Kristoffer Berg Rønning

Evaluation of a Scenario-Based approach to Systems Engineering

June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Master's thesis

2019

Kristoffer Berg Rønning

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Evaluation of a Scenario-Based approach to Systems Engineering

Kristoffer Berg Rønning

Reliability, Availability, Maintainability and Safety (RAMS)
Submission date: June 2019
Supervisor: Antoine Rauzy, MTP

Norwegian University of Science and Technology
Department of Mechanical and Industrial Engineering

Preface

This master's thesis was carried out during the spring of 2019 at the Department of Mechanical and Industrial Engineering, Norwegian University of Science and Technology. It is a part of a two-year international master's program in RAMS (Reliability, Availability, Maintainability and Safety). There is no assumed background of the readers of this report, other than to have some knowledge about RAMS, modelling and system terminology.

Trondheim, 11-06-2019

Kristoffer Berg Rønning

Kristoffer Berg Rønning

Acknowledgment

I would like to thank my supervisor, Professor Antoine Rauzy, for his great help during the writing of this master's thesis. I would like to thank him for always being available for meetings and for his quick feedback to my questions.

K.B.R.

Abstract

The increasing emergence of complexity in engineering systems requires good interaction between the involved stakeholders. Systems engineering is an interdisciplinary approach to develop balanced system solutions that meets diverse stakeholders needs. It is a practice that addresses complex and technologically challenging problems.

Model based systems engineering is an emerging approach to systems engineering where the model of a system is the center of all system engineering activities. The benefits of this approach are many. However, even though most systems are complex and dynamic, there exists few models that are complex or dynamic. They are mostly simple and static. This thesis is focused around ScOLa, a domain specific modelling language that is created with the intention of supporting system architecture studies and make it possible to describe and play scenarios. ScOLa has been conducted to an existing level crossing system, to form an impression and evaluate the benefits and usefulness of this type of modelling.

Through the project and the experiment, knowledge about ScOLa has been acquired. The discussion is focused around what ScOLa offers compared to other types of models in system architecture studies.

Sammendrag

Den økende forekomsten av kompleksitet i tekniske systemer krever godt samspill mellom de involverte interessentene. Industrielt systemdesign (systems engineering) er en tverrfaglig tilnærming til utvikling av balanserte systemløsninger som oppfyller ulike interessenters behov. Det er en praksis som prøver å løse komplekse og teknologisk utfordrende problemer.

Modellbasert industrielt systemdesign (model-based systems engineering) er en voksende tilnærming til industrielt systemdesign hvor modellen av et system er sentrum for alle aktiviteter. Fordelene med denne tilnærmingen er mange. Selv om de fleste systemer er komplekse og dynamiske, finnes det imidlertid få modeller som er komplekse eller dynamiske. For det meste er de enkle og statiske. Denne oppgaven er fokusert rundt ScOLa, et domenespesifikt modelleringsspråk som er opprettet med formålet om å støtte systemarkitekturstudier, og gjøre det mulig å beskrive og spille scenarier. Modellering i ScOLa har blitt utført på en eksisterende planovergang for å danne et inntrykk og vurdere fordelene og nytten av denne typen modellering.

I løpet av dette prosjektet har kunnskap om ScOLa blitt tilegnet. Diskusjonen i slutten av oppgaven er fokusert på hva ScOLa tilbyr sammenlignet med andre typer modeller i systemarkitekturstudier.

Contents

Preface . . . i
Acknowledgment . . . ii
Abstract . . . iii
Sammendrag . . . iv

1 Introduction **1**
1.1 Background . . . 1
1.2 Problem Formulation . . . 2
1.3 Objectives . . . 2
1.4 Limitations . . . 2

2 Systems Engineering **3**
2.1 Systems Engineering . . . 3
2.2 The use of Systems Engineering . . . 4
2.3 Model-Based Systems Engineering . . . 6

3 ScOLa **8**
3.1 Introduction to ScOLa . . . 8
3.2 ScOLa Wizard . . . 14

4 Experimental Study **17**
4.1 The System . . . 17
4.2 Modelling with ScOLa . . . 20
4.2.1 Version 1 . . . 20
4.2.2 Version 2 . . . 23
4.3 Version 3 . . . 27

5 Summary **32**
5.1 Summary . . . 32
5.2 Discussion . . . 33
5.3 Recommendations for Further Work . . . 34

Bibliography **35**

A Acronyms **37**

B BPMN **38**

C Scola Codes **40**

C.1 Water Faucet Model 40
C.2 Version 1 41
C.3 Version 2 43
C.4 Version 3 45

List of Figures

2.1 Committed LCC against time 5

3.1 Hierarchy of blocks 9
3.2 ScOLa model of a water faucet system 10
3.3 ScOLa code of the system 11
3.4 States in ScOLa 11
3.5 Scenario in the water faucet model 13
3.6 Output window in ScOLa Wizard 14
3.7 Process window in ScOLa Wizard 15
3.8 System window in ScOLa Wizard 16
3.9 History window in ScOLa Wizard 16

4.1 The crossing system 18
4.2 The system architecture for version 1 20
4.3 The first four steps of the process in the scenario. (Version1) 21
4.4 Example of a choice-gateway. 22
4.5 The history of a successful scenario. 22
4.6 The railway track divided into five positions. 23
4.7 Domains for each component in version 2 23
4.8 The system architecture in version 2 24
4.9 Example of a movable block. 25
4.10 Example of a movable block in the middle of a scenario. 26
4.11 Example of a movable block at the terminal state. 26
4.12 The system architecture in version 3 28
4.13 Example of a split-gateway. 29
4.14 The updated system architecture for version 3 in the middle of a scenario. 31

B.1 BPMN 39

C.1 Scola Code - Water Faucet 40

C.2 Scola Code - Version 1 42
C.3 Scola Code - Version 2 44
C.4 Scola Code - Version 3 51

List of Tables

3.1 Base types for ports 9

4.1 Design structure matrix for the crossing system 30

Chapter 1

Introduction

1.1 Background

The challenges of the 21st century is met by more effective use of science and technology. Science provides the insight to understand the world, while engineering uses technology to build the systems that meets our needs. The systems must work as they are intended to, be built in time and within a budget, while also being safe and reliable. As the problems are becoming more complex, so are the engineered systems. It is impossible to design one part of a system in isolation without considering the problem and its solution as a whole. (Freng and Freng, 2007) Traditional engineering disciplines do not provide the necessary education and experience to ensure a successful development of large, complex system from initiation to operational use. (Kossiakoff et al., 2008)

The field of systems engineering (SE) aims to deal with the modern complex and multidisciplinary systems by concentrating on the system as a whole. Model-based systems engineering (MBSE) is an emerging approach to systems engineering, where the model is the center of all the systems engineering activities. The benefits of using a model-based approach are many, and includes reduced development time, improved analysis capability, and increased potential for reuse. (Ramos et al., 2012) (Holt et al., 2015) However, even though most systems are complex and dynamic, there are not many models that are complex or dynamic. They are mostly simple and static. (Dekker, 2011)

1.2 Problem Formulation

ScOLa (Scenario-Oriented Language) is a domain specific modelling language created by Professor A. Rauzy. It is created with the intention of supporting system architecture studies and make it possible to describe and play scenarios. (Rauzy, 2018)

Unlike most existing models that describes system architecture statically, is ScOLa a dynamic model that offers the possibility to change the system's structure as scenarios are played. The purpose of this thesis is to form an impression of the benefits and the usefulness of the scenario based approach, ScOLa, to systems engineering.

1.3 Objectives

The thesis is divided in the following way: Firstly, a theoretical background is presented where systems engineering and model based systems engineering is explained. The strengths and weaknesses of different types of models are also discussed. Then, ScOLa is introduced and applied to an existing system for an architectural study to acquire knowledge about ScOLa. The models are described as they are made. Lastly, the results are summarized from the experimental study, and the findings are discussed. The reason for doing the mentioned, is to be able to answer the main objective of this thesis:

- Evaluate ScOLa for its ability to support system architecture studies.

1.4 Limitations

There are limitations to this thesis. ScOLa has only been applied to one concrete system, and has not been studied in detail at a component level. The thesis have been performed by a student, and the background knowledge about the models are based on study context, not from experience in development of systems. The thesis is also based on the student's ability of using ScOLa, and it might not have been used to its optimality. Although, it gives a certain indication of the difficulties in the learning of the modelling language.

Chapter 2

Systems Engineering

This chapter introduces systems engineering, the use of it, and model-based systems engineering.

2.1 Systems Engineering

To define systems engineering, it is necessary to firstly define what a system is. ISO 15288:2015 (2015) describes a system as a "*combination of interacting elements organized to achieve one or more stated purposes.*" They are also clarified as "*man-made, created and utilized to provide products or services in defined environments for the benefit of users and other stakeholders.*"

Systems engineering (SE) is an interdisciplinary approach to develop balanced systems solutions to meet diverse stakeholders needs. It is a practice that addresses complex and technologically challenging problems. The SE process includes activities to establish top-level goals that a system must support, specify the system requirements, synthesize alternative system designs and evaluate the alternatives. The process also includes allocation of requirements to the components, integrating the components into the system, and verifying that the system requirements are satisfied. Having Interdisciplinary teams is an essential part of systems engineering. This is necessary to address the diverse stakeholder perspectives and technical domains to achieve a successful solution. The practice of SE continues to evolve with a focus on dealing with systems as a part of a larger whole. The SE practices are therefore becoming codified in different standards. This is essential to advance and institutionalize the practice across industry domains. (Friedenthal et al., 2012)

The systems engineering perspective is based on systems thinking. Systems thinking recognizes the importance of the whole system, and the importance of the relation of the interrelationships of the system elements to the whole. A systems thinker understands how systems fits into a larger context, how they behave, and how to manage them. Systems thinking arises through discovery, learning, modeling, sensing and talking, to better understand, define and work with systems. (INCOSE, 2015)

Since systems engineering is based on a systems thinking perspective, it differs from other traditional engineering disciplines in several ways. Systems engineering is focused on the system as a whole and its interactions with its environment and other systems. It is not only focused on the engineering design of the system, but also with the external factors. These factors includes the identification of customer needs, the system's operational environment, interfacing systems, and other factors that must be accurately reflected in system requirements documents and accommodated in the system design. (Kossiakoff et al., 2008)

A system engineer is responsible for leading the concept development stage. Critical design decisions in the development stage cannot be based entirely on quantitative knowledge, as in traditional engineering disciplines, but instead, must often rely on qualitative judgements balancing a variety of quantities, and make use of experience from a variety of disciplines. (Kossiakoff et al., 2008)

Systems engineering works as a bridge between the traditional engineering disciplines. The different engineering disciplines needs to be involved in the design and development of the large diversity of elements in a complex system. Each element in the system must function properly in combination with the other elements for the system to perform correctly. The various elements in a system cannot be engineered independently, and then be assembled together to produce a working system. The systems engineers must guide and coordinate the design of each element to assure that the interactions and interfaces between the elements are compatible and supporting. Coordination of elements is especially important when individual elements are designed and supplied by different organizations. (Kossiakoff et al., 2008)

2.2 The use of Systems Engineering

The need for systems engineering is increasing as the complexity in system design is escalating. Kossiakoff et al. (2008, p.3) defines the function of systems engineering to be to "*guide the engineering of complex systems.*" Reducing risk associated with new systems or modification to complex systems is still one of the primary goals of systems engineers. (INCOSE, 2007)

The Defense Acquisition University performed a statistical analysis on projects in the US Department of Defense. They reported that the life cycle cost (LCC) is highly determined by decisions in the earlier phases of a project. Fig 2.1 shows that the design phase of a new system averages 15% of the total LCC. The curve for committed cost illustrates that when 15% of the actual cost has been accrued, 70% of the total LCC have been determined. Errors are less expensive to deal with in the earlier phases, which demonstrates the consequences of taking decisions without the necessary information and analysis. Systems engineering increases the effort performed in the concept and design phase to exceed the percentages in the cumulative step-curve. Thereby, reducing the risk of commitments without the sufficient study. The execution of the various life cycle phases is not linear as illustrated, but the consequences of the decisions is the same. (INCOSE, 2007)

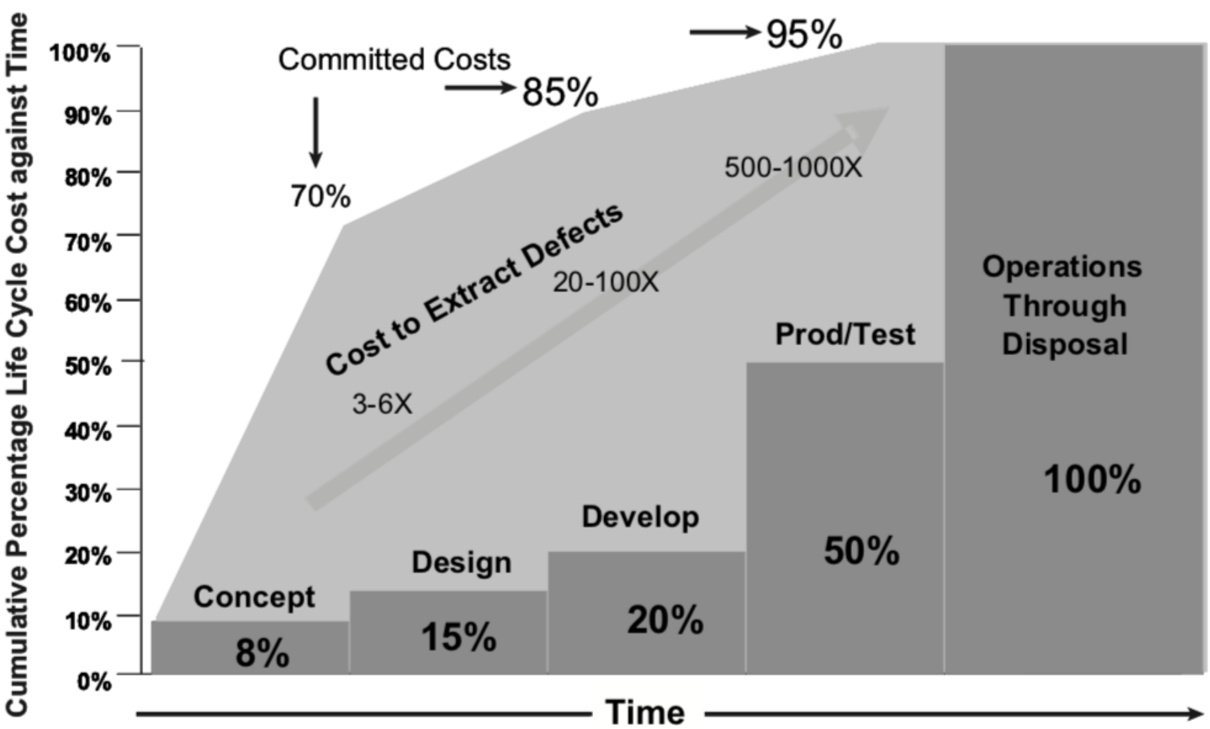


Figure 2.1: *Committed LCC against time. (INCOSE, 2007, p.2.6)*

Another factor to why systems engineering is necessary, is that the time from prototype to market penetration of new products has dropped significantly in the last 50 years. The reason for this is that complexity has an impact on innovation and that there are fewer new product inventions. The products and services are rather a result of incremental improvement, which means that the life cycle of products and services is longer and exposed to increasing uncertainty. Systems engineering processes are crucial to establish and maintain a competitive edge. (INCOSE, 2007)

2.3 Model-Based Systems Engineering

The increase of complexity in systems is demanding more rigorous and formalized systems engineering practices. In response to this demand, the practice of systems engineering undergoes a fundamental transition from a document-based approach to a model-based approach. The attention is shifted from producing and controlling documentation about the system, to producing an controlling a coherent model of the system. Model-based systems engineering (MBSE) can help with managing complexity, improve design quality, improve communications among a diverse development team, and facilitate knowledge capture and design evolution. (Friedenthal et al., 2012)

Systems Modeling Language (OMG SySML™)[1] is a general-purpose modelling language that supports the specification, design, analysis and verification of systems that includes hardware, software, data, personnel, procedures and facilities. It is a graphical modelling language with a semantic foundation that represents the requirements, behaviour, structure and properties of the system and its components. It is intended to model systems from nearly every industry domains. (Friedenthal et al., 2012)

Models and diagramming techniques have been used in the document-based systems engineering approach for years. However, the use of the models has been limited to support specific types of analysis or selected aspects of system design. The respective models have not been integrated into a coherent model of the whole system. Neither have the modelling activities been integrated into the systems engineering process. The transition from document-based SE to MBSE is a shift in emphasis form controlling documentation about the system, to controlling the model of the system. (Friedenthal et al., 2012)

A model is a representation of one or more concepts that can be realized or exists in the physical world. It describes a domain of interest. A model is an abstraction that does not contain every detail of the modeled entities within the domain of interest. Models can be abstract mathematical and logical representations, or concrete physical prototypes. The abstract representation may be a combination of graphical symbols. Such as nodes and arcs on a graph or geometric representations, or text as in a programming language. An example of a model is a blueprint of a building and a prototype physical model. The blueprint is a specification for one or more buildings that are built. It is an abstraction that does not contain all the detail of the building, such as detailed characteristics of its materials. (Friedenthal et al., 2012)

[1]OMG SySML™includes nine types of diagrams. They will not be discussed individually since it is not the scope of this thesis. This also applies to other types of models.

