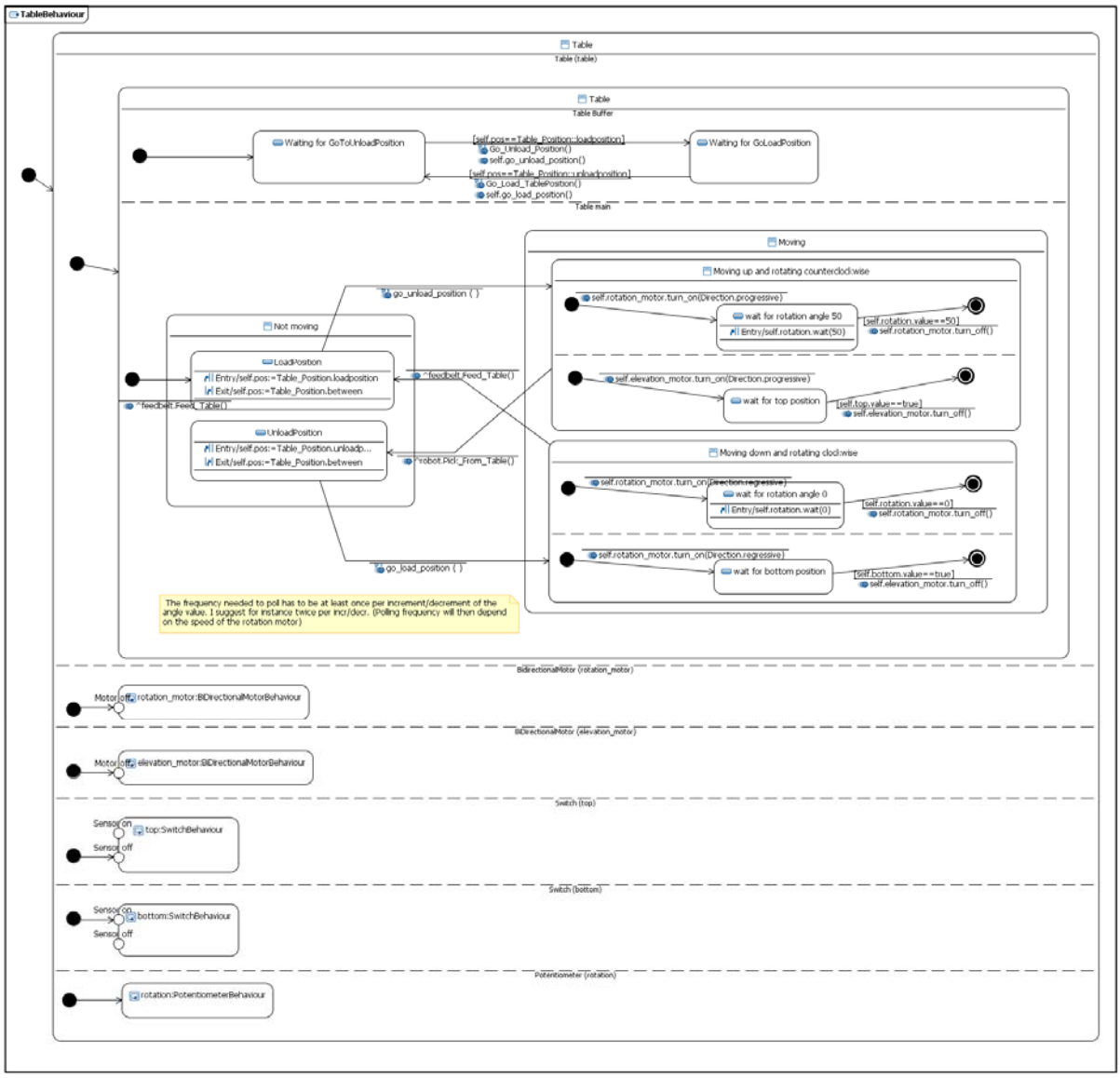

FIGURE 44 - STATE MACHINE FOR BLOCK TABLE, WHOLE.

