Formal Development and Verification of Railway Control Systems - In the context of ERTMS/ETCS Level 2

Vu, Linh Hong; Haxthausen, Anne Elisabeth

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Ph.D. Thesis
Doctor of Philosophy

DTU Compute
Department of Applied Mathematics and Computer Science

Formal Development and Verification of Railway Control Systems

In the context of ERTMS/ETCS Level 2

Linh Hong Vu

Kongens Lyngby 2015

RobustRailS

The work presented in this dissertation is
part of the RobustRailS project funded by
by *Innovation Fund Denmark*.
www.robustrails.man.dtu.dk

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Richard Petersens Plads
Building 324
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk
PHD-2015-395
ISSN: 0909-3192

Summary

(In English)

This dissertation presents a holistic, formal method for efficient modelling and verification of safety-critical railway control systems that have product line characteristics, i.e., each individual system is constructed by instantiating common generic applications with concrete configuration data. The proposed method is based on a combination of formal methods and domain-specific approaches. While formal methods offer mathematically rigorous specification, verification and validation, domain-specific approaches encapsulate the use of formal methods with familiar concepts and notions of the domain, hence making the method easy for the railway engineers to use. Furthermore, the method features a 4-step verification and validation approach that can be integrated naturally into different phases of the software development process. This 4-step approach identifies possible errors in generic applications or configuration data as early as possible in the software development cycle, and facilitates debugging/troubleshooting if errors are discovered. The proposed method has successfully been applied to case studies of the forthcoming Danish railway interlocking systems that are compatible with the European standardized railway control systems ERTMS/ETCS Level 2. Experiments showed that the method can be used for specification, verification and validation of systems of industrial size.

Resumé

(På Dansk)

Denne afhandling præsenterer en holistisk, formel metode til effektiv modellering og verifikation af sikkerhedskritiske jernbanestyresystemer, der har produktlinje-egenskaber, dvs hvert enkelt system konstrueres ved at instantiere fælles generiske applikationer med konkrete konfigurationsdata. Den foreslåede metode er baseret på en kombination af formelle metoder og domæne-specifikke metoder. Mens de formelle metoder tilbyder matematisk stringent specifikation, verifikation og validering, indkapsler de domæne-specifikke metoder brugen af de formelle metoder med velkendte begreber og notationer for det givne domæne, og gør dermed metoden let at bruge for jernbaneingeniører. Metoden tilbyder en 4-trins verifikations-og valideringsproces, der kan integreres naturligt i de forskellige faser af software-udvikling. Denne 4-trins proces identificerer eventuelle fejl i generiske applikationer og konfigurationsdata så tidligt som muligt i softwareudviklingsprocessen, og faciliterer debugging/fejlfinding. Den foreslåede metode har med succes været anvendt i casestudier af de kommende danske jernbanesikringsanlæg, der er kompatible med det europæiske standardiserede jernbanestyresystem, ERTMS/ETCS niveau 2. Forsøg har vist, at fremgangsmåden kan anvendes til specifikation, verifikation og validering af systemer af industriel størrelse.

To my family.

Acknowledgements

"Every party must come to an end and when it does, there are kisses and promises and waves goodbye." – LIFE Magazine

The last three years have been one of the hardest journeys of my life. Luckily, I didn't have to walk it alone: my two supervisors, colleagues, friends, and family have always been there for me through all the ups and downs.

Anne (Haxthausen): I am grateful to you for your guidance and help with everything from the beginning to the end of my study! The lessons that I learned from you through all these years have equipped me well for the future.

Jan (Peleska): I admire you for your excellences in both theory and practice, and for being inspiring professionally and personally. Thank you for your invaluable suggestions, and for your hospitality during my visits in Bremen!

Jan (Bertelsen), *Nikhil* (Mohan Pande), and *Ross* (Edwin Gammon): thank you for sharing your immense expertise about Danish interlocking systems with me, and for being always helpful when I had questions!

