MASTER

Specification of a distributed real-time reactive control system, using software/hardware engineering method

Reuter, E.A.J.

Award date:
1997

Link to publication
Master's Thesis:

**Specification of a Distributed Real-time Reactive Control System, using Software / Hardware Engineering Method**

By ing. E.A.J. Reuter

Coaches : dr. ing. P.H.A. van der Putten  
          dr. ir. J.P.M. Voeten  

Supervisor : prof. ir. M.P.J. Stevens

Period : September 1996 - August 1997
Abstract

The thesis describes the specification of a next generation Buhrs personalised mailing machines using Software / Hardware Engineering method. A mailing machine creates mailing packets by stacking mailing items on each other at a transporter. Current mailing machine architecture and implementation of control software restrict integration of new features. In a co-operative project of Buhrs Zaandam B.V., Eindhoven university of Technology and TNO section applied physics a new architecture and control is specified. For the Eindhoven University of Technology the project is an interesting case to put the Software / Hardware Engineering Method in practice. By putting the method in practice the method is verified, extended and improved.

Next generation mailing machine requirements and technologies to be used are specified. Different conceptual solutions have been explored, only the final conceptual solution is described. Current generation mailing machine architecture is explored to find the restrictions that make new requirements hard to implement. With the identified problems in mind a new architecture is developed. Major new features in mailing machine architecture are distribution of control and data and separation of control and configuration. To transport mailing items in the mailing machine, a model is specified that handles all transporter layouts. Further a rotary feeder is specified.

Sensors and actuators of transporters and a rotary feeder are identified and modelled. Distinctive behaviour of the mailing machine is identified and specified in scenarios. In a system level analysis for different scenarios collaborating objects are specified guided by a framework offered by SHE. The framework incorporates different graphical representations in which collaborating objects, the interactions between the objects, relations between objects and structure have been visualised. In a sub-level system analysis the internal behaviour of the transporter and rotary feeder are specified. Finally functional behaviour and structure are formalised by a description in the formal language named POOSL (Parallel Object Oriented Specification Language). The different models are validated in a simulator and behave as specified.

Buhrs provides a prototype mailing machine with the proposed architecture to verify the specification. An implementation of the specification must be realised to implement the specification on the prototype. The project is continued by a graduate student.
Acknowledgements

I like to thank everybody who contributed in the completion of my master’s project. A number of them I like to mention explicitly.

I thank my coaches Piet van der Putten and Jeroen Voeten for the opportunity to specify a control system using a specification method and their support during the project. My supervisor prof. Mario Stevens. Marc Geilen who developed the simulator the models are verified with and his support. Robin Michielsen for being there. Terry Dennemans, who continues the project, for co-operation during the last months. Rian van Gaalen who cares for everything. And all other members of the section Information and Communication Systems.

I thank Buhrs Zaandam B.V. for the opportunity to specify a mailing machine and providing a prototype mailing machine. I thank the people of Buhrs I co-operated with: Jan Hendriks, Jan-Willem van Kampen, Nico van Kampen, Ron Timmerman and Mathijs Vonder. From TNO-TPD, I like to thank Frank Verhofstad.

Finally I like to thank my parents, sister, relatives and all my friends for their interest, understanding and contribution in my well-being.
## Contents

1. Introduction .................................................................................. 9  
   1.1 BAST ................................................................. 9  
   1.2 Evaluating Software Hardware Engineering ................................. 9  
   1.3 Project .................................................................. 9  
   1.4 Thesis Organisation ...................................................... 10  

2. Problem Definition ...................................................................... 13  
   2.1 Next Generation Personalised Mailing Machines ......................... 13  
   2.2 Evaluation of specification method ........................................... 14  
   2.3 Initial Requirements Description ........................................... 15  
      2.3.1 Functional Requirement List ........................................ 15  
      2.3.2 Purpose Description .................................................. 16  
      2.3.3 Prescribed Technologies and Topologies ............................ 16  

3. Introduction to Software / Hardware Engineering ......................... 19  
   3.1 Elements of SHE ...................................................... 19  
   3.2 SHE context and Phases .................................................. 20  
      3.2.1 Initial Requirement Description ................................ 20  
   3.3 Essential specification ...................................................... 21  
      3.3.2 Architecture Structure Diagram .................................. 22  
      3.3.3 Message Flow Diagram .......................................... 22  
      3.3.4 Instance Structure Diagram ..................................... 23  
      3.3.5 Interaction Diagrams ............................................. 23  
      3.3.6 Object Class Diagram ........................................... 23  
      3.3.7 Unified Model ...................................................... 23  
      3.3.8 Requirements catalogue .......................................... 23  
   3.4 Extended specification Modelling ........................................... 24  
   3.5 Implementation ................................................................ 24  

4. Essential Specification .................................................................. 25  
   4.1 System Components Analysis .............................................. 25  
      4.1.1 System Info Handler ............................................... 26  
      4.1.2 Production Line ..................................................... 27  
      4.1.3 Transporter Section ................................................ 27  
      4.1.4 Transporter .......................................................... 27  
      4.1.5 Station ................................................................. 29  
      4.1.6 Function ............................................................... 29  
      4.1.7 Feeders ............................................................... 30  
      4.1.8 Ejector ................................................................. 32  
      4.1.9 Reader ................................................................. 32  
      4.1.10 Sealer ................................................................. 32  
      4.1.11 Stacker ............................................................... 32  
      4.1.12 Town Separator .................................................... 32  
      4.1.13 Fly-over ............................................................ 32  
   4.2 Conceptual Solution ................................................................ 33  
   4.3 Mailing machine architecture ................................................ 35  
      4.3.1 Inventory of Current System Architecture ....................... 35  
      4.3.2 Distributed Control Prepared System Architecture with remote I/O .................................................... 37  
      4.3.3 Fully Distributed Control ......................................... 38  
      4.3.4 Meta Model of mailing machine .................................. 39  
   4.4 Behaviour Definition using Scenarios ....................................... 40  
      4.4.1 Identified Scenarios ................................................ 40  
      4.4.2 Booting Scenario .................................................... 41  
      4.4.3 Job Preparation Scenario ......................................... 41  
      4.4.4 Normal Operation Scenario ....................................... 41  

-5-
Chapter 1

Introduction

1.1 BAST
Buhrs Zaandam is a world leading manufacturer of mailing machines. A mailing machine composes mailing packets by stacking single items. The mailing packets can be labelled and sealed. Current mailing machine architecture and control are exploited completely, new features and larger mailing machines are hard to implement. In co-operation with TNO section applied physics and Eindhoven University of Technology, Buhrs started the BAST-project to develop a new (sub) control software and a new architecture.

In co-operation with Buhrs engineers and TNO a new control sub-system for a next generation mailing machine is specified during collective sessions. Eindhoven University of Technology takes charge of the specification of a generic model of transport of mailing packets and a rotary feeder. TNO specifies a controller that prepares, initiates and corrects mailing packet creation. Buhrs provides information on mailing machine components and current implementation.

1.2 Evaluating Software Hardware Engineering
By putting the Software Hardware Engineering (SHE) method in practice, SHE can be verified, extended and improved. A mailing machine is a complex real-time distributed reactive system, which makes it an interesting case to put SHE in practice. The new architecture features topological distributed control on local controller boards that are connected by a Controller Area Network, additionally completely new features must be specified and modelled. The complete specification and design path can be exercised from specification to implementation on a prototype mailing machine. Tools are used in practice and can be adapted to practical requirements.

1.3 Project
My contribution to the specification and design of new control software for a next generation mailing machine and evaluation of the SHE-method consists of:
- modelling a generic model for mailing machine transporters
- modelling a rotary feeder
- verifying the generic model of transport and the model of the rotary feeder
- documenting specification
- evaluation of Software / Hardware Engineering

---

1 BAST is the dutch abbreviation for 'Buhrs Analyse en Specificatie Traject', the neglisch equivalent is 'Buhrs Analysis and Specification Trajectory'.

.9.
Chapter 1.

- proof developed models on prototype mailing machine

**Generic model of a transporter**
A generic model of a transporter is specified and modelled. The model takes care of all aspects that concern transport, like tracking mailing packets, couplings between transporters, side production lines etc. The model handles all transporter layouts and is incorporated in models of functions that image transport.

**Model of a rotary feeder**
A rotary feeder will be specified and modelled. The rotary feeder components that take part in control are identified and modelled. The behaviour of the rotary feeder is specified, and an essential specification created. The specification is formalised in POOSL.

**Verifying**
To verify the models interactions with a system info handler are required. The system info handler is not yet formalised in POOSL by TNO. Therefore a specification of the system info handler with a minimal required behaviour is formalised in POOSL to enable testing the different models.

**Documenting specification**
Specification and design have been documented so far. The SHE method prescribes documentation, like requirements list, architecture decisions, design decisions etc.. This document incorporates documentation on specification and design of next generation mailing machines.

**Evaluation of Software Hardware Engineering**
Providing feedback on practising SHE to developers of the SHE-method is one of the goals of the project. Feedback is necessary to verify, extend and improve the SHE method. The Buhrs case is used as the major case for the verification and evaluation of the simulation tools that have been developed simultaneously with the Buhrs project.

**Proofing models**
To proof correctness of the specification, Buhrs provides a prototype of a next generation mailing machine. On the prototype the implementation of the specification can be verified. The prototype incorporates the proposed architecture. Unfortunately lack of time prevented me to implement the specification.

### 1.4 Thesis Organisation

The thesis describes the essential specification of the mailing machine specified using the SHE-method.

- **Problem Definition**, describes objectives of the project: requirements of next generation mailing machine and evaluation of the Software/Hardware Engineering method (chapter 2).
- **Introduction to Software / Hardware Engineering**, gives a brief discussion on elements of the she method (chapter 3).
- **Essential Specification**, documents the essential modelling phase of the mailing machine based on the SHE-frame-work (chapter 4).
- **Implementation on prototype mailing machine**, describes requirements and purpose of prototype mailing machine (chapter 5).
- **Conclusion**, conclusions and recommendations (chapter 6).
- **System Level Analysis**, visualisation of system level objects and behaviour (appendix A).
- **Generic Transporter Model**, visualisation of objects and behaviour of the generic transporter model (appendix B).
- **Rotary Feeder**, visualisation of objects and behaviour of the rotary feeder (appendix C).
- **System Info Handler**, visualisation of objects and behaviour of the system info handler (appendix D).
- **Formalisation**, formalisation of behaviour and architecture in formal specification language (appendix E).
Chapter 2

Problem Definition

Buhrs is developing a next generation mailing machines based on new technology. The control system of the new mailing machines will be more complex than current control systems and will introduce complete new problems to the Buhrs engineers, like concurrency and distribution of control and data. To overcome this technology jump Eindhoven University of Technology and TNO section applied physics support the development of the new control system. The control system will be specified in the specification method Software Hardware Engineering (SHE) [P+97] developed by Piet van der Putten en Jeroen Voeten at Eindhoven University of technology. TNO\(^2\) provides expertise in project management and development of machinery for industry.

Problem definition of Buhrs Case:
- developing a new state of the art control system for a next generation mailing machines using the co-specification method SHE (paragraph 2.1)
- evaluate the co-specification method SHE on a real life case in order to improve, extend and verify the SHE- method (paragraph 2.2)

2.1 Next Generation Personalised Mailing Machines

Introduction to Mailing Machines

A mailing machine is a machine that creates mailing packets. A mailing packet consists of one ore more items, p.e. magazines, news papers, advertisement cards or gadgets like CD’s. Mailing packets can be sealed, labelled and/or stacked on other mailing packets.

A mailing machine consists of a main production line built from mechanical stations (see Figure 2-1). A mechanical station consists of a substructure on which zero or more functional modules are built. Further each station can transport a mailing packet. When a mailing packet passes a function, the function can act on the mailing packet. At the Gathering Section different specialisations of feeders place a mailing item on a mailing packet or insert a mailing item between pages of a magazine. The Packaging Section wraps a plastic or paper package around a mailing packet. The section can add a pre-printed address label or print an address on the mailing packet. The Stacking Section bundles mailing packets. A town separator can be included in control that sorts mailing packets on postal region. A mailing packets that is not composed as desired can be ejected.

\(^2\) Netherlands Organization for Applied Scientific Research
Developing a future control system

Buhrs mailing machines are continuously improved and extended with new features. The architecture and technology currently used, are exploited completely. New features are hard to implement in current implementation. Buhrs' solution is the development of a new generation mailing machines with a complete new architecture and use of new technologies.

Buhrs main objectives (requirements) towards next generation mailing machines are:

- plug and play of stations
- reduce installation-, preparation- and test time
- stand alone testing of functions
- complex side production lines
- remote control
- version management
- extendible with future features

All requirements of current and next generation mailing machines, completed with technologies and topologies that must be used, are described in an initial requirement description (paragraph 2.3), as imposed by the SHE-method. This description also contains a brief discussion that fundamentals customers requirements.

2.2 Evaluation of specification method

The development of a completely new control system for a personalised mailing machine is an interesting case to verify, extend and improve the SHE-method. During the development of the new control system the method deals with distribution of control, distribution of information, real-time constraints and co-operation of hardware and software. Further complete new features not implemented yet, will be developed.

Objectives of the TU-Eindhoven in this case are:

- verify, extend and improve the SHE method
- introduce new technology to the industry, in this case to Buhrs and TNO
- offer students experience with a specification method on a real life case

During the case the complete SHE-method is put into practice. Basis for evaluation of the method:

- is the method used as it is intended to be used?
- do all aspects of the method function as intended?

3 Paragraph 4.3.1 discusses problems due to current architecture.
• is SHE complete? does SHE cover all aspects necessary to develop a specification?
• experiences of development in contrast to other methods
• acquiring heuristics

2.3 Initial Requirements Description

SHE specification starts with the creation of an Initial Requirement Description. The Initial Requirements Description (IRD) is the input for modelling with the SHE-method. The IRD includes three textual descriptions a Requirement List, a Purpose Description and a Prescribed Technologies and Topologies item. The requirements list contains functional requirements, environmental conditions, physical constraints, price and implementation oriented details of the new generation mailing machines. The Purpose Description describes functionality the mailing machine should offer to the users. To identify desired functionality, needs of the users are made explicit. Prescribed Technologies and Topologies describes technologies and topologies imposed by Buhrs Zaandam BV.

2.3.1 Functional Requirement List

Plug and Play of stations
Mailing machine control is able to determine the layout configuration of a mailing machine. The layout configuration contains information about composure of the mailing machine. That is the ordering of transporters and what functions are available on transporters.

Reducing installation time
The time necessary for testing, programming and configuration must be reduced. Use of a field bus and flexible configuration reduce installation time and reduce developing time of control software.

Stand alone testing and preparation of functions
A function can be tested and prepared apart from a complete mailing machine. During a test functionality of a function and its control system can be tested. During preparation the function can be set up for a job.

Updates, maintainability and version management
Control software must be easily updated, new control software is able to use old configuration. Mailing machine control software is composed of mailing machine independent software modules. Both points improve maintainability and version management.

Remote Control
A mailing machine can be accessed using remote control, to control the mailing machine from distance by telephone. Using remote control the mailing machine can be configured, new or updated control software can be installed and management information can be accessed.

Complex side production lines
A side production line can be equally complex as a main production line, i.e. a side production line can have functions and have side production lines it self.

Personalised mailing packets
The mailing machine has to create personalised mailing packets. A personalised mailing packet is a mailing packet composed of items that are only required by the mailing packet. A personalised mailing packet is created by selectively activating functions, p.e. feeding a required item or print a unique label. Personalised information is provided by an editor digitally or by bar-codes on address labels.

Town separator
The mailing machine must be able to bundle mailing packets destined for the same postal region. The mailing machine identifies postal regions by zip-codes that are provided by an editor.

Address reader
The mailing machine must control address readers that read address labels. An address label can contain personalised mailing information and/or addresses.
Chapter 2.

Management Information
The mailing machine registers all relevant information that is valuable for evaluation of the production process.

Machine speed
Mailing machine control is capable of achieving a performance of 40,000 mailing packets per hour, depending on mailing machine composure.

2.3.2 Purpose Description

Personalised Mailing Packets
Editors want to create mailing packets composed of solely items that are of interest to a certain consumer. P.e advertising campaigns, mail-orders or information retrieval.

Town Separator
Editors want mailing packets ordered on postal region to bargain a cheaper postal rate from the postal service.

Management Information
Management information is necessary to identify bottle-necks in production, optimise production for maximal through-put, estimate performance of operators and overall performance of the mailing machine.

Stand alone preparable and testable
Stand alone testing and preparation has as advantage that a function can be prepared and tested without being included in a complete mailing machine. Production of the mailing machine continuous which reduces maintenance and preparation costs. Functions can be prepared more thoroughly to achieve a better performance.

Plug and Play
Future generation mailing machines have stand alone functions separated mechanically from a transporter. They are equipped with their own drives and can be placed where ever possible. For that generation the plug and play feature is required.

Local control in each function
A function can only be tested stand alone when control is a part of the function. Further local control introduces concurrency, which relieves central control and reduces the amount of communications on the field bus. For plug & play local intelligence is required to identify a function or transporter.

2.3.3 Prescribed Technologies and Topologies

Specification method
The co-specification method Software / Hardware Engineering will be used for the specification of a new control system for next generation Buhrs personalised mailing machines.

Development Tool
A micro controller development tool from Keil Software for the 8051 and 251 family of microprocessors will be used during development [Kei96]. The tool contains an integrated development environment, a C-compiler, a macro assembler, a debugger / simulator, a tiny real time system kernel and other utilities. The compiler is a complete implementation of the ANSI standard for the C-Language. The real time kernel supports 16 concurrent tasks.

80C592 controller board
Each transporter and function will be equipped with a controller board. The controller board control sensors and actuators of functions and transporters and can be equipped with local intelligence.

The controller board contains a 16 Mhz. 80C952 Microprocessor, 64 KB code memory, 32 KB data memory, a RS-232 interface, a Control Area Network interface, 12 digital input ports and 6 digital output ports.
2.3. Initial Requirements Description

Field Bus
A field-bus connects all transporters and functions with each other using a minimum number of cables. The reduction of the number of cables results in faster and easier installation. The layout of the mailing machine is easily changed.

Controller Area Network
A Controller Area Network (CAN) [Bos91] will be used to connect the different modules. CAN is a reliable communication medium. Calculations show that CAN has enough performance to transmit all control and data required between the transporters and functions. The maximum length of the network and the maximum number of nodes cover mailing machines to be build in future. CAN-controllers are available in popular micro controllers which can be implemented on controller boards.

CAN is a reliable multi-master shared broadcast bus which runs at 500 Kbits per second at 100 meters. The protocol is based on sending frames which have a variable length, between 0 and 8 bytes. Each frame has an identifier, the identifier of a frame with the lowest binary value has the highest priority. The residual error probability is several magnitudes less than 3x10⁻⁵.
Chapter 3

Introduction to Software / Hardware Engineering

SHE is a specification, analysis and design method for complex real-time reactive distributed systems that must be realised in hardware and/or software. The method focuses specially on the modelling of the behaviour of a system to be designed as a collection of collaborating objects.

This paragraph is an introduction to the SHE method. It offers a brief discussion on the meaning and purposes of elements that the SHE-method offers. A complete discussion on the SHE-methodology is described in the Ph.D. thesis ‘Specification of Reactive Hardware/Software Systems’, of Piet van der Putten en Jeroen Voeten [P+97].

3.1 Elements of SHE

SHE contains a framework, various forms of representations heuristics and a formal language. In addition specification analysis and design of complex system requires experience skills and automated tools. Figure 3-3 visualises elements of the SHE-method.

![Figure 3-1 Elements of SHE](image)

**Framework**
The framework gives guidance to the specification and modelling process. It offers an imaginary work space where various modelling activities take place.

**Informal Representations**
Various informal representations are used:

- Graphical representations
  - message flow diagram
  - instance structure diagram
  - object class diagrams
  - interaction diagram
- Textual representations
  - initial requirement description
  - architecture decision statement
Functional behaviour and structure of a system are formalised by a description in the formal language POOSL (‘Parallel Object Oriented Specification Language’). POOSL is a parallel object oriented specification language designed for specification, design and description of systems that contains a mixture of software and hardware. It is based on the object oriented design paradigm to support flexible and reusable design. Formal verification, simulation and transformations of specifications are supported.

The language enables explicit representations of system architecture and hierarchy. Objects communicate through channels using a one-way synchronous message passing mechanism, allowing true parallelism. POOSL distinguishes ‘process objects’ that are statically interconnected from ‘data objects’ that dynamically move. Tail recursion is supported.

### Heuristics
Heuristics are acquired practising the SHE-method on several cases, like specification of a mailing machine, an elevator and an ATM-switch.

### Experience, skills
Experience and skills are acquired by putting the SHE-method in practice.

### Tools
Several tools are used and developed:
- the drawing tool VISIO, for which templates have been developed, is used for drawing the graphical representations.
- a simulator for the formal language POOSL.
- automatic conversion from the formal language POOSL to C++.

### 3.2 SHE context and Phases
Figure 3-2 shows phases of the SHE method in a system specification and design project, marked by milestones in between. SHE consists of two phases Essential Specification and Extended Specification separated by a milestone.

#### 3.2.1 Initial Requirement Description
The starting point for the creation of an essential specification is a completed Initial Requirements Description (IRD). The IRD consists of three descriptions:
- requirement list
- purpose description
- prescribed and predefined implementation and topology choices (design constraints)

### Requirement List
The requirement list contains a mix of functional requirements, environmental conditions and other constraints.

### Purpose Description
The purpose description makes needs of the users to the system explicit.

### Prescribed Technologies and Topologies
The prescribed technologies and topologies description define pre-defined implementation choices. They restrict design space and influence structure.
3.3 Essential specification

Essential specification analyses essential system requirements and creates both an abstract behaviour model and the highest level system architecture. The result of essential specification is a correct abstract system model. Essential specification starts with exploring a number of conceptual solutions.

**Conceptual Solution**

A conceptual solution is a feasible idea that matches systems requirement description and gives an initial structure to the model. For each conceptual solution a preliminary model is created using the Framework.

**Scenario**

Behaviour of a system is too complex to describe as a whole. A scenario describes a part of the complete behaviour using the presentations of the framework. Scenarios are multi-disciplinary, for each discipline involved a scenario can be created that concerns only that discipline.

**Frame-Work**

Figure 3-3 shows an essential specification framework. It incorporates graphical re-presentations that visualise architecture and behaviour of collaborating objects and textual descriptions that formalises the specification.
3.3.2 Architecture Structure Diagram

An Architecture Structure Diagram (ASD) is a free style graphical representation of the system structure. It makes the system topology visible and visualises independent modules and their communication channels.

3.3.3 Message Flow Diagram

A message flow diagram (MFD) visualises message flows between collaborating instances of classes (objects and clusters). Figure 3-4 shows elements of a message flow diagram.

![Figure 3-4 Elements of Message Flow Diagram](image)

Clusters / boundaries

A cluster is an abstraction boundary around a group of collaborating process objects and clusters. Different specialisations of boundaries exist:

- **Abstraction boundary**, is an enclosure with a hiding property, inward visibility is limited.
- **Concurrency boundary**, is an enclosure that prescribes internal sequential behaviour.
- **Distribution boundary**, decomposes a (sub)system into physically separated parts.
- **Implementation boundary**, is an enclosure that establishes implementation technology used for realisation.
- **System boundary**, encloses system to be specified.

Terminator

A terminator symbol represent an external object outside the system boundary.

Process Object

A process object represents independent system parts connected in a static structure. Process object are a part of a static communication structure that fits in the architecture. Process objects communicate with each other solely via message on channels.

Data Object

A data object contain data and has the ability to act on it. Data can only be read and changed by the object itself. Data objects can be passed between process objects.

Message Flows

A message flow symbol represents a message that can be passed between process objects. Contents of a message (data objects) are placed after the message name, between brackets and separated by commas. The contents are passed from the sender object to the receiver object.

![Figure 3-5 Message Flow Symbols](image)

Single Message Flow

A single message flow symbol represents synchronous communication without reply. After communication the sending and receiving object can continue their activities.
Message Flow with Reply
A double message flow symbol represents synchronous communication with reply. The activities of the sending object are suspended until a reply message from the receiving object is returned.

Continuous Message Flow
The continuous message flow symbol represents a time-continuous flow. A sending object is always prepared to deliver a message immediately. A receiving object initiates the communication.

Interrupt Message Flow
The interrupt message flow symbol represents asynchronous communication. A sending object forces a receiving object to suspend its current activity. The receiving object is always prepared to receive the message.

Interrupt Message Flow with Reply
The interrupt message flow with reply is an interrupt message flow where in addition the sender suspends its activities until a reply message from the receiver is returned.

Buffered Message Flow
A buffered message flow symbol represents asynchronous message passing. A sender can always send to a buffer. A receiver can always read available messages from the buffer.

3.3.4 Instance Structure Diagram
An Instance Structure Diagram visualises channels between collaborating instances of classes. It is a static structure restricted by the architecture structure diagram.

3.3.5 Interaction Diagrams
An interaction diagram visualises the order of messages between collaborating objects.

3.3.6 Object Class Diagram
An object class diagram (OCD) shows how different classes relate to each other and the composure of classes. In the OCD a class is represented by a box, in which three regions exist. The upper field contains class sort and name. The class sort is either ‘P’ which denotes a process object or ‘D’ which denotes a data object. An abstract class is indicated by an ‘A’ before the class sort. An abstract class is a class that has no direct instances and holds features common to its sub-classes. The middle field contains attributes which is a set of instance variables. The message field contains a set of messages. A relation between two classes is depicted by a line between the two classes and a symbol on the line that visualises the kind of relation. A symbol on the line indicates the relation between the classes with the following meaning:

- specialisation relation (‘a kind of’), a sub class is a specialisation of its super class, ‘is a’ relation
- generalisation relation (‘a part of’), a sub class is a part of its super class
- zero or one instances relate to an instance of an associated class
- zero or more instances relate to an instance of an associated class
- link attribute, is a property of the links din the relation

3.3.7 Unified Model
Behaviour and structure are formalised in the formal language POOSL.

3.3.8 Requirements catalogue
Items that are specified but can not be formalised are described in a requirements catalogue.

Listed Requirements
Extension of the initial requirement description by non essential requirements.

System Dictionary
A list of all objects from the various models with a brief description of their meaning.
**Chapter 3.**

**Architecture Decision Statement**
Justifies architecture decisions of architecture design.

**Implementation Decision Statement**
Justifies design decisions made for the design of implementation structure.

### 3.4 Extended specification Modelling

In extended specification modelling phase restrictions laid up by implementation technologies are incorporated in the essential specification. Extended specification is build on a framework similar to essential specification (Figure 3-6).

![Diagram](image)

**Figure 3-6 SHE extended modelling framework**

In extended specification communication protocols can be incorporated. Channels between collaborating objects implemented in software, can be dissolved by passing of references.

### 3.5 Implementation

In implementation phase the specification is implemented on a system. Hardware and software are designed for the implementation technologies used.
Chapter 4

Essential Specification

The essential specification paragraph specifies essential system behaviour and formalises behaviour in a formal language. Essential specification is based on an initial requirement description. Several conceptual solutions are explored, only the final satisfying conceptual solution, is discussed in this thesis. Explored conceptual solutions, focused particularly on: the internal flow of mailing packets representations in functions and transporters, complex side production lines, a-synchronous couplings and tracking position of mailing packets on transport to activate functions.

In paragraph 4.1 mailing machine components that participate in controlling the mailing machine and mailing machine structure are identified. Paragraph 4.2 describes the conceptual solution on which modelling is based. In paragraph 4.3 current architecture is examined using Architecture Structure Diagrams. With bottle-necks of current architecture in mind and requirements described in the Initial Requirement Description, improved architectures are explored. Paragraph 4.4 identifies different views on required behaviour of the mailing machine using scenarios. The behaviour of the mailing machine at system level is specified according to the identified scenarios in paragraph 4.5, using message flow diagrams. Paragraph 4.6, 4.7 and 4.8 specify the mailing machine at sub-system level using message flow diagrams. Completed message flow diagram, Interaction diagrams, Object Diagrams are described in appendix A,B,C and D for each identified model. The formalisation in the formal specification language POOSL is presented in appendix E.

4.1 System Components Analysis

To specify a mailing machine all components participating in control must be identified. Figure 4-1 visualises a mailing machine with named, units a mailing machine is built from. Figure 4-2 shows an abstract view on mailing machine visualised in an object class diagram.

