Model Based Testing of a PLC Based Interlocking System

Master's Thesis
University of Twente

Thijs ten Hoeve
November 2, 2012

Committee:
Prof. Dr. J.C. van de Pol (*University of Twente*)
Dr. M.I.A. Stoelinga (*University of Twente*)
W.M.T. Brandt-Mennen MSc MBA (*ProRail*)

Abstract

Interlocking systems control all wayside elements in a railway yard. These systems are responsible for safe train operations and must prevent collisions and derailments from happening. It is absolutely essential that interlocking systems operate flawlessly, and as a result a lot of effort is put in their verification.

ProRail (the Dutch railway infrastructure manager) is developing a new type of interlocking system based on off-the-shelf PLC hardware, the *PLC-Interlocking*.

This report introduces a conformance testing methodology for PLC-Interlocking systems that is based around the JTorX test tool. The test method has been applied to the first instance of a PLC-Interlocking system, which is installed at the Santpoort Noord station.

To get an SUT that can be tested, the interlocking logic is integrated in a program that adds interfacing code and is then recompiled for a regular PC system. As a result, testing does not require access to the PLC hardware.

A number of test purpose models have been created to direct test case generation. These test purpose models were implemented in Java, using a custom framework that creates an LTS representation of inputs and outputs and handles communication with JTorX

A partial specification model has been implemented in mCRL2. Because of the complexity of the requirements, and the limited available time, it was not possible to create a complete model.

The created test setup allows automated testing of the interlocking logic of PLC-Interlocking systems. The entire test setup can be run on a regular PC (if it is equipped with sufficient memory) and does not require PLC hardware. Of 73 automated test runs that were conducted, 36 ended with the observation of a failure. Four of these failures must be further investigated, but it seems likely that all failures were caused by faults in the model.

Creating a complete and correct model is the biggest obstacle to using this testing methodology more in the future.

Acknowledgements

A lot of people have helped me in one way or another during the period in which I have written this thesis, and during my study in general; and I owe many thanks to all of them. I would like to use this page as an opportunity to name and thank some of the people that have helped me.

Firstly, I would like to thank the members of my graduation committee, Jaco van de Pol, Mariëlle Stoelinga and Wendi Mennen, as well as my daily supervisor Jaco Schoonen, for all the time and effort that they have spent on helping me conduct my research and writing my thesis.

I am grateful that ProRail allowed me to do an internship at the *Treinbeveiligings* department.

For most of my time at ProRail I shared a room with Wendi, Jaco S. and André Broersen. I am very grateful to them for introducing me to the fascinating world of railway signalling. They have spent countless hours explaining railway technology to me, and have included me in interesting discussions, meetings and *borrels*.

I would also like to thank everyone else at ProRail for being nice roommates, interesting colleagues, and very helpful to me in general. Likewise, I am very thankful to all Movares colleagues that have been very welcoming, helpful, good discussion partners, and a valuable source of information.

Furthermore, I would like to thank Axel Belinfante who always provided quick answers to any question I had regarding JTorX. I am thankful to everyone who has ever proofread any of my many drafts, this includes all members of my committee, but also: my father, my girlfriend and Simon Beijer.

My girlfriend has always offered loving and encouraging words, even when at some points I did not seem to progress in my studies. Thanks Johanna.

I would like to conclude this page by thanking my parents. From as early as I can remember they have always encouraged me to learn a lot, and do well in school. My mom has always taken an active interest in my education, and when I was young helped me with my homework and volunteered at my schools. My father first introduced me to programming which sparked my interest for computers and eventually led to me starting my computer science study. I cannot thank both of them enough.

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xii

Listings xiii

1 Introduction 1
- 1.1 Motivation 2
- 1.2 Summary of Research and Results 3
  - 1.2.1 Model Based Testing 3
  - 1.2.2 The Created Test Setup 3
  - 1.2.3 Results and Conclusions 5
- 1.3 Related Work 5
  - 1.3.1 Verification of Interlocking Systems 5
  - 1.3.2 Verification of PLC Systems 6
  - 1.3.3 Model Based Testing 7
  - 1.3.4 Testing Interlocking Systems 7
- 1.4 Report Structure 8
- 1.5 Sources 9

I Background 11

2 Train Safety on the Dutch Railways 13
- 2.1 Train Control Architecture 13
  - 2.1.1 Safety Responsibility 14
- 2.2 Devices in the Physical Layer 15
  - 2.2.1 Signals 15
  - 2.2.2 Points 16
  - 2.2.3 Train Detection 16
- 2.3 Interlocking Systems 17
  - 2.3.1 Logic 17
  - 2.3.2 Interlocking System Types 18
- 2.4 Railway Design and Verification Process 19

3 PLC-Interlocking Systems 21

3.1 Programmable Logic Controllers . . . 21
3.2 PLC-Interlocking Architecture . . . 22
3.3 Interlocking Logic . . . 24
3.3.1 Function Block Diagram Programming . . . 24
3.3.2 Used Functions and Types . . . 25
3.3.3 Interfaces . . . 26
3.3.4 Programming Specific Instances . . . 28
3.4 First Instance: Santpoort Noord . . . 29

II Testing a PLC-Interlocking 31

4 Model Based Testing 33
4.1 Input-Output Conformance Testing . . . 33
4.2 Test Setup . . . 35
4.2.1 SUT and Models . . . 35
4.2.2 Test Run Execution and Configuration . . . 36
4.2.3 On the Fly Model Exploration . . . 36
4.3 The mCRL2 Modelling Language . . . 38
4.3.1 Processes . . . 38
4.3.2 Data . . . 39
4.3.3 Mappings . . . 40
4.3.4 Constructed Data Types . . . 42

5 Behaviour of the PLC-Interlocking Model 45
5.1 Specifications . . . 45
5.1.1 Generic Sources . . . 46
5.1.2 Location Specific Sources . . . 46
5.1.3 Imperfect Specifications . . . 47
5.2 Scope of the Model . . . 48
5.3 Input and Output Transitions . . . 49
5.3.1 Variables . . . 49
5.3.2 Transitions . . . 50
5.3.3 Timer Transitions . . . 51
5.4 Implemented Functionality . . . 53
5.4.1 Signals: Yellow or Better . . . 53
5.4.2 Signals: Determining Aspects . . . 55

6 Implementation of the PLC-Interlocking Model 57
6.1 Model Execution Flow . . . 57
6.1.1 Input . . . 58
6.1.2 Computation: Yellow or Better . . . 58
6.1.3 Output: EBP Output Yellow or Better . . . 60
6.1.4 Computation: Signal Aspects . . . 61
6.1.5 Ouput: aspects . . . 62
6.2 The State Data Structure . . . 62
6.2.1 ActiveRoute . . . 63
6.2.2 Signal . . . 63
6.2.3 physicalSections and symbolicSections . . . 64
6.2.4 Timer . . . 64

6.3 Data Mappings . . . 65
6.3.1 Static Mappings . . . 65
6.3.2 Convenience Mappings . . . 67
6.3.3 Computation Mappings . . . 68
6.4 Pre-Processor Usage and File Structure . . . 71
6.4.1 Pre-Processor . . . 71
6.4.2 File Structure . . . 72
6.5 Adaptability . . . 72

7 The System Under Test **75**
7.1 Compilation Process . . . 75
7.2 Program Structure . . . 77
7.2.1 C++ Code Structure . . . 77
7.2.2 Code Structure for the SUT Executable . . . 79
7.3 Reimplementing the IEC Standard Library . . . 80
7.4 Interfaces . . . 81
7.4.1 Interface to PLC Code . . . 81
7.4.2 External Interfaces . . . 82
7.5 Problems and Alternatives . . . 83

8 Test Case Generation **85**
8.1 Random Test Generation Problems . . . 85
8.1.1 Generating Useful Scenarios . . . 85
8.1.2 Generating Consistent Scenarios . . . 87
8.1.3 Eliminating Redundant Transitions . . . 87
8.2 Test Purpose Models . . . 88
8.2.1 Java as Modelling Language . . . 88
8.2.2 Framework for Java Test Purpose Models . . . 90
8.3 Test Scenarios . . . 91
8.3.1 Constrained Random Scenario Generator . . . 91
8.3.2 Fixed Scenarios . . . 92
8.3.3 Simulated Train Movements . . . 94

9 Java Framework for Test Purpose Models **99**
9.1 The Torx-Explorer Protocol . . . 99
9.2 Framework: Bridge between JTorX and Test Purpose Models . . . 100
9.2.1 Example Scenario . . . 101
9.3 Model Structure . . . 102
9.4 Main Loop . . . 103
9.5 Determining Successor States . . . 105
9.5.1 Fields of the State Class . . . 105
9.5.2 Santpoort Noord Specific Subclass . . . 106
9.5.3 The determineTransitions Method . . . 107
9.6 Adaptability . . . 108

10 Testing and Results **111**
10.1 Executed Test Runs . . . 111
10.1.1 Test Setup Details . . . 112
10.2 Performance Measurements . . . 113
10.2.1 Measurements . . . 113

10.3 Performance: Execution Time 114
10.3.1 Complete Test Setup 114
10.3.2 Individual Components 115
10.4 Performance: Memory Usage 118
10.4.1 Inefficiencies in the Test Setup 119
10.5 Ability to Find Faults 120
10.5.1 Types of Errors Found 121
10.5.2 Constrained Random Scenario Generator 121
10.5.3 Simulated Train Movements 122
10.6 Finding Faults: Evaluation 122

11 Conclusions and Future Work **125**
11.1 Conclusions 125
11.1.1 The SUT 125
11.1.2 Model 126
11.1.3 Test Purpose Model 126
11.1.4 The Complete Test Setup 126
11.1.5 Main Conclusions 128
11.2 Future Work 128

Glossary **131**

Bibliography **134**

List of Figures

1.1 JTorX test setup . . . 3

2.1 Train control architecture . . . 14
2.2 Point positions . . . 16
2.3 Line block system on a single track . . . 18

3.1 The components of a PLC-Interlocking system . . . 23
3.2 A snippet of FBD program code . . . 25
3.3 Interfaces of the interlocking logic . . . 27
3.4 Schematic view of the Santpoort Noord station area . . . 29

4.1 JTorX test setup . . . 34
4.2 JTorX and the SUT in the test setup . . . 35
4.3 Compilation and execution of models . . . 37

5.1 The modelled part of the Santpoort Noord railway yard . . . 48
5.2 Inputs and outputs of the model . . . 50
5.3 Simplified view of the model's input and output transitions . . . 52
5.4 Timing interaction between model and SUT adapter . . . 53

7.1 Compilation chain for the SUT . . . 76
7.2 Graph showing the behaviour of a TOF function block . . . 81
7.3 Runtime structure and internal interfaces of the SUT executable 82

10.1 Execution time of the complete test setup . . . 114
10.2 Execution speed of the complete test setup . . . 115
10.3 Execution times of individual components (CRSG test purpose) . 116
10.4 Execution times of individual components (STM test purpose) . 116
10.5 Memory usage of individual components (CRSG test purpose) . . 118
10.6 Memory usage of individual components (STM test purpose) . . 118

List of Tables

2.1 Signalling aspects 15

3.1 Used standard function blocks 26

5.1 Input variables of the model 49
5.2 Output variables of the model 50

7.1 Main header files 78
7.2 Conversion of special characters in the generated code 78
7.3 Main items in the SUT's source tree 79

9.1 Overview of classes in the Java test purpose model 102

10.1 Observed failures with the CRSG test purpose 122
10.2 Observed failures with the STM test purpose 123
10.3 Observed failure types per test purpose model 124

Listings

4.1 Simple example model 38
4.2 Example model using action composition and recursion 39
4.3 Example model with data parameters 39
4.4 Example model with the sum keyword 39
4.5 Example model with conditional behaviour 40
4.6 Example model with a simple mapping 40
4.7 Example model with a recursive mapping 41
4.8 Example model with a list 42
4.9 Example model with a custom sort struct 43

6.1 Handling input transitions 58
6.2 Processes handling timer transitions 59
6.3 Outputting the yellow or better EBP outputs 60
6.4 Determining the signal aspects 61
6.5 Outputting the signal aspects 62
6.6 The isVirtualSignal mapping 65
6.7 InputId mappings 66
6.8 The stateSetSignals mapping 67
6.9 The determineAspects mapping 69

8.1 The update function of the CRSG test purpose model 92
8.2 The update function of the first fixed scenario test purpose model 93
8.3 The update function of the STM test purpose model 95
8.4 Simulation of train movements by the STM'S Train class 96

9.1 Example communication between JTorX and a torx-explorer . . 100
9.2 Example communication with a test purpose model 101
9.3 Main loop of the test purpose model framework 103
9.4 Definition of transition labels per transition stage in SptnState . 106
9.5 The determineTransitions method 107

Chapter 1

Introduction

Prorail is the Dutch railway infrastructure manager. As such, Prorail is tasked with maintaining, improving and expanding the railway network and transfer capacity (railway stations). The Dutch railway network consists of 7000 km of railway tracks on which trains travel almost 150 million kilometres annually. ProRail does not operate any train services on the network; it controls the railway traffic and distributes the available railway and transfer capacity among train operating companies.

An important part of ProRail's responsibilities is guaranteeing safe train movements. The *Treinbeveiliging* department (English: Signalling department) is specially tasked with maintaining and improving the safety of train movements. This department maintains knowledge about train safety related systems, creates and maintains specifications for these safety systems and monitors trends among deployed safety systems. Furthermore this department is involved in developing new safety systems.

I carried out the research described in this report while working as an intern at ProRail's Treinbeveiliging department.

An important link in the railway infrastructure with regard to safety is formed by interlocking systems. Interlocking systems control all wayside elements and are responsible for disallowing all train movements that might lead to collisions or derailments.

There are a number of different interlocking types in use. Because of economical and strategic reasons, ProRail was not satisfied with the previously available alternatives. Therefore ProRail commissioned the development of a new type of interlocking system based on commercial off-the-shelf PLC hardware. This *PLC-Interlocking* system is being developed by a project team consisting of people from Movares (an external engineering firm) and people from ProRail. The project has not yet been completed entirely, but the first PLC-Interlocking instance is already in use. It is installed at the Santpoort Noord railway station and has been in operation since June 2012.

The newly developed PLC-Interlocking (and specifically the installation at Santpoort Noord) was the target of the research that is described in this report.

The main research question, as formulated beforehand:

> *'How can the safety of the PLC-Interlocking logic best be tested using JTorX based testing methods.'*

1.1 Motivation

Interlocking systems have an important safety responsibility in the railway infrastructure. Therefore it is absolutely essential that these systems will not fail in an unsafe manner.

As a result, a lot of effort is put in the verification and validation of interlocking systems. An important part of the overall verification effort is the verification of the interlocking logic with regard to safety requirements. Most of the interlocking logic in the PLC-Interlocking is derived from legacy systems that have been verified in the past. However, some new functionality of the interlocking logic has never been implemented before. This prompted the research described in this report; ProRail is interested in new methods to verify the correctness of (new) interlocking logic, specifically for the PLC-Interlocking with regard to safety.

In the past there has been a considerable amount of research aimed at verifying interlocking systems and general PLC systems. Preliminary literature research [1] has made it clear that, with current technology, it is not possible to prove the correctness of PLC-Interlocking systems using exhaustive verification techniques such as model checking. The internal state space of PLC-Interlockings is simply too big. Another problem is the complexity of the safety requirements. It is not clear how a set of safety requirements that is provable complete can be defined for a PLC-interlocking system.

Since using exhaustive techniques was not possible, the research has focused on using a model based testing technique. Testing is not affected by the problems with state space explosions that make exhaustive techniques impossible to use. On the other hand, testing is inherently incomplete and cannot give a proof of correctness. It can however increase the confidence in the correctness of a system.

The literature does not report on any past attempts to use model based testing in the verification of interlocking systems. As such it is novel research, and an interesting test case to see whether model based testing can be successfully applied to interlocking systems.

The ultimate goal of the research has been to increase confidence in the correctness of the PLC-Interlocking installation at Santpoort Noord and of PLC-Interlocking systems in general through the use of model based testing. However, just finding out whether model based testing can be applied to an interlocking system constitutes an interesting goal on its own.

1.2 Summary of Research and Results

The following sections give a brief introduction to model based testing, followed by an overview of the conducted research, and the main results and conclusions.

1.2.1 Model Based Testing

The method presented in this report tests PLC-Interlocking systems using a model based testing approach. More specifically, the approach is based on the input-output conformance (ioco) theory. Timmer et al. [2] comprehensively surveyed this testing theory and its formal underpinning. The allowed behaviour of the system under test is expressed in a specification model based on a Labelled Transition System (LTS), where the transitions express input and output actions. This model is also used to derive (on the fly) test cases.

The theoretical basis of the ioco theory is formed by the ioco conformance relation, which defines under what conditions an implementation conforms to a specification. An implementation ioco-conforms to a specification, if at all times it can handle at least all inputs from the specification, and produces at most all outputs from the specification. With the exception that the implementation is not allowed not to provide any output when the specification requires one.

Practical application of this theory requires tooling. JTorX [3] is a test tool that can test whether an implementation ioco-conforms to a given specification model. Given a specification model and an SUT, JTorX can execute test runs fully automated. If JTorX observes a failure during a test run it will report a *fail* verdict; if no failures are observed a *pass* verdict will be given after a test run has finished. JTorX allows the use of a test-purpose model, which is put in parallel with the specification model, to steer test case generation.

Figure 1.1 gives a schematic view of a typical JTorX test setup.

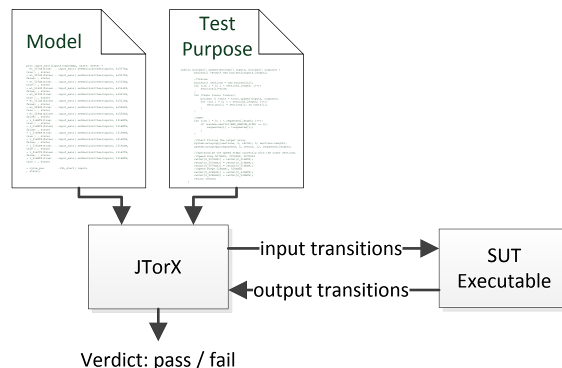


Figure 1.1: JTorX test setup

1.2.2 The Created Test Setup

The conducted research has focused on the interlocking logic of the PLC-Interlocking installation that is installed at the Santpoort Noord railway station. Besides JTorX which was a given, the setup consists of three main parts that have been

developed as part of the conducted research: the SUT, a specification model and a test purpose model.

Using a PLC-Interlocking as the SUT is problematic because such a system is very expensive and hard to interface with. Therefore an executable has been created that can be run on a regular PC, which embeds the interlocking logic of a PLC-Interlocking, and adds an interface for JTorX to interact with this logic. The interlocking logic is written in the non-parsable graphical Function Block Diagram (FBD) format, but a C++ representation of the logic is generated during the regular compile process for PLC executables. This C++ representation of the interlocking logic forms the heart of the created executable SUT.

JTorX works with LTS based models, as a result all inputs and outputs are communicated using transitions. A PLC-Interlocking continuously executes an input-computation-output cycle, in which the computation part is taken care of by the interlocking logic. This cycle is simulated in the models by having transitions to set input variables, a transition to indicate the start of the computation phase, and transitions to output computed values. However, the interlocking logic is not aware of these transitions. The executable in which the interlocking logic is embedded has to translate all transitions to function calls that set variables, execute the interlocking logic and output computed values.

A specification model has been created in mCRL2, that (partially) specifies what constitutes correct behaviour for the PLC-Interlocking.

The generic behaviour of this model is based on the generic specifications of the PLC-Interlocking in the Subsystem/System Specification (SSS) [4] and OVS Application Engineering document [5] as well as expert consultations [6, 7]. The Santpoort Noord specific behaviour is based on the OBE-blad [8], OS-blad [9] and DO document [10] for Santpoort Noord.

The developed model is an initial research model that serves as a proof of concept, and does not capture all functionality of a PLC-Interlocking. The behaviour of PLC-Interlocking systems is so complex that it was not possible to create a complete model in the allotted time. In fact, the complexity of the requirements led to faults in the partial model.

Random test case generation leads merely to extremely unlikely and inconsistent test cases, which is not useful for testing. Therefore, test purpose models have been developed that steer test case generation. Four different test purpose models have been implemented: Two create fixed scenarios, one generates test scenarios randomly with some constraints (the *CRSG* test purpose model), and one generates scenarios based on simulated train movements (the *STM* test purpose model).

These models have been implemented in Java, rather than in mCRL2, because Java was deemed to be the superior language for the task. Since Java cannot be translated to an LTS or generate an LTS representation natively (which JTorX requires), a custom Java framework was created, which handles communicating inputs and outputs with JTorX using transitions.

1.2.3 Results and Conclusions

The conducted research has led to a working test setup, that with some effort could be used to test the interlocking logic of any PLC-Interlocking system.

In total 73 test runs have been done with the CRSG and STM test purpose models.

The performance of the test setup depends heavily on the used test purpose model. The execution time for a test run with 100.000 transitions varies from about 3:20 minutes (STM test purpose) to over 21 minutes (CRSG test purpose). The memory consumption of the total test setup can be heavy. In the longest conducted test runs (650.000 transitions), both JTorX and the used test purpose model used well over 20 GiB of memory. The specification model (about 2.5 GiB) and the SUT (only about 1 MiB) contribute relatively little to the overall memory consumption of the test setup.

Of the 73 test runs performed with the CRSG and STM test purpose models, 36 ended with the observation of a failure. Four of the found failures must be further examined, but as it stands now it seems to be that all failures are caused by faults in the model.

The STM test purpose model was more successful in generating test cases that found failures (33 failures in 50 test runs) than the CRSG test purpose model (3 failures in 23 test runs). But it is not possible to draw conclusions as to which test purpose model is most useful for finding faults in the SUT, since the found faults were in the model.

Considering that the model only models a part of the SUT's behaviour, yet generates so many false positives, it must be concluded that the quality of the test setup is not at (or close to) a level where it can be used in the regular validation process of interlocking systems.

1.3 Related Work

In the past, there has been some considerable research into verifying interlocking systems, as well as some limited research into verifying PLC systems in general.

1.3.1 Verification of Interlocking Systems

Most research into verification of interlocking systems has focused on using exhaustive techniques: SAT-solving, symbolic model checking and explicit state model checking. Typically, case studies focus on a specific interlocking installation and try to prove that the interlocking installation will never allow train movements that might lead to collisions or derailments.

An approach used in multiple case studies [11, 12, 13, 14, 15] and benchmarks [16] is to translate the interlocking logic to a model (for model checking) or a Boolean equation (for SAT-solving). Properties are defined as formulae, representing requirements over the inputs and outputs of the system (e.g. If input X is true, then output Y must be false). These properties can mostly be derived

directly from the track layout. Model checkers[17, 18] or SAT-solvers [19] are then used to prove that the desired properties hold for the models or equations.

Researchers were able to prove many properties of interlocking systems using this methodology. However, it is very hard to prove that the used properties are complete (i.e. that they are sufficient to prove that the system will disallow all dangerous train movements). One author also mentioned [13] that some safety requirements could simply not be expressed in the used formalism (Boolean equations in that case).

Other papers [20, 21, 22, 23] describe research in which models were created from control tables. Control tables are stateless abstractions of interlocking systems that define simple rules that an interlocking should implement. These models were coupled with an environmental model with simulated trains and the basic requirements: no collisions and no derailments. These requirements were verified to hold in the models using model checkers.

The used requirements are easier to define than those of the earlier mentioned approach. However verifying an interlocking system using this approach is still far from trivial. The models abstract a lot of real world details away (e.g. they have a fixed number of trains), which makes their behaviour incomplete. Furthermore, the cited works only verify control tables that define high level behaviour, but not actual program code that implements those control tables.

A few research papers [11, 12, 13, 15] (describing the first approach) are of exceptional interest because they report on attempts to verify interlocking systems in the Netherlands during the 1990s. These Vital Processor Interlocking (VPI) systems are used in the same context as PLC-Interlocking systems (namely the Dutch railways), must fulfil the same safety requirements and even have many architectural similarities with PLC-Interlocking systems.

None of the approaches in the literature are likely to be usable for verification of PLC-Interlocking systems.

Even the VPI research is not applicable to PLC-Interlockings, despite the similarities in the systems. A practical reason for this is that the PLC-Interlocking code is in a graphical format that cannot easily be transformed to a model. A more fundamental reason is that a model of a PLC-Interlocking system will be much more complex than those of the systems verified in the literature. The main cause of this is the fact that the PLC-Interlocking is much faster than other interlocking systems, which will result in much more states during certain fixed time intervals that are relevant to the requirements.

1.3.2 Verification of PLC Systems

There is also some limited research on the verification of software for Programmable Logic Controller (PLC) systems. Both Gourcuff et al. [24] and Pavlovic and Ehrich [25] used the NuSMV [17] symbolic model checker to verify PLC applications. Their methods used automated translations of PLC programs (from different source formats) to NuSMV models. The authors report successfully applying their methods in different case studies. Pavlovic and Ehrich even verified a component of an interlocking system in their case study. However,

the systems in the case studies were significantly less complex than a complete PLC-Interlocking system.

1.3.3 Model Based Testing

JTorX is not the only tool in its class. The following lists some other tools that are used for automated model based conformance testing.

Firstly, JTorX is the successor of an earlier tool, TorX [26]. TorX is similar in functionality to JTorX, but implements an older version of the ioco theory, and is harder to deploy according to Belinfante [3].

TGV [27] is a tool that has many similarities with JTorX in the kind of models that it can handle, and how it generates test cases. Contrary to JTorX, TGV can only generate test cases and requires external tools to execute those test cases.

The AGEDIS testing toolset [28] includes tools for automated test case generation and execution. The AGEDIS tools perform conformance testing similarly to what JTorX does, but use custom data formats for the models (whereas JTorX and TGV support any language that can be translated to an LTS).

Spec-Explorer [29] is an integrated tool for automated conformance testing, that has been developed by Microsoft. It can natively test non-distributed .NET programs, without needing any adapter 'glue' code, but can also be used to test other types of components. Specification models can be written in one of two custom languages, one of which is an extension of the common C# language.

JTorX has been used in at least one published case study [30] as well as a recently presented one [31]. The other tools listed above have also all been used in case studies. But there are no known cases in which conformance testing tools have been used to test interlocking systems.

1.3.4 Testing Interlocking Systems

Although there are no known cases of model based conformance testing being applied to interlocking systems, there are some other test approaches that have been published.

Mutlu et al. [32] reported on a method that they developed for testing complete interlocking systems, where they simulate all wayside elements using a PLC. This method can test a complete interlocking system instead of just the software, because the simulation PLC is connected to the interlocking system directly. The simulated inputs are generated by simulated train movements, similar to the control table verification work covered in section 1.3.1. The simulation system does simple consistency checks on the interlocking's outputs, and also verifies the absence of collisions and derailments in the simulated scenarios.

The work of Calame et al. [33] has probably the most in common with the work described in this report. They ran test cases defined in the TTCN-3 language against the interlocking logic of VPI interlocking systems. The interlocking logic was translated to a modelling language which could be executed in a simulator on a regular PC. The test cases were derived manually from a set of

given test scenarios, with the aid of a model that modelled part of the behaviour of the interlocking system [34]. However, information on what exactly was modelled, what the model was used for, and how that model was implemented is not available.

An important part of an ioco testing setup is the specification model of the SUT. Past research also used models of interlocking logic, but these models were generally generated from the interlocking code itself (see section 1.3.1). This is not an option for model based testing, since this would copy any implementation flaws to the specification model. The literature offers no clear examples of complete interlocking logic models that are constructed purely from a specification.

1.4 Report Structure

The first part of this report provides general background information on railway safety in the Netherlands and the PLC-Interlocking system that is being tested. The second part of this report covers the actual research.

Chapter 2 gives an introduction to the techniques and procedures used to guarantee safety on the Dutch railways. The functions and responsibilities of interlocking systems and how they relate to the overall safety on the railways are explained.

Chapter 3 describes the PLC-Interlocking system. It describes the hardware and general architecture of the system, how it interacts with its environment, and how it is programmed.

The second part of the report starts at chapter 4, which gives a short introduction to ioco testing and describes the used test setup. Furthermore, this chapter also gives an introduction to the mCRL2 language which has been used for the implementation of the specification model.

Chapter 5 describes the behaviour of the PLC-Interlocking model that has been created. Firstly, it covers the specification from which the model has been implemented. After that the transitions that the model uses to communicate with JTorX are covered, as well as a description of the model's functionality. How that behaviour is implemented using the mCRL2 language is detailed in chapter 6, which discusses the structure of the model, its execution flow, and the used data structures.

Chapter 7 discusses the executable that serves as the SUT. First, the C++ code that is generated during the compilation of the interlocking logic is discussed. This is followed by an explanation of how that code is used in the executable, and what additional code has been written. The implementation of the interfaces to the interlocking logic is also explained.

Chapter 8 describes why random test generation does not lead to useful test cases, and describes the test purpose models used to steer test case generation. These test purpose models are implemented in Java using a custom built framework. This framework is the subject of chapter 9.

The created test setup has been tested in over 70 test runs. Chapter 10 discusses the execution of these test runs, and the data gathered during these

test runs. Both the performance of the test setup and its ability to find faults are discussed in detail.

Finally, chapter 11 gives conclusions and recommendations for future research.

This report contains a lot of terminology that is not commonly used but is specific to the railway industry, or even specific to the Dutch railway industry. Furthermore, some model based testing terminology is used as well as some other terms which are otherwise not used commonly. The glossary at the end of this report gives an overview of the used terms and their meaning. Possibly interesting to readers that are not familiar with English railway terminology at all is the glossary with translations created by the INESS projetct [35].

1.5 Sources

Chapters 2 and 3, and parts of the chapters 1 and 4 are based on texts from an earlier report [1], which was written for the course *research topics* and details the preliminary research.

This report presents a lot of information on the workings of the Dutch railway infrastructure. All this information, unless credited otherwise, comes from Dutch signalling course manuals [36, 37], books on signalling [38] and on the Dutch railway infrastructure [39] and from papers [11, 12, 13], but mostly from personal communications with railway signalling experts from ProRail and Movares [40, 6, 7].

All information regarding the PLC-Interlocking, unless credited otherwise, has been gathered from the PLC system's manuals [41, 42], PLC-Interlocking design documents [5, 43] and personal communications with railway signalling experts and system engineers from ProRail and Movares [40, 6, 7, 44, 45].

Part I

Background

Chapter 2

Train Safety on the Dutch Railways

The Dutch railway infrastructure should at all times guarantee safe train movements. There are two concrete safety goals that have to be met:

1. No collisions
2. No derailments

Besides collisions among trains, this also includes collisions with other vehicles (at level crossings) and fixed objects (at the end of a track). Furthermore collisions with maintenance personnel or vehicles must be prevented at all times. Derailments can be caused by an interruption of the railway (e.g. a point in the wrong position) or by riding at an excessive speed.

A number of systems play an important part in ensuring that the safety requirements are met. These systems and their interactions are considered in the next three sections. The final section of this chapter details how railway engineers construct safe systems.

2.1 Train Control Architecture

The train control architecture consists of multiple systems that are spread over different layers. Figure 2.1 gives a schematic view of the different layers of the train control architecture, the systems in them, and the interactions between these systems.

On the lowest level (the physical layer) are all wayside elements such as signals, points, level crossings and movable bridges, but also train detection devices along the track. These elements control the movement of trains and relay information from and to trains and their drivers. The devices in the physical layer are controlled by and give feedback to a nearby interlocking system, which controls all elements in a certain area.

Each interlocking system is connected to a system in the logistics layer, from which it receives requests and to which it relays information. This logistics

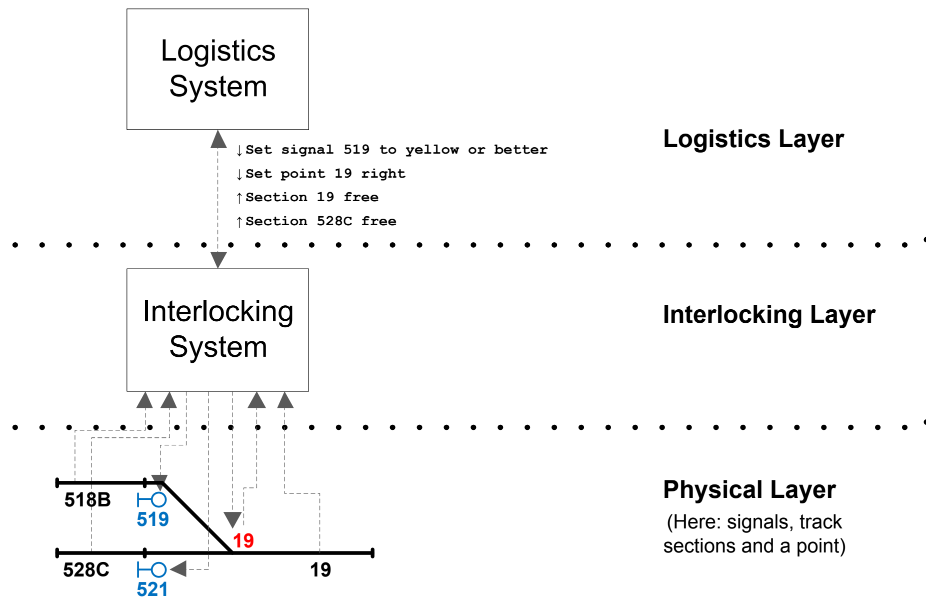


Figure 2.1: Train control architecture

system directs train movements over a bigger area and is controlled by human operators (called *signallers*) or computer systems executing set train schedules.