I am thankful to my colleagues and partners in RobustRailS project for their valuable inputs and discussions.

Florian (Lapschies) and *Uwe* (Schulze): I am grateful to you for your precious help with the implementation of the method in RT-Tester.

Birgit (Michaelis), *Blagoy* (Genov), *Cécile* (Braunstein), *Cornelia* (Zahlten), *Christoph* (Hilken), *Elena* (Gorbachuk), *Felix* (Hübner), and *Wen-ling* (Huang): many thanks to you for making it feel like home when I was in Bremen.

Andreas (Foldager), *Jacob* (Hansen), *Kim* (Sørensen), and *Peter* (Østergaard): thanks for the interesting discussions! It has been fun working with you.

Janne (Lassen) and *Hanne* (Jensen): thanks for your help with all administrative procedures! If it hasn't been for you, it would have taken me long time to get around.

Bahram (Zarrin), *Dilshan* (Makavitage), *Jóhan* (Davidsen), and *Vlad* (Acretoaie): thanks for your accompany and discussions about everything! I have been enjoying having you guys as office mates.

Anh-Dung (Phan), *Nhut* (Nguyen), *Huynh* (Luong), *Trung* (Trinh), *Thang* (Pham), *The* (Ngo), *Nong* (Ngo), *Hiep* (Nguyen), *Hung* (Tran), and their families; *Ninh* (Pham), *Hong* (Phan), *Ha* (Nguyen), *Hang* (Cao), *Tuan* (Nguyen), and *Hoa* (Le): I am grateful to you for recharging me with laughters, good time, good food, and always reminding me to enjoy every moment in the long journey.

To friends and colleagues that I couldn't list all here: thanks for being there when I need you.

A special thank is dedicated to my friend, my companion, my ally, and my sweetheart Soňa. You made the journey an adventure.

Finally, I would have never come this far, if it has not been for the unconditional love, trust, and support from my family.

Preface

This dissertation was prepared at the department of Applied Mathematics and Computer Science (DTU Compute) at the Technical University of Denmark in partial fulfilment of the requirements for acquiring a degree of Doctor of Philosophy (PhD).

The work presented in this dissertation deals with formal development and verification of railway control systems. The primary focus is to develop a holistic method and an associated toolchain to facilitate the efficient development of safe railway interlocking systems that are compatible with European Train Control System (ETCS) Level 2.

The dissertation summarises all the studies conducted during the period 2012-2015. Some of the work has been presented in publications published during the PhD study.

The work presented in this dissertation is part of the work package WP.4.1 of the RobustRailS project, which is funded by Innovation Fund Denmark.

Kongens Lyngby, October 31, 2015

Linh Hong Vu

Contents

List of Papers

The dissertation is based on the following publications that have been submitted/published as part of the studies during 2012–2015.

[VHP14a] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "A Domain-Specific Language for Railway Interlocking Systems". In: *FORMS/FORMAT 2014 - 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*. Edited by Eckehard Schnieder and Géza Tarnai. Best paper award. Institute for Traffic Safety and Automation Engineering, Technische Universität Braunschweig., 2014, pages 200–209. ISBN: 978-3-9816886-6-5

[VHP14b] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "Formal Verification of the Danish Interlocking Systems". In: *AVoCS 2014 Pre-proceedings*. University of Twente, 2014

[VHP15] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release". English. In: *Formal Techniques for Safety-Critical Systems*. Edited by Cyrille Artho and Peter Csaba Ölveczky. Volume 476. Communications in Computer and Information Science. Springer International Publishing, 2015, pages 223–238. ISBN: 978-3-319-17580-5. DOI: 10.1007/978-3-319-17581-2_15. URL: http://dx.doi.org/10.1007/978-3-319-17581-2_15

[VHPss] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release". In: *Science of Computer Programming - Special Issue: Formal Techniques for Safety-Critical Systems* (Under minor revision process)

Additionally, the following papers have been published as a result of the research collaboration with University of Bremen on testing of ETCS Level 2 onboard modules. This research provides also background for the work on model-based testing of the Danish interlocking systems.