A mailing machine consists of a system info handler and a main production line that has zero or more side production lines. A side production line can have zero or more side production lines it self. The main production line and the side production line are specialisations of a production line. A production line is built from one or more transporter sections. A transporter section is built from mechanical stations and a transporter. A mechanical station is a substructure that consists of zero or more functions. Transporters transport mailing items along functions mounted on the substructure. Functions are devices that participate in the creation of mailing packets, like feeders and label printers. A function images transport of the transporter the function is mounted, to acquire information about position and identity of mailing packets on transport. Both transporter and functions consist of control modules. Control modules are devices used for control purposes, like control buttons and touch-screens. Transporters, functions and control modules consist of sensors and actuators.
4.1.1 System Info Handler

The system info handler controls mailing packet creation. The system info handler is controlled by an operator. It takes care of:

- preparation of composure mailing packets to be created, create routing and required functions information
- handling failure situations
- processing information of editors
- booting of complete mailing machine
- configuration of mailing machine
- collecting diagnostic information
4.1. System Components Analysis

- creating mailing machine topology layout

In current implementation control of transporters, functions and control modules are implemented in the system info handler.

4.1.2 Production Line

A production line transports mailing packets along functions. Two types of production lines are distinguished, a main production line and a side production line which can be equally complex. A production line is built from one or more transporter sections.

Main production line

A main production line is the production line all mailing packets end up eventually. A main production line can have side production lines.

Side production line

A side production lines intersects with a production line. At the tail of a side production line mailing packets combine on the production line it intersects with. A side production line can have side production lines as well.

4.1.3 Transporter Section

A transporter section is built from a transporter and one or more mechanical stations. Transporters are consecutively coupled in two distinct ways:

- synchronously, movement of two sections is synchronised, the speed of both sections is relative to each other. The transporter sections are synchronised by a sub-control system.
- a-synchronously, movement of transporter sections is independent of each other.

A-synchronous coupling

In case of a-synchronous coupling a photo-transistor/LED pair is mounted on the substructure (Figure 4-3) to detect the physical arrival of a mailing packet in the transporter. The photo-transistor detects the interruption of a light beam, by a mailing packet. To arrive at a desired position on the receiving transport, the mailing packets are accelerated or decelerated by a sub-control system.

4.1.4 Transporter

A transporter physically transports mailing packets on a transporter medium. Figure 4-4 visualises specialisations of transporter mediums used. In the thesis the transporter medium is referred as transport. The transporter is equipped with a position encoder that tracks movement of transport. Some specialisations of transport are equipped with a cam, against which a mailing packet rests. The cam is forces the mailing packet to move forward with transport.

![Diagram of transporter section](image)
4.1.4.1 Position encoder

A transporter is equipped with a position encoder that monitors movement with a resolution of one unit distance. One unit distance movement equals movement of transport by one chain of the conveyor belt. The encoder generates a revolution pulse and a quadrature squared wave. The quadrature squared wave is composed of two squared waves named A and B having a duty cycle of 50%. The squared waves have a cycle time of four units distance movement of transport and are phase-shifted with respect to each other by 90 degrees (one unit distance movement, see Figure 4-5). Both waves have two transitions (edges) per revolution, the composed quadrature square wave contains 4 edges per cycle. Each edge indicates movement of transport by one unit distance. The direction of movement is determined by analysing the order of the edges.

![Figure 4-5 Quadrature square wave of position encoder](image)

The encoder generates a revolution pulse, named Z-pulse, the pulse indicates revolution of the encoder by one cycle. The cycle time is a fixed number of unit distance of transport movement. The Z-pulse does not occur at the same moment as a transition. The encoder is mechanically coupled, it runs not out of phase with transport. The Z-pulse is used as new cycle indicator. During each cycle a cam comes available against which a new mailing packet can be placed.

**Modelling the encoder**

The position encoder is modelled as part of the transporter in a terminator named Transporter. Figure 4-6 shows models of the Transporter used for essential and extended modelling. For essential modelling an abstract model is used, it only shows key points of behaviour. For discussions on behaviour of the encoder only direction and movement of transport are of interest.

![Figure 4-6 Modelling the encoder](image)

For extended modelling the edges of the A- and B wave and the Z-pulse are modelled. The edges of the A- and B wave are modelled by the messages `aPositiveEdge`, `aNegativeEdge`, `bPositiveEdge`, `bNegativeEdge`. The Z-pulse is modelled by a `newCycle` message. The `newCycle` message indicates the start of a new cycle.
4.1. System Components Analysis

Key points in essential behaviour of the Transporter are movement of transporter by one unit distance and direction of movement. Instead of modelling the different transitions that represent movement and direction of transport by one unit distance, direction and movement are modelled separately in essential specification. Movement is modelled by sending a \textit{fineShiftPulse} message on each transition of both the a and b squared wave. The \textit{fineShiftPulse} message indicates movement of transport by one unit distance in the known direction. The direction of the transporter is set by a \textit{forwardDirection} and \textit{backwardDirection} message. The \textit{newCycle} message indicates a new cycle.

**Backward movement**
The transporter can move back a number of unit distance, as result of a uncontrolled stop, when transport is touched or is unstable.

**4.1.4.2 External transporter control**
Transporters and functions can be equipped by one or more of these buttons to control transport:

- **Emergency stop**
  Pressing the emergency stop button, immediately stops transport without controlled decelerate. Mailing packets are completely disordered by the abrupt stop and must be removed from transport. Synchronisation between the transporter sections is lost.

- **Reset**
  In current implementation pressing the reset button, notifies the system info handler that all mailing packets have been removed from transport. In new specification the meaning is the correction of the problem as described by the system info handler.

- **Start**
  Pressing start the first time accelerates the mailing machine to a pre-set low speed (300 mailing packets per hour). Pressing start again accelerates transport to a pre-set final speed.

- **Stop**
  Pressing the stop button, stops transport with a controlled decelerate. All mailing packets stay in position.

**4.1.5 Station**
A station is a substructure that mounts zero or more functions. A station is a part of a production line.

**4.1.6 Function**
Functions are devices that contribute in the creation of mailing packets, like feeding mailing packets or placing a label. Whether a function is required or not required depends on the personalised requirements of the mailing packet it acts on.

Functions require information on position and identity of mailing packets on the transporter. A function only acts on a mailing packet when its action is required by the particular mailing packet (personalised mail).

![Figure 4-7 Specialisations of functions](image-url)
Figure 4-7 shows specialisations of functions, discussed in this paragraph.

4.1.7 Feeders
A feeder places an item from a loader (storage of mailing item, to be fed) on transport or on a mailing packet on transport. Many specialisations of feeders exist that all have different features with their particular advances. A feeder transports mailing items from a loader to transport.

4.1.7.1 Feeder Features
Specialisations of feeders have many aspects in common\(^4\). These aspects should be categorised to specify a generic feeder. Components and behaviour incorporated by specialisations of feeders:

- **light**
  - *fail blink*, when a feeder fails on one mailing packet in succession. The light only blinks once.
  - *error blink*, a number of actions on a mailing packet failed in succession or a serious error occurred. The light blinks slowly.
  - *configuration blink*, the function is configured. The light blinks fast.
  
  Priority order: config (highest), error, fail.

- **feed fail detection**
  - *mis feed*, zero items are fed.
  - *double feed*, two or more items are fed.

- **enable / disable transport of mailing item**
  - *vacuum valve*, a drum with a vacuum valve picks a mailing item up by creating a vacuum that glues a mailing item to the drum.

- **mailing items to be fed are stored in a loader**
  - *minimum level detection*, detects a minimum level of mailing items in the loader.
  - *empty detection*, detects the absence of mailing packets in the magazine.

- **double feeder**, two independent feed sections on one feeder both fully equipped with feed fail detection, etc.. The angle between the two sections is set mechanically. The difference between the sections and position from the start of the transporter result in different activating moments.

- **orientation on transport**
  - *in-line*, mailing items are fed in the direction of mailing packet flow.
  - *off-line*, mailing items are fed perpendicular to mailing packet flow.

- **half feeding speed**, a feeder can set to work at half speed that is feed one item during two cycles.

In the thesis a rotary feeder is specified. Design of a generic feeder model is kept in mind but hard when not all feeders are analysed.

4.1.7.2 Rotary feeder
To specify a rotary feeder all components that contribute to the behaviour in a control point of view are identified. Encountered components in a rotary feeder:

1. *vacuum valve*, sucks an item to a drum, in order to transport an item from the loader to transport.
2. *light*, gives a visual indication of the state of the feeder.
3. *loader*, stores a number of items ready to be fed.
4. *double feed detector*, detects when the vacuum valve sucks more than one item.
5. *mis feed detector*, notices the presence of a mailing item during a feed.

\(^4\) Other functions also have features in common with the feeders, like light and configuration
Modelling components

Figure 4-8 shows rotary feeder components as modelled.

Discussion on behaviour of rotary feeder components:

**Loader**
A Loader stores a number of mailing items, ready to be fed. A sensor detects a minimum level of mailing items available in the Loader. The transitions from a minimum available to less than a number of mailing packets available and vice versa are of interest. Transitions are modelled by a `productSensed` and a `productNotSensed` message. The `productSensed` message is send when mailing items are added in the magazine and the level reaches at least minimum level. The `productNotSensed` is send when the number of available items becomes less than the minimum level.

**Vacuum Valve**
A Vacuum Valve transports mailing items from the Loader to transport. (Figure 4-9) visualises a vacuum valve. The Vacuum Valve is part of a drum that is mechanically coupled with the transporter. To feed a mailing item, the Vacuum Valve is enabled at 180 degrees for a complete cycle. In case the next mailing packet does not require a mailing item the Vacuum Valve is disabled after a complete cycle. The Vacuum Valve is enabled by sending a message `vacuumOn` and disabled by sending a `vacuumOff` message.

**Light**
The Light can be set on and off. The light blinks as described above in point 1 of the feeder features. The message `lightOn` turns the Light on, the message `lightOff` turns the Light off.

**Feed fail detection**
Two kinds of feed fails are detected, mis feeds and double feeds. The absence of a position encoder on the feeder, makes it impossible to determine on which mailing packet the feeder acts on around 180 degrees. Around 180 degrees feed fail must be disabled. In current implementation feed fail is enabled between 90 degrees and 270 degrees. Feed fail detectors:
- **Doublefeed Detector**, a sensor detects when the Vacuum Valve feeds more than one mailing item. On detection of more than one mailing packet, a message `doubleFeed` is send.
- **Misfeed Detector**, a sensor detects if the Vacuum Valve did suck a mailing item. On detection of an item a message `productSensed` is generated which indicates the presence of an item. For extended specification the model must be by a `productNotSensed` message, that is generated when the mailing item passed the sensor.

**4.1.7.3 Alternately feeding**
Speed of transport can be doubled by using two independent feeders that feed alternately on half transport speed. Both feeders have a feed cycle of two cycles of transport movement instead of one cycle.

**4.1.7.4 Current implementation of control**
To provide a notion of the feeding process, current implementation is discussed. Figure 4-9 visualises the behaviour of currently implemented control.
Figure 4-9 Controlling vacuum valve and feed fail detection

<table>
<thead>
<tr>
<th>Angle</th>
<th>Action Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>180°</td>
<td>Enable vacuum valve</td>
</tr>
<tr>
<td>270°</td>
<td>Enable feed fail detection, double feed register is cleared, item detected register is cleared</td>
</tr>
<tr>
<td>90°</td>
<td>Disable feed fail detection, on a set double feed register a double feed occurred, on a cleared item detected register a mis feed occurred</td>
</tr>
<tr>
<td>180°</td>
<td>If next mailing packet does not require an item to be fed then disable the vacuum valve</td>
</tr>
</tbody>
</table>

### 4.1.8 Ejector
When a mailing packet is not composed as required, an ejector is able to remove mailing packets from transport.

### 4.1.9 Reader
Readers, like an address head reader and a card reader, read information from pre-printed cards or labels that contain address information and / or information used for personalised mailing purposes.

### 4.1.10 Sealer
A sealer consist of a packer and a separator. The packer winds foil around mailing packets, the separator separates mailing packets by cutting the foil.

### 4.1.11 Stacker
A Stacker stacks and bundles a number of mailing packets. Stacked backs of mailing packet items cause slant bundles. To prevent this, bundles are turned half a cycle to create stable bundles.

### 4.1.12 Town Separator
A town separator controls bundling mailing packets destined for the same postal region. It controls the readers, the ejector and the stacker. The bundles a town separator creates, consist of a pre-set maximum number of mailing packets. On top of that maximum a number of mailing packets may be added that would create a very small bundle. The town separator can be set to eject bundles that would have less then a pre-set number of mailing items. After a number of mailing packets are stacked, bundles are turned half a cycle. The last two succeeding bundles within one postal region have about the same number of mailing packets. On completion of a postal region mailing packet creation must be suspended to provide the stacker with time to carry of a bundle. When a number of small bundles are handled successively mailing machine speed must be reduced to provide the stacker with enough time to handle the mailing bundles.

### 4.1.13 Fly-over
A fly-over temporarily stores mailing packets while other mailing packets pass. The order of mailing packets is changed by a fly over. The fly-over can store failed mailing packets or mailing packets that could not be bundled in their postal region, on release a separate bundle could be created.
4.2 Conceptual Solution

This paragraph presents the final conceptual solution that provides the basis to modelling. Key problems are modelling transport and creating mailing packets. The solution presents a generic transporter model that models transport. All transporter layouts with their particular difficulties are handled by the model. Functions are modelled by a generic function model that interfaces the generic model of transport.

A mailing packet is created by required functions that act on a mailing packet in creation. When a mailing packet arrives at a position on a transporter a function must start acting on the mailing packet, that function must be activated. The function is only activated when it is required by that mailing packet. The mailing packet must pass all required functions to create a correct mailing packet. To create a mailing packet, mailing packet requirements and position of the mailing packet are required by control. Required functions and route of a mailing packet are stored in a data object product information (Pi). Each Pi is associated with a product information identification (Pild). The Pild is associated with the position of the mailing packet in the generic transporter. The Pild is passed to another generic transporter model, when:

- the associated mailing packet moves on to a consecutive transporter.
- a side production line must start the creation of a mailing packet.
- when a function starts acting on a mailing packet as will be described next.

Key problems:

- activating (complex) side production lines, a side production line starts creation of a mailing packet when a correct combination is assured on the production line it intersects.
- activating functions, a function starts it action when a mailing packet arrives at a certain position on the transporter.

Design space:

- a-synchronous coupling limits prediction of arrival moment of a mailing packet in a transporter.
- use of a field bus, allows communication between:
  - transporters not connected to each other.
  - a function and transporters the function is not mounted on.
- all functions and transporters are equipped with a controller board.

To activate transporters and functions, the generic transporter model must track the position of a mailing packet on the transporter.

Modelling transport

A transporter is modelled by a generic transporter model. The transporter model must handle both synchronous coupling and a-synchronous coupling. In case of a-synchronous coupling it is not possible to predict the arrival of a mailing packet in the transporter. When a mailing packet moves on to a consecutive transporter, the Pild associated with the mailing packet is passed to the generic transporter model of the consecutive transporter. Figure 4-10 visualises modelling of transporters.

![Figure 4-10 Modelling transporters](image)

Activating side production lines

At the intersection between a side production line and a production line two mailing packets combine into one. To realise a correct match, the mailing packets must arrive at the intersection at the same moment and belong to each other (have the same Pild). From a control point of view, the side production line can be placed parallel to the production line. Positions on both production lines that take an equal 'time' to reach the intersection can be mapped on each other. When a mailing packet reaches a position in the production line mapped on the start of the side production line, the side production line must start the creation of the mailing packet. The production line notifies the side production line that it must start the creation of a mailing packet, by sending the Pild of the mailing packet to the side production line. When the created
mailing packet reaches the end of the side production line, the representation is passed back to check for a correct match. Figure 4-11 visualises modelling of a side production line.

**Activating functions**

A function is activated when a mailing packet arrives at a certain position on the transporter. Functions are equipped with their own local control and are not aware of the mailing packets on the transporter. A function must image the region of the transporter it acts on in its local control. The imaged transporter is modelled by a generic transporter model. Functionality of the function will be modelled in a generic function model that interfaces the generic transporter model. When a mailing packet arrives in the region on the transporter the function images, the function is notified by the pass of the *Pild* of the mailing packet. The function is triggered by its imaged transport.

The imaged transporter behaves like a side production line. Both the function and side production line are activated when a mailing packet reaches a certain position on the transporter. The difference is that a side production line passes the representation back to check for a correct match. For genericity of the transporter model, a function passes the representation back to check for a correct match. Figure 4-12 visualises modelling of a function.

Functionality of the function is modelled in a generic function model that interfaces the transporter model that models the transporter the function images. The function acts on a mailing packet when the mailing packet reaches a certain position in the imaged transporter and only if it is required by the mailing packet. This generic behaviour is provided by the generic transporter model. It will trigger a function when a mailing packet arrives at a trigger position only when the function is required.

**Control modules**

Control modules are modelled by the generic function model. The generic function model that models the control module, interfaces with the generic transporter model that models the transporter it is a part of.
4.3 Mailing machine architecture

Modelling mailing machine
In case a function or side production line must be activated before the start of the transporter they respective
are mounted on or intersect, an other transporter must activate them. This transporter must be synchronously
coupled to the transporter that should activate it. In case of an a-synchronous coupling this is not possible,
the physical arrival of a mailing packet in the transporter that should activate it can not be predicted. An
other option is to start transport modelled by the transporter model before the physical start of the
transporter. Figure 4-13 visualises a mailing machine modelled.

4.3 Mailing machine architecture
The design space of the control system that is developed is restricted by the architecture of the system. An
Architecture Structure Diagram (ASD) makes the topology of the mailing machine visible. It is an abstract
view of the mailing machine that shows all isolated modules and communication channels between the
modules. It is a static structure, it is not possible to introduce new modules or communication channels
dynamically during design. Communication between the modules is limited by the defined communication
channels. All other views are limited by restrictions laid up by the ASD. The ASD shows aspects that
concern architecture, like distribution of data and control, separation of configuration and control,
introduction of a field-bus, independent modules, local control, version management. Design decisions in
the ASD are briefly justified in an Architecture Decision Statement (ADS). Alternative solutions and
properties considered for making an 'optimal' design are documented as well.

In paragraph 0 restrictions of current architecture are discussed. With discovered problems in mind a new
improved architecture, distributed with remote I/O, is developed in paragraph 4.3.2. An architecture with
fully distributed control is discussed in paragraph 4.3.3.

4.3.1 Inventory of Current System Architecture
It is hard to extend the existing mailing machine control system with new features due to current
architecture. In this paragraph problems that arise due to architecture are identified and kept in mind to
develop an improved mailing machine architecture. Figure 4-14 shows an Architecture Structure Diagram of
the mailing machine as it is build half a year ago, at present cables are replaced by CAN. Current mailing
machines are controlled by a Programmable Logic Controller. All sensors and actuators are connected to the
PLC by a single cable. The PLC processes sequential software that continuously makes a loop to respond to
state changes. Mailing packet positions and requirements are available in a shift-register.

Figure 4-13 Mailing machine modelled by generic transporter and function models
Figure 4-14 Architecture Structure Diagram of the current implementation

Figure 4-15 visualises the shift register. Transport is mapped on the shift-register with the same resolution as transport movement is measured. The shift-register contains information that triggers functions, p.e. information to open a vacuum valve or to eject a mailing packet. Functions peek in the shift-register, on a transition an action is initiated. When personalised mailing packets are composed the trigger information is based on information of the database. The database gets its data from a reader or from an editor. One PLC continuously processes all input during a large software loop. The width of the shift-register is limited.

Figure 4-15 Triggering functions by shift register

Cables
To reduce installation time the amount of cables must be reduced. A field bus decreases the number of cables drastically.

Hard Configuration
As a result of hard configuration (configuration coded in control software code), configuration and control software are inseparable. For each mailing machine unique software that includes configuration for that particular machine, is created. Even for each different layout of a mailing machine that is used, different control software with configuration for that particular layout must be created. All these unique versions of control software complicate version management. To update control software all configuration must be transferred to a new release, which makes control software hard to update. These problems can be solved by separating configuration from control software.

One large module
A function is only separate testable if the control software that belongs to the function can be transported to the test environment. In the current architecture, control software of a function is a part of the complete mailing machine control software. The control software of a function can not be extracted since it depends on control software of the complete machine. Partitioning control per function and transporters in independent modules overcomes this problem.
4.3. Mailing machine architecture

**Shift-register**
It is hard to implement a mailing machine with an a-synchronous complex side production lines using a single shift register. A side production line should have its own shift register that co-operates with the shift register of the main production line. A more flexible mechanism must be explored that tracks mailing packets and triggers functions. The restricted width of the shift-register limits number of functions that act personalised.

**Sequential Processing**
In some large mailing machines the PLC can't deal with the amount of data and control it has to process. Introduction of local control in the transporters and functions would relieve a main controller. The transporters and functions handle actions on mailing packets locally. A more power-full main control only is a temporary solution, new features and bigger mailing machines will exhaust additional processing power in time.

**Conclusion**
To overcome problems due to current architecture, the following features must be introduced in next generation mailing machines architecture:
- a field bus
- configuration separated from control software
- modelling control of a transporter and function as one independent modules
- flexible mechanism to control transport and track mailing packets
- distributed control

**4.3.2 Distributed Control Prepared System Architecture with remote I/O**

**Architecture Structure Diagram**
The first architecture developed is a distributed control prepared architecture with remote I/O (Figure 4-16). This architecture can be implemented on an implementation with central control using remote i/o. Transporters and functions are equipped with a controller board each that contains a micro controller featuring CAN. In this architecture the controller boards only handle I/O of the transporter and functions. The solutions to the problems with current architecture (discussed in previous paragraph) is taken care of. Independent modules are introduced, configuration and control are separated, a field-bus is introduced, control is prepared for distribution and the shift-register is replaced by a new mechanism.

![Architecture Structure Diagram of Distributed Control Prepared Implementation](image)

**Architecture Decision Statement**

**Control Area Network**
A broad cast field-bus is used to reduce the number of cables between the system info handler and transporters and functions. A Control Area Network will be used.
Alternatives of a broadcast field-bus is an architecture that connects all transporters and functions to the system info handler. Transporters and functions only communicate with each other via the system info handler. Disadvantages are: routing required and restricts plug and play.

**Separation of Configuration and control software**
Separation of configuration and control software makes it possible to create control software that is independent of configuration. As result it is not necessary to create new control software for each mailing machine. Only configuration differs for each mailing machine. The Software of a mailing machine is easily updated since it uses the existing configuration. Version management is greatly improved.

**Independent modules**
All distinguishable parts of control are placed in an independent module. The modules work independently and concurrent of each other. Stand alone testing and preparing is possible, although control software and its configuration must be extracted from mailing machine control to the test environment. Physical distributed control will solve this problem.

**Distributed control prepared**
Introduction of independent modules makes the architecture prepared for topological distribution of control. Independent modules make it possible to move modules within the ASD to local controllers in transporters and functions using behaviour preserving transformations. Behaviour preserving transformations assure that the behaviour of the mailing machine does not change.

**Passing mailing packets**
The shift register is abandoned. Mailing packets are represented internally by data objects (product information) having mailing packet properties. When a mailing packet passes to a consecutive transporter, a data object \((P_i/d)\) associated to the \(P_i\) of the mailing packet is passed between transporters. Internally a position on the transporter is associated to the \(P_i/d\). A Function starts its action on a mailing packet when its representation reaches a certain position in the station (as discussed in the conceptual solution).

### 4.3.3 Fully Distributed Control

**Architecture Structure Diagram**

In the fully distributed control architecture, control of the transporter and functions moved from the mailing machine controller to the micro controllers available on controller boards.

---

\[5\] if control knows the layout of the mailing machine it is able to create layout configuration automatically.
4.3. Mailing machine architecture

Architecture Decision Statement

Distributed Database
A distributed database is necessary, calculations show that the capacity of CAN is not sufficient when each transporter and function in a huge mailing machine has to retrieve / store for each mailing packet information from / to a centralised database. The distributed database contains information for each mailing packet present in the station.

Physically distributed control
The independent control modules of the transporters and functions are placed on the local controller boards available in the stations and functions. Stand alone testing and preparation is performed by placing a complete module in a suitable test environment. Physically distributed control is a step towards future mailing machines with stand alone functions.

4.3.4 Meta Model of mailing machine

Analysing the fully distributed control architecture (paragraph 4.3.3) three kind of modules are distinguished: the system info handler, transporters and functions. In contrast to transporters and functions, the system info handler does not handle mailing packets physically. The system info handler is modelled in a separate model, named System_Info_Handler. It is a task of TNO to specify and model the system info handler.

Both transporters and functions handle mailing packets physically. Properties required to handle mailing packets:
- track exact position of mailing packet on transport.
- identity of the mailing packet on transport.

A generic transporter model is developed that incorporates these properties to model transporters. Functions must image transport to acquire information on position and identity of mailing packets. The imaged transport of a functions is also modelled by a generic transporter model. Additional functionality of transporter and function is modelled in a generic function model that interfaces the generic transporter model.

System_Info_Handler
The System_Info_Handler models tasks described in paragraph 4.1.1.

Generic transporter model
The generic transporter model models transport of mailing packets / items:
- track exact position of mailing packets on transport
- identity of mailing packet on transport
- complex side production lines
- synchronous / a-synchronous coupling
- handles information about mailing packets (personalised mail)

Generic function model
The generic function is an interface to the generic transporter model. The generic function model models:
- control modules
- functionality of a function, p.e. feeding, labelling
- additional functionality to the transporter or function, p.e. diagnostic tools

Overview
Figure 4-18 shows a meta model of the mailing machine completed with the discussed models used to model the mailing machine. The meta model shows the relation between the different models and hardware parts of the mailing machine.

The system info handler is modelled by the System_Info_Handler. The Transporter is modelled by the generic transporter model. A function is modelled by a generic function that interfaces the generic transporter model that models the transporter it images. Control modules are modelled by the generic function model. The generic function model interfaces the generic transporter model the control module is a part of.
4.4 Behaviour Definition using Scenarios

Behaviour of a mailing machine is too complex to describe as a whole. A scenario describes a part of the complete behaviour, it is a textual description of distinguishable behaviour. P.e. behaviour during a job of a mailing machine is described in a normal operation scenario. Scenario’s are visualised in a Message Flow Diagram (MFD). Within a scenario more detailed sub-scenarios are distinguished. The normal operation scenario of the generic transporter describes passing of mailing packets and triggering functions in sub-scenarios. Scenarios are multi-disciplinary, for each discipline involved a scenario can be created that concerns only that discipline. The required behaviour can be described and discussed with people of a certain discipline, without being bothered by details of other scenarios. A maintenance and repair scenario describes the behaviour understandable to maintenance people and a management information scenarios describes how and what information is gathered for people that concern mailing machine performance.

This paragraph discusses scenarios identified during planar sessions. The meaning for transporters and functions is discussed separately for each scenario discussed in the thesis. The meaning of the scenarios for the System_Info_Handler is discussed in the machine model paragraph of Frank Verhofstad.

4.4.1 Identified Scenarios

Meaning of identified scenarios:
1. booting scenario, brings the system in a defined state on power-up.
2. job preparation scenario, prepares mailing machine for a job.
3. normal operation scenario, describes creation of mailing packets.
4. flushing scenario, flushes the mailing machine from all mailing packets.
5. maintenance, test and repair scenario, defines behaviour to support maintenance, test and repair of the mailing machine.
6. management information scenario, gathers information on mailing machine and operator performance.
7. reordering scenario, reorders the order of mailing packets to be created on a failure.
8. product information handling scenario, describes distribution of product information through mailing machine.
9. power down scenario, temporarily stops mailing machine.
10. failure effect scenario, evaluates independent failures of different scenarios for correlation.
11. Configuration scenario, describes configuration during booting, job preparation and normal operation.
4.4 Behaviour Definition using Scenarios

All scenarios incorporate failure handling for errors elected by that scenario. Actions to be taken on a failure are best specified within the context of that particular scenario.

4.4.2 Booting Scenario
The booting scenario brings the system in a defined state with or without mailing packets in the system. The scenario is active after power up of the system.