All wayside elements are connected with an interlocking system through one or more dedicated electrical wires. A wire either controls one function of the wayside element or gives a single bit of feedback information to the interlocking system. Each wire is linked (possibly through an external relay) with a single one-bit input or output port of an interlocking system.

The requests from a logistics system to an interlocking system are on the level of individual wayside elements (e.g. 'set point X in the left position'). Feedback from the interlocking system is also on the level of individual wayside elements (e.g. 'point X is positioned left'). The requests and feedback are communicated between the systems using a custom protocol over a long distance serial line.

2.1.1 Safety Responsibility

Systems in the logistics layer are not safety critical, neither is the communication between logistics systems and interlocking systems. The responsibility for train safety lies with the interlocking system that controls a certain area. This implies that interlocking systems need to detect and ignore potentially dangerous requests from the logistics layer.

Maintaining safety is of the utmost importance. Therefore interlocking systems and the devices they control are designed around the fail-safe philosophy. Concretely this means that if a system would fail that this might impact functionality, but should not impact safety, i.e. a failure might mean that trains cannot reach their destination, but should never lead to an accident.

Of course safety can only be guaranteed within the parameters that the systems were designed for. If people do not follow signalling or regulations (e.g. parking a car on a railroad crossing), accidents cannot always be prevented.

2.2 Devices in the Physical Layer

2.2.1 Signals

Interlocking systems communicate with train drivers through signals. There are different kinds of signals, the most common of which consists of three lamps (red, yellow, green) and possibly a number display which can show a number to indicate a speed ($speed = number \times 10\ km/h$). With these a signal can show certain *signalling aspects*, which convey instructions to the drivers of passing trains as shown in table 2.1.

The signalling protocol has a partly fail safe design. In case the signal fails altogether, trains will have to stop, which -although inconvenient- will be safe.

Table 2.1: Signalling aspects

Aspect	Indication
Nothing (failure)	Stop before the signal.
Red	Stop before the signal.
Yellow blinking	Signal can be passed; speed: < 40 km/h. Expect other trains on the track.
Yellow	Slow down, signal can be passed; speed: 40 km/h. Expect a red aspect at the next signal.
Yellow with number	Slow down, signal can be passed; speed: as indicated.
Green blinking	Signal can be passed; speed: 40 km/h.
Green with number	Signal can be passed; speed: as indicated.
Green	Signal can be passed; speed: the locally allowed speed.

There are restrictions on the sequences of signalling aspects that a train may encounter on a certain route. These restrictions are needed to allow train drivers to anticipate and slow down enough before a next signal. A general restriction is that a signal showing a green aspect cannot be followed by a signal showing a red aspect without a signal showing a yellow aspect first. The allowed sequences of aspects are determined per route and are noted on the so-called *OS-blad* drawings (OS-blad: Overzicht Seinbeelden blad (Dutch)).

Even when signals show the correct aspect, train drivers are fallible. Therefore all trains are equipped with the ATB system (ATB: Automatische Trein Beïnvloeding (Dutch)). There are two versions of this system. The older version transfers the allowed speed for a certain section via electric pulses that run through the tracks in that section. The newer version gives an allowed speed profile for an upcoming section through radio beacons at certain way points. The ATB systems will automatically enforce the signalled speed if the driver ignores it. However for practical reasons, ATB is not enforced everywhere at all speeds.

The European Train Control System (ETCS) (which is a part of the European Rail Traffic Management System (ERTMS)) is a more advanced European standard that has been designed to one day replace all national safety systems (such as ATB) in the European Union. The ETCS standard defines a number of implementation levels (0-3) that build up one another, with each successive level being more advanced. A select number of railway lines in the Netherlands have been equipped with ETCS level 2 safety systems. ETCS level 2 has significant safety advantages over ATB and makes physical signals obsolete altogether (all communication with train drivers is done through radio signals). However as said, it has only been installed on a select number of railway lines in the Netherlands; the great majority of railway lines in the Netherlands are still equipped with regular signals and will be for many years to come.

2.2.2 Points

Points (or in American English: railroad switches) are the only means to alter the direction of a train, and are as such essential for railway operations. There are different kinds of points, but the most common model has two positions: a straight track (*normal*) and a curved track (*reverse*).

Points also pose a potential danger. A train might derail if it drives over a point that is not set fixed in the correct position. A number of problems (e.g. ice chunks) can cause a point not to be in the correct position, despite a command from the interlocking. Therefore an interlocking can only allow train travel over a point if the point explicitly signals that it is in a correct position.

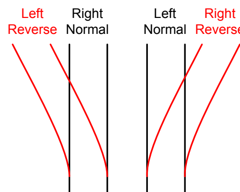


Figure 2.2: Point positions

In the Dutch railway world the terms ‘left’ and ‘right’ (which can be either normal or reverse depending on the point, see figure 2.2) are used to indicate the direction in which a train can travel. International literature often uses normal and reverse to indicate the position of a point, or even left and right with the opposite meaning.

Note that this report uses the term ‘point’ for the singular case, however it is also not uncommon to use ‘points’ or ‘set of points’ as singular.

2.2.3 Train Detection

For interlocking systems to guide trains, they must know where trains are. There are different kinds of train detection systems for this goal. One system relies on the train shorting an electric circuit, while another system counts the magnetic disturbances caused by train wheels passing detectors. With all systems the

railway track is divided in sections. Per section there is a signal indicating whether the section is free or occupied.

All systems have their drawbacks which on rare occasions can lead them to detect trains that are not there or not detect a train that is in fact there. Obviously this is a threat to train safety. If this is not a structural problem, but merely caused by temporary bad conductivity, then this can be handled by the interlocking logic. An interlocking will not accept that a track section is cleared by a train, unless it is detected in one of the next sections. Incorrect detection can also lead to problems that have to be solved outside of the system. For example by signallers ordering trains to drive through red signals slowly.

2.3 Interlocking Systems

The area covered by an interlocking system consists typically of a single railway yard or railway station with the surrounding area, but can also contain longer stretches of rail. Literature often refers to the controlled area as 'railway yard', regardless whether it actually is a railway yard, a station, or a stretch of a railway line. This report follows that convention: The term railway yard will be used to identify the area controlled by a single interlocking system.

The number of input and output ports that an interlocking system needs to handle, depends on the complexity of the railway yard that it is installed on. A yard might contain less than a dozen points and signals, but there are also yards with several hundred points and signals. An interlocking also needs to consider signals it might receive from neighbouring interlocking systems about trains near exits and entrances to the controlled railway yard.

Because of their safety function, interlocking systems must be absolutely fail-safe. The overall design of systems and their communication enables this: When an interlocking system gives no outputs, then all signals will turn red. However that in itself secures not much. For an adequate level of safety the hardware and software quality of an interlocking are crucial. The European standards organisation CENELEC has set standards for the development of hardware and software for interlocking systems, which ProRail adheres to. One aspect of this is that the hardware platform of the PLC-Interlocking in combination with its tooling had to have the highest possible Safety Integrity Level rating: SIL-4. A system with a SIL-4 rating is extremely unlikely to have an unsafe failure.

2.3.1 Logic

To guarantee the safety on the railways, interlocking systems implement (simple) rules.

On long stretches of rail the so called *line block* system is used. The railway is divided in blocks, with the start of each block secured by a signal. The exact rules depend on the kind of railway stretch, but the next scenario (shown in figure 2.3) is common for single track railways: After the signaller has set a direction of travel on a railway stretch, all signals in the opposite direction (not shown in the diagram) will be set to red. The signals in the direction of

travel will be set to green, but only if the next block can be safely entered by a train (i.e. no collisions possible). After a train passes a signal, the current block obviously does not fulfil that requirement anymore, so the signal turns red, one signal back turns yellow, while yet another signal back can be turned to green again.

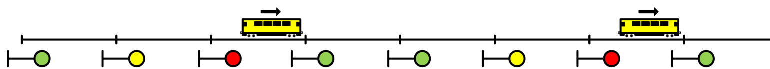


Figure 2.3: Line block system on a single track

Interlocking systems must implement more rules for railway yards than for straight stretches of railway, since there are more possibilities for collisions and derailments. On railway yards, the logistics system has to explicitly request the interlocking system to create routes for trains; all signals are set to red by default. In the context of railway yards and interlocking systems, a single route runs from one signal to the next. Only if a route is safe will it be implemented by the interlocking system (i.e. its entry signal will be set to a non-red aspect). All elements of a route are blocked to other trains by signals and points, and cannot be used in other routes until the train has passed completely. A common exception is when trains couple, for which they obviously have to be on the same track.

The above two paragraphs are merely to illustrate the main points, and omit a lot of details, possible problems and exceptions, which actual interlocking systems do take into consideration.

An important idea, which cannot go unmentioned, is the prevention of accidents that might result from failures of other systems (e.g. the train driver or the train detection system). An example of this is flank protection: An interlocking will often change the position of points without an explicit request from the logistics system to do so. This is done to make sure that no collisions can occur, even if a train were to overrun a red signal. Because of a number of reasons, each year more than a hundred trains run through a red signal. So although the obedience to signals is one of the foundations of train safety, interlocking systems do not just rely on that. At the same time, driver errors and other failures that violate rail protocol can cause unsafe situations that an interlocking cannot always prevent.

2.3.2 Interlocking System Types

There are several types of interlocking systems in use in the Netherlands.

The most common technique in use at the moment is based on relays. In such a system all interlocking logic is expressed by the relays and their interconnections. Even though this is a very simple technology, it certainly is possible to build big interlocking systems with it. The interlocking systems at the two busiest (in terms of daily passengers) railway station in the Netherlands (Utrecht Centraal station and Amsterdam Centraal station) are built entirely from relays.

More modern interlocking system types are based on microprocessors. One of the most used types, which has been in use since the early 1990s, is the already in section 1.3.1 mentioned Vital Processor Interlocking (VPI). A VPI works in a one second cycle in which it reads all inputs, evaluates a set of Boolean formulae and writes to the output ports. A lot of the programming for PLC-Interlocking systems is based on the programming of these VPI systems. The programming of the VPI systems in turn was based on the design of relay based interlocking systems.

There are also some other computer based interlocking systems, but they are only used in a few locations. The design of their programming is generally not as much inspired by earlier interlocking systems, but rather based on proprietary techniques developed by the manufacturers.

Although there are significant difference in the way that interlocking systems are implemented, all currently used systems fulfil the same functions. When observed as black boxes, the behaviour of most interlocking types will be the same in most situations (modulo timing differences).

2.4 Railway Design and Verification Process

The programming of safe interlocking logic is part of a bigger process. This starts with the design of the track and signal layout. After that follows a detailed design of the signalling installation and the interlocking programming. This is then again followed by a design step in which the layout and wiring of all technical installations is specified.

Throughout the design process, the designers have to implement the standards that are set in design instruction documents, the *OVS* documents (OVS: OntwerpVoorSchrift (Dutch)). For the PLC-Interlocking for example, the OVS Application Engineering [5] prescribes exactly what logical functions to use, how to use them and in which situations to use them.

After each step, the created designs are verified by a team of independent experts that work according to a set protocol. During verification the experts check whether the OVS's standards have been followed precisely. For the interlocking logic this means that two persons verify that the interlocking design implements the logic exactly as can be expected from the track layout design.

Verification does not stop when the design process is completed. After the design is finalised and the interlocking system built, testing starts. During the so called *Logic Safety Test*, tests are done in a laboratory environment using the interlocking hardware in its final configuration. A test plan is executed to verify that the interlocking system was built correctly.

Note that in this process it is assumed that the OVS documents are correct. For interlocking systems specifically, it is assumed that individual pieces of logic are correct and that their (correct) composition gives a safe system. Mostly this has not been formally proven. There is an extensive process to verify the correctness of new interlocking logic. However, in the end this process boils down to expert judgments based on reasoning about the logic; true proofs are usually not constructed. This is not without reason. As the preliminary research

[1] showed, it is often incredibly hard to prove a property for interlocking logic. Nonetheless, confidence in the correctness of the interlocking logic does not seem misplaced; so far there has never been an accident that could be attributed to a fault in the logic of an electronic interlocking system.

Chapter 3

PLC-Interlocking Systems

The PLC-Interlocking is the result of a development project that is still running. This project has so far yielded two main products that are of interest for the research on model based testing:

- A set of OVS documents that prescribe in detail how to build a PLC-Interlocking system for a given railway yard.
- A first instance of a PLC-Interlocking system.

In this report, the term 'PLC-Interlocking' is often used to refer to the PLC-Interlocking system architecture in general, i.e. the specifications laid out by the OVS documents. However, the term is also used to refer to a specific PLC-Interlocking system (this will be apparent from the context when this is the case).

This chapter describes the architecture of the PLC-Interlocking and the working of its main components with special attention to the interlocking logic. Furthermore the first PLC-Interlocking instance, which is the subject of the testing research described in this report, is shortly covered.

3.1 Programmable Logic Controllers

The core of any interlocking system is the interlocking logic. In a PLC-Interlocking system this logic runs on a Programmable Logic Controller (PLC), hence the name PLC-Interlocking.

PLCs are computer systems that are often used to control industrial processes. They can be acquired as commercial off-the-shelf hardware, and are then tailored to specific situations through configuration and programming. PLCs constantly execute the same program, which can receive input from and give output to a great number (hundreds) of external devices. This makes them a good match for railway situations where a lot of wayside elements need to be controlled.

The PLC-Interlocking is built around the HIMA HIMax[1] PLC platform. HIMax PLCs are built from CPU, input, output and communication modules that are mounted on one or more base plates, possibly spread over multiple physical locations, and connected through redundant system buses. The HIMax PLCs used for PLC-Interlockings are equipped with redundant processor and input modules to increase reliability, availability and safety. Synchronisation between the different modules is completely transparent to the software (i.e. the interlocking logic) that runs on the systems. The resulting platform (including its tooling) is SIL-4 certified.

The main function of a PLC system is executed single-threaded and consists of a constant loop, the so called *scan cycle*. During each scan cycle a PLC does the following (in order):

1. Read values from the input ports.
2. Execute the loaded program.
3. Write the computed values to the output ports.

In step 1 the system takes a snapshot of the values at all input ports simultaneously. Each physical input port is coupled with an input variable. These input variables are set in accordance with the read inputs and do not change value for the remainder of the scan cycle.

The program in step 2 must be defined by the system's programmer. This program takes the input variables as well as internal variables as arguments. Using those it computes new values for the output variables and the internal variables. In the PLC-Interlocking this program is responsible for handling the interlocking logic (more on that in section 3.3).

After the program from step 2 has finished executing, the PLC updates all output ports simultaneously in step 3. Each output variable is coupled with a physical output port that (after the update) outputs the variable's value. After the output ports are set, the next scan cycle starts.

In general, the length of a scan cycle depends on the used hardware, the program that is running, and the number of inputs and outputs.

For PLC-Interlockings the average cycle will take between 20 and 200 milliseconds, depending on the size of the installation. The cycle length for any PLC-Interlocking installation is variable, but the deviation between cycles should be small (in the order of a few per cent). Only in exceptional circumstance (e.g. when hot-swapping one of the CPU modules) can the length of a cycle deviate a lot from the average. In all cases there is an upper-limit on the cycle length, which is set to about one second. If that limit is exceeded, the system will automatically be turned-off (and thus be in a safe state).

3.2 PLC-Interlocking Architecture

As has been noted already, the core of a PLC-Interlocking system is formed by the interlocking logic, that runs on the PLC hardware. However a complete

[1] `http://www.hima.com/Products/HIMax_default.php`

PLC-Interlocking system consists of more components than just the PLC hardware. There are external systems that are responsible for the power supply, data logging, diagnostics, maintenance, communication and the connections with the wayside elements. Figure 3.1 shows a schematic view of a PLC-Interlocking and its high-level components. The complete installation is housed in one or more relay-houses and relay-boxes.

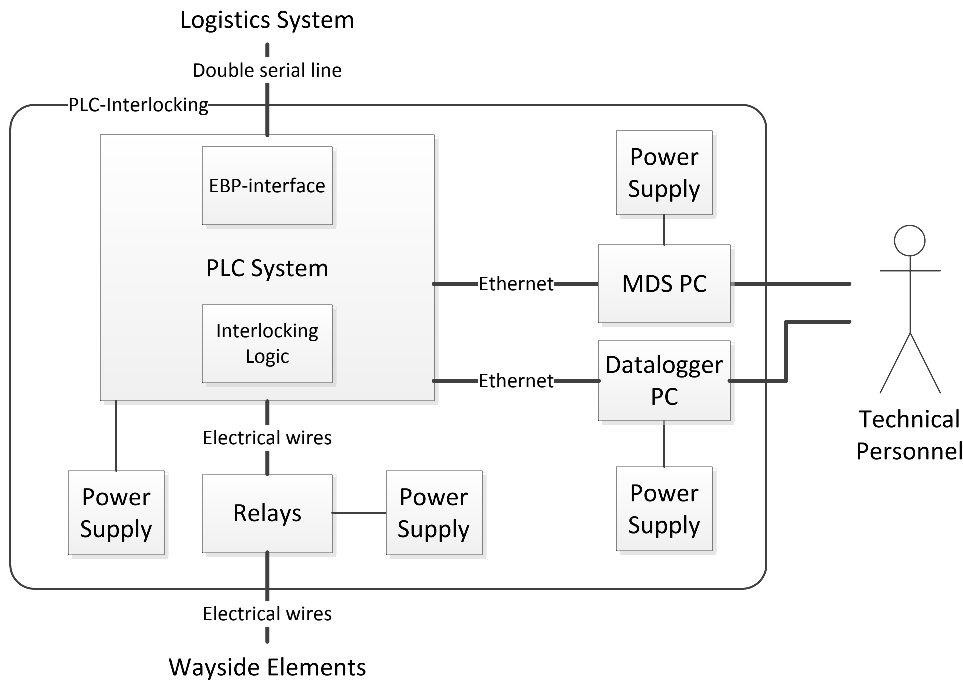


Figure 3.1: The components of a PLC-Interlocking system

Two regular PC systems, the Datalogger and the Maintenance and Diagnostics Subsystem (MDS), are responsible for data logging, and maintenance and diagnostics. These systems are connected to the PLC via Ethernet interfaces and run software that communicates with the PLC using the standardised OPC protocol. During laboratory testing, such an OPC connection is also used to manipulate the input variables and read out the output variables. Manipulating variables is of course not possible during normal operations.

The PLC hardware is not directly connected with the wayside elements. All inputs and outputs of the PLC system are connected to relays. These relays either switch the input signal for the PLC based on the external signal, or switch the external signal based on the output of the PLC. Using the relays allows the PLC-Interlocking to switch different voltages and more powerful currents than the PLC system can handle.

The PLC system handles the interlocking function and is of course the main component of the PLC-Interlocking.

However the PLC hardware not only hosts the interlocking logic, it also hosts another piece of software: the *EBP-interface* (EBP: Elektronische BedienPost (Dutch)). The EBP-interface handles communication with the higher-level logistics system that requests routes for trains (see also figure 2.1). The correct working of this component is very important to the well-functioning of railway traffic, but in general not critical to safety.

The interlocking logic on its own should at all times guarantee safety; faults in the EBP-interface should in no way affect this. This is secured on the hardware side by the fact that the EBP-interface runs on a separate communication module that has its own processor which runs independent from the main computation cycle. Communication between the EBP-interface and the interlocking logic is done asynchronously through shared variables that are latched at the beginning of a logic cycle.

Together, the different components of the PLC-Interlocking run a significant amount of software. A lot of that software (e.g. everything running on the Datalogger) is not safety critical. Some of it is safety critical but can be assumed to be safe (e.g. the firmware handling hot swapping of CPU modules) because of the PLC system's certifications.

Only the interlocking logic is both specifically designed for the PLC-Interlocking and truly safety critical. Furthermore it implements the core functionality of the entire PLC-Interlocking. Because of that, the research detailed in this report was focused on the interlocking logic.

3.3 Interlocking Logic

The interlocking code is executed every scan cycle. On every execution it evaluates the inputs received from the environment and computes new outputs to enable safe train movements.

The code is focused on the interlocking task and has to handle almost no other concerns. For example, there is no code needed for using multiple CPU modules. The PLC firmware automatically executes the code in parallel on all available CPUs, as well as handle CPU failures. Comparable, all input and output happens through global variables. The code is not aware of which physical i/o modules and ports belong to which variables; that is dealt with in the configuration.

There are a few exceptions to this though. There is a function that checks the integrity of the system and its operating conditions (e.g. check the temperature sensors and the self-check values of the system's modules). Another function verifies that the software's checksum matches a supplied value. However these are exceptions, the great majority of the code is used to implement the interlocking task.

3.3.1 Function Block Diagram Programming

The interlocking code is written using the Function Block Diagram (FBD) language. This is a language that was standardised by the IEC standards organisation in part 3 of their standard for PLC architectures [46].

A program in FBD notation is built up from function blocks that take one or more inputs and give one or more outputs. These function blocks can be connected such that (part of) the output of one block serves as (part of) the input for a next block. There are standard blocks defined for common operators, but also for more complex functions such as timers. Furthermore, the programmer can define his/her own function blocks by combining and connecting existing function blocks. Inputs and outputs of function blocks can be connected so that the values are read from or written to variables. This way it is possible to read or save state, but also to manipulate the inputs and outputs of the PLC, which are mapped to global variables. The composed FBDs are executed left-to-right top-to-bottom.

Programming in the FBD language is normally not done by using a textual representation, but rather by creating and manipulating a graphical representation of the function blocks and their connections. For HIMA PLCs this is done on a regular PC using the *SILworX*[2] development environment.

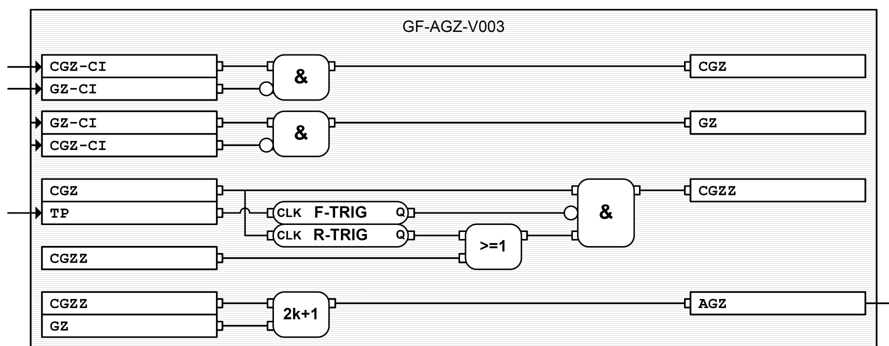


Figure 3.2: A snippet of FBD program code

Figure 3.2 gives an example of (part of) an FBD program. The sample shows a function block that has three inputs (`CGZ-CI`, `GZ-CI` and `TP`), one output (`AGZ`) and three local variables (`CGZ`, `GZ` and `CGZZ`). All the variables in this example happen to be Booleans. The function block connects the variables using a number of standard function blocks: AND (‘`&`’), OR (‘`>=1`’), XOR (‘`2k+1`’), F_TRIG and R_TRIG, as well as the NOT operator (the small circle at some function block inputs). More information regarding the semantics of the used function blocks can be found in table 3.1.

3.3.2 Used Functions and Types

The FBD language supports numerous value types and comes with an extensive collection of standard function blocks. However, only a restricted subset of the available standard FBD functions and value types is used in the interlocking implementation.

[2]`http://www.hima.com/Products/SILworX_default.php`

All inputs and outputs from the wayside elements and the EBP-interface are Booleans. Only in the code that checks the system's integrity are some integer inputs (e.g. the temperature sensor returns an integer). These non-Boolean values all originate from within the PLC system; either from internal sensors or from internal variables.

Table 3.1: Used standard function blocks

Function	Inputs	Description
AND	2 to 16	Logical AND.
OR	2 to 16	Logical OR.
XOR	2 to 16	Logical XOR: True iff an odd number of inputs is true.
NOT	1	Logical NOT.
TON	1 + 1	Time On delay: True iff the input Boolean has been true uninterrupted for at least the specified time.
TOF	1 + 1	Time Off delay: False iff the input Boolean has been false uninterrupted for at least the specified time.
R_TRIG	1	Rising Trigger: True iff the input is true this cycle and was false last cycle.
F_TRIG	1	Falling Trigger: True iff the input is false this cycle and was true last cycle.
SR	2	SR flip-flop: $output = input1 \vee (\neg input2 \wedge output)$
EQ	2 to 16	Equal: True iff all inputs have the same value.
NE	2 to 16	Not Equal: True iff none of the inputs have the same value.
LE	2	Less or Equal: True iff $input1 \leq input2$.
GE	2	Greater or Equal: True iff $input1 \geq input2$.
MOVE	1 + 1	Move: If input Boolean = true: *input*, else: datatype's default value.
SEL	2 + 1	Select: If input Boolean = true: $input1$, else: $input2$.

Table 3.1 shows the subset of standard function blocks that is used in PLC-Interlocking code. The functions are divided in three groups.

All functionality that is directly interlocking related is implemented using only function blocks from the first two groups. The functions in the bottom group are only used in functions that verify the system's integrity.

The first group contains the standard Boolean operators, while the second group consists of more complicated functions which keep an internal state between invocations. These functions are only used with Boolean inputs, with the exception of the TON and TOF functions which take a Boolean and a real value. In both these cases the real value is always a compile time constant and expresses the length of a time interval. The functions from the third group are only used with integer values as input, with the exception of MOVE and SEL which, besides their integer input(s), also require a single Boolean input.

3.3.3 Interfaces

The interlocking logic communicates with external systems solely through global (Boolean) variables. These variables are either mapped to physical input or output ports, or are mapped to variables in the EBP-interface component.

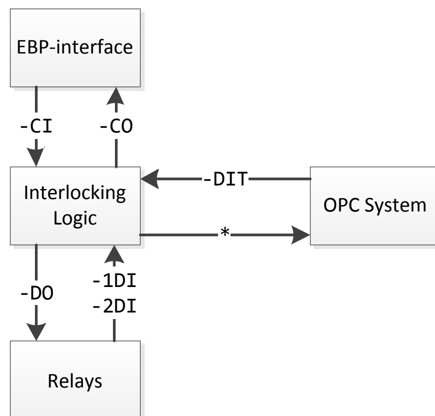


Figure 3.3: Interfaces of the interlocking logic

Interactions between the interlocking logic and external systems (including the EBP-interface) will never influence the duration of a scan cycle. All input variables are latched at the beginning of a scan cycle and all output variables are written at the end. During both input and output, the logic does not wait on external systems.

A naming scheme has been instituted for the variables to keep some order among them. All variables that are used in communication with other modules/systems have fixed suffixes. Figure 3.3 shows the communication of the interlocking logic with other systems through variables and their suffixes.

Communication with the EBP-interface is done through variables suffixed with `-CI` and `-CO`. The input from wayside elements (which runs through relays) is received redundantly (via `-1DI` and `-2DI`). A function block compares the two values to detect possible problems. If both values are true, then the corresponding variable with the suffix `-DI` is set to true.

Internally, the interlocking logic uses variables suffixed `-DII` for input data from the wayside elements. These variables are set every cycle by function blocks. During laboratory testing, the values of these variables are copied from test variables with the suffix `-DIT`. The value of these test variables can be set from a system connected over Ethernet using the OPC protocol. Normally however, the `-DII` variables are set to the value of the corresponding `-DI` variables.

Also shown in the diagram is that external PCs connected with the OPC protocol (during normal operations: the Datalogger and MDS PCs, see section 3.2) can read all variables of the interlocking logic.

The physical interfaces between wayside elements and the PLC-Interlocking are documented in *Interface Requirements Specifications*. The OVS Application Engineering [5] describes exactly how those inputs and outputs are mapped to variables. Depending on the specific track-layout there are multiple variables per wayside element to keep track of the status of said wayside element. The name of such variables depends on the purposes of that variable (e.g. it might indicate whether a section is free) and which specific element of that kind it

is. All such variables refer to wayside elements by an identifying number that matches the number on the track layout maps.

The EBP-interface serializes and deserialises communication between the interlocking logic and a remote logistics system. Typically EBP requests are to set a point or a signal. Typical feedback information is the position of points, occupancy of sections and the aspect of signals. The variables to communicate this information between the EBP-interface and the interlocking logic depend on the track layout; the naming scheme for this is also documented in the OVS Application Engineering [5].

3.3.4 Programming Specific Instances

The OVS Application Engineering prescribes exactly how the interlocking logic needs to be constructed for a specific installation. There are 30 types of functions and function templates that need to be wired together such that they form correct interlocking logic.

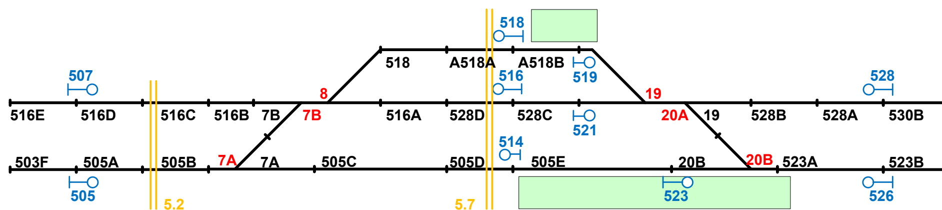
Typically a function has a generic template, which is then applied by the engineers to fit in the program of a specific track layout. Generic functions are defined as function blocks that can be applied by merely connecting them with the correct variables. The exact variables names are prescribed for (virtually) all possible track layouts and situations.

However, the exact layout of a railway yard is (almost) always unique, and the generic function blocks only cover the common cases. As a result the complete logic also always contains 'free-wired-logic', i.e. customly connected function blocks. How this custom logic should be constructed is also outlined in the OVS Application Engineering.

The generic function block diagram in figure 3.2 is a good example of logic that is very generic. This function block is used as part of the overall logic that determines which aspect a signal should show. Typically this generic function block is used for every signal in a yard and each time wired to different variables. Two of the block's input variables are wired to input signals from the EBP-interface, the other input variable is wired to a global variable indicating the occupations of the section directly behind the signal. The output variable is wired to a global variable that is read by other function blocks that decide which aspect the signal should show.

Other parts of the logic that determine the signalling aspects need more customisation depending on the track layout. The OVS Application Engineering prescribes how to do this, for example it prescribes: which variables names to use for which aspect, how to check the track occupation in the different routes that are possible from a signal, how to handle interlocking boundaries behind a signal and many more details.

The used process (see section 2.4) should ensure that the engineering team applies the guidelines from the OVS document correctly and completely. This is essential in order to get a correctly functioning and safe interlocking system.



Legend

All wayside elements have an alphanumeric identifier.	
Black	Track sections
Blue	Signals
Red	Points
Yellow	Level crossings
Light green	Platforms

Figure 3.4: Schematic view of the Santpoort Noord station area

3.4 First Instance: Santpoort Noord

The first and currently only instance of a PLC-Interlocking is deployed at the small railway station Santpoort Noord. A second PLC-Interlocking is planned at the nearby Beverwijk railway station, which is bigger. However, during the research period the interlocking logic for that installation had not yet been developed. Thus the Santpoort Noord installation was the only available case study, and all testing efforts were applied to the interlocking logic of that installation.

Santpoort Noord is a small railway station with three parallel tracks. Train traffic is directed over 6 points by 10 signals, train detection is done with 25 track sections and there are 2 level crossings. A schematic view of the track layout can be seen in figure 3.4.

The PLC-Interlocking for Santpoort Noord is already quite complicated. The physical system has 70 ($\times 2$) redundant input ports and 110 output ports connected with wayside elements. The interlocking code contains over a 1000 Boolean variables (input variables, output variables and variables keeping state between cycles) and almost 400 timings blocks.