[Bra+14b] Cécile Braunstein et al. "Complete Model-Based Equivalence Class Testing for the ETCS Ceiling Speed Monitor". In: *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*. Edited by Stephan Merz and Jun Pang. Volume 8829. Lecture Notes in Computer Science. Springer, 2014, pages 380–395. ISBN: 978-3-319-11736-2. DOI: 10.1007/978-3-319-11737-9_25. URL: http://dx.doi.org/10.1007/978-3-319-11737-9_25

[Bra+14a] Cécile Braunstein et al. *A SysML Test Model and Test Suite for the ETCS Ceiling Speed Monitor*. Technical report. Embedded Systems Testing Benchmarks Site, 2014. URL: `http://www.mbt-benchmarks.org`

Acronyms

ATP Automatic Train Protection.

BDD Binary Decision Diagram.

BMC Bounded Model Checking.

BNF Backus–Naur Form.

BTM Balise Transmission Module.

CBTC Communication Based Train Control.

CEGAR CounterExample-Guided Abstraction Refinement.

COI Cone of Influence.

CSS Cascading Style Sheets.

DMI Driver Machine Interface.

DSL Domain-specific Language.

EC European Commission.

EDL Early Deployment Line.

EMF Eclipse Modelling Framework.

ERTMS European Rail Traffic Management System.

ETCS European Train Control System.

EURIS European Railway Interlocking Specification.

EVC European Vital Computer.

GMF Graphical Modelling Framework.

GPL General-purpose Language.

GSM Global System for Mobile Communications.

GSM-R GSM-Railways.

IC3 Incremental Construction of Inductive Clauses for Indubitable Correctness.

ICL Interlocking Configuration Language.

IDL Interlocking Dynamic Language.

IOSTS Input/Output State Transition System.

ITG Interlocking Table Generator.

LLD Ladder Logic Diagrams.

LTE Long Term Evolution.

LTL Linear Temporal Logic.

MA Movement Authority.

MBT Model-based Testing.

OBU On-board Unit.

OCL Object Constraint Language.

PDF Portable Document Format.

RAISE Rigorous Approach to Industrial Software Engineering.

RBC Radio Block Center.

RCSD Railway Control Systems Domain Language.

RobustRailS Robustness in Railway OperationS.

RSL RAISE Specification Language.

SMT Satisfiability Modulo Theories.

STS State Transition System.

SUT System Under Test.

TMS Traffic Management System.

V&V Verification and Validation.

XML eXtensible Markup Language.

Tag cloud of this dissertation generated† by Wordle™*

*https://en.wikipedia.org/wiki/Tag_cloud

†http://www.wordle.net/

CHAPTER 1

Introduction

This chapter gives an introduction to the context of the work presented in this dissertation, namely, the Danish Signalling Programme, a program that replaces the entire railway signalling in Denmark; and the RobustRailS project, a research project accompanying the Signalling Programme on a scientific level. Motivated by the challenges in verification and validation of the new railway control systems that are going to be deployed in Denmark in the Signalling Programme, our goal in this work is to provide methods and tools supporting *efficient* development and verification and railway control systems. With a primary focus on railway control systems with product line characteristics, this dissertation shows how the above goal is fulfilled by the main contributions of the work: a holistic method for verification and validation of railway control systems with product line characteristics, and its application to the forthcoming Danish interlocking systems.

The remainder of this chapter is organised as follows. First, Section 1.1 and Section 1.2 introduce the Danish Signalling Programme and the RobustRailS project, respectively. Afterwards, the goals, scope, and the main contributions of the work presented in this dissertation are described in Section 1.3. Section 1.4 concludes the chapter with an outline of the dissertation.