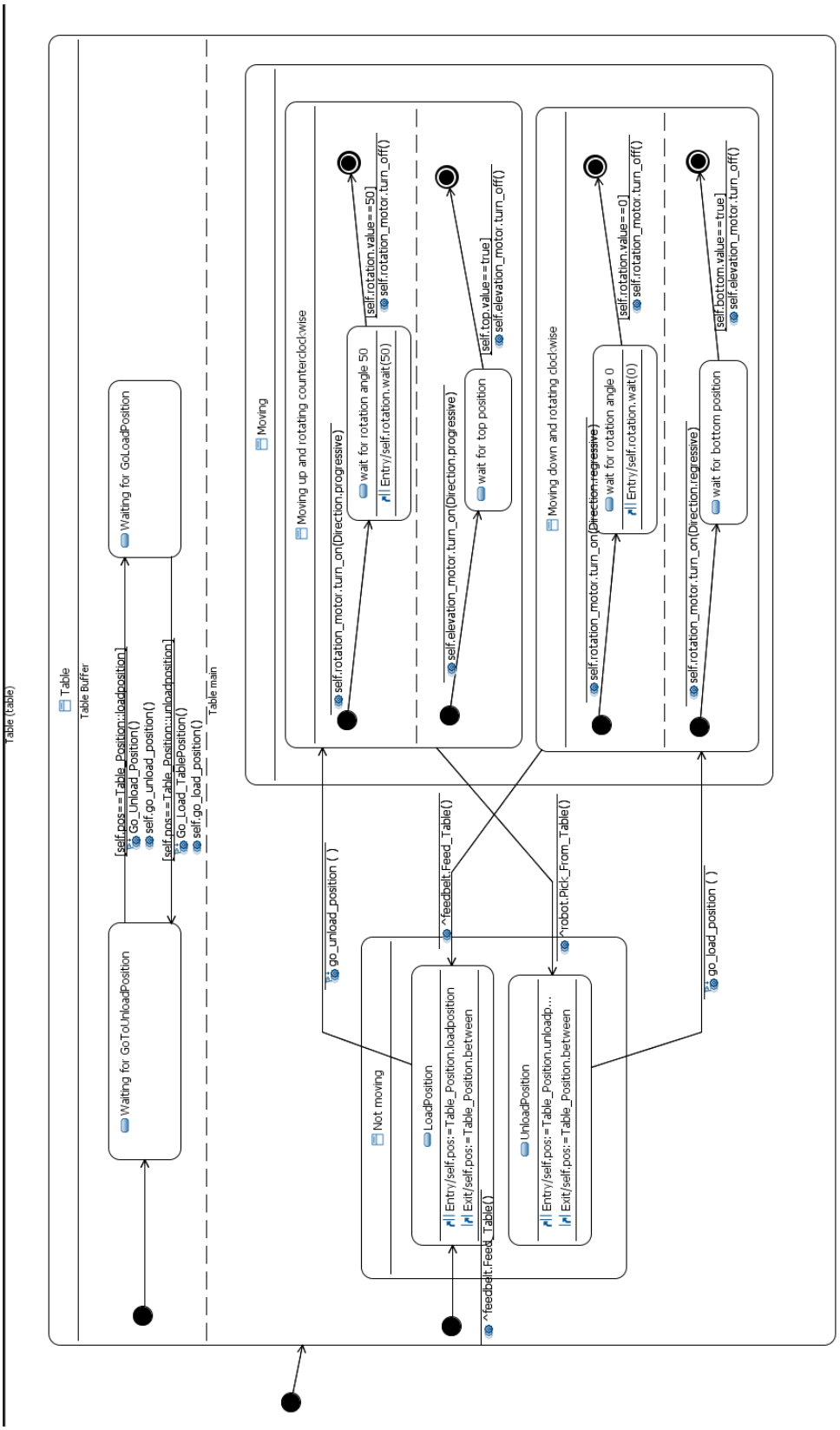

FIGURE 45- PART OF STATE MACHINE FOR TABLE, SHOWING ONLY BUFFER AND MAIN REGIONS. FOR WHOLE DIAGRAM, SEE FIGURE 44.

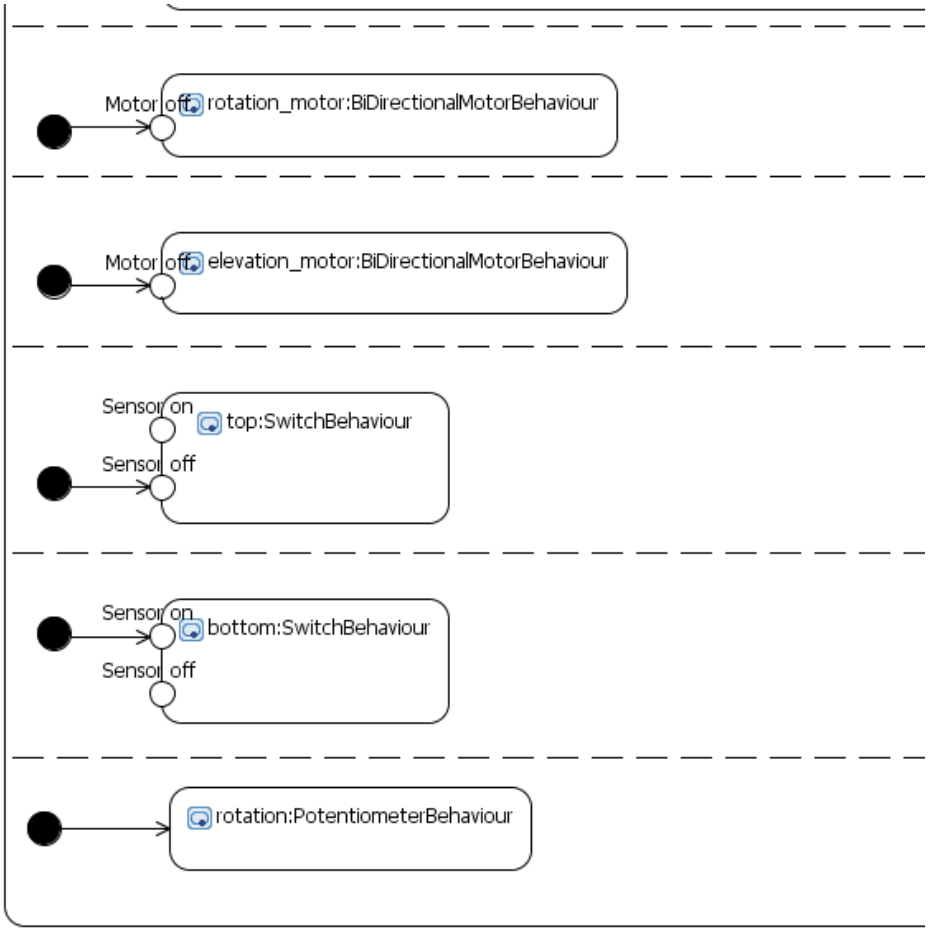

FIGURE 46 - PART OF STATE MACHINE FOR TABLE, SHOWING ONLY THE PART REGIONS. FOR WHOLE DIAGRAM SEE FIGURE 44.