Part II

Testing a PLC-Interlocking

Chapter 4

Model Based Testing

In general, testing is used to assess (and improve) the quality of the System Under Test (SUT) by interacting with it and observing its behaviour. Model based testing (partly) automates this process by using models for the generation of test cases and as oracle to verify observed outputs.

The main goal of the conducted research has been to develop and use a model based test setup for the PLC-Interlocking. This chapter describes the used test methodology (section 4.1) and a high level overview of the created test setup (section 4.2).

Later chapters cover the different parts of the test setup in more detail. Some background information needed for those chapters is also covered in this chapter: Section 4.3 gives an introduction to the modelling language used for the specification model.

4.1 Input-Output Conformance Testing

The method used in testing the PLC-Interlocking is based on the input-output conformance (ioco) theory.

The ioco testing theory and related theories have been extensively covered in the literature. Timmer et al. [2] comprehensively surveyed this testing theory and its formal underpinning. The allowed behaviour of the SUT is expressed in a specification model based on a Labelled Transition System (LTS), where the transitions express input and output actions. This model is also used to derive (on the fly) test cases.

The theoretical basis of the ioco theory is formed by the ioco conformance relation, which defines under what conditions an implementation conforms to a specification. An implementation ioco-conforms to a specification, if at all times it can handle at least all inputs from the specification, and produces at most all outputs from the specification. With the exception that the implementation is not allowed not to provide any output when the specification requires one.

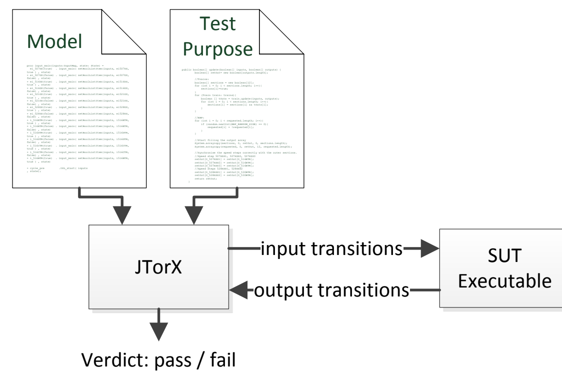


Figure 4.1: JTorX test setup

Practical application of this theory requires tooling. JTorX [3] is an automated test tool that can test whether an implementation ioco-conforms to a specification. It automatically derives test cases from a specification model and then executes those on the SUT. If JTorX observes a failure during the execution of a test case it will halt execution and report a *fail* verdict. Otherwise, if no failures are observed during the execution of a test case a *pass* verdict will be given after execution has finished.

Interaction with the SUT can be done through custom programmed adapters, allowing integration with all sorts of systems. JTorX also allows the use of a *test purpose* (also called *model guide*). This is an extra model that is put in parallel (on the fly) with the specification model. This is useful for limiting the generated test cases, without having to modify the specification model.

Figure 4.1 (a repeat of figure 1.1) gives a schematic view of a typical JTorX test setup.

Although its roots are in academics, JTorX has also seen usage in the industry. Sijtema et al. [30] used JTorX to test a new protocol implementation, and reported finding some very subtle bugs in the implementation that might have gone undetected otherwise. Meijer reported finding faults during a case study in which an X-ray detector was tested using JTorX [31]. Earlier, JTorX's predecessor TorX had been used in several case studies already (all summarised in [26]).

Ioco based testing techniques are also used commercially (although seldom), which generally remains unpublished. In fact, a part of the PLC-Interlocking was tested by an external party that specializes in model based testing. They used a model based testing approach to test the EBP-interface component, specifically its communication with the logistics system. These tests highlighted some imprecisions in the specification, which led to differences between the intended behaviour and the initially implemented behaviour.

So applying (ioco) model based testing to a real product is certainly not unprecedented. However using it to test the interlocking logic of an interlocking system seems unprecedented; at least there is no literature describing such research. As such the research in this report is novel.

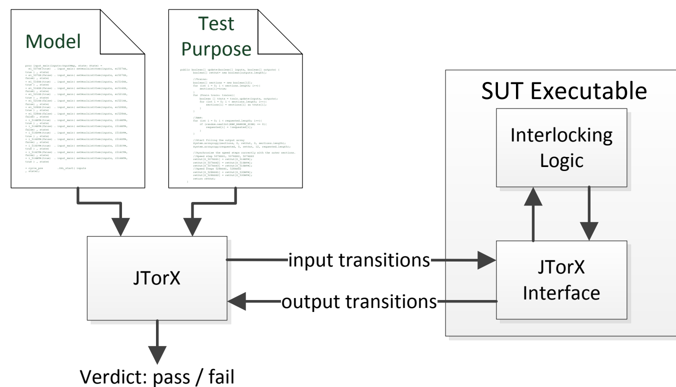


Figure 4.2: JTorX and the SUT in the test setup

4.2 Test Setup

As figure 4.1 already showed, an ioco model based testing setup with JTorX consists of the following ingredients:

- A specification model.
- An SUT.
- A test purpose model (optional).
- The JTorX test tool.

All four ingredients are used in the PLC-Interlocking test setup. The JTorX tool is a given, but the other three parts of the test setup either had to be developed or modified.

4.2.1 SUT and Models

The used SUT is a stand-alone executable that runs on a regular PC and does not require actual PLC hardware. This executable is based on the code of the regular PLC executable, with an added interface for communication between JTorX and the interlocking logic.

During a test run, the inputs given to the SUT will match exactly with the inputs given to the specification model, and each output of the SUT must also match with an output of the specification model. JTorX works only with LTS based models where all inputs and outputs are modelled by transitions, so all inputs and outputs to the SUT will have to be transitions also.

A PLC-Interlocking works in a cycle with an input phase, a computation phase and an output phase. The created model also models this cycle, and uses transitions for the inputs and outputs, and to signal transition between the different phases. Chapter 5, section 5.3 covers in detail the transitions used and how they represent the PLC's normal cycle and input/output behaviour.

The interfacing code of the SUT communicates with JTorX via the standard input and output streams. This code is responsible for decoding input transitions into inputs to the interlocking logic, and transforming all outputs from

the interlocking logic in output transitions. Figure 4.2, which is a refinement of figure 4.1, shows the interaction between JTorX and the interlocking logic which is wrapped in the SUT. More information on the SUT can be found in Chapter 7.

The specification model has been implemented in mCRL2 [47]. mCRL2 is a formal specification language that has a lot of useful abstractions, but can still be used to present an LTS to JTorX. Section 4.3 gives a short introduction to the mCRL2 language. For more information on the specification model consult chapters 5 and 6.

Multiple test purpose models have been created in order to generate different kinds of test scenarios. These test purpose models were not implemented in mCRL2 like the specification model, but in Java. Java was chosen because it is (arguable) a more powerful language and it seemed a better choice than mCRL2 given the feature sets of both Java and mCRL2 (for more in-depth reasoning see section 8.2.1). A Java program does not translate to an LTS naturally, like an mCRL2 model does. This was solved by creating a small framework that can wrap an actual test purpose model and create an LTS representation from its inputs and outputs. The usage and implementation of the test purpose models is discussed in detail in chapters 8 and 9.

4.2.2 Test Run Execution and Configuration

A test run is always initiated by JTorX, which is responsible for starting the other parts of the test setup. During a test run, JTorX will randomly explore the specification model, and give inputs to the SUT for any input transitions encountered and match outputs from the SUT to output transitions in the model.

JTorX needs to be made aware of which actions are input actions and which are output actions, because the model actually does not differentiate between input transitions and output transitions. Thus for JTorX to know when to apply stimuli and when to observe, it needs to know which transitions represent inputs and which represent outputs.

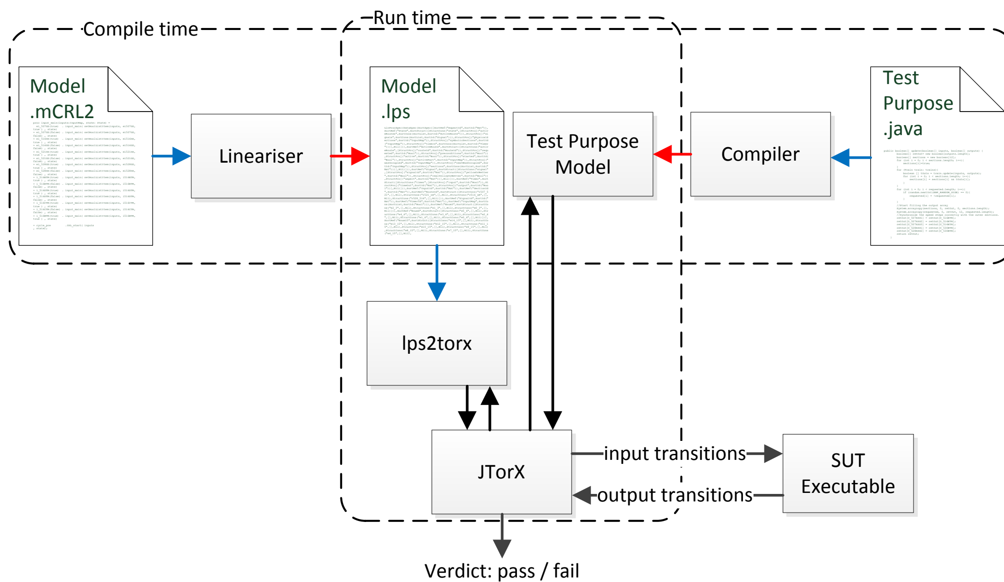
The main configuration details for the test setup, such as the classification of the transitions and the paths to the models and SUT, are stored in a file that JTorX reads at start-up.

Configuration of the SUT is done with a script that is executed by the SUT. This script sets all variables that are not part of the model but are nonetheless important to the interlocking logic. Such variables include input variables from points (which are not used in the model) and internal variables that indicate the status of PLC hardware parts such as cooling fans and input modules.

4.2.3 On the Fly Model Exploration

Figure 4.1 is actually a severe simplification in that JTorX cannot directly interpret a model written in Java or mCRL2, but requires an LTS representation of that model. JTorX can either take a complete LTS as input, or explore an LTS *on-the-fly* with the aid of an external program.

The mCRL2 toolset[1] can easily convert simple mCRL2 models to LTSs. However, for a lot of models (e.g. the specification model used here) such an LTS representation will be too big to generate. This is not a problem because of JTorX's ability to use on-the-fly exploration: JTorX does not need to know the structure of the whole LTS that underlies a model; only relevant states and transitions have to be explored, and only when they are needed. With on-the-fly exploration, an intermediate program, a *torx-explorer*, is responsible for interpreting the model. This torx-explorer communicates with JTorX via a set protocol over the standard input and output streams. JTorX drives the model exploration by request the torx-explorer to explore certain transitions, the torx-explorer will reply to such requests with information on the transitions that are possible in the state following the explored transition.



Legend	
Red	Writing file.
Blue	Reading file.
Unlabelled Black	Torx-explorer protocol over standard input/output streams.

Figure 4.3: Compilation of models and JTorX's interaction with the models

For mCRL2 models there is an established method of using them with JTorX's on-the-fly exploration mode. An mCRL2 model is first linearised and converted to the Linear Process Specification (LPS) format, which is saved as a .lps file. The *lps2torx* tool is then used to interpret that .lps file on the fly and act as a torx-explorer.

Such a standard practice does not exist for Java based models. Therefore a framework was developed in Java that implements the torx-explorer protocol and creates a simple interface for the actual model code to interact with. The complete test purpose model (including the framework) is compiled to form

[1] `http://www.mcrl2.org/`

a Java program that acts toward JTorX as a torx-explorer. More details on the implementation of the test purpose model can be found in chapter 8, the torx-explorer protocol and the developed framework are discussed in chapter 9.

Figure 4.3, another refinement of figure 4.1, shows JTorX's interaction with the models' torx-explorers, and the relation between the torx-explorers and the original model code.

4.3 The mCRL2 Modelling Language

Chapter 6, which covers the specification model's implementation, assumes some basic knowledge about the mCRL2 language from the reader. However, mCRL2 is not a language that is commonly used, so some readers might not be familiar with the language. For those, this section provides a short, and very much incomplete, introduction to mCRL2 that covers all concepts needed to understand the code samples from chapter 6.

All information in this section is based on information found in the mCRL2 user manual [48] which is much more complete than this section.

4.3.1 Processes

The core of any mCRL2 model is formed by one or more *processes*. The primitive operation of processes is the *action*, which can represent any event. Each mCRL2 action corresponds to a transition in the LTS that the model expresses.

Listing 4.1 shows a very simple process `P` that does a single action `read`. Line 1 declares the existence of the `read` action, line 2 then defines the behaviour of the process `P`, and finally line 3 sets the process `P` to be the initial process of the model. When translated to a LTS this process has a single `read` transition from its initial state.

```
1 act read;
2 proc P = read;
3 init P;
```

Listing 4.1: Simple example model

Actions can be combined using the sequential composition operator (notation: '`.`') and the alternative composition operator (notation: '`+`'). The sequential composition operator puts actions in sequence, whereas the alternative composition operator creates a choice. Process identifier can also be used on the right hand side of a process definition to create recursion.

Listing 4.2 shows a model with a process `P` that can start with either a `read` or `write` action because of the `+` operator. A `read` is always followed by a second `read` because of the use of the `.` operator, just as a `write` is always followed by a second `write`. After either two actions, the process `P` is called again recursively. Strictly speaking this process is not 'called', because `P` is not a function that returns but a process; the execution simply continues with process `P`. However, this report does use the term 'called' by lack of a better term.

```
1 act read, write;
2 proc P = ((read . read) + (write . write)) . P;
3 init P;
```

Listing 4.2: Example model with sequential and alternative composition, and recursion

mCRL2 has a lot of facilities for modelling multiple parallel processes that can communicate using actions. This is very useful for things such as modelling distributed systems, but is not used in the created specification model and is thus not covered here.

4.3.2 Data

Data types in mCRL2 are called *sorts*.

Processes can have such data types as parameters, whose values are set by calling processes. And actions can have concrete data values as arguments.

A number of basic data types have been predefined in mCRL2, two of those are used in the model:

- `Bool`: Boolean.
- `Nat`: Natural number, i.e. an integer >= 0.

Listing 4.3 shows an example model that has two actions that take a Bool as a parameter (line 1), and one process with a Bool data parameter (line 4). Line 2 shows a `read` transition with the value `true` as argument, after the action the process `W` is called with `true` as parameter value. The behaviour of the model is very simple: It either starts with a `read(true)` action which is followed by a `write(true)` action, or with a `read(false)` action which is followed by a `write(false)` action. After two actions the model is back in the initial state.

```
1 act read, write: Bool;
2 proc R          = (read(true) . W(true)) +
3                   (read(false) . W(false));
4 proc W(b: Bool) = write(b) . R ;
5 init R;
```

Listing 4.3: Example model with data parameters

The '`sum <variable>:<type> .`' construction can be used as a shorthand notation when there are alternative paths for each value of a data type which are the same modulo the data values. For example in listing 4.3, the same actions are taken for both values of a `Bool`. The code in listing 4.4 uses the `sum` keyword and has exactly the same behaviour as the code in listing 4.3.

```
1 act read, write: Bool;
2 proc R          = sum bParam:Bool . (read(bParam) . W(bParam));
3 proc W(b: Bool) = write(b) . R ;
```

```
4 init R;
```

Listing 4.4: Example model with a summation

Process can behave differently based on the value of a data object by use of the '`c -> p1 <> p2`' construct, which translates to 'if `c` then `p1` else `p2`'.

Listing 4.5 shows a model that uses such a conditional construct. Based on the value of the parameter `b` to the process `W` on line 4, either `(write(1) . R)` happens if `b` is `true` or `(write(0) . R)` if `b` is `false`.

```
1 act read: Bool;
2     write: Nat;
3 proc R          = sum bParam:Bool . (read(bParam) . W(bParam));
4 proc W(b: Bool) = b -> (write(1) . R)
5                    <> (write(0) . R);
6 init R;
```

Listing 4.5: Example model with conditional behaviour

When mCRL2 models are translated to LTSs the actions become transitions. However, the transition labels are not the same as the action notation in mCRL2. For example, the action `read(true)` will become a transition with label `read!true` when used with JTorX.

4.3.3 Mappings

Mappings are functions that take zero or more data objects as argument. A map definition consists of three parts:

- A `map` statement declaring the name and type (or *sort* in mCRL2 terminology) of a mapping.
- `var` statements declaring the type and name of the parameter variables used in the equations that define the function of a mapping. No `var` statements are needed for mappings that have zero arguments or only have concrete values in the equations and no variables.
- `eqn` statements which are the equations that define how inputs map to outputs.

Listing 4.6 shows a model with two very simple mappings. The mapping `five` takes no arguments and simply returns a `Nat` (line 1); the returned value is 5 (line 2). The mapping `addTimesFive` takes two `Nat`s as arguments and returns a `Nat` (line 4). The returned `Nat` is the sum of the first two arguments multiplied by the the return value of `five`, i.e. $a + b \times 5$ (line 6).

The behaviour of the model from listing 4.6 is very straightforward: It performs the action `write(5)` (since $(0 + 1) * 5 = 5$), followed by the action `write(15)` (since $(1 + 2) * 5 = 15$), after which the model terminates.

```
1 map five: Nat;
2 eqn five = 5;
3
```

```
4 map addTimesFive: Nat # Nat -> Nat;
5 var a,b:Nat;
6 eqn addTimesFive(a,b)= (a+b) * five;
7
8 act write:Nat;
9 proc P = write(addTimesFive(0,1)) . write(addTimesFive(1,2));
10 init P;
```

Listing 4.6: Example model with a simple mapping

mCRL2 has a lot of predefined mappings that implement standard functions. Some of these mappings can also be used as infix operators (e.g. the plus operator already used in listing 4.6).

Examples of predefined mappings are the standard arithmetic operators (`+`, `-`, `*` and `/`) for `Nat`s, the logical operators (`!`, `&&`, `||` and `=>` (implication)) for `Bool`s, and comparison operators (`==`, `!=`, `<` and `>`) which are defined for all sorts. Other commonly used mappings are the `if(a,b,c)` mapping which returns `b` if `a` is true and `c` otherwise, and conversion mappings that map data types to more restrictive data types (e.g. `Int2Nat` to convert an `Int` into a `Nat`).

Mappings can also be defined recursively, an example of which is shown in listing 4.7. The recursive mapping `power` takes two arguments (`base` and `expo`) and returns the value of the power function $base^{expo}$. If the second argument to the mapping (`expo`) is 0 (line 3), then the function returns 1 (line 4). Otherwise the function returns the value of the recursive call `power(base, expo-1)` times `base` (line 5). Note that the return sort of `expo-1` is `Int` (which can be negative also), but can be converted to `Nat` because the condiction to the if mapping guarantees that `expo-1` is at least 0.

The resulting behaviour of the model from listing 4.7 is as follows: It performs the action `write(1)` (since $2^0 = 1$), followed by the action `write(2)` ($2^1 = 2$), followed by `write(4)`, followed by `write(8)` and `write(16)`, after which the model terminates.

```
1 map power: Nat # Nat  -> Nat;
2 var base,expo:Nat;
3 eqn power(base,expo) = if(expo == 0
4                          , 1
5                          , power(base, Int2Nat(expo-1)) * base
6                          );
7
8 act write:Nat;
9 proc P = write(power(2,0)) . write(power(2,1)) .
10          write(power(2,2)) . write(power(2,3)) . write(power(2,4));
11 init P;
```

Listing 4.7: Example model with a recursive mapping

With the current mCRL2 toolchain, which compiles .mCRL2 files to .lps files that are then interpreted, all mappings are implemented using rewriting of the data structures on which they operate. All data is implemented as expressions,

and the equations (`eqn`) of mappings are used as rewrite rules. This 'functional' paradigm is fundamental to how the mCRL2 tools work and the used file formats.

However, the performance of this approach can be very poor. Some of the early versions of the specification model had some serious performance problems, which were (in all likeliness) caused by rewrite expressions that got too complicated. In any case, the performance characteristics of languages that use term rewrite systems like mCRL2, are less predictable than those of imperative languages.

4.3.4 Constructed Data Types

mCRL2 has a built in data type `List` that provides a list implementation. Lists are built using the empty list constructor ('`[]`') and the 'cons' constructor ('`|>`'). For example the expression `3 |> 2 |> 1 |> 0 |> []` defines a list of type `List(Nat)` containing the numbers 3 to 0. A short hand notation for the same list is `[3, 2, 1, 0]`.

A number of standard mappings on lists are predefined (the types of the mappings are given in parentheses):

- `a in l`: True iff a occurs in l. (`S # List(S) -> Bool`)
- `l.n`: Gives the n-th item in the list l. (`List(S) # Nat -> S`)
- `head(l)`: Gives the first item in the list l. (`List(S) -> S`)
- `tail(l)`: Gives list l without its first item. (`List(S) -> List(S)`)

Listing 4.8 shows a model with a process (`P`) that takes a `List(Nat)` as argument. The process `P` verifies that the size of the list is greater than zero (line 3). If the list is non-empty, the process does an action `write` with as argument the first item in the list, after which it calls itself with the remainder of the list (line 4). If the list is empty, process `R` is called (line 5).

The resulting behaviour of the model is such after a `read` action, five `write(Nat)` actions follow with the numbers 1 to 5 as arguments. After the `write(5)` action the model returns to the initial state again.

```
1 act write: Nat;
2     read;
3 proc P(numbers: List(Nat)) = (#numbers > 0)
4                               -> write(head(numbers)) . P(tail(numbers))
5                               <> R;
6      R = read . P([1,2,3,4,5]);
7 init R;
```

Listing 4.8: Example model with a list

mCRL2 also allows the creation of custom data structures, using the following construction: `sort <type> = struct <constructor>{<fields>}`. The name of the sort will be <type>, and <constructor> is used to create an object of that sort. The <fields> can be any number of fields of any type that are identified by a unique name. Individual fields of a struct can be accessed by using the field

name as a mapping on the structure. For example to access the field 'field1' of a struct called 's', the following code is used: `field1(s)`.

Listing 4.9 shows a model with a custom sort struct of type `Sample` (lines 1-4). An object of sort `Sample` is created by process `R` using the constructor `sample` (line 11). The created object is passed as argument to the process `P`. In process `P` execution branches depending on the value of the field `do_print` (line 8). If that field is true, then a `write(Nat)` action is done with as argument the length of the list in the field `numbers` (line 9). In either case is process `R` called again by process `P`.

The behaviour of the model is as follows: It start by either performing a `read(true)` action or a `read(false)` action. After a `read(false)` action the model is in the initial state again. A `read(true)` action is followed by a `write(5)` action (since the list has a size of 5), after which the model is also in the initial state.

```
1  sort Sample = struct sample(
2                           do_print: Bool,
3                           numbers: List(Nat)
4                       );
5  act write: Nat;
6      read: Bool;
7
8  proc P(s: Sample) = do_print(s)
9                           -> write(#numbers(s)) . R
10                          <> R;
11      R = sum bParam:Bool .read(bParam) .P( sample(bParam, [1,2,3,4,5]) );
12 init R;
```

Listing 4.9: Example model with a custom sort struct

The last example concludes this introduction to mCRL2. This introduction is very much incomplete, and focusses exclusively on the features used in the specification model, which is described in the next two chapters. The mCRL2 user manual [48] is a much more complete reference work for the mCRL2 language.

Chapter 5

Behaviour of the PLC-Interlocking Model

The specification model is one of the main products of the conducted research and an important part of the test setup. Because of its complexity, the description of the model is spread over two chapters. This chapter describes the externally observable behaviour of the model, while chapter 6 describes how that behaviour is implemented.

The specification model that has been developed models (part of) the behaviour of the Santpoort Noord PLC-Interlocking installation. Although the developed model is tailored to the Santpoort Noord installation, the design is such that it can also be adapted to model other PLC-Interlocking installations. Unless the text explicitly mentions something is Santpoort Noord specific, anything discussed in this chapter or the next applies to the general design of the model and not merely to the Santpoort Noord model.

This chapter is organised as follows: Section 5.1 covers the specifications that have been used in the construction of the model. The model only covers a part of the functionality of the PLC-Interlocking, section 5.2 clarifies this. The actual description of the model's observable behaviour starts with section 5.3, which describes the transitions that the model uses to communicate inputs and outputs with its environment. This is followed by section 5.4 which explains the functionality implemented by the model.

5.1 Specifications

In order to get a model that correctly expresses the desired behaviour of the SUT, a number of sources have been consulted. These sources can be divided in two groups:

1. Generic sources: Sources specifying the behaviour of PLC-Interlocking systems in general.

2. Location specific sources: Sources specifying the local situation at Santpoort Noord and the (implied) requirements on the Santpoort Noord PLC-Interlocking installation.

5.1.1 Generic Sources

There are three main sources for the generic behaviour of the PLC-Interlocking:

1. The Subsystem/System Specification (SSS) [4].
2. The OVS Application Engineering [5].
3. Expert consultations from members of the PLC-Interlocking development team [6, 7].

The SSS was the main specification used for the construction of the model. This document contains a non-formal textual specification of the intended behaviour of the PLC-Interlocking system. It (should have) served as input for the engineers constructing the PLC-Interlocking and accompanying OVS documents. In reality however, a lot of this document was written in tandem with the PLC-Interlocking code and documentation. The SSS contains high-level requirements and is not detailed enough to serve as a complete specification.

The OVS Application Engineering was used to get the details (such as the names of input and output variables) that were not specified in the SSS.

Both the SSS and OVS document assume certain domain knowledge which goes deeper than the basic understanding of railway safety as expressed in chapter 2. Because of that assumption, the documents on their own did not always form a sufficient specification. In such cases the signalling experts and system engineers from the PLC-Interlocking development team were consulted.

5.1.2 Location Specific Sources

The OVS document specifies how a PLC-Interlocking must be programmed for a given railway yard. The main inputs for the programming process are the properties of the railway yard for which the PLC-Interlocking is being programmed. Those properties are also needed for the installation specific part of a PLC-Interlocking model. The needed railway yard specific information was distilled from the following documents:

1. Track layout map: OBE-blad [8].
2. Signalling layout map: OS-blad [9].
3. Final Design: DO document [10].

The OBE-blad (OBE-blad: Overzicht Baan en Emplacement blad (Dutch)) specifies the track layout and physical relations between the different wayside elements (e.g. which sections lie between which signals). Furthermore this document specifies the names of all wayside elements (from which the names of the input and output variables can be deduced).

The OS-blad provides information regarding the aspects that the different signals can show and the allowed relations between aspects of different signals.

The interlocking has to allow the beams of the level crossings to close before some signals can change their aspects. The amount of time to wait is specified in the DO document (DO: Definitief Ontwerp (Dutch)).

5.1.3 Imperfect Specifications

Ideally, the system would be treated as a black box, and the model would be merely based on a specification. Unfortunately, there is not one definitive specification of what constitutes correct behaviour of a PLC-Interlocking. The main specification, the SSS, is not written totally independent from the developed product. The main problem though, is that it is informal and not very detailed. As a result a lot of the specification details have been extracted from the OVS Application Engineering.

The OVS Application Engineering is a manual (or one could say a specification) on how to construct the logic for a PLC-Interlocking. However, at the same time that OVS is also a result of the PLC-Interlocking development project and thus a part of the PLC-Interlocking product. As such it should be subject to tests, and not part of the test specification. The expert consultations are also troublesome: some of the specific knowledge of these people comes from the fact that they worked on the interlocking system.

The programming of a PLC-Interlocking installation is done based on the OVS Application Engineering. As a result, if there are any faults in the produced interlocking system, then these faults fall in one of two categories:

1. Programming faults that are introduced by incorrectly applying the (correct) instructions of the OVS Application Engineering to the situation at a specific railway yard.

2. Programming faults that result from faults in the OVS Application Engineering document.

There are no particular reasons to assume that faults from the first category would be in the model also, even considering the fact that the OVS Application Engineering was used as one of the specifications for the model.

However, faults from the second category might be duplicated in the model, after all the model is also partly based on the OVS Application Engineering.

The problem of faults from the OVS being duplicated in the model is partly (but only partly) mitigated by the fact that both the SSS and the OVS Application Engineering documents have been extensively reviewed by industry experts. Furthermore the model is written in mCRL2, which is based on a completely different programming paradigm as the FBD language. This in itself guarantees that the model is not an exact copy of the OVS, but rather an interpretation of the specification that was aided by the information in the OVS.

But still there is no true guarantee that the model does not contain the same faults as the OVS Application Engineering.

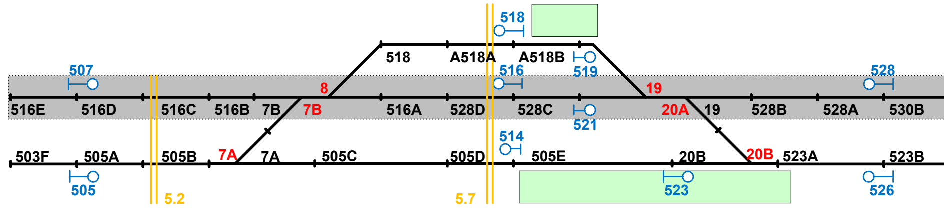


Figure 5.1: The modelled part of the Santpoort Noord railway yard

5.2 Scope of the Model

The model that has been developed during the research period is an initial research model that serves as a proof of concept, and does not capture all functionality of a PLC-Interlocking.

As mentioned above, the SSS [4] was the main specification used. The SSS contains a total of 103 identified functional requirements, with some of those requirements containing additional sub-clauses. A number of these requirements do not apply to the Santpoort Noord PLC-Interlocking because they concern functionality that is not needed at Santpoort Noord (e.g. there are no movable bridges at Santpoort Noord). However, even of the requirements that do apply, only a fraction is implemented in the model.

The reason that not all requirements are implemented is because of time constraints. From the start it was clear that creating a model that implements all requirements is not a trivial matter and would require a lot of effort and knowledge about railway signalling. It was obvious that creating a complete model would take much more time than was available. Thus, the decision was taken to focus on a subset of the requirements.

One of the main limitations in the model is that not all wayside elements are supported. Only inputs from and outputs to signals and track sections are implemented in the model. Most notably, points are not supported by the current model, but some other wayside elements are also not modelled (for Santpoort Noord these are the level crossings and the ATB systems).

Furthermore, not all behaviour of the PLC-Interlocking is modelled. For example, the yellow blinking aspect is not supported, but also the system's behaviour with respect to (possibly inconsistent) redundant inputs is not modelled. Also crucially, the model is less detailed and contains less safety checks than the SUT should implement.

There are a lot of small omissions and abstractions that have been made in the model. The next sections in this chapter detail exactly what functionality is implemented and which inputs and outputs are supported; any functionality not mentioned in this chapter is not implemented.

The model for the Santpoort Noord railway yard is limited to a single straight stretch of rail. Figure 5.1 shows the modelled area marked grey. Even with the above mentioned general limitations on the current version of the model, a bigger area of the Santpoort Noord area could have been modelled (e.g. both straight rail stretches). However, for the sake of simplicity and because of time constraints this has not been done for the research model.

5.3 Input and Output Transitions

5.3.1 Variables

The model has the same basic phases as the SUT: input, computation and output. All relevant input and output of the SUT runs through Boolean variables that are set during the input and output phases. The model must handle those same variables in its input and output phases.

Table 5.1: Input variables of the model

For Each	Variable Name	Description
Signal	`<signal-id>GZ-CI`	Request from the EBP-interface to set the signal to yellow or better (true), or to red (false).
Track section	`<section-id>-TR-DII`	Output from a train detection device; indicates whether a section is free (true) or occupied (false).
Railway yard border	`<signal-id>-XSS1R-DII`, `<signal-id>-XSS2R-DII`, `<signal-id>-XSS3R-DII`	*Speed step variables*: Indicate at which speed (if at all) a train may enter the area of a neighbouring interlocking system. (Signal-id refers to the nearest signal).

Table 5.1 shows the input variables that the model uses.

The 'yellow or better' inputs from the EBP-interface are used to request the interlocking to set a signal to a yellow aspect or an aspect better than yellow. All aspects except for red, and yellow blinking (which is not included in the model) are considered yellow or better. If this input is not set, then the signal will show a red aspect. Whether yellow or better is actually yellow, green, green with a number or one of the other possibilities is decided by the interlocking system.

The 'speed step variables' are used by neighbouring interlocking systems to communicate whether a train can safely enter their area. If these variables are all false, then a train cannot cross the border to the area of the neighbouring interlocking system; if the first Boolean is true it can cross the border at a slow speed; if the first two are true it can cross the border at a higher speed; if all three are true it can cross the border at the highest speed. The exact speeds are railway yard dependent and are specified on the OS-blad.