**System_Info_Handler**
- check type of power-down
- create system availability image
- build system layout configuration (topology)
- check for mailing packets in mailing machine
- check if job was interrupted
  - if job is interrupted take care of mailing packets in mailing machine
- reach a defined state

**Generic transporter**
- reach a defined state
- provide system info handler with position in mailing machine topology (plug and play)

**Function**
- self test (check sensors, actuators and available options)
- identify itself to the generic transporter
- reach a defined state or starting point

**Operator**
- handle problems as result of type of power down

4.4.3 Job Preparation Scenario
The job preparation scenario prepares the mailing machine for the first or the next job. The mailing machine is configured with job specific configuration.

**System_Info_Handler**
- set-up town separator

**Generic transporter**
- configure job specific settings, only accept relevant messages
- initialise for new job

**Function**
- inform System_Info_Handler about actual settings and options
- configure job specific settings

**Operator**
- prepare mailing machine mechanically
- fill loaders

4.4.4 Normal Operation Scenario
The normal operation scenario describes the main function of the system, creating mailing packets.

**System_Info_Handler**
- initiate mailing packet creation
- feed system with product information
- accept product information of finished mailing packets

**Generic transporter**
- pass representations of mailing packets
- track position of mailing packet
- identify mailing packets
Chapter 4.

- activate required functions

**Function**
- act on mailing packet

**Operator**
- check mailing packet quality visually
- re-fill loader on minimum level

### 4.4.5 Flushing Scenario

The flushing scenario describes the removal of mailing packets from the mailing machine when a job is finished or interrupted. A job can be interrupted by an emergency stop, power-down etc.

**System_Info_Handler**
- depending on kind of interruption:
  - run transport empty (job with empty mailing packets)
  - let operator remove items
    - from transporter
    - from loader

**Generic transporter**
- update internal image of transport when mailing packets are removed by hand.

**Function**
- cancel actions on mailing packets, no feeding, printing labels etc.

**Operator**
- remove mailing packets manually from transport, when instructed by system info handler.
- remove mailing packets from loader, when instructed by system info handler.

### 4.4.6 Product Information Handling Scenario

Distribute and update information on requirements of mailing packets through the mailing machine.

**System_Info_Handler**
- prepare product information records
- pass product information records to generic transporter
- update product information records in transporter

**Generic transporter**
- accept and store product information records in database
- update product information records

**Function**
- update product information records

### 4.4.7 Configuration scenario

Describes configuration during booting scenario, normal operation scenario and configuration scenario. Configuration means for these scenarios:

- booting scenario: configure topology
- job preparation scenario: configure job specific settings
- normal operation scenario: configure topology and job specific settings

**System Info Handler**
- prepare configuration
- pass configuration to transporters and functions
- assure complete and correct configuration
4.5. System Level Analysis using Message Flow Diagrams

Generic Transporter
- accept configuration from system info handler
- confirm valid configuration to system info handler

Function
- accept configuration
- confirm valid configuration to system info handler

4.4.8 Notes

Movable functions
A scenario must be specified for future generation mailing machines that have functions equipped with their own drive independent of the substructure.

4.5 System Level Analysis using Message Flow Diagrams

The system level analysis paragraph specifies collaborating behaviour between the different identified models. Message flow diagrams are used to visualise messages between the different models. Based on the conceptual solution data objects that represent mailing packets and process objects that route mailing packet representations are identified in paragraph 4.5.1. Distribution and consistency of the product information is discussed in paragraph 4.5.2. Paragraph 4.5.3 describes the configuration principle used to configure generic transporters and functions. Finally failure handling and generic communications are introduced in paragraphs 4.5.4 and 4.5.5.

4.5.1 Creating mailing packets (normal operation)

Obvious the creation of mailing packets is the key point to model. This paragraph introduces key problems and solutions in the creation of mailing packets. The concept of mailing packet creation is presented in the conceptual solution (paragraph 4.2). Creation of mailing packets is part of the normal operation scenario.

4.5.1.1 Problem Definition

The following problems must be conquered:

Tracking mailing packets
A transporter model must know position and identity of a mailing packet on transport. Position information is required to trigger functions and to notify a successor transporter of the arrival of a mailing packet on its transport. The side of the mailing packet that rests against the cam is used as reference point of the mailing packet.

Personalised mailing packets
A personalised mailing packet is created by executing functions required by the mailing packet and by passing the mailing packet to transporter sections it needs to pass. The mailing packet must be uniquely identifiable.

Side production lines
Mailing packets created by a side production line are combined at the production line it intersects. At the intersection two mailing packets that belong to each other are combined. A side production line has to predict when to start the creation of a mailing packet to have a correct match at the intersection.

Synchronous / A-Synchronous coupling
Two transporter sections are coupled either synchronously or a-synchronously. In the a-synchronous coupling case, the arrival of the mailing packet in the transporter is notified by a position sensor. The moment of arrival of a mailing packet can not be predicted.

Activating functions
A function starts its action (only if required) on a mailing packet when it arrives at a certain position on the transporter.
4.5.1.2 Internal Representation of a Mailing Packet

A mailing packet needs an internal representation that can be handled by the different models. A data object product information ($Pi$) is introduced that contains properties and status information of the mailing packet in creation:

1. functions required
2. functions failed, completed
3. route information

Mailing packet representations are passed between transporter models and the system info handler, two options on representations have been explored:

1. pass $Pi$ of mailing packet.
2. pass data object associated to $Pi$ of mailing packet, the associated process object references a local copy of $Pi$ stored in a local database.

The second option has as advantages compared to the first option:

- mailing packet representation is stored in a local database which is easier to keep consistent as representations whose location is not known (p.e. during a pass).
- less complex data objects are send between the models (which reduces required band-with of CAN during implementation).
- status of mailing packets is available in all generic transporters.

The second option is used for modelling mailing packets. Instead of passing $Pi$ a data object associated to $Pi$ will be passed. A data object product information identification ($Pild$) is introduced that identifies $Pi$ uniquely. The $Pild$ will be passed between the different models and references product information of the mailing packet in a local database. The $Pild$ must identify $Pi$ uniquely to avoid ambiguities.

Product Information

A Product Information ($Pi$) contains the following information about a mailing packet:

- representations of the functions required by the mailing packet
- representation of the path the mailing packet must follow within the mailing machine to pass all required functions.
- the status of the mailing packet:
  - for each required function:
    - function is to be performed on mailing packet.
    - function is active on mailing packet.
    - functionality failed on mailing packet.
    - functionality completed on mailing packet.
  - if function is cancelled.
  - the correct path is followed.

Points 1 and 2 are represented by process classes that are referred to as Services (Services are introduced in next paragraph). Services are uniquely identifiable by an attribute $Serviceld$. Ergo $Pi$ contains a set of $Serviceld$s that represent required route and required functions. To satisfy point 3, a status of the Service is associated with each $Serviceld$ in the $Pi$. A data object $ServiceStatus$ is introduced that represents the following status of a Service:

- unServiced, Service did not start functionality on mailing packet yet.
- pending, Service started functionality but did not complete yet.
- failed, functionality of Service failed.
- completed, functionality of Service finished successfully.
- cancelled, functionality of Service is not required any more.

Point e. is independent of the other status.

A data class that represents a required Service is introduced, named $RequiredService$. It contains as stated above the $Serviceld$ and the $ServiceStatus$.

As result the content of a $Pi$ is a set of $RequiredServices$. $Pi$ is stored in a database local to the transporter model. A process object $Pi_Database$ includes this database. The relation between the introduced classes is visualised in Figure 4-22.
4.5. System Level Analysis using Message Flow Diagrams

4.5.1.3 Introduction to Services

A personalised mailing packet is created by selectively activating required functions and routing the mailing packets through the mailing machine. Activating functions is handled by a process object `Function_Handler` and routing mailing packets is handled by process objects `Product_Input_Handler`, `Product_Output_Handler` and `Side_Product_Input_Handler`, these process objects are introduced in next paragraphs. These process objects have in common:

1. they act on a mailing packet when it moved a number of unit distance on transport.
2. they only act if its action is required by the mailing packet, information about requirement is stored in `Pi`.
3. the result of their action is stored in `Pi`.

This generic behaviour is bundled in an abstract super process class `Service`. The `Product_Input_Handler`, `Product_Output_Handler`, `Side_Product_Input_Handler` and `Function_Handler` are sub classes of super class `Service` (they inherit behaviour of super class `Service`). Figure 4-20 visualises the relation. In future these process objects are referred as `Services`. As imposed by the previous list, process class `Service` features:

1. a representation of the number of unit distance a mailing packet has to be transported in the transporter, before the `Service` starts its functionality. This representation is named `ServicePosition`.
2. determine if the `Service` is required by querying the `Pi_Database`. A `Service` inquires the `Pi_Database` by sending an `isServiceRequired` message to the `Pi_Database`. The `Pi_Database` responds with a message `required` or `notRequired` whether the `Service` is required or not.
3. the `Service` returns success or failure of its action on the mailing packet to the `Pi_Database` with a `serviceCompleted` or a `serviceFailed` message respective.

Product Information (`Pi`) contains representations of `Services` that represent required functions and route within the mailing machine. The representation of a `Service` in the `Pi` needs a unique identification. `Services` must therefore have a unique identifier, process class `Service` is extended with a data object class `ServiceId`.

---

6 data class `ServicePosition` is discussed in the sub system-level analysis of the transporter model in paragraph 4.6.2
4.5.1.4 Tracking mailing packets

To create a mailing packet its associated PiId must pass all functions required by the mailing packet. The PiId is associated to a position on transport. This paragraph discusses how PiIds flow through the mailing machine still associated to the correct position of the mailing packet on transport.

To create a mailing packet the System_Info_Handler must:
1. create a Pi that contains route information and functions required.
2. associate a PiId to the Pi, that follows the route and passes all required functions.
3. initiate creation of a mailing packet by sending a PiId to the main production line

The generic transporter model has to:
- when a mailing packet moves on to a consecutive transporter, its associated PiId must pass to a consecutive generic transporter.
- keep track of mailing packets position and identity on the transporter.
- handle side production lines and functions:
  - pass PiId to activate functions and side production lines
  - check for a correct match, when the mailing packet representation is passed back from the function or side production line.
- image transporter movement.
- route PiId as required by the mailing packet.
- activate function models when a mailing packet reaches a certain position in the transporter.
- only activate functions if required by a mailing packet.

Note that functions are modelled as side production lines, as discussed in the conceptual solution.

PiId Flow

The PiId flow starts and ends in the System_Info_Handler. The System_Info_Handler sends a PiId to a generic transporter at the head of the main production line. The transporter passes the PiId to side production lines, functions and consecutive transporters (all modelled by a generic transporter) as required by Pi of the mailing composed. The PiId is returned at intersections to check for a correct match. When the mailing machine completes the creation of a mailing packet its associated PiId is returned to the System_Info_Handler. The return of the PiId notifies the system info handler of the completion of a mailing packet.

Re-using PiIds

The System_Info_Handler contains a set of PiIds. On each new cycle a PiId is accepted by a generic transporter of the head transporter of a main production line. When the PiId is returned to the System_Info_Handler it may be used again. The set must have a size of at least the number of mailing packets the mailing machine creates simultaneously.

Alternatives:
• assigning on each assignment a successor identification to the \textit{Pild} with the advantage that a transporter model can predict what \textit{Pild} arrives. This alternative does not work when the order of mailing packets can change (p.e. a fly over).
• assigning unique identifiers to the \textit{Pilds} for a complete job results in inefficient use of memory resources in implementation phase. To overcome the problem during implementation, identifiers have to be made reusable anyway.

\textbf{Passing and accepting mailing packets representations}

\textit{Pilds} are passed between generic transporters when:

• a mailing packet moves on to a consecutive transporter.
• a side production line must start the creation of a mailing packet.
• a mailing packet arrives from a side production line.

In the transporter model a \textit{Pild} is passed by process objects with a \textit{passPild} message. Process objects \textit{Product Input Handler}, \textit{Product Output Handler} and a \textit{Side Product Input Handler} are introduced for this purpose. Figure 4-22 shows a message flow diagram of these objects in a simple configuration of generic transporters. The \textit{Product Output Handler} passes \textit{Pilds} to the \textit{Product Input Handler} of a consecutive transporter section or a side production line. The \textit{Product Output Handler} of a side production line passes \textit{Pilds} to the \textit{Side Product Input Handler} of a generic transporter it intersects with. The process object updates product information of the mailing packet the process object acted on in the \textit{Pi Database}, with the result of its action.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4-22.png}
\caption{Modelling Pild-flow}
\end{figure}

The previous introduced process objects are assigned a position at the transporter absolute to the start of the transporter, named \textit{ServicePosition}. The \textit{ServicePosition} of the \textit{Product Input Handler}, \textit{Side Product Input Handler} and \textit{Product Output Handler} indicate respectively the position a mailing packet arrives in a transporter, the position a partial mailing packet arrives from a side production line and the position a mailing packet leaves the transporter or passes a representation to a side production line.

\textbf{Start of creation}

On each new cycle of the transporter, the mailing machine can start the creation of a new mailing packet. The \textit{System Info Handler} must be notified of the new cycle, to initiate the start of the creation by sending a \textit{Pild} to the head generic transporter of the main production line. The \textit{Pi} of a mailing packet must be available, before the \textit{Pild} arrives in a generic transporter. Otherwise the arrival of the \textit{Pild} can not be updated in \textit{Pi}. When the \textit{Pild} and \textit{Pi} are send directly after each other the \textit{Pild} could be accepted, while the \textit{Pi} is not processed yet. To prevent this the \textit{System Info Handler} must prepare and distribute the \textit{Pi} and \textit{Pild} before a newCycle occurs. The \textit{Pild} will be accepted by a generic transporter on a newCycle, the \textit{Pi} certainly is processed and available.

\textbf{Pre-Information}

A side production line needs pre-information to start the creation of a mailing packet to have a correct match. The start moment of the side production line is provided by the production line it intersects with. This moment is indicated by the pass of a \textit{Pild} associated with the mailing packet to be created, from the production line to the side production line. The \textit{Pild} represents the mailing packet that will intersect. The concept is described in the conceptual solution.

\textbf{Combining mailing packets at intersections}

At an intersection between a production line and its side production line two mailing packets are combined into one. The two mailing packets that are combined are a part of the same final mailing packet,
consequently both mailing packets are represented by the same \( \text{Pild} \). When the \( \text{Pilds} \) equal a correct combination occurred, the match is checked after movement of transporter by one unit distance. If the side production line does not pass a \( \text{Pild} \), the delayed check detects that the side production line failed to create the mailing packet.

### A-synchronous coupling

Two transport sections are either coupled synchronously or a-synchronously. In the synchronous case the pass of the \( \text{Pild} \) indicates the physical arrival of a mailing packet in the transporter. In the a-synchronous case a \text{Position_Sensor} is mounted (see Figure 4-3). The \text{Position_Sensor} notices the physical arrival of a mailing packet in the generic transporter. On detection of a mailing packet a \text{productSensed} message is send. When the mailing packet passes the \text{Position_Sensor} completely, a \text{productNotSensed} is send. This \text{productNotSensed} message indicates the end of a mailing packet which is the reference point of a mailing packet.

In the a-synchronous coupling case, the arrival of an empty mailing packet (mailing packet contains no items yet) is not noticed. In this case the pass of the \( \text{Pild} \) indicates the arrival of the mailing packet in the transporter, like in the synchronous case. The order of physical arrival of the mailing packet and the pass of the \( \text{Pild} \) is not fixed within one position of transport movement. Which means that on arrival of a \( \text{Pild} \) of an empty mailing packet, the \text{Product_Input_Handler} waits till the notification of the physical arrival. A fixed order that assures the pass of the \( \text{Pild} \) after the physical arrival solves the problem. The order is assured by a delayed pass of the \( \text{Pild} \) by a number of units distance movement. The number of unit distance depends on the speed difference between the two transporter sections. The postponed pass is compensated for by placing the \text{Product_Input_Handler} that accepts the \( \text{Pild} \), the compensated number of unit distance after the position a mailing packet physically arrives.

### Extending transporter length virtually

When a \text{Service} must be waked before the start of the transporter, the \text{Service} can either be waked by a predecessor transporter or the length of the transporter can be extended virtually. A virtually extended transporter is created by placing the \text{ServicePosition} of the \text{Product_Input_Handler} before the physical start of the transporter. The \text{Product_Output_Handler} of a predecessor transporter passes the \( \text{Pild} \) of the mailing packet before the mailing packet physically leaves the transporter. This principle is very useful when a \text{Service} needs to be waked at the start of the mailing machine. In case of a-synchronous coupling virtual extending is not possible, the physical arrival of a mailing packet is indicated by a \text{Position_Sensor}.

### Gaps and empty mailing packets

On cycles that no mailing packets are generated it is not required to pass \( \text{Pis} \) and \( \text{Pilds} \) to the main production line.

### 4.5.1.5 Activating Functions

A function is activated when a mailing packet moved a number of unit distance on transport, only if its action is required. A function images the section of transport it acts on the mailing packet. Within this section the function is activated again when the mailing packet moved a number of unit distance, to complete its action. A process objects \text{Function_Handler} is introduced that is a sub class of abstract process class \text{Service}. Additionally to the behaviour of the \text{Service}, the \text{Function_Handler} must:

- trigger the function again when a mailing packet moved a certain number of unit distance on the transporter.
- pass the result of the functionality of the function to the \text{PiDatabase}.
- configure the function.

Figure 4-23 visualises a sample mailing line with the process objects introduced to create mailing packets.
4.5.2 Product Information Handling Scenario

The product information handling scenario describes:

- creation of product information (described detailed in system info handler paragraph of Frank Verhofstad), the system info handler prepares product information.
- distribution of product information, the system info handler distributes product information to the generic transporters.
- keeping Pi_Databases consistent, product information is available in all generic transporters and must all contain equal (consistent) production information.
- changing route and required functions of a mailing packet, on a failure of a function product information is adjusted to compensate for the failure or eject the mailing packet.
- determining requirement of Service on a mailing packet

4.5.2.1 Creation of product information

The system info handler composes product information for all mailing packets to be created. Requirements on composition of a mailing packets is obtained from an editor or a reader. To compose product information the system info handler must determine the Services that correctly routes the mailing packet on the transporters and the Services that represent required functions. The ServiceIds that represent these Services are stored in a unServiced state, represented by a RequiredService object in product information (Pi).

4.5.2.2 Distribution of product Information

The system info handler distributes newly created Pi to all generic transporters in the mailing machine. Product information must be available in the Pi_Database before the PiId that references the Pi in the transporter model. Otherwise a Product_Input_Handler updates its completion of its action on the mailing packet in the associated Pi in the Pi_Database that is not available yet. The system info handler broadcasts new product information to all Pi_Databases in the generic transporters with a piNew message with contents the new Pi and PiId that identifies the Pi. In case a Pi identified by the same PiId already exists in the Pi_Database the Pi has become absolute (the PiId is re-used which means that the mailing packet represented by the PiId completed) and can be replaced.
4.5.2.3 Keeping Pi_Database consistent

The Pi_Databases in the different generic transporter must retain consistent, which can be accomplished in two ways:
1. lock product information in all databases before changes are allowed to be made.
2. restrict changes to be made to a part of product information for each generic transporter.

The first option is not feasible, it requires a number of communications to lock and unlock product information to be changed. The communications result in an extra time delay and increases load on the network. The other option implies to split up data to be changed in a collection of disjunct sets. Restricting a generic transporter to change data belonging to Services the generic transporter includes satisfies. A generic transporter is only allowed to update the result of Services it includes. When the Pi_Database is changed the updated Pi is distributed to other generic transporters with a piUpdate message, with contents the updated Pi and Pild that identifies the Pi. The Pi_Databases that accepts a update of a Pi, only updates the status of Services not included by the generic transporter the Pi_Database resides.

4.5.2.4 Changing mailing packet requirements

During production functions required by a mailing packet can be altered as result of p.e.:
1. a function that failed on a mailing packet
2. malfunction of a function (broken)

In both cases an other function with equal functionality can act on the mailing packet to create a correct mailing packet. When the failure can not be compensated for, the mailing packet might be ejected, an ejector are activated and all other functions must be cancelled. Consistency restricts changing required Services for a mailing packet to the system info handler. Suppose that more than one function fails and these functions decide independently to correct the failure by activating a function with equal functionality, which can have unexpected consequences.

Services can be cancelled by a piCancelService with contents the Pild that identifies the Pi Services are cancelled from and a Pi containing the required Services cancelled. A cancelled Service may not be removed from Pi, in case a Service acts or intends to act on the mailing packet when it is cancelled it can not update the Pi or query if its Service required in the Pi_Database! The ServiceStatus must be extended by a cancelled state. New required Services can be added by a piAddService message with same contents as the piCancelService.

Only the system info handler is allowed to add or cancel required functions. Adding a Service is no problem, the Service added is not active yet and can not access a product information. To cancel a Service the cancel field must be set. Only the system info handler is allowed to change this field. Consistency is guaranteed since only the system info handler changes the field.

4.5.2.5 Implementing a database

In essential modelling the database is realised by a dictionary. The Pild is used as key to the element Pi. In implementation the database can be implemented in a two dimensional array with in one dimension all possible Pilds and in the other dimension all ServiceIds, the Pild is used as index. On each Pild ServiceId pair requirement and status is stored. Memory resources requirement can be reduced by storing only parts of Pi concerning Services local to the generic transporter.

4.5.2.6 Determining requirement of Service on mailing packet

A Service is required by a mailing packet when its associated Pi contains a RequiredService representing the Service and the ServiceStatus of the RequiredService is either:
• unServiced, Service did not yet act on mailing packet
• failed, Service failed but did not yet terminate its action on a mailing packet (a re-wake)
• pending, Service is acting on mailing packet and waked again (even on cancelled status, the Service must complete its action on the mailing packet)

4.5.3 Configuration Scenario

The configuration scenario describes configuration of the mailing machine. Three types of configuration are distinguished
4.5. System Level Analysis using Message Flow Diagrams

1. **fixed**, depends on mailing machine hardware, like number of degrees transport movement a function acts. Only accessible to Buhrs engineers.
2. **topology**, layout configuration (topology)
3. **job**, configuration for specific job

Mailing machine components that required configuration:
1. all transporters and functions in mailing machine.
2. all process objects that need configuration in transporter model and function model.
3. all parameters that need configuration in process objects.

To assure a correct and complete configuration of mailing machine components, next behaviour is required:
- create a set of objects to be configured.
- configuration is initiated by sending a configuration start message to the object to be configured.
- accept configuration parameters and remove configured configuration parameters from set.
- on completion send a configuration ready message to the object configured.
- check for valid configuration within a scenario by validating the set of objects configured and the values configured:
  - on valid configuration confirm configuration ready.
  - on invalid configuration generate failure.

### 4.5.3.1 Controlling configuration

The **System_Info_Handler** controls configuration. Configuration is initiated by a **Machine_Mode_Controller** in the system info handler. To configure process objects in generic transporters and functions the **System_Info_Handler** has to:
- create a set of transporters to be configured
- for all transporters to be configured:
  - indicate the start of configuration to generic transporter to be configured.
  - pass configuration parameters to generic transporter.
  - indicate configuration completed to generic transporter.
  - wait for confirmation of configuration completed and remove transporter from set.
- check for valid configuration, in case of an empty set, all transporters are configured completely and correctly, else transporters still in set are not configured correctly.

**Configure process objects**

Process object can contain parameters that require configuration. To configure a process object it must be addressable. Each process object that needs configuration is equipped with a **ProcessId** that identifies the process object uniquely within a function or generic transporter. In combination with the identification of the function or transporter the process object resides, it is uniquely identifiable. In case the process object has parameters that need configuration it must:
- create a set of parameters to be configured.
- announce to be configured to the generic transporter by sending its **ProcessId** to a local configuration handler.
- accept configuration parameters from configuration handler. Relevant configuration messages include its **ProcessId**.
- process accepted configuration parameters.
- on end of configuration notification check for valid configuration, in case:
  - valid configuration, confirm configuration completed.
  - invalid configuration, generate failure indicating configuration error.

This behaviour is joined in an abstract super class **Configurable_Process**. All process objects that require configuration are derived from the abstract process class **Configurable_Process**.

**Configure services**

A **Service** can use the configuration principle of a process object. To re-use the mechanism, a **Service** is a specialisation of abstract process class **Configurable_Process**. Differences between a **Configurable_Process** and a **Service** is that a **Service** is uniquely identifiable in the mailing machine by a **ServiceId** and multiple **Services** exist. Additional behaviour:
• a Service must be assigned a ServiceId. The ServiceId must be assigned to a Service that is of the type the system info handler intended.
• on assigning a ServiceId the generic transporter is aware that the Service needs configuration, all Services require configuration.

Configure generic transporter model
The transporter model is configured by the System_Info_Handler. A generic transporter controls configuration of:
• all process objects in the generic transporter that need configuration.
• all Services in generic transporter.
• all functions that interface the transporter model (a function is interfaced by a Function_Handler which is a Service).

To address a generic transporter it must be uniquely identifiable. The generic transporter model is extended by a unique identification TransporterId.

The transporter model is extended by a process object Config_Handler, that takes care of configuring objects mentioned above. Behaviour required by Config_Handler:
• create set of Configurable_Processes that need configuration
  – Configurable_Processes announce their selves.
  – Services must always be configured when a ServiceId is assigned.
• accept relevant configuration parameters from the system info handler:
  – accept configuration parameters for Configurable_Processes. Relevant messages are recognised by the TransporterId of the generic transporter the Config_Handler resides.
  – accept messages that assign ServiceIds to Services in the generic transporter. Relevant messages are recognised by the TransporterId of the generic transporter the Config_Handler resides. Create a list of Services available in the generic transporter.
  – accept configuration parameters for Services. Relevant messages include ServiceIds of Services included by the generic transporter.
• pass configuration parameters to Configurable_Processes and Services ment for.
• notify Configurable_Processes of completion of configuration.
• check for valid configuration
  – valid, all process objects in set confirm correct configuration, the system info handler is notified of valid configuration
  – invalid, not all process objects in set confirm correct configuration, the system info handler is notified of that configuration failed.

Configure function model
A function interfaces a generic transporter by a Function_Handler. The Function_Handler (Service) included in the generic function is not aware of the configuration requirements of the function. To assure that the function is configured correctly it notifies the Function_Handler that configuring of the function is required. The Function_Handler confirms correct configuration to the Config_Handler when the function confirms to be configured valid.

The generic function must configure all Configurable_Processes in the function that require configuration. A process object Function_Config_Handler is introduced that handles configuration of Configurable_Processes in the function. The Function_Config_Handler behaves like the Config_Handler of the transporter model.

Note: during booting scenario the function notifies the Function_Handler of its identity. To enable the Function_Handler to accept a ServiceId ment for that type of function.

4.5.3.2 Modelling behaviour
Figure 4-24 visualises process objects and messages that participate in configuration of a generic transporter. Figure 4-25 visualises the interactions between the process objects. Behaviour and messages are discussed next.
4.5. System Level Analysis using Message Flow Diagrams

Figure 4-24 Message Flow Diagram of configuration

Figure 4-25 Interaction Diagram of configuration of a generic transporter

**Configuration sequence**
Messages that indicate configuration progress, include nouns in message name:
- **start**, indicates start of configuration.
- **param**, contains configuration parameters.
- **end**, indicates end of configuration.
- **ready**, indicates configuration completed successfully.

**Configurable_Process**
Properties required by a Configurable_Process:
• a list of parameters to configure, all Configurable_Processes create a set of parameters that must be configured. Class Configurable_Process is extended with a set attribute toConfigure that contains all parameters to be configured.

• announce to be configured, announce to the Config_Handler that configuration is required. A message cfgProcess(Processld) to the Config_Handler announces the process object with identification Processld needs to be configured.

• accept configuration parameters, accept parameters destined to the Configurable_Process. Configuration parameters are passed by a cfgProcessParam(Processld, Paramld, Param) message. The content Processld identifies the Configurable_Process the parameter is destined for. The contents CfgParamld and CfgParam indicate identity of configuration parameter and value of configuration parameter respective. The configuration parameters are processed and removed from the set toConfigure.

• accepts end of configuration notification, at this point the Configurable_Process should have a valid configuration. End of configuration is notified by a cfgProcessEnd message form the Function_Handler.

• Two situations are possible on notification of completion:
  - successful configuration, confirm completion of configuration to Config_Handler with a cfgProcessReady(Processld).
  - invalid configuration, the System_Info_Handler must be notified of the invalid configuration. Two types invalid configurations are distinguished:
    - not all parameters are configured
    - configuration is contradictory

Service
Main differences between configuring a Service and a Configurable_Process are:

• a Service must be assigned a Serviceld
• Services are identified by a Serviceld, Configurable_Processes by a Processld

Configuring a Service equals configuring a Configurable_Process. Main difference is point 2, a Service does not need to announce willing to be configured. By assigning a Service a Serviceld the Config_Handler is aware that the Service need configuration. The Serviceld is assigned by sending a cfgServiceAssignld(ServiceType, Serviceld). The Service only accepts the message if the ServiceType is the type of the Service or function in case of a Function_Handler. Names of messages ment for Services are succeeded by the noun service and instead of Processlds Servicelds are send.