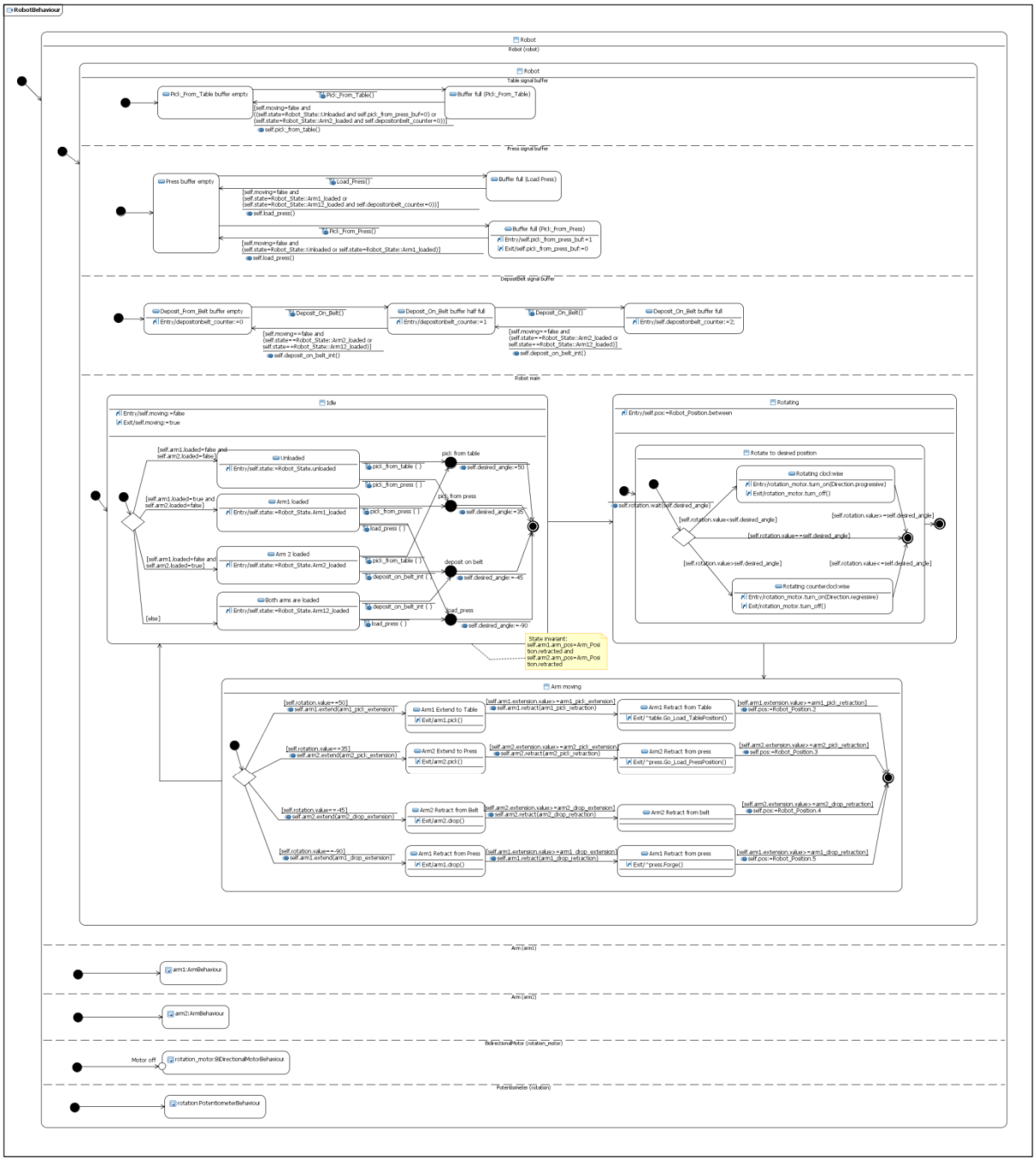

FIGURE 47 - STATE MACHINE FOR BLOCK ROBOT.