1.1 The Danish Signalling Programme

In 2009, the Danish government decided to invest in a total renewal of the Danish railway signalling systems [Ban10] by 2021 in the Danish Signalling Programme*. The program aims at replacing the entire Danish railway signalling systems, which are coming to the end of their lives, by the European standardized signalling systems – European Rail Traffic Management System (ERTMS)/European Train Control System (ETCS) Level 2 [ERT14]. It is the first time a renewal at this scale has been attempted, with an estimated investment of EUR 2.5 billion. This accounts for a full ERTMS/ETCS Level 2 implementation including all onboard and trackside equipments on the Fjernbane – the long-ranged regional and intercity

*http://www.bane.dk/signalprogrammet

railway, a full Communication Based Train Control (CBTC) on the local system S-bane, all interlocking systems, country-wide traffic management systems, GMS-R, interface management, safety approvals, design, testing, implementation, training, and changes in the internal processes of Banedanmark – the Danish railway infrastructure owner [Ban10; Ban14].

The Danish Signalling Programme is expected to bring multiple benefits for customers, railway infrastructure owners, and railway operators [Ban10; Ban14]:

- Better punctuality, increased line speed, and higher capacity. It is expected that an 80 percent decline in signal-related delays on main and regional lines and 50 percent on the S-bane as a result of the Signalling Programme.
- Higher and more homogeneous level of safety.
- Economical maintenance in the future.
- Better centralised traffic control, better energy optimisation, and better passenger information.
- Double the number of rail passengers by 2030, target a greener public transport system.

The Signalling Programme is currently in progress, with some parts of the new systems in the early deployment phase.

1.2 RobustRailS Project

Robustness in Railway OperationS (RobustRailS)[†] is an interdisciplinary project funded by Innovation Fund Denmark[‡] for the period 2012-2016. The project attempts to answer the question: *Can we get trains to run on time?*. Accompanying the Danish Signalling Programme mentioned in Section 1.1, the RobustRailS project aims at providing extra confidence on the systems that are about to be deployed. Furthermore, the project investigates also sustainable rail transport for the future where robustness is integrated in the planning, development, and operations of railway systems. This means that signalling systems and communications need to be rethought and redesigned with robustness in mind. Moreover, models shall be developed to assist the analysis and validation of the effect of robust rail operations.

The RobustRailS project is partitioned into multiple work packages, each work package studies a perspective of a robust railway system, from robust infrastructure to robust operations or passenger flow. The work presented in this dissertation is part of the deliverables of work package WP.4.1 entitled "Formal Development and Verification of Railway Control Systems". The work package focuses on how domain-specific languages and formal development and verification methods can be used to facilitate the efficient manufacture of robust and safe railway control systems.

[†] `http://robustrails.man.dtu.dk`
[‡] `http://innovationsfonden.dk`

1.3 Goals, Scope, and Contributions of the Thesis

Conventionally, the verification and validation process of railway control systems is informal and mostly manual, hence time-consuming, costly, and error-prone. Thus, automated verification and validation of railway control systems is an active research topic, investigated by several research groups, see e.g., [HBK10; Fer+10; Win12; Jam+14; HPP14].

Goals. As part of the RobustRailS research project, our goal is to establish methods and tools for efficient development, verification and validation of safety-critical railway control systems. The method should be formal and facilitate automation in order to provide a better verification process compared to the conventional one.

Our *hypothesis* is that a proper combination of formal methods and domain-specific approaches leads to efficient development and verification of systems with product line characteristics. Namely,

- Formal methods offer rigorous specification, verification, and validation, while domain-specific approaches encapsulate the use of formal methods with familiar concepts and notions of the domain, hence making formal methods more accessible to end-users.
- The use of reconfigurable, generic artefacts such as generic models and properties would increase the reusability and efficiency of the formal modelling and verification process.
- A combination of bounded model checking and inductive reasoning would be used to address the challenge of applying formal verification on systems of industrial size.

Scope. The primary focus of the thesis is railway control systems that are safety-critical, and have *product line* characteristics: each individual system is constructed by instantiating common generic applications with concrete configuration data. *Railway interlocking systems* are perfect examples of such systems, hence they are chosen as the case study for our proposed method.