A model expressed in SysML is comparable to a building blueprint that specifies a system to be implemented. Rather than a geometric representation of the system, the SysML model represents the behaviour, properties, structure, constraints and requirements of the system. SySML has a semantic foundation. It specifies the types of model elements and the relationships that can appear in the model. The model elements are stored in a model repository and can be represented graphically. (Friedenthal et al., 2012)

Modelling can support many purposes, such as representing a system concept or specifying system components. A satisfying model meets its intended purpose within the resource constraints of the modelling effort. (Friedenthal et al., 2012)

There exist different type of models, at different levels of abstraction, in different modelling formalisms. It is possible to divide the models in two fundamental categories. Pragmatic models, that primarily supports the communication between stakeholders, and formal models that primarily aims at calculating, simulate or generate artifacts such as computer codes or physical objects. SysML-models written in graphical notation are pragmatic models. As mentioned, their purpose is to facilitate communication, and therefore they keep implicit a lot of knowledge and take a broad outlook on the system under study. Formal models encodes and organizes mathematical equations. The models make everything explicit, by focusing on some specific feature of the system under study. (Rauzy and Haskins, 2018)

The two categories can easily be separated by obfuscation. If the elements in a pragmatic model is renamed to something abstract as X, Y, Z, the model is not understandable, because the model has to refer to the system under study. Stakeholders that share a common knowledge about the system will now struggle with the understanding of the model, since its components have been renamed. By making the same obfuscation for a formal model, nothing changes. The calculations performed on the model will give the same result. Formal models have semantics, meaning that they are interpreted as mathematical objects. Unlike pragmatic models, that are interpretations of the "real" or "physical" world. Pragmatic and formal models have different purposes, but both are useful in system engineering processes. (Rauzy and Haskins, 2018)

Chapter 3

ScOLa

This chapter describes what ScOLa is, what to include in the making of a successful scenario, and an example. The chapter is based on the PowerPoint-presentation, "Scola: a scenario-oriented language. (2018)", made by Professor Antoine Rauzy.

3.1 Introduction to ScOLa

ScOLa is a domain specific modelling language and is an acronym for Scenario-oriented language. It is a textual language that aims at supporting systems architecture studies, by giving the architecture the possibility to describe and play scenarios. Any text editor can be used in the making of the models. (Rauzy, 2018)

ScOLa consists of three important concepts (Rauzy, 2018):

- **System architecture**, which is the decomposition of a system into a hierarchy of connected components.
- **Scenarios**, which are the sequences of actions that is performed on the system, and may reconstruct the system architecture.
- **Processes**, which is the execution of the scenarios.

The model itself is made of two parts. It is a description of the functional or physical decomposition of the system, and a description of scenarios applying on this system. Rauzy (2018)

System architecture The description of the system consists of a hierarchy of blocks, where the top-most block represent the system. Every block can compose any number of sub-blocks as graphically shown in Figure 3.1. (Rauzy, 2018)

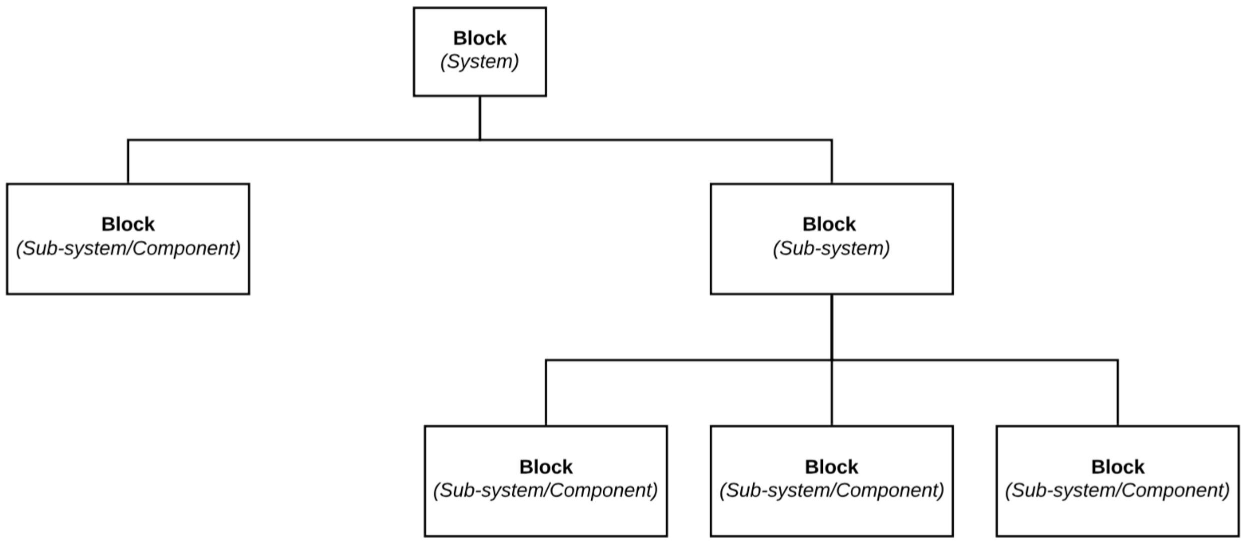


Figure 3.1: *Hierarchy of blocks.*

Each block can include ports and assertions. A port is a holder for an atomic value (Boolean, integer, symbol, string etc.), and an assertion is an instruction that updates the values of these ports. Every block, port and assertion can be dynamically created, moved and removed. (Rauzy, 2018)

The included base types for ports in ScOLa are Boolean, integers, reals, symbols and strings (See Table 3.1). Rauzy (2018)

Base Type	Description
Boolean	True/False
Integer	Any number that can be written without a fractional component
Real	Any non-imaginary number
Symbol	Any symbol (the symbol must belong to a defined domain)
String	A set of characters. Typically used to represent text

Table 3.1: Base types for ports

Note that the symbolic base type is restricted by declaring domains. The value of the symbolic port can only be set to a value included in the domain. For example, for the domain "UnitState" as shown below, the value can be set as either WORKING, FAILED or REPAIR. (Rauzy, 2018)

domain UnitState WORKING, FAILED, REPAIR

The assertions that can be included in the blocks make it possible to describe the connections existing between a system's components. For example energy or flow of matters. The assertions are instructions that updates the values of ports after executions of tasks. (Rauzy, 2018)

A small system consisting of a water supply and a faucet is modelled in ScOLa to show the blocks, ports and assertions. Figure 3.2 shows two screenshots of this model. The blocks are as mentioned representing the system and sub-systems/components. From this point on, every sub-system that is listed as the bottom-most block, will be referred to as a component. Here, the top-most block (the whole system) consists of two sub-blocks, which are the water supply and the water faucet. The water supply block consists of a port with a Boolean base type. It is dependent on whether there is an outflow of water from the tank or not (true or false). The water faucet block consists of three ports. One port with a symbolic base type that defines if the water faucet is open or closed, and two ports with Boolean base types that defines the flow of water. The included assertions make it possible to describe the connections and dependability of the system's components. For example, the inflow to the water faucet is dependant on the outflow from the water supply. Another example is the outflow from the water faucet which is dependant on both the inflow of the faucet and the state of the faucet.

```
block System
  block WaterSupply
    Boolean outFlow true
  end
  block WaterFaucet
    Faucet _state CLOSED
    Boolean inFlow true
    Boolean outFlow false
  end
end
```

(a) Screenshot of the model with a closed faucet.

```
block System
  block WaterSupply
    Boolean outFlow true
  end
  block WaterFaucet
    Faucet _state OPEN
    Boolean inFlow true
    Boolean outFlow true
  end
end
```

(b) Screenshot of the model with an open faucet

Figure 3.2: Two screenshots of the water faucet model.

Figure 3.3 shows the code that defines the system[1]. The first line defines the domain of the faucet, which can be set to either *OPEN* or *CLOSED*. Line 3-18 describes the system and its components. The assertion within the water faucet block (line 11-13) describes the connections within this block. The assertion at line 15-17 describes the connections between the blocks. The ports are selected by stating which block, and then which port, separated by a dot.

[1]The screenshot of the code only shows the system architecture. The code also includes a scenario which is executed by a process, but it is not shown in this figure. The full code can be found in Appendix C.1

```
domain Faucet {OPEN, CLOSED} end

block System
  block WaterSupply
    Boolean outFlow true
  end
  block WaterFaucet
    Faucet _state CLOSED
    Boolean inFlow false
    Boolean outFlow false
    assertion Transfer
      set outFlow (if (eq _state OPEN) inFlow false)
    end
  end
  assertion Transfer
    set WaterFaucet.inFlow WaterSupply.outFlow
  end
end
```

Figure 3.3: *ScOLa code of the water faucet system.*

Scenarios Every scenario can compose any number of sub-scenarios. The scenarios are made of states, tasks and gateways. The states primarily works as the initiator and the completer of scenarios, and can be categorized into three types. Looking at a scenario with a timeline from left to right, they can be described as: (Rauzy, 2018)

- **Initial states**, which are the states that do not occur as the right member of a next directive.
- **Terminal states**, which are the states that do not occur as the left member of a next directive.
- **Intermediate states**, which are the other states.

The states in ScOLa are graphically represented as circles as shown in Figure 3.4. The initial and terminal states are important to include in ScOLa since they define the start and the end of the scenarios. The intermediate states are not necessary to include, but they may contribute to a more clearly scenario. (Rauzy, 2018)

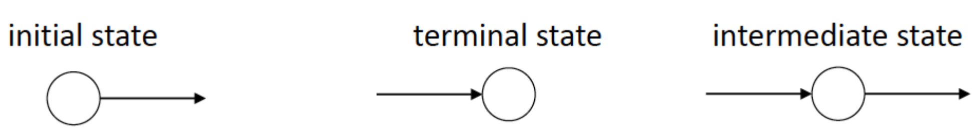


Figure 3.4: *States in ScOLa (Picture retrieved from (Rauzy, 2018, p.26))*

The tasks are containers of instructions in the scenarios, that can modify the system description. Instructions are used in both tasks and assertions, and can be divided into two groups: (Rauzy, 2018)

- Instructions that are set to assign, instructions that are conditional and blocks of instructions. (Can be used both in tasks and assertions)
- Instructions that create, remove and move components.
  (Can only be used in tasks)

A gateway in ScOLa is a choice maker in the scenarios, which makes it possible to define the path of the process. ScOLa provides seven different types of gateways that offers different possibilities: (Rauzy, 2018)

- **Test** - This gateway can have any number of output branches. A process located on the *test*-gateway can only move forward if one and only one of the conditions labeling the branches is verified.
- **Choice** - This gateway can have any number of output branches. A process located on the *choice*-gateway can move forward on any of the output branches.
- **Fork** - This gateway can have any number of output branches. If a process is located on the *fork*-gateway, the process is deactivated, and new processes is created on each the branches. The new processes are not related to the previous process that created them.
- **Join** - This gateway can have any number of input branches. It does the opposite of a *fork*-gateway. When there is a process in each of the input branches, the *join*-gateway can advance. The processes are then deactivated, and a new process is created. If several processes are arriving on an input branch, they are stored into a queue. The first one in, will be the first one out.
- **Split** - This gateway can have any number of output branches. The *split*-gateway is similar to a *fork*-gateway since new processes are started on each branch. However, the *split*-gateway stores the deactivated process (parent process) and links it to the created processes (children processes).
- **Merge** - This gateway can have any number of input branches. It does the opposite of a *split*-gateway. The processes that arrives on the input branches are stored. When every children process of a parent process is located at the *merge*-gateway, they can advance. The children processes are then deactivated and the parent process from the *split*-gateway is reactivated.
- **Meet** - This gateway can have any number of branches. Both input and output branches. The gateway manages incoming processes first, and store them in queues. First one in, is first one out. When there is a process in each input branch, the processes can advance. The processes are then moved to a new location of output branches.

Process The scenarios are executed by processes. A process always starts at the initial state of the scenario and then moves on through the scenario performing every task and gateways until it reaches the terminal state (if there is one). The process can perform a task if it can execute all the instructions of the task. The instructions are performed completely without interruption.

Figure 3.3 shows the scenario of the water faucet system described earlier in this chapter. In the scenario, there are included an initial state and two tasks. The tasks includes an instruction to open (or close) the faucet handle. Line 29-31 shows how the process moves through the scenario. *Next* couples states, tasks and gateways together. The process starts at the initial state and moves to the task *OpenHandle*, which performs the instruction. The process then moves to the next task which is *CloseHandle* and performs the instruction. Note that the scenario does not include a terminal state, and the scenario therefore never has an ending. The process only switches between the two tasks.

```
20  scenario SystemPool as System
21    scenario BathroomFaucet as WaterFaucet
22      state Initial
23      task CloseHandle
24        set _state CLOSED
25      end
26      task OpenHandle
27        set _state OPEN
28      end
29      next Initial OpenHandle
30      next CloseHandle OpenHandle
31      next OpenHandle CloseHandle
32    end
33  end
```

Figure 3.5: *Scenario in the water faucet model*

3.2 ScOLa Wizard

ScOLa Wizard is the software that displays the models. It consists of four windows; *output*, *processes*, *system* and *history*. The water faucet model described in chapter 3.1 is being used.

Output This window shows which model that is being displayed, if the simulation has started or stopped, and possible errors. Figure 3.6 shows the layout of the *output*-window. It shows that the water faucet model is being displayed and that there are zero errors. It also shows that the simulation has started at the initial state.