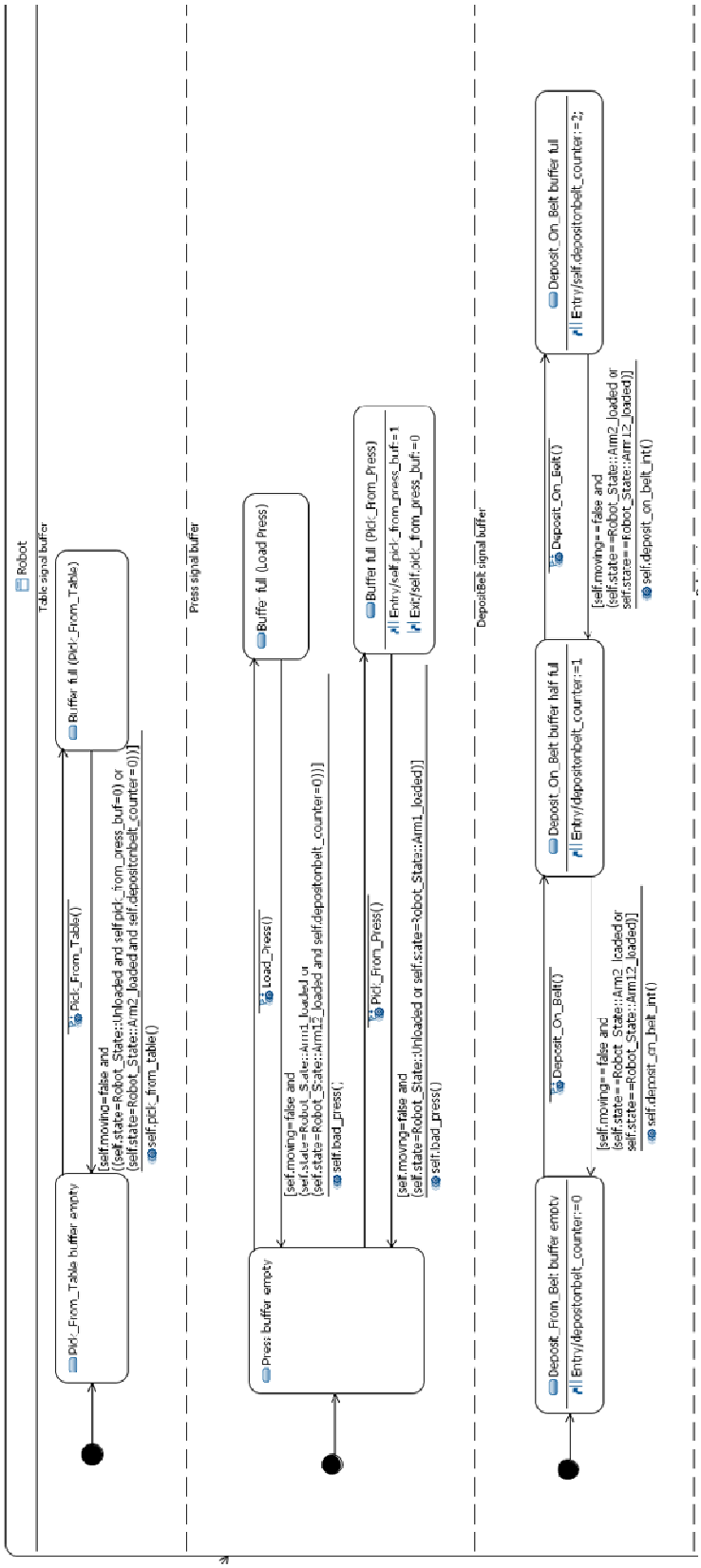

FIGURE 48 – DETAILED VIEW OF BUFFER REGIONS FROM STM [BLOCK] ROBOT [OPERATING]. WHOLE DIAGRAM IN FIGURE 47.

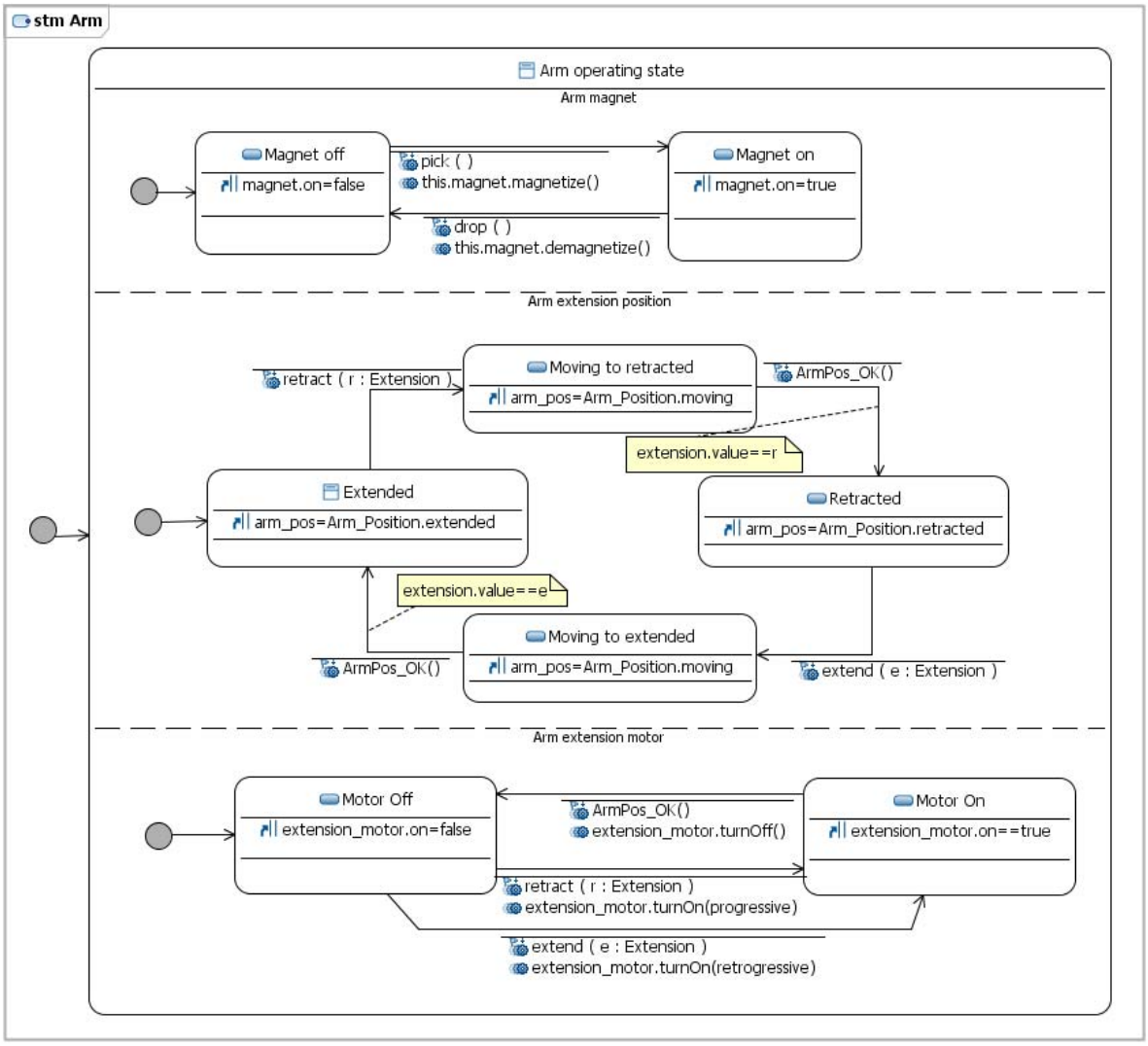

FIGURE 49 - STATE MACHINE FOR ARM.