Contributions. The main contributions of work presented in this dissertation are:

(1) *A holistic, formal method* for development of safety-critical systems that have product line characteristics. The method is built on a combination of formal methods and domain-specific approaches. The method has the following characteristics.

   - *Holistic.* The method covers different phases of the development process, from specification, over design, to verification and validation.

- *Formal.* The specification, verification and validation in the method are mathematically rigorous. Thus, the method provides a higher level of confidence compared to conventional manual and informal methods.
- The method offers a *4-step verification and validation approach*, allowing bugs to be revealed as early as possible in the development cycle.
- *User-friendly.* Using domain-specific languages to encapsulate the use of formal methods, the method provides different levels of abstraction that are suitable for different user groups: railway engineers, software engineers, or validators.
- *Scalable.* The method is capable of verifying the safety properties of systems of industrial size by using a combination of bounded model checking and inductive reasoning.
- The method includes a *model-based, domain-specific testing strategy* which verifies the conformance of the implementation with the design and requirements.

(2) *Application* of the method to the forthcoming Danish interlocking systems that will be deployed in the Danish Signalling Programme.

The subsequent chapters of this dissertation will elaborate these contributions in detail.

1.4 Structure of the Thesis

The remaining chapters of the dissertation are organised as follows.

- **Chapter 2** gives a brief introduction to different background topics that are relevant to the work presented in this dissertation including the revolutionary railway standard ERTMS/ETCS, railway interlocking systems, and in particular the forthcoming Danish interlocking systems that will be deployed by the Signalling Programme. The chapter also gives a short summary about formal methods and domain-specific languages. Some mathematical preliminaries are also presented.

- **Chapter 3** presents an overview of the proposed method and its application to railway interlocking systems. The subsequent chapters, Chapter 4 to Chapter 7, elaborate the method in detail.

- **Chapter 4** presents Interlocking Configuration Language (ICL) – a domain-specific language for describing interlocking configuration data. The chapter describes also how the wellformedness of configuration data can be automatically checked.

- **Chapter 5** presents the syntax and semantics of Interlocking Dynamic Language (IDL) – a domain-specific language for specifying generic interlocking models.

- **Chapter 6** presents a generic model of the forthcoming Danish interlocking systems and a verification technique that is able of verifying the safety properties for models of realistic size. Furthermore, a comparison with other verification techniques is also presented.
- **Chapter 7** describes a model-based, domain-specific testing strategy tailored for interlocking systems.
- **Chapter 8** concludes the dissertation and presents ideas and suggestions for future work.

Remarks on Formalisms. Throughout this dissertation, different formalisms are used in order to give the clearest presentation. RAISE Specification Language (RSL) [Gro92; Geo04] – the specification language of Rigorous Approach to Industrial Software Engineering (RAISE) development method [Geo+95; Geo04] – is used to specify both the abstract syntax and static semantics of ICL in Chapter 4 because RSL supports algebraic data types and executable specifications. Whereas Backus–Naur Form (BNF) is used to specify the concrete syntax for IDL in Chapter 5. Note that the concrete syntax of IDL is presented in Chapter 5 instead of abstract syntax like the way ICL is presented in Chapter 4 because IDL is used to specify the generic model, safety properties, and test objectives of the Danish interlocking systems in Chapter 6 and Chapter 7. Denotational semantics of ICL and IDL are given in Chapter 5 for readability.

Related Work. Other work related to content of a chapter is presented in the respective chapter.

Prerequisites. Readers are assumed to be familiar with formal specification, formal verification, and domain-specific languages. Knowledge about railway domain is not mandatory for understanding the content of this dissertation.

Remarks on Hyperlinks. In the electronic version – i.e., Portable Document Format (PDF) – of this dissertation, most of the abbreviations, citations, inline macro name listings, and cross references are *hyperlinks*: they are linked to their corresponding definitions, sources, full specifications, and content, respectively. One can follow (click) the links to reach the corresponding content within the dissertation.