Table 5.1 is not completely accurate in all cases: There are exceptions to the naming scheme depending on how inputs are wired electrically, and the number of speed step variables can vary as a result of differences in the allowed speeds.

All SUT output variables that the model uses are listed in table 5.2.

The Booleans that determine the signal aspects each control a part of the signal aspect (e.g. `<signal-id>-DR-DO` controls the non-flashing green light). However there is no specific variable to set a signal to red; signals show a red aspect if all variables are false. Since false is communicated by the absence of a signal, this means that signals will always default to a red aspect if there is no active signal from their interlocking system.

Table 5.2: Output variables of the model

For Each	Variable Name	Description
Signal	`<signal-id>H-CO`	Feedback to the EBP-interface regarding whether the signal is set to yellow or better (true), or to red (false).
Signal	`<signal-id>-GR-DO`, `<signal-id>-HR-DO`, `<signal-id>-DFR-DO`, `<signal-id>-DR-DO`, `<signal-id>-4R-DO`, `<signal-id>-6R-DO`, `<signal-id>-8R-DO`	A collection of Boolean outputs that together determine the aspect that a signal shows.

Table 5.2 is also not entirely accurate in all cases. Not all signals can show all aspects, so for some signals there are less outputs. Furthermore there are signals that are controlled through additional output variables (although not at Santpoort Noord).

Figure 5.2 (a variant on figure 3.3) shows the communication of the model with systems external to the logic. The 'Variables' columns show the names of the variables that are communicated between the different systems. The figure also shows the transitions that are used to communicate those variables, more on that in the next section.

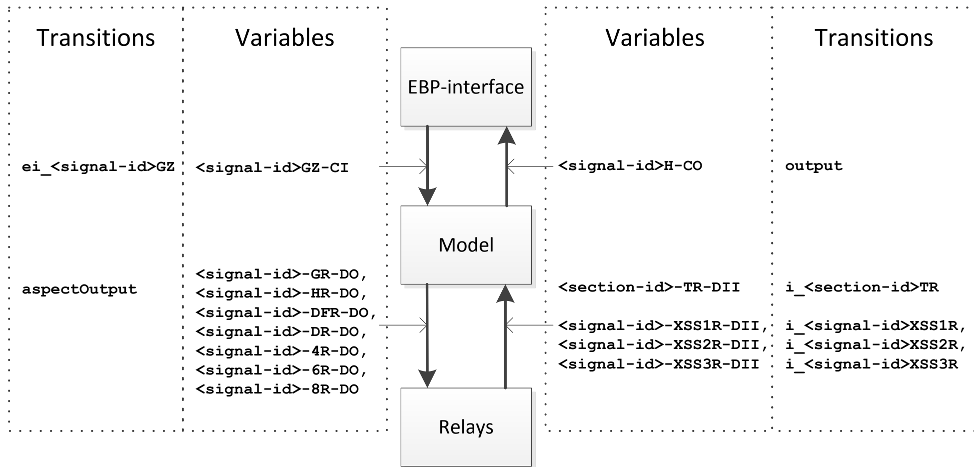


Figure 5.2: Inputs and outputs of the model

5.3.2 Transitions

Input and output of the model is done through transitions.

Each input transition has one Boolean parameter and is used to represent setting a single Boolean variable. During the input phase all input transitions

are always enabled. Variables that are not set during the input phase keep the same value as they had during the previous cycle. Before the first cycle all input variables are initialised to false.

The names of the input transitions are based on the names of the variables that they represent. However the variable names cannot be used verbatim as names for the transitions, because mCRL2 does not allow transitions to start with a number. Therefore the transitions names follow one of two patterns: `ei_<variable name>` for EBP inputs or `i_<variable name>` for the other inputs. Suffixes (e.g. `-CI`) and non-alphanumeric characters are left out of the transition names. An example: setting the variable `7B-TR-DII` to true is represented by the transition `i_7BTR!true`.

The output transitions have fixed names that do not depend on the names of the output variables. The transitions that output the variables that indicate to the EBP-interface whether a signal is set to yellow or better are simply called `output` and have a single Boolean parameter. They are always all outputted, and always in the same order such that is clear which transition represents the variable for which signal.

The seven variables that determine which aspect is shown can only be combined in certain ways. As a result there are in total only ten possible aspects. The outputted aspect is represented by a transition labelled `aspectOutput`, with a single integer parameter. The integer (1 to 10) represents the outputted aspect. Again these transitions are always all outputted, and in the same order such that is clear which transition represents the aspect of which signal. The `aspectOutput` transitions directly follow the `output` transitions, and conclude the output phase.

A simplified diagram of the complete transition flow is shown in figure 5.3. Note that the diagram shows the situation for a model with two signals, whereas the developed Santpoort Noord model has four signals. The computation phase of the model is started by the `cycle` transition, that transition also serves to indicate the end of the input phase. During the computation phase there is no input/output and thus no visible transitions; the output phase starts after internal (τ) transitions that follow the `cycle` transition. The output phase does not have a special end transition, because the fixed number of transitions make it clear when that phase ends.

Figure 5.3 only shows the input transitions, and the different input and output phases of the model. The model has more internal states than shown in the figure; the values of the transition parameters determine the state of the model.

5.3.3 Timer Transitions

As said, figure 5.3 is a simplified representation of the input/output behaviour of the model; it actually omits certain timing related transitions. These timing related transitions do not represent real inputs and outputs of the SUT, but rather are an aid to the model itself.

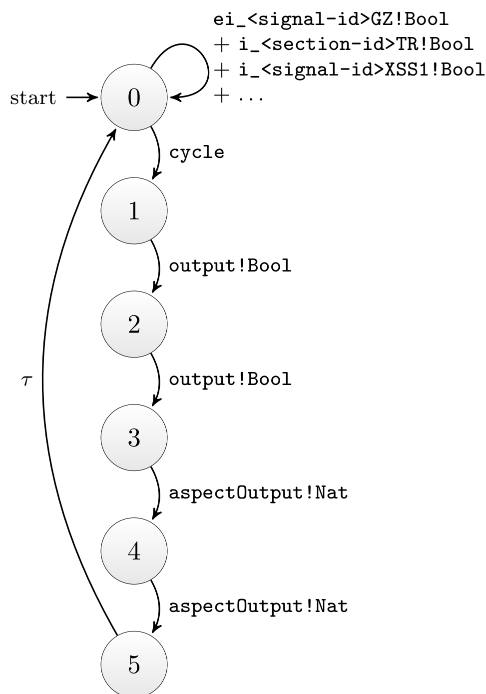


Figure 5.3: Simplified view of the model's input and output transitions for a railway yard with two signals

The model checks a lot of preconditions before executing a request from the EBP-interface. Mostly these checks merely require the available input Booleans as input. However, one check involves time also: Per signal a condition must hold for a certain amount of time before the signal can be set to yellow or better (for more details on this see section 5.4).

This presents a problem, because the model has no knowledge of either the absolute time or the amount of time that passes between cycles. Even if the model was time aware, that would not be useful because the used SUT relies on a virtual time (see chapter 7) rather than the real time.

The chosen solution to the timing problem relies on an adapter at the SUT side of the test setup to keep track of all timers. The model uses the `timerInput!Nat!Bool` transition to communicate to the adapter whether the precondition for a timer holds, and the duration for which that precondition should hold.

After all `timerInput` transitions are done, the SUT adapter returns Boolean values using `timerOutput!Bool` transitions. These transitions are in the same order as the transitions from the model, i.e. the n-th `timerOutput` transition is a reply to the n-th `timerInput` transition. The Boolean parameter of a `timerOutput` transition is true if and only if the precondition has been true uninterrupted for at least the specified duration (equivalent to the way that the output of an FBD TON function is computed, see section 3.3.2).

Figure 5.4 shows the interaction between the model and the SUT adapter.

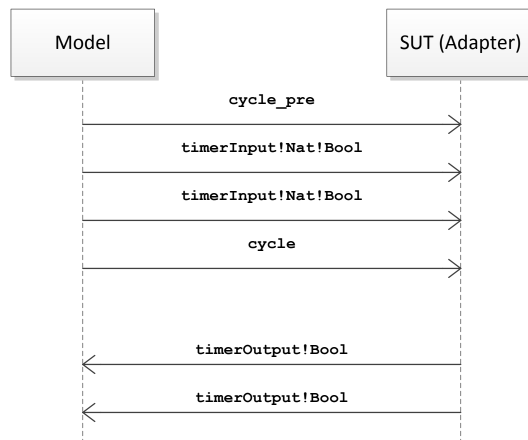


Figure 5.4: Timing interaction between model and SUT adapter

These transitions take place in the computation phase of the model. That is, after the input phase but before the output phase. This 'timing phase' starts with the `cycle_pre` transition, which ends the input phase (instead of the `cycle` transition shown earlier). The `cycle` transition then indicates that all `timerInput` transitions have been outputted and the SUT adapter should start replying with the same number of `timerOutput` transitions, after which the normal output transitions follow.

There are also alternative solutions to this problem, but they were judged to be less suitable.

One alternative solution would be to exploit the fact that all scan cycles in the test setup last equally long, which makes that there is a fixed relation between the amount of cycles done and the amount of time passed. This makes it possible to count the number of cycles and deduce the passed time from that.

Another possible solution is to keep a virtual time in the model, and communicate time increases to the SUT adapter.

Both alternative solutions put the actual timer logic in the model. Given that the constructed model only has one kind of time dependent check, it was judged that the chosen solution was easiest and complicated the model the least.

5.4 Implemented Functionality

5.4.1 Signals: Yellow or Better

The main implemented functionality of the model is the ability to set signals to yellow or better when requested to do so by the EBP-interface. Each cycle the interlocking logic gives feedback information to the EBP-interface indicating per signal whether that signal is actually set to yellow or better.

The constructed model implements three checks that must be passed before a signal is set to yellow or better. Additionally there is a fourth condition that

only applies to signals that lie before a level crossing (for the Santpoort Noord model this is the case for signals 507 and 516).

1. There must be a request from the EBP-interface.

2. There must be no conflicting requests from the EBP-interface.

3. All sections of the route behind the signal must be free and must remain free.

4. The first three conditions must hold during the signal delay time; and that delay must have passed.

The first condition is very straightforward, the model checks whether the variable `<signal-id>GZ-CI` is set.

The second condition is also a simple check. Each signal is the start of a route, which runs from that signal to the next. Two routes conflict when they cannot both be used at the same time. Usually this is because they have track sections in common, but different tracks sections might also be too close to one another for simultaneous use. Since the model does not have points, each signal only has a single route associated with it and all possible conflicts stem from the fact that two signals give access to overlapping routes from opposite sides. So the model can simply verify for each route that none of the (known) conflicting signals is requested.

The third condition is more complicated, because all sections must not just be reported to be empty, they must also be 'logically' empty.

As explained in section 2.2.3, a train detection device can sometimes incorrectly determine a section to be free when there is in fact a train there. Therefore the PLC-Interlocking and the model have some additional checks. A track section that is occupied, will be considered to be occupied until that section is reported to be unoccupied and a train is detected in the next section of the route. If the section is merely reported to be unoccupied by the train detection system, but the train is not detected in the next section, the PLC-Interlocking will consider the section to still be occupied logically.

Furthermore, the PLC-Interlocking and model keep track of which sections are part of an active route and have yet to be travelled by the train on that route. Once a train starts a route (by passing a yellow or better signal), all sections in the route will be considered to be logically occupied, until the train has passed those sections.

The fourth condition verifies that the beams of a level crossing have had enough time to close. Only after the first three conditions have held for a set time (which is given for each signal), can a signal be set to yellow or better. The timer for this check is actually implemented in the SUT, as previously described in section 5.3.3.

Imperfections in the Model

The model does not completely model the PLC-Interlocking. As a result the model will more often declare a signal to be yellow or better than the real PLC-Interlocking. For example, the PLC-Interlocking contains a check for every route that verifies that no train has passed a red signal recently that might affect that route, the model does not contain that check.

If the SUT and the model report different outputs, then JTorX will conclude that a failure has been observed. Incompleteness in the model should not lead to such differences being observed. Therefore the model allows non-determinism in the yellow or better EBP output, if the model determines a signal to be yellow or better.

Concretely if the model determines that the output for a certain signal should not be yellow or better, then only one output transition is enabled: `output!false`. If however the model determines the output to be yellow or better, then both the `output!true` as well as the `output!false` transitions are enabled. This allows the SUT to be more strict than the model, but not the other way around.

5.4.2 Signals: Determining Aspects

Above, only the outputs to the EBP-interface that indicate whether a signal is yellow or better, or red were considered. However, the model also determines the exact aspects that are outputted to the actual signals.

The logic that computes the aspects for all signals is completely deterministic given the following inputs:

- The yellow or better Booleans for all signals in the railway yard.
- The speed step variables for the railway yard borders.

The speed step variables are known to the model since they are read as inputs.

The yellow or better Booleans are determined by the model, but these are not guaranteed to be correct (since the model is imperfect). Therefore the `output` transitions, which indicate yellow or better, are outputted (non-deterministically) before the aspects are determined. The model tracks which `output` transitions were taken by the SUT, and as such knows the actual values for all yellow or better variables and not just the values that the model determined.

If a signal is set to yellow or better, then the OS-blad document states its aspects given the aspect of the next signal, or given the enabled speed step variables (in case its the last signal on the yard). As such the model simply works backwards, starting at the railway yard borders and determining the signals in reverse order according to the OS-blad data.

Chapter 6

Implementation of the PLC-Interlocking Model

This chapter gives an overview of how the functionality that was discussed in chapter 5 is implemented in an mCRL2 model.

The structure of this chapter is as follows: Section 6.1 covers the overall control flow of the model, how inputs and outputs are handled and at what points which outputs are determined. Section 6.2 discusses the data structures in which the model keeps its state. Section 6.3 looks at all the mappings that are defined, and how they are used for performing computations on the data structures, but also for storing configuration data. After that section 6.4 looks at the file structure of the model, and the usage of a pre-processor in the compilation process. Finally section 6.5 discusses briefly how the model can be adapted to different PLC-Interlocking installations.

6.1 Model Execution Flow

The model consists of two dozen processes that each handles a specific task. The model's execution is straight forward: All processes are executed consecutively and in a fixed order; there is no parallel composition. The first process represents the start of a PLC scan cycle which concludes with the execution of the last process, after which the first process is executed again.

Each process calls the next process with two arguments: the values of the current inputs and a struct holding the system's state. The current inputs are passed as a list of `Bool` variables, where each position in the list corresponds to a certain input variable. The system's state is modelled by a custom struct type `State`, which is described in detail in section 6.2.

The model's processes can be divided in five global task categories (which are executed in the following order):

1. Input
2. Computation: yellow or better
3. Output: EBP output yellow or better
4. Computation: signal aspects
5. Output: aspects

Generally input and output is handled by processes, whereas a lot of the logic is encoded in data-mappings that are called by processes. Furthermore processes are used to define the execution flow within the model, because they can pass control to a next process without returning, whereas mappings always have to return a value to their caller. In the following subsections the most important processes of the model are discussed using snippets of code from the model. The mappings that these processes rely on cannot all be covered in this report, however section 6.3 gives a short overview of them.

Note: the code snippets sometimes contain τ transitions (in mCRL2: `tau`), which are internal transitions that are not visible to the SUT. They are needed for performance reasons: Without these transitions the linearisation process will not terminate (or at least not within an acceptable time frame), and run time performance would also suffer.

6.1.1 Input

The `input_main` process is responsible for handling all regular input transitions, a part of it can be seen in listing 6.1. After taking an input transition, it calls itself with the inputs list updated (using the function `setBoolListItem`). The process contains code for all possible input transitions, the majority of this code has been left out of the code listing because it is repetitive and the pattern is clear. The `cycle_pre` transition stops the input phase by calling the `IXL_start` process, which is the starting point of the processes that determine which signals should be set to yellow or better.

```
1 proc input_main(inputs:InputMap, state: State) =
2   ei_507GZ(true) .input_main(setBoolListItem(inputs, ei507GZ, true ), state)
3 + ei_507GZ(false).input_main(setBoolListItem(inputs, ei507GZ, false), state)
4 + ei_516GZ(true) .input_main(setBoolListItem(inputs, ei516GZ, true ), state)
5 + ei_516GZ(false).input_main(setBoolListItem(inputs, ei516GZ, false), state)
6   ...
7 + cycle_pre      . IXL_start(inputs                                  , state);
```

Listing 6.1: Handling input transitions

6.1.2 Computation: Yellow or Better

A major part of the model is devoted to determining for each signal whether it should be set to yellow or better, or to red. In total 15 processes are involved in this, which are executed in order and can be divided in three sub-phases. In these phases the model does the following:

1. Determining for each signal whether the first three conditions hold.
2. Determining for each signal whether the fourth condition holds.
3. General processing of received data.

Determining for each signal whether the first three conditions hold

In the first phase the model determines whether the first three preconditions, as mentioned in the previous chapter (section 5.4), are met. This is done through a number of processes that update the data in the `State` structure that each process passes on. These updates are based on the inputs and the current state, and are handled by data mappings.

Determining for each signal whether the fourth condition holds

If the model has established that the first three conditions have been met for a certain signal, it must still verify that the fourth condition also holds: the first three conditions must have been met sufficiently long. The SUT adapter is actually responsible for verifying this, because as discussed previously in section 5.3.3, this seemed the solution that was easiest to implement. The implementation of this can be seen in code listing 6.2.

First the model communicates to the SUT for each signal whether the first three conditions currently hold using the process `timer_input` (lines 4-7).

Then it receives per signal a transition back indicating whether the fourth condition holds. If the outgoing transition indicated that the first three conditions held for a signal, then the model can either read back (line 12) a `timerOutput!true` transition indicating that the fourth condition also holds, or an `timerOutput!false` transitions indicating that the fourth condition does not hold.

After receiving each of these transitions the process invokes itself again (lines 13 - 22), with the `State` object properly updated to reflect the value reported by the SUT through the `timerOutput` transition.

Note that some lines of code were skipped at line 23; this was done so as not to clutter the code snippet with less interesting code. If no more transitions are expected the code calls on the next process `IXL_start_again`, with which this sub-phase of the model concludes.

```
1 proc timer_input_start(inputs:InputMap, state: State) =
2     timer_input(timers(state), inputs, state);
3
4 proc timer_input(timers:List(Timer), inputs:InputMap, state: State) =
5     (#timers > 0)
6         -> timerInput(timeOut(head(timers)), input(head(timers))) .
    timer_input(tail(timers), inputs, state)
7         <>  cycle . timer_get_outputs(0, inputs, state);
8
9 proc timer_get_outputs(timerCounter: Nat, inputs: InputMap, state: State) =
10    (timerCounter < #timers(state))
11        -> input(timers(state).timerCounter)
12            -> ( sum b:Bool . timerOutput(b) .
```

```
13                    timer_get_outputs(
14                        timerCounter+1,
15                        inputs,
16                        stateSetTimersItem(
17                            state,
18                            timerCounter,
19                            timerSetOutput(timers(state).timerCounter, b)
20                        )
21                    )
22                )
23            ...
24        <> IXL_start_again(inputs, state);
```

Listing 6.2: Processes handling timer transitions

General processing of received data

In the final sub-phase of this task, the model uses the information received through the timer transitions to update the `State` structure. Specifically the substructures that hold information on signals (a list of `Signal` objects) and on the active routes (a list of `ActiveRoute` objects) are updated. This is done using some mappings that use information already present in the `State` structure (such as the values received through the timer transitions).

After this the structure contains all information to continue to the next main phase of the model: outputting whether a signal is set to yellow or better, or to red.

6.1.3 Output: EBP Output Yellow or Better

The `output_EBP_signals` process handles outputting the computed values of the yellow or better bits for each signal; its implementation is shown in listing 6.3.

The process takes, besides the usual two arguments, a third argument: a list of `Signal` objects. This list is taken from the state object by the calling process and each object represents one of the signals in the railway yard. The process outputs the value of the first `Signal` object in the list and then calls itself again with the remainder of the list, or if the list is empty proceeds with the next process: `IXL_aspect_start`.

As explained in section 5.4.1, if the model has determined a signal to be red (i.e. not yellow or better) it will output a `false` transition (line 18). However, if the model determines the signal to be yellow or better, it can output either a `true` or a `false` transition (line 4). The model then stores which of those two is taken in the state object (lines 7-14).

```
1 proc output_EBP_signals(inputs: InputMap, state: State, signals: List(Signal)) =
2 tau . (#signals > 0)
3     -> yellowOrBetter(head(signals))
4         ->  ( sum b: Bool . output(b) .
5                 output_EBP_signals(
6                     inputs,
7                     stateSetSignals(
```

```
8                       state,
9                       updateSignalListRealYellowOrBetter(
10                          signals(state),
11                          signalId(head(signals)),
12                          b
13                      )
14                  ),
15                  tail(signals)
16              )
17          )
18          <> output(false) . output_EBP_signals(inputs, state, tail(signals))
19      <> tau . IXL_aspects_start(inputs, state);
```

Listing 6.3: Outputting the yellow or better EBP outputs

6.1.4 Computation: Signal Aspects

Two processes play an important role in determining the aspects of signals: `IXL_aspects_translate_speed_steps` and `IXL_aspects_determine_aspects`. Both are shown in listing 6.4.

As explained in section 5.4.2, to determine the aspects of signals, the yellow or better bits of all signals must be known as well as the speed step variables. The speed step variables influence the aspect of the signal before them in the same way as a signal influences the aspect of another signal before it. As such the speed step variables are treated in the code as sort of virtual signals. The virtual aspects of these virtual signals are given by the speed step variables. The process `IXL_aspects_translate_speed_steps` translates the values of the speed step variables to a natural number (identifying a virtual aspect).

After the aspects of these virtual signals are known, the aspects of the signals before them can be determined, followed by the signals before those. This is done by the `IXL_aspects_determine_aspects` process which calls the `determineAspects` mapping to that end.

```
1  proc IXL_aspects_translate_speed_steps(inputs: InputMap, state: State) =
2      tau . IXL_aspects_determine_aspects(
3          inputs
4          , stateSetSignals(
5                state,
6                setSignalAspect(
7                setSignalAspect(
8                  signals(state)
9                  , SIG_SA, speedStepBoolsToAspects(true, inputs.i507XSS1,
       inputs.i507XSS2, inputs.i507XSS3))
10                 , SIG_NA, speedStepBoolsToAspects(true, inputs.i528XSS1,
       inputs.i528XSS2, false))
11           )
12     );
13
14 proc IXL_aspects_determine_aspects(inputs: InputMap, state: State) =
15     output_aspects(
16         inputs
17         , stateSetSignals(
18             state,
```

```
19            determineAspects(signalAspectDeterminationOrder, signals(state))
20        )
21    );
```

Listing 6.4: Determining the signal aspects

6.1.5 Ouput: aspects

Outputting the computed signal aspects is very straight forward, as shown in listing 6.5.

The process `output_aspects_signals` takes a list of `Signal` objects as third argument, just like the function `output_EBP_signals` described in section 6.1.3. The process simply outputs the aspect of the first element in the list, and then calls itself with the remainder of the list.

If the whole list has been processed, then the scan cycle is complete and the `input_start` process is called. This process brings the model in the input phase again and will in turn call the `input_main` process which will process the new input transitions as described in section 6.1.1.

```
1 proc output_aspects_signals(inputs: InputMap, state: State, signals:
    List(Signal)) =
2    (#signals > 0)
3        -> aspectOutput(aspect(head(signals))) . output_aspects_signals(inputs,
    state, tail(signals))
4        <> input_start(inputs, state);
```

Listing 6.5: Outputting the signal aspects

6.2 The State Data Structure

As already explained above, each process takes a variable of type `State` as argument. `State` is a custom struct that holds all information needed (with the exception of the input variables, which are held in a separate list) to determine the outputs of the system.

The `State` struct requires further explanation, because it is the main data structure used in the model and is an integral part of its overall structure. This section provides that explanation.

The `State` struct contains five fields:

- `activeRoutes`: a list of `ActiveRoute` structs.
- `signals`: a list of `Signal` structs.
- `physicalSections`: a list of `Bool` variables.
- `symbolicSections`: a list of `Bool` variables.
- `timers`: a list of `Timer` structs.

For each signal that is part of the modelled area there is a `Signal` object, an `ActiveRoute` object (for the route behind the signal) and a `Timer` object. The `physicalSections` and `symbolicSections` lists each contain a `Bool` variable for every section that is part of the modelled area.

In the following, the functions of the different data fields in a `State` struct are discussed in more detail.

6.2.1 ActiveRoute

The `ActiveRoute` structs are used to keep track of which routes are requested, which routes are used, and on which part of the active routes trains have travelled already. The information about trains on active routes is used to determine the logical occupation of sections. Ultimately, the information kept in this struct is used (among other information) to determine whether the first three conditions, as defined in section 5.4, hold.

An `ActiveRoute` struct consists of the following fields:

- `routeId`: Unique identifier of this route.
- `requested`: A `Bool` indicating whether or not this route has been requested by the EBP.
- `preconditions`: A `Bool` indicating whether or not the first three preconditions have been met for this route.
- `active`: A `Bool`, when true indicating that the signal guarding the route is set to yellow or better, or that a train is riding on the route.
- `started`: A `Bool`, when true indicating that a train is riding on the route.
- `activePart`: List of `Bool` variables with one entry for each section. Each variable indicates whether a train still has to pass that section if the route is active.
- `wasOccupied`: List of `Bool` variables with one entry for each section. Each variable indicates whether a train has been on that section (since the route became active).
- `nextWasOccupied`: List of `Bool` variables with one entry for each section. Each variable indicates for a section whether the next section in the route has been occupied (since the route became active).
- `sections`: An ordered list containing the `sectionId`s of the sections that make up this route.

The fields `routeId` and `sections` are constants, i.e. if two `activeRoute` objects have the same `routeId`, then they will also have the same `sections`. These fields are not really part of the system's state (they are constants after all), but storing this information in the `ActiveRoute` struct simplifies some of the mapping code.

6.2.2 Signal

A `Signal` object stores information about a signal in the railway yard. It has the following fields:

- `signalId`: Unique identifier of this signal.

- `yellowOrBetter`: A `Bool` indicating whether this signal is set to or should be set to yellow or better (on true) or to red (on false) according to the model's computation.
- `realYellowOrBetter`: A `Bool` indicating whether the signal has actually been set to yellow or better, or to red by the SUT.
- `aspect`: Holds an `AspectId` which indicates the exact aspect of this signal.

`Signal` objects are not only used to model real signals, but are also used for 'virtual signals' to represent the speed step variables. To that end a number of extra `AspectId`s have been defined that represent the different combinations of speed step variables.

This is done to simplify the aspect determination code: An aspect always depends on the next aspect on a route or on speed step variables. Representing speed step variables as aspects means that the aspect determination code only has to look at the next signal, regardless of whether there is actually a next signal or whether it should look at the speed step variables.

6.2.3 physicalSections and symbolicSections

Both `physicalSections` and `symbolicSections` are lists of `Bool` variables. In both lists, each position corresponds to a section on the railway yard.

Each cycle, the `physicalSections` list is set to match the section occupations as reported through the input transitions.

The `symbolicSections` list contain logical occupations. A section in the `symbolicSections` list is occupied if at least one of the following holds:

- The section is occupied according to the `physicalSections` list.
- The section is part of an active route in which a train still has to pass over the section.

6.2.4 Timer

The `Timer` objects keep some state information that is input to the processes that are tasked with handling the timer transitions. The return value from the SUT adapter is also written in the `Timer` object by these processes, and used by later processes.

A `Timer` objects has three fields:

- `input`: A `Bool` variable that is the input to the SUT adapter (so actually output of the model).
- `timeOut`: A `Nat` (Natural number; integer) variable that holds the timer timeout. This is actually a constant (that differs from one `Timer` to the next), and not truly a state variable.
- `output`: A `Bool` variable that holds the reply from the SUT adapter, i.e. this variable indicates whether `input` was true for `timeOut` milliseconds.

6.3 Data Mappings

The majority of the model's code base consists of data mappings. These mappings are used for a number of things: configuration data, updating data structures and the implementation of business logic.

Configuration data is defined as static mappings, i.e. mappings for which the result is defined explicitly for every input. Other mappings are more 'dynamic': The result is expressed in a functional programming style.

The following discusses the different kinds of mapping in more detail and give some examples. However, as said the majority of the code base consists of mapping, as a result only the main themes can be covered here. For different kind of mappings an example is discussed, followed by a discussion of the usage of that kind of mapping in the model.

Note that mCRL2 has a functional programming style for data mappings that is implemented using a rewrite system. As a result variables cannot be assigned, so for example when the text reads 'update a field in a struct' this should be interpreted as 'create a structure with the fields set to be the same as the fields from an existing structure, except for the field that is being updated which is set to a new value'.

6.3.1 Static Mappings

Static mappings are used extensively for configuration purposes, i.e. to customise the generic model for a specific system such as the Santpoort Noord installation. A lot of other mappings rely on static mappings to supply certain information regarding the configuration, such as the railway yard topology. Furthermore, static mappings are used for some data initialisation at the start of a model's execution.

An Example

Section 6.2.2 describes how the model uses virtual signals to codify information from the speed step variables. However, some processes and mappings require a list that only contains the non-virtual `Signal` objects. The mapping `isVirtualSignal` provides a mapping from `SignalId` to `Bool` that can be used to distinguish `Signal` objects that represent real signals from `Signal` objects that represent virtual signals. Listing 6.6 shows the implementation of `isVirtualSignal`.

```
1 map
2 isVirtualSignal: SignalId -> Bool;
3 eqn
4 isVirtualSignal(SIG_507) = false;
5 isVirtualSignal(SIG_516) = false;
6 isVirtualSignal(SIG_521) = false;
7 isVirtualSignal(SIG_528) = false;
8 isVirtualSignal(SIG_SA)  = true;
9 isVirtualSignal(SIG_NA)  = true;
```

Listing 6.6: The isVirtualSignal mapping

More Static Mappings

All information that changes from one railway yard to the next is encoded with static mappings. Below is a list of most other important static mappings:

- `map initialActiveRoutes: List(ActiveRoute);`
  Gives a list of `ActiveRoute` objects with the correct initial values set.
- `map initialSignals: List(Signal);`
  Gives a list of `Signal` objects with the correct initial value set.
- `map initialTimers: List(Timer);`
  Gives a list of `Timer` objects with the correct initial values set.
- `map routeToSectionList: RouteId -> List(SectionId);`
  Gives a list containing all `SectionId`s (in order) for a route.
- `map sectionAfterRoute: RouteId -> SectionId;`
  Gives the `SectionId` of the section directly after the last section of a route.
- `map routeConflictSet: RouteId -> Set(RouteId);`
  Given a `RouteId`, gives a set containing the `RouteId`s of all conflicting routes.
- `map aspectRelation: SignalId # AspectId # SignalId -> AspectId;`
  Gives the `AspectId` for a signal (if it is yellow or better), when given the next signal in the route and its aspect.
- `map routeToStartSignal: RouteId -> SignalId;`
  Gives the `SignalId` of the signal at the start of a certain route.
- `map startSignalToEndSignal: SignalId -> SignalId;`
  Gives the `SignalId` of the signal at the end of a certain route, given the signal at the start of the route.
- `map routeToDelayTimerId: RouteId -> TimerId;`
  Gives the `TimerId` (which is just a `Nat` that indicates the `Timer`'s position in a list) that belongs to the `Timer` associated with a certain route..

Mappings for List Indexes

Another usage of mappings out of convenience is in creating constants which map to `Nat`s that are used as indexes in lists.

For an example of this, see listing 6.7 (Note that lines have been skipped from the snippet, indicated by '...'). The sample code defines two constants (`ei507GZ` and `ei516GZ`), that are then assigned a value (`0` and `1`). All input variables are stored in a list; the constants are the indexes at which the variables `507GZ-CI` and `516GZ-CI` are stored. The earlier shown code listing 6.1 shows how the constants declared in listing 6.7 are used to refer to positions in the `inputs` list in a call to the mapping `setBoolListItem`.

```
1 sort InputId = Nat;
2 map
3 ei507GZ  : InputId;
4 ei516GZ  : InputId;
```

```
5 ...
6 eqn
7 ei507GZ  =  0;
8 ei516GZ  =  1;
9 ...
```

Listing 6.7: InputId mappings

Constants that represent indexes in lists are used for numerous kinds of lists. However, because of the way that mCRL2's term rewrite system works, such constants cannot be used as arguments in equations that define other mappings. As a result not all constants could be defined using mappings; in such cases an alternative solutions was used that relies on a pre-processor, see section 6.4 for more on that.