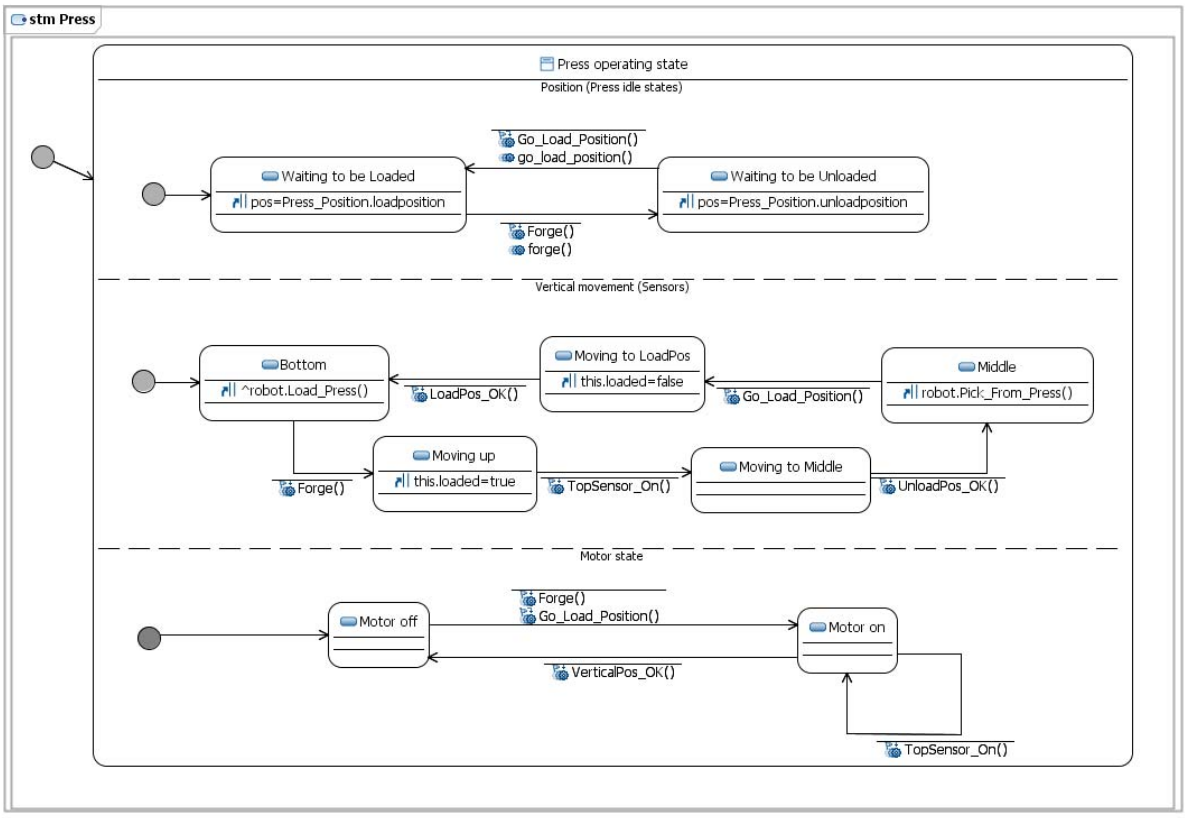

FIGURE 50 - STATE MACHINE PRESS.

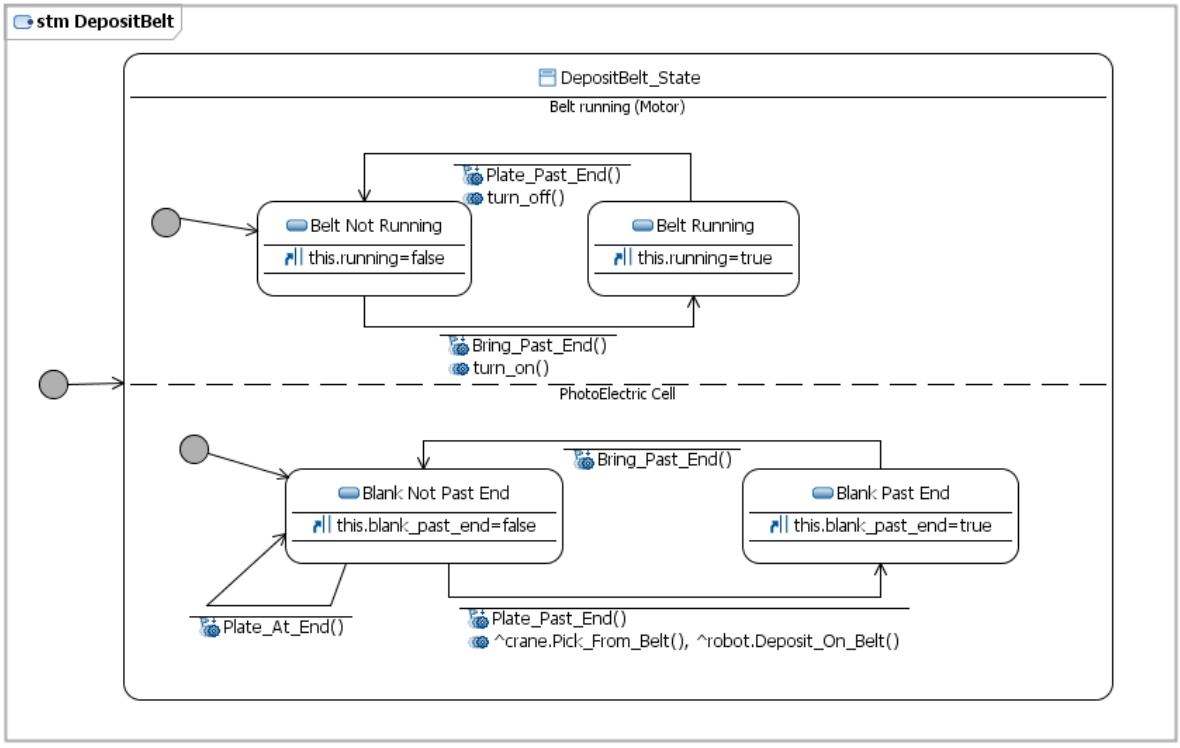

FIGURE 51 - STATE MACHINE FOR BLOCK DEPOSITBELT.