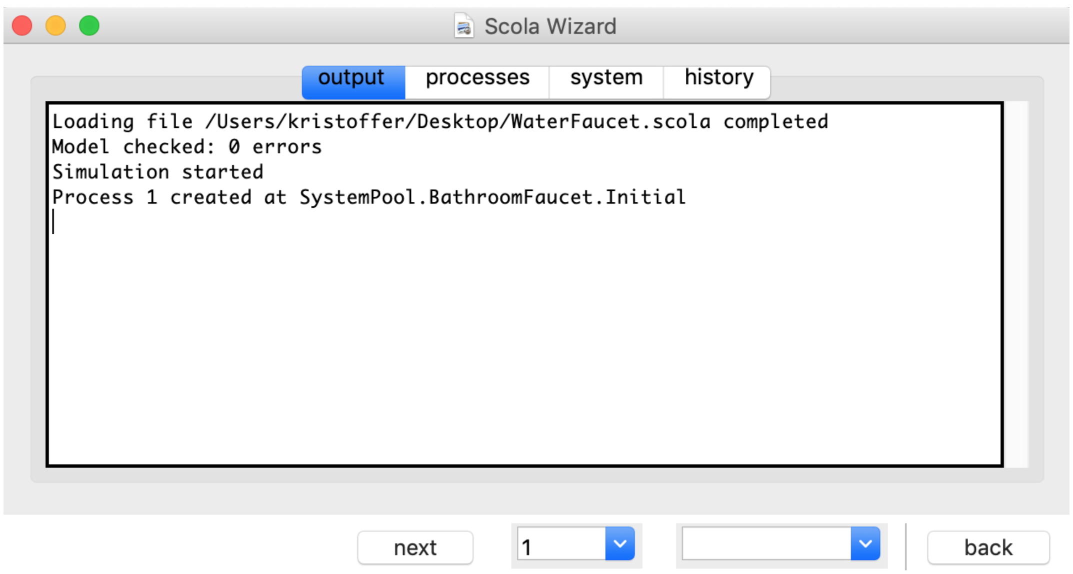
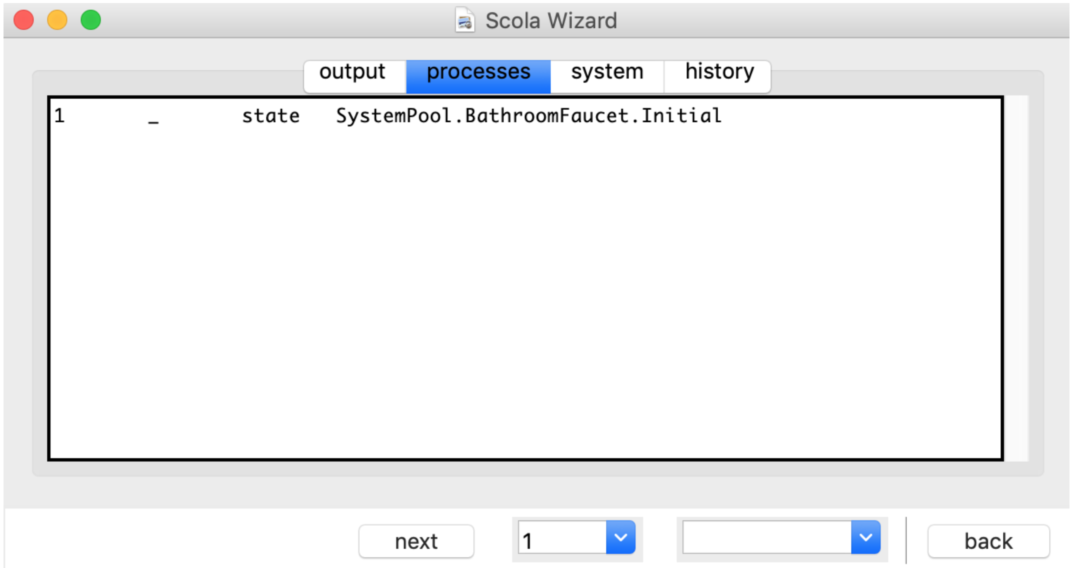


Figure 3.6: *Output window in ScOLa Wizard*

Processes This window shows where the process(es) is/are located in the scenario. Figure 3.7 shows the initial state of the scenario, and the first task which is *OpenHandle*. The process is moved by the use of the *next*-button on the bottom of the screen. If an error is made, it is possible to go back with the *back*-button. The number at the bottom indicates which process is chosen. If a model has several processes, each process has their own number, and it will be possible to choose between them.



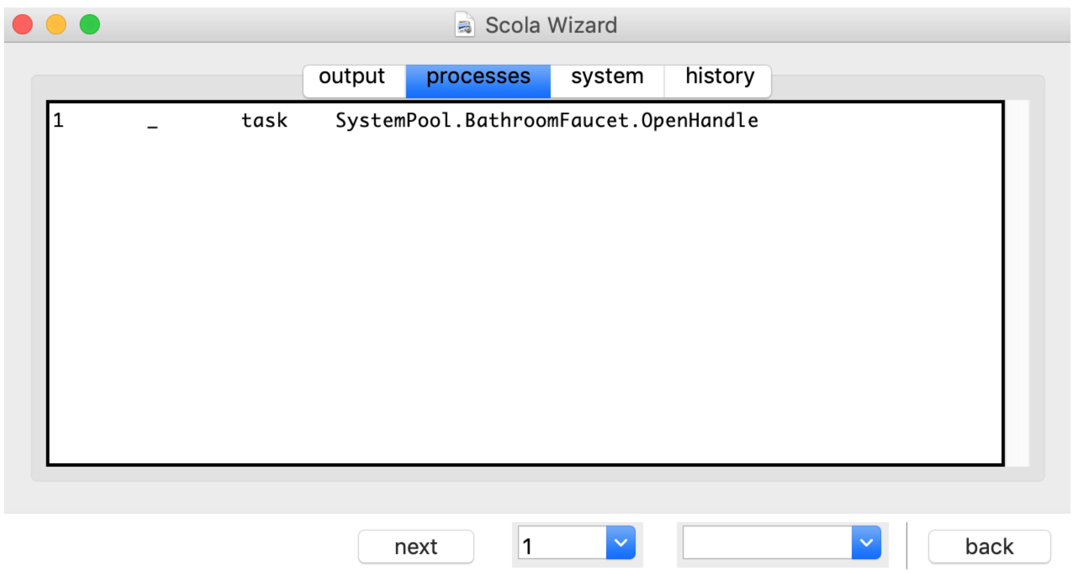


Figure 3.7: *The process window in ScOLa Wizard. The first picture shows the initial state. The second picture shows the next step of the process in the scenario, which is the task to open the handle.*

System This window shows the system architecture. The architecture might change if a scenario is played. It is decided by the position of the process in the scenario. Figure 3.8 shows the water faucet system with a closed faucet.

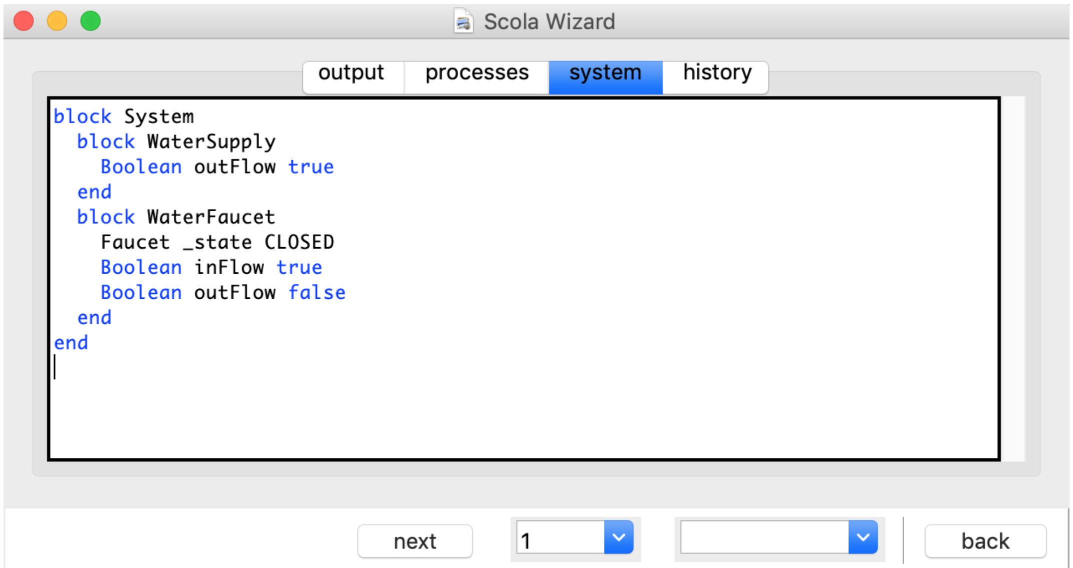


Figure 3.8: *System window in ScOLa Wizard. It shows the whole system and its components, represented by blocks.*

History This window shows the history of each step of the process(es). See Figure 3.9. The history has greater importance when the scenario includes more than only two tasks.

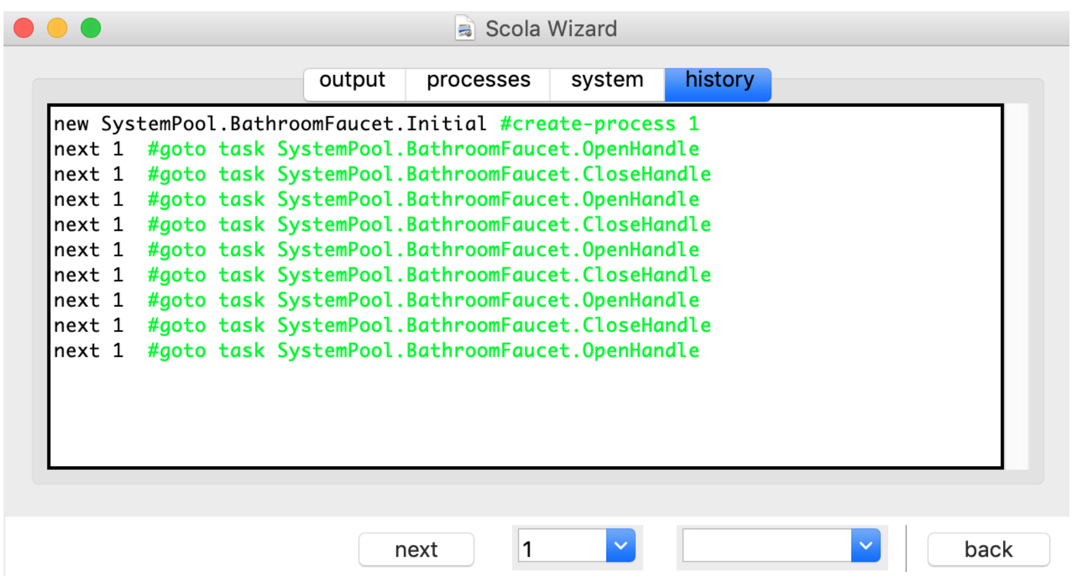


Figure 3.9: *History window in ScOLa Wizard. It shows the whole history of the scenario that has been played, i.e. each step of the process.*

Chapter 4

Experimental Study

This chapter describes a level crossing system modelled with ScOLa. The same system is described throughout the whole chapter, but is gradually changed into more complex versions. This is done with the intention of making it easier for the reader to understand ScOLa and the system itself. The level crossing described in this chapter is based on one of the barrier crossing systems from ORR (2011). Small changes have been done to fit the Norwegian right hand traffic.

4.1 The System

A level crossing is a crossing point between railway traffic and regular road traffic (cars, pedestrians etc.) To ensure a safe interaction for the stakeholders at the level crossing, there exists a safety system. This safety system makes sure that when a train approaches the level crossing, no other traffic is able to cross until the train has passed.

The safety system that enables this safe interaction is here called a crossing system, and consists of light signals for road traffic, an audible warning, four barriers, an obstacle detector and light signals for the railway traffic. Figure 4.1 shows an image of the crossing system. The numbers shows the order of which the events of the system reacts, and are described in detail under the figure.

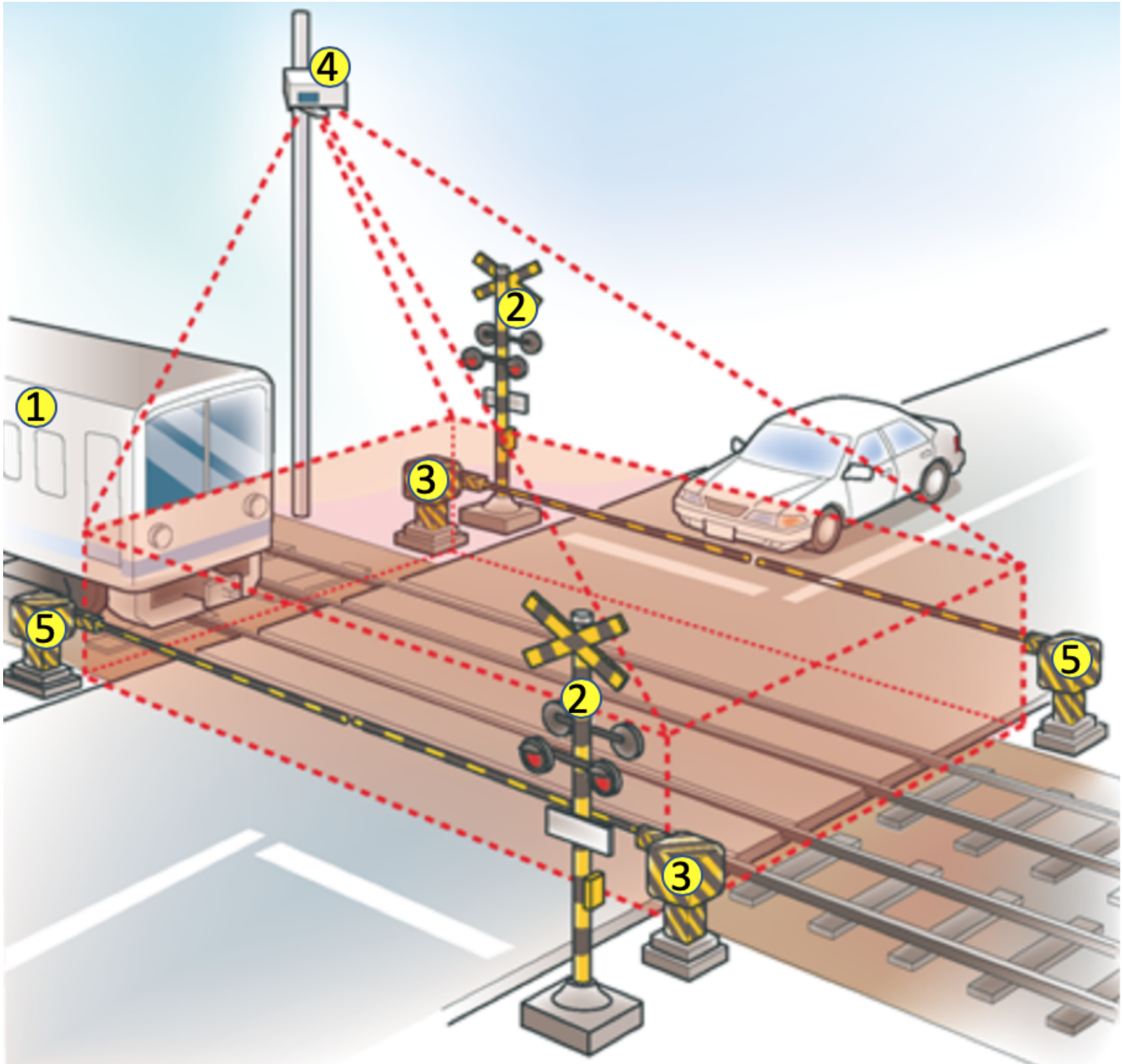


Figure 4.1: *Crossing system with numbers (Picture retrieved from (IHI, 2018))*

The scenario and the order of events of the crossing system are as following: (A BPMN of the same scenario is included in Appendix B for a graphical view.)