6.3.2 Convenience Mappings

mCRL2 lacks mappings and/or syntactic sugar for a lot of common tasks such as replacing an entry in a list or updating a field in a structure.

For example updating a field in a struct requires recreating the struct with the original fields and the changed field. Doing this whenever a field is updated complicates the model's code, but the approach also lacks a desirable form of encapsulation: if a field is added to the struct, then all code updating some part of that struct must be changed.

To make the model's code simpler and better maintainable a lot of basic mappings have been added for updating fields in the main data structures. Now a change to one of the data structures only leads to changes to the mappings that work directly on that data structure.

An Example

The previously shown listing 6.4 shows examples of calls to such a mapping on lines 4 and 17 where it calls `stateSetSignals`. This mapping, takes two arguments: a `State` object and one of type `List(Signal)`, and returns a new `State` object with the `signals` field changed. Listing 6.8 shows the implementation of `stateSetSignals`.

```
1  map stateSetSignals: State # List(Signal) -> State;
2  var
3  s: State;
4  update: List(Signal);
5  eqn stateSetSignals(s, update) =
6            state(
7                activeRoutes(s),
8                update,
9                physicalSections(s),
10               symbolicSections(s),
11               timers(s)
12               );
```

Listing 6.8: The stateSetSignals mapping

Implemented Mappings

Mappings such as in the above example have been implemented for all fields of the `State` struct, but also for (some fields of) other structs. The other types of structs for which such mappings have been defined are: `ActiveRoute`, `Signal` and `Timer`.

Furthermore comparable mappings have been defined to update items in lists of various kinds. For example the mapping `map setBoolListItem: List(Bool) # Nat # Bool -> List(Bool);` updates an item in a list of `Bool` variables. Similar mapping functions have been defined for lists that contain `ActiveRoute`, `Signal` or `Timer` variables.

Finally, some mapping functions have been defined that combine the above. These mappings update a single item in one of the lists of a `State` object. The mapping `stateSetSignalsItem` is an example of this: `map stateSetSignalsItem: State # Nat # Signal -> State`. Instead of updating the whole `signals` field like the code in listing 6.8, this mapping merely changes one of the items in the list that is the `signals` field. Besides for the `signals` fields, such mappings have also been implemented for all other fields of the `State` object that are lists.

6.3.3 Computation Mappings

Most of the computations in the model are performed by mapping functions. The processes handle input and output, and determine the execution order, but for most of the actual computations they rely on mappings.

Most of these computation mappings revolve around updating the `State` data structure based on inputs, and on values already in that data structure. Some additional mappings are focused on extracting specific data from either the `State` data structure or from the inputs. Each mapping has a specific task, and does only part of the total needed computation. The model's processes call these mappings in a specific order and with specific arguments, which ensures that the `State` data structure becomes in a state that is consistent with the processed input transitions. from the input transitions. The processes can then extract the data from the `State` data structure that is needed for the output transitions.

An Example

Listing 6.9 shows the mapping `determineAspects` and its supporting mapper function `determineAspectSignal`.

The mapping `determineAspectSignal` actually does the real work: It determines the aspect of a signal. It sets the aspect to red (defined as `A_R`) if the signal does not show a yellow or better aspect (lines 35-39). If, on the other hand, the signal is set to a yellow or better aspect, it determines the aspect based on the aspect of the next signal (lines 23-34). For this it relies on the static `aspectRelation` mapping that defines the relations between signal's aspects (lines 28 - 32).

The `determineAspects` mapping merely makes sure that `determineAspectSignal` determines the aspects in the right order. For

this, the first argument to the `determineAspects` mapping has to hold the `SignalIds` in the right order. Earlier, listing 6.4 showed the code were the `determineAspects` mapping is called, the order of signals is set there by the static mapping `signalAspectDeterminationOrder` (line 19 of listing 6.4).

```
1 map
2 determineAspects: List(SignalId) # List(Signal) -> List(Signal);
3 var
4 signal_id: SignalId;
5 signal_ids: List(SignalId);
6 signals: List(Signal);
7 eqn
8 determineAspects([], signals) = signals;
9 determineAspects(signal_id |> signal_ids, signals) =
10         determineAspects(
11             signal_ids,
12             determineAspectSignal(signal_id, signals)
13         );
14
15 map
16 determineAspectSignal: SignalId # List(Signal) -> List(Signal);
17 var
18 signal_id: SignalId;
19 signals: List(Signal);
20 eqn
21 determineAspectSignal(signal_id, signals) =
22         if(realYellowOrBetter(signals.signal_id)
23           , setSignalListItem(
24                 signals,
25                 signal_id,
26                 signalSetAspect(
27                     (signals.signal_id),
28                     aspectRelation(
29                         startSignalToEndSignal(signal_id),
30                         aspect(signals.startSignalToEndSignal(signal_id)),
31                         signal_id
32                     )
33                 )
34             )
35           , setSignalListItem(
36                 signals,
37                 signal_id,
38                 signalSetAspect((signals.signal_id), A_R)
39             )
40           );
```

Listing 6.9: The determineAspects mapping

Implemented Mappings

In total 14 mappings, such as the above `determineAspects`, are called directly from the model's processes. Often these mappings themselves rely on other computation mappings (such as `determineAspectSignal` in the above exam-

ple), convenience mappings (such as `signalSetAspect`) and/or static mappings (such as `aspectRelation`).

These mappings cannot all be discussed as detailed as `determineAspects` was discussed above. Below is a list of declarations of all computation mappings that are called directly by processes with a short description per mapping. Unfortunately it is not possible to fully explain all details without at least replicating the whole model here.

- `map updateTrainProgressInAllRoutes: List(ActiveRoute) # List(Bool) -> List(ActiveRoute);`
  Updates the `nextWasOccupied` and `activePart` lists for all `ActiveRoutes` in the list that have started.
- `map recallRoutes: List(ActiveRoute) -> List(ActiveRoute);`
  Sets routes' `active` field to false, if they are active, but not started and there is no EBP-request for a 'go' aspect anymore.
- `map startRoutes: List(ActiveRoute) # List(Bool) -> List(ActiveRoute);`
  Sets routes' `started` field to true, if they are active, but not yet started, and the first section of the route is occupied.
- `map stopRoutes: List(ActiveRoute) -> List(ActiveRoute);`
  Sets routes' `active` and `started` fields to false if the train travelling on the route has completed the whole route.
- `map isNotActivePartOfRoute : SectionId # List(ActiveRoute) -> Bool;`
  Returns true if the section is not part of the active part of an active route (i.e. a train still has to pass over it).
- `map updateRoutePreconditions: List(ActiveRoute) # List(Bool) # List(ActiveRoute) -> List(ActiveRoute);`
  Given a list of `Bool`s indicating symbolic section occupations, updates the `preconditions` field of `ActiveRoute` objects indicating whether the first 3 conditions hold.
- `map updateRouteDelayTimerInputs: List(ActiveRoute) # List(Timer) -> List(Timer);`
  Given a list of `ActiveRoute`s, updates the inputs for the `Timer`s belonging to the routes.
- `map routeTimerFinished: RouteId # List(Timer) -> Bool;`
  Returns true if the timer for the route with `routeId` returned the value true.
- `map activateRoutes: List(ActiveRoute) # List(Signal) -> List(ActiveRoute);`
  Sets routes' `active` fields if they were not active, and the signal at the start of the route does not show a stop aspect.
- `map updateSignalListRealYellowOrBetter: List(Signal) # SignalId # Bool -> List(Signal);`
  Sets the `realYellowOrBetter` field of a single `Signal` in a `Signal` list.

- `map setSignalAspect: List(Signal) # SignalId # AspectId -> List(Signal);`
  Sets the `AspectId` of a specific `Signal` in a list of `Signals`.

- `map speedStepBoolsToAspects: Bool # Bool # Bool # Bool -> AspectId;`
  Translates the speed-step Booleans from a neighbouring interlocking system to an `AspectId`.

- `map determineAspects: List(SignalId) # List(Signal) -> List(Signal);`
  Updates the aspects of all signals in the provided list of `Signal` objects.

- `map filterVirtualSignals: List(Signal) -> List(Signal);`
  Returns the given list of signals with the virtual signals filtered out.

6.4 Pre-Processor Usage and File Structure

The created model does not consists of a single file that can simply be linearised, but rather it consists of multiple files that have to be pre-processed to create a single, valid, mCRL2 file that can then be linearised. As will be explained, this approach offers some advantages.

6.4.1 Pre-Processor

A pre-processor is used to overcome two shortcomings in the normal mCRL2 toolchain:

1. mCRL2 has no way of defining constants that can replace literals universally.

2. mCRL2 has no mechanism for modularisation.

The next paragraphs explain these problems in a bit more detail. Furthermore they detail how these problems are solved using a regular C pre-processor.

Section 6.3.1 showed how mappings can be used to define constants in mCRL2. However, the constants defined with these mappings cannot be used universally: They cannot always replace the literals that they represent. For example, listing 6.6 shows the definition of the `isVirtualSignal` mapping, which takes one parameter. In the equations that define that mapping, constants are used for the parameter to help readability and make the code understandable. These constants are however not defined using mappings, but using the pre-processor. The mCRL2 term rewrite system used in the data mappings would otherwise not be able to properly rewrite rules that use `isVirtualSignal` with an argument of type `Nat` (`SignalId` is really just a `Nat`).

The `#define` directive of the C pre-processor can be used to define constants universally. With the pre-processor, magic numbers are kept out of the code, yet the mCRL2 tools only have to deal with the literal values that the constants represent.

In total the model consists of almost 2000 lines of mCRL2 code (including comments and empty lines). The mCRL2 tools expect a single file that contains a complete model without any options for modularisation.

The pre-processor makes it possible to divide the code over multiple files, and include those (using the `#include` directive) from one main file. This makes it possible to group code in units smaller than a whole model, which has been used to group similar code together and to split generic code from installation (Santpoort Noord) specific code.

6.4.2 File Structure

The model has been divided over four files:

- `SPTN-model.mcrl2-pre`: Main file from which other files are included. This file contains all processes (as described in section 6.1) and initialisation code to start the first process.
- `SPTN-model-generic-sorts.mcrl2-pre`: Defines all generic data structures (as described in section 6.2) and the convenience mappings defined to update these (as described in section 6.3.2).
- `SPTN-model-generic-mappings.mcrl2-pre`: Contains all generic mappings that are used by the processes for computations on the data structures; These are the mappings described in section 6.3.3 .
- `SPTN-model-specific-mappings.mcrl2-pre`: File that contains all static mappings (as described in section 6.3.1).

6.5 Adaptability

One of the research goals was to make the model's design such that it can also be used to model other PLC-Interlocking installations.

Most of the code in the model is generic and models general PLC-Interlocking behaviour that is not installation specific. However some code is Santpoort Noord specific, and would have to be changed to be part of a model of another PLC-Interlocking installation. Code that needs to be modified is restricted to two files: `SPTN-model-specific-mappings.mcrl2-pre` and `SPTN-model.mcrl2-pre`.

The static mappings in `SPTN-model-specific-mappings.mcrl2-pre` are purely used for installation specific configuration. Modifying this file should not be hard; the mappings are merely declarations of the track and signalling topology, which can be easily deduced from documents such as the OBE-blad, OS-blad and DO.

Some of the processes in `SPTN-model.mcrl2-pre` that are concerned with input and output will also have to be changed when adapting the model to another installation. This concerns only a subset of the processes defined in the file, but doing so requires a bit more understanding of the model's architecture than the changes to the configuration mappings.

Of course making changes to a model is unavoidable when using it to model another installation; every railway yard is different, as are the inputs and outputs of every PLC-Interlocking installation.

Chapter 7

The System Under Test

Clearly the goal is to test PLC-Interlocking systems. However, using a PLC-Interlocking as the SUT in a model based testing approach presents some problems. Firstly, a PLC-Interlocking consists of expensive hardware; as a result using a PLC-Interlocking for testing purposes is expensive also. Secondly, interfacing a tool like JTorX with a real PLC system is a complicated matter.

Instead of using a real PLC-Interlocking as SUT, an alternative method was used. The program code that would normally run on a PLC-Interlocking has been recompiled into an executable that can run on a regular PC. This executable can communicate with JTorX using the standard input and output streams.

7.1 Compilation Process

As explained in chapter 3, a PLC-Interlocking is programmed using the graphical FBD language. Unfortunately no standard compiler is available that can compile a program in FBD form to a PC executable.

The used development environment (SILworX) cannot export FBD programs in any (documented) form that lends itself to automatic processing or compilation. However, the way that SILworX creates binaries out of the graphical FBD notation creates an opportunity to obtain a C++ representation of those programs.

To the SILworX user, the whole compilation process is an atomic operation started by the push of a button. However, in the background SILworX executes a number of separate tasks:

1. All FBDs are translated to C++ code.
2. The generated C++ code is compiled with a regular C++ compiler.
3. The temporary C++ files are deleted.

The files that are generated in the first step can be copied to another location during the time that the compiler needs to compile them. These files contain

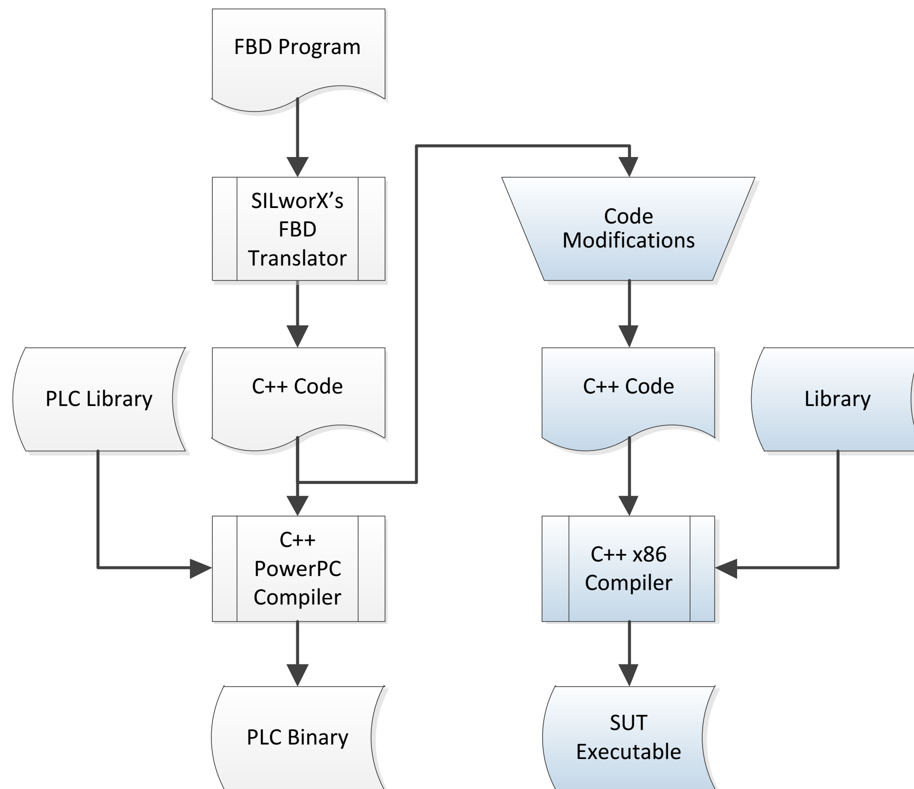


Figure 7.1: Compilation chain for the SUT

all the code that is specific to the PLC program that is being created. A pre-compiled library and its header files, contain the standard FBD function blocks that are defined by the IEC FBD standard [46] (see table 3.1 for the used subset). From now on this library will be referred to as the *'IEC standard library'*

The compiler translates all C++ files and links everything together to create a complete PLC binary. The processors used in HIMax PLCs are based on the Power instruction set[1]. The GNU C++ compiler is used as a cross compiler to generate native code for this instruction set.

The two left most columns of figure 7.1 depict the high-level steps that happen in the background when compiling with SILworX.

To recompile the code of a PLC program for execution on a regular PC, some additional steps have to be taken. The right hand side of Figure 7.1 gives a high level overview of those steps.

The normal IEC standard library cannot be used on the x86 architecture, since it is pre-compiled for the Power architecture. Thus, an alternative library needs to be linked for the PC version of PLC programs.

Furthermore the generated C++ code only contains the program logic specified in the FBDs, and expects some tasks to be done by its environment. Specif-

[1]`http://en.wikipedia.org/wiki/Power_Architecture`

ically, interfacing with the environment, calling the functions in the generated code and allocating the input and output variables are tasks that are not handled by the generated code. Therefore extra code has to be added (mainly generic, i.e. the same for any PLC-Interlocking) plus some specific changes to the generated code need to be done. Currently these code modifications are done manually and take about half an hour, but this process could be automated. After the manual code modifications, the rest of the compile process is fully automated through the use of a Makefile.

The next three sections give more detailed information about the structure of the C++ code, and the steps to get a working program out of the generated C++ code.

7.2 Program Structure

7.2.1 C++ Code Structure

There are two directories with C++ source code for any FBD program created in SILworX. One directory contains generic header files that are used with every FBD program. The other directory contains code that is generated based on the specific FBDs of that program.

Note that there is no official documentation for the C++ code that SILworX creates. As a result, most information in this section about the structure of that code is based on observations.

Generic Code

The generic code is subdivided in 'base' headers and platform specific headers for different 'targets' (HIMA produces multiple PLC types).

The target specific code only consists of two or three header files that contain define statements (partly using non-standard C++) that have to do with the memory and register layout of specific targets. These files are not very interesting, because there are no special memory layout requirements when compiling for a normal PC.

More interesting are the nine header files in the base directory. These header files specify the data types that are needed to interface with the generated code as well as functions that the generated code depends on. There are four header files that are of special interest because their content is used by PLC-Interlocking programs; table 7.1 gives an overview of their contents. The other five header files are not used at all or only sparsely, mostly because they specify function blocks that PLC-Interlocking programs do not use.

Both `iec_std_types.h` and `iec_std_lib.h` specify functions that they don't implement. These functions have to be implemented in the IEC standard library.

Generated Code

The generated code is also subdivided into two directories: the 'common' and the 'source' directory.

Table 7.1: Main header files

Header file	Contents
iec_std_base.h	Contains macros that control which version of a variable is read out. This allows overwriting variable values during regular tests, but this functionality is not needed in the SUT.
iec_std_lib.h	Declares classes that implement the functionality of the standard function blocks. Furthermore declares a struct that holds general system information, such as the time and the number of scan cycles executed.
iec_std_types.h	Declares and (partly) implements wrapper classes for all data types. These classes are used for all variables, function arguments and return values in the generated code and other headers.
iec_types.h	Contains type definitions for the basic types that are wrapped by the classes of iec_std_types.h.

The common directory contains the code for every (generic) function that is defined in the FBD program. For each (generic) function there is a class that has two functions, one for initialisation (`InitVarTemp`) and one to execute the logic of that function (`CallFunctionContent`). The input, output and internal variables of each function are members of the matching class. The implementation of each class is spread over three files: a header and a source file for each function.

The main function of the FBD program (i.e. the function that (indirectly) calls all other functions) is defined in three files in the source directory. This function is encoded in C++ as a class in the same way as the other functions are encoded in classes in the common directory. The `CallFunctionContent` function of this main class executes the whole program logic for a single cycle of a PLC; that is, the `CallFunctionContent` function is the second step in the execution of a PLC as described in section 3.1

The source directory also contains a file, SRCgVProgram.h, that instantiates the main function block and initialises all member variables of all classes that the main function is composed of. Furthermore there is a file, Himamain_KEVars.h, that declares all global variables (including all input and output variables).

All variable names, class names, function names and file names are directly derived from the names of the corresponding objects in the FBD notation of the program. In all names, special characters are replaced by certain sequences of two characters. Table 7.2 shows how this conversion is done.

Table 7.2: Conversion of special characters in the generated code

Special Character	Replacement
(space)	_w
- (dash)	_k
_ (underscore)	_u
((left parenthesis)	_f
) (right parenthesis)	_g

Depending on the type of object, and its function there are certain prefixes or suffixes added. For example, a class modelling a generic function block will get the prefix `USRGF`, and the file declaring this class will get the additional string `SRC` prefixed to its name.

7.2.2 Code Structure for the SUT Executable

There is no official documentation on how to compile the obtained C++ code into a regular x86 binary. The obtained code lacks certain features and functions needed for a complete program. Therefore a method has been devised to create a fully working program from the available C++ sources.

The main part of a PLC program, the program logic, is present in the generated code. Furthermore there is code to initialise all the logic classes, and all global variables are declared (but not defined).

What is still missing to form a complete program is the code for the other parts of a PLC cycle, i.e. the input and output code that allows setting and reading variables. Furthermore the code that defines all global variables is also missing. To remedy this, a directory 'program' with multiple source files is created. The main part of the source code in these files is generic and can be used for any PLC-Interlocking program. Two files have to be adapted for every PLC-Interlocking program; one defines all global variables, the other maps the names of the global variables (strings) to pointers to the actual variables. A specific interface for JTorX also has to be created, this is program and model specific, more on this in section 7.4.2.

Another important piece that is missing is an x86 implementation of the IEC standard library. More on this can be found in the section 7.3.

Table 7.3: Main items in the SUT's source tree

File or directory	Source	Description
`gen/`	Available	The generated source code.
`target_base/`	Available	The header files that the generated code uses.
`own_iec_lib/`	Created	Implementation of the IEC standard library.
`program/`	Created	
`|->main.cpp`	Created	Main file that initialises everything.
`|->plc_interface.cpp`	Created	An API to the generated program logic.
`|->cli_shell.cpp`	Created	A command line interface to plc_interface.
`|->variable_definitions.h`	Custom	Defines all global variables.
`|->mapping.cpp`	Custom	Mapping from variable names to pointers.
`\->jtorx_shell.cpp`	Custom	An interface for a certain JTorX model.

Table 7.3 gives an overview of the most important files and directories in the resulting source tree. The second column indicates whether the file (or directory) was part of the C++ files that SILworX created ('Available'), whether it was a file that was created to work with any PLC-Interlocking instance ('Created') or whether it was a file that was created and adapted for a specific PLC-Interlocking instance ('Custom').

7.3 Reimplementing the IEC Standard Library

Table 3.1 shows the standard function blocks that a PLC-Interlocking program might use. Most of these function blocks are directly translated to C++ code by SILworX (e.g. an 'AND'-block becomes `&&`). The functions that keep an internal state between invocations (the second group in the table) form the exception to this: They are not translated to C++ code, but are merely replaced by function calls to instantiated class objects. The generated code does not implement these classes, but assumes them to be provided by the IEC standard library.

Unfortunately there was no usable x86 version of the IEC standard library available. Therefore a new implementation was created that implemented classes for the limited subset of used functions. In total five function blocks needed to be implemented in the library: TON, TOF, F_TRIG, R_TRIG and SR. It turned out that the implementation of these classes also had to use the binary `+` and `-` operators on one of the data types defined in `iec_std_types.h`. These operators were also implemented.

Of course it is of critical importance that the created SUT executable has the same behaviour as the original PLC-Interlocking. Since the IEC standard library is part of that executable, its behaviour should be 100% compatible with that of the version linked with the original PLC executable.

Ensuring compatibility starts by having a correct and complete specification. The description of the function blocks in the SILworX help files [42] was used as a specification. Although these descriptions are not formal, they are aimed at PLC programmers for whom understanding the semantics of the blocks is critically important. As a result the specifications are clear and can be trusted to be correct.

Additionally, the help files also contain example graphs. These graphs show, using example scenarios, how inputs to function blocks influence the outputs of those blocks over time. Figure 7.2 shows such a graph for a TOF function block ('IN' is input, 'Q' is output and 'PT' is time delay). Note that the 'ET' output in the graph is not used by the PLC-Interlocking code; the library code however does calculate that output.

Systematic unit testing was employed to make sure that the library implemented the specifications correctly. The scenario's in the graphs from the help files were recreated by the unit tests. For example the unit test for the TOF function block, recreates the graph in figure 7.2 with 68 data points on the time axis. That is, 68 times the inputs are given, is the time virtually increased and are the output values checked.

Note that the real use scenarios are a lot simpler than the scenarios of the unit tests. For example, the TOF scenario also tests whether the function block behaves correctly when the time delay is changed; in PLC-Interlocking code this delay is always constant.

The unit tests were not used until after some ad-hoc manual testing was done. The unit tests then found one fault, which was easily fixed.

Considering the minimal length of the code (only 75 lines), simple usage scenarios and extensive testing, it is unlikely that there are faults in the library code that affect the behaviour of the SUT executable.

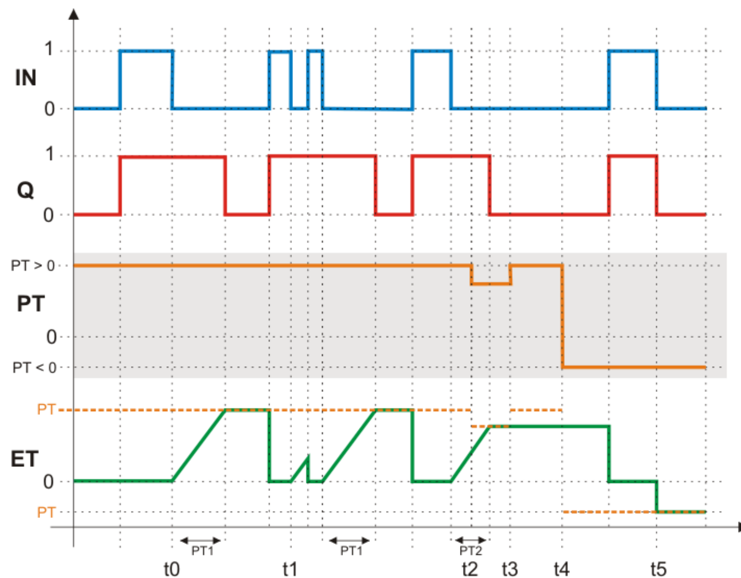


Figure 7.2: Graph showing an example of correct behaviour for a TOF function block, source: [42]

7.4 Interfaces

The main part that is missing from the code that SILworX generates, is the code that handles the first and third phase of every PLC cycle: input and output.

A normal PLC acts autonomously, i.e. it initiates every phase of its cycle on its own. The created program however acts as a slave: it only acts on commands that it receives. Through the external interface the inputs can be set, time can be increased and the program logic can be executed; afterwards variables can be read out again.

7.4.1 Interface to PLC Code

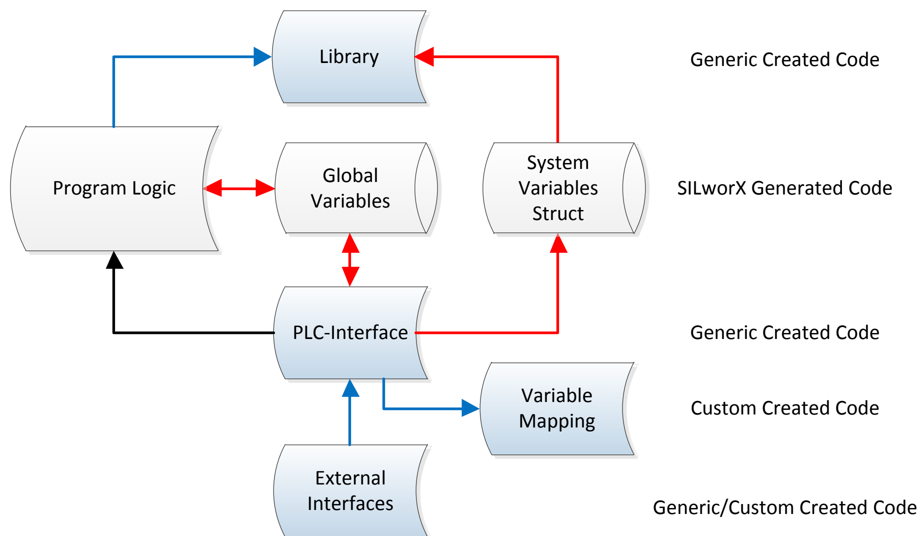
An abstraction layer was created that offers an API to the program logic for the external interfacing code: the plc-interface. This abstraction layer is simple and offers four main functions:

1. Set a global variable.
2. Read a global variable.
3. Increase the virtual time.
4. Execute the program logic.

All global variables (including all input and output variables) from the FBD logic are just that in the C++ code: global variables. A map data structure maps the names of all global variables (as they occur in the FBD logic) to a pointer to the actual variable.

The virtual time is a global variable that holds a number denoting the relative time in milliseconds since an epoch. This variable is part of the 'system variables struct' which also holds some other, unused, fields. The functions in the IEC standard library that are time aware all use the time field from the system variables struct. As a result, the time can be easily manipulated by changing the value of this time field.

Executing the program logic involves nothing more than executing the `CallFunctionContent` function of the class that models the main function of the FBD program.



Legend

Red	Reading or writing a variable (arrow indicates the direction of the data flow).
Blue	Function call, with information being returned.
Black	Execute the program logic (call `CallFunctionContent`).

Figure 7.3: Runtime structure and internal interfaces of the SUT executable

Figure 7.3 shows how the plc-interface communicates with the program logic through variables, and how it executes the program logic by calling the `CallFunctionContent` function. Furthermore it shows how the program logic depends on function calls to the library, that gets information from the global system variables struct. It is important to notice that the most important part of the code, the program logic, is totally unchanged and compiled from exactly the same source code that is used for the binary of a real PLC-Interlocking.

7.4.2 External Interfaces

The SUT executable can switch between two external interfaces that both use the standard input and output streams: A generic command line interface and a specialised interface for communication with JTorX.

Command Line Interface

The command line interface is a shell around the plc-interface. It allows the user to list, read and manipulate all global variables, as well as progress the time and execute the program logic. Furthermore it is also possible to check whether a variable has a certain value, or print an error otherwise. This interface is practical for debugging and (scripted) testing purposes.

The SUT executable has an option to load initialisation scripts. This is useful because a lot of variables are normally set to certain values either by the PLC itself or its environment. To correctly simulate the PLC's behaviour, these variable need to be set to their regular values. The initialisation scripts are interpreted by the command line interface. To enable modularity the interface also has a command to load additional scripts.

JTorX Interface

The JTorX interface translates input transitions from JTorX to function calls to the plc-interface, and creates output transitions from the SUT's outputs.

The exact input and output transitions differ between models. As a result, the JTorX interface has to be tailored to the specific model and PLC-Interlocking instance that are being tested. The JTorX interface that was created is aimed specifically at the model described in the chapters 5 and 6 (where this interface is also referenced as 'SUT adapter'). It implements all transitions of that model:

- The `cycle` transition executes the program logic and increases time by some constant factor.
- The timing related transitions (`timerInput`, `timerOutput` and `cycle_pre`, see section 5.3.3) are implemented by using TON objects from the IEC standard library.
- The parameter values of the `output!Bool` transitions are automatically determined by generic code, based on a custom list with the names of the output variables.
- The argument for each `aspectOutput!Nat` transition is computed by a function that takes all output Booleans related to a single signal as parameters. The calls to this function are customly coded for the model.
- Each input transitions sets a single input variable. A map data structure maps the transition labels to variable names. Given this map, the interface's generic code then automatically processes all input transitions.

More complicated models or models for different PLC-Interlockings will require a renewed (limited) effort to create an interface on the SUT side of the test setup.

7.5 Problems and Alternatives

Testing the PLC logic on a PC instead of a PLC using the method described in this chapter has one major drawback: There is no absolute guarantee that the

SUT executable on a PC behaves exactly the same as the real PLC-Interlocking system.

Any fault, either in the IEC standard library or the modifications of the C++ code, might introduce very subtle behaviour differences between the created PC version and the original PLC version of the software. Even if the code for the PC version is perfect, the use of a different compiler might also create differences in behaviour.

SILworX can be used to create software for SIL-4 systems. However in order to produce such software, strict procedures must be followed. Copying the source code of the program during the compilation is obviously not part of the prescribed procedure. As a result there are no official guarantees about the resulting products. So even the obtained C++ code cannot be guaranteed to be correct.

On the other hand, the chance that a fault in the process of creating the executable SUT hides a fault in the real implementation seems small. It seems more likely that introduced faults would give false positives during testing, than that an introduced fault accidently fixes behaviour that was originally wrong. However this is based on intuition; the exact chances for either scenario cannot be determined.

Ideally the manufacturer would offer tooling to make it possible to create stand-alone SUT executables with a similar text based interface. The manufacturer would be able to better guarantee the correctness of the created executables than is possible with the current approach.

As it stands, the used method was the only viable method for the conducted research. It has great advantages over using a real PLC-Interlocking:

- Access to the PLC hardware is not required. Otherwise, this would have severely limited the testing possibilities.
- Tests are not limited to real time, but can run as fast as the test system allows.
- Multiple tests can run in parallel on one or more test systems.
- Inputs and outputs can directly be manipulated and observed. Testing directly on PLC hardware requires the use of an intermediate system, which complicates the setup and possibly influences timing aspects.