Activity diagram

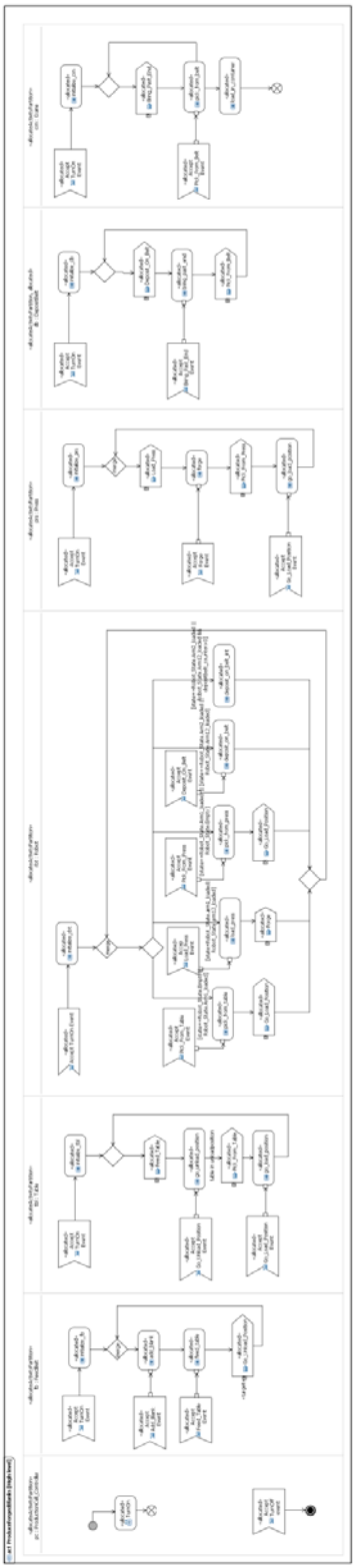

FIGURE 52 - ACTIVITY DIAGRAM HIGH LEVEL OVERVIEW, ALL PARTITIONS.

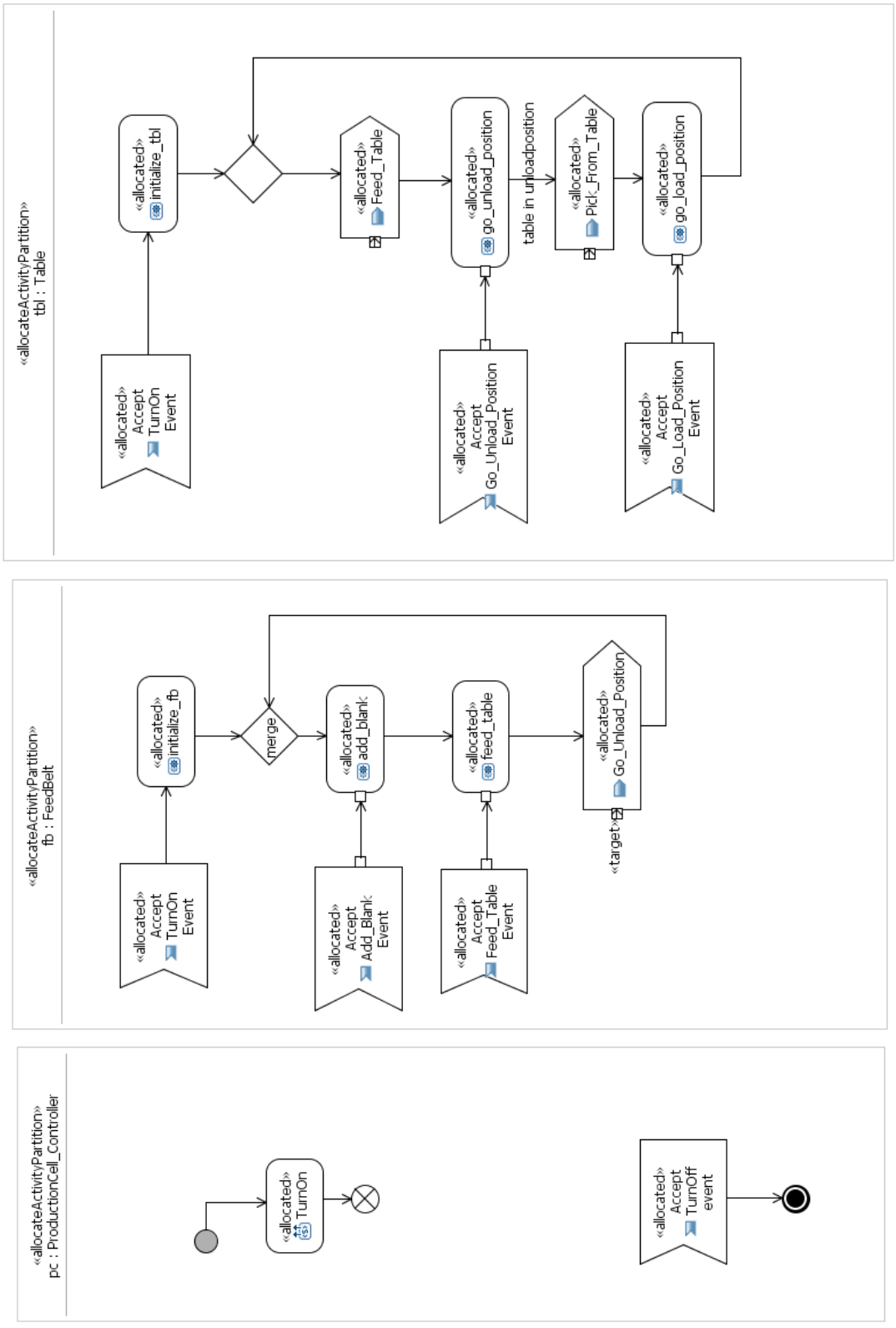

FIGURE 53 - ACTIVITY PARTITIONS FOR CONTROLLER, FEEDBELT AND TABLE.