CHAPTER 2

Background

This chapter describes briefly some background knowledge that is relevant to the work presented in this dissertation. First, an introduction to the standardised ERTMS/ETCS, railway interlocking systems, and their relation are given in Section 2.1 to Section 2.4. Afterwards, the forthcoming Danish interlocking systems are described in detail in Section 2.5. Product line characteristics of interlocking systems are introduced in Section 2.6. Section 2.7 to Section 2.10 introduce shortly formal methods, model checking, and model-based testing, and domain-specific languages, respectively. Finally, Section 2.11 presents some mathematical preliminaries, including Kripke structures, k-induction, and input/output state transitions systems, that will be used throughout this dissertation.

Note that this chapter does not aim to give an exhaustive study about these topics, but rather a brief introduction that is adequate for understanding the work presented in this dissertation. Readers are advised to consult the respective references for more thorough studies.

2.1 ERTMS/ETCS

The *European Rail Traffic Management System (ERTMS)* is an initiative implemented by the European Commission (EC) in order to increase *cross-border*[*] *interoperability* and *safety* of rail traffic across Europe. ERTMS is the first international standard for train command-control and train-to-ground communication systems. ERTMS consists of two complementary subsystems:

(A) GSM-Railways (GSM-R)

(B) European Train Control System (ETCS)

ERTMS is not technologically advanced in terms of its subsystems: the technologies for the subsystems have been there for years, some technologies are even going to be obsolete, e.g., GSM-R. However, ERTMS is revolutionary in the sense that it pieces existing technologies together to create a comprehensive solution for interoperability and safety in railway. Therefore, EC obliges European railways to deploy ERTMS for their new railway systems via European Council Directive 96/48/EC [Eur96] and European Commission Decision 2001/260/EC [Eur01]. Due to its advantages, ERTMS has grown out of Europe and become a global standard for railways: many countries outside Europe have started implementing ERTMS for their railway systems [Sys15].

GSM-R. *GSM-Railways (GSM-R)* provides data channels for train-to-ground communication by adapting the well-known commercial wireless communication standard Global System for Mobile Communications (GSM) to railway applications. Although GSM-R is mandated by ERTMS standard, it has been shown that GSM-R has a number of shortcomings such as limited technical support in future, low capacity, and many others [Sni15]. In order to overcome such shortcomings, newer wireless communication standards have been investigated as alternatives for GSM-R in the future. In the dissertation for another work package within RobustRailS project, Sniady has studied the possibility of using Long Term Evolution (LTE) – the latest commercial telecommunication standard, also known as 4G – as an alternative for GSM-R and strategies for migrating to LTE from GSM-R [Sni15].

ETCS. *European Train Control System (ETCS)* is a state-of-the-art railway signalling, train control, and train protection system that enhances the *interoperability, safety*, and *capacity* of rail traffic. Based on the communication provided by GSM-R, ETCS manages and supervises train movements. Two primary features provided by ETCS are (1) in-cab signalling, and (2) Automatic Train Protection (ATP) [TVA09].

(1) *In-cab signalling*: In traditional railway signalling systems, classical colour light signals – which are physically installed along the tracks in a given railway

[*] The term *border* in *cross-border* should be understood generally as either a border between countries, or a border between two different railway signalling and train control systems.

network – allow or disallow trains to move forward at a certain speed. In ETCS[†], these colour signals are replaced by the Driver Machine Interface (DMI) inside the driver cabin. The DMI displays all commands from the control centre and other information to the driver. This is referred as *in-cab signalling*. In-cab signalling reduces the risk of human errors made by train drivers. First, it reduces the risk that the driver may read a wrong signal, miss a signal, or misinterpret the meaning of a signal. Second, the DMI provides drivers much more precise, detailed, and frequent information, which is great support for train drivers in driving the train.