Chapter 8

Test Case Generation

When given a specification model with non-deterministic inputs, JTorX will randomly choose input transitions. Unfortunately this strategy of randomly picking input transitions seldom leads to useful test cases when used with the PLC-Interlocking model described in chapters 5 and 6.

As already mentioned in chapter 4, a test purpose model can be used to guide the model exploration and thus test case generation. The test setup relies on test purpose models implemented in Java to steer test case generation. Creating such models in Java is nontrivial; therefore a framework was created that does most of the heavy lifting, such that the test purpose models only have to deal with one concern.

Section 8.1 explains in detail why randomly generated test cases are not very useful for testing PLC-Interlocking systems, and why using a test purpose model can improve test case generation. Section 8.2 briefly discusses how test purpose models influence test case generation in the test setup. Furthermore it discusses the responsibilities of the Java test purpose models within the created framework. The actual framework is not discussed in this chapter, but in chapter 9. Section 8.3 discusses the different test purpose models, the kind of test cases they generate, and how they are implemented.

Note: Parts of this chapter and the next assume the reader to have basic knowledge about the Java language.

8.1 Random Test Generation Problems

The interlocking model has in total 21 inputs (4 for signals, 12 for track sections and 5 for speed step variables, see table 5.1 and figure 5.1).

8.1.1 Generating Useful Scenarios

The majority of test cases that are randomly generated using the interlocking model are incredibly unlikely to happen and are not very useful.

Because of the non-determinism in the input phase every input variable has a $\frac{1}{3}$ chance of being toggled from true to false or from false to true on every cycle.

After all there is one transition (`cycle_pre`) that fixes all input variables for the current cycle, and for every variable there is one transition to change the variable's value and one transition to set the variable to the value that it already has (this is actually a source of inefficiency, see section 8.1.3). From this it can be easily shown that the chance of a variable changing is $\frac{1}{3}$.

Note that the above assumes that JTorX makes an uniformly distributed random choice out of the possible transitions. The author has confirmed that it behaves this way (in this case) [49].

The $\frac{1}{3}$ chance of an input value flipping will generally cause the generated scenarios to be unrealistic. This is best illustrated by an example: On the Santpoort Noord railway yard (see figure 3.4), the route from signal 507 to signal 521 has a signal delay of 20 seconds. This means that signal 507 is not set to yellow or better until the preconditions for that signal have been met for 20 seconds (in the meantime the beams of level crossing 5.2 will be closed). Assuming a cycle time of one second (actual test value, see chapter 10) this means that there are 21 cycles during which the preconditions have to be met for signal 507 to be set to yellow or better.

One of the preconditions for signal 507 being set to yellow or better is that the input variable `507GZ-CI` should be true. The chance of `507GZ-CI` being true at some point during the model execution is almost 50 %. (Already after four cycles the chance for either false or true is more than 49%, regardless of the initial value (which is false).) Then from that point on every cycle there is a $\frac{1}{3}$ chance of the variable being set to false. So the chance of the `507GZ-CI` part of the precondition being true in a random cycle (given that it is not one of the first 25 cycles) is: $\frac{1}{2} \times \left(\frac{2}{3}\right)^{20} \approx 0.00015$.

However, the variable `507GZ-CI` is only a part of the preconditions for the signal. The seven input variables for each of the track sections behind the signal are part of the preconditions in a similar manner. So the chance of signal 507 being yellow or better during a random cycle is no bigger than $(0.00015)^8 \approx 2.6 \times 10^{-31}$. There are even some more complicated sub conditions, for which the chances of them being met are hard to quantify, that make the chance of the precondition being met even smaller (e.g. conditions related to possible incorrect train detection, see section 5.4.1).

The result is simply that it is extremely unlikely that signal 507 will ever show a yellow or better aspect during a test run where the input variables are randomly determined once per second.

Real world usage scenarios of interlocking systems follow certain patterns: Generally routes are requested through the EBP-interface until a train has entered the route, and trains move in one direction and abide by the shown signalling aspects. Randomly generated test scenarios show completely different behaviour: trains will randomly appear and disappear at track sections, and route requests will be made and cancelled again at random. As a result in generated test scenarios the interlocking system will not do much, because the preconditions for setting a signal to yellow or better are unlikely to be met.

Of course there are exceptions to the patterns of normal usage. Testing such edge cases is one of the goals of model based testing, however merely testing scenarios which are incredibly unlikely to happen is not useful.

A test purpose model can be used to steer the test case generation to include realistic scenarios. However, care must be taken when using a test purpose model, because using it might also exclude test cases that would have resulted in the observation of failures.

8.1.2 Generating Consistent Scenarios

The SUT makes some (allowed) assumptions about relations between input variables. It assumes that the first speed step variable at a railway border is true if and only if the input variable indicating occupation of the section closest to that border is true also. For example on the Santpoort Noord yard, `507-XSS1R-DII` is true if and only if `516E-TR-DII` is true. Furthermore the SUT assumes that the speed step variables themselves are consistent, i.e. speed step 3 can only be true if speed step 2 is true which can only be true if speed step 1 is true. These are valid assumptions, because they are guaranteed by the neighbouring interlocking systems. The SUT does not need to double check these assumptions because the neighbouring systems are also fail safe systems. The developed model makes similar assumptions as the SUT.

Test scenarios where the above mentioned assumptions don't hold give unplanned behaviour and can lead to the observation of failures which are not really failures since they fall outside of the specification. Thus the generated test cases must abide by the assumptions; a test purpose model can be used to make sure that they do.

8.1.3 Eliminating Redundant Transitions

There are 42 input transitions enabled in the input phase: For each variable there is one transition to set the variable to true and one transition to set the variable to false. Additional there is the `cycle_pre` transition to end the input phase. During the input phase, JTorX might take any of the 42 input transitions one or multiple times in any order.

This is a cause of inefficiency, because transitions might cancel out earlier transitions or be taken multiple times without effect. For example the transition sequence `i_7BTR!true i_7BTR!true i_7BTR!false i_7BTR!true pre_cycle` will have the exact same effect as the transition sequence `i_7BTR!true pre_cycle`, namely: setting the input variable `7BTR` to true and ending the input phase. If `7BTR` is already true, then the whole sequence might even be replaced by just `pre_cycle`.

During initial testing it became clear that even taking simple input transitions can take a significant amount of time, thus ideally in the input phase only transitions should be taken that ultimately have an effect on the state in that cycle. A test purpose can force JTorX to only take transitions that change an input variable and only take one transition per input variable per cycle, and thus increase efficiency.

8.2 Test Purpose Models

All of the problems mentioned in section 8.1 could (at least partially) be solved by changing the model of the SUT. However that would severely complicate that model with code that serves a purpose other than the model's main purpose (modelling the SUT's behaviour). A test purpose model offers a much cleaner solution to the mentioned problems, and makes it possible to switch test generation strategy without having to change the specification model.

Given a test purpose model, JTorX puts it in parallel with the model of the SUT. That is, JTorX only considers a transition when it is both in the SUT's model and in the test purpose model. For example, if in some state the system's model has two transitions (e.g. `i_7BTR!true` and `i_7BTR!false`), but the test purpose model has only one of those transitions in the parallel state (e.g. `i_7BTR!true`) then in the combined model only that transition exists in the combined state.

All output transitions of the specification model (`output`, `aspectOutput` and `timerOutput`) are determined by the SUT. For these transitions, a test purpose model should contain all the possible transitions and let the SUT decide which one to take. The input transitions of the SUT's model are the transitions that should be limited by a test purpose model. Without a test purpose model these transitions will be randomly picked, which leads to the problems described in section 8.1.

So the input transitions of the SUT's model have to be the output transitions of the test purpose model. That is, the test purpose model should model the environment of a PLC-Interlocking and the inputs that that environment provides to the PLC-Interlocking. The output transitions of the SUT and the SUT's model can be considered by the test purpose model as inputs.

In the remainder of this chapter as well as in the next chapter, the term 'inputs' refers to the inputs of the test purpose model (i.e. the outputs of the SUT's model). Similarly 'outputs' refers to the test purpose model's outputs which are the inputs to the SUT's model.

8.2.1 Java as Modelling Language

The used test purpose models have not been implemented in mCRL2, like the SUT's model, but rather in Java. After some initial experiments with mCRL2 test purpose models, Java seemed a better choice. The main reasons for this is that Java is (arguably) a more powerful language than mCRL2, which has many limitations.

mCRL2 is a process based language with a data language that supports functions (mappings) that are written in a functional programming style. However, mCRL2's data language lacks the concept of higher order functions which severly hinders the language in comparison to regular functional languages (e.g. any loop construct needs to be implemented recursively). Furthermore, mCRL2 often requires a lot of code for simple tasks, such as changing a field in a data structure, or changing a value in a list.

Part of the limitations of mCRL2 can be explained by the LPS representation that is used by much of the mCRL2 toolset and to which mCRL2 files are compiled. This representation has nice properties which make it very useful for model checking, but model checking is not a concern for test purpose models. Some of mCRL2's other limitations are caused by a lack of 'syntactic sugar' in the mCRL2 language itself (e.g. there is no standard construct to change the n-th item in a list).

The linearisation process of mCRL2 models can also lead to performance problems, both at compile time and at run time, which can be hard to predict beforehand. Multiple development versions of the SUT's model have had such problems, that made it either impossible to compile them or impractical to use them. The performance impact of a statement or construction is much easier to predict with an imperative language than with mCRL2.

The mCRL2 language is designed for modelling distributed systems [47]. However a test purpose model is clearly not a distributed system, so some of mCRL2's strong points (e.g. easily modelling communicating processes) are useless for the purpose of creating test purpose models.

Java on the other hand is a powerful general purpose language. Often, its syntax allows a much compacter notation than needed in mCRL2 to achieve the same result.

Java also has good mechanisms for modularisation and abstraction, which make it possible to isolate the essence of a test purpose model from supporting code. As a result the code of the implemented test purpose models (discussed in section 8.3) is merely concerned with the input values that are provided to the SUT. How that happens (through transitions) and what happens around that (timer transitions etc.) is not a concern solved in the test purpose models, but in a supporting framework. Such a clear separation of concerns is not possible in mCRL2.

Furthermore, some things that are not possible at all in mCRL2 are trivial in Java. For example, creating certain random behaviour in mCRL2 is very hard: Non-determinism with two possible scenarios always defaults to a 50% chance for either scenario when testing with JTorX. With Java on the other hand, it is trivial to create random numbers in any range (this was actually the most important reason not to use mCRL2).

Finally, the availability of development tools (IDEs, debuggers, compilers) for Java is excellent.

Java also has a severe drawback: Its programming paradigm does not match with the LTS based models that JTorX expects (whereas mCRL2's paradigm based on processes with transitions is a perfect match). However, as already mentioned in chapter 4 and depicted in figure 4.3: JTorX does not really need a complete LTS as a model. With on-the-fly model exploration an external program (called a torx-explorer) describes, on request from JTorX, specific parts of the LTS structure that underlies the model. Such a torx-explorer communicates with JTorX over the standard input and output streams using the *torx-explorer protocol*. This same protocol is also used by the Java test purpose models for their communication with JTorX.

8.2.2 Framework for Java Test Purpose Models

Creating test purpose models that communicate with JTorX using the torx-explorer protocol is not trivial. To simplify the test purpose models, a small framework has been built that abstracts away all communication and model exploration tasks. Test purpose models are implemented by extending an abstract class in this framework. Coupling the thus created test purpose class with the framework's classes gives a complete Java program that implements the test purpose model logic and can communicate directly with JTorX.

The implementation of the actual framework is described in chapter 9. The test purpose models themselves are described in section 8.3. The remainder of this section describes the interface that the framework provides to test purpose models.

A test purpose model is created by extending the abstract class `Simulator` (named so because it simulates the PLC-Interlocking's environment). The `Simulator` class has one abstract function that must be implemented: `update`.

A test purpose model can manipulate the inputs to the PLC-Interlocking model (and the SUT) by changing the values returned by its `update` function. The `update` function returns an array of `boolean` variables and takes two parameters:

- `boolean[] inputs`: The inputs to the test purpose model.
- `boolean[] outputs`: The outputs of the test purpose model in the previous scan cycle.

The `boolean` variables in the returned array each correspond with an input of the interlocking model/SUT. For example for the Santpoort Noord model, the `update` function should return an array containing 21 `boolean` variables, matching the 21 input variables of the PLC-Interlocking model. The matching from array indexes to input variables is fixed and governed by the order of the transitions generated by the framework (e.g. the item with index 0 in the returned array corresponds to the `516E-TR-DII` variable in the Santpoort Noord SUT). Constants can be used in the test purpose model's code to identify specific variables. For example the constant `O_516ETR` has value 0 and is used as index for the variable `516E-TR-DII`.

Likewise the items in the `inputs` array correspond to outputs of the PLC-Interlocking model. For the Santpoort Noord model, the `inputs` array will contain four items, each of which will contain the value of one of the EBP yellow or better outputs (e.g. index 0 corresponds to variable `507H-CO`). The `outputs` parameter contains the values returned by the `update` function in the previous scan cycle; passing this as parameter prevents this state information from having to be stored twice.

Implementing the `update` function is the main responsibility of a test purpose model class. However, it must also make sure to implement the `clone` function according to the regular Java conventions, because the framework will clone the `Simulator` subclasses for every state that it encounters.

With those two functions implemented, the created framework can take care of all communication with JTorX and anything needed to support that. The

framework will call `update` for every scan cycle and generate output transitions based on the returned values.

The test purpose model has one responsibility with regard to the values returned by `update`: The values must together form a consistent scenario (as discussed in section 8.1.2). That is, the output variables that correspond to the speed step variables must be consistent with the values of the variables that indicate occupation of the border sections. This is very straight forward to implement as the code samples in the next section will show (e.g. lines 10 to 14 in listing 8.1 show code that takes care of this).

8.3 Test Scenarios

Four different test purpose models have been implemented. One generates test scenarios randomly with some constraints, two create fixed scenarios and one generates scenarios with 'realistically' simulated train movements.

8.3.1 Constrained Random Scenario Generator

The Constrained Random Scenario Generator (from here on abbreviated to CRSG) is a simple test purpose model that generates test scenarios. These scenarios are mostly generated randomly, by setting the values of all section occupations and EBP requests to random values on each cycle.

However, there is a difference between using this test purpose model and having JTorX do a purely random run without a test purpose model: In the scenarios generated by this test purpose model, the speed step variables have values that are consistent with the values of other variables (see section 8.1.2). This is done by setting the speed step variables to the same value as the variables that indicate occupation of the related border sections. As a result all related speed step variables have the same value in the generated test cases. So (valid) scenarios in which the speed step variables 2 or 3 have value different than speed step variable 1 are not generated; this was a small oversight.

Using this test purpose model instead of using no test purpose model also leads to some changes in how transitions are generated that do not affect the set of possibly generated test scenarios (When considering the combined effect of all transitions in a scan cycle instead of looking at individual transitions).

Firstly, with this test purpose model, the Boolean input variables of the SUT are set with an independent 50% chance for either value (disregarding predictability in the randomness source). Without a test purpose model, the chances for the variable values are dependent on the values in the previous cycle (see section 8.1.1).

A more noteable difference in the generated transitions is caused by the used framework for test purpose models: The framework only generates non-redundant transitions. This is further discussed in the next chapter, particularly in section 9.2.

Implementation

This scenario is implemented in the `SptnSimulatorRandom` class. Listing 8.1 shows the `update` function of that class.

Lines 6 to 8 show the elements of the `retOut` array being set to random values. This array is returned at the end of the function, but not before the speed step variables are set to consistent values (lines 10 to 14) using constants for the indexes of the variables in the array.

```
1  public boolean[] update(boolean[] inputs, boolean[] outputs)
2  {
3      /* retOut will contain the new output values */
4      boolean[] retOut = new boolean[outputs.length];
5      /* Set all output variables to a random value. */
6      for (int i = 0; i < retOut.length; i++){
7          retOut[i] = random.nextBoolean();
8      }
9      /* Set the speed step variables to values consistent with the variables
       516E-TR and 530B-TR. */
10     retOut[O_507XSS1] = retOut[O_516ETR];
11     retOut[O_507XSS2] = retOut[O_516ETR];
12     retOut[O_507XSS3] = retOut[O_516ETR];
13     retOut[O_528XSS1] = retOut[O_530BTR];
14     retOut[O_528XSS2] = retOut[O_530BTR];
15     return retOut;
16 }
```

Listing 8.1: The update function of the Constrained Random Scenario Generator test purpose model

8.3.2 Fixed Scenarios

Two fixed scenarios have been defined that are known to trigger faults in the model of the PLC-Interlocking. These faults were discovered during some initial manual testing with the model. Both of the scenarios are only two cycles long.

The first scenario reveals a bug in the model with regard to the preconditions for signal 516. The model assumes that section 528D needs to be empty for signal 516 to be set to yellow or better. However that is actually not the case, because of some special circumstances surrounding that section (which are unrelated to the interlocking system). When determining whether signal 516 can be set to yellow or better, the SUT regards section 528D to lie before signal 516. This is a situation specific exception to the normal interlocking rules, and as such the model's general design is correct. However, the model has not been adapted for this particular situation and as such is wrong.

The scenario sets all sections to unoccupied and all EBP request to false in the first cycle. In the second cycle section 528D is set to occupied and the EBP request for yellow or better of signal 516 is set to true. This will make the SUT set signal 516 to yellow or better, while the model finds that the signal should be red.

The second scenario also involves signal 516, but reveals another fault in the model. The first scenario actually also triggers this fault, but the second scenario makes it clear that it is an unrelated fault. The scenario is basically the same as the previous scenario, with the exception that section 528D is not set to occupied. The SUT will set signal 516 to yellow or better when this scenario is executed, but the model still expects the signal to stay red in that situation.

This scenario actually shows a fundamental flaw in the model. The model assumes that the pre-conditions for signal 516 must hold for 25 seconds before the signal can be set to yellow or better (such that the beams of level crossing 5.7 have time to close). However, this turned out to be a misinterpretation of the specifications: If all sections before signal 516 are also empty, then signal 516 can be set to yellow or better immediately and there is no need to wait for 25 seconds.

Implementation

The above described scenarios are implemented in the classes `SptnSimulatorScenario1` and `SptnSimulatorScenario2`. Both implementations are similar and very straightforward: There is a cycle counter variable which indicates at which point in the scenario the test purpose model is, and based on its value the SUT's input variables are set.

Listing 8.2 shows the `update` function of the `SptnSimulatorScenario1` class. Lines 4 to 21 show how all variables are set in the first cycle of the scenario. Lines 23 to 25 show how in the second cycle the EBP request for signal 516 is set to true (`retOut[O_516GZ] = true`) and section 528D is set to be occupied (`retOut[O_528DTR] = false;`). All other SUT input variables are assigned the same values in the second cycle as in the first cycle; these assignments are thus not shown in the code listing.

The implementation of `SptnSimulatorScenario2` is nearly identical to the code shown in listing 8.2. The only difference is in line 24 of the code listing: In the `SptnSimulatorScenario2` class this line sets the variable to true.

```
1  public boolean[] update(boolean[] inputs, boolean[] outputs)
2  {
3      boolean[] retOut = new boolean[outputs.length];
4      if (stage == 0){
5          retOut[O_507GZ]  = false;
6          retOut[O_516GZ]  = false;
7          retOut[O_521GZ]  = false;
8          retOut[O_528GZ]  = false;
9          retOut[O_516ETR] = true;
10         retOut[O_516DTR] = true;
11         retOut[O_516CTR] = true;
12         retOut[O_516BTR] = true;
13         retOut[O_7BTR]   = true;
14         retOut[O_516ATR] = true;
15         retOut[O_528DTR] = true;
16         retOut[O_528CTR] = true;
17         retOut[O_19TR]   = true;
18         retOut[O_528BTR] = true;
19         retOut[O_528ATR] = true;
```

```
20          retOut[O_530BTR] = true;
21          stage = 1;
22      } else {
23          retOut[O_516GZ]  = true;
24          retOut[O_528DTR] = false;
25          ...
26      }
27      /* Set the speed step variables to consistent values. */
28      ...
29      return retOut;
30  }
```

Listing 8.2: The update function of the SptnSimulatorScenario1 test purpose model

8.3.3 Simulated Train Movements

The Simulated Train Movements (from here on abbreviated to STM) test purpose model tries to generate 'good' scenarios, that is: scenarios which are not completely unlikely, but also not predictable, and might lead to the observation of failures. The algorithm used to generate these scenarios implements heuristics that are based on intuition and some experimentation, not on any formal research into what constitutes good scenarios.

The basic idea behind the test purpose model is that it simulates the interlocking's environment. Two kinds of entities are simulated in this environment: The logistics system that makes EBP-requests and trains that cause section occupations. The simulation of these entities is simple and does not even follow the usual protocol (e.g. trains completely ignore the signals). The simulation uses random numbers for decision making, such that every sufficiently long generated scenario is unique.

The simulation of the logistics system is very simple: Every cycle, each EBP request is set individually. In principle an EBP request remains the same as the previous cycle, unless a random variable has a certain value (a chance of $\frac{1}{150}$).

Two trains are simulated to ride on the tracks of the railway yard. The simulated trains randomly enter the railway yard on a side and ride to the other side with a variable speed, where they exit the yard again.

Each train is modelled by an object with three variables: a position, travel direction and speed. Each cycle a new position is calculated based on the previous position, direction of travel and speed.

The above results in trains driving randomly over the railway track. To make the generated scenarios more interesting, an extra element of randomness is added: Once in a while the variables of a train are set randomly. This should lead to the generation of scenarios that are unlikely but possible, for example scenarios with trains that reverse, or scenarios with trains that are not detected until they appear in a certain sector.

The simulation of the logistics system is markedly different from real world scenarios were EBP requests will almost never conflict and EBP requests are mostly recalled as soon as a train has passed a signal.

Similarly, trains are normally more predictable: They abide by the signalling, have certain speeds at certain points, and take a certain amount of time to cross sections, etcetera. The simulated trains are a lot more erratic: they ignore signalling completely, the time they need to cross a section is randomised and bears no direct relation to the physical length of that section, and they might even reverse.

However the goal was not to create scenarios in which all entities behave perfectly according to protocol and normal constraints, but to create scenarios that are 'good' or 'interesting' as test cases for the SUT. As said before, the created implementation is based on intuition and some experimentation; It is very probable that 'better' test purpose models (i.e. models that find faults more consistently) can be constructed.

Implementation

The Simulated Train Movements test purpose model is implemented in the `SptnSimulator3` class (there where some experimental iterations, hence the 3 in the name). The simulation of train movements is implemented in a separate `Train` class.

Listing 8.3 shows the main parts of the `update` function of the `SptnSimulator3` class.

The section occupations are determined on the lines 4 to 11. First a section array is created with 12 fields (one for each section on the modelled part of Santpoort Noord), then the fields in this array are set depending on the trains' positions. This relies on the `update` function of `Train` objects, that each return an array with section occupations for that train (line 7). The implementation of the `Train` class is further discussed below.

The EBP requests for yellow or better are determined on the lines 13 to 17, based on the values of these variables in the previous cycle. After which both the section occupations and the ebp requests are copied to the output array in lines 19 and 20.

```
1  public boolean[] update(boolean[] inputs, boolean[] outputs) {
2      boolean[] retOut = new boolean[outputs.length];
3      /* Trains: */
4      boolean[] sections = new boolean[12];
5      ...
6      for (Train train: trains){
7          boolean [] tOuts = train.update(inputs, outputs);
8          for (int i = 0; i < sections.length; i++){
9              sections[i] = sections[i] && tOuts[i];
10         }
11     }
12     /* Logistic System: */
13     for (int i = 0; i < requested.length; i++){
14         if (random.nextInt(EBP_RANDOM_SIZE) == 0){
15             requested[i] = !requested[i];
16         }
```

```
17     }
18     /* Start filling the output array: */
19     System.arraycopy(sections, 0, retOut, 0, sections.length);
20     System.arraycopy(requested, 0, retOut, 12, requested.length);
21     /* Set the speed step variables to consistent values. */
22     ...
23     return retOut;
24 }
```

Listing 8.3: The update function of the `SptnSimulator3` test purpose model

Listing 8.4 shows the `update` function of the `Train` class. The code starts by either determining a new speed randomly (lines 12 to 18), or if the train is currently not on the railway yard it starts by determining whether it should enter the railway yard and then decide a speed (lines 4 to 11).

After this a new position is calculated on line 20. The position is expressed as an integer, that ranges from 0 (inclusive) to 120 (exclusive) to indicate a position on the Santpoort Noord railway yard. Each of the 12 track sections is represented by 10 integer values. Values below 0, and 120 and above indicate that the train is either at the north or south side of the railway yard waiting to enter the yard.

The lines 21 to 26 occasionally set the the `Train`'s variables to random values. After which the positions are converted to section occupations on line 29, that are then returned to the calling function (see listing 8.3, line 7).

```
1  boolean[] update(boolean[] inputs, boolean[] outputs){
2      boolean[] retOut = new boolean[outputs.length];
3      Random r = SptnSimulator3.random;
4      /* Determine new speed, and/or whether to enter the railway yard: */
5      if (position < 0 || position >= 120){
6          if (r.nextInt(ENTRY_CHANCE) == 0){
7              speed = r.nextInt(MAX_SPEED);
8          } else {
9              speed = 0;
10         }
11         direction = (position < 0);
12     } else {
13         int nSpeed = (r.nextBoolean() ?
14                         speed+r.nextInt(MAX_SPEED_CHANGE) :
15                         speed-r.nextInt(MAX_SPEED_CHANGE)
16                      );
17         speed = Math.max(0, Math.min(MAX_SPEED, nSpeed));
18     }
19     /* Update position: */
20     position = position + (direction ? speed : (-1 * speed) );
21     /* Random mutations: */
22     if (r.nextInt(RANDOM_MUTATION_RATE) == 0){
23         position = r.nextInt(120);
24         speed = r.nextInt(MAX_SPEED);
25         direction = r.nextBoolean();
26     }
27     /* Calculate the outputs based on the position: */
28     for (int i = 0; i <= O_530BTR; i++){
```

```
29          retOut[i] = (((position + 10) / 10) - 1) != i;
30      }
31      ...
32      return retOut;
33  }
```

Listing 8.4: Simulation of train movements by the `Train` class of the STM test purpose model

Chapter 9

Java Framework for Test Purpose Models

The test purpose models described in the previous chapter rely on a framework for generating an LTS representation out of their output. The previous chapter covered the interface that the framework requires from the test purpose models (section 8.2.2). This chapter describes how the framework has been implemented, and how it communicates the outputs of a test purpose model to JTorX.

As already said in the previous chapter, Java has no native way of generating an LTS like mCRL2 has, and doing so is non-trivial. Furthermore, there is no known precedent for using Java for the test purpose model in a test setup with JTorX. As such the framework that is described in this chapter had to be developed specially for the PLC-Interlocking test setup.

Before describing the framework itself, section 9.1 explains the protocol used to communicate with JTorX. Section 9.2 then describes how the outputs of a test purpose model relate to the transitions communicated to JTorX by the framework. After that starts a description, spread over three sections, of how the framework has been implemented. Section 9.3 gives a short overview of the classes that make up the framework, their responsibilities and their interrelations. Section 9.4 describes the main loop of the program. Section 9.5 describes an important function that the main loop relies on: The function that determines the outgoing transitions from a state, and the successor states that follow these transitions. Finally section 9.6 discusses the generality of the created solution, and some of its limitations.

9.1 The Torx-Explorer Protocol

The protocol between JTorX and a torx-explorer is very simple and well documented in the manual pages of the original TorX tool [50].

After an initial reset command (‘r’), the torx-explorer gives the identifier of the transition that leads to the initial state. JTorX can then continually use the expand command (‘e <id>’) to instruct the torx-explorer to expand a certain transition. The torx-explorer replies to the expand command with the list of the (uniquely numbered) transitions that are enabled in the successor state.

The transition list is preceded by a line containing just the string ‘EB’ and succeeded by a line containing ‘EE’. Each line containing a transitions starts with a prefix (‘Ee’) followed by multiple tab separated data fields: the first field is the unique transition identifier and the fourth field is the transition label; the other fields are for extensions to the basic protocol and need not be used.

The protocol allows expanding any communicated transition at any time, such that JTorX can explorer multiple paths through the state space. As a result of this, the torx-explorer has to keep all past states in memory. However, in practice this is not really needed, because JTorX follows a single path through the state space with some limited branching around the main path.

The protocol contains a command for JTorX to indicate to the torx-explorer that a list of states will not be expanded further and can be discarded (‘d <ids>’). However, this command is not implemented by lps2torx, nor used by JTorX.

```
1  ->  r
2  <-  R   0   1
3  ->  e 0
4  <-  EB
5  <-  Ee  1   1   1   i_516ETR!true
6  <-  EE
7  ->  e 1
8  <-  EB
9  <-  Ee  2   1   1   i_516DTR!true
10 <-  EE
11 ->  q
12 <-  Q
```

Listing 9.1: Example communication between JTorX and a torx-explorer

Listing 9.1 shows an example communication scenario between JTorX (messages prefixed with ‘->’) and a torx-explorer (messages prefixed with ‘<-’). Line 1 shows the initial reset command, with the torx-explorer’s reply on line 2. The initial transition is then expanded on the next line. The reply to this (lines 4 - 6) contains one transition (i_516ETR!true) with identifier 1. This transition is then further expanded in the lines 7 to 10. The successor state again has one outgoing transition (labelled i_516DTR!true with identifier 2). Further model exploration is then stopped.

9.2 Framework: Bridge between JTorX and Test Purpose Models

As already said in the previous chapter, the framework is responsible for the interface between JTorX and the update function. Transitions are automatically

generated by the framework to respond to expand commands from JTorX. To do this the framework tracks in which stage of the scan cycle the system is, and which transition should be generated next.

In the communication with JTorX, the generated transitions can be divided in two main groups: the output transitions and non-output transitions (which can be further divided in input and other transitions).

If, at the current stage of the scan cycle, a non-output transition is generated, then all (at that point) possible transitions are generated. This guarantees that the choice of available transitions for the SUT and PLC-Interlocking model is not restricted by the test purpose model at all.

The output transitions are a different case. The test purpose model's output transitions are the transitions that set the input variables of the PLC-Interlocking model and the SUT. The framework calls the `update` function to determine the values communicated with these transitions before the first output transition of a scan cycle is outputted. Transitions are only generated for variables whose values have changed, and these variables are outputted in fixed order. This prevents redundant transitions (as discussed in section 8.1.3).

9.2.1 Example Scenario

```
1  -> e 76
2  <- EB
3  <- Ee 92    1   1   aspectOutput!0
4  <- Ee 93    1   1   aspectOutput!1
5  <- Ee 94    1   1   aspectOutput!2
6  <- ...
7  <- Ee 107   1   1   aspectOutput!15
8  <- EE
9  -> e 92
10 <- EB
11 <- Ee 108   1   1   i_528DTR!false
12 <- EE
13 -> e 108
14 <- EB
15 <- Ee 109   1   1   ei_516GZ!true
16 <- EE
17 -> e 109
18 <- EB
19 <- Ee 110   1   1   cycle_pre
20 <- EE
21 -> e 110
22 <- EB
23 <- Ee 111   1   1   timerInput!20000!true
24 <- Ee 112   1   1   timerInput!20000!false
25 <- EE
```

Listing 9.2: Example communication with a test purpose model

Listing 9.2 shows a snippet from a sample communication with a test purpose model.

The first lines (3 - 7) show how the last input transitions of a scan cycle are generated. Since these are input transition, all possible transitions are generated (notice that multiple lines have been skipped from the listing at line 6). One of these transitions is then expanded with the command on line 9, which starts the next scan cycle.

The start of a new scan cycle prompts the test purpose framework to call on the `update` function, which in this case gives new values for two output variables. This results in two output transitions that are generated one at a time (lines 11 and 15). After having communicated all changes, the test purpose model signals the end of its output phase (i.e. the SUT's input phase) with the `cycle_pre` transition.

The final lines show again that all possible transitions are generated during the input phase (lines 23-24).

9.3 Model Structure

The framework consists of a number of generic classes (`ModelGuide`, `State`, `Transition` and `Simulator`) and one class that specialises `State` with static data for the Santpoort Noord model (`SptnState`). Furthermore multiple test purpose models have been implemented (in classes named `SptnSimulatorXYZ`, with XYZ variable). Table 9.1 contains an overview of all classes of the test purpose models.