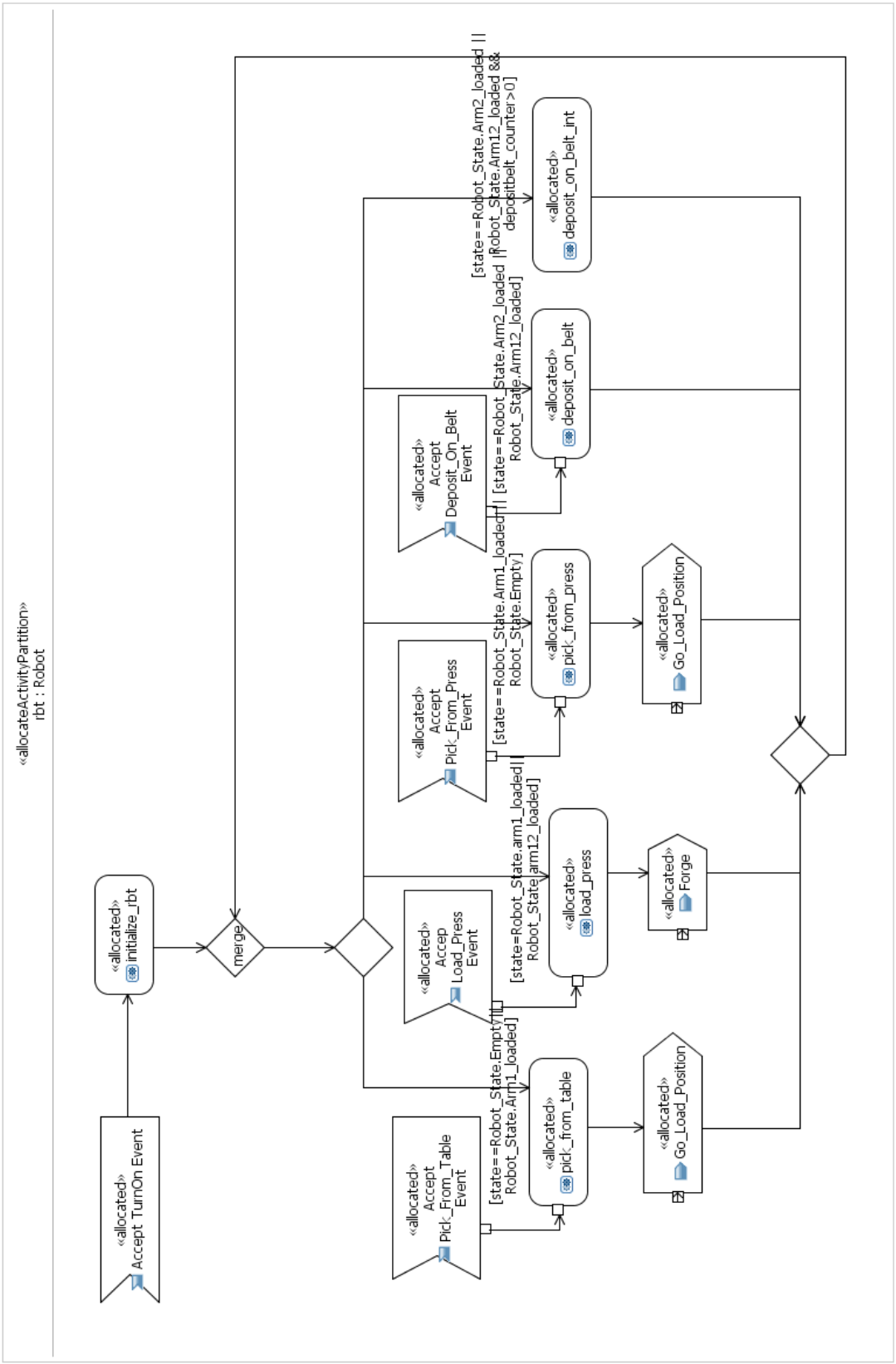

FIGURE 54 - ACTIVITY PARTITION FOR ROBOT.