Config_Handler
Properties of Config_Handler required to configure Configurable_Processes and Services:

• build a set of process objects to configure, Configurable_Processes that require configuration pass their Processld as contents of a cfgProcess message to the Config_Handler. The Config_Handler creates a list with these Processlds.

• enter configuration, the Transporter_Mode_Controller indicates start of configuration with a cfgStart message.

• accept relevant configuration messages, three kind of configuration message ment for the generic transporter are accepted, messages for:
  - a cfgTransporterParam(Transporterld, CfgParamld, CfgParam) contains configuration for Configurable_Processes, the message is accepted when the content Transporterld of the message equals the Transporterld of the generic transporter the Config_Handler resides. A cfgTransporterParam(CfgParamld, CfgParam) message passes the configuration parameters to the Configurable_Process. The message is accepted by all Configurable_Processes that require configuration of a certain CfgParamld.
  - a cfgServiceAssignld(Transporterld, ServiceType, Serviceld) assigns a Serviceld to a Service of a certain ServiceType, the message is accepted when the content Transporterld of the message equals the Transporterld of the generic transporter the Config_Handler resides. The Serviceld is added to the set of process objects to configure. A cfgServiceAssignld(ServiceType, Serviceld) message assigns the Serviceld to a Service, the message is only accepted by one Service that is of the correct type.
  - a cfgService(Transporterld, CfgParamld, CfgParam) message contains configuration for a Service. Service that are assigned a Serviceld are present in the generic transporter, the message is accepted when the content Serviceld is in the set of process objects to configure. A cfgService(CfgParamld, CfgParam) message passes the configuration parameters to a Service. The message is only accepted by a Service that is assigned the Serviceld from the message contents.
4.5. System Level Analysis using Message Flow Diagrams

- **pass configuration messages**, the received configuration must be passed to the process objects the configuration is ment for.
  - configuration ment for a Configurable_Process is passed by a cfgProcess message.
  - the Serviceld is passes to a Service by a cfgService message.
- **accept end of configuration notification**, the Config_Handler is notified of the completion of configuration by a cfgEnd message, all process objects should be configured at this point. All Configurable_Processes and Services are notified of the completion of configuration by a cfgEnd messages. These process objects determine if they a valid configuration. On valid configuration a Configurable_Process notifies the Config_Handler with a cfgProcessReady(ProcessId) message, a Service notifies the Config_Handler with a cfgServiceReady(ServiceId) message. The objects that have a valid configuration are removed from the set.
- **check valid configuration of generic transporter**, if all process objects that require configuration configurable are configured valid (configuration list is empty) notify the system info handler by a cfgTransporterReady(TransporterId) message. Else send failure to system info handler, if not all Configurable_Processes are configured

### Configuring a Function

A function model is configured by its Function_Handler. Figure 4-24 shows a message flow diagram with process objects that participate in the configuration of a function model. The function is notified of the start of configuration by a cfgStart message. The function must be aware of the start for example to blink a light. The Function_Handler is a Service and handles configuration as discussed for a Service. The function announces with a cfgFunctionRequired message to the Function_Handler that it requires configuration. The Function is appended to the toConfigure set. The Function_Handler passes configuration messages to a Function_Config_Handler in the function model. Like a Config_Handler, the Function_Config_Handler configures Configurable_Processes in the function model. On completion of configuration the Function_Config_Handler passes a cfgFunctionReady message to the Function_Handler and the function is removed from the set.

![Figure 4-26 Message Flow Diagram of configuring a function](image)

### Notes
- configuration principle is a proposal and needs discussion with TNO
- the configuration principle must be incorporated for configuration during the booting scenario, job preparation and normal operation.
- mechanism should be developed to re-configure process during normal operation. Problem is the moment configuration parameters effectuate:
  - effectuate when a process object finished on a mailing packet. Consequence is when transport is halted configuration parameters will not effectuate directly. Which is in some circumstances not desired.
  - effectuate directly, the new value is used directly, on one mailing packet different values for a configuration parameter are used.
A solution is to use a deepcopy of the configuration parameters used when configuration parameters may not change during an action, instead of using parameters that contain configuration.

4.5.4 Failure Handling (all scenarios)

Failures are corrected, recorded and analysed by the system info handler. Failures that occur in process objects in the transporter model and function model are send to a local process object that handles failures. The process objects pass the failure to the system info handler. A generic transporter or function might be able to correct a failure by itself, none failures are identified yet that can be corrected safely (the system info handler must be notified anyway).

In the generic transporter model a process object **Failure_Handler** is introduced that handles failures in the generic transporter model and passes both local failures and failures of the function model to the system info handler. In the function model a **Function_Failure_Handler** is introduced that handles local failures and passes the failures to the generic transporter model it interfaces. The failure handler incorporates failure handling for all scenarios. Figure 4-27 shows a message flow diagram of the failure mechanism.

![Figure 4-27 Message Flow Diagram of failure messages](image)

Failures are passed by a **failure** message with contents the failure and the process object that generates the failure. Failure message contents:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransporterId</td>
<td>failure: identifies generic transporter that generated failure</td>
</tr>
<tr>
<td></td>
<td>failureReset: identifies generic transporter reset is meant for</td>
</tr>
<tr>
<td>ProcessId</td>
<td>failure: identifies process object that generated failure</td>
</tr>
<tr>
<td></td>
<td>failureReset: identifies process object reset is meant for</td>
</tr>
<tr>
<td>ServiceId</td>
<td>only specified in case of a Service</td>
</tr>
<tr>
<td></td>
<td>failure: identifies Service failure generated</td>
</tr>
<tr>
<td></td>
<td>failureReset: identifies Service reset is meant for</td>
</tr>
<tr>
<td>FailureId</td>
<td>identifies failure</td>
</tr>
<tr>
<td>FailureParam</td>
<td>additional parameter which value depends on failure</td>
</tr>
</tbody>
</table>

4.5.5 Generic communications between functions and system info handler (all scenarios)

Function and system info handler must be able to notify each other of some event. A generic transporter must pass the message that represents the event between the function and system info handler. It is impossible to identify all possible events and define messages for each event. Would identification be possible, the message that represents events must be incorporated in the generic transporter model, which obscures genericity.

A mechanism is included by which the system info handler and functions can communicate. The mechanism includes an **eventUp** and **eventDown** which are messages from a function to the system info handler and vice versa respective. The contents indicate the kind of event and the function or system info handler the event is ment for. The generic transporter model is extended by a process object **Event_Handler**. The
Event_Handler accepts all event messages ment for a function that interfaces the generic transporter the Event_Handler includes and passes the events to the function. Event messages generated by a function are passed to the system info handler ment.

Figure 4-28 show a message flow diagram of the event mechanism.

Meaning of contents of both eventUp and eventDown messages:

<table>
<thead>
<tr>
<th>ServiceIdFrom</th>
<th>identifies function that generated event</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceIdTo</td>
<td>identifies function event is ment for</td>
</tr>
<tr>
<td>ProcessIdFrom</td>
<td>identifies process object that generated event</td>
</tr>
<tr>
<td>ProcessIdTo</td>
<td>identifies process object event is ment for</td>
</tr>
<tr>
<td>EventId</td>
<td>identifies event</td>
</tr>
<tr>
<td>EventParam</td>
<td>additional parameter which value depends on event</td>
</tr>
</tbody>
</table>

Failure handling is a specialisation of event handling. Failure handling is extracted since failure handling is required by all specialisation of functions. The generic transporter model does not use the event mechanism it self. All events in the generic transporter model are identified and represented by a unique message.

4.5.6 Controlling Transporter Mode

The generic transporter model is controlled by a Transporter_Mode_Controller. It initiates and controls different scenario. Currently only configuration is controlled by the Transporter_Mode_Controller.

4.6 Sub Level System Analysis using MFD’s of Transporter Model

Sub level system analysis of the generic transporter model, specifies internal behaviour of the transporter model. The transporter model is extended by a process object that images the transporter and a number of data objects that relate to positions on transport. A mechanism is incorporated in the generic transporter...
Chapter 4.

model that activates Services when a mailing packet moved a number of unit distance on transport. The Service queries the Pi_Database to determine if its action is required.

4.6.2 Representations of Position in a transporter

Figure 4-29 shows data objects that represents positions in a transporter in unit distance introduced in this paragraph. These representations are required to represent:

- the current position of transport with respect to the substructure
- position a function starts acting on a mailing packet
- absolute movement of transport

![Figure 4-29 Different data object that represent positions](image)

**Position** is the number of unit distance between a fixed point on the substructure and a fixed point on transport. The fixed point on transport is visualised at different time stamps (t1, t2, .. ti). The difference between two Positions is a PositionDelta, it is a measure for the number of unit distance of mailing packet transport. A PositionAbsolute is a specialisation of PositionDelta, it also is the difference between two Positions, but one of these Positions is fixed and represent the Position at the start of the transporter. A PositionAbsolute is a measure for the number of unit distance of mailing packet transport with respect to the start of the transporter. PositionDelta is a measure for the number of unit distance of mailing packet transport to complete a revolution. It is a specialisation of PositionDelta with the largest possible difference between two Positions.

**Relative position**

The relative position is represented by a data object Position. Data class Position must have these properties to image transport:

1. a representation of the number of unit distance transport moved with respect to the reference point. An integer attribute PositionNumber will indicate the number of unit distance transport moved with respect to the fixed reference point.
2. a representation of the length of the conveyor belt in unit distance. When transport moved the length of the conveyor belt, transport completed a revolution and is back at its starting position. The length of transport is represented by a PositionsInTransporter class.
3. messages that indicate movement backward and forwards. Class Position is notified of movement by one unit distance by sending a movedUnitDistanceForward or a movedUnitDistanceBackward message to the class.

Data class ArrivalPosition and WakePosition are specialisations of data class Position.

**PositionDelta**

Data class PositionDelta must have a property that represents the number of unit distance between two Positions. The number of unit distance will be represented in an integer attribute deltaNumber.

Data class PositionDelta and PositionsInTransporter are specialisations of data classes PositionDelta.
4.6. Sub Level System Analysis using MFD's of Transporter Model

Introduction to specialisations of related classes

The following classes are specialisations of classes discussed above. In the remainder of this paragraph, process objects are introduced that use these data classes:

**ServicePosition**

Data class *ServicePosition* represents the number of unit distance the trigger point of the *Service* is from the start of the substructure. Data class *ServicePosition* is a specialisation of data class *PositionAbsolute*.

**ArrivalPosition**

Data class *ArrivalPosition* represents the position the mailing packet arrived at the start of the substructure. The position a mailing packet arrives is the current *Position* subtracted by the position of the *ProductInputHandler* (*ServicePosition*). This correction is required when more than one *ProductInputHandler* exist that have different *ServicePositions*. Without correction the *ArrivalPosition* would be related to the *ServicePosition* of the *Service* the mailing packet accepted. Data class *ArrivalPosition* is a specialisation of data class *Position*. Data class *Position* is extended with a message *asArrivalPosition*(*ServicePosition*) that creates an *ArrivalPosition* from a *Position*.

**WakePosition**

Data class *WakePosition* represents the position a *Service* wants to be waked up for a certain mailing packet. The position a *Service* wants to be waked up is the position the mailing packet arrived (*ArrivalPosition*) added with the position on the substructure of the *Service* (*ServicePosition*). Data class *WakePosition* is a specialisation of data class *ArrivalPosition*. Data class *ArrivalPosition* is extended with a message *asWakePosition*(*ServicePosition*) that creates a *WakePosition* from an *ArrivalPosition*.

Figure 4-30 visualises the relation between the different objects that concern *Positions*.

![Figure 4-30 Classes related to data class position](image)

4.6.2.2 Imaging transporter movement

To track transporter movement a process object *Transporter_Image* is introduced. The *Transporter_Image* images movement of the transporter. The *Transporter_Image* keeps track of the relative position of the transporter in a data class *Position*. The transporter generates a *fineShiftPulse* message on each unit distance movement of the transporter in the direction set by a *forwardDirection* or *backwardDirection*, as discussed in paragraph 4.1.4. The relative position is adjusted accordingly by sending a *movedUnitDistanceForward* or *movedUnitDistanceBackward* to *position*.

New cycles are imaged by the *Transporter_Image*. During backward movement *newCycle* messages are re-generated. The original *newCycle* message is send during forward movement, during backward movement and forward movement again it is generated twice again. The duplicated *newCycle* message must be avoided, otherwise three mailing packets are generated on one cycle.

**Distributing transporter movement**

Transporters and Functions not equipped with position encoders image a *Transporter_Image* of a transporter equipped with position encoders. Transporters are equipped with position encoders. A *Transporter_Image* connects to a position encoder distributes movement of the transporter to other *Transporter_Images* with a *distributeFineShiftPulse*, *distributeForwardDirection*,
Chapter 4.

distributeBackwardDirection and a distributeNewCycle message with contents the TransporterId of the transporter model that includes the Transporter Image. Transporter Images not equipped with an position encoder can image the Transporter Image required by accepting messages with contents the TransporterId of the transporter model that includes the Transporter Image they image.

4.6.2.3 Triggering Services

Service are activated when a mailing packet moved a certain number of unit distance in a transporter section. On arrival of a Pild in the transporter model, the transporter model must:
1. notify all Services of the Position the mailing packet arrived (ArrivalPosition).
2. each Service determines position to be waked (WakePosition).
3. wake Service when the mailing packet moved a required number of unit distance in the transporter (ServicePosition).
4. the Service inquires if its action is required for the mailing packet waked for.
5. start action of Service on mailing packet.

Figure 4-33 shows a message flow diagram with process objects involved in activating Services. New process objects and behaviour are discussed next.

![Message Flow Diagram](image)

**Figure 4-31 Message flow diagram of process objects involved in activating Services**

A Wakeup_Unit is introduced that wakes Services when a mailing packet arrives at a position on the transporter they want to act on the mailing packet. A Service notifies the Wakeup_Unit to be waked on a WakePosition for a certain mailing packet represented by a Pild. Each unit distance transport moves on, a deepcopy of Position in the Wakeup_Unit is updated by the Transporter Image. When transport moves on to a new Position, all Services that must be waked at the new Position are waked for the mailing packet they want to act on.

On arrival of a Pild (mailing packet) with a passPild(Pild) message in a Product_Input_Handler, the ArrivalPosition is determined. Which initiates the points mentioned above:
1. all Services must be notified of the arrival of a mailing packet in the transporter, for this purpose a Pild_Distributor is introduced. The Pild_Distributor receives from the Product_Input_Handler with a pildArrived( Pild, ArrivalPosition ) message the ArrivalPosition and the Pild of the mailing packet arrived. The Pild_Distributor notifies all Services with a distributePild( Pild, ArrivalPosition ) message that a mailing packet arrived identified by a Pild at an ArrivalPosition.
2. the Services accept the message and determines a WakePosition. The Service programs the Wakeup_Unit with a wakeAt(ServiceId, Pild, WakePosition) message. Which notifies the Wakeup_Unit

---

An interrupt message flow is used that interrupts the current behaviour of the Service to determine the WakePosition

-60-
4.6. Sub Level System Analysis using MFD's of Transporter Model

to wake the Service when transport reaches the WakePosition for the mailing packet represented by the Pild.

3. the Wakeup_Unit is updated with the Position of transport on each unit distance transports moves. All Services that have a WakePosition that equals Position are waked with a wake(ServiceId, Pild) message. The ServiceId represents the Service to be waked and the Pild represents the mailing packet waked for. The message is only accepted by a Service which ServiceId equals the ServiceId in the content of the wake message (conditional receive).

4. when a Service is waked the Service determines if it is required for that mailing packet. The Service queries the Pi_Database by sending an isServiceRequired(ServiceId, Pild) message. The Pi_Database returns a required(ServiceId, Pild) or notRequired(ServiceId, Pild) message to the Service depending on requirement.

5. if required the Service starts acting on the mailing packet

6. when the action fails a serviceFailed(ServiceId, Pild) message is send to the Pi_Database to update the product information of the mailing packet in the Pi_Database.

7. on successful completion of its action the Service notifies the Pi_Database with a serviceCompleted(ServiceId, Pild). The product information of the mailing packet the Service acted on is updated.

Example

Figure 4-32 Creating mailing packet flow and contents of Wakeup_Unit

Figure 4-32 shows a transporter with a feeder mounted and a table with the contents of the Wakeup_Unit for a certain mailing packet X, at different Positions on transport. A PIH accepts the representation of a certain mailing packet X in the transporter at arrival position 0 (Pos 0). All Services are notified of the arrival of mailing packet X and determine the position they have to be waked. The Services program the Wakeup_Unit, the table shows the WakePositions: POH_1 'wants to be waked at Pos 1, SIH at Pos 5 and POH_2 at Pos 6. When transport moves on and arrives at Pos 1, the Wakeup_Unit wakes POH_1. The wake information is removed from the table. POH_1 passes the Pild of mailing packet X to the PIH of the feeder. The Services of the feeder program the Wakeup_Unit, FH_1 at Pos 2, FH_2 at Pos 3 and POH at Pos 4. At this moment the transporter and feeder both act on he same mailing packet. As mailing packet X arrives at Pos 2, FH_1 is waked and at Pos 3, FH_2 is waked. At Pos 4 the Pild is passed by the POH of the feeder to the SIH of the transporter. The feeder is completed with mailing packet X, the table is empty. At Pos 5 the SIH of the transporter is waked, the SIH checks if a correct match occurred. At Pos 6, POH_2 is waked and passes the representation to a consecutive transporter, a side production line, a function or the system info handler.

\[8\] The feeder does not represent a realistic model
4.7 Sub system level analysis of a generic function model

This paragraph specifies behaviour required to model all functions using a generic function model. The Generic function model consists of a Function_Handler and a model of a function. The function interfaces with a generic transporter using a Function_Handler (which is a Service). Required behaviour of a generic function:

1. uniquely identifiable.
2. wake function when mailing packet moved a number of unit distance in the transporter, only if function required by mailing packet.
3. wake function again when mailing packet moved a number of unit distance on transport.
4. update result of action on mailing packet to Pi_Database.
5. enable configuration.
6. enable events.
7. enable failures.

Figure 4-33 shows a normal operation message flow diagram of a generic function. The Function_Handler and messages are discussed next:

Figure 4-33 Normal operation message flow diagram of a generic function

Behaviour 1,2 and 3 are provided by the Service part of the Function_Handler:

1. identification is provided by the ServiceId of the Function_Handler. The Function_Handler hides its ServiceId to the function. All contents of messages from the function that require identification, like events, failures and database updates are extended by its ServiceId.
2. the number of unit distance a mailing packet moves in a transporter before the Service acts is represented by a ServicePosition. On arrival of a mailing packet the Service programs the Wakeup_Unit to be waked on a certain position for the mailing packet arrived. The function is only waked by the Function_Handler if its action is required.

The Function_Handler is extended with the remaining behaviour:

4. program the Wakeup_Unit to be waked again, when a mailing packet moved a number of unit distance in the transporter. A wakePositionDelta(WakePositionDelta, Pild) notifies the Wakeup_Unit that the Service must be waked again when transport moved a number of unit distance indicated by the PositionDelta for a mailing packet represented by a Pild. The Service is re-waked by a wake(ServiceId, Pild) message.
5. the Function_Handler passes the serviceCompleted and serviceFailed messages from the function to the Pi_Database. The contents of the message is extended by the ServiceId that represents the function.
6. configuration of a function is discussed in the configuration scenario paragraph in system level analysis (paragraph 4.5.3.1)
4.8 Sub level analysis of a Rotary Feeder

In this paragraph a rotary feeder is specified and modelled. First behaviour of rotary feeders components is specified. Based on the specification the rotary feeder is modelled, collaborating objects are introduced and messages between them defined.

4.8.1 Transporter

A rotary feeder consists of a drum with a vacuum valve that is mechanically coupled to the transporter. The vacuum valve has the same cycle time as the transporter. The rotary feeder can be placed on transporters that have different number of chains per cycle. The number of chains per cycle is not fixed, triggering information is represented by a number of degrees per cycle. The degrees must be converted to unit distance movement of transport (number of chains). Position and identity of mailing packets on the transporter is modelled using the transporter model.

4.8.2 Specification of a rotary feeder

For all system components behaviour is specified that is required to correctly feed a mailing item and enable detecting a correct feed.

Vacuum Valve

An item is fed by enabling and disabling the Vacuum Valve at appropriate moments. In case an item is required, the Vacuum Valve must be enabled at 180 degrees. At twelve o'clock the item is picket from the loader. When the Vacuum Valve returns at 180 degrees, the feed is completed. If next mailing packet does not require an item from the feeder, the Vacuum Valve must be disabled. If the item is required the cycle starts again, and the Vacuum Valve stays on to feed next item. Switching the Vacuum Valve on and off shortly after each other is undesired.

Feed fail detection

Feed fail detection is enabled when the Vacuum Valve is between 270 degrees and 180 degrees. Feed fail detection must be disabled at 180 degrees, around that position it is not known for which mailing packet the fail is generated. This uncertainty can be avoided by equipping the feeder with its own decoder.

Combined behaviour of vacuum valve and feed fail detection

Combining both behaviour of the Vacuum Valve and feed fail detection, the behaviour in Table 4-1 is desired. The behaviour can be modelled as consecutive wakes of a 'feeder controller' by the Wakeup_Unit.

<table>
<thead>
<tr>
<th>Degree</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>180°</td>
<td>enable vacuum valve</td>
</tr>
<tr>
<td>270°</td>
<td>enable feed fail detection</td>
</tr>
<tr>
<td>180°</td>
<td>disable vacuum valve and feed fail detection</td>
</tr>
</tbody>
</table>

Table 4-1 Behaviour of vacuum valve and feed fail

Table 4-2 shows the combined behaviour of the feeder on two successive mailing packets. At 180 degrees the vacuum valve is disabled and enabled at the same moment for both mailing packet a and mailing packet b. The order of disabling and enabling is unknown since 'Feeder_Control' is waked by two independent wakes for two different mailing packets. In case the vacuum valve is enabled for mailing packet b and next the vacuum valve is disabled for mailing packet a, the vacuum valve is disabled for mailing packet b and none items are fed. This situation must be avoided.

---

9 movement of a conveyor belt by one chain indicates movement of transport by one unit distance (paragraph 4.1.4.1)
Chapter 4.

<table>
<thead>
<tr>
<th>Mailing packet a</th>
<th>Mailing packet b</th>
</tr>
</thead>
<tbody>
<tr>
<td>180° enable vacuum valve</td>
<td>enable vacuum valve</td>
</tr>
<tr>
<td>270° enable feed fail detection</td>
<td>enable feed fail detection</td>
</tr>
<tr>
<td>180° disable vacuum valve and feed fail detection</td>
<td>enable vacuum valve</td>
</tr>
<tr>
<td>270°</td>
<td>enable feed fail detection</td>
</tr>
<tr>
<td>180°</td>
<td>disable vacuum valve and feed fail detection</td>
</tr>
</tbody>
</table>

Table 4-2 Behaviour on two successive mailing packets

The vacuum valve may only be disabled if next mailing packet does not require an item from the feeder. A solution is to delay the disabling by at least one position movement or at least one degree, to assure the order of wake ups. If the vacuum valve is enabled by next mailing packet the vacuum valve must not be switched off. Table 4-3 shows behaviour of the feeder with a fixed enabling and disabling order. Figure 4-34 shows the behaviour graphically.

<table>
<thead>
<tr>
<th>Mailing packet A</th>
<th>Mailing packet B</th>
</tr>
</thead>
<tbody>
<tr>
<td>180° enable vacuum valve</td>
<td>enable vacuum valve and disable feed fail detection if enabled by previous mailing packet</td>
</tr>
<tr>
<td>270° enable feed fail detection</td>
<td>enable feed fail detection</td>
</tr>
<tr>
<td>180° disable vacuum valve and feed fail if not enabled by next mailing packet</td>
<td>enable feed fail detection</td>
</tr>
<tr>
<td>270°</td>
<td>disable vacuum valve and feed fail if not enabled by next mailing packet</td>
</tr>
</tbody>
</table>

Table 4-3 Desired behaviour on two successive mailing packets

Figure 4-34 Desired behaviour graphically represented

Feed fail
On a mis feed or double feed a failure must be generated and the light must blink once. On a pre-set number of successive mis feeds or double feeds the system info handler must be notified, depending on configuration the mailing machine stops. The light continuously blinks slowly.

Loader
When the number of items is less than a pre-set minimum the operator must be notified. The operator must re-fill the loader. The attention of the operator is attracted by a slow blinking light.

Light
The light of a feeder blinks (in order of priority, highest first):
- during configuration, the light continuously blinks fast.
- on an error, the light continuously blinks slowly.
  - a successive number of feed fails
  - minimum level detection of the loader
- a feed fail, the light only blinks once.

4.8.2.2 Modelling
Four objects are introduced that model behaviour specified above:
1. Feeder_Control, controls the vacuum valve and handles mis feeds and double feeds
2. **Feeder_Failure_Control**, handles failures and situations that can cause failures
3. **Feeder_Config_Handler**, handles configuration
4. **Feeder_Light_Control**, controls light

The externals are imaged by process objects:

- **Vacuum_Valve_Image**: controls the vacuum valve. By sending a `vacuumOn` respective `vacuumOff` message to the `Vacuum_Valve_Image` the vacuum valve is enabled and disabled.
- **Doublefeed_Detector_Image**: images the `Doublefeed_Detector`. A `doubleFeed` message is generated on a double feed detection of the `Doublefeed_Detector`.
- **Misfeed_Detector_Image**: images the `Misfeed_Detector`. A `productSensed` message is generated when the `Misfeed_Detector` detects an item.
- **Min_Level_Image**: images the number of items in the `Loader`.

Figure 4-35 shows a normal operation message flow diagram of the rotary feeder with the introduced process objects. The behaviour and message are discussed next.

**Figure 4-35 Message Flow Diagram of rotary feeder (normal operation)**

### Feeder_Failure_Control
Process object **Feeder_Failure_Control** counts the number of successive mis feeds and double feeds. On a configured number of successive fails a `failure` message is generated. On each mis feed and double feed **Feeder_Control** sends a `misFeed` and `doubleFeed` message respective. On fail a counter that represent the number of successive fail is increased and the light is imposed to flash by sending a `flash` message to the **Feeder_Light_Control**. To distinct successive feed fails from incidental fails **Feeder_Control** sends a `correctFeed` message on each correct feed. On a correct feed the counters are reseted. When the counter equals a pre-set value a `failure` message is generated and the light is set to blink by a `errorBlinkOn` message. On reset of the failure by a `failureReset` message the counter is reset and a message `errorBlinkOff` is send to the light to stop blinking.

Process object **Feeder_Failure_Control** also takes care of availability of items in the `Loader`. The **Minimum_Level_Detector** notifies **Feeder_Failure_Control** of a minimum number of items available in the loader by sending a `minLevelDetected` message. **Feeder_Failure_Control** notifies the **System_Info_Handler** using an `eventUp` message indicating the approaching shortage. The **System_Info_Handler** notifies the operator. If the loader is filled above the minimum level the **Minimum_Level_Detector** generates a `storageFilled` message. The **System_Info_Handler** is notified by sending an `eventUp` message that indicates that the Loader is re-filled.
Feeder_Config_Handler
Process object Feeder_Config_Handler is a specialisation of the Function_Config_Handler described in the configuration scenario (paragraph 4.5.3.2). Additionally the Feeder_Config_Handler notifies Feeder_Light_Control when the transporter enters and leaves configuration by a configBlinkOn and configBlinkOff message respective.

Feeder_Light_Control
Feeder_Light_Control controls blinking of the light as described above in three distinctive ways: flash once, slow blinking and fast blinking. The light blinks fast during configuration. Start and end of configuration are indicated by respective a configBlinkOn and a configBlinkOff message. During a failure the light blinks slowly. Start and end of a failure are indicated by respective a errorBlinkOn and a errorBlinkOff message. A flash is generated when a flash message is send.

Feeder_Control
The behaviour of Feeder_Control is discussed using an interaction diagram, visualised in Figure 4-36.