Table 9.1: Overview of classes in the Java test purpose model

Class	Description
`ModelGuide`	Is responsible for all input and output, and state space exploration in accordance with the received inputs. Contains the main program loop (described in detail in section 9.4).
`State`	Models a single state of the model. Furthermore has the responsibility of determining the outgoing transitions and successor states (described in detail in section 9.5).
`Transition`	Simple data container that models a transition to a `State`.
`Simulator`	Abstract class that is contained in each `State`. Subclasses of this class must implement the test purpose logic.
`SptnState`	Subclass of `State` that contains static data regarding the transitions that exist in the Santpoort Noord Model.
`SptnSimulatorXYZ`	A number of these classes exist, each of which extends `Simulator`. These classes are not part of the generic framework; each implements a different test purpose model.

All classes combined form a complete program. The whole program's execution is, except for some initialisation, spent in the program's main loop implemented in `ModelGuide.explorer`. The initialisation code calls the `explorer` function with as argument an initial `State` object. This initial `State` object is simply a `SptnState` object with all fields initialised to some default values. This `State` object also contains an initial `Simulator` object, which is set by the initialisation code. The initialisation code can determine the test case genera-

tion strategy used in the main loop, by setting particular subtypes of `Simulator` (which implement the test purpose models) in the initial `state`.

9.4 Main Loop

The `ModelGuide` class contains the `explorer` function which, given an initial state, explores the state space in accordance with the received inputs. Furthermore this class also contains all input and output related functions on which the `explorer` function relies.

The `explorer` function runs in a loop in which it reads torx-explorer commands from the standard input, acts on them, and prints replies to the standard output. It can reset to the initial state, explorer a specific state, discard previously explored states or quit the program. All states (modelled by the `State` class) that have already been explored are kept in a data structure that maps transition identifiers to the states to which they lead.

Listing 9.3 shows a simplified version of the main loop. Most code that is not essential to executing the reset and expand commands has been left out for readability purposes (note: not all of these omissions are marked by ...).

```
1  public static void explore(State initialState){
2      long nextTransitionId = 0;
3      Map<Long, State> stateStore = new HashMap<Long, State>();
4      ReadInput pick = readTransition();
5      while (true){
6          if (pick.type == ReadInput.QUIT){
7              break;
8          } else if (pick.type == ReadInput.RESET){
9              /* Clear state space and reset to the initial state. */
10             nextTransitionId = 0;
11             stateStore = new HashMap<Long, State>();
12             stateStore.put(Long.valueOf(nextTransitionId),
    initialState.clone());
13             nextTransitionId++;
14             stdOutPrintLine("R 0\t1");
15         } else if (pick.type == ReadInput.DELETE){
16             ...
17         } else if (pick.type == ReadInput.EXPAND){
18             /* Expand transition. */
19             if (stateStore.containsKey(Long.valueOf(pick.transitionId))){
20                 State curState= stateStore.get(Long.valueOf(pick.transitionId));
21                 if (curState.getTransitions() == null){
22                     nextTransitionId =
    curState.determineTransitions(nextTransitionId);
23                     for (Transition transition: curState.getTransitions()){
24                         stateStore.put(transition.transitionId,
    transition.nextState);
25                     }
26                 }
27                 writeTransitions(curState.getTransitions());
28             }
29         }
```

```
30          pick = readTransition();
31      }
32  }
```

Listing 9.3: Main loop of the test purpose model framework

Input is read before the loop on line 4 in the above listing, using the function `readTransition` and later again in the loop on line 30.

The processing of reset commands is shown on lines 8 to 14. First the `stateStore` variable is initialized to an empty `HashMap<Long, State>`. Then a clone of the initial state is added as the first state with identifier `0`. The `nexTransitionId` is set to the first unused transition identifier (`1`). After this, a reply with the initial state's identifier is communicated to JTorX.

The lines 17 to 28 show the processing of expand commands. First the `State` object representing the state to which the expanded transition leads is retrieved from the `stateStore` (line 20) If the transition has not been expanded before (which is the typical scenario), then the outgoing transitions and successor states of the `State` have yet to be determined. In that case these will be determined first (lines 21 - 26). The transitions and states are determined by calling the `determineTransitions` method on the current `State` object, which also computes a new value for `nextTransitionId`. (Note: How a `State` objects determines its outgoing transitions is explained in section 9.5.) After determining the transitions, they are retrieved and the states to which they lead are stored in the `stateStore` mapping. Finally, the outgoing transitions from the current state are outputted to JTorX (line 27).

Note that for every transition expanded, all successor transitions and states are stored (i.e. a `State` object is created and stored in the `stateStore`). This is somewhat inefficient, because most of these states will not be expanded further (e.g. the scenario in listing 9.2 starts with 16 `aspectOutput` transitions, of which only one will be expanded in a normal use case). More efficient, with regard to both run time and memory consumption, would be to only store all successor transitions of expanded transitions and to determine the state only for transitions that are actually expanded. However, the solution presented here was significantly easier to implement and thus chosen. Furthermore, the impact on the run time of the complete test setup seems not very significant (considering all parts that contribute to this). The impact on memory usage on the other hand turned out to be significant because of a combination with other issues (see section 10.4.1), this was unforeseen.

Noteworthy is also that every newly explored `State` object is distinct from any previously explored `State`. As such for each `State` in the mapping there is only one transition identifier that maps to it.

This is in contrast to state space exploration strategies that consider states with the same instance variables to be equal. Such strategies can save memory and run time when reencountering previously explored states.

However, considering each state to be distinct also has an advantage. It allows the test purpose model to decide the outgoing transitions (through the `update` function) for otherwise similar states every time the states are visited. This for example makes it possible to create random outputs, even if a state with the same input variable values has already been visited.

9.5 Determining Successor States

Each `State` object is responsible for determining its outgoing transitions and successor states. To this end it offers the `determineTransitions` method. But before discussing the implementation of that function, some more information on the `State` class will be provided.

9.5.1 Fields of the State Class

A `State` object models the state of the model using a number of instance variables:

- `boolean[] inputs`: Each item in this array represent one of the Boolean output variables of the SUT/specification model, which can be used as inputs by the test purpose models.

- `OutputBit[] outputBits`: Each item in this array models one Boolean output variable (i.e. an input variable to the SUT/specification model). Each `OutputBit` not only holds the current value, but also the value of that same variable in the previous scan cycle.

- `int transitionStage`: This integer indicates at which stage in the scan cycle the model is in the current state. Before the very first output variable is communicated this variable is set to `0`, after the first is communicated to `1`, and after the next to `2`. The amount of possible consecutive transitions in each scan cycle is fixed for each model. As a result `transitionStage` can exactly pinpoint at which stage in the scan cycle the model is.

- `Transition[] transitions`: This array is either `null`, if the current state has not been expanded yet, or contains the `Transition` objects that model the outgoing transitions if a call to the `determineTransitions` method has occurred. The `Transition` objects held in this array contain multiple fields:

  - `transitionId`: A `long` containing the unique transition identifier.
  - `label`: A `String` containing the transition label.
  - `nextState`: A `State` object modelling the state following the transition.

- `Simulator simulator`: The class implementing the actual test purpose model. Each `State` has its own instance of this class so that the `Simulator` classes can also hold local state that is related to the current `State` object.

Whenever a `State` is expanded, all its successor `State`s are objects that are created using the `clone` function of the original `State`, and with the relevant fields updated. For that reason it is also important for `Simulator` classes using the framework that hold local state to implement the `clone` function properly.

9.5.2 Santpoort Noord Specific Subclass

The `State` class is very generic and has no knowledge of the actual transitions that need to be generated during a scan cycle of a specific interlocking model. Subclasses of `State` (which is an abstract class) provide this information by implementing six abstract functions that return information regarding the transitions that need to be generated. For the Santpoort Noord model the class `SptnState` has been defined as a subclass of `State`.

The functions implemented in `SptnState` are the following:

- `boolean isOutputTransition(int transitionStage);`
  Indicates whether the transition in the given transition stage communicates an output variable.
- `boolean isInputTransition(int transitionStage);`
  Indicates whether the transition in the given transition stage sets an input variable.
- `int inputTransitionStageToInput(int transitionStage);`
  Given a transition stage in which an input variable is set (i.e. isInputTransition(transitionStage)), gives the index of that variable in the inputs array.
- `int getNumTransitionStages();`
  Gives the number of transition stages in the model (i.e. the max value for `transitionStage` plus one).
- `int getTransitionLabelSetSize(int transitionStage);`
  Gives the number of enabled transitions for the transitionStage (e.g. `1` for an stage with an output transition, and `16` for an `aspectOutput` transition stage).
- `String getTransitionLabel(int transitionStage, int transition;`
  Gives the label of a transition.

Besides implementing the above functions, subclasses of `State` are also responsible for initialising the `inputs` and `outputBits` arrays.

The implementation of `SptnState` is very straight forward. It has an array that defines per transition stage the possible transition labels. Combined with a few constants that define at which transition input and output transitions start give enough information for very straight forward implementations of the above mentioned functions.

Part of the definition of the array that is at the heart of `SptnState` can be seen in listing 9.4. Each row in the array represents a transition stage, the array thus reflects the order in which transitions need to be generated.

```
1  public static final String[][] TRANSITION_LABELS =
2 { {"i_516ETR!true" , "i_516ETR!false" }
3 , {"i_516DTR!true" , "i_516DTR!false" }
4 ...
5 , {"cycle_pre"}
6 , {"timerInput!20000!true", "timerInput!20000!false"}
7 ...
```

```
8 , {"cycle"}
9 , {"timerOutput!true", "timerOutput!false"}
10 ...
11 , {"output!true", "output!false"}
12 ...
13 , {"aspectOutput!0", "aspectOutput!1", "aspectOutput!2", "aspectOutput!3",
    "aspectOutput!4", "aspectOutput!5", "aspectOutput!6", "aspectOutput!7",
    "aspectOutput!8", "aspectOutput!9", "aspectOutput!10", "aspectOutput!11",
    "aspectOutput!12", "aspectOutput!13", "aspectOutput!14", "aspectOutput!15"}
14 ...
15 };
```

Listing 9.4: Definition of transition labels per transition stage in `SptnState`

9.5.3 The determineTransitions Method

Using the functions providing model specific information, and the information held in the `State`'s instance variables, the `determineTransitions` method computes the outgoing transitions for a `State` object.

The method takes one argument (`long startId`) that indicates the next free transition identifier. The method also returns a `long`, which indicates the next free transition identifier after the method has determined the outgoing transitions. Listing 9.5 shows the `determineTransitions` method with some selected parts left out for readability.

```
1 public long determineTransitions(long startId){
2     /* State cycle: First invoke the update function on the Simulator. */
3     if (transitionStage == getNumTransitionStages()){
4         this.performComputations();
5         transitionStage = 0;
6     } /* Skip output transitions if the output values haven't changed: */
7     int virtualTransitionStage = transitionStage;
8     while (isOutputTransition(virtualTransitionStage))
9     {
10        if (outputBits[virtualTransitionStage].value !=
    outputBits[virtualTransitionStage].lastValue) {
11            break;
12        } else {
13            virtualTransitionStage++;
14        }
15    } /* Determine transitions: */
16    Transition[] newTransitions;
17    if (isOutputTransition(virtualTransitionStage)){
18        /* Boolean output: Only show a transition with the determined output. */
19        newTransitions = new Transition[1];
20        State nextState = this.clone();
21        nextState.transitionStage = virtualTransitionStage + 1;
22        String label = getTransitionLabel(virtualTransitionStage,
    (outputBits[virtualTransitionStage].value ? 0 : 1));
23        newTransitions[0] = new Transition(label, startId, nextState);
24    } else if (isInputTransition(virtualTransitionStage)){
25        /* Input: Create successor states for the true and false branches. */
26        ...
```

```
27      } else { /* All other transitions. */
28          ...
29      }
30      this.transitions = newTransitions;
31      return startId + this.transitions.length;
32  }
```

Listing 9.5: The `determineTransitions` method

The `performComputations` function is called before the first transitions in a scan cycle (lines 3 to 6). This function in turn will call the `update` function on the `State`'s `Simulator` and update the `outputBits` array based on the returned values.

Then lines 7 to 15 are concerned with skipping needless output transitions. If the output transition that is associated with the `transitionStage` is redundant (i.e. the value has not changed since last cycle), then the transition stage will skip ahead until a non-redundant transition is found.

From line 16 onwards the code is concerned with creating the needed `Transition` objects. There are three separate cases here:

- Output transitions (lines 17 to 23).
- Input transitions (mostly skipped from the code listing).
- Other transitions (mostly skipped from the code listing).

On line 20 the next state for an output transition is created by creating a clone of the current state. On the next line, the `transitionStage` variable of the new `State` object is changed so that it correctly represent the next state. Line 22 selects the label for the transition leading to the next state; the selected label depends on the value of the output variable that is communicated with the transition. Finally, the code on line 23 actual creates the `Transition` object.

The code for the 'other' transitions is similar to that of the output transitions. The main difference is that multiple `Transition` objects are created instead of just one (one for each possible transition).

The code for creating input transitions always creates two `Transition` objects: one for the `output!true` transition and one for the `output!false` transition. The `nextState` objects of these `Transitions` have the input variable to which the transitions correspond set, so that the states reflect the taken input transition.

Finally, the return statement on line 31 returns the first transition identifier that is still free.

9.6 Adaptability

The created framework can easily be adapted and used for creating test purpose models for test setups of other PLC-Interlocking systems, or test setups of the Santpoort Noord installation with more inputs and/or outputs. All that is needed for that is adding a subclass of `State` that defines the model's transitions

similar to how `SptnState` does that for the model used in the current test setup. Of course, the classes implementing the actual test purpose models would also have to be rewritten or adapted.

The framework is not even necessarily limited to test purpose models for PLC-Interlocking systems. It could in principle be used for any deterministic model that can be modelled by a single function (`update`) that is executed cyclically. As such it could be used to write the test purpose model for any test setup that tests a PLC based product. Or it could even be used to write the model for any PLC based SUT, as long as that model can be fully deterministic. For example, the model of the PLC-Interlocking could also have been implemented in Java using the framework presented here (with some slight adaptations).

However, the framework cannot be used for models that need to be non-deterministic. Nor can it be used for models that have a different structure than the here discussed PLC-based cyclic models, which can be characterised by a single function that is executed every cycle. In fact, non-deterministic models with a complicated structure might be hard to implement in Java in general. For such models, a language built around (non-deterministic) transitions like mCRL2 might be more appropriate. However, further research is needed before statements about the usefulness of Java for (test purpose) models in general can be made with certainty.

Chapter 10

Testing and Results

Previous chapters have described the developed test setup. This test setup has been used in a number of test runs to determine its effectiveness. This chapter describes these test runs and their results.

The performed tests had two main goals:

1. Determining the test setup's performance.
2. Determining how well the test setup performs its task of finding faults.

The structure of the rest of this chapter is as follows: Section 10.1 describes the executed test runs, and the system on which they were conducted. Section 10.2 describes how the performance of these test runs has been measured. Section 10.3 and 10.4 give an overview of the execution time of test runs and the memory usage during those test runs.

Section 10.5 then discusses the qualitative behaviour of the used test setup and lists the failures that have been observed. Finally, section 10.6 evaluates the findings from section 10.5.

10.1 Executed Test Runs

More than 70 test runs have been executed that are documented here. The two test runs with the fixed scenarios (see section 8.3.2) were known beforehand to (practically) instantly yield a 'fail' verdict. More interesting are the test runs conducted with the Simulated Train Movement (STM) test purpose model (see section 8.3.3) and the Constrained Random Scenario Generator (CRSG) test purpose model (see section 8.3.1). Therefore the rest of this chapter will mostly focus on those. The purpose of these test runs was twofold: finding faults and gathering performance measurements.

Beforehand it was clear that the CRSG test purpose model is very unlikely to create test runs that lead to failures. This test purpose model has been used in a series of 20 test runs of increasing length (from 100 to 650,000 transitions, with

the test lengths increasing roughly exponentially). The main goals of these test runs were to get a baseline to which to compare the other test purpose model, and to get performance measures for a great variety of workloads. Therefore, any runs in which a failure was observed and that did thus not run until the end (three runs) were rerun to get measures for the full run.

The simulated train movements test purpose model has a good chance of finding a fault during a test run. Most tests have therefore been performed with this test purpose model. In total 50 test runs were executed with a goal length of 100,000 transitions each. However a test run is stopped immediately once a failure is observed; as a result a lot of test runs were much shorter.

There are remarkable differences in the results gathered with the different test purpose models; not only with regard to the likelihood of observing a failure, but also with regard to the performance.

10.1.1 Test Setup Details

All tests were performed on a system with dual Intel Xeon X5550 CPUs and 144 GiB of memory running under GNU/Linux. JTorX version 1.10.0-beta6 was used in console only mode (no GUI) with all outputs redirected to log files. The specification model has been explored using the lps2torx implementation from the mCRL2 toolkit, version: January 2012 (development).

The tested SUT was created from version 3 of the interlocking logic for Santpoort Noord.

Scan Cycle Length

As mentioned in chapter 7 the `cycle` transition executes the program logic, but also increases the virtual time by some constant. In the version of the SUT used in the executed tests the `cycle` transition takes one second.

The actual PLC-Interlocking system at Santpoort Noord has a scan cycle length of about 30 milliseconds (with a few milliseconds variation between cycles). Nonetheless, a fixed one second cycle length for the model seems appropriate, because it allows a test run to cover a longer simulated time interval, yet seems unlikely to prevent potential failures from occurring.

The above is actually a bit simplified, internally the SUT executes two cycles of half a second each when receiving a `cycle` transition. Before each of these cycles some input variables (which are not part of the model) are flipped. Flipping these variables indicates to the interlocking logic that an external system is working correctly, without this the SUT will show alternative aspects. Dividing one scan cycle of the model in two scan cycles for the SUT can potentially cause differences in behaviour between the model and the SUT. However with the currently used model, no such discrepancies are foreseen.

The internal behaviour of the SUT with regard to scan cycles is mostly ignored in this report; unless noted otherwise, a mention of a scan cycle will refer to a scan cycle according to the model.

10.2 Performance Measurements

The goals of the conducted performance measurements were simple: Determining how well the test setup performs, and how well it scales to larger test runs. A secondary goal is determining the performance of the individual components of the system, such that relative performance and bottlenecks can be identified. Particularly, two performance characteristics are of interest and have been measured:

- The execution time of a test run.
- The memory usage during a test run.

These performance characteristic have been measured during the test runs for the complete system, as well as for the individual components (JTorX, lps2torx, the test purpose model and the SUT) of the test setup.

10.2.1 Measurements

The execution time of a test run was measured with the standard Unix utility *time* (using the GNU implementation). The used measurements are the values that time reports as the 'Elapsed (wall clock) time'.

The memory usage and run time of individual components was determined using the standard Unix utility *top*. A shell script ran top once every 10 seconds during the execution of each test run.

The values reported by top in the 'TIME+' column are used as the execution time of the individual components. Notice that this is not the elapsed time (like the measurements of the time utility), but the used CPU time. That is, if a process runs for 1 second and fully utilises 4 CPU cores, this field will read 4 seconds. JTorX as well as the test purpose model have some (implicit) multi-threading (e.g. garbage collections is done in a background thread). As a result, for those programs the used CPU time reported by top is higher than the elapsed time.

The values reported by top in the 'RES' column are taken as the amount of memory used by the individual components (and together as the total memory used). This column reports the 'resident memory', i.e. the amount of physical memory used. This is often considerable less than the amount of virtual memory (shown in the 'VIRT' column) that a process has reserved. However, the amount of virtual memory reserved does not reflect the actual amount of memory needed or used. For example, the Java Virtual Machine (JVM) on the test system always reserves about 32 GiB of (virtual) memory. The test system had sufficient physical memory available during all test runs; and it is thus unlikely that any memory that was actually used got swapped out of memory. Therefore, the 'RES' column is very likely to show the actual total memory usage.

The measurements were taken as precise as practically possible, but some variables could not be controlled during the tests. For example the execution of other processes (the test system is multi-user) and how these are divided over the CPU cores might influence runtimes. The processor's clock rate can even

vary ('Intel Turbo Boost Technology') based on variables such as the number of used CPU cores and the CPU temperature.

Furthermore even with all other variables constant, there can be significant difference in the amount of work needed for different test scenarios of equal length. Repeating the tests will thus not lead to the exact same measurements.

However despite the uncertainties and imperfections in the tests, the general trends in the collected data are very clear. When data from a single test run is presented, care has been taken to select a representative example.

10.3 Performance: Execution Time

10.3.1 Complete Test Setup

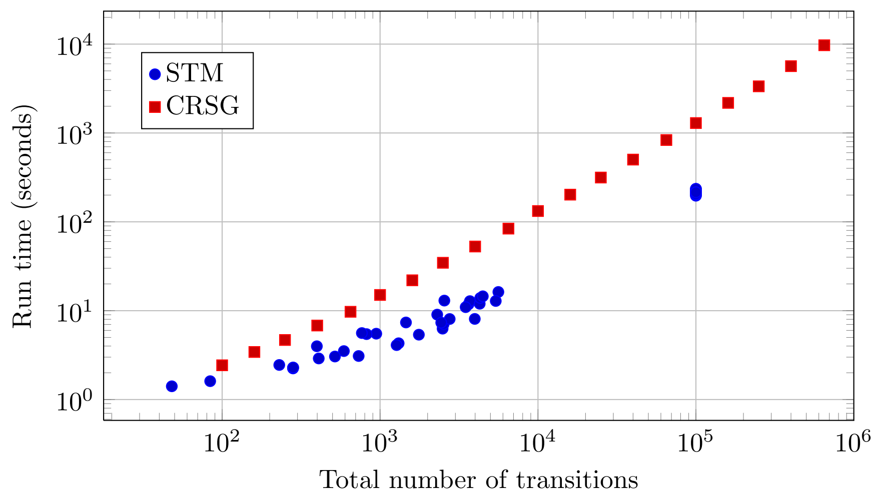


Figure 10.1: Execution time of the complete test setup

Figure 10.1 shows the execution times of the conducted test runs graphed against the number of transitions in the test runs. The results with the STM test purpose model and the CRSG test purpose model are differentiated by their different markers and colours.

Most test runs with the STM test purpose did not run until the end because a failure was observed, as anticipated. Three of the test runs with the CRSG model initially also did not run until the end because a failure was observed (not shown in the graph). These three runs were restarted in order to get data for the full spectrum of input lengths as shown in the graph.

The graph shows that test runs with the STM test purpose model execute in less time than test runs of the same length with the CRSG test purpose model. The reason for this performance difference is further discussed in section 10.3.2, where the performance of the individual components are regarded.

For both test purpose models, the execution times seem roughly linear with the number of transitions (notice that both axes are logarithmic). The same data is re-plotted in figure 10.2 as the average transition rate per second during

a run against the number of transitions in that run (notice that the vertical axis is non-logarithmic now).

This second plot shows that the test runs with the CRSG test purpose model scale almost linearly. There is a very small decline in performance after 250,000 transitions, and a small start-up delay of about a second that affects the average transition rates of the shortest test runs, but overall the graph shows a very clear roughly linear trend.

The STM test purpose model is another story. Clearly test runs with the STM test purpose model are much more efficient than with the CRSG test purpose model. The rate of transitions per second also increases significantly with longer test runs. But the data also shows a lot more variance; in some cases a test run can take twice as long as another test run of the same length. Furthermore, it is also not clear from the test data how the performance scales for long test runs, whether the performance plateaus out or keeps increasing.

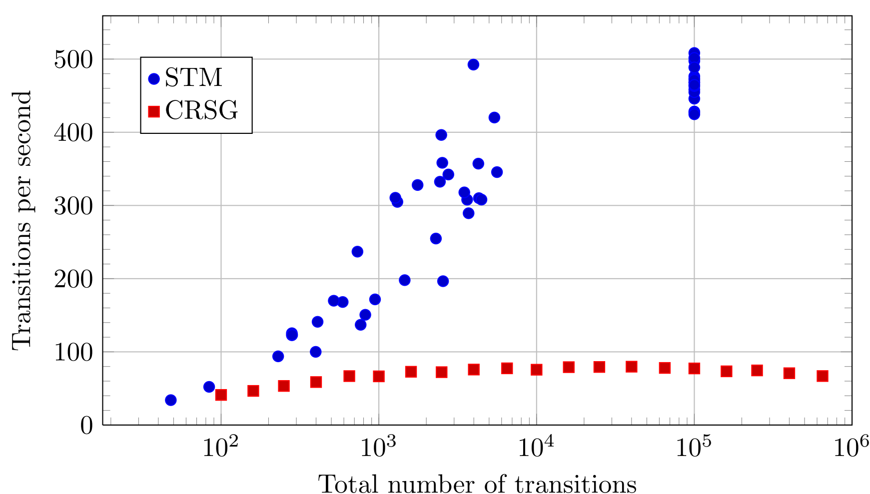


Figure 10.2: Execution speed of the complete test setup

10.3.2 Individual Components

Figure 10.3 shows the used CPU time of the individual components during a single test run with the CRSG test purpose model. The shown test run is of the longest run with 650,000 transitions that took about 162 minutes.

Notice that the y axis shows the CPU time (which adds time of multiple threads) whereas the x axis shows the actually elapsed time. Because of this, the impact of both JTorX and the test purpose model (which have (implicit) multi-threading) are overstated by the graph in comparison with lps2torx and the SUT (which are both single threaded). The total CPU utilisation of the whole test setup during the graphed test run was 123%, implying that JTorX and the test purpose model added significantly less time to the total run time than the graph suggests.

Very clear is that lps2torx dominates the execution time, and that the SUT has used almost no CPU time. In fact at the end of the test run lps2torx has taken up over 8500 seconds of CPU time, and the SUT only just over 9 seconds.

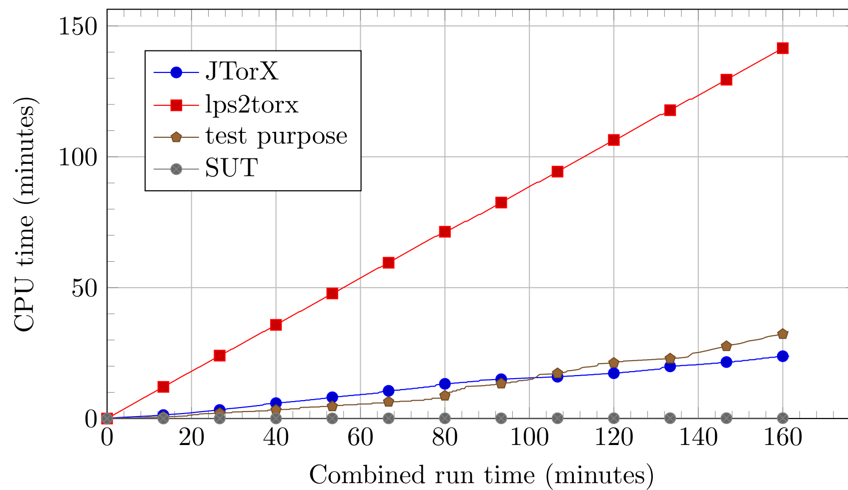


Figure 10.3: Execution times of individual components using the CRSG test purpose model

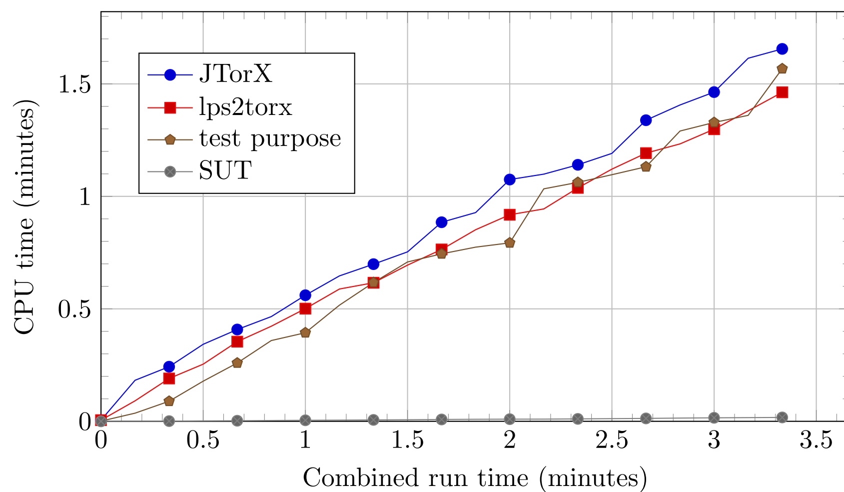


Figure 10.4: Execution times of individual components using the STM test purpose model

Figure 10.4 shows the used CPU time of the individual components during a single test run with the STM test purpose model. The shown test run is of one of the test runs that finished without observing a failure and took 3:29 minutes (however because of the 10 second measurement interval the last data points in the graph are at 3:20 minutes).

Lps2torx does not dominate the CPU time like it does with the CRSG test purpose model; in fact, in the selected test run both JTorX and the test purpose model use slightly more CPU time than lps2torx (although, they added less to the overall run time, because part of this CPU time was consumed by parallel threads).

The SUT uses the least CPU time again by far: just over 1 second in total.

The great difference in CPU time usage of lps2torx between the CRSG and the STM test purpose model mostly explains the differences in overall run times for these test setups. The reason that lps2torx can interpret the PLC-Interlocking model so much faster with the STM test purpose model is probably related to the fact that the STM test purpose model restricts the possible state space. With the STM test purpose model, inputs to the specification model will often be the same as earlier communicated inputs (e.g. if the simulated trains do not move and the EBP requests don't change). This means that the model will often be in a state that has already been visited (especially since timing is not part of the state of the PLC-Interlocking model). With the CRSG test purpose model, the chance of the model coming in a previously visited state is much smaller. Lps2torx keeps all previously expanded states in memory, thus it will not have to compute a state when that state has already been visited, thus saving time.

Revisiting past states also has a slight positive effect on the run time of JTorX (probably also because of state caching). In the test run with the STM test purpose model shown in the graph, JTorX used 1:39 minutes of CPU time. In the 100,000 transition test run with the CRSG test purpose model, this was 2:37 minutes. Do notice that these measurements are not very precise (because of the 10 second measurement interval); there is also quite some variance between similar measurements. Nonetheless, JTorX consistently used less CPU time with the STM test purpose.

In all test runs, the CPU time used by the SUT is only a fraction of the total run time. In the test run graphed in figure 10.3 lps2torx uses over 900 times as much CPU time as the SUT (8528 seconds versus 9.15 seconds). Lps2torx has some overhead because it has to keep track of all expanded states, but the model that it interprets only implements a part of the logic of the SUT, and the SUT's logic even gets executed twice for every cycle in the model. So, the model and its interpretation by lps2torx are very inefficient compared to the actual system that is modelled. In fact the overhead of the total test setup in that run was over a factor 1000 (i.e. if the test setup used 0 additional time, then a 1000 test runs could have been done in the same time span). With the STM test purpose model this overhead is less, but still a factor 200.

The SUT executable is also much faster than an actual PLC-Interlocking installation. In the 650,000 transitions long test run with the CRSG test purpose model, the SUT executed 45,666 cycles (note that these are SUT scan cycles, the model did only half that amount). The SUT required 9.15 seconds to execute these cycles, so its execution rate was almost 5000 cycles per second. In the test run with the STM test purpose model graphed in figure 10.4, the execution rate was even higher. Meanwhile, the real PLC-Interlocking installation at Santpoort Noord only manages an average of 36 scan cycles per second. So the SUT executable is two orders of magnitude faster than the real PLC-Interlocking. This difference in execution speed can easily be explained by the additional integrity checks, inter-module communication overhead and input/output overhead on the PLC platform.