- Detection of train (**nr.1**); start sequence of events to close road traffic.
- Traffic lights in both directions switches to amber light, and the audible warning begins (**nr.2**). The light shows for approximately 3 seconds.
- Immediately after the amber light are extinguished, the red light shows.
- Approximately 4 to 6 seconds later, the right hand barriers should start to descend (**nr.3**). The barriers reach the lowered position in 6 seconds.
- After the right hand barriers are lowered, a scan of the crossing area is performed by the

obstacle detector (**nr.4**). If the crossing is clear, the left hand barriers will begin to descend immediately (**nr.5**). If an obstacle is detected, there will be an interval before the left hand barriers starts to descend.

- The audible warning should stop after all the barriers are lowered.
- The crossing is scanned again to check whether the crossing is clear.
- Railway signals gives signal to the train that the passage is clear.
- Barriers rises after the train has passed, and the red light is extinguished as the barriers rise.

4.2 Modelling with ScOLa

This section describes the modelling of the crossing system with ScOLa. The model is updated along with the versions to include a larger amount of components and functions. This means that the first versions does not include every aspect possible in ScOLa, however, they describe the system in a good way.

4.2.1 Version 1

The first version of the model[1] includes the crossing system components and a train. This version can be compared to a BPMN, however, instead of a static graphical view, this version shows a dynamic textual description. The system in this model consists of seven blocks (see figure 4.2). Each block represents a component of the system. The crossing system (whole system) is the the top-most block, while the six other components are sub-blocks, and consists of the train[2], light signals, audible warning, barriers, obstacle detector and railway signals. Figure 4.2 shows a picture of the system architecture in ScOLa and the hierarchy of nested blocks.

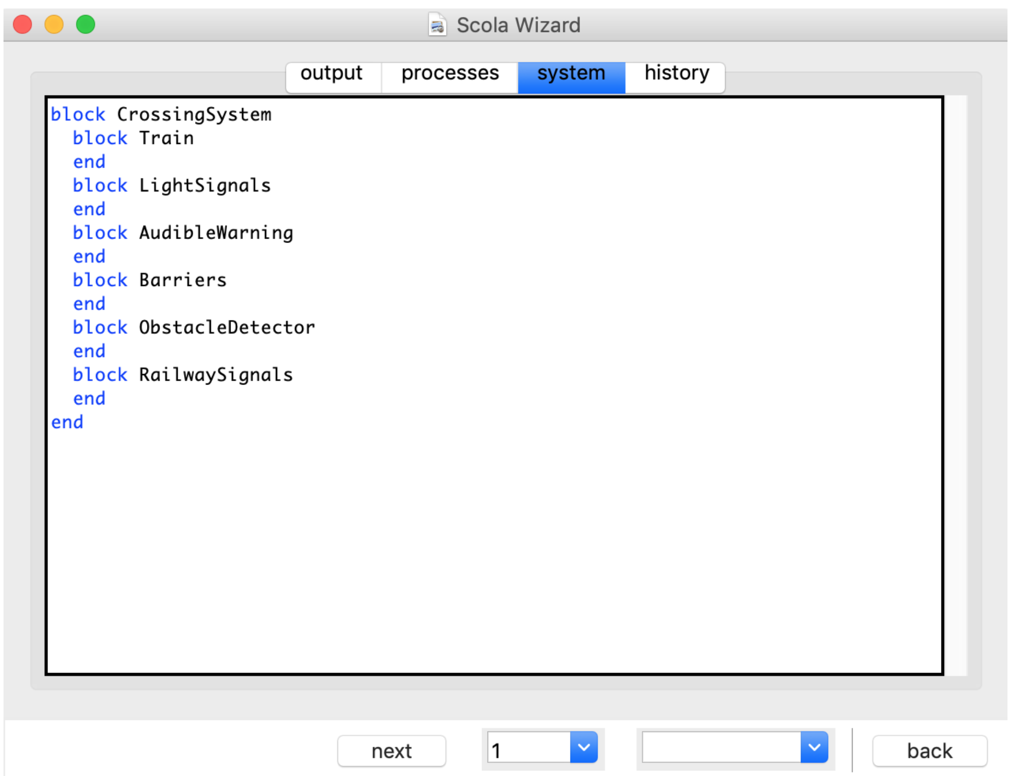


Figure 4.2: *System architecture of the crossing system.*

[1]The ScOLa code can be found in Appendix C.2

[2]It is not entirely correct that the train is a part of the crossing system. It should instead be listed as another system that interacts with the crossing system. In version 2 and 3, the train is an interacting system.

Since the system architecture is defined, it is possible to construct scenarios. The scenarios can then be executed, step by step in ScOLa. The following figures shows the execution of the successful passing of a train as described in chapter 4.1. It is possible to play other scenarios as well, but they are not included in this version. The scenario is named *TrainPassing*[3] and consist of several sub-scenarios, which are the scenarios of each component. The sub-scenarios are named after the components, but with a following *-Lane* at the end. For example, the sub-scenario of the train is named *TrainLane*. This is done with the intention of making the model similar to BPMN. The BPMN of the same scenario can be found in appendix B. The *Lane*-name can be can be compared to the lanes in the BPMN.

The scenario is executed as mentioned in chapter 3.2 by using the *next*-button. Figure 4.3 shows the initial state, and the three following tasks of the scenario. It is possible to read from steps that the light signals switches to amber light after the train has passed a certain point along the tracks.

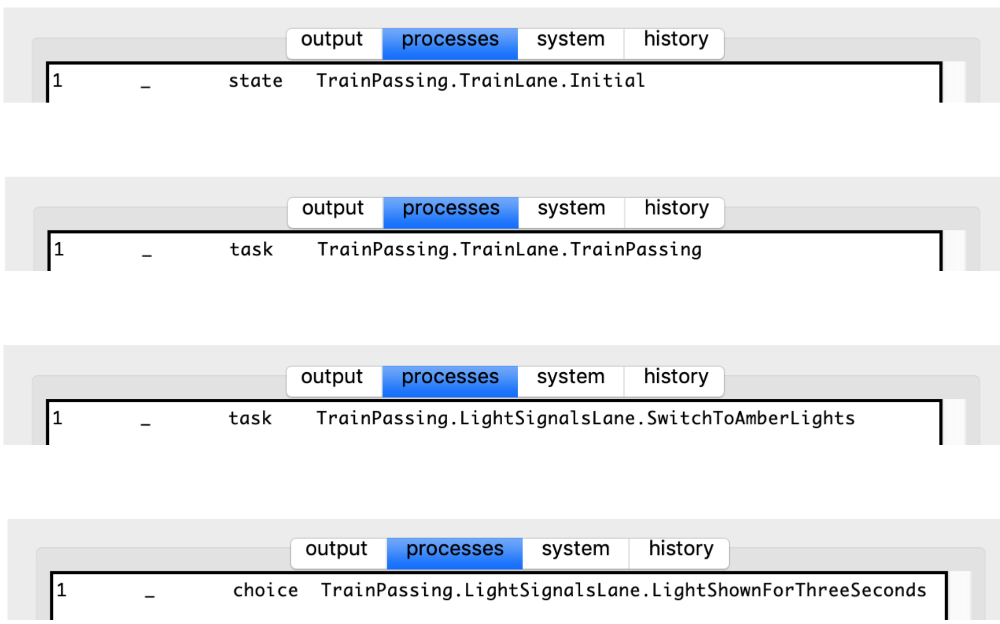


Figure 4.3: *The process' first four steps in the scenario.*

Step four is a *choice*-gateway, where several (in this case two) paths in the scenario are possible. The *choice* that the process has to take here, is whether or not the amber lights have been shown for three seconds.

[3]The name of the scenario (and sub-scenarios) can be set to what is desired.

The *choice* is taken by the use of the *yes/no*-button as shown in Figure 4.4. If the process follows the wrong path, it is possible to go back with the use of the *back*-button and chose another path.

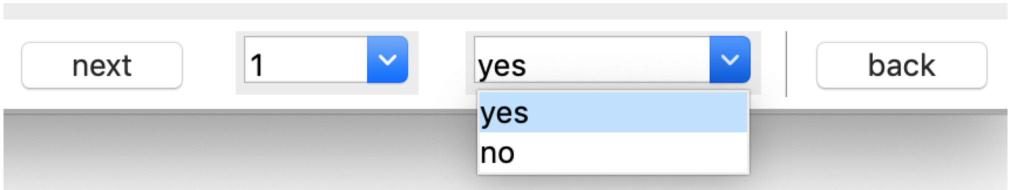


Figure 4.4: *When a gateway is reached in a scenario, there is possible to chose a path. In this case, there are two paths possible, dependent on whether or not the amber lights have been shown for three seconds.*

The history of the scenario of the successful passing of the train is shown in Figure 4.5. Comparing it to the BPMN in appendix B, shows the similarity. The ScOLa model described the scenario dynamically step by step, while the BPMN shows the whole scenario graphically with one picture.

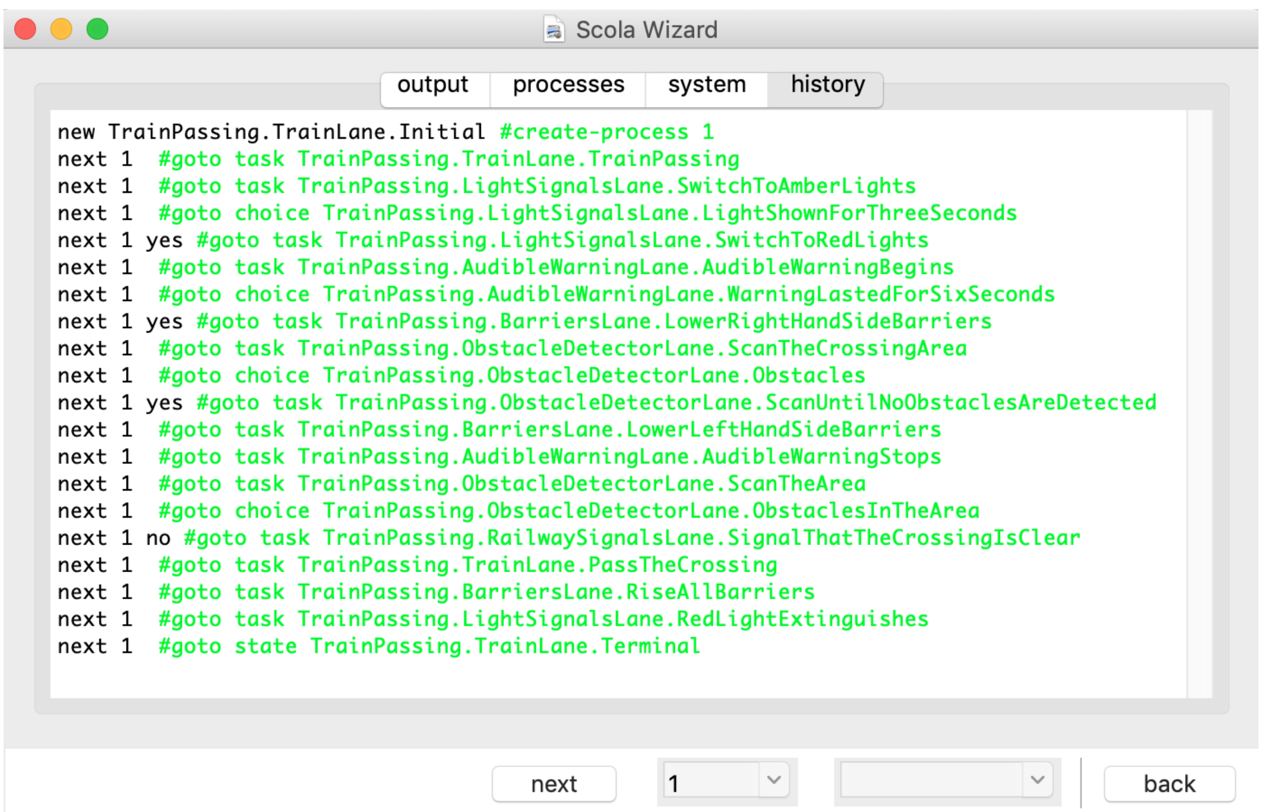


Figure 4.5: *History of a successful train passing scenario.*

4.2.2 Version 2

The second version of the model introduces two other functions of ScOLa. The first function is the possibility of having mobile components, that can be moved from one place to another in the model. The other function is the possibility of updating the states of the components. To display these functions in a proper way, the railway track is now divided into five parts. See Figure 4.6. The crossing system is placed at the level crossing (coloured rectangle).

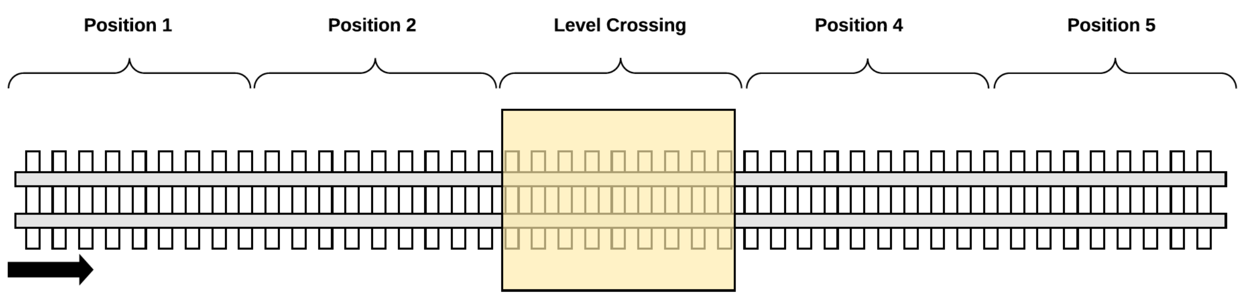


Figure 4.6: *The railway track divided into five parts with the level crossing in the middle.*

In this version, the train is placed outside the crossing system and works as an interacting system, contrary to version 1, where it was placed inside. The train's initial position is in *Position 1* and follows the direction of the arrow through the level crossing until it reaches *Position 5*. The position of the train determines how the crossing system reacts, and updates the states of each component throughout the scenario. The states of the components are determined by ports with a symbolic base type, and the values included in the domains can be seen in figure 4.7. For example, the domain for the light signals have three values. *NONE*, *AMBER* and *RED*.

```
domain LightSignals {NONE, AMBER, RED} end
domain AudibleWarner {NONE, SOUND} end
domain RightBarriers {UP, DOWN} end
domain LeftBarriers {UP, DOWN} end
domain ObstacleDetector {OFF, DETECTED, CLEAR} end
domain RailwaySignals {STOP, GO} end
```

Figure 4.7: *The domains for each component in version 2.*

The system architecture for the components of the crossing system in version 2 (See Figure 4.8) is equal to the one in version 1. However, instead of being sub-blocks of the crossing system, they are now ports of the crossing system block. The barriers have also been divided into right hand barriers and left hand barriers. The crossing system is now positioned at the correct place, which is at the level crossing.