**Figure 4-36 Interaction diagram of Feeder_Control**

1. Feeder_Control is waked for the first time to act on a mailing packet.
2. Feeder_Control is waked by the Function_Handler to act on a mailing packet represented by PIdA.
3. The vacuum valve is enabled if not enabled by previous mailing packet.
4. Feeder_Control programs the Wakeup_Unit to be waked after 90° to act on the mailing packet represented by PIdA.
5. Enable misfeed detection
6. Enable misfeed fail detection, record if the Doublefeed_Detector detects a double feed and if the Misfeed_Detector detects an item
7. Feeder_Control programs the Wakeup_Unit to be waked after 300° to act on the mailing packet represented by PIdA.
8. Misfeed_Detector detects a mailing item
9. A mailing item is sensed, which indicates that an item is fed.
4.10. Requirements catalogue

- **Doublefeed_Detector** detects a double feed
7. A double feed is detected.
8. Make the light flash on a single feed fail, make the light blink when a number of successive blinks occurred.
9. Notify **Pi_Database** that feeder acted incorrectly on the mailing packet represented by **PildA**.
10. Feeder_Failure_Control notifies the system info handler of the double feed

110° - Feeder is waked for the first time to act on next mailing packet
11. **Feeder_Control** is waked for next mailing packet represented by mailing **PildB**. Since **Feeder_Control** is waked for an other mailing packet, its action is required on the successor of mailing packet A.
12. In this case the vacuum valve is still enabled for mailing packet A and needs not to be enabled.
13. Like point 3, program the **Wakeup_Unit** to wake after 300° to act on mailing packet represented by mailing packet B.

210° - Feeder finishes on (first) mailing packet
14. **Feeder_Control** is waked to complete its action on mailing packet A. It must determine if a correct feed occurred and if it needs to disable the vacuum valve.
15. If next mailing packet does not require an item then disable the vacuum valve. In this case mailing packet B requires an item, the vacuum valve is not disabled.

- If the Misfeed_Detector did not detect a mailing item, a misfeed occurred
16. **Feeder_Failure_Control** is notified of the misfeed
17. Notify **Pi_Database** that feeder acted incorrectly on the mailing packet represented by **PildA**.
18. **Feeder_Failure_Control** notifies the system info handler of the misfeed

- If a correct feed occurred, that is no double feed and an item is detected
19. Notify **Feeder_Failure_Control** of a correct feed.
20. Notify **Pi_Database** that feeder completed action correctly on mailing packet A

4.9 Unified Model

The unified model formalises the structure visualised in an instance structure diagrams and the behaviour of all cluster classes, process classes and data classes. POOSL-code of generic transporter model, rotary feeder model and model of the system info handler are described in the appendices that present the graphical representations.

4.10 Requirements catalogue

The requirements catalogue contains all items that can not be formalised in the unified model. All aspects of the requirements are discussed in the thesis and are not restated.
Chapter 5

Development of prototype controller

Buhrs provides a prototype mailing machine to verify the generic transporter model and model of the rotary feeder. To realise the implementation an extended specification must be realised. The extended model incorporates additional behaviour imposed by used technologies like CAN and the controller board with limited resources. On completion of extended specification an implementation is realised on the prescribed hardware coded in ANSI C.

5.1 Objectives

Specification of the mailing machine is split up in phases. It is undesirable to make a complete specification with all features incorporated, without validating some key problems on a prototype.

First phase specification that is realised implements:
• full distributed control architecture
• scenarios:
  – Booting Scenario
  – Job Preparation Scenario
  – Normal Operation Scenario
  – Product Information Handling Scenario
  – Configuration Scenario
• minimal System_Info_Handler

Main objective is verification of the transporter model:
• correct tracking of mailing packets
• functions are triggered at correct positions
• synchronous / a-synchronous coupling
• side production lines
• personalised mail

5.2 Prototype structure

To verify the model a prototype mailing machine is created on which the transporter model can be tested. The prototype production line consists of two consecutive transporter sections. The head transporter section is built from four stations that mount a main insertion, two rotary feeders and a label head 310, the other transporter section is a variajet basis. The production line is build from two transporter sections coupled a-synchronously, both transporter sections have their own position encoder. Figure 5-1 shows the prototype mailing machine from a side view and a top view.
5.3 Modelling prototype mailing machine

The prototype mailing machine is modelled by the models specified. For the transporters and rotary feeders models are completed. The main insertion and label head are not specified yet. The System_Info_Handler is not specified completely yet. A system info handler with a minimum behaviour required to verify the models, is specified and formalised in POOSL.

5.3.1 System_Info_Handler

TNO did not specify a System_Info_Handler in POOSL, a temporary System_Info_Handler is specified. The System_Info_Handler that will be (temporarily) used has only a minimal set of functionality that is necessary to test the transporter model. The external behaviour is as specified. Main differences on the minimal behaviour with respect to the proposed System_Info_Handler by TNO:

- the layout configuration database is created by hand
- failure handling not included

5.3.2 Distributed control

The architecture with distributed control will be implemented on the prototype. All functions and transporters are equipped with a controller board. All transporters and functions will be implemented on a controller card. Both transporters, main insertion and both rotary feeders are equipped with a controller board.

5.4 Implementing specification

To implement extended specification, the POOSL description must be realised in implementation technologies. The formalisation in POOSL must be converted to ANSI C. The conversion must handle message passing inside distribution boundaries and between distribution boundaries. Messages between distribution boundaries must comply to the CAN-protocol.

5.4.1 Converting POOSL to ANSI C

The initial requirements description prescribes use of a KEIL development tool, featuring a complete implementation of the ANSI standard of the C-language. An automatic mapping tool that converses of POOSL to C++ is completed. Unfortunately available resources on the controller board (data and code
memory) prevent using C++. An implementation of POOSL in ANSI C being developed. The mapping on ANSI C is also restricted by controller board resources.

### 5.4.1.1 Restrictions laid up by mapping

#### Bounded Integers
Integers are bounded, they have a limited range from \(-2^{15}\) to \(2^{15}-1\).

#### Maximum delay
Delay time is restricted by the bounded integers as well.

#### Garbage Collection
It is not possible to implement garbage collection. Garbage collection requires a lot of resources like processor time and data memory. All data objects returned from a method must be assigned to a variable. On discartion of the variable (method finishes) or variable is assigned to an other data object, the data object referenced by the variable is deleted when it is referenced once.

#### Abort not implemented
The abort primitive is not implemented. For an easy implementation the aborted block must be a thread and garbage collection is required. In that case, on an abort the thread is terminated and all data objects used by the aborted block do not have to be eliminated, eventually garbage collect will eliminate them. An abort can only be implemented with a huge overhead. Each block (at least those that can be aborted) requires a list with data objects used. On abort all these data objects have to be deleted. During modelling aborts have not been used yet. If an abort is necessary, it has to be decided if an abort has to be implemented or the model will be restricted.

#### Cyclic Data objects
Cyclic data objects may not be used, deepcopies and removal of those data objects is hard to implement. A double linked list is an example of a cyclic data object (Figure 5-2).

![Figure 5-2 Example of a cyclic data object](image)

#### Data object size
Complexity of a data object is limited by a pre-set setting. A data object is composed of three parts:
1. size of data object contains size of primitive part and size of references.
2. primitive part, contains primitive data.
3. references to other data objects.

Part 2 and 3 are limited in size and number respective are represented by part 1.

#### No Complex Selects
Nested selects and statements in a select that can be interrupted are not implemented. Implementation would require additional threads.

#### Side Effect Free Expressions
Use of expression with side effects is restricted. An expression with a side effect will change variables which makes re-evaluation impossible. Re-evaluation is required in some case. Only an implementation with a huge overhead that will copy all variables for evaluating and restoring all variables after evaluating would overcome this problem. In the following cases an expression with a side effect is not allowed.
- an expression that can be interrupted like expressions in guards and expressions in parameter list of a message. The interrupt can change variables used by the expression. If variables are changed the expression can evaluate to a different result. Therefore after each interrupt the expression has to be re-evaluated.
- expressions in parameter list of a message that are received with a condition. It is recommended to never use an expression in an expression list of a message that is not side effect free. One never knows (at least
not in the future) if a message is received with a condition. Each time a process object is willing to receive the message the expression is evaluated.

**Parameters referring to the same data object**

It is not allowed to use data objects in a parameter list that have data objects in common. The implementation will fail to create a correct deep copy. Instead it will create two independent data objects. Figure 5-3 visualises a failed deep copy.

![Figure 5-3 Example of failed deep copy](image)

**Restrictions by Real Time Operating System**

The number of process objects and interrupt created is limited. A maximum of 254 tasks is available in the CMX real time kernel. Each process or interrupt needs one task. E.g., 127 process objects with each one interrupt can be created. Note that the RTX51-tiny allows a maximum of 16 tasks.

### 5.4.1.2 Generic Set-up

The implementation of POOSL in ANSI-C can be compiled with the UNIX-POSIX thread library and the RTX-tiny library.

### 5.4.2 Assigning identifiers to messages

Message passing between distribution boundaries are realised by using frames of the Control Area Network (CAN). The CAN-protocol defines frames that roughly consists of an identifier, data and control. Each message must be assigned an identifier. The identifier in a CAN message identifies the CAN-message and defines bus-priority. An algorithm must be developed that assigns identifiers to messages.

**Restrictions imposed by CAN-Protocol**

Constraints laid up by CAN-protocol:
1. an identifier consist of 11 bits (ID.10 to ID.0), the most significant bit is sent first.
2. an identifier with the seven most significant bits (ID.10 to ID.4) recessive is forbidden.
3. transmitting more than one message at the same time with the same identifier is forbidden.
4. the identifier determines bus access priority, the identifier with the lowest binary value has the highest priority.
5. the CAN-controller contains an acceptance filter that consists of 8 bits that can either be set (1), reset (0) or don’t care (x). The acceptance filter is compared to the eight most significant bits of the identifier (ID.10 to ID 3), when the relevant bits match the message is accepted.

**POOSL requirements on messages**

The message passings offered by POOSL must be modelled for a mapping with a minimum of restrictions. A process object can send a message that is received by only one process object or can broadcast a message that is received by all process objects ready to receive the message. A message can be received depending on its contents (conditional receive). Sending and receiving can be a part of a select.

Selects are abandoned, a number of additional message passings are required to guarantee a dead-lock free implementation. Analysing message passings between distribution boundaries shows that less than 16 unique message are send between two transporter models or a transporter model and the system info handler.
Assigning identifiers

A message between distribution boundaries must be identified uniquely. A CAN frame contains an identifier field that can be used to identify the message. Each message must be assigned an identifier that fills the constraints and requirements. A field in the CAN-identifier is used to identify the message.

As result of constraint 3, two problems arise:

1. All transporter models send equally named messages to other transporters and the system info handler. This problem can be solved by either placing the TransporterId of the sending generic transporter or the TransporterId of the receiving generic transporter in the identifier. The advantage of placing the TransporterId of the receiving generic transporter is that the message can be received by a conditional receive on the TransporterId.

2. Each transporter model sends equally named messages to the system info handler. This can be solved by placing the TransporterId of the sender in the identifier. The TransporterId of the receiver (system info handler) won’t work, all messages will still be the same.

Conclusion: A field in the CAN-identifier is required that contains a TransporterId, the field must be accessible to the acceptance filter for conditional receiving purposes.

The system info handler can not determine on these two fields if the message is ment for the system info handler. A third field in the CAN-identifier is required that is accessible to the acceptance filter. The field indicates that a message is ment for the system info handler.

The three identified parts must be split over 11 bits. The part that indicates if a message is ment for the system info handler requires 1 bit. To use the acceptance filter optimal, a maximum number of identifiers that identify generic transporters must be realised. Using 6 bit for an identifier that have 3 bit set and 3 bit reset results in 20 unique identifiers that can uniquely received by the mask. Using 7 bits with 3 bit set and 4 bit reset realises 35 unique identifiers. Additional identifiers can be created by using 1 bit set more. The identifiers are not uniquely identifiable any more. An additional software filter is required to accept messages with a identifier desired.

The identifier (11 bit) can be split up in three parts, two split ups are of interest:

<table>
<thead>
<tr>
<th>Part</th>
<th>(1)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. part that represents the sender or receiver of the message</td>
<td>6 bit</td>
<td>7 bit</td>
</tr>
<tr>
<td>2. part that indicates a message between two generic transporters or a message between a generic transporter and a system info handler</td>
<td>1 bit</td>
<td>1 bit</td>
</tr>
<tr>
<td>3. part that identifies the message</td>
<td>4 bit</td>
<td>3 bit</td>
</tr>
</tbody>
</table>

The first two parts must be accessible to the mask. In the first option one bit of the message part is accessible to the mask. In this case the message influences priority and 16 unique message are possible. In the second option the message does not influence priority of the message and only 8 unique messages are possible.

Masks used for option 1:

<table>
<thead>
<tr>
<th>TransporterId</th>
<th>Send to Generic transporter</th>
<th>Send to System_Info_Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransporterId</td>
<td>x 1 x 1 x 0 TransporterIdTo xxx</td>
<td>x 0 TransporterIdFrom xxx</td>
</tr>
<tr>
<td>System Info Handler</td>
<td>x 0 x 1</td>
<td>not possible</td>
</tr>
</tbody>
</table>

A disadvantage of assigning identifiers by this mechanism is that priority is largely determined by the TransporterId and if the message is ment for system info handler or not.

5.5 Behaviour preserving transformations

To reduce resource requirements, process objects can be assembled using behaviour preserving transformations, p.e. the Product_Input_Handler and the Pild_Distributor can be assembled. Reduction of the number of process objects, reduces resources required.
Chapter 6

Conclusions

6.1 Conclusions
A generic model of transport is specified and formalised in POOSL, based on a conceptual solution accepted by all parties. The generic transporter model solved problems like complex side production lines, a-synchronous coupling and activating functions. In co-operation with Buhrs engineers behaviour of a rotary feeders is specified and collaborating process objects identified. The specification is formalised in POOSL. To verify the behaviour of the generic transporter model and rotary feeder, a mailing machine is modelled in a simulator. A minimal set of behaviour of the system info handler required for the simulation, is formalised in POOSL. Simulations of the models resulted in the behaviour specified. To implement the specification on a prototype mailing machine a conversion from POOSL to ANSI-C is developed. Only communications within a distribution boundary are implemented yet. Most of the design and the specification is described in the thesis.

Specification is realised using the Software/Hardware Specification method which proofed to be capable of specifying complex reactive systems. Scenarios are helpful to focus on distinctive behaviour. The different graphical representations enable discussion on behaviour and identification of collaborating process objects. Formalisation of behaviour and structure in POOSL enabled verification of the different models in the simulator.

6.2 Future Work
To realise an implementation on the prototype mailing machine, both the extended and implementation phase must be realised. To realise an implementation of the specification of the prototype mailing machine, next steps are needed:

- realise an extended specification, restrictions laid up by implementation technologies are incorporated in the essential specification.
  - Control Area Network: model the Control Area Network in essential specification.
  - Controller board: optimise specification for limited resources: combine process objects using behaviour preserving transformations. Avoid creating instances, to avoid memory defragmentation.
  - ANSI C, optimise data structure like sets and dictionaries to ANSI C friendly data structures.
  - formalise extended specification in POOSL.
- implement extended specification on prototype mailing machine:
  - transform POOSL to implementation language ANSI C, automatic conversion enables easy upgrades and extension of the specification. The implementation is consistent with the specification.
  - implement CAN in POOSL to C mapping.
  - assign identifiers to messages.
- On completion of the first phase specification:
  - specify and implement scenarios not completed or implemented yet.
6.3 Remarks on co-operation

Buhrs
Buhrs activities mainly concerned (in my graduation period) of transferring knowledge of current implementation to me and TNO and participation in discussions on the specification of the next generation mailing machine. Buhrs is recommended to start modelling themselves, p.e. main insertion or a scenario of the mailing machine, like re-ordering or town separation in order to be able to continue future development using SHE. To understand essential specification and to realise a specification in POOSL themselves POOSL must be learned.

TNO
Specification of the system info handler is detained due to half year absence of TNO. This specification must be completed including formalisation in POOSL and simulation.

6.4 Recommendations

Conversion from POOSL to C
The POOSL to C mapping is coded in ANSI C. This mapping is a huge bottle-neck in processing speed, realisation in assembler improves processing speed drastically. The number of concurrent tasks can be increased by using a CMX-tiny kernel.

CAN-module
A next generation CAN-module should be extended with the following:
- more inputs, at least 3 for the position encoder (A,B and Z).
- 64 KB RAM instead of 32 KB RAM. To use a large part of it, the address decoder must be extended. Currently input output requires 32 KB addressing space.
Chapter 7

References


[Phi96] Philips Semiconductors, Data Sheet P8xC592 8-bit microcontroller with on-chip CAN, product specification, Philips Semiconductors 1996.

Appendix A

System Level Analysis

A.1 Object Class Diagram

A.1.1 Process classes
Figure A-1 shows abstract process classes, all process classes in the different models are specialised from.

A.1.1.1 Abstract process class Buhrs_Process
All process objects are a sub-class of the abstract process class Buhrs_Process. The name of the class is stored in an attribute ProcessId.

A.1.1.2 Abstract process class Observable_Process
Abstract process class Observable_Process provides behaviour to access instance variables. The value of the instance variable can be obtained and set.

A.1.1.3 Abstract process class Configurable_Process
Abstract process class Configurable_Process provides behaviour required to configure instance variables of a process object.

A.1.1.4 Abstract process Class Service
Abstract process class Service provides basic behaviour required to act on a mailing packet. The class is discussed in appendix 3.

Figure A-1 Overview of process classes
A.1.2 Abstract data class Identifier

Many data objects are used as identifier. An abstract data class \textit{Id} is introduced that incorporates features required for identification:

- a field that stores the identifier, data class \textit{Id} contains an attribute \textit{identifier}
- a message that sets the identifier, by sending a message \textit{setIdentifierTo} with as contents the new identifier attribute \textit{identifier} is set to the new identifier
- a message that obtains the identifier, by sending a message \textit{identifier} to data class \textit{Id}, the \textit{identifier} is returned

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{specialisations of abstract process class \textit{Id}}
\end{figure}

All specialisations of abstract data class \textit{Id} are presented in Figure A-2. The meaning of the sub-classes is discussed in the paragraphs that uses these objects.
A.1.3 Abstract process class Service

Figure A-3 visualises all classes related to abstract process class Service. A Service executes its functionality only when it is required by the mailing packet. Services are uniquely identified by a ServiceId.

Configuration
During booting the Service is assigned a ServiceId by a cfgServiceAssignId(ServiceType, ServiceId) message. The cfgParamId represents the kind of Service a ServiceId is assigned to. The cfgParam is the ServiceId that is assigned. Only one Service of the correct type will receive the message using a conditional receive. Configuration equals configuration of a Configurable_Process, with the exception that the ServiceId is used to address the Service. The cfgService( ServiceId, CfgParam, CfgParamId ) message passes configuration. On completion of configuration a cfgServiceReady( ServiceId ) is send.

Normal operation
On arrival of a mailing packet in the generic transporter all Services are notified of the ArrivalPosition and the PId of the mailing packet with a distributePId( PId, ArrivalPosition ) message. ServicePosition represents the position the Service has to start its functionality. The ArrivalPosition summed with the ServicePosition is the position the Service wants to be waked. The Service sets the position it wants to be waked for the mailing packet by the Wakeup_Unit with a wakeAt( ServiceId, PId, ArrivalPosition ). The ServiceId must be provided to identify the Service to the Wakeup_Unit. When transport reaches the WakePosition, the Service is waked with a wake( ServiceId, PId ) message. Only the Service whose ServiceId is send as content of the message, receive the message with a conditional receive. The PId represents the mailing packet it executes its functionality on. Depending on the ExecuteMode attribute the Service has to inquire its requirement into the Pi_Database. The ExecuteMode attribute represents the following modes:
- off: never execute functionality of Service.
- selective: query Pi_Database if functionality is required.
- stand alone: always execute functionality of Service

In selective mode the Service queries the Pi_Database with an isServiceRequired( ServiceId, PId ) message. The Pi_Database returns an isRequired( ServiceId ) or an isNotRequired( ServiceId ) message whether the Service is required or not. When a Service fails its completion a serviceCompleted( ServiceId, PId ) message is send to the Pi_Database, otherwise on completion of the Service a serviceCompleted( ServiceId, PId ) message is send.
Appendix B

Transporter Model

B.1 Message Flow diagram

B.1.1 Normal operation scenario

Figure B-1 Normal Operation Message Flow Diagram of generic transporter model
B.2.1 Normal Operation Scenario

B.2 Instance Structure Diagram

Figure B.3 Instance Structure Diagram of Genetic Transporter Normal Operation Scenario
B.2.2 Configuration Scenario

Figure B-4 Instance Structure Diagram of generic transporter configuration scenario
### B.3 Object Class Diagram

#### B.3.1 Transporter/Image

Figure B-5 shows all classes related to process class *Transporter/Image*. The *Transporter/Image* images movement of transport. Movement is tracked in a data class *Position*.

![Diagram](image)

**Figure B-5 Classes related to process class Transporter/Image**

#### B.3.1.1 Configuration

The attributes `distributeTransport`, `receiveTransportFrom` and `encoderConnected` must be configured. The boolean attribute `encoderConnected` indicates the presence of a position encoder. If an encoder is not present the attribute `receiveTransport` is the *TransporterId* of the transporter model transport is imaged from. Boolean attribute `distributeTransport` indicates if its transporter is imaged by other transporter models.

#### B.3.1.2 Normal Operation

The *Transporter/Image* translates *fineShiftPulses* from the *Transporter* into a unity of *Position*. The *fineShiftPulse* message indicates movement of transport by one unit distance in the known direction. The direction is set by the *Transporter* with a `forwardDirection` and a `backwardDirection` message. On each unit distance movement forward and backward, a `movedUnitDistanceForward` and a `movedUnitDistanceBackward` message is sent to data class *Position*. Distribution of movement of transport and new cycles is realised by two messages. A `distributeFineShiftPulse(TransporterId, Position)` notifies other generic transporters that the transporter identified by the *TransporterId* moved to a new *Position*. 

-87-
When transport moves on, the Position in the Wakeup_Unit is updated with a movedUnitDistance(Position) message. The current Position can be obtained by receiving a message position(Position).

### B.3.1.3 Position and related classes

Data class *Position* represents the relative position of transport. The difference between two *Positions* is a measure for the distance of product transport. The difference is represented by a data class *PositionDelta*. A *PositionDelta* can be added and subtracted from a *Position* by sending an *add(PositionDelta)* or *minus(PositionDelta)* message to *Position*. Data class *PositionsInStation* represents the maximum number of unit distance of transporter movement to complete a revolution. The number of unit distance transport moves to complete a cycle is represented by a *PositionsInStation* data object. *nmbPosInDegrees* can be send to *PositionsInStation* to convert degrees of a cycle to a *PositionDelta*. A *PositionAbsolute* represents the number of unit distance between the *Position* that indicates the start of the generic transporter and a *Position*. A *ServicePosition* is a specialisation of a *PositionAbsolute*, it represents the number of unit distance between the *Position* that indicates the start of the generic transporter and the *Position* a *Service* must be waked to act on a mailing packet.

Data class *ArrivalPosition* represents a *Position* a *Pil* arrived at the start of the transporter. When an *asArrivalPosition* with content *ServicePosition* is sent to a data class *Position* an *ArrivalPosition* is created. An *ArrivalPosition* is a *Position* minus *ServicePosition*. A *WakePosition* is a specialisation of an *ArrivalPosition* and represents the *Position* a *Service* is triggered. When an *asWakePosition* message with content *ServicePosition* is send to an *ArrivalPosition* an *WakePosition* is created. A *WakePosition* is the *ArrivalPosition* added by the *ServicePosition*.

*Position* contains an integer attribute *positionNumber* that indicates the number of unit distance the reference point on transport compared to a fixed point on the transporter moved. The transporter notifies *Position* of movement by a *movedUnitDistanceForward* and *movedUnitDistanceBackward* message, the *positionNumber* is update accordingly. By sending a *setToZero* message to *Position*, *positionNumber* is set to Zero. *PositionDelta* contains an integer attribute *deltaNumber*. By sending a *setToZero* message *deltaNumber* is set to zero. A *setDeltaNumberTo* message with contents an integer, sets the *deltaNumber* to the value of the integer.

### B.3.2 Pi_Database

The *Pi_Database* consists of a database that is implemented as dictionary. The dictionary uses *Pil* as key to the element *Pi* as visualised in Figure B-6a.

![Object class diagrams of classes related to process Pi_Database](image)

By sending an *isServiceRequired(ServiceId, Pil)* message to the *Pi_Database*, it determines if the *Service* represented by the *ServiceId* must act on the mailing packet represented by the *Pil*. The result is returned to
the Service by returning a \textit{required}(ServiceId, PId) or a \textit{notRequired}(ServiceId, PId) to the Service respective when required or not required. A new Pi is appended to the database by a \textit{piNew}(Pi, PId) message.

Pi consists of a set of \textit{RequiredServices}. A \textit{RequiredService} has a \textit{ServiceStatus} and a \textit{ServiceId}. The \textit{ServiceId} represents the Service required. The \textit{ServiceStatus} contains the status of a Service on the mailing packet represented by the Pi it is a part of. Properties of \textit{ServiceStatus}:

<table>
<thead>
<tr>
<th>status</th>
<th>meaning</th>
<th>set status</th>
<th>test status</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancelled</td>
<td>Service is cancelled</td>
<td>markCancelled</td>
<td>isCancelled</td>
</tr>
<tr>
<td>completed</td>
<td>Service completed successfully on mailing packet</td>
<td>markCompleted</td>
<td>isCompleted</td>
</tr>
<tr>
<td>failed</td>
<td>Service failed on mailing packet</td>
<td>markFailed</td>
<td>hasFailed</td>
</tr>
<tr>
<td>pending</td>
<td>Service is acting on mailing packet</td>
<td>markPending</td>
<td>isPending</td>
</tr>
<tr>
<td>unserviced</td>
<td>Service did not act on mailing packet yet</td>
<td>markUnServiced</td>
<td>isUnserviced</td>
</tr>
</tbody>
</table>

Initially the \textit{ServiceStatus} of a Service on a mailing packet is \textit{unserviced}. The moment a Service queries the Pi\textsubscript{Database} if its action is required, indicates that the Service starts acting on the mailing packet, the \textit{ServiceStatus} changes to \textit{pending}. On completion or fail of a Service on a mailing packet the \textit{ServiceStatus} changes to \textit{completed} or \textit{failed} respective.

### B.3.3 Wakeup\textsubscript{Unit}

Figure B-7 shows an object class diagram with classes related to the Wakeup\textsubscript{Unit}.

![Diagram](image.png)

The \textit{WakeUp_Unit} contains a set of \textit{WakeUpData} in an attribute \textit{wakeUpData}Set. The \textit{WakeUpData} keeps information on what Service must be waked at which \textit{WakePosition} for a certain \textit{PId} in its respective attributes \textit{service}, \textit{WakePosition} and \textit{PId}. A Service programs the \textit{WakeUp_Unit} by sending a \textit{wakeAt}(ServiceId, PId, ArrivalPosition) to the \textit{WakeUp_Unit}. A \textit{transportMoved}(Position) message indicates movement of transport to the Position passed as contents. On each \textit{transportMoved} message the \textit{wakeUpData}Set is checked for Services that want to be waked at the current Position. All Services that want to be waked at that Position are waked by sending a \textit{wake}(ServiceId, PId) to the Services to be waked. The \textit{ServiceId} represents the Service the wake is ment for and the \textit{PId} represents the mailing packet waked for.
Appendix B. Transporter Model

B.3.4 Normal Operation

Interaction diagrams of the normal operation scenario:
1. a mailing packet leaves the transporter
2. a mailing packet passes to consecutive transporter (synchronous coupling)
3. a mailing packet passes to consecutive transporter (a-synchronous coupling)
4. a mailing packet arrives from a side production line
5. a function starts acting

B.3.4.1 A mailing packet leaves a transporter

When a mailing packet moves on to a successor generic transporter, its associated $Pild$ is passed to the generic transporter. In the synchronous coupling case the $Pild$ is passed the moment the mailing packet leaves, in the a-synchronous coupling case the $Pild$ is passed with a fixed $PositionDelta$ delay.