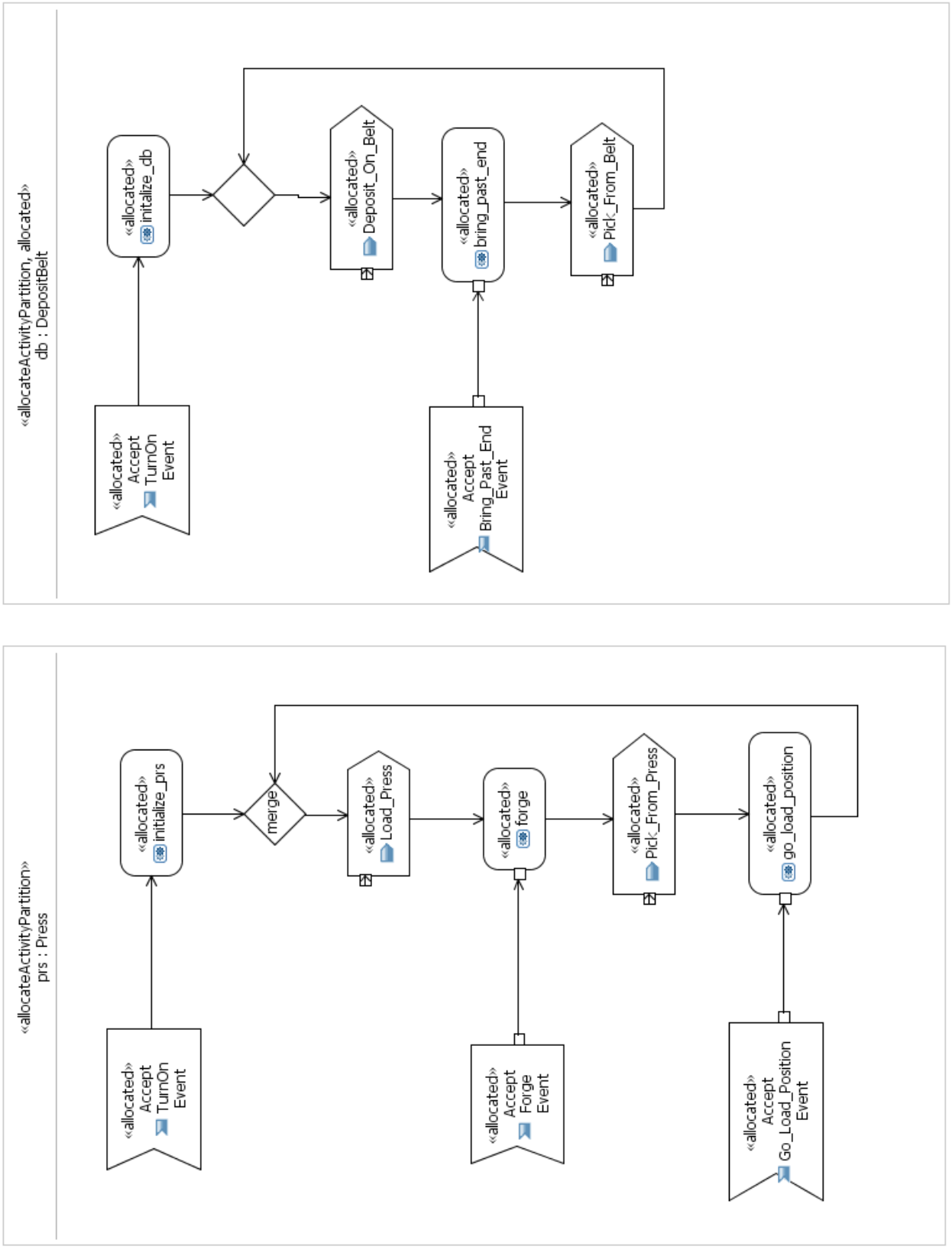

FIGURE 55 - ACTIVITY PARTITIONS FOR PRESS AND DEPOSITBELT.

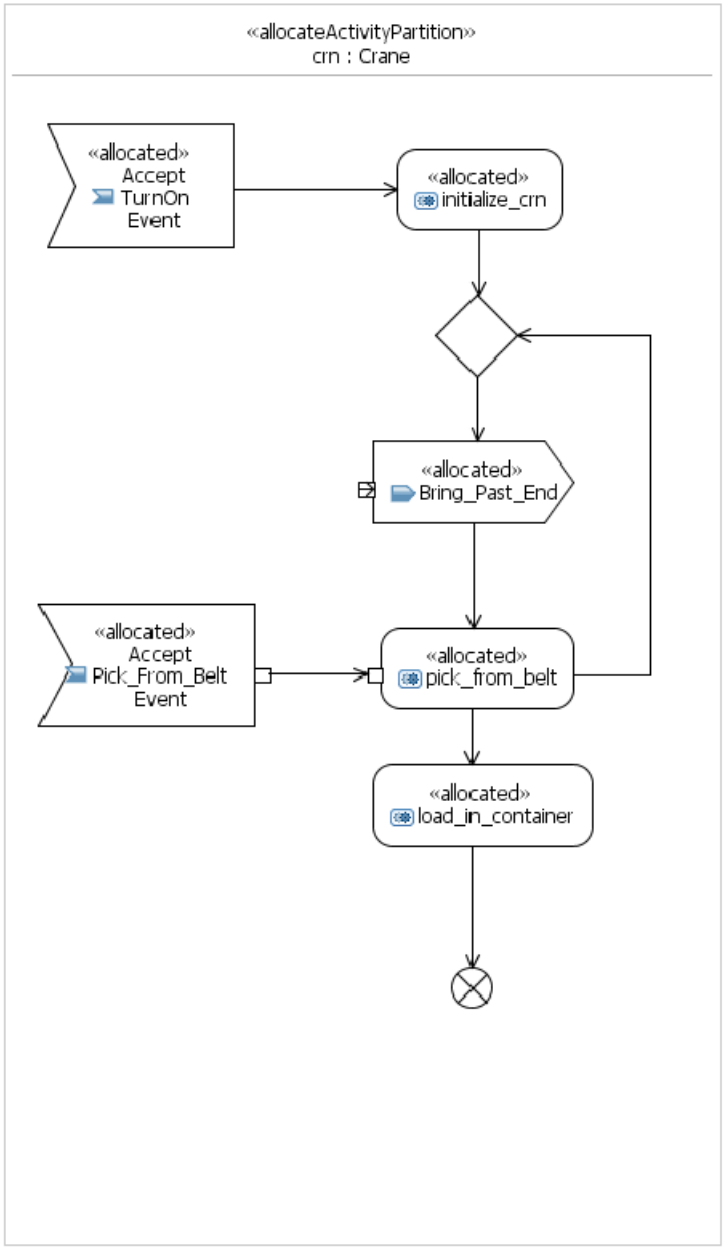

FIGURE 56 - ACTIVITY PARTITION FOR CRANE.

6.4. Use Case Descriptions

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.

Use Case	*Produce forged blank*
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 blank 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 blank 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

6.5. System 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 representing 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	