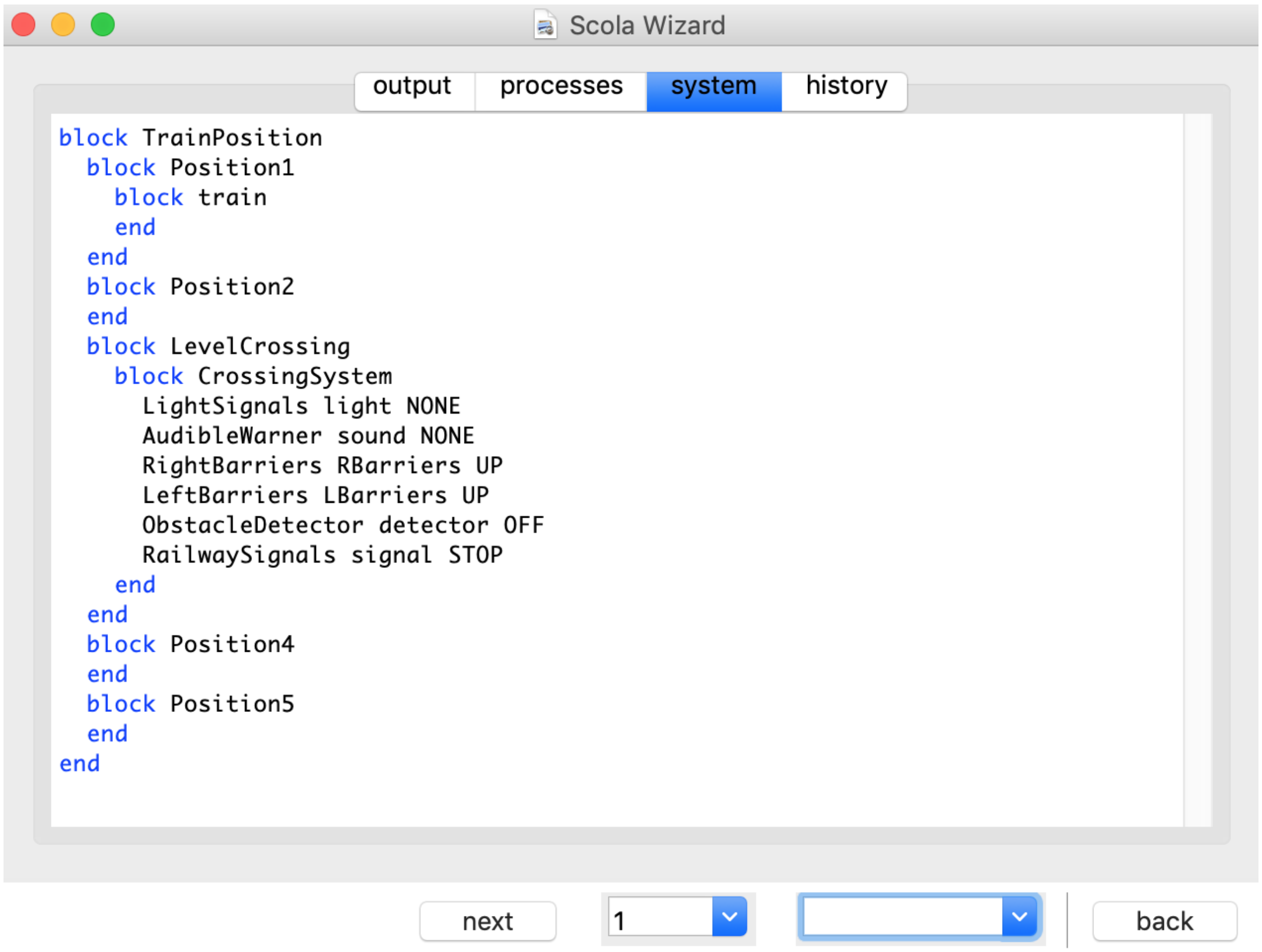


Figure 4.8: *The system architecture in version 2*

In this version it is more interesting to see the evolution of the system architecture as the train moves through the positions. Figure 4.9 shows that the train has moved from *Position 1* to *Position 2* and the values of the light signals and the audible warner have been updated.

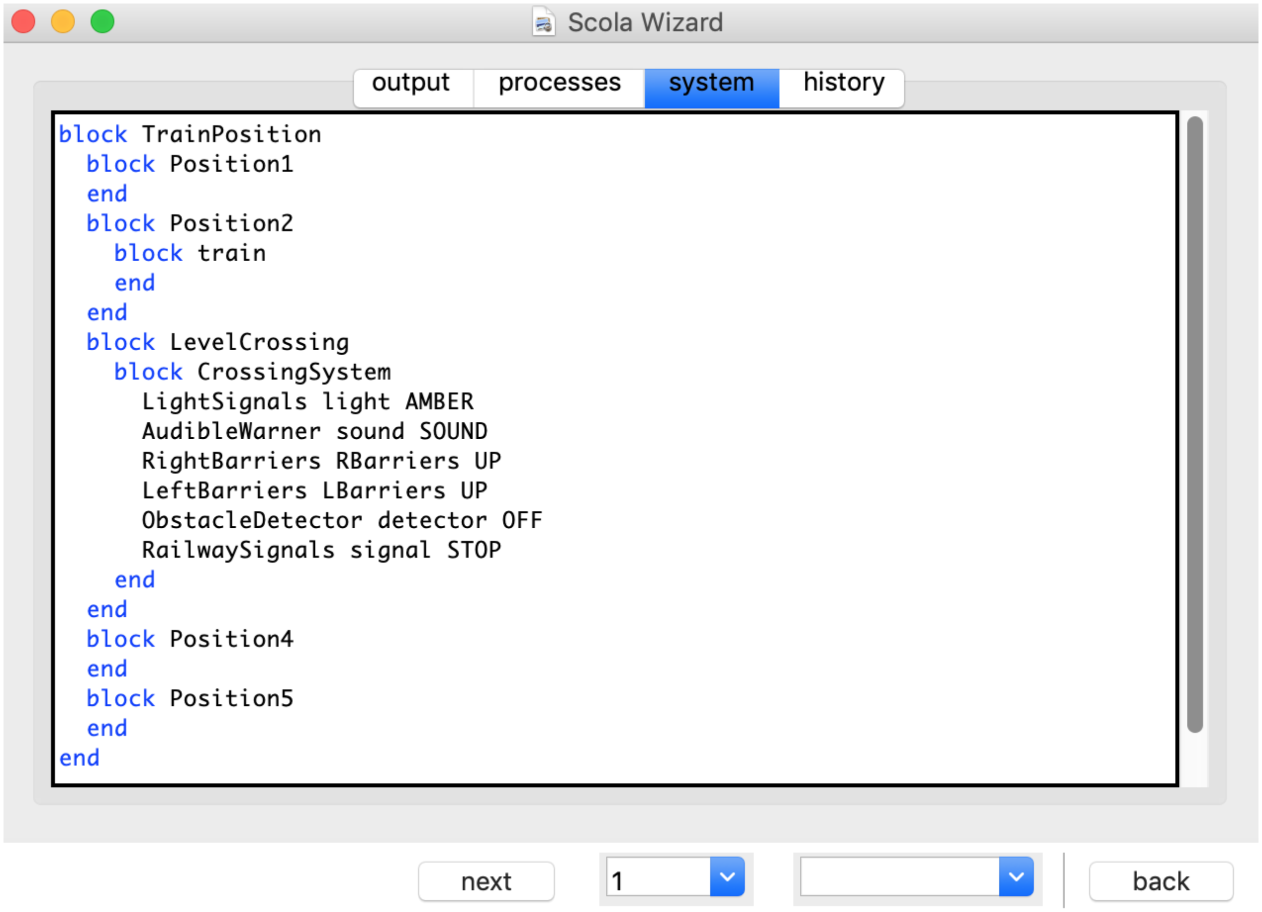


Figure 4.9: *The train at Position 2.*

In figure 4.10, the process have been moved further, which have resulted in the train being located inside the level crossing. Figure 4.11 shows the terminal state, and the train being located at *Position 5*. The ports of the level crossing have now been changed back to their initial value.

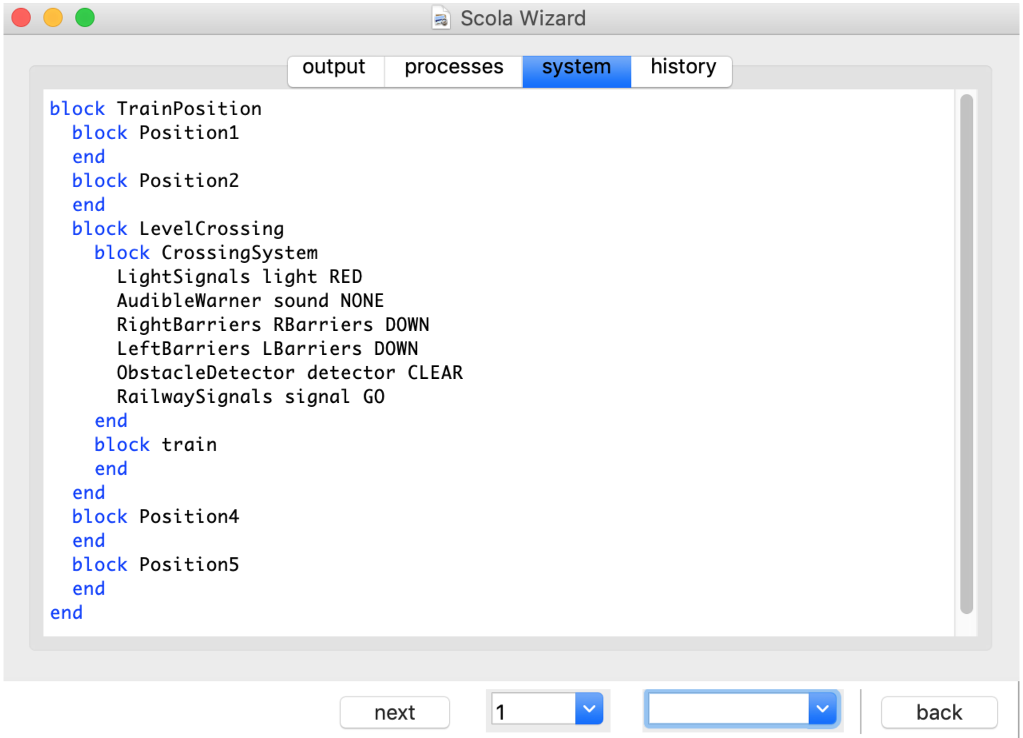


Figure 4.10: *The train placed inside the level crossing.*

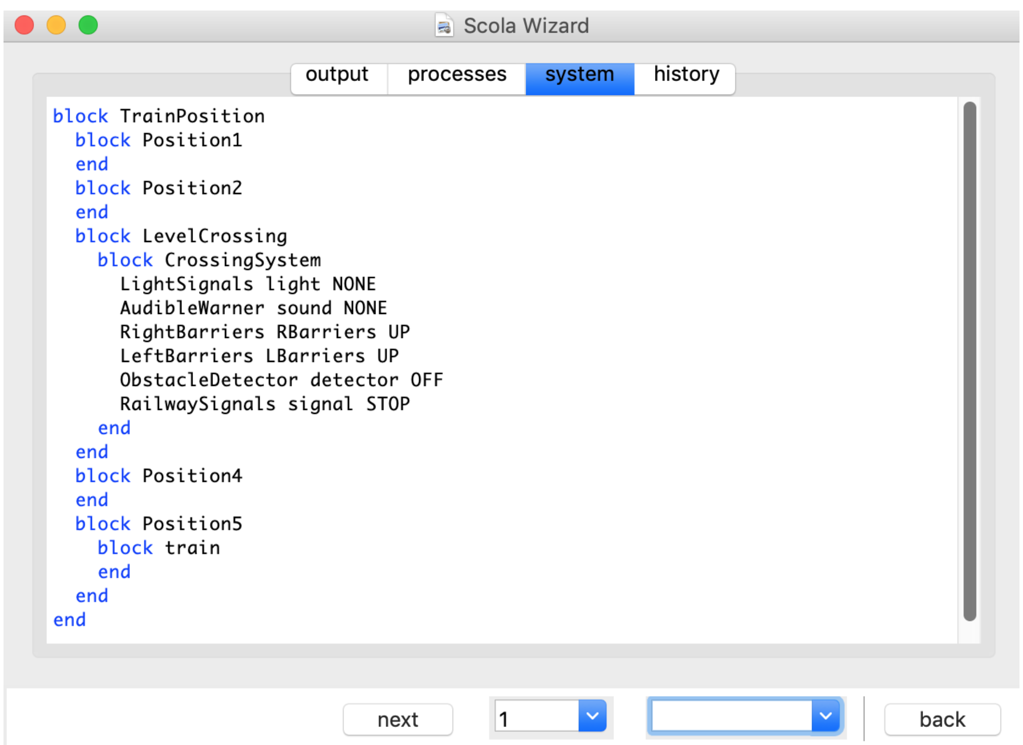


Figure 4.11: *The train at position 5.*

4.3 Version 3

The third version includes assertions and several gateways. It also includes a power supply, and the possibility of adding and removing (create and destroy) trains in the model. Figure 4.12 shows the initial system architecture. The system has three sub-blocks that describes the power supply, the crossing system, and the train position, respectively. The power supply and the crossing system includes their own sub-blocks with assertions. It is the assertions that makes it possible to describe the connections between the components.

The power supply exists of two sub-systems and has a standby redundancy. The emergency power is the standby element, and will only be activated if the main power fails. If the emergency power also fails, the crossing system will have no power source. Every component in the crossing system will then enter a failed state.

The crossing system looks similar to the crossing system in version 2 (figure 4.8). However, now the crossing system block consists of sub-blocks instead of only ports. The sub-blocks have their own ports that are dependent on the power.

The train position block is still divided into five parts, but includes the possibility of adding trains. There are no trains included at the initial position of the process.

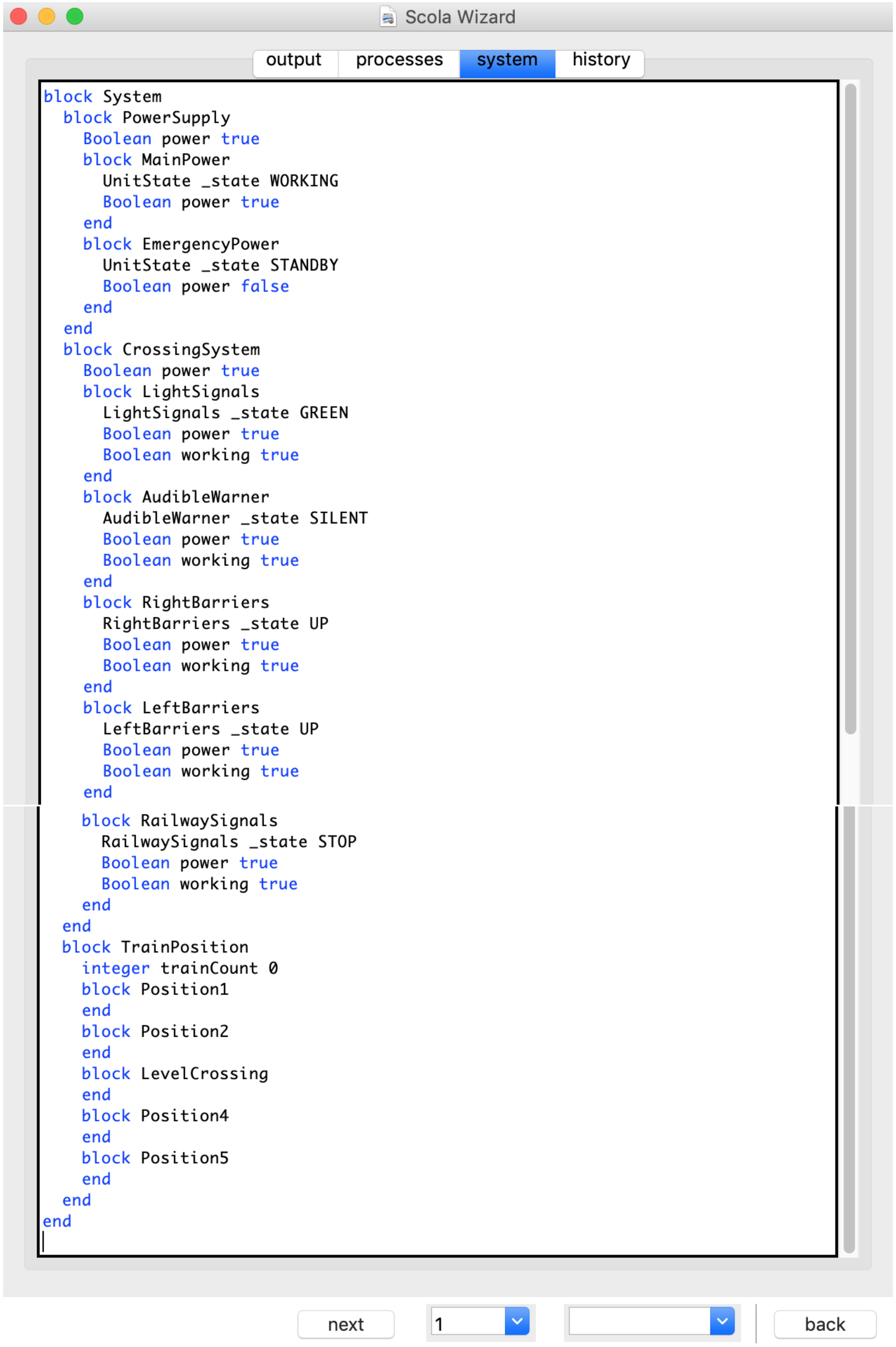


Figure 4.12: The system architecture in version 3.