Figure B-9 a production line is notified to start on a mailing packet

1. the $Wakeup\_Unit$ wakes the $Product\_Output\_Handler$ with a $wake$ message for a certain mailing packet
2. the $Product\_Output\_Handler$ queries the $Pi\_Database$ if its action is required on that mailing packet
3. the $Pi\_Database$ responds with a $required$ message when required or $notRequired$ message when not required
4. if required the $Pild$ is passed to a successor generic transporter
5. the *Product_Output_Handler* notifies the *Pi_Database* of the completion on the mailing packet with a *serviceCompleted* message

**B.3.4.2 A mailing packet passes to a consecutive transporter (synchronous coupling)**

As a mailing packet moves on to a consecutive transporter section, its *PId* is passed to the *Product_Input_Handler* of the generic transporter of the receiving transporter section. In the synchronous case the pass of the *PId* indicates the physical arrival of a mailing packet in the transporter model. On arrival all *Services* are notified of the arrival and program the *Wakeup_Unit*. Figure B-10 shows an interaction diagram that shows the responses elected by the pass of the *PId*.

![Interaction Diagram](image)

**Figure B-10 Arrival of mailing packet in transporter section (synchronous case)**

1. A *Predecessor_Transporter* indicates the physical arrival of a mailing packet by sending a message *passPId* to a *Product_Input_Handler*.
2. Immediately current *Position* of the transporter is obtained from the *Transporter_Image*.
3. The *Product_Input_Handler* determines the position the mailing packet arrived and notifies the *PId_distributor* of the arrival of a mailing packet and is position
4. The *Product_Input_Handler* notifies the *Pi_Database* of the completion on the mailing packet with a *serviceCompleted* message.
5. The *Pi_Distributor* notifies all *Services* of the arrival of a mailing packet and its arrival position with a *distributePId* message.
6. The *Services* determine the position they want to be waked. The *Services* program the *Wakeup_Unit* to be waked for the mailing packet arrived wit a *wakeAt* message.

**B.3.4.3 A mailing packet passes to consecutive transporter (a-synchronous coupling)**

In contrast to synchronous coupling, the physical arrival of a mailing packet is indicated by a *productArrived* message from a *Position_Sensor*. The transporter the mailing packet leaves, still passes the *PId* associated with the mailing packet, but after the physical mailing packet left the transporter with a fixed *PositionDelta* delay. The delayed pass enables receiving of empty mailing packets that are not noticed by the *Position_Sensor*. The *Position_Sensor* does not generate a *productArrived* message, the pass of the *PId* indicates the moment of arrival (like in the synchronous coupling case). Figure B-11 shows an combined interaction diagram for both cases.
Appendix B. Transporter Model

Figure B-11 Arrival of a mailing packet in a transporter section (a-synchronous case)

In case the mailing packet has no items, the interaction diagram starts at time stamp 3, which equals the behaviour described in the synchronous case. In case a mailing packet with items arrives, the interaction diagram starts at time stamp 1.

1. A Position_Sensor indicates the complete arrival of a mailing packet by sending a productArrived message.
2. Immediately current Position of the transporter is obtained from the Transporter_Image with a position(Position) message.
3. A Predecessor_Transporter passes the Pild of the mailing packet arrived
4. In case the mailing packet is empty the current Position of the Transporter is obtained.
5. The determination of the ArrivalPosition, notification to the Services of the arrival and setting the WakePosition equals the behaviour from point 3 of the synchronous case.

B.3.4.4 A mailing packet arrives from a side production line

A mailing packet that leaves a side production line is combined with an other mailing packet on a receiving production line. A Side_Product_Input_Handler checks the correct combination of the two mailing packets, both must have the same Pild. If the Pilds differ a match fail occurred. The Side_Product_Input_Handler is waked a PositionDelta after the mailing packet physically arrives.

Figure B-3 A mailing packet arrives of a side production line

1. The Side_Product_Input_Handler receives a representation of a mailing packet from a side production line.
2. The Wakeup_Unit wakes the Side_Product_Input_Handler with a wake message for a certain mailing packet.
3. The Side_Product_Output_Handler queries the Pi_Database if its action is required for that mailing packet.
4. The Pi_Database responds with a required or notRequired message
5. If the received mailing packet representations is not the expected mailing packet representation a match fail occurred
6. The Side_Product_Input_Handler notifies the Pi_Database of the result of the match
B.3.4.5 A function starts acting

Figure B-12 shows a sequence chart of the activation re-activation of a Function.

1. the Wakeup_Unit wakes the Function with a wake message for a certain mailing packet.
2. the Function_Handler queries the Pi_Database if its action is required for that mailing packet.
3. the Pi_Database responds with a required or notRequired message
4. the function is waked for the mailing packet.
5. if the function wants to be waked again when transport moved a number of unit distance, it sends a wakePositionDelta message to the Function_Handler.
6. the Function_Handler passes the messages to the Wakeup_Unit. The Wakeup_Unit determines the WakePosition.
7. the Wakeup_Unit re-wakes the function again.
8. the Function_Handler passes the message to the function.
9. on completion, the function notifies the Function_Handler of the result of its action.
10. the Function_Handler passes the result of the function on the mailing packet to the Pi_Database.
Rotary Feeder

C.1 Message Flow Diagram

C.1.1 Normal Operation Scenario

Appendix C
Function_Handler

Feeder_Config_Handler
configureFunction
cfg
(cfgParamId, cfgParam)
cfgEnd
cfgReady
cfgStart

Feeder_Control
cfgProcessReady(ProcessId)
cfgProcessEnd
cfgProcess(CfgParamId, CfgParam)
cfgProcess(ProcessId)

Feeder_Failure_Control
cfgProcess(CfgParamId, CfgParam)

Rotary_Feeder

Light
C.2 Instance Structure Diagram

C.2.1 Normal Operation Scenario

Figure C-3 Instance Structure Diagram of Normal operation scenario
C.2.2 Configuration scenario

Figure C-4 Instance structure diagram of configuration
C.3 Interaction Diagrams

C.3.1 Normal Operation

Figure C.5 Normal Operation Interaction Diagram
Appendix D

System Info Handler

D.1 Message Flow Diagram

D.1.1 Normal Operation Scenario

Figure D-1 Normal Operation Scenario Message Flow Diagram of the System Info Handler
Appendix E

Formalisation

This paragraph presents POOSL-code\(^{1}\) of the different models.

E.1 Process Objects

```poosl
process class Buhrs_Process(processId: ProcessId)
  /* no superclass */
instance variables
communication channels

message interface

initial method call
initialize()()

instance methods

process class Configurable_Process(processId: ProcessId)
  /* superclass: (Observable_Process) */
instance variables
toConfigure: Set
communication channels
chop, chch
message interface
cchch ? cfg(Object, Object);,
cchch ! cfgProcess(UNKNOWN);
cchch ! cfgReady(UNKNOWN)
initial method call
blaas()()
instance methods
startJob()()
startJobObservable()().

configureProcess()()
| aProcessId : ProcessId, aCfgParamId, aCfgParam : Object |
  chch ! cfgProcess( processId );
  while (toConfigure isEmpty) note) cd
    chch ? cfg( aCfgParamId, aCfgParam ) toConfigure includes( aCfgParamId )
    configure( aCfgParamId, aCfgParam )();
    configured( aCfgParamId )();
    cd;
  chch ! cfgReady( processId ).
configure( cfgParamId : Object, cfgParam : Object )()
skip.
```

\(^{1}\) POOSL retrieved from POOSL40.im
Appendix E. Formalisation

startJobCfg(){}
    normalOperation();

cfgured ( aParamId : Object ){}
toConfigure remove( aParamId ).

**process class** Configuration_Controller(processId: ProcessId)
/* superclass: {Observable_Process} */

**instance variables**

**communication channels**
cccd, scsi, chp, chsi, cmm

**message interface**
chsi ! stationCfg(UNKNOWN, UNKNOWN, UNKNOWN);
cccd ? cfgStationEnd();
scsi ! stationCfgStart(UNKNOWN);
cccd ? cfgStart();
cccd ! cfgReady();
cccd ? cfgServiceParam(ServiceId, Object, Object);
scsi ? stationCfgReady(ServiceId);
cccd ? cfgStationStart(ServiceId);
cccd ? cfgEnd();
scsi ! stationCfgEnd(UNKNOWN);
scsi ? stationCfgNotReady(ServiceId);
chsi ! serviceCfg(UNKNOWN, UNKNOWN, UNKNOWN)

**initial method call**
normalOperation();

**instance methods**
normalOperation();
    | aServiceId : ServiceId; aStationId : StationId; aCfgParamId : Object; aCfgParam : Object;
    stationCfgReady : Boolean; cfgReady : Boolean |
    cccmm ? cfgStart;
    ccccd ! cfgStart;
    cfgReady := false;
    while( cfgReady not() ) od
    les
    ccccd ? cfgStationStart( aStationId );
    scsi ! stationCfgStart( aStationId );
    stationCfgReady := false;
    while( stationCfgReady not() ) od
    delay(1);
    les
    ccccd ? cfgStationParam( aCfgParamId, aCfgParam );
    chsi ! stationCfg( aStationId, aCfgParamId, aCfgParam );
or
    ccccd ? cfgStationEnd;
    stationCfgReady := true;
    scsi ! stationCfgEnd( aStationId );
    les
    scsi ? stationCfgReady( aStationId );
or
    scsi ? stationCfgNotReady( aStationId ); /* perform actions to start configuration
    again for this station*/
or
    ccccd ? cfgServiceParam( aServiceId, aCfgParamId, aCfgParam );
    chsi ! serviceCfg( aServiceId, aCfgParamId, aCfgParam );
or
    ccccd ? cfgEnd;
    cfgReady := true;
    les
    od;
    cccmm ! cfgReady;
    normalOperation();

**process class** Configuration_Database(processId: ProcessId)
Process Objects

/* superclass: {Observable_Process} */

instance variables
serviceConfiguration: Dictionary; stationConfiguration: Dictionary

communication channels
cccd, chcp

message interface
cccd | cfgStationStart(UNKNOWN);
cccd | cfgServiceParam(UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN);
cccd | cfgStart();
cccd | cfgStationParam(UNKNOWN, UNKNOWN);
cccd | cfgEnd();
cccd | cfgStationEnd()

initial method call
init()()

instance methods

normalOperation()()

| configData : Dictionary;
| serviceData : Dictionary;
| cfgParam : Object;
| cfgParamIdCfgParamId:CfgParamId; configSet: Set |
| cccd | cfgStart;

stationConfiguration iterate;

while((stationConfiguration endOfDictionary) not() ) od
  cccd | cfgStationStart( stationConfiguration iterateKey );
  configData := stationConfiguration at( stationConfiguration iterateKey);
  configData iterate;
  while((configData endOfDictionary) not() ) od
    cccd | cfgStationParam( configData iterateKey );
    cccd | cfgStationEnd;
end;

stationConfiguration := new( Dictionary ) init
putAt( new( Dictionary ) init
  putAt( new( PositionsInStation) setDeltaNumberTo(30), "positionsInStation" )
  putAt( true, "isHeadStation")
  putAt( true, "transporterConnected" )
  putAt( new(Set) clear
    insert( new(ServiceId) setIdentifierTo( 10 ), "assignPIH")
    insert( new(ServiceId) setIdentifierTo( 11 ), "assignPOH")
    insert( new(ServiceId) setIdentifierTo( 14 ), "assignSPIH")
    insert( new(StationId) setIdentifierTo(1) ) );

putAt( new( Dictionary ) init
  putAt( new( PositionsInStation) setDeltaNumberTo(30), "positionsInStation" )
  putAt( false, "isHeadStation")
  putAt( false, "transporterConnected" )
  putAt( new(Set) clear
    insert( new(ServiceId) setIdentifierTo( 10 ), "assignPIH")
    insert( new(ServiceId) setIdentifierTo( 11 ), "assignPOH")
    insert( new(ServiceId) setIdentifierTo( 14 ), "assignSPIH")
    insert( new(StationId) setIdentifierTo(2) ) );

.E.1 Process Objects
Appendix E. Formalisation

```typescript
putAt(new Dictionary()) init
putAt( new ( PositionsInStation) setDeltaNumberTo(20), "positionsInStation")
putAt( false, "isHeadStation")
putAt( false,"transporterConnected" )
putAt( new ( StationId ) setIdentifierTo( 1 ),"receiveTransportFrom")
putAt(new Set) clear
insert( new(ServiceId) setIdentifierTo( 30 ),"assignPH")
putAt(new Set) clear
insert( new(ServiceId) setIdentifierTo( 31 ),"assignPOH")
putAt(new Set) clear
insert( new(ServiceId) setIdentifierTo( 32 ),"assignFH")
putAt( new ( Dictionary) init
putAt( new ( ServicePosition) setDeltaNumberTo(0), "servicePosition")
putAt('auto","executeMode")
putAt( true, "isHeadStation")
.new(( ServiceId ) setIdentifierTo( 10 ) )

putAt( new Dictionary()) init
putAt( new ( ServicePosition) setDeltaNumberTo(15), "servicePosition")
putAt( new( ServiceId) setIdentifierTo( 20 ),"successorServiceId")
.putAt( new( ServiceId) setIdentifierTo( 11 ) )

putAt( new Dictionary()) init
putAt( new( ServicePosition) setDeltaNumberTo(0), "servicePosition")
putAt('auto","executeMode")
putAt( false, "isHeadStation")
.new(( ServiceId ) setIdentifierTo( 20 ) )

putAt( new Dictionary()) init
putAt( new( ServicePosition) setDeltaNumberTo(15), "servicePosition")
putAt( new( ServiceId) setIdentifierTo( 20 ),"successorServiceId")
.putAt( new( ServiceId) setIdentifierTo( 21 ) )

putAt( new Dictionary()) init
putAt( new( ServicePosition) setDeltaNumberTo(0), "servicePosition")
putAt('auto","executeMode")
putAt( false, "isHeadStation")
.new(( ServiceId ) setIdentifierTo( 30 ) )

putAt( new Dictionary()) init
putAt( new( ServicePosition) setDeltaNumberTo(10), "servicePosition")
putAt('auto","executeMode")
putAt( false, "isHeadStation")
.new(( ServiceId ) setIdentifierTo( 30 ) )

putAt( new Dictionary()) init
putAt( new( ServicePosition) setDeltaNumberTo(10), "servicePosition")
putAt('auto","executeMode")
putAt( false, "isHeadStation")
.new(( ServiceId ) setIdentifierTo( 30 ) )

process class Config_Handler(processId: ProcessId)
/* superclass: (Observable_Process) */
```

-108-
instance variables
configuring: Set; services: Set

communication channels
chsc, ohop, chsi, chch

message interface
chsi ? stationCfg(StationId, Object, Object);
chch ? cfgReady(ProcessId);
chch ! serviceCfg(UNKNOWN, UNKNOWN, UNKNOWN);
chsc * cfgStart();
chsc * cfgEnd();

chsi ? serviceCfg(ServiceId, Object, Object);
chch ? serviceCfgReady(ServiceId);
chsc * cfg(UNKNOWN, UNKNOWN);
chsc * cfgNotReady();
chsc * cfgEnd();

chch ? assignServiceId(UNKNOWN, UNKNOWN);
ohop ! statusInfo(UNKNOWN, UNKNOWN);
chch ? cfgProcess(ProcessId)

initial method call
init()

instance methods
init()

services := new( Set ) clear;
configuring := new( Set ) clear;
startJobObservable()

getStatus()

observationInfo := new(Dictionary) init
putAt(configuring, "configuring")
putAt(services, "services")
ohop ! statusInfo(processId, observationInfo).

normalOperation()

| aServiceId : ServiceId; aStationId : StationId; aCfgParamId : Object; aCfgParam : Object; |
| cfgReady : Boolean; aProcessId : ProcessId; aServiceId : ServiceId; timeOut: Boolean |
chsc ? cfgStart;
delay(1);
chch * cfgStart;
cfgReady := false;
timeOut := false;
while ((cfgReady not()) & (timeOut not())) od

les
chsi ? stationCfg( aStationId, aCfgParamId, aCfgParam );
if( aCfgParamId = "assignPH" ) & (aCfgParamId = "assignFH") |
(aCfgParamId = "assignSPIH") & (aCfgParamId = "assignPOH") then
chch ! assignServiceId( aCfgParamId, aCfgParam );
services insert( aCfgParam );
configuring insert( aCfgParam );
else
chch * cfg( aCfgParamId, aCfgParam );
fi;
or
chsi ? serviceCfg( aServiceId, aCfgParamId, aCfgParam | services includes( |

aServiceId ) );
chch ! serviceCfg( aServiceId, aCfgParamId, aCfgParam );
or
chsc ? cfgEnd;
chch !* cfgEnd;
cfgReady := true;

| aServiceId : ServiceId; aProcessId : ProcessId |
chsc ? cfgReady( aProcessId );
configuring remove( aProcessId )
or
chch ? serviceCfgReady( aServiceId )
configuring remove( aServiceId )
or
chch ? cfgProcess( aProcessId )
configuring insert( aProcessId )
or
[cfgReady] delay(10); /*set timeOut to generate cfgNotReady*/

les

| cfgReady |
if (timeOut not()) then
chsc ! cfgReady( )
else if (configuring isEmpty) not() ) then
chsc ! cfgNotReady( )
fi;
fi;

normalOperation().

-109-
Appendix E. Formalisation

**process class** DoubleFeed_Detector(processId: ProcessId)

/* superclass: (Observable_Process) */

instance variables

communication channels
chop, ddvv

message interface
ddvv ? feed()

initial method call
normalOperation()

instance methods

normalOperation()

  ddvv ? feed;
  /* random create misfeed */
  normalOperation().

**process class** DoubleFeed_Detector_Image(processId: ProcessId)

/* superclass: (Observable_Process) */

instance variables

communication channels
ddfc, chop, dddi

message interface
dddi ? doubleFeed();
ddfc ! doubleFeedDetected()

initial method call
normalOperation()

instance methods

normalOperation()

  dddi ? doubleFeed;
  ddfc ! doubleFeedDetected;
  normalOperation().

**process class** Event_Handler(processId: ProcessId)

/* superclass: (Observable_Process) */

instance variables

communication channels
ehsi, chop, ehfh

message interface
ehsi ? eventDown(Object, Object);
ehsi ! eventUp(UNKNOWN, UNKNOWN);
ehfh ! eventDown(UNKNOWN, UNKNOWN);
ehfh ? eventUp(Object, Object)

initial method call
init()

instance methods

init()

  ehsi ? eventDown(id,Param);
  ehsi ! eventUp(id,Param);
  ehfh ! eventDown(id,Param);
  ehfh ? eventUp(id,Param);

  normalOperation().

  les;

  normalOperation().

init()

  normalOperation().

**process class** Failure_Control(processId: ProcessId; stationId: Object)

/* superclass: (Configurable_Process) */
instance variables

communication channels
fcsi, fcfh, chch, chlc, chfc

message interface
fcsi ? resetFailure(StationId, Id, Object, Object);
fchf ? failure(Id, Object, Object);
fchf ? resetFailure(UNKNOWN, UNKNOWN, UNKNOWN);
chch ! statusInfo(UNKNOWN, UNKNOWN, UNKNOWN);
fcsi ? failure(UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN)

initial method call
init()

instance methods
init()

toConfigure := new( Set ) clear;

configureProcess();

startJob();

getStatus()
[observationInfo:Object]
observationInfo := new(Dictionary) init
putAt(toConfigure,"toConfigure");
chch ! statusInfo(processId,observationInfo).

normalOperation()
{| failureld : Object; anProcessId : Id; failureParam : Object; aStationld:Stationld|
les

fchf ? failure( anProcessId, failureld, failureParam );
fcsi ? failure( stationId, anProcessId, failureld, failureParam );

or

fcsi ? resetFailure( aStationld, anProcessId, failureld, failureParam |aStationld=stationId);
fchf ? resetFailure( anProcessId, failureld, failureParam );

les;

normalOperation();

process class Feeder_Config_handler(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables
configuring: Set

communication channels
chch, chop, chlc, chfc

message interface
chch ? cfg(Object, Object);
chlc ? configBlinkOn();
chlc ? configBlinkOff();
chch ? configStart();
chch ? configReady();
chch ? configureFunction();
chch ? configProcess(ProcessId);
chch ? configEnd();
chch ? config(UNKNOWN, UNKNOWN);
chch ? configReady(ProcessId);
chch ! statusInfo(UNKNOWN, UNKNOWN)

initial method call
init()

instance methods
init()

configuring := new( Set ) clear;

chch ! configureFunction;

startJobObservable();

getStatus()
[observationInfo:Object]
observationInfo := new(Dictionary) init
putAt(configuring,"configuring");
chch ! statusInfo(processId,observationInfo).

normalOperation() |
| aServiceId : ServiceId; aStationId : StationId; aCfgParamId : Object; aCfgParam : Object; |
| cfgReady : Boolean; aProcessId : ProcessId; aServiceId : ServiceId |
| chch ! configStart;
| chlc ! configBlinkOn;
| cfgReady:=false;
| while( (cfgReady) not() | (configuring isEmpty) not() ) od
Appendix E. Formalisation

```prolog
les
chfc ? cfg( aCfgParamId, aCfgParam );
chch 1* cfg( aCfgParamId, aCfgParam );
or
  chfc ? cfgEnd;
  cfgReady := true;
or
  chch ? cfgReady( aProcessId );
  configuring remove( aProcessId )
or
  chch ? cfgProcess( aProcessId );
  configuring insert( aProcessId );
les;

od;
chfc ! cfgReady( );
chch ! configBlinkOff;
normalOperation().
```

**process class** Feeder_Control(processId: ProcessId)

```prolog
/* superclass: (Configurable_Process) */
```

**instance variables**

- serviceFailed: Boolean
- currentPild: Pild
- misFeedFailEnabled: Boolean
- vacuumValveEnabled: Boolean
- positionsPerCycle: PositionsPerCycle
- doubleFeedFailEnabled: Boolean
- productSensed: Boolean

**communication channels**

- chch
- ddfc
- fcff
- fcvv
- fcmd
- ohop
- cofh

**message interface**

- fcff I correctFeed();
- cofh I serviceCompleted(UNKNOWN);
- fcvv I vacuumOff();
- fcff I doubleFeed();
- cofh I serviceFailed(UNKNOWN);
- ddfc I doubleFeedDetected();
- cofh I wake(Pild);
- fcff I misFeed();
- fcmd I productSensed();
- fcvv I vacuumOn();
- ohop I statusInfo(UNKNOWN, UNKNOWN);
- cofh I wakeDeltaPosition(UNKNOWN, UNKNOWN)

**initial method call**

```prolog
init()()
```

**instance methods**

- disableVacuumValve(){}
  fcvv I vacuumOff;
  vacuumValveEnabled := false.

```prolog
init()() | aPild : Pild |

| toConfigure := new( Set ) clear
  insert( "positionsPerCycle" );
| configureProcess();
| misFeedFailEnabled := false;
| doubleFeedFailEnabled := false;
| /*vacuumValveEnabled := false;*/
| disableVacuumValve();
| serviceFailed := false;
| productSensed := false;

/* create not initialized identifier */
| currentPild := new( Pild ) setIdentifierTo( -1 );

```prolog
startJob()() interrupt {
les
  ddfc ? doubleFeedDetected;
  if( doubleFeedFailEnabled ) then
    disableVacuumValve();
    toff I doubleFeed;
    cofh I serviceFailed( currentPild );
    serviceFailed := true;
  fi;
  or
  fcmd I productSensed;
  productSensed := true;
les;
).

```prolog
normalOperation()() | aPILD : Pild |

| cofh ? wake( aPild ); /* wait for 0 degree */
```

-112-
currentPild := aPild;

enableVacuumValve();
fcvv | vacuumOn;
vacuumValveEnabled := true.

getStatus()
|observationInfo| := new(Dictionary) init
putAt(currentPild,'currentPild')
putAt(doubleFeedFailEnabled,'doubleFeedFailEnabled')
putAt(misFeedFailEnabled,'misFeedFailEnabled')
putAt(positionsPerCycle,'positionsPerCycle')
putAt(serviceFailed,'serviceFailed')
putAt(toConfigure,'toConfigure')
putAt(vacuumValveEnabled,'vacuumValveEnabled');

configure(cfgParamId : Object; cfgParam : Object)()
|cfgParamId := "positionsPerCycle"
|positionsPerCycle := cfgParam.

feed() |
|aPild := Pild |)

/*0 degrees*/

if( (vacuumValveEnabled) note ) then
   enableVacuumValve();
cfhh ! wakeDeltaPosition( currentPild, positionsPerCycle nmbPosInDeg(90) );
fi;
cfhh ? wake( aPild ); /* wait for 90 degrees or 300 degrees */

if( aPild := currentPild ) then
   /* 300 degrees, wakeup of previous Pild */
   if( productSensed not() ) then /* check if a misfeed occurred */
      disableVacuumValve();
cfhh ! misFeed;
cfhh ! serviceFailed( currentPild );
serviceFailed := true;
   /* vacuum valve has been switched off and on because
   of previous misfeed*/
   fi;
   /* serviceFailed := false; /*disable servicefailure for currentPild*/
   if( serviceFailed not() ) then
cfhh ! serviceCompleted( aPild );
   else
      serviceFailed := false;
   fi;
   /* wake( aPild ); */ /* wait for 90 degrees*/

else
   productSensed := false;
   misFeedFailEnabled := true;
   doubleFeedFailEnabled := true;
   cfhh ! wakeDeltaPosition( aPild, positionsPerCycle nmbPosInDeg(300) );
   /*90 degrees*/
   if( productSensed not() ) then /* will always be disabled, no new Pild arrived */
      disableVacuumValve();
   cfhh ! misFeed;
cfhh ! serviceFailed( currentPild );
serviceFailed := true;
   fi;
   if( serviceFailed not() ) then
cfhh ! serviceCompleted( currentPild );
   else
      serviceFailed := false; /*disable servicefailure for currentPild*/
   fi;
   disableVacuumValve();
fi;
/*0 degrees*/
currentPild := aPild;

feed() ()

process class Feeder_Failure_Control(processId: ProcessId)
Appendix E. Formalisation

```plaintext
/* superclass: (Configurable_Process) */

instance variables
currentFailures: Set; stopOnMinLevelDetected: Boolean; misFeedFailCount: Integer; doubleFeedFailCount: Integer; misFeedFailMax: Integer; doubleFeedFailMax: Integer; checkDoubleFeedFailOverflow: Boolean; checkMisFeedFailOverflow: Boolean

communication channels
chch, fffl, fcmd, ohop, cofh

message interface
fcff ? correctFeed();
cofh ! failure(UNKNOWN, UNKNOWN);
fcmd ? storageFilled();
fcff ? doubleFeed();
cofh ! eventUp(UNKNOWN, UNKNOWN);
fcco ? misFeed();
fffl ! flash();
cofh ! errorBlinkOff();
fffl ? minLevelDetected();
cofh ? resetFailure(Object, Object);
fffl ! errorBlinkOn()

initial method call
init()()

instance methods
misFeedFail( )()
if (checkMisFeedFailOverflow ) then
misFeedFailCount := misFeedFailCount + 1;
if( misFeedFailCount >= misFeedFailMax ) then
  cofh ! failure( "misFeedFailOverflow", nil );
  setFailure( "misFeedFailOverFlow" )();
else
  fffl ! flash;
fi;
else
  fffl ! flash;
fi;
cofh ! failure( "misFeedFail", nil ).

init()()
toConfigure := new( Set ) clear
insert( "misFeedFailMax" )
insert( "doubleFeedFailMax" )
insert( "stopOnMinLevelDetected" );
configureProcess()();
misFeedFailCount := 0;
doubleFeedFailCount := 0;
currentFailures := new( Set ) clear;
startJob()().

doubleFeedFail( )()
if (checkDoubleFeedFailOverflow ) then
doubleFeedFailCount := doubleFeedFailCount + 1;
if( doubleFeedFailCount >= doubleFeedFailMax ) then
  cofh ! failure( "doubleFeedFailOverflow", nil );
  setFailure( "doubleFeedFailOverflow" )();
else
  fffl ! flash;
fi;
else
  fffl ! flash;
fi;
cofh ! failure( "doubleFeedFail", nil ).

normalOperation()()
[FailureId:Object;failureParam:Object]
les
  fcco ? misFeed;
misFeedFail( )();
or
  fcco ? doubleFeed;
doubleFeedFail( )();
or
  fcco ? correctFeed;
misFeedFailCount := 0;
doubleFeedFailCount:= 0;
or
  fcmd ? minLevelDetected;
if( stopOnMinLevelDetected ) then
  cofh ! failure( "stopMinLevelDetected", nil );
else
  cofh ! failure( "misFeedFailOverFlow", nil );
```