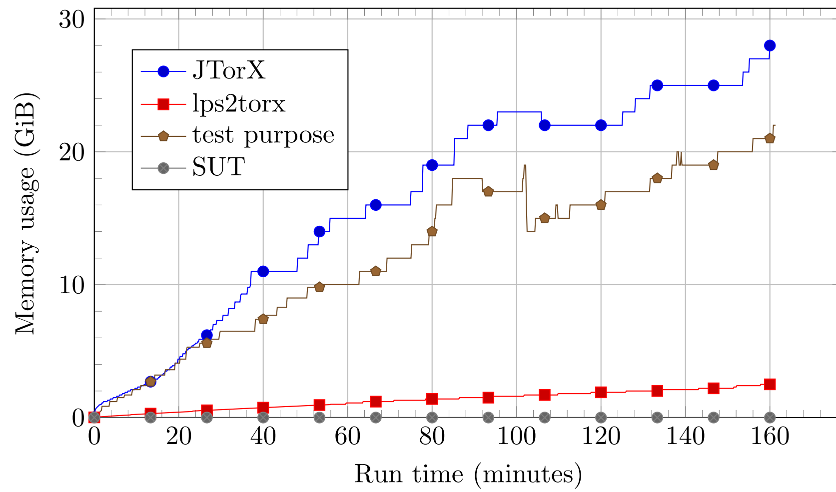


Figure 10.5: Memory usage of individual components during a test run with the CRSG test purpose model

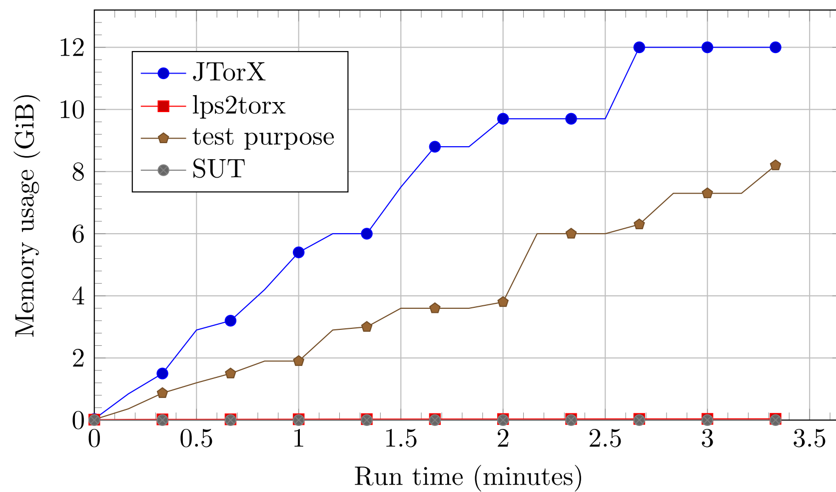


Figure 10.6: Memory usage of individual components during a test run with the STM test purpose model

10.4 Performance: Memory Usage

Memory usage has only been measured for the individual components of the test setup. Figure 10.5 shows the memory usage during the 650,000 transition test run conducted with the CRSG test purpose model. Figure 10.6 shows the memory usage of a 100,000 transition test run with the STM test purpose model (the graphed run is the same as shown in figure 10.4).

Both JTorX and the test purpose model use significant amounts of memory during test runs, regardless of the used test purpose model. For both processes the graphs show stepwise increases and decreases in memory usage. This is most likely caused by the use of standard Java data structures that are backed by

arrays. These data structures increase their capacity in steps (often doubling) by reallocating a new array, after which the garbage collector can reclaim the previously used array. It should be noted that the reported memory usage might be higher than the amount of memory that the Java processes really need, because Java's garbage collection does not free unused memory directly.

The memory usage of lps2torx is also significant (2.5 GiB after 650,000 transitions with the CRSG test purpose), but much less than the memory usage of JTorX and the test purpose model.

The memory usage of the SUT is constant through all measurements at just over 1 megabyte. This is of course as expected, the FBD language in which the logic is expressed is extremely simple and does not need dynamic memory allocation or recursive calls in its translated C++ form.

The memory usage of JTorX and the test purpose model are such that even relatively short (in terms of time) test runs require more memory than present in contemporary desktop systems. For example the 3:29 minute test run graphed in figure 10.6 used over 20 GiB of memory, which is considerable more than found in most desktop computer system (a state of affairs that will no doubt change in the near future).

On the other hand, computation servers with hundreds of gigabytes in memory are already not that uncommon. Furthermore, the performance hit from swapping on systems without enough memory might not be too severe, because the test setup might predominantly access memory objects that are recently created and have thus not been swapped out. However, no benchmarks have been run on less equipped systems, so this remains speculation.

Even though the excessive memory usage does not pose a direct problem because of access to a well-equipped test system, it seems unnecessary. Some changes to the used tools might severely decrease the test setup's memory consumption.

10.4.1 Inefficiencies in the Test Setup

JTorX and the test purpose model together are responsible for the biggest part of the memory consumption during a test run. It appears that JTorX accumulates a lot of data during a test run which it then keeps in memory. Exactly what it keeps in memory is unclear, but it is known that it keeps information on every state along the explored path. For the test purpose model it is very clear what it keeps in memory: Its memory consumption is caused virtually entirely by the data structure that holds all states that have been discovered during a test run.

Most of the memory consumed by the test purpose model is not used for anything useful. The main problem is that JTorX never instructs the torx-explorers to discard old states (which is possible through the delete command of the torx-explorer protocol). In principle, with the used setup, there is no need for JTorX or the test purpose model to retain any state other than the very last explored state. Discarding all states but the current state would completely solve the memory usage problem, however this is not implemented in the current version of JTorX. Note that an important consideration when implementing this, is that the computational performance of lps2torx actually profits from storing

previously visited states (the used version of lps2torx also does not support the delete state command); the test purpose model however gains no speed advantage from storing old states since it never revisits a state.

The problems caused by the fact that JTorX never discards old states are compounded by two smaller problems with the test purpose models that by themselves would not have a big impact on the test purpose's performance:

1. JTorX pre-emptively expands states of the test purpose model that will never be visited.

2. The test purpose framework computes and stores states when they are first communicated instead of when they are expanded.

Analyses of log files showed that JTorX pre-emptively explores all output transitions (i.e. the transitions that model outputs of the SUT), before the SUT has decided which one of the possible transitions to take. There is a rational behind this: JTorX tries to find a short path to an end state of the test purpose model. However that rational does not apply in this particular case for two reasons: The path to take at that point is decided by the SUT's outputs (JTorX has no choice), and the used test purpose model has no end states. As a result the test purpose model visits a lot of states that the SUT never visits. In the conducted test runs, the amount of expanded transitions of the test purpose model was typically between 3 and 5 times as high as strictly necessary. JTorX's author has indicated that future JTorX versions will have an option to disable this behaviour [49].

The second problem listed above is in the implementation of the test purpose framework: It computes every successor state that is communicated to JTorX, not just those that are expanded (which would save considerable memory). This problem was already discussed in more detail in section 9.4.

Solving the two smaller problems would also result in a speed up of the test setup. However, this speed up would not be very dramatic, since neither the test purpose model nor JTorX dominates the execution time of the complete test setup. Implementing functionality in JTorX to discard older states on the other hand, could reduce the memory requirements of (at least) the test purpose model to almost nil.

10.5 Ability to Find Faults

Of course the ultimate goal of doing tests is to find faults in the SUT; or to gain confidence in the correctness of the SUT when no faults are found. Crucial for this is the ability of the test setup to find faults.

Using test purpose models it is possible to run certain fixed scenarios and use the model as an oracle that predicts the correct output. This does not offer many advantages over classical manual testing approaches. More interesting is using test purpose models to dynamically generate test scenarios. The question is, are these generated test scenarios useful for finding faults?

In total 36 failures were observed in 73 automated test runs. However, observing a failure does not necessarily indicate a fault in the SUT. The fault may also lie in the model's implementation, or even in the test setup itself (e.g. in interfacing code). In fact most of the observed failures were easily linked to one of a few defects in the model.

10.5.1 Types of Errors Found

All observed failures concern the yellow or better outputs, i.e. in all cases the SUT outputted an `output!true` transition where the model expected an `output!false` transition. There are no failures observed that concern the specific aspects outputted by the SUT (the `aspectOutput` transitions).

All observed failures have been analysed to determine whether the failure was caused by a fault in the interlocking logic or by a fault in the model or other parts of the test setup. Beforehand was known that the model contained some flaws (see section 8.3.2). A number of the observed failures could directly be attributed to these flaws. All other failures were analysed with the help of a senior ProRail engineer.

All failures that have been observed can be classified in one of the following categories:

- **Type 1 failures**: Failures that are caused by the first fault in the model described in section 8.3.2.
- **Type 2 failures**: Failures that are caused by the second fault in the model described in section 8.3.2.
- **Type 3 failures**: Failures that are caused by faults in the way that the model tracks trains and how it decides when a route is free.
- **Type 4 failures**: Failures that do not represent a dangerous situation, but cannot be traced to a fault in the model without further analyses.
- **Unknown failures**: Failures that must be further analysed to determine whether there is a dangerous situation, and what causes the failure.

Unfortunately due to time constraints not all faults could be analysed thoroughly enough to pin point the exact cause of the failure. At the moment it is not clear for failures in the categories 'type 4' and 'unknown' where the fault lies, whether it is in the model, the rest of the SUT or the specification (or possibly for 'unknown': in the SUT).

10.5.2 Constrained Random Scenario Generator

Of the initial 20 runs, three runs ended with the observation of a failure. These runs were re-executed to get additional performance measures. No failures were observed during these retries.

The originally scheduled batch of 20 runs, was supposed to have 1,733,160 transitions in total. With the interrupted test runs, in total 1,993,067 transitions were executed. These transitions amounted to 69,929 simulated PLC scan cycles.

These 69,929 scan cycles contained 279,692 `output!Bool` transitions (some test runs finished before completing the output phase of the scan cycle). Of those output transitions only 28 were `output!true`, all other were `output!false`. Three of those 28 true outputs were failures according to the model.

Table 10.1 shows the observed failures, in which run they occurred (target number of transitions), at which transition, and what caused the failure observation. It should be noted that the type 1 failure listed, also constituted a type 2 failure. That is even if the model did not contain the type 1 fault, a failure would still have been observed because of the type 2 failure.

Table 10.1: Observed failures with the CRSG test purpose

Test Run	At Transition	Type
160000	100,202	Type 2
250000	14,372	Type 1
650000	145,333	Type 4

10.5.3 Simulated Train Movements

The STM test purpose model was used for 50 test runs of each 100,000 transitions maximum. No faults were found in 17 of these test runs; the other 33 terminated early after the observation of a failure.

All test runs in which a failure was observed terminated quickly. The longest test run in which a failure was observed was 5612 transitions long, the average length of such runs was just over 2000 transitions. That is, in all test runs with the STM test purpose model, either a fault was found quickly, or no fault was found at all.

The conducted test runs contain 1,766,131 transitions total (of which 1,700,000 transitions in the 17 failure-less runs). Of these transitions, 386,256 are yellow or better output transitions (i.e. `output!Bool`). Of these `output` transitions, 1850 are `output!true` transitions. These `output!true` are all within the first 15,000 transitions of each test run. That is, all signals are set to red in the last 85,000 transition of the 17 test runs that last 100,000 transitions. It seems that in all test runs, after a number of transitions the system comes in a state in which no signal can be set to yellow or better anymore.

Table 10.2 shows the failures that were observed during the test runs. The first column contains the sequence number of the test run (0 to 49), the second and third column again contain the transition at which the failure was observed and the kind of failure.

10.6 Finding Faults: Evaluation

A lot of test runs ended with the observation of a failure concerning an `output!true` transition. On the other hand, no failures related to `aspectOutput!Nat` transitions were observed. This is not very surprising since the algorithm for computing a signal aspect is very simple (given whether the signal is yellow or better and the aspect of the next signal) and well documented. Determining whether

Table 10.2: Observed failures with the STM test purpose

Test Run	At Transition	Type
0	946	Type 2
3	409	Type 2
6	3630	Type 2
7	2489	Type 2
8	281	Unknown
10	3979	Type 2
11	84	Type 2
12	3707	Type 3
13	732	Type 2
14	5612	Type 3
19	2526	Type 4
20	4275	Type 3
22	398	Type 2
23	590	Type 3
24	2553	Type 2
26	821	Type 2
27	2759	Type 2
29	2301	Type 2
31	281	Unknown
32	518	Type 2
34	767	Type 3
35	1311	Type 2
36	230	Type 2
37	4309	Type 3
40	1273	Type 2
41	2440	Type 2
43	3484	Type 3
44	1761	Unknown
45	282	Type 2
46	1459	Type 3
47	5403	Type 2
48	48	Type 2
49	4473	Unknown

a signal should be yellow or better on the other hand is much harder, involves a lot more requirements and requires a lot more code.

Given that yellow or better related failures can only be observed if the SUT outputs `output!true` transitions, it is crucial that the test purpose models generate test cases where such transitions occur. The test cases generated with the CRSG test purpose model only contained 28 such transitions (in almost 2 million transitions total). Of the 23 performed test runs, 14 had not a single `output!true` transition in them.

The STM test purpose model generated much more viable test cases: The test cases generated with this test purpose model contained 1850 `output!true` transitions. That being said, if no failure was observed in these test cases,

then after a while this test purpose model brings the SUT in a state in which `output!true` are no longer outputted. In three of the generated test cases, the SUT even came in such a state before any `output!true` transitions were outputted. As a result of this, most of the executed transitions were useless. A possible solution, which has not been further investigated, is to introduce a sequence of transitions in the test purpose model that resets the SUT's state after a while. Such a sequence of transitions is certainly possible, but might require some inputs that are not available in the current model. Another solution would be to simply limit test cases to, for example, 10,000 transitions.

Table 10.3: Observed failure types per test purpose model

		Observed Failures				
Test Purpose	Total Runs	Type 1	Type 2	Type 3	Type 4	Unknown
Fixed scenario 1	1	1	0	0	0	0
Fixed scenario 2	1	0	1	0	0	0
CRSG	23	1	1	0	1	0
STM	50	0	20	8	1	4

Table 10.3 summarises the data from tables 10.1 and 10.2, and the results from the test runs with the fixed scenario test purpose models. Note that both observations of a type 1 error were also observations of type 2 errors, but are only counted as a type 1 error in the table.

Even though the STM test purpose model generates test cases with a lot of 'interesting behaviour', it failed to generate a test case that led to the observation of a type 1 failure. Of course the number of test runs could be expanded to increase the chance of finding all types of faults. Furthermore a model with less other faults might significantly increase the chances of finding a type 1 fault for the STM test purpose model. Nevertheless, it is a pity that the fault is not found consistently with the STM test purpose model, because it is easy to see how such a fault could occur in a real interlocking system by accident: A section is simply assumed to be on the wrong side of a signal.

Unlike the STM test purpose, the CRSG test purpose did generate a test case that observed a type 1 failure. Of course the fixed scenario also found this fault. So even though the STM test purpose models seems to have a better chance of finding faults than the CRSG test purpose model, it might not find all faults just as well as other test purpose models.

As said before, a lot of failures were observed in often relative short test runs. Analysis showed that the faults causing these failures were in the model, not the SUT (for as far as the failures could be classified).

It is important to note that classifying failures is a lot of work, even with the help of a tool to visualise the behaviour in the generated log files. As such, the current PLC-Interlocking specification model is not very useful for testing PLC-Interlocking systems, because it yields a lot of false positives. Determining that these false positives are indeed false positives involves a lot of work, and ultimately does not increase the confidence in the correctness of the system, it merely shows that the model contains a lot of faults.

Chapter 11

Conclusions and Future Work

This chapter lists the conclusion of the research described in this report, followed by an overview of possible directions for future research.

11.1 Conclusions

This report presents a model based testing methodology for testing the interlocking logic of PLC-Interlocking systems. The presented method uses the JTorX test tool, which incorporates the ioco testing theory. The PLC-Interlocking system that is in operation at the Santpoort Noord railway station has been used as a case study for the application of the method throughout this report.

The test setup consists, besides the test tool JTorX, of three components:

1. The SUT.
2. A model that specifies correct behaviour for the SUT.
3. A test purpose model.

Each of these items took considerable research effort. In the following the items are reflected upon independently before the complete test setup is considered. Finally the main conclusions regarding the presented test methodology are drawn in section 11.1.5.

11.1.1 The SUT

A method has been devised to create an executable from the interlocking logic that can be run on a regular PC. First, the actual interlocking code has to be captured in C++ form during the regular compilation process. The thus acquired interlocking logic is then made part of an application that adds interfaces to interact with the logic, and library functions to make the acquired logic functional.

The regular process of creating a binary (that can only run on a PLC) is SIL-4 certified. The process used to get an executable SUT for the test setup is

a custom process, and offers no such official ‘guarantee’ of correctness. However, the solution is nonetheless very satisfying, given that there are no manufacturer supported alternatives and the created SUT opens up the possibility of (automated) testing using just a regular PC. Furthermore, the resulting product has very good performance: It runs two orders of magnitude faster on the test system than a program with the same logic on an actual PLC system.

11.1.2 Model

A (partial) model of the interlocking logic has been created in mCRL2. The developed model is tailored to the Santpoort Noord PLC-Interlocking installation, but could easily be adapted to another installation.

Constructing a model of the correct behaviour of an interlocking system had not been attempted before (in the known literature). Past research efforts verified interlocking systems based on simple properties to which the systems must adhere or fixed test scenarios, but not using a model of the supposed behaviour of the system.

The model is, besides its limited functionality, imperfect and contains many flaws. This is not surprising, beforehand it was clear that creating a complete and correct model would not be possible in the allotted time. The development of the model was hindered by imprecise specifications and lack of railway signalling knowledge. Creating a complete and correct model would require a lot of railway signalling knowledge and time.

11.1.3 Test Purpose Model

Multiple test purpose models have been implemented with different test case generation strategies. The STM test purpose model simulates the movement of trains in the interlocking's environment and deduce inputs for the SUT from that. The CRSG test purpose generates inputs randomly, with a restriction on the generated values to be consistent.

The test purpose models have been implemented in Java. Using Java instead of mCRL2 resulted in clearer and more concise code. Furthermore it aided the development process because the Java development tools are superior to the available mCRL2 tools.

Java is a more powerful language than mCRL2, but cannot be made to generate an LTS as easy as mCRL2, and had never been used for such models before. A custom framework was created that abstracts away all details involved in communicating inputs to JTorX for the created test purpose models. The created framework is not necessarily limited to PLC-Interlocking test purpose models, but could be used for PLC models in general. With small adaptations it can be used to construct any deterministic cyclic model with a single computation phase per cycle.

11.1.4 The Complete Test Setup

The conducted research has led to a functioning test setup for (a part of) the Santpoort Noord PLC-Interlocking installation.

Two factors are important for the usefulness of the test setup in testing the Santpoort Noord PLC-Interlocking: the performance of the test setup, and the quality of the test setup. A third factor is important for testing PLC-Interlocking installations in general: the effort needed to adapt the test setup to another PLC-Interlocking system.

Adaptability

The test setup was created during a master thesis project that took 11 man-months to complete. These 11 man-months do include a lot of prelimenary research and a lot of time spent on writing this thesis, but nonetheless creating the test setup took a considerable amount of time. The method would not be practical if creating a test setup would take the same amount of time every time.

To create a test setup for another PLC-Interlocking installation each of the three main components (SUT, specification model and test purpose model) need to be adapted, and some minor configuration details for the overall test setup must be changed. Each of these components has a very clear separation between generic and installation specific code (see sections 6.5, 7.4 and 9.6). As a result of this, the effort needed to adapt each component is very limited compared to the initial effort.

As it stands the test setup cannot be used out-of-the-box by a railway signalling engineer. But with knowledge about the setup it is already possible to quickly create a similar test setup for another PLC-Interlocking installation (perhaps in less than a week, although that is speculation).

Performance

The performance of the test setup is adequate, in the sense that its performance is absolutely no impediment to its use. The test setup does consume a lot of memory; for the longer runs even considerable more than a typical PC has available. However, even if one wants to do longer test runs this does not have to be a problem: Computer systems with enough memory are not rare nor prohibitively expensive. That being said, there are many improvements possible to the performance of the test setup, especially with regard to memory consumption.

Quality

The quality of the test setup is more of a concern than its performance. During the conducted test runs, many failures have been observed. Four of these failures must be analysed further, but as it stands now it seems that all failures are a result of faults in the used model.

Failures classified as type 1 failures are of extra interest because the SUT's behaviour is an exception to regular behaviour (but allowed because of external circumstances). Exceptions to regular behaviour normally are caused by faults in the SUT, and as such should be found. All other classified faults, thus not including the four unclassified faults, are clearly faults in the model, and not of interest.

The usefulness of the test purpose models for finding faults in the SUT (if there are any) is hard to predict. The STM test purpose model has proven to generate test cases that will often lead to the observation of a failure. And with further research, the test purpose model could be improved to find faults more consistently. However, the STM test purpose model has not produced a test case that led to the observation of a type 1 fault, whereas the SQRM has produced such a test case.

The biggest problem quality-wise with the test setup is the number of faults in the model and the frequency with which these cause failures. Considering that the model only models a part of the SUT's behaviour, yet generates so many false positives, it must be concluded that the quality of the whole test setup is not at (or close to) a level where it can be used in the regular validation process of interlocking systems.

11.1.5 Main Conclusions

The main conclusions regarding the presented test methodology:

- The quality of the whole test setup is not at (or close to) a level where it can be used in the regular validation process of interlocking systems. The main problem is the lack of a correct specification model.
- The application of a model based testing approach was considerably less successful than in other case studies that employed model based testing (e.g. [30, 31]). The main reason for this is the complexity of the railway signalling domain, and the resulting problems in creating a correct specification model.

11.2 Future Work

Whether the method on itself will be useful mainly depends on the quality of the used model. Therefore, future research into ioco testing of PLC-Interlocking systems (or any other type of electronic interlocking system) should focus on the model.

Creating a complete and correct model is complicated and will require considerable resources. It is a task that cannot be fulfilled by people that merely have computer science backgrounds; deep knowledge of both railway signalling in general and the PLC-Interlocking system in particular is required too.

Such research, if undertaken, should not necessarily extend the mCRL2 model described in this report. Other modelling languages (e.g. Java) or model architectures must also be considered, because it is far from clear whether mCRL2 is the ideal language for such a model, and alternative modelling language and model structures have not been explored.

So, in order to improve the test setup, the focus should firstly be on the used model. Only after that can the usefulness of the whole test setup (and its other components) be determined.

That being said, there are also interesting research possibilities with regard to the used tools that can be done independently from any research on the

specification model. There are relatively small projects (1 and 2) as well as a possibly complicated project (3):

1. Improving the performance of the test setup:
   - Improve JTorX by only keeping recent states in memory, and instructing the test purpose model to delete older states.
   - Improve the test purpose model by not computing states until they are expanded by JTorX.
2. Developing analysis tools. Analysing error traces without good visual aids and debugging tools can be very time consuming.
3. Developing a better mCRL2 tool chain: Look into the possibilities to compile models to native code or a byte code format, instead of the now used LPS format which is interpreted. This might bring great performance improvements (faster and more predictable), both at run time and execution time.

Glossary

Notation	Description	Page List
ATB	Automatic train protection (Dutch: Automatische Trein Beïnvloeding)	15
CRSG	Constrained Random Scenario Generator test purpose model. One of the created test purpose models, further described in section 8.3.1.	4, 91
DO document	High level document containing general information about the design and planned construction of a specific interlocking installation. (Dutch: Definitief Ontwerp)	4, 46
EBP	Protocol used between logistics systems and interlocking systems (Dutch: Elektronische BedienPost)	24
ERTMS	European Rail Traffic Management System	16
ETCS	European Train Control System	16
FBD	Function Block Diagram	4, 24
ioco	input-output conformance	3, 33
JTorX	An automated test tool that can test whether an implementation ioco-conforms to a specification. This tool is presented in [3].	3, 34
JVM	Java Virtual Machine	113
LPS	Linear Process Specification	37
LTS	Labelled Transition System	3, 33
MDS	Maintenance and Diagnostics Subsystem	23
OBE-blad	Technical map detailing the track layout and the positions of the wayside elements in an area. (Dutch: Overzicht Baan en Emplacement blad)	4, 46

Notation	Description	Page List
OPC	Standardized protocol for communication with PLC systems (originally: OLE for Process Control)	23
OS-blad	Technical map detailing the allowed aspects of signals, as well as the relations between the aspects of different signals. (Dutch: Overzicht Seinbeelden blad)	4, 15
OVS	Design instructions (Dutch: OntwerpVoorSchrift)	19
OVS Application Engineering	OVS document prescribing how the PLC system that is at the core of a PLC-Interlocking should be programmed and configured. Document: [5]	19
PLC	Programmable Logic Controller	6, 21
point	Assembly of rails, blades and of auxiliaries, certain ones being movable, which effect the tangential branching of tracks and allows to run over either one track or another (description from: [35]). See also section 2.2.2.	13
railway yard	In general an area with multiple railway tracks. In this report it is used to refer to the area of control of a single interlocking system.	17
section	A portion of track which the interlocking system can recognise by means of a train detection system (description based on: [35]). See also section 2.2.3.	17
signal	Apparatus by means of which a conventional visual indication is given, generally concerning the movements of railway vehicles (description based on: [35]). See also section 2.2.1.	13
signaller	The person responsible for the operation of the signalling system in accordance with the requirements of the railway operating rules and regulations (description from: [35]).	14
SIL-4	Safety Integrity Level 4	17
SILworX	Application used to program and configure HIMA PLC systems.	25
speed step variables	A set of PLC-Interlocking input variables that are received from a neighbouring interlocking system. The variables indicate at what speed a train may cross the border to the neighbouring area, if at all.	49
SSS	Subsystem/System Specification	4, 46
STM	Simple Train Movements test purpose model. One of the created test purpose models, further described in section 8.3.3.	4, 94
SUT	System Under Test	33
torx-explorer	Either: 1) A program that communicates transitions with JTorX via a set protocol over the standard input and output streams. Or 2) The protocol used for the communication between such a program and JTorX.	37

Notation	Description	Page List
VPI	Vital Processor Interlocking	6, 19
wayside element	Used in this report to refer to objects in the physical layer of the train control architecture (see figure 2.1). Examples: points, signals and train detection devices.	1, 13

Bibliography

[1] T. ten Hoeve, “Verification of PLC based interlocking systems.” Preliminary research report, Unpublished, University of Twente, May 2012.

[2] M. Timmer, H. Brinksma, and M. I. A. Stoelinga, “Model-based testing,” in *Software and Systems Safety: Specification and Verification* (M. Broy, C. Leuxner, and C. A. R. Hoare, eds.), vol. 30 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pp. 1–32, Amsterdam: IOS Press, April 2011.

[3] A. Belinfante, “JTorX: A tool for on-line model-driven test derivation and execution,” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 266–270, 2010.

[4] ProRail, *System/Subsystem Specification*, 1.1 ed., June 2012. Not publicly available.

[5] ProRail, *Ontwerpvoorschrift Eurolocking Application Engineering*, 000.15 ed., May 2012. Not publicly available.

[6] ProRail signalling experts, 2012. Personal communications.

[7] Movares signalling experts, 2012. Personal communications.

[8] Movares and ARCADIS, “OBE-blad 2 Santpoort Noord,” 2012. Not publicly available.

[9] Movares and ARCADIS, “OS-blad 1 and 2 Santpoort Noord,” 2009. Not publicly available.

[10] Movares, “Definitief ontwerp pilot 1:1 venti fase 3,” August 2011. Not publicly available.

[11] J. Groote, S. van Vlijmen, and J. Koorn, “The safety guaranteeing system at station hoorn-kersenboogerd,” *Logic Group Preprint Series*, 1994.

[12] J. Groote, S. van Vlijmen, and J. Koorn, “Safety guaranteeing system at station hoorn-kersenboogerd (extended abstract),” in *The 10 th Annual Conference on Computer Assurance*, pp. 57–68, 1995.

[13] W. Fokkink, "Safety criteria for the vital processor interlocking at hoorn-kersenboogerd," in *Proceedings 5th Conference on Computers in Railways–COMPRAIL*, vol. 96, pp. 101–110, 1996.

[14] A. Borälv, "Case study: Formal verification of a computerized railway interlocking," *Formal Aspects of Computing*, vol. 10, no. 4, pp. 338–360, 1998.

[15] C. Eisner, "Using symbolic model checking to verify the railway stations of hoorn-kersenboogerd and heerhugowaard," *Correct Hardware Design and Verification Methods*, pp. 99–109, 1999.

[16] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi, "Model checking interlocking control tables," *FORMS/FORMAT 2010*, pp. 107–115, 2011.

[17] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model verifier," in *Computer Aided Verification*, pp. 682–682, Springer, 1999.

[18] G. Holzmann, "The model checker SPIN," *Software Engineering, IEEE Transactions on*, vol. 23, no. 5, pp. 279–295, 1997.

[19] K. Claessen, N. Een, M. Sheeran, and N. Sorensson, "SAT-solving in practice," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 61–67, IEEE, 2008.

[20] K. Winter, "Model checking railway interlocking systems," *Australian Computer Science Communications*, vol. 24, no. 1, pp. 303–310, 2002.

[21] K. Winter and N. Robinson, "Modelling large railway interlockings and model checking small ones," in *Proceedings of the 26th Australasian computer science conference-Volume 16*, pp. 309–316, Australian Computer Society, Inc., 2003.

[22] A. Mirabadi and M. Yazdi, "Automatic generation and verification of railway interlocking control tables using FSM and NuSMV," *Transport Problems: an International Scientific Journal*, vol. 4, no. 103–110, 2009.

[23] F. Moller, H. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne, "Railway modelling in CSP||B: the double junction case study," in *Proceedings of the 12th International Workshop on Automated Verification of Critical System (AVoCS 2012)*, EASST, 2012. to appear.

[24] V. Gourcuff, O. De Smet, and J. Faure, "Efficient representation for formal verification of PLC programs," in *Discrete Event Systems, 2006 8th International Workshop on*, pp. 182–187, Ieee, 2006.

[25] O. Pavlovic and H. Ehrich, "Model checking PLC software written in function block diagram," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 439–448, IEEE, 2010.

[26] G. J. Tretmans and H. Brinksma, "Torx: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany* (A. Hartman and K. Dussa-Ziegler, eds.), pp. 31–43, December 2003.

[27] C. Jard and T. Jéron, "TGV: theory, principles and algorithms," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 4, pp. 297–315, 2005.

[28] A. Hartman and K. Nagin, "The AGEDIS tools for model based testing," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 129–132, ACM, 2004.

[29] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," *Formal methods and testing*, pp. 39–76, 2008.

[30] M. Sijtema, M. Stoelinga, A. Belinfante, and L. Marinelli, "Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at Neopost," *Formal Methods for Industrial Critical Systems*, pp. 117–133, 2011.

[31] J. Meijer, "Model based system testing in practice." Presentation at the FMT group, Twente University, June 2012.

[32] I. Mutlu, T. Ovatman, M. Soylemez, and L. Sumer, "A new test environment for PLC based interlocking systems," in *Transportation, Mechanical, and Electrical Engineering (TMEE), 2011 International Conference on*, pp. 686–690, IEEE, 2011.

[33] J. Calame, N. Goga, N. Ioustinova, and J. van de Pol, "Ttcn-3 testing of hoorn-kersenboogerd railway interlocking," in *Electrical and Computer Engineering, 2006. CCECE'06. Canadian Conference on*, pp. 620–623, IEEE, 2006.

[34] J. Calame, 2012. Personal communications.

[35] Florian Lesné (UIC), *Unified glossary of Terms*. INESS, SB_Finalised ed., May 2009. Available at: `http://www.iness.eu/IMG/pdf/INESS_WS_D_Deliverable_D.1.1_SB_Finalised_Report_Ver2009-05-29.pdf`.

[36] Railinfra Opleidingen, *Introductie Railinfra Structuur*, November 2002. course manual.

[37] NS Opleidingen, Rail Infra, *Beheersing, sturing, en beveiliging treinverkeer*, December 2000. course manual.

[38] G. Theeg and S. Vlasenko, *Railway Signalling & Interlocking - International Compedium*.

[39] H. Sulmann, *Van sein tot sein*. ProRail, January 2010.

[40] ProRail signalling experts, 2011. Personal communications.

[41] HIMA Paul Hildebrandt GmbH + Co KG, *HIMax System Manual*, 4.01 ed., 2011.

[42] HIMA Paul Hildebrandt GmbH + Co KG, *SILworX Help*, v1.05 ed., May 2011. Ships as a part of the HIMA SILworX application.

[43] ProRail, *Ontwerpvoorschrift Eurolocking Hardware Engineering*, 0.12 ed., March 2012. Not publicly available.

[44] ProRail engineers, 2012. Personal communications.

[45] Movares engineers, 2012. Personal communications.

[46] I. E. C. (IEC), "Programmable controllers - part 3: Programming languages," 2003.

[47] J. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. Van Weerdenburg, "The formal specification language mCRL2," *Methods for Modelling Software Systems (MMOSS)*, vol. 6351, 2007.

[48] Technische Universiteit Eindhoven, *mCRL2 user manual*, 2011. Available at: `http://www.mcrl2.org/release/user_manual/user.html`.

[49] A. Belinfante, 2012. Personal communications.

[50] University Twente, *TorX Manual Pages: torx-explorer(5) - interface to program to explore a labelled transition*, 3.9.0 ed., June 2006. Available at: `http://fmt.ewi.utwente.nl/tools/torx/torx-explorer.5.html`.