In this version it is possible to define the states of the crossing system before they are introduced. Firstly, the power supply is defined. If the main power is chosen as working, the process continues to defining the crossing system states. If the main power is chosen as failed, the state of the emergency power has to be chosen. The same choice applies here. If the emergency power is chosen as working, the process continues to defining the crossing system states. If the emergency power also is chosen as failed, all the components in the crossing system fails, since they are dependent of power.

When the power supply state has been defined, the process is deactivated and six new processes are created. One for each component of the crossing system. Here, the states of the components are chosen. Some of the components are dependent of each other for safety reasons. For example, if the right barriers are in a failed state, the left barriers also enters a failed state. If a car is able to enter the level crossing, there should not be a barrier that disables the car from leaving the crossing area. A few other components also have dependencies, and are described in a design structure matrix (Table 4.1).

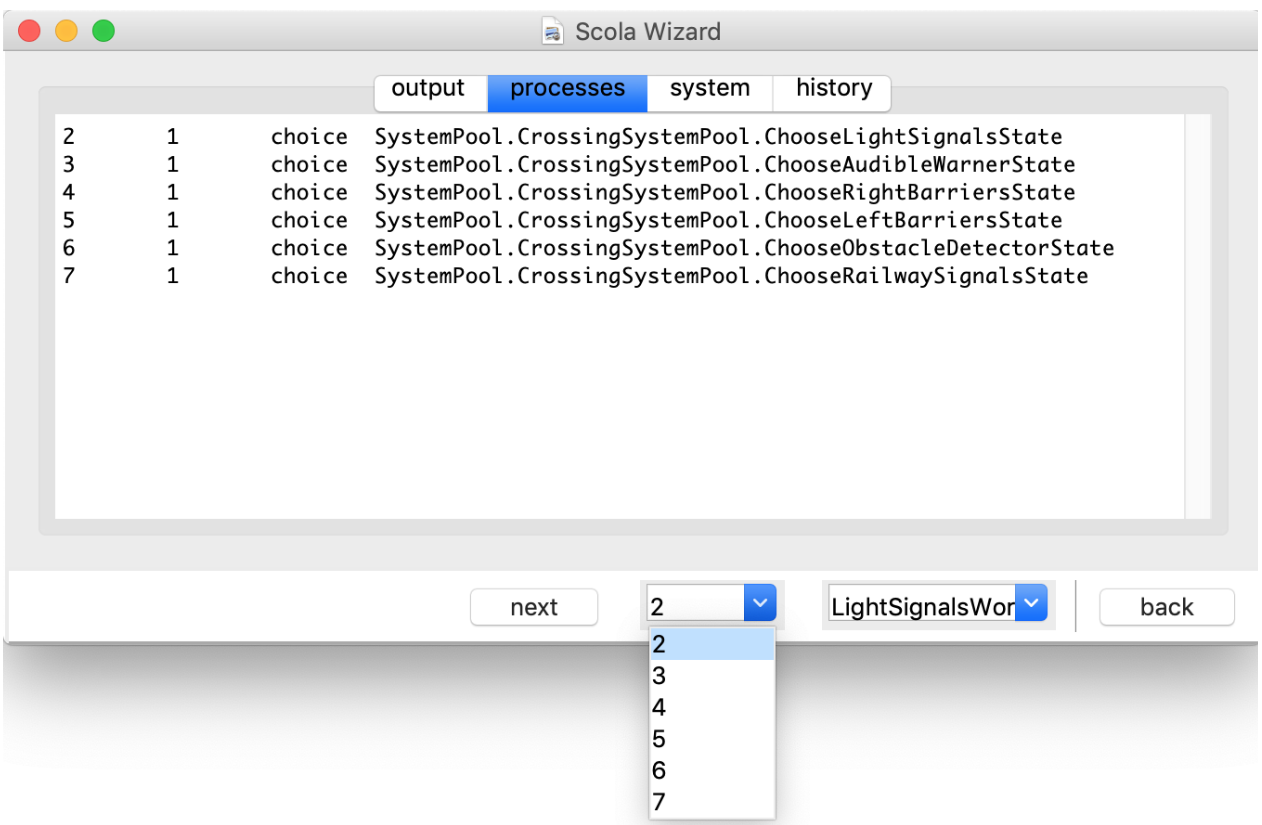


Figure 4.13: *The split-gateway deactivates the parent process (but stores it), and activates six new processes. One for each component of the crossing system. The list of numbers allows the user of the software to change between the processes. After the states of every component has been defined, the children processes are deactivated, and the parent process is reactivated.*

DSM	6	1	2	3	4	5	6
Light Signals	1		x	x	x	x	x
Audible Warning	2						
Right Barriers	3				x		
Left Barriers	4						
Obstacle Detector	5				x		
Railway Signals	6						

Table 4.1: Design structure matrix for the crossing system.

Table 4.1 shows the dependencies between the components in the crossing system. Especially one component; the light signals, have a lot of dependencies. If the light signals fail, all the other components enters a failed state for safety reasons. It is seen as safer to have no crossing system, than to have a crossing system without light signals. For example, a barrier should not start to descend when a driver is about to enter level crossing.

Even though the crossing system has been modelled to satisfactory accuracy, some assumptions have been made:

- There exists a sign next to the railway signals, that tells the train driver to drive a certain speed if the railway signals are in a failed state. If the speed of the train is reduced due to a system/component failure, the possibility of a collision is decreased. This also allows trains to pass even if there is a failure. There also exists a sign next to the light signals that tells the driver to give way for the trains.
- If there exist more than one train in the model at the same time, the following train(s) are not allowed to enter *Position 2* before the foremost train has passed *Position 4*.
- After the state of the system has been defined, no failure can happen.

Figure 4.14 shows the state of the system in the middle of a scenario. The system is defined as powered by the emergency power, and that the obstacle detector and the left hand side barriers are in a failed state. Two trains are positioned in the model, at *Position 2* and *Position 5*. From the integer number, it is possible to see that it is train number 6 and 7 that has passed the defined railway distance under study.

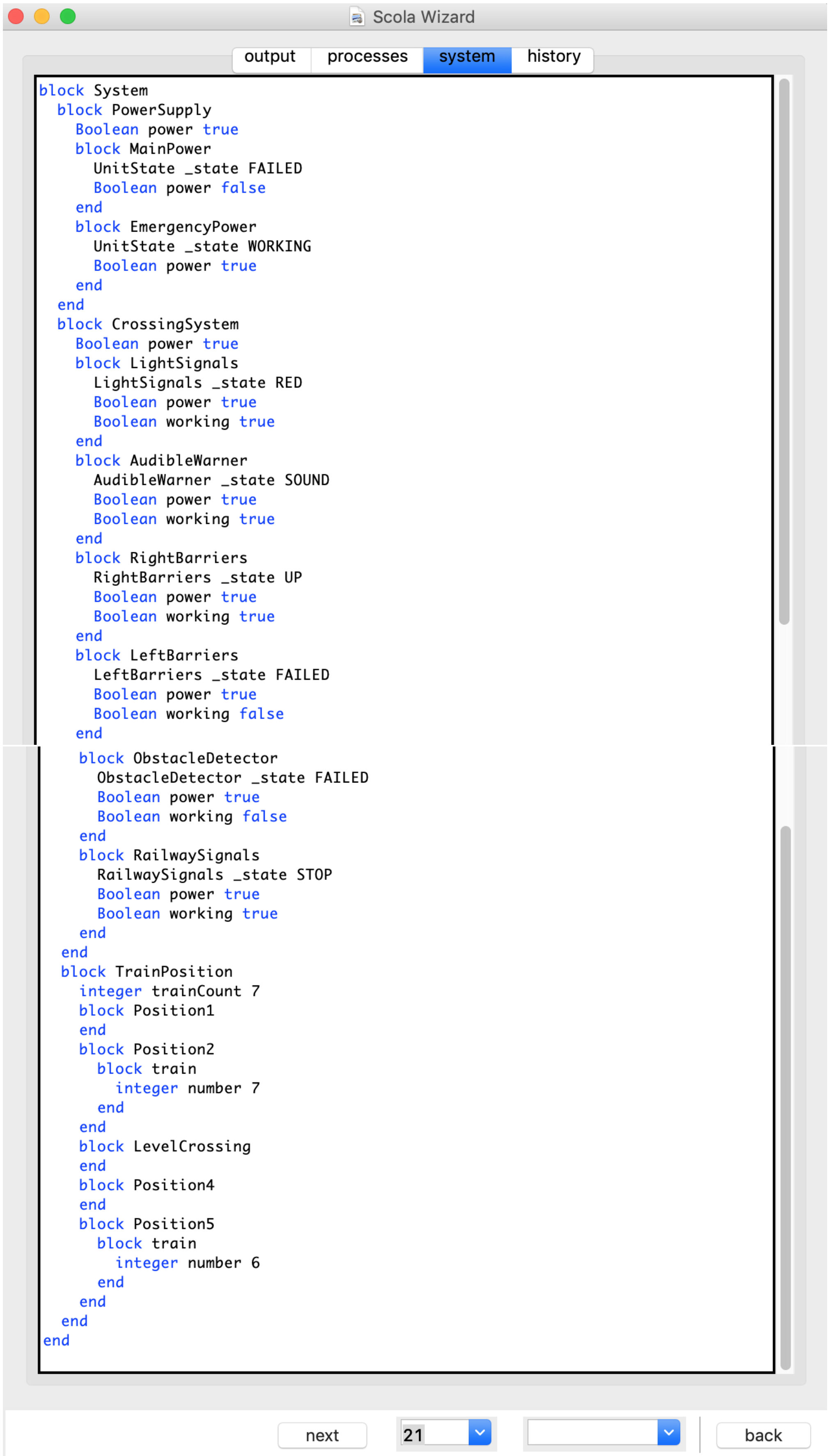


Figure 4.14: The updated system architecture for version 3 in the middle of a scenario.

Chapter 5

Summary and Recommendations for Further Work

This chapter concludes the thesis, and proposes some recommendations for future work.

5.1 Summary

In this thesis, a scenario based approach to modelling in systems engineering was conducted to form an impression of the benefits and usefulness of this type of modelling. Firstly, in chapter 2, theoretical background about systems engineering and model based systems engineering was described. This was done to gain knowledge about the systems engineering field and why there is a need for it. Another reason for its included purpose, was to acquire knowledge about the different existing types of models that could be compared to ScOLa. Chapter 3 introduced ScOLa by explaining the different concepts of ScOLa, and how the modelling language is built in order to make it possible to play scenarios and change the systems architecture by the execution of processes. The chapter also included a small guide of how to understand and operate in the layout of the software, ScOLa Wizard. The following chapter, (chapter 4), introduced a level crossing system that was modelled with ScOLa in three different versions. The reasons for dividing it into three versions was the intention of introducing the various possibilities of ScOLa in parts. Each version of the model was reviewed based on the chosen scenario and the included function(s). The findings and the acquired knowledge from the experimental study laid the foundation for the discussion in chapter 5.2.

5.2 Discussion

Many models used in MBSE (mainly OMG SySML™) are pragmatic models. Their purpose is to facilitate communication, keep a lot of information and take a broad outlook on the system under study. These kind of models use graphical notation and works excellent for their purpose, but lack information when it comes to focusing on details of the system components and the behaviour of the system. The formal (or semantic) model, ScOLa, provides the possibility to model the system architecture and its response to different scenarios, and also to see the behaviour of its connected components.

However, ScOLa should be complemented by another graphical model that also describes the system to fully understand the system under study. It is complicated to interpret a system based on only the textual information ScOLa provides. In some cases, a graphical model might also be needed in the making of a model in ScOLa.

Another point important to mention, is the modelling language itself. For a person with limited knowledge about programming, there might be some issues. The modelling is relatively easy to understand when it comes to making a system with few sub-blocks, a gateway and a couple of transitions between the states and tasks. The problem arises when trying to include several sub-blocks of sub-blocks and the assertions between these. The assertions might also be dependent on different operators that is difficult to decipher without any programming background. ScOLa Wizard gives an indication of which line the error is in, but even then, it can be difficult to detect it. The author of this thesis has only one point of view on this matter and cannot therefore conclude with the opinion of others.

The objective for this thesis was to "Evaluate ScOLa for its ability to support system architecture studies." (chapter 1.3) To answer this question, it is necessary divide it into two perspectives. Does ScOLa contribute to a better understanding of the system architecture? Yes, it clearly does, since it offers something differently compared to the popular models used in MBSE. However, in a competitive market, where the focus is on delivering solutions at the lowest possible cost, and at the shortest possible time, there is a question about the gain vs. the work load. Creating a model in ScOLa for simple systems might be easy, but it provides information that can be extracted from other models. It is in complex systems that ScOLa offers the most, since it is hard to understand the system architecture and its behaviour. The development of models by the use of ScOLa are difficult when they become complex, and the time spent here might not be cost efficient.

5.3 Recommendations for Further Work

The recommendations for further work have been divided into two groups; recommendations for further work and recommendations for inclusions in ScOLa.

Recommendations for further work

- It is possible to make the system even more complex, at a much more detailed component level. The scalability of ScOLa should therefore be tested to check its ability to comprehend additional work load.
- It can be interesting to make a test project where engineers working with system design tests ScOLa in a work context. Engineers that have worked with designing of systems have a completely different view of how things work, since they have practical knowledge. Also, the test subjects should be divided into two groups; one group where all the subjects have background related to programming, and one group where the subjects have none. This makes it possible to state whether or not the language is too complicated.

Recommendations for ScOLa

- One thing encountered while working with ScOLa, was the difficulties with the controlling of assertions. For example, the assertions between the components in version 3 (chapter 4.3). Here, the desirable transition when one of the components failed, was not for the other components to enter a failed state. Instead, they should be turned off. When working with a port with a symbolic base type with >2 values, the assertions were hard to control, since different scenarios should make different transitions. An assertion that makes a transition based on what the previous state was, is desirable. (See next paragraph).

In retrospective, the desired transition might have been possible if a third port was added and were dependent on both the port with the symbolic base type and the port with the Boolean base type.

Bibliography

Dekker, S. (2011). *Drift into Failure, From Hunting Broken Components to Understanding Complex Systems*. Ashgate Publishing Limited.