-114-
E.1 Process Objects

```plaintext
fi;
setFailure( "minLevelDetected" )();
or
or
fcmd ? storageFilled;
resetFailure( "minLevelDetected" )();
cofh ! eventUp( "storageFilled", nil );
or
cohf ?resetFailure(failureId,failureParam);
if ( ((failureId="misFeedFailOverflow")||(failureId="doubleFeedFailOverflow")) then
resetFailure(failureId());
fi;
les;
resetFailure( failureId );
fi.

getFailure( failureId : String )()
currentFailures remove( failureId );
if ( currentFailures isEmpty ) then
  ffff ! errorBlinkOff;
fi.
setFailure( failureId : String )()
if( currentFailures isEmpty ) then
  ffff ! errorBlinkOn; /*Only start blinking when this is first failure*/
currentFailures insert( failureId );

configure( cfgParamId : Object; cfgParam : Object )()
les
  [cfgParamId = "misFeedFailMax"]
  misFeedFailMax := cfgParam;
  if( misFeedFailMax = 0 ) then
    checkMisFeedFailOverflow := false;
  else
    checkMisFeedFailOverflow := true;
  fi
or
  [cfgParamId = "doubleFeedFailMax"]
  doubleFeedFailMax := cfgParam;
  if( doubleFeedFailMax = 0 ) then
    checkDoubleFeedFailOverflow := false;
  else
    checkDoubleFeedFailOverflow := true;
  fi
or
  [cfgParamId = "stopOnMinLevelDetected"]
  stopOnMinLevelDetected := cfgParam;
les.

process class Feeder_Light_Control(processId: ProcessId)
/* superclass: (Configurable_Process) */

instance variables
flashBlink: Boolean; criticalSection: Boolean; configBlink: Boolean; errorBlink: Boolean; lightStatus: String

communication channels
lcli, chch, ohop, ffff, chlc

message interface
chop ! statusInfo(UNKNOWN, UNKNOWN);
cho ! configBlinkOn();
fff ! errorBlinkOff();
chic ! configBlinkOff();
fff ! flash();
fff ! errorBlinkOn();
lcli ! lightOn();
lcli ! lightOff()

initial method call
init()

instance methods
lightToggle()
```
Appendix E. Formalisation

```plaintext
else
    lightOn();
fi.

lightOn()
lightStatus := "on";
ccli ! lightOn.

init()
toConfigure := new( Set ) clear;
lightStatus := "off";
configBlink := false;
errorBlink := false;
flashBlink := false;
criticalSection := false;

startJob() interrupt [criticalSection not()]{
    chlc ? configBlinkOn;
    if( errorBlink | flashBlink | not() ) then
        lightOn();
    fi;
    configBlink := true;
    or
    chlc ? configBlinkOff;
    if( errorBlink | flashBlink | not() ) then
        lightOff();
    fi;
    configBlink := false;
    or
    chlc ? errorBlinkOn;
    if( configBlink | flashBlink | not() ) then
        lightOn();
    fi;
    errorBlink := true;
    or
    chlc ? errorBlinkOff;
    if( configBlink | flashBlink | not() ) then
        lightOff();
    fi;
    errorBlink := false;
    or
    chlc ? flash;
    if( configBlink | errorBlink | not() ) then
        lightOn();
    fi;
    flashBlink := true;
    les

    normalOperation()
    delay();
    criticalSection := true;
    if( configBlink ) then
        lightToggle();
    else
        if( errorBlink ) then
            if( flashBlink ) then
                flashBlink := false; fi;
            criticalSection := false;
        else
            if( flashBlink ) then
                if( flashBlink ) then
                    flashBlink := false; fi;
                criticalSection := false;
                delay();
            else
                if( errorBlink ) then
                    lightToggle(); fi;
            fi;
        fi;
    fi;
    criticalSection := false;
    normalOperation();

lightOff()
lightStatus := "off";
ccli ! lightOff.

getStatus()
[observationInfo:Dictionary] init
observationInfo := new(Dictionary) init
putAt(configBlink, "configBlink")
putAt(criticalSection, "criticalSection")
putAt(errorBlink, "errorBlink")
putAt(flashBlink, "flashBlink")
putAt(lightStatus, "lightStatus")
putAt(toConfigure, "toConfigure");
ohop ! statusinfo(processId,observationInfo).
```

-116-
process class Feeder_Storage(processId: ProcessId; minItemsAvailable: Integer; stationId: StationId)
/* superclass: (Observable_Process) */

instance variables
itemsAvailable: Integer

communication channels
mlm, ohop, operator, stvv

message interface
stvv ! passItem();
operator ? fillStorage(StationId, Integer);
stvv ! ObtainItem();
stvv ! empty();
mlm ! productSensed();
mlm ! productNotSensed()

initial method call
init()()
Appendix E. Formalisation

chch ? assignServiceId(Object, Object);
chfc | cfg(UNKNOWN, UNKNOWN);
chfh | eventUp(UNKNOWN, UNKNOWN);
wuch | wakeAt(UNKNOWN, UNKNOWN, UNKNOWN);
cofh | wake(UNKNOWN);
fcfh | failure(UNKNOWN, UNKNOWN, UNKNOWN);
chch | cfgEnd();
cofh | resetFailure(Object, Object);
chch | cfgEnd();
chch | serviceCfgReady(UNKNOWN);
chch | statusInfo(UNKNOWN, UNKNOWN);

initial method call
init(){}

instance methods
init(){}
toConfigure := new( Set ) clear;
les

chfc ? configureFunction;
configureFunction := true;
toConfigure insert( "Function ");
or
delay( 1 );
configureFunction := false;
les;
serviceConfigure("assignFH"());
serviceStart()();.

normalOperation(){}
| eventId : Object; eventParam : Object; position : Position; aPild : Pild; aDeltaPosition : PositionDelta; failureId : Object; failureParam : Object ; aServiceld:Service1dl

les

cofh ? eventUp(eventId, eventParam);
chfh | eventUp(eventId, eventParam)
or
cofh ? wakeDeltaPosition( aPild, aDeltaPosition );
fhfh | position( position );
wuch | wakeAt( serviceId, aPild, position asWakePosition( aDeltaPosition ) )
or
cofh ? failure( failureId, failureParam );
fcfh | failure( serviceId, failureId, failureParam );
or
fcfh | resetFailure( aServiceId, failureId, failureParam |aServiceId=serviceId);
cofh ? resetFailure( failureId, failureParam );
or
or
cofh ? serviceFailed( aPild );
pich | serviceFailed( serviceId, aPild );
or
or
cofh ? serviceCompleted( aPild );
pich | serviceCompleted( serviceId, aPild );
les;
normalOperation()();

wake( aPild : Pild )()
cofh | wake(aPild).
jobStart(){}
cofh | normalOperation()();

getStatus(){}
| obser:
observationInfo := new(Dictionary) init
| insert( configureFunction, "configureFunction")
| insert( executeMode, "executeMode")
| insert( serviceId, "serviceId")
| insert( servicePosition, "servicePosition")
| insert( toConfigure, "toConfigure");
| statusInfo( observeId, observationInfo).

serviceConfigure( kindOfService : String ){
| aCfgParamId : Object; aCfgParam : Object ; aServiceId : ServiceId; cfgReady:Boolean
| toConfigure insert( "executeMode")
| insert( "servicePosition");
if ( configureFunction ) then
| chch ? cfgStart;
| chfc | cfgStart;
| fi;

chch ? assignServiceId(
aCfgParamId, aCfgParam | aCfgParamId = kindOfService);
| serviceId := aCfgParam;

 cfgReady:=false;
while ( (cfgReady) not() | (toConfigure isEmpty) not() ) od
les

-118-
E.1 Process Objects

set serviceId ;

if( aCfgParamId = "executeMode" ) then
executeMode := aCfgParam;
else
if( aCfgParamId = "servicePosition" ) then
servicePosition := aCfgParam;
else
configure( aCfgParamId, aCfgParam );
fi;
fi;
configured( aCfgParamId )();

chch ? cfgEnd;
if( toConfigure isEmpty ) then
cfgReady := true;
else
cfgReady := true;
fi;
or
[cfgReady=false] chfc ? cfgReady;
toConfigure remove( 'Function ' );
cfgReady := true;
les;

chch ! serviceCfgReady( serviceId ).

configure( cfgParamId : Object; cfgParam : Object )()
chfc := cfg( cfgParamId, cfgParam );
skip.

**process class** Function_RequestPi(processId: ProcessId)

/* superclass: (Observable_Process) */

**instance variables**

communication channels
coh, cofh

message interface
cofh ? wake(Piid)

initial method call
normalOperation()()

**instance methods**

normalOperation()()
| aPiId : PiId |
cofh ? wake( aPiId );
/* will be done by systeminfo handler*/
/*cofh ! eventUp("createPi",aPiId);*/

**process class** Light(processId: ProcessId)

/* superclass: (Observable_Process) */

**instance variables**

lights: String

communication channels
coh, lcli

message interface
lcli ? lightOn();
lcli ? lightOff()

initial method call
init()()

**instance methods**

normalOperation()()

| les |
lcli ? lightOn; light := "on";
or
lcli ? lightOff; light := "off";
les;

init()()
light := "off";
normalOperation()().
process class Machine_Mode_Controller(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables

communication channels
pcmm, ohop, operator, ccmm, mmtr

message interface
mmtr ! transportStop();
operator ! cfgReady();
ccmm ! cfgStart();
ccmm ? cfgReady();
operator ? startCfg();
operator ? jobStart();
operator ? transportStop();
operator ? transportStart();
pcmm ! jobStart();
mmtr ! transportStart();

initial method call
normalOperation();

instance methods
normalOperation();

les
operator ? startCfg;
cmm ? cfgStart;
ccmm ? cfgReady;
operator ? cfgReady;
operator ? jobStart;
pcmm ? jobStart();
or
operator ? transportStop;
mmtr ! transportStop;
or
operator ? transportStart;
mmtr ! transportStart;

les;
normalOperation().

process class Min_Level_Image(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables

communication channels
mImI, ohop, fcmd

message interface
fcmd ! storageFilled();
mImI ? productNotSensed();
fcmd ! minLevelDetected();
mImI ? productSensed()

initial method call
normalOperation();

instance methods
normalOperation();

les
mImI ? productNotSensed;
fcmd ! minLevelDetected;
or
mImI ? productSensed;
fcmd ! storageFilled;
les;
normalOperation().

process class MisFeed_Detector(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables

communication channels
mdvv, ohop, mdmi

message interface
mdvv ? feed();
mdmi ? productNotSensed();
process class MisFeed_Detector_Image(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables

communication channels
ohop, mdmi, fcmd

message interface
mdmi ? productNotSensed();
fcmd ! productNotSensed();
mdmi ! productSensed();

initial method call
initialize()()

instance methods
startJobObservable()()

| aProcessId:ProcessId | normalOperation()()
| interrupt( les

| ohop ? statusGet();
getStatus();;

| or

| ohop ? processStatusGet(aProcessId | aProcessId=ProcessId);
getStatus();;

| les).

getStatus()()
skip.

process class Observation_Handler(processId: ProcessId; stationId: StationId)
Appendix E. Formalisation

/* superclass: (Buhrs_Process) */

instance variables

communication channels
ohsi, ohop

message interface
ohop ? statusInfo(ProcessId, Object);
ohsi ? processStatusGet(StationId, ProcessId);
ohsi ! statusInfo(UNKNOWN, UNKNOWN, UNKNOWN);

ohsi ? stationStatusGet(StationId);

ohsi ! processStatusGet(UNKNOWN);

initial method call
init()()

instance methods

normalOperation()
| aStationId:StationId; aProcessId:ProcessId; UnKnownDataClass:Object |
les

ohsi ? stationStatusGet(aStationId | aStationId=stationId);
ohop !* statusGet();
or

ohop ? statusInfo(aProcessId, UnKnownDataClass);

ohsi ! statusInfo(stationId, aProcessId, UnKnownDataClass);
or

ohsi ? processStatusGet(aStationId, aProcessId | aStationId=stationId);
ohsi !* processStatusSet(aProcessId);

les;
normalOperation().

init()()

normalOperation().

process class Operator()
/* no superclass */

instance variables

storageEmpty: Set; machineStop: Boolean

communication channels

operator

message interface
operator ? resetFailure(UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN);
operator ! jobStart();
operator ! startCfg();
operator ? cfgReady();
operator !* transportStart();
operator ? storageFilled();
operator ! fillStorage(UNKNOWN, UNKNOWN);
operator ? failure(StationId, Id, Object, Object);
operator ? eventUp(StationId, Id, Object, Object);
operator ? statusInfo(StationId, ProcessId, Object);
operator !* transportStop()

initial method call
init()()

instance methods

normalOperation()
| aStationId : StationId |
operator ! startCfg;
operator ? cfgReady;
operator ! jobStart;
delay(1);
operator !* transportStart;
while( true ) od
les

operator ! storageFilled;

"test stationStatusGet"/
"operator ! stationStatusGet(new(StationId) setIdentifierTo(3));" /* test status
info for station 3*/
"operator ! stationStatusGet(new(StationId) setIdentifierTo(2));" /* test status
info for station 2*/
"operator ! stationStatusGet(new(StationId) setIdentifierTo(1));" /* test status
info for station 1*/
or

[ machineStop ]
delay(100);
machineStopped(); /* machine stopped */
or

[ (storageEmpty isEmpty) not() ]
delay(15);

aStationId := storageEmpty get;
init() ()
  | aStationId : StationId; anId : Id; failureId : Object; failureParam : Object; eventid : Object; eventParam : Object; aProcessId: ProcessId;ObservationInfo:Object |
  machineStop := false;
  storageEmpty := new( Set ) clear;
  normalOperation()() interrupt {
    les
    operator ? failure( aStationId, anId, failureId, failureParam );
    les
    [ failureId = "misFeedFailOverflow" ]
      machineStop := true;
      operator ! transportStop;
      delay(20);
      operator ! resetFailure( aStationId, anId, failureId, failureParam );
    les
    or
    [ failureId = "doubleFeedFailOverflow" ]
      machineStop := true;
      operator ! transportStop;
      delay(20);
      operator ! resetFailure( aStationId, anId, failureId, failureParam );
    les
    or
    or
    [ failureId = "minLevelDetected" ] skip; 
      storageEmpty insert( aStationId ); 
    les
    or
    [ failureId = "misFeedFail" ] skip; 
    les
    [ failureId = "doubleFeed Fail" ] skip; 
    les abort ( delay(1) ); error(); } /* non existing failureId */
  le
  / /
  les
  or
  le
  les
  operator ? statusInfo(aStationId,aProcessId,ObservationInfo);

process class Pild_Container(processId: ProcessId)
  /* superclass (Observable_Process) */
  instance variables
    receiveFrom: StationId; container: FIFOBuffer

  communication channels
    pcpi, ohop, pcpo, pcch, pcti

  message interface
    pcpo ? passPid(ServiceId, PiId);
    pcti ? distributeNewCycle(StationId);
    pcpp ? passPid(UNKNOWN, UNKNOWN);
    pcch ? createPi(UNKNOWN);

  initial method call
  init()()
  instance methods
  normalOperation()()
    | aPill : Pill; aServiceId : ServiceId |
    pcpp ? passPid( aServiceId, aPill | aServiceId = new( ServiceId ) setIdentifierTo( 0 ) );
    container write( aPill );
    normalOperation()() .
  init()()
    i := Integer; aPill : Pill; fromStation : StationId |
    container := new( FIFOBuffer ) clear;
    receiveFrom := new( StationId ) setIdentifierTo( 1 );
    i := 0;
    while( i<20) od
    aPill := new( Pill ) setIdentifierTo( i );
    container write( i );
    ir := ir+1;
Appendix E. Formalisation

process class Pild_Distributor(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables

communication channels
chop, pdch, pdpi

message interface
pdch !* distributePild(UNKNOWN, UNKNOWN);
pdpi ? pildArrived(Pild, ArrivalPosition)

initial method call
init()

instance methods
init()
startJobObservable().

normalOperation()()
| arrivedPild : PiId; anArrivalPosition : ArrivalPosition |
pdpi ? pildArrived(arrivedPild, anArrivalPosition); delay(1);
pdch !* distributePild( arrivedPild, anArrivalPosition);

process class Pild_Database(processId: ProcessId)
/* superclass: (Configurable_Process) */

instance variables
pidiDictionary: Dictionary

communication channels
pdsi, pich, chch, chop

message interface
pich ! notRequired(UNKNOWN, UNKNOWN);
pdsi ? piddNew(PiId, ServiceId);
pdsi ? piddCancel(PiId, ServiceId);
pich ? piddServiceFailed(ServiceId, PiId);
pich ? piddServiceCompleted(ServiceId, PiId);
pdsi ? piddUpdateGlobal(PiId, Pi);
pich ? required(UNKNOWN, UNKNOWN);
pich ? isServiceRequired(ServiceId, PiId);
chop ? statusInfo(UNKNOWN, UNKNOWN);
pdsi ? piddNew(PiId, ServiceId)

initial method call
init()

instance methods
init()
| aPi : Pi; aPiId : PiId; aServiceId : ServiceId |
pidiDictionary= new(Dictionary) clear;
normalOperation()()

interrupt
| les
|  pich ? isServiceRequired(aServiceId, aPiId);
|  if (pidiDictionary at(aPiId) isServiceRequired(aServiceId)) then
|  | pich ! required(aServiceId, aPiId);
|  | pidiDictionary at(aPiId) markPending(aServiceId)
|  else
|  pich ! notRequired(aServiceId, aPiId)
| fi

or
pdsi ? updatePiddGlobal(aPiId, aPi);
/* update pi info, but make sure your own pending, failed and completed services are not being changed!! Only change unserviced services, or change services to completed when they were pending or failed, but never change from completed or pending to failed */

or
pdsi ? piddNew(aPiId, aPi);
pidiDictionary putAt(aPi, aPi piId);

or
pdsi ? piddServiceAdd(aPiId, aServiceId);
piDictionary at(aPild) setRequiredService(new(RequiredService)) init

setServiceIdTo(aServiceId); or

psdi ? piServiceCancel(aPild, aServiceId); if (piDictionary at(aPild) isServiceRequired(aServiceId)) then
  /* Service still is unserviced, so we can remove it*/
  piDictionary at(aPild) removeRequiredService(aServiceId) ;
  fi;

getS tas()()

|observationInfo:object|

observationInfo := new(Dictionary) init

putAt(piDictionary,"piDictionary") putAt(toConfigure,"toConfigure");

ohop ! statusInfo(processId,observationInfo).

normalOperation()()

| aServiceId : ServiceId; aPild : Pild |

les

piDictionary at(aPild) markCompleted(aServiceId)
or

piDictionary at(aPild) markFailed(aServiceId);

"* psdi */ updatePiGlobal(aPild,piDictionary at(aPild))="/" /*update Pi info in all other databases*/

les;

normalOperation()()

process class Position_Sensor_Image(configPositionsInStation: PositionsInStation)

/” no superclass “/

instance variables

position: Position

communication channels

pswu, pgps, pips

message interface

pgps ? position(UNKNOWN);
pgps ? newCyclus();
pgps ? fineShiftPulse();
pswu ? position(UNKNOWN);
pips ? newCyclus()

initial method call

init()()

instance methods

normalOperation()()

les

pgps ? fineShiftPulse;
position movedUnitDistance;
pswu ! position(position);
or

pgps ? newCyclus;
pips ! newCyclus;

les;

normalOperation()().

init()()

position := new (Position)

setPositionInStation( configPositionsInStation );

interrupt

pips ! position(position).

process class Product_Input_Handler(processId: ProcessId)

/” superclass: (Service) “/

instance variables

isHeadStation: Boolean

communication channels

chch, pdch, piti, pdpi, pspi, wuch, pich, ohop, pips

message interface

pdpi ? pIdArrived(UNKNOWN, UNKNOWN);
piti ? position(Position);
piti ? newCyclus();
pspi ? passPild(ServiceId, Pild);
pspi ? passPId(PId);
pips ? productSensed();
Appendix E. Formalisation

pips ? productNotSensed();
ohop ! statusInfo(UNKNOWN, UNKNOWN);
pich ! serviceCompleted(UNKNOWN, UNKNOWN);

initial method call
init(){},

instance methods
init(){},
toConfigure := new( Set ) clear
insert( "isHeadStation" );
serviceConfigure( "assignPOH" );
serviceStart();

normalOperation(){},
| aPosition : Position; anArrivalPosition : ArrivalPosition; aPild : PiId; aServiceId : ServiceId | les
| papi ? passPild(aServiceId, aPild | aServiceId = serviceId );
| if( isHeadStation ) then
| piti ? newCyclus();
| fi;
| piti ? position(aPosition)
or
| pips ? productSensed; /*product is entering the station*/
| pips ? productNotSensed; /*product is completely in station*/
| piti ? position(aPosition);
| pips ? passPild(aPild) /*Pild is passed after it left the station*/
les;

anArrivalPosition := aPosition asArrivalPosition( servicePosition );
papi ? pildArrived(aPild, anArrivalPosition);
/* delay(); to make sure the update of the pi in the pi-database is done*/
pich ! serviceCompleted( serviceId, aPild );
normalOperation();

jobStart(){},
normalOperation();

getStatus(){},
| observationInfo:Object
| observationInfo := new(Dictionary) init
| putAt(executeMode, "executeMode")
| putAt(isHeadStation, "isHeadStation")
| putAt(serviceId, "serviceId")
| putAt(servicePosition, "servicePosition")
| ohop ! statusInfo( processId, observationInfo ).

setWakePosition( aPild : PiId; anArrivalPosition : ArrivalPosition ){}

configure( cfgParamId : Object; cfgParam : Object ){}
| [cfgParamId = "isHeadStation" ]
| isHeadStation := cfgParam.

process class Product_Output_Handler(processId: ProcessId)
/* superclass: (Service) */

instance variables
successorServiceId: ServiceId; processId: ProcessId

communication channels
chch, pchch, wuch, pich, ohop, poss

message interface
pocs ? passPild(UNKNOWN, UNKNOWN);
pich ? serviceCompleted(UNKNOWN, UNKNOWN);
ohop ? statusInfo(UNKNOWN, UNKNOWN);

initial method call
init(){},

instance methods
init(){},
processId := "Product_Output_Handler";
toConfigure := new( Set ) clear
insert( "successorServiceId" );
serviceConfigure( "assignPOH" );
serviceStart();

normalOperation(){},
delay(){}
normalOperation();

wake( aPild : PiId ){}
poc ? passPild( successorServiceId, aPild )

-126-
pich ? serviceCompleted( serviceId, aPlId ).

jobStart()()
  normalOperation()().

getStatus()()
  observationInfoObject;
  observationInfo := new(Dictionary) init
  putAt(executeMode, "executeMode")
  putAt(processId, "processId")
  putAt(serviceId, "serviceId")
  putAt(servicePosition, "servicePosition")
  putAt(successorServiceId, "successorServiceId")
  putAt(toConfigure, "toConfigure");
  ohop ! statusInfo(processId, observationInfo).

configure( cfgParamId : Object; cfgParam : Object )()
  [ cfgParamId = "successorServiceId" ]
  successorServiceId := cfgParam.

process class Service(processId: ProcessId)
  /* superclass: (Configurable_Process) */

instance variables
  servicePosition: ServicePosition; executeMode: String; serviceId: ServiceId

communication channels
  pich, wuch, ohop, pdch, chch

message interface
  wuch ? wake(ServiceId, PlId);
  chch ? serviceCfg(ServiceId, Object, Object);
  chch ! serviceCfgReady(UNKNOWN);
  pich ? required(ServiceId, PlId);
  pdch ? distributePlId(PlId, ArrivalPosition);
  pich ? isRequired(ServiceId, PlId);
  wuch ! wakeAt(UNKNOWN, UNKNOWN, UNKNOWN);
  chch ? assignServiceId(Object, Object)

initial method call
  initialize()()

instance methods
  serviceConfigure( kindOfService : String )()
    | aCfgParamId, aCfgParam : Object | aServiceId : ServiceId | toConfigure insert( "executeMode" )
    insert( "servicePosition");
    chch ? assignServiceId(
      aCfgParamId, aCfgParam | aCfgParamId = kindOfService);
    serviceId := aCfgParam;
    while (toConfigure isEmpty) not() od
      chch ? serviceCfg( aServiceId, aCfgParamId, aCfgParam | aServiceId = serviceId );
      if( aCfgParamId = "executeMode" ) then
        executeMode := aCfgParam;
      else
        if( aCfgParamId = "servicePosition" ) then
          servicePosition := aCfgParam;
        else
          configure( aCfgParamId, aCfgParam )();
        fi;
      fi;
      configured( aCfgParamId )();
      od;
    chch ! serviceCfgReady( serviceId ).

setWakePosition( aPlId : PlId; anArrivalPosition : ArrivalPosition )()
  | aWakePosition : WakePosition
  | aWakePosition := anArrivalPosition asWakePosition( servicePosition );
  wuch ! wakeAt(serviceId, aPlId, aWakePosition).

serviceStart()()
  | aPlId : PlId; anArrivalPosition : ArrivalPosition; aServiceId : ServiceId |
  jobStart()()
  interrupt(]
    | aPlId : PlId; anArrivalPosition : ArrivalPosition; aServiceId : ServiceId |
  jobStart()()
  interrupt(
    | aPlId : PlId; anArrivalPosition : ArrivalPosition | aServiceId : ServiceId |
  jobStart()()
  interrupt(
    | aPlId : PlId; anArrivalPosition : ArrivalPosition | aServiceId : ServiceId |

Appendix E. Formalisation

```plaintext

aServiceId serviceId

else

if (executeMode="stand alone") then

fi

process class Sideline_Product_Input_Handler(processId: ProcessId)
/* superclass: (Service) */

instance variables
arrivedPId: PId; productArrived: Boolean

communication channels
chch, pdch, sssp, wuch, pich, ohop

message interface
pich ! serviceFailed(UNKNOWN, UNKNOWN);

initial method call
init()()

instance methods
getStatus()()

getObservationInfo()()

observationInfo := new(Dictionary) init
putAt(arrivedPId, "arrivedPId")
putAt(executeMode, "executeMode")
putAt(productArrived, "productArrived")
putAt(serviceArrived, "serviceArrived")
putAt(servicePosition, "servicePosition")
putAt(toConfigure, "toConfigure")

obs ! statusInfo(processId, observationInfo).

init()()

toConfigure := new( Set ) clear;
serviceConfigure( "assignSPIH" )();

productArrived := false;

serviceStart()().

normalOperation()()

| aServiceId : ServiceId |

| sssp ? passPId( aServiceId, arrivedPId | aServiceId = serviceId ) |

if( productArrived ) then

error()(); /* previous PId not accepted */

fi;

productArrived := true;

normalOperation()().

wake( aPId : PId )()

if( aPId = arrivedPId ) then

pich ! serviceCompleted( serviceId, aPId );

productArrived := false;

else

pich ! serviceFailed( serviceId, aPId ); /* other PId expected */

fi.

jobStart()()

normalOperation()().

process class SI_Event_Handler(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables

communication channels
ohop, operator, what

message interface

```

-128-
operator $ storageFilled();
event $ eventUp(Object, Object)

initial method call
normalOperation()()

instance methods

normalOperation()()

| eventId : Object; eventParam: Object |
| event ? eventUp( eventId, eventParam ) ;
| if (eventId $= "storageFilled") then operator $ storageFilled; fi;
| normalOperation()().

process class SI_Failure_Control(processId: ProcessId)

/** superclass: (Observable_Process) */

instance variables

communication channels
fcsi, ohop, operator

message interface
fcsi $ failure(StationId, Object, Object, Object);
operator ? resetFailure(StationId, Object, Object, Object);
fcsi $ resetFailure(UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN);
operator $ failure(UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN)

initial method call

normalOperation()()

instance methods

normalOperation()()

| stationId : StationId; id: Object; failureId: Object; failureParam : Object |
| fcsi $ failure( stationId, id, failureId, failureParam );
or
| operator $ resetFailure( stationId, id, failureId, failureParam );
fcsi $ resetFailure( stationId, id, failureId, failureParam );
| operator ? resetFailure( stationId, id, failureId, failureParam );
| normalOperation()().

process class SI_Observation_Handler(processId: ProcessId)

/** superclass: (Buhrs_Process) */

instance variables

communication channels
operator, ohsi

message interface
operator $ processStatusGet(StationId, ProcessId);
operator $ stationStatusGet(StationId);
operator $ statusInfo(StationId, ProcessId, Object);
operator $ statusInfo(UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN);
operator $ statusInfo(UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN);
operator $ statusInfo(UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN)

initial method call

init()()

instance methods

init()()

normalOperation()()

| aStationId:StationId; aProcessId:ProcessId; ObservationInfo:Object |
| les
| operator $ stationStatusGet( aStationId); 
| ohsi $ statusInfo( aStationId, aProcessId, ObservationInfo); 
| or
| ohsi $ statusInfo( aStationId, aProcessId, ObservationInfo); 
| operator $ processStatusGet( aStationId, aProcessId, aProcessId); 
| normalOperation()().

process class SI_PID_Controller(processId: ProcessId)
Appendix E. Formalisation

/* superclass: (Observable_Process) */

instance variables
receiveFrom: StationId; container: FIFOBuffer

communication channels
ptti. pcpd, pcpo, pcpi, pcmn, ohop

message interface
pcpo ? passPild(ServiceId, Pild);
pcpd ? piSendNext(UNKNOWN);
pcpi ? passPild(UNKNOWN, UNKNOWN);
pcm ? jobStart();
ptti ? distributeNewCylcus(StationId)

initial method call
init()()

instance methods
init()()
  | i :Integer; aPild : Pild; fromStation : StationId |
  container := new( FIFOBuffer ) clear;
  receiveFrom := new( StationId ) setIdentifierTo( 1 );
  i := 0;
  while ( i<=20 ) od
  aPild := new( Pild ) setIdentifierTo( 1 );
  container write( i );
  i:=i+1;
  od;
  pcmn ? jobStart();
  aPild := container read; /*first Pild should be ready before first new cycle*/
  pcpd ? piSendNext(aPild);
  pcpi ? passPild( new( ServiceId ) setIdentifierTo( 10 ), aPild );
  normalOperation() interrupt { ptti ? distributeNewCylcus( fromStation |
  receiveFrom ) ;
    aPild := container read;
    pcpd ? piSendNext(aPild);
    pcpi ? passPild( new( ServiceId ) setIdentifierTo( 10 ), aPild ) }.
  normalOperation()()

process class SL_Pi_Database(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables

communication channels
pdsi. pcpd, ohop

message interface
pdsi ? pINew(UNKNOWN, UNKNOWN);
pcpd ? piSendNext(Pild)

initial method call
normalOperation()()

instance methods
generatePi( aPild : Pild )()
  aRequiredService : RequiredService; aServiceId : ServiceId; newPi : Pi |
  newPi = new(Pi) init setPild(aPild);
  aServiceId = new( ServiceId );
  aServiceId setIdentifierTo( 10 );
  aRequiredService = new(RequiredService) init setServiceIdTo(aServiceId);
  newPi setRequiredService(aRequiredService);
  aServiceId = new( ServiceId );
  aServiceId setIdentifierTo( 11 );
  aRequiredService = new(RequiredService) init setServiceIdTo(aServiceId);
  newPi setRequiredService(aRequiredService);
  aServiceId = new( ServiceId );
  aServiceId setIdentifierTo( 12 );
  aRequiredService = new(RequiredService) init setServiceIdTo(aServiceId);
  newPi setRequiredService(aRequiredService);
  aServiceId = new( ServiceId );
  aServiceId setIdentifierTo( 13 );
  aRequiredService = new(RequiredService) init setServiceIdTo(aServiceId);
  newPi setRequiredService(aRequiredService);
  aServiceId = new( ServiceId );
  aServiceId setIdentifierTo( 14 );
  aRequiredService = new(RequiredService) init setServiceIdTo(aServiceId);
E.1 Process Objects

...
Appendix E. Formalisation

/* configuration should start over again, jobstart is not allowed because configuration not completed succesfully*/

process class Transporter(positionsPerCycle: PositionsPerCycle; delayTime: Integer)

/* no superclass */

instance variables
direction: String; position: PositionDelta; backwardGeneratorCount: Integer

communication channels

message interface

initial method call

instance methods

transport()

if (position deltaNumber = positionsPerCycle deltaNumber) then
    direction := "Forward";
    if (backwardGeneratorCount > 0) then
        backwardGeneratorCount := backwardGeneratorCount - 1;
        if (position deltaNumber = positionsPerCycle deltaNumber) then
            position setDeltaNumberTo(0);
    fi;
    delay delayTime;
    position nextPosition;
    transport().
fi;

if (backwardGeneratorCount < 0) then
    backwardGeneratorCount := 0;
fi;

normalOperation()()
process class Transporter_image (processId: ProcessId; stationId: StationId)

instance variables
- distributeTransport: Boolean
- position: Position
- isHeadStation: Boolean
- transporterConnected: Boolean
- receiveTransportFrom: StationId

communication channels
- chch, piti, tiwu, chop, fhti, titi, titr

message interface
- piti \ newCyclus();
- fhti \ position(UNKNOWN);
- titr ? fineShiftPulse();
- titi ? distributeNewCyclus(UNKNOWN);
- titi ? distributeNewCyclus(StationId);
- piti \ position(UNKNOWN);
- titr ? distributeFineShiftPulse(UNKNOWN);
- titr ? newCyclus();
- tiwu \ transportMoved(UNKNOWN);
- titr ? backwardDirection();
- titr ? forwardDirection();
- chop \ statusInfo(UNKNOWN, UNKNOWN)

initial method call
- init()

instance methods
- init()

  toConfigure := new( Set ) clear insert( "positionsInStation")
  insert( "isHeadStation" )
  insert( "distributeTransport" )
  insert( "receiveTransportFrom" )
  insert( "transporterConnected" );

- position := new( Position ) init();
- configureProcess();
- startJob() interrupt {
  les piti \ position( position );
  or fhti \ position( position );
  les .
}

- normalOperation() {
  | aStation : StationId |
  if transporterConnected then les
  titr ? forwardDirection;
  forwardDirection(0,0){};
  or titr ? backwardDirection;
  backwardDirection(0,0){};
  les;
  else les
  titi ? distributeFineShiftPulse( aStation | aStation = receiveTransportFrom
  position movedUnitDistance;
  tiwu \ transportMoved(position);
  or titi ? distributeNewCyclus( aStation | aStation = receiveTransportFrom );
  if( isHeadStation ) then
  piti \ newCyclus
  fi
  fi;
  normalOperation().
}

- getStatus()()

observationInfo : Object |
observationInfo := new(Dictionary) init
  putAt(distributeTransport, "distributeTransport")
  putAt(distributeTransport, "distributeTransport")
  putAt(position, "position")
  putAt(receiveTransportFrom, "receiveTransportFrom")
  putAt(toConfigure, "toConfigure")
  putAt(transporterConnected, "transporterConnected");
  chop \ statusInfo(processId, observationInfo);

forwardDirection(positionsBackward: Integer; cyclesBackwardCount: Integer)() {
  | forward: Boolean |
  forward := true;
  if ( (positionsBackward>0) not() ) | (cyclesBackwardCount>0 not()) then les
  titr ? fineShiftPulse;

-133-
Appendix E. Formalisation

positionsBackward := positionsBackward - 1;

or

titr ? newCycle;
cyclesBackwardCount := cyclesBackwardCount - 1;

or

titr ? backwardDirection;
forward := false;

else

titr ? fineShiftPulse;
if distributeTransport then

titr !* distributeFineShiftPulse(stationId);
fi;

position movedUnitDistance;
tiw ! transportMoved(position);

or

titr ? newCycle;
if distributeTransport then

titr !* distributeNewCycle(stationId);
fi;

if (isHeadStation) then

piti ! newCycle
fi;

or

titr ? backwardDirection;
forward := false;

fi;

if (forward) then

forwardDirection(positionsBackward, cyclesBackwardCount);
else

backwardDirection(positionsBackward, cyclesBackwardCount);
fi.

backwardDirection(positionsBackward, Integer; cyclesBackwardCount, Integer)
[backward : Boolean]
backward := true;

les

titr ? fineShiftPulse;
positionsBackward := positionsBackward + 1;

or

titr ? newCycle;
cyclesBackwardCount := cyclesBackwardCount + 1;

or

titr ? forwardDirection;
backward := false;

les;

if (backward) then

backwardDirection(positionsBackward, cyclesBackwardCount);
else

forwardDirection(positionsBackward, cyclesBackwardCount);
fi.

configure(cfgParamId : Object; cfgParam : Object)
les

cfgParamId = "positionsInStation"
position setPositionsInStationTo(cfgParam)
setToZero();

or

cfgParamId = "isHeadStation"
isHeadStation := cfgParam;

or

cfgParamId = "transporterConnected"
transporterConnected := cfgParam;
if (transporterConnected = false) then

configured("distributeTransport")
else

configured("receiveTransportFrom")
fi;

or

cfgParamId = "distributeTransport"
distributeTransport := cfgParam;

or

cfgParamId = "receiveTransportFrom"
receiveTransportFrom := cfgParam;

les.

process class Vacuum_Valve(processId : ProcessId; positionsPerCycle : PositionsPerCycle;
receivePositionFromStation : StationId)
/* superclass: (Observable_Process) */

instance variables
vacuumValveStatus : String; newCycleStarted : Boolean; positionCount : PositionDelta

communication channels
E.1 Process Objects

message interface
stvv ? passItem();
mdvv ! feed();
ddvv ! feed();
vivv ? vacuumOff();
trvv ? distributeFineShiftPulse(Stationld);
mdvv ! misFeed();
stvv ! empty();
vivv ? vacuumOn()

initial method call
init()()

instance methods
init()()
vacuumValveStatus := "off";
newCycleStarted := false;
positionCount := new( PositionDelta ) setDeltaNumberTo( 0 );
normalOperation()().

normalOperation()()
| aStationId : StationId | les

vivv ? vacuumOn;
vacuumValveStatus := "on";
positionCount := new( PositionDelta ) setDeltaNumberTo( positionsPerCycle numbPosInDeg(210) deltaNumber );
newCycleStarted := true;

or
vivv ? vacuumOff;
vacuumValveStatus := "off";
or
{ vacuumValveStatus='on' } trvv ? distributeFineShiftPulse( aStationId | aStationId = receivePositionFromStation );
positionCount := new( PositionDelta ) setDeltaNumberTo( positionCount deltaNumber - 1 );
if (positionCount = new( PositionDelta ) setDeltaNumberTo ( 0 ) ) then
feed()();
positionCount := positionsPerCycle;
fi;
les;
normalOperation()().

feed()() stvv ! obtainItem;
les
stvv ? passItem;
mdvv ! feed;
ddvv ! feed;
or
stvv ? empty;
mdvv ! misFeed
les.

process class Vacuum_Valee_Image(processId: ProcessId)
/* superclass: (Observable_Process) */

instance variables

communication channels
fcvv, ohop, vivv

message interface
fcvv ? vacuumOn();
fcvv ? vacuumOff();
vivv ? vacuumOn();
vivv ? vacuumOff()

initial method call
normalOperation()()

instance methods
normalOperation()()
| les

fcvv ? vacuumOn;
vivv ? vacuumOn;
or
fcvv ? vacuumOff;
vivv ? vacuumOff;
les;
normalOperation()().
Appendix E. Formalisation

**process class** WakeUp_Unit(processId: ProcessId)

/* superclass: (Observable_Process) */

(instance variables)

wakeUpDataList: Set

(message interface)

wuch ! wake(UNKNOWN, UNKNOWN);

orep ! statusInfo(UNKNOWN, UNKNOWN);

wuch ? wakeAt(ServiceId, PiId, WakePosition);

orep ? transportMoved(position)

(initial method call)

init()

(instance methods)

init()

| aServiceId : ServiceId; aPiId : PiId; aWakePosition : WakePosition; aWakeUpData : WakeupData |

| wakeUpDataList := new( Set ) clear; |

| startJobObservable(); |

| interrupt |

| wuch ? wakeAt(aServiceId, aPiId, aWakePosition); |

| aWakeUpData := new(WakeUpData) setElement(aWakePosition, aServiceId, aPiId); |

| wakeUpDataList insert(aWakeUpData). |

getStatus()

| observationInfo := new(Dictionary) init |

| putAt(wakeUpDataList, "wakeUpDataList"); |

| orep ! statusInfo(processId, observationInfo). |

normalOperation()

| aPiId: PiId; aPosition: Position; aWakeUpData: WakeupData |

| orep ? transportMoved(aPosition); |

| aWakeUpData := new(Set) iterate(wakeUpDataList); |

| while (aWakeUpData endOfSet) not() od |

| if (aPosition equals(aWakeUpData element wakePosition) ) then |

| wuch ! wake(aWakeUpData element ServiceId, |

| aWakeUpData element PiId); |

| wakeUpDataList remove( aWakeUpData element ); |

| return( aWakeUpData.next); |

| normalOperation(). |

E.2 Data Objects

data class ArrivalPosition

/* superclass: (Position) */

(instance variables)

asWakePosition(aServicePosition : ServicePosition) : WakePosition

| temp : WakePosition |

| temp := new(WakePosition) init; |

| temp := newSet( inStationTo(positionsInStation)); |

| temp := newSet( positionNumber + aServicePosition deltaNumber); |

| return( temp ). |

data class CfgParamld

/* superclass: (Id) */

(instance variables)

instance variables

data class Dictionary

/* superclass: (Object) */

(instance variables)

instance variables
iterateLink : DictLink; firstLink : DictLink; listDepth : Integer

instance variables
iterateAt : DictLink
return (iterateLink element).

nextKey : DictLink
if (iterateLink == nil) not() then
iterateLink := iterateLink nextLink
fi;
return iterateLink.

putAt(anElement, aKey: Object): Dictionary
if (self includesKey(aKey)) then
self removeKey(aKey)
fi;
firstLink := new(DictLink) setElement(aKey, anElement) setNextLink(firstLink);
listDepth := listDepth + 1;
return self.

clear: Dictionary
listDepth := 0;
firstLink := nil;
return self.

iterate : Dictionary
iterateLink := firstLink;
return self.

removeKey(aKey : Object) : Set
| aLink : Link |
| aLink := firstLink;
firstLink := self removeKeyR(aLink, aKey);
return( self ).

endOfDictionary: Boolean
return (iterateLink •• nil).

isEmpty : Boolean
return listDepth = 0.

removeKeyR(aLink: Link; aKey: Object) : Link
| result : Link |
| result := nil |
| if aLink •• nil then
| result := nil |
| else |
| if aLink key = aKey then
| listDepth := listDepth - 1;
| result := aLink nextLink |
| else |
| aLink setNextLink( self removeKeyR(aLink nextLink, aKey) ) |
| result := aLink |
| fi; |
| fi; |
| return( result ).

includesKey(aKey: Object): Boolean
return self includesKeyR(firstLink, aKey).

at(aKey: Object): Object
return self atR(firstLink, aKey).

atR(aLink: DictLink; aKey: Object) : Object
| anElement : Object |
| anElement := nil |
| if aLink •• nil then |
| anElement := nil |
| else |
| if (aLink key = aKey) then |
| anElement := aLink element; |
| else |
| anElement := self atR(aLink nextLink, aKey) |
| fi; |
| fi; |
| return anElement.

init: Dictionary
listDepth := 0;
firstLink := nil;
return self.

iterateKey: Object
return( iterateLink key ).

includesKey(aLink: DictLink; aKey: Object): Boolean
| result : Object; dictLink : DictLink |
| result := false |
| else |
| if aLink key = aKey then |
| result := true |
| else 

E.2 Data Objects
result := self includesKeyR(aLink nextLink, aKey)
fi;
return result.

data class DictLink
/* superclass: (LinkedListLink) */

instance variables
key: Object

instance variables
setElement(aKey, anElement: Object): DictLink
key := aKey;
 element := anElement;
return self.

key: Object
return key.

data class FIFOBuffer
/* superclass: (Object) */

instance variables
fifoDepth: Integer; firstLink: FIFOLink; lastLink: FIFOLink

instance variables
isEmpty: Boolean
return fifoDepth = 0.

write(anElement: Object): FIFOBuffer
| aLink: FIFOLink |
if fifoDepth = 0 then
  firstLink := new(FIFOLink) setElement(anElement);
  lastLink := firstLink
else
  aLink := new(FIFOLink) setElement(anElement) setNextLink(firstLink);
  firstLink setPreviousLink(aLink);
  firstLink := aLink
fi;
fifoDepth := fifoDepth + 1;
return self.

read: Object
| aLink: FIFOLink |
if fifoDepth = 0 then self error() fi;
 aLink := lastLink;
if fifoDepth = 1 then
  lastLink := lastLink previousLink;
  lastLink setNextLink(nil)
else
  firstLink := nil;
  lastLink := nil;
fi;
fifoDepth := fifoDepth - 1;
return aLink element.

clear(): FIFOBuffer
fifoDepth := 0;
firstLink := nil;
lastLink := nil;
return self.

data class FIFOLink
/* superclass: (Object) */

instance variables
element: Object; nextLink: FIFOLink; previousLink: FIFOLink

instance variables
setNextLink(aLink: FIFOLink): FIFOLink
nextLink := aLink;
return self.

previousLink: FIFOLink
return previousLink.

nextLink: FIFOLink
return nextLink.
element: Object
return element.
setElement(anObject: Object): FIFOlink
  element = anObject;
  return self.
setPreviousLink(aLink: FIFOlink): FIFOlink
  previousLink = aLink;
  return self.

data class Id
  /* superclass: (Object) */
instance variables
  identifier: Object

instance variables
  identifier: Object
  return identifier.
printString : String
  return (identifier printString).
setIdentifierTo(newId: Object): Id
  identifier = newId;
  return self.

data class LinkedListLink
  /* superclass: (Object) */
instance variables
  element: Object; nextLink: LinkedListLink

instance variables
  element: Object
  nextLink: LinkedListLink
  return element.
setElement(anElement: Object): LinkedListLink
  element = anElement;
  return self.
setNextLink(aLink: LinkedListLink): LinkedListLink
  nextLink = aLink;
  return self.

data class Pi
  /* superclass: (Object) */
instance variables
  piId: PIId; requiredServices: Dictionary

instance variables
  init: PIId
    piId = new(PIID);
    requiredServices = new(Dictionary) clear;
    return self.
markUnserviced( aServiceId : ServiceId ) : Pi
  requiredServices at( aServiceId ) markUnserviced;
  return( self ).
setRequiredService( newRequiredService: RequiredService): Pi
  requiredServices = requiredServices putAt( newRequiredService, newRequiredService serviceId);
  return self.

piId: PIId
  return piId.
markCompleted( aServiceId : ServiceId ) : Pi
  requiredServices at( aServiceId ) markCompleted;
  return( self ).
setPid(newId: PIId): Pi
  piId = newId;
  return self.
markPending( aServiceId : ServiceId ) : Pi
  requiredServices at( aServiceId ) markPending;
Appendix E. Formalisation

```
return ( self ).

setIdentifierTo(newId: PiId) : Pi
  piId setIdentifierTo(newId);
return self.

markCancelled( aServiceId : ServiceId ) : Pi
  requiredServices at( aServiceId ) markCancelled;
return ( self ).

removeRequiredService(aRequiredService: RequiredService): Pi
  requiredServices removeKey( aRequiredService serviceId);
return self.

isServiceRequired( aServiceId: ServiceId): Boolean
  /*return( requiredServices includesKey( aServiceId ) )*/ /* doesn't look at pending services*/
  return( requiredServices at( aServiceId ) required )

markFailed( aServiceId : ServiceId ) : Pi
  requiredServices at( aServiceId ) markFailed;
return ( self ).

data class PiId
  /* superclass: (Id) */

instance variables

instance variables

data class Position
  /* superclass: (Object) */

instance variables

instance variables

| aPosition : Position |
  aPosition := new( Position ) init
  setPositionNumberTo(self positionNumber - p positionNumber);
return ( aPosition ).

setPositionTo( p : Position ) : Position
  aPosition := new( Position ) init
  setPositionNumberTo( p positionNumber + aPositionDelta deltaNumber );
return ( aPosition ).

wakePosition( aPositionDelta :PositionDelta ) : WakePosition
  temp := new( WakePosition ) init;
  temp setPositionNumberTo( positionNumber + aPositionDelta deltaNumber );
return ( temp ).

positionNumber :Integer
return(positionNumber).

printString : String
return( positionNumber asString ).

init : Position
  positionNumber:=0;
  positionsInStation:=new ( PositionsInStation ) init
setDeltaNumberTo(1);
return( self )

setPositionInStationTo( p : PositionsInStation ) : Position
  positionInStation := p;
  positionNumber := positionNumber + (positionsInStation deltaNumber);
return ( self )

equal( p : Position ) : Boolean
  return ( p positionNumber = self positionNumber ) & ( p positionsInStation = self positionsInStation).
```
movedUnitDistance : Position
  positionNumber := positionNumber + 1;
  positionNumber := positionNumber \ (positionsInStation deltaNumber);
  return (self).

asArrivalPosition( aServicePosition : ServicePosition ) : ArrivalPosition
  temp := ArrivalPosition;
  temp := new( ArrivalPosition ) init;
  temp setPositionsInStationTo( positionsInStation );
  temp setPositionNumberTo( positionNumber - aServicePosition deltaNumber );
  return (temp).

setPositionNumberTo( aPositionNumber : Integer ) : Position
  positionNumber := aPositionNumber;
  if (positionNumber < 0) then positionNumber := positionNumber + (positionsInStation deltaNumber); fi;
  positionNumber := positionNumber \ (positionsInStation deltaNumber);
  return (self).

minus( d : PositionDelta ) : Position
  aPosition := new( Position ) init
  aPosition := new( Position ) init
  aPosition setPositionsInStationTo( positionsInStation );
  aPosition setPositionNumberTo( self positionNumber - d deltaNumber );
  return (aPosition).

data class PositionDelta
  /* superclass: (Object) */

instance variables
  deltaNumber : Integer

instance variables
  init() : PositionDelta
    deltaNumber := 0;
    return (self).

nextPosition : PositionDelta
  deltaNumber := deltaNumber + 1;
  return (self).

setDeltaNumberTo( i : Integer ) : PositionDelta
  deltaNumber := i;
  return (self).

deltaNumber() : Integer
  return (deltaNumber).

previousPosition : PositionDelta
  deltaNumber := deltaNumber - 1;
  return (self).

printString : String
  return (deltaNumber asString).

add( d : PositionDelta ) : PositionDelta
  return ( new (PositionDelta) setDeltaNumberTo ( self deltaNumber + d deltaNumber ) ).

data class PositionsInStation
  /* superclass: (PositionDelta) */

instance variables

instance variables
  convertDegrees( degrees : Integer ) : Integer
    return ( (deltaNumber * degrees) div (360) ).

data class PositionsPerCycle
  /* superclass: (PositionDelta) */

instance variables

instance variables
  nmbPosInDeg( nmbDegrees : Integer ) : PositionDelta
    return ( new (PositionDelta) nmbDegrees ) div (360).
data class ProcessId
   /* superclass: (Id) */

instance variables

instance variables

data class RequiredService
   /* superclass: (Object) */

instance variables
   serviceId: ServiceId; status: ServiceStatus

instance variables
   init: RequiredService
      serviceId=new(ServiceId); return self.

markUnserviced : ServiceResult
   status="UnServiced"; return self.

setServiceIdTo(newId: ServiceId): RequiredService
   serviceId=newId; self.markUnserviced; return self.

isPending : Boolean
   return status="Pending".

markCompleted : ServiceResult
   status="Completed"; return self.

required : Boolean
   return (status="UnServiced").

markPending : ServiceResult
   status="Pending"; return self.

unServiced : Boolean
   return status="UnServiced".

markCancelled : ServiceResult
   status="Cancelled"; return self.

serviceId: ServiceId
   return serviceId.

serviceStatus : ServiceStatus
   return status.

markFailed : ServiceResult
   status="Failed"; return self.

data class Serviceld
   /* superclass: (ProcessId) */

instance variables

instance variables

data class ServicePosition
   /* superclass: (PositionDelta) */

instance variables

instance variables

data class ServiceStatus
   /* superclass: (Object) */

instance variables
   status: String

-142-
instance variables
init : ServiceResult
status := "StatusNotInitialized";
return self.

markUnserviced : ServiceResult
status := "Unserviced";
return self.

isPending : Boolean
return status = "Pending".

hasFailed : Boolean
return status = "Failed".

isSuccessful : Boolean
return status = "Successful".

status : String
return status.

markPending : ServiceResult
status := "Pending";
return self.

markCancelled : ServiceResult
status := "Cancelled";
return self.

isUnserviced : Boolean
return status = "Unserviced".

markSucceeded : ServiceResult
status := "Successful";
return self.

markFailed : ServiceResult
status := "Failed";
return self.

data class Set
/* no superclass */

instance variables
iterateLink : LinkedListLink; firstLink : LinkedListLink; listDepth : Integer

instance variables
insert( anElement : Object ) : Set
| aLink : Link |
if (self includes(anElement)) not() then
firstLink := new.LinkedListLink
setNextLink( firstLink );
listDepth := listDepth + 1;
fi;
return self.

nextElement : Object
if (iterateLink == nil) not() then
iterateLink := iterateLink nextLink
fi;
return iterateLink.

clear : Set
listDepth := 0;
firstLink := nil;
return self.

removeR( aLink : Link; anElement : Object ) : Link
| result : Link |
if aLink == nil then
result := nil;
else
if aLink element = anElement then
listDepth := listDepth - 1;
result := aLink nextLink
else
aLink setNextLink( self removeR( aLink nextLink, anElement ) );
result := aLink;
fi;
fi;
return( result ).

iterate : Set
iterateLink := firstLink;
return self.
Appendix E. Formalisation

```plaintext
includesR(aLink: DictLink; anElement: Object): Boolean
  | result: Boolean; dictLink: DictLink|
  if aLink=nil then
    result=false
  else
    if aLink element=anElement then
      result=true
    else
      result=self includesR(aLink nextLink, anElement)
  fi;
  return result.

element : Object
  return iterateLink element.

isEmpty: Boolean
  return listDepth=0.

endOfSet: Boolean
  return (iterateLink=nil).

remove( anElement : Object ) : Set
  | aLink : Link |
  firstLink := self removeR( aLink, anElement);
  return( self ).

firstLink: LinkedListLink
  return firstLink.

includes( anElement : Object): Boolean
  return self includesR(firstLink, anElement).

data class SetIterator
  /* superclass: (Object) */

  instance variables
  currentLink: LinkedListLink; firstLink: LinkedListLink

instance variables
  element(): Object
    return currentLink element.

data class SortedLinkedList
  /* superclass: (Object) */

  instance variables
  listDepth: Integer; firstLink: LinkedListLink

instance variables
  peekFirst( getWakeShiftPulse: Integer ) : Boolean
    | aLink: LinkedListLink; result: Integer |
    if listDepth=0 then result=false
    else
      aLink=firstLink;
```
result := aLink element sortableElement = getWakeShiftPulse;

if fi;
return result.

clear: SortedLinkedList
listDepth := 0;
firstLink := nil;
return self.

insertR(aLink: LinkedListLink; anElement: Object): LinkedListLink
| result: LinkedListLink |
  if aLink nil then
    result := new(LinkedListLink) setElement(anElement) setNextLink(nil)
  else if anElement sortableElement < aLink element sortableElement then
    result := new(LinkedListLink) setElement(anElement) setNextLink(aLink)
  else
    aLink setNextLink (self insertR(aLink nextLink, anElement));
  result := aLink
fi;
return result.

insertR (anElement: Object); SortedLinkedList
firstLink := self insertR(firstLink, anElement);
listDepth := listDepth + 1;
return self.

removeFirst: Object
| aLink: LinkedListLink |
| aLink := firstLink;
| firstLink := aLink nextLink;
| listDepth := listDepth - 1;
| return aLink element.

isEmpty: Boolean
return listDepth == 0.

data class StationId
/* superclass: (Id) */

instance variables

instance variables

data class TransporterId
/* superclass: (Id) */

instance variables

instance variables

data class WakePosition
/* superclass: (ArrivalPosition) */

instance variables

instance variables

data class WakeupData
/* superclass: (Object) */

instance variables

instance variables

instance variables

wakePosition: WakePosition; piId: PiId; serviceId: ServiceId

instance variables

wakePosition: WakePosition
return wakePosition;

sortableElement: Integer
return wakePosition PositionNumber;

piId: PiId
return piId;

setLabel(aWakePosition: WakePosition; aServiceId: ServiceId; aPiId: PiId); Object
  wakePosition := aWakePosition;