Freng, C. E. and Freng, P. D. (2007). RAE Report Web. (293074).

Friedenthal, S., Moore, A., and Steiner, R. (2012). *A Practical Guide to SysML The Systems Modeling Language*. Morgan Kaufmann, 225 Wyman Street, Waltham, MA 02451, USA, 2nd edition.

Holt, J., Perry, S., Payne, R., Bryans, J., Hallerstede, S., and Hansen, F. O. (2015). A model-based approach for requirements engineering for systems of systems. *IEEE Systems Journal*, 9(1):252–262.

IHI (2018). IHI's technologies adopted for cold, snowy regions. *Intelligent Information Management Headquarters of IHI Corporation.*

INCOSE (2007). SYSTEMS ENGINEERING HANDBOOK A GUIDE FOR SYSTEM LIFE CYCLE PROCESSES AND ACTIVITIES. *INCOSE SYSTEMS ENGINEERING HANDBOOK, version 3.1.*

INCOSE (2015). About systems engineering.

ISO 15288:2015 (2015). Systems and software engineering - System life cycle processes. Standard, International Organization for Standardization, Geneva, CH.

Kossiakoff, A., N. Sweet, W., J. Seymour, S., and Biemer, M. (2008). *Systems engineering principles and practice*, volume 20. Wiley, 2nd edition.

ORR (2011). Level crossings : A guide for managers , designers and operators Railway Safety Publication 7. *Regulation*, (December).

Ramos, A. L., Ferreira, J. V., and Barceló, J. (2012). Model-based systems engineering: An emerging approach for modern systems. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, 42(1):101–111.

Rauzy, A. (2018). Scola: a scenario-oriented language. *Norges teknisk-naturvitenskapelige universitet.*

Rauzy, A. B. and Haskins, C. (2018). Foundations for model-based systems engineering and model-based safety assessment. *Department of Mechanical and Industrial Engineering, Norwegian University of Science and Technology (NTNU).*

Appendix A

Acronyms

BPMN Business Process Model and Notation

DSM Design Structure Matrix

MBSE Model-Based Systems Engineering

SE Systems engineering

RAMS Reliability, Availability, Maintainability, and Safety

ScOLa Scenario-Oriented Language

Appendix B

BPMN

A BPMN of the scenario described in chapter 4.1 is shown on the following page.

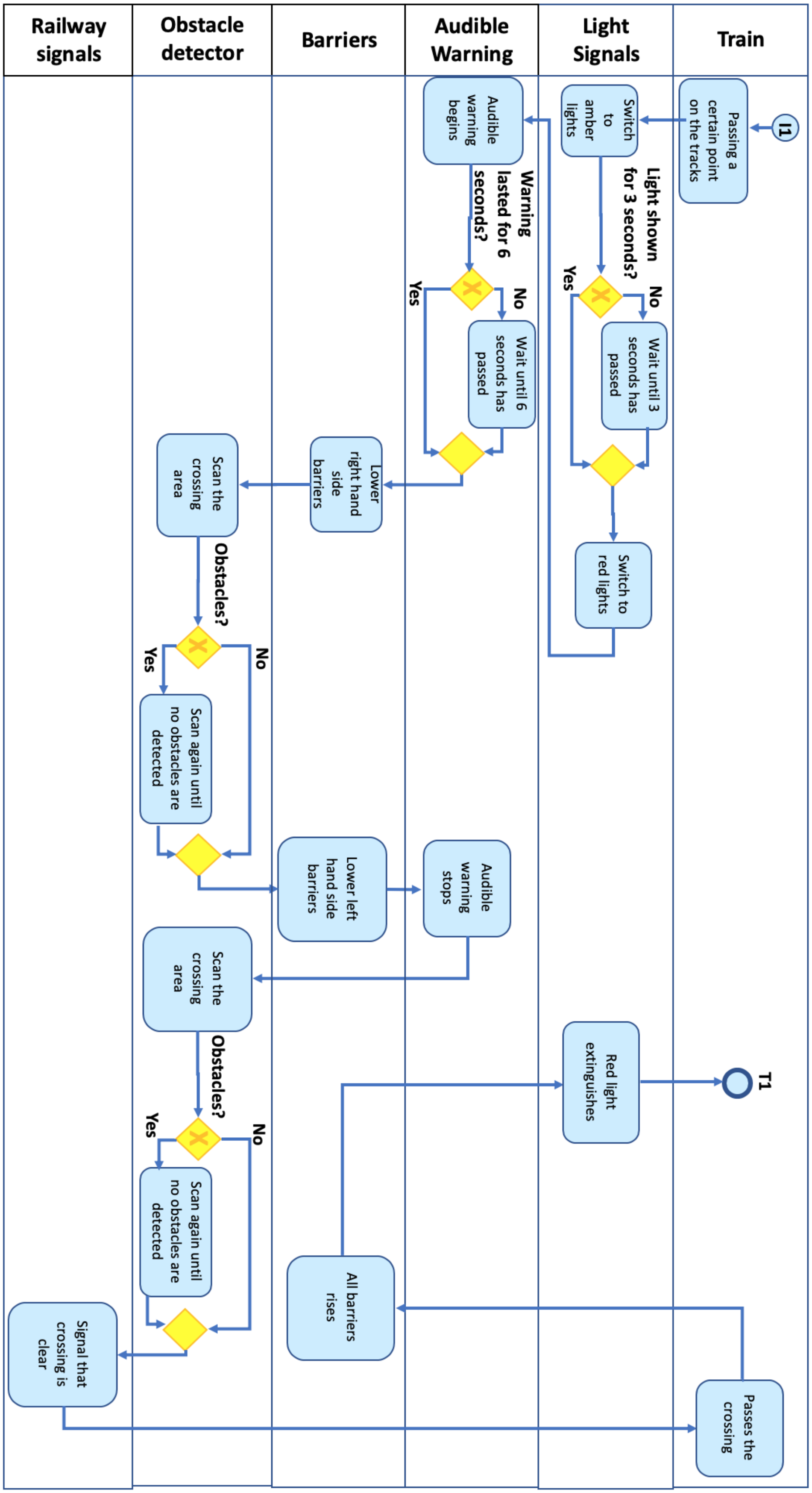


Figure B.1: *BPMN*

Appendix C

Scola Codes

This appendix shows every Scola code that has been discussed through this thesis, and are listed in the same order as presented in the thesis.

C.1 Water Faucet Model

```
domain Faucet {OPEN, CLOSED} end

block System
  block WaterSupply
    Boolean outFlow true
  end
  block WaterFaucet
    Faucet _state CLOSED
    Boolean inFlow false
    Boolean outFlow false
    assertion Transfer
      set outFlow (if (eq _state OPEN) inFlow false)
    end
  end
  assertion Transfer
    set WaterFaucet.inFlow WaterSupply.outFlow
  end
end

scenario SystemPool as System
  scenario BathroomFaucet as WaterFaucet
    state Initial
    task CloseHandle
      set _state CLOSED
    end
    task OpenHandle
      set _state OPEN
    end
    next Initial OpenHandle
    next CloseHandle OpenHandle
    next OpenHandle CloseHandle
  end
end
```

Figure C.1: *Scola Code - Water Faucet*

C.2 Version 1

```
block CrossingSystem
  block Train
  end
  block LightSignals
  end
  block AudibleWarning
  end
  block Barriers
  end
  block ObstacleDetector
  end
  block RailwaySignals
  end
end

scenario  TrainPassing
  scenario TrainLane as  CrossingSystem.Train
    state Initial
    task  TrainPassing end
    task  PassTheCrossing end
    state Terminal
    next Initial  TrainPassing
  end
  scenario LightSignalsLane as CrossingSystem.LightSignals
    task  SwitchToAmberLights end
    task  WaitThreeSeconds  end
    task  SwitchToRedLights end
    task  RedLightExtinguishes end
    choice  LightShownForThreeSeconds
      branch  yes
      branch  no
    end
```

```
    next SwitchToAmberLights  LightShownForThreeSeconds
    next LightShownForThreeSeconds.no WaitThreeSeconds
    next LightShownForThreeSeconds.yes SwitchToRedLights
    next WaitThreeSeconds SwitchToRedLights
  end
  scenario AudibleWarningLane as CrossingSystem.AudibleWarning
    task AudibleWarningBegins end
    task WaitSixSeconds end
    task AudibleWarningStops end
    choice WarningLastedForSixSeconds
      branch  yes
      branch  no
    end
    next AudibleWarningBegins WarningLastedForSixSeconds
    next WarningLastedForSixSeconds.no WaitSixSeconds
  end
  scenario BarriersLane as CrossingSystem.Barriers
      task  LowerRightHandSideBarriers end
      task  LowerLeftHandSideBarriers end
      task  RiseAllBarriers end
  end
  scenario ObstacleDetectorLane as CrossingSystem.ObstacleDetector
    task ScanTheCrossingArea end
    task ScanTheArea end
    task ScanUntilNoObstaclesAreDetected end
    task ScanAgainUntilNoObstaclesAreDetected end
    choice Obstacles
      branch yes
      branch no
    end
```

```
    choice ObstaclesInTheArea
      branch yes
      branch no
    end
    next ScanTheCrossingArea Obstacles
    next Obstacles.yes ScanUntilNoObstaclesAreDetected
    next ScanTheArea ObstaclesInTheArea
    next ObstaclesInTheArea.yes ScanAgainUntilNoObstaclesAreDetected
  end
  scenario RailwaySignalsLane as CrossingSystem.RailwaySignals
    task SignalThatTheCrossingIsClear end
  end
  next TrainLane.TrainPassing LightSignalsLane.SwitchToAmberLights
  next LightSignalsLane.SwitchToRedLights AudibleWarningLane.AudibleWarningBegins
  next AudibleWarningLane.WaitSixSeconds  BarriersLane.LowerRightHandSideBarriers
  next AudibleWarningLane.WarningLastedForSixSeconds.yes  BarriersLane.LowerRightHandSideBarriers
  next BarriersLane.LowerRightHandSideBarriers ObstacleDetectorLane.ScanTheCrossingArea
  next ObstacleDetectorLane.Obstacles.no BarriersLane.LowerLeftHandSideBarriers
  next ObstacleDetectorLane.ScanUntilNoObstaclesAreDetected BarriersLane.LowerLeftHandSideBarriers
  next BarriersLane.LowerLeftHandSideBarriers AudibleWarningLane.AudibleWarningStops
  next AudibleWarningLane.AudibleWarningStops ObstacleDetectorLane.ScanTheArea
  next ObstacleDetectorLane.ObstaclesInTheArea.no RailwaySignalsLane.SignalThatTheCrossingIsClear
  next ObstacleDetectorLane.ScanAgainUntilNoObstaclesAreDetected RailwaySignalsLane.SignalThatTheCrossingIsClear
  next RailwaySignalsLane.SignalThatTheCrossingIsClear TrainLane.PassTheCrossing
  next TrainLane.PassTheCrossing BarriersLane.RiseAllBarriers
  next BarriersLane.RiseAllBarriers LightSignalsLane.RedLightExtinguishes
  next LightSignalsLane.RedLightExtinguishes TrainLane.Terminal
end
```

Figure C.2: *Scola Code - Version 1 (divided into three figures)*

C.3 Version 2

```
domain LightSignals {NONE, AMBER, RED} end
domain AudibleWarner {NONE, SOUND} end
domain RightBarriers {UP, DOWN} end
domain LeftBarriers {UP, DOWN} end
domain ObstacleDetector {OFF, DETECTED, CLEAR} end
domain RailwaySignals {STOP, GO} end

block TrainPosition
  block Position1
    block train end
  end
  block Position2 end
  block LevelCrossing
    block CrossingSystem
      LightSignals light NONE
      AudibleWarner sound NONE
      RightBarriers RBarriers UP
      LeftBarriers LBarriers UP
      ObstacleDetector detector OFF
      RailwaySignals signal STOP
    end
  end
  block Position4 end
  block Position5 end
end

```

```
scenario PositionOfTrain as TrainPosition
    state Initial
    task MoveTrainToPosition2
      move Position1.train Position2.train
    end
    task MoveTrainToLevelCrossing
      move Position2.train LevelCrossing.train
    end
    task MoveTrainToPosition4
      move LevelCrossing.train Position4.train
    end
    task MoveTrainToPosition5
      move Position4.train Position5.train
    end
  scenario Position2Lane as Position2
  end
  scenario LevelCrossingLane as LevelCrossing
    state StartSequenceToCloseRoadTraffic
    task Warning
      set CrossingSystem.light AMBER
      set CrossingSystem.sound SOUND
    end
    task ChangeToRedLight
      set CrossingSystem.light RED
    end
```

```
    task LowerRightHandSideBarriers
      set CrossingSystem.RBarriers DOWN
    end
    choice ObstaclesInTheArea
      branch yes
      branch no
    end
    task DetectedObstacles
      set CrossingSystem.detector DETECTED
    end
    task NoDetectedObstacles
      set CrossingSystem.detector CLEAR
      set CrossingSystem.LBarriers DOWN
    end
    task StopAudibleWarning
      set CrossingSystem.sound NONE
    end
    choice DetectedObstaclesInClosedArea
      branch yes
      branch no
    end
    task ObjectsInClosedArea
      set CrossingSystem.detector DETECTED
    end
```

```
    task NoObjectsInClosedArea
      set CrossingSystem.signal GO
      set CrossingSystem.detector CLEAR
    end
    task TrainPassedLevelCrossing
      set CrossingSystem.signal STOP
    end
    task StartSequenceToOpenRoadTraffic
      set CrossingSystem.detector OFF
      set CrossingSystem.RBarriers UP
      set CrossingSystem.LBarriers UP
    end
    task ChangeToGreenLight
      set CrossingSystem.light NONE
    end
    state SafeForTrainToPassLevelCrossing
    next StartSequenceToCloseRoadTraffic Warning
    next Warning ChangeToRedLight
    next ChangeToRedLight LowerRightHandSideBarriers
    next LowerRightHandSideBarriers ObstaclesInTheArea
    next ObstaclesInTheArea.yes DetectedObstacles
    next DetectedObstacles ObstaclesInTheArea
    next ObstaclesInTheArea.no NoDetectedObstacles
    next NoDetectedObstacles StopAudibleWarning
    next StopAudibleWarning DetectedObstaclesInClosedArea
    next DetectedObstaclesInClosedArea.yes ObjectsInClosedArea
    next ObjectsInClosedArea DetectedObstaclesInClosedArea
```

```
    next DetectedObstaclesInClosedArea.no NoObjectsInClosedArea
    next NoObjectsInClosedArea SafeForTrainToPassLevelCrossing
  end
  state Terminal
  next Initial MoveTrainToPosition2
  next MoveTrainToPosition2 LevelCrossingLane.StartSequenceToCloseRoadTraffic
  next LevelCrossingLane.SafeForTrainToPassLevelCrossing MoveTrainToLevelCrossing
  next MoveTrainToLevelCrossing MoveTrainToPosition4
  next MoveTrainToPosition4 LevelCrossingLane.TrainPassedLevelCrossing
  next LevelCrossingLane.TrainPassedLevelCrossing LevelCrossingLane.StartSequenceToOpenRoadTraffic
  next LevelCrossingLane.StartSequenceToOpenRoadTraffic LevelCrossingLane.ChangeToGreenLight
  next LevelCrossingLane.ChangeToGreenLight MoveTrainToPosition5
  next MoveTrainToPosition5 Terminal
end
```

Figure C.3: *Scola Code - Version 2 (divided into five figures)*

C.4 Version 3

```
domain UnitState {STANDBY, WORKING, FAILED} end
domain LightSignals {GREEN, AMBER, RED, FAILED} end
domain AudibleWarner {SILENT, SOUND, FAILED} end
domain RightBarriers {UP, DOWN, FAILED} end
domain LeftBarriers {UP, DOWN, FAILED} end
domain ObstacleDetector {STANDBY, DETECTED, CLEAR, FAILED} end
domain RailwaySignals {STOP, GO, FAILED} end

block System
  block PowerSupply
    Boolean power true
    assertion Transfer
      set power (or MainPower.power EmergencyPower.power)
    end
    block MainPower
      UnitState _state WORKING
      Boolean power true
      assertion Transfer
        set power (eq _state WORKING)
      end
    end
    block EmergencyPower
      UnitState _state STANDBY
      Boolean power false
      assertion Transfer
        set power (eq _state WORKING)
      end
    end
  end
```

```
  block CrossingSystem
    Boolean power true
    block LightSignals
      LightSignals _state GREEN
      Boolean power true
      Boolean working true
      assertion Transfer
        set working (df _state FAILED)
      end
    end
    block AudibleWarner
      AudibleWarner _state SILENT
      Boolean power true
      Boolean working true
      assertion Transfer
        set working (df _state FAILED)
      end
    end
    block RightBarriers
      RightBarriers _state UP
      Boolean power true
      Boolean working true
      assertion Transfer
        set working (df _state FAILED)
      end
    end
```

```
    block LeftBarriers
      LeftBarriers _state UP
      Boolean power true
      Boolean working true
      assertion Transfer
        set working (df _state FAILED)
      end
    end
    block ObstacleDetector
      ObstacleDetector _state STANDBY
      Boolean power true
      Boolean working true
      assertion Transfer
        set working (df _state FAILED)
      end
    end
    block RailwaySignals
      RailwaySignals _state STOP
      Boolean power true
      Boolean working true
      assertion Transfer
        set working (df _state FAILED)
      end
    end
    assertion Powering
      set LightSignals.power power
      set AudibleWarner.power power
      set RightBarriers.power power
      set LeftBarriers.power power
      set ObstacleDetector.power power
      set RailwaySignals.power power
    end
  end
```

```
  block TrainPosition
    integer trainCount 0
    block Position1 end
    block Position2 end
    block LevelCrossing end
    block Position4 end
    block Position5 end
  end
  assertion Powering
    set CrossingSystem.power PowerSupply.power
  end
end

scenario SystemPool as System
  scenario PowerSupplyPool as PowerSupply
    choice ChoosePower
      branch MainPowerWorking
      branch MainPowerFailed
    end
    task MainPowerWorking
      set MainPower._state (if (eq MainPower._state FAILED) WORKING FAILED)
    end
    task MainPowerFailed
      set MainPower._state (if (eq MainPower._state WORKING) FAILED WORKING)
    end
    choice ChooseEmergencyPower
      branch EmergencyPowerWorking
      branch EmergencyPowerFailed
    end
    task EmergencyPowerWorking
      set EmergencyPower._state (if (eq EmergencyPower._state STANDBY) WORKING STANDBY)
    end
```

```
    task EmergencyPowerFailed
      set EmergencyPower._state (if (eq EmergencyPower._state STANDBY) FAILED STANDBY)
    end
    state Power
    next ChoosePower.MainPowerFailed MainPowerFailed
    next ChoosePower.MainPowerWorking Power
    next MainPowerFailed ChooseEmergencyPower
    next ChooseEmergencyPower.EmergencyPowerWorking EmergencyPowerWorking
    next EmergencyPowerWorking Power
    next ChooseEmergencyPower.EmergencyPowerFailed EmergencyPowerFailed
  end
  scenario CrossingSystemPool as CrossingSystem
    split DefineCrossingSystem
      branch LightSignals
      branch AudibleWarner
      branch RightBarriers
      branch LeftBarriers
      branch ObstacleDetector
      branch RailwaySignals
    end
```

```
    merge DefinedCrossingSystem
      branch LightSignalsWorking
      branch LightSignalsFailed
      branch AudibleWarnerWorking
      branch AudibleWarnerFailed
      branch RightBarriersWorking
      branch RightBarriersFailed
      branch LeftBarriersWorking
      branch LeftBarriersFailed
      branch ObstacleDetectorWorking
      branch ObstacleDetectorFailed
      branch RailwaySignalsWorking
      branch RailwaySignalsFailed
    end
    choice ChooseLightSignalsState
      branch LightSignalsWorking
      branch LightSignalsFailed
    end
    task LightSignalsWorking
      set LightSignals._state GREEN
    end
    task LightSignalsFailed
      set LightSignals._state FAILED
      set AudibleWarner._state FAILED
      set RightBarriers._state FAILED
      set LeftBarriers._state FAILED
      set ObstacleDetector._state FAILED
      set RailwaySignals._state FAILED
    end
    choice ChooseAudibleWarnerState
      branch AudibleWarnerWorking
      branch AudibleWarnerFailed
    end
```

```
    task AudibleWarnerWorking
      set AudibleWarner._state (if (df AudibleWarner._state FAILED) SILENT FAILED)
    end
    task AudibleWarnerFailed
      set AudibleWarner._state FAILED
    end
    choice ChooseRightBarriersState
      branch RightBarriersWorking
      branch RightBarriersFailed
    end
    task RightBarriersWorking
      set RightBarriers._state (if (df RightBarriers._state FAILED) UP FAILED)
    end
    task RightBarriersFailed
      set RightBarriers._state FAILED
      set LeftBarriers._state FAILED
    end
    choice ChooseLeftBarriersState
      branch LeftBarriersWorking
      branch LeftBarriersFailed
    end
    task LeftBarriersWorking
      set LeftBarriers._state (if (df LeftBarriers._state FAILED) UP FAILED)
    end
    task LeftBarriersFailed
      set LeftBarriers._state FAILED
    end
    choice ChooseObstacleDetectorState
      branch ObstacleDetectorWorking
      branch ObstacleDetectorFailed
    end
```

```
    task ObstacleDetectorWorking
      set ObstacleDetector._state (if (df ObstacleDetector._state FAILED) STANDBY FAILED)
    end
    task ObstacleDetectorFailed
      set ObstacleDetector._state FAILED
      set LeftBarriers._state FAILED
    end
    choice ChooseRailwaySignalsState
      branch RailwaySignalsWorking
      branch RailwaySignalsFailed
    end
    task RailwaySignalsWorking
      set RailwaySignals._state (if (df RailwaySignals._state FAILED) STOP FAILED)
    end
    task RailwaySignalsFailed
      set RailwaySignals._state FAILED
    end
    task Warning
      set LightSignals._state (if (df LightSignals._state FAILED) AMBER FAILED)
      set AudibleWarner._state (if (df AudibleWarner._state FAILED) SOUND FAILED)
    end
    task RedLight
      set LightSignals._state (if (df LightSignals._state FAILED) RED FAILED)
    end
    task LowerRightHandSideBarriers
      set RightBarriers._state (if (df RightBarriers._state FAILED) DOWN FAILED)
    end
    choice ScanTheCrossingArea
      branch DetectedObstacles
      branch NoDetectedObstacles
    end
```

```
    task DetectedObstaclesInArea
      set ObstacleDetector._state (if (df ObstacleDetector._state FAILED) DETECTED FAILED)
    end
    task NoDetectedObstaclesInArea
      set ObstacleDetector._state (if (df ObstacleDetector._state FAILED) CLEAR FAILED)
    end
    task LowerLeftHandSideBarriers
      set LeftBarriers._state (if (df LeftBarriers._state FAILED) DOWN FAILED)
    end
    task StopAudibleWarning
      set AudibleWarner._state (if (df AudibleWarner._state FAILED) SILENT FAILED)
    end
    choice ScanTheClosedArea
      branch DetectedObstacles
      branch NoDetectedObstacles
    end
    task DetectedObstaclesInClosedArea
      set ObstacleDetector._state (if (df ObstacleDetector._state FAILED) DETECTED FAILED)
    end
    task NoDetectedObstaclesInClosedArea
      set ObstacleDetector._state (if (df ObstacleDetector._state FAILED) CLEAR FAILED)
    end
    task SetRailWaySignalsGo
      set RailwaySignals._state (if (df RailwaySignals._state FAILED) GO FAILED)
    end
    task SetRailwaySignalsStop
      set RailwaySignals._state (if (df RailwaySignals._state FAILED) STOP FAILED)
    end
    task TrainPassedLevelCrossing
      set LightSignals._state (if (df LightSignals._state FAILED) GREEN FAILED)
      set LeftBarriers._state (if (df LeftBarriers._state FAILED) UP FAILED)
      set RightBarriers._state (if (df RightBarriers._state FAILED) UP FAILED)
    end
```

```
    task NoPower
      set LightSignals._state FAILED
      set AudibleWarner._state FAILED
      set RightBarriers._state FAILED
      set LeftBarriers._state FAILED
      set ObstacleDetector._state FAILED
      set RailwaySignals._state FAILED
    end
    state SystemDefined
    next Warning RedLight
    next RedLight LowerRightHandSideBarriers
    next LowerRightHandSideBarriers ScanTheCrossingArea
    next ScanTheCrossingArea.DetectedObstacles DetectedObstaclesInArea
    next DetectedObstaclesInArea ScanTheCrossingArea
    next ScanTheCrossingArea.NoDetectedObstacles NoDetectedObstaclesInArea
    next NoDetectedObstaclesInArea LowerLeftHandSideBarriers
    next LowerLeftHandSideBarriers StopAudibleWarning
    next StopAudibleWarning ScanTheClosedArea
    next ScanTheClosedArea.DetectedObstacles DetectedObstaclesInClosedArea
    next DetectedObstaclesInClosedArea ScanTheClosedArea
    next ScanTheClosedArea.NoDetectedObstacles NoDetectedObstaclesInClosedArea
    next NoDetectedObstaclesInClosedArea SetRailWaySignalsGo
    next DefineCrossingSystem.LightSignals ChooseLightSignalsState
    next ChooseLightSignalsState.LightSignalsWorking LightSignalsWorking
    next LightSignalsWorking DefinedCrossingSystem.LightSignalsWorking
    next ChooseLightSignalsState.LightSignalsFailed LightSignalsFailed
    next LightSignalsFailed DefinedCrossingSystem.LightSignalsFailed
    next DefineCrossingSystem.AudibleWarner ChooseAudibleWarnerState
    next ChooseAudibleWarnerState.AudibleWarnerWorking AudibleWarnerWorking
    next AudibleWarnerWorking DefinedCrossingSystem.AudibleWarnerWorking
    next ChooseAudibleWarnerState.AudibleWarnerFailed AudibleWarnerFailed
    next AudibleWarnerFailed DefinedCrossingSystem.AudibleWarnerFailed
    next DefineCrossingSystem.RightBarriers ChooseRightBarriersState
```

```
    next ChooseRightBarriersState.RightBarriersWorking RightBarriersWorking
    next RightBarriersWorking DefinedCrossingSystem.RightBarriersWorking
    next ChooseRightBarriersState.RightBarriersFailed RightBarriersFailed
    next RightBarriersFailed DefinedCrossingSystem.RightBarriersFailed
    next DefineCrossingSystem.LeftBarriers ChooseLeftBarriersState
    next ChooseLeftBarriersState.LeftBarriersWorking LeftBarriersWorking
    next LeftBarriersWorking DefinedCrossingSystem.LeftBarriersWorking
    next ChooseLeftBarriersState.LeftBarriersFailed LeftBarriersFailed
    next LeftBarriersFailed DefinedCrossingSystem.LeftBarriersFailed
    next DefineCrossingSystem.ObstacleDetector ChooseObstacleDetectorState
    next ChooseObstacleDetectorState.ObstacleDetectorWorking ObstacleDetectorWorking
    next ObstacleDetectorWorking DefinedCrossingSystem.ObstacleDetectorWorking
    next ChooseObstacleDetectorState.ObstacleDetectorFailed ObstacleDetectorFailed
    next ObstacleDetectorFailed DefinedCrossingSystem.ObstacleDetectorFailed
    next DefineCrossingSystem.RailwaySignals ChooseRailwaySignalsState
    next ChooseRailwaySignalsState.RailwaySignalsWorking RailwaySignalsWorking
    next RailwaySignalsWorking DefinedCrossingSystem.RailwaySignalsWorking
    next ChooseRailwaySignalsState.RailwaySignalsFailed RailwaySignalsFailed
    next RailwaySignalsFailed DefinedCrossingSystem.RailwaySignalsFailed
    next DefinedCrossingSystem SystemDefined
    next NoPower SystemDefined
  end
  scenario PositionOfTrain as TrainPosition
    test IncomingTrain
      case yes (not (is_block train))
    end
    task NewTrain
      set trainCount (add trainCount 1)
      new block train
      new integer train.number trainCount
    end
```

```
    fork PlaceTrainPosition1
      branch IncomingTrain
      branch place
    end
    test IsPosition1Free
      case yes (not (is_block Position1.train))
    end
    task MoveTrainToPosition1
      move train Position1.train
    end

    test IsPosition2Free
      case yes (not (is_block Position2.train))
    end
    task MoveTrainToPosition2
      move Position1.train Position2.train
    end
    test IsLevelCrossingFree
      case yes (not (is_block LevelCrossing.train))
    end
    task MoveTrainToLevelCrossing
      move Position2.train LevelCrossing.train
    end
    test IsPosition4Free
      case yes (not (is_block Position4.train))
    end
    task MoveTrainToPosition4
      move LevelCrossing.train Position4.train
    end
    test IsPosition5Free
      case yes (not (is_block Position5.train))
    end
```

```
    task MoveTrainToPosition5
      move Position4.train Position5.train
    end
    task TrainPassed
      delete Position5.train
    end
    state Initial
    next Initial IncomingTrain
    next IncomingTrain.yes NewTrain
    next NewTrain PlaceTrainPosition1
    next PlaceTrainPosition1.IncomingTrain IncomingTrain
    next PlaceTrainPosition1.place IsPosition1Free
    next IsPosition1Free.yes MoveTrainToPosition1
    next MoveTrainToPosition1 IsPosition2Free
    next IsPosition2Free.yes MoveTrainToPosition2
    next MoveTrainToPosition2 IsLevelCrossingFree
    next IsLevelCrossingFree.yes MoveTrainToLevelCrossing
    next MoveTrainToLevelCrossing IsPosition4Free
    next IsPosition4Free.yes MoveTrainToPosition4
    next MoveTrainToPosition4 IsPosition5Free
    next IsPosition5Free.yes MoveTrainToPosition5
    next TrainPassed Terminal
    state Terminal
  end
```

```
  state Initial
  next Initial PowerSupplyPool.ChoosePower
  next PowerSupplyPool.Power CrossingSystemPool.DefineCrossingSystem
  next PowerSupplyPool.EmergencyPowerFailed CrossingSystemPool.NoPower
  next CrossingSystemPool.SystemDefined PositionOfTrain.Initial
  next PositionOfTrain.MoveTrainToPosition2 CrossingSystemPool.Warning
  next CrossingSystemPool.SetRailWaySignalsGo PositionOfTrain.IsLevelCrossingFree
  next PositionOfTrain.MoveTrainToLevelCrossing CrossingSystemPool.SetRailwaySignalsStop
  next CrossingSystemPool.SetRailwaySignalsStop PositionOfTrain.IsPosition4Free
  next PositionOfTrain.MoveTrainToPosition5 CrossingSystemPool.TrainPassedLevelCrossing
  next CrossingSystemPool.TrainPassedLevelCrossing PositionOfTrain.TrainPassed
end
```

Figure C.4: *Scola Code - Version 3 (divided into 14 figures)*