(2) *Automatic Train Protection (ATP)*: ATP is a system that supervises the train driver on a train. The ATP calculates a *braking curve* [ERT14] describing the safe upper limit of the train speed at a certain location based on the current speed profile, the characteristics of the train, and the current movement authority granted to the train. The actual speed and position of the train are then compared with this braking curve in order to ensure that the train driver obeys all the signalling rules. The ATP will take immediate actions to prevent hazardous situations from happening. For example, the ATP will trigger the emergency brakes if the train travels too fast compared to the allowed speed. Additionally, the braking curve can serve as a guideline for the train driver to have an optimal and smooth driving. An example of such monitoring is the ceiling speed monitoring module (CSM) in ETCS onboard computer. The CSM has been modelled and analysed in [Bra+14b].

ETCS is specified at five different *application levels*: 0, NTC, 1, 2, and 3 [ERT14]. These levels differ in terms of efficiency, safety, and investment cost: the level 0 is the least autonomous level where ETCS-fitted trains – i.e., trains equipped with ETCS onboard equipments – operate in non-fitted trackside systems, while the level 3 is the most autonomous level where movement authorities and the position of trains are monitored and communicated via radio communication network. Railway infrastructure owners can choose an appropriate level based on their specific requirements and strategies. ETCS Level 2 will be deployed in Denmark by the Danish Signalling Programme as mentioned in Section 1.1. Therefore, the research presented in this dissertation considers ETCS Level 2. Throughout the remaining of this dissertation, ETCS refers to ETCS Level 2 if not indicated otherwise.

2.2 ETCS Level 2 Architecture

In this section, the architecture of ETCS Level 2 will be introduced briefly. Note that the introduction here is merely for understanding the context of the work presented in this dissertation, hence only a simplified architecture is presented. Figure 2.1 shows a simplified schematic architecture of ETCS Level 2. ETCS is divided into two general subsystems: onboard and trackside.

[†]at an appropriate application level, e.g., level 1 to 3

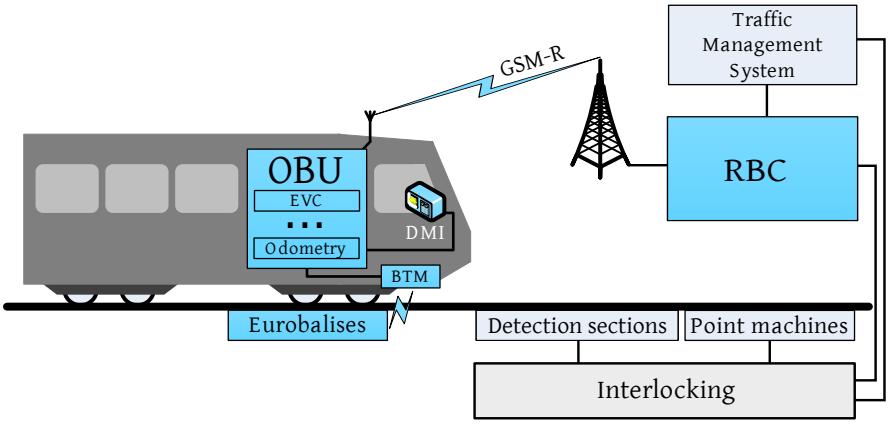


Figure 2.1: ETCS Level 2 schematic architecture

Onboard Subsystem. The onboard subsystem, also known as *On-board Unit (OBU)* in ETCS terminology, is a set of ETCS elements installed on a train. A train equipped with an ETCS onboard subsystem is referred as an *ETCS-fitted* train. Some of the elements of the OBU are listed in the following.

- *European Vital Computer (EVC)*: The EVC is the brain of an OBU. It is responsible for all the computation and logic of the system onboard the train. The EVC has interfaces with the train and other modules in the OBU. It orchestrates all the operations in the onboard subsystem.
- *Driver Machine Interface (DMI)*: The DMI is the interface between the train driver and the OBU. The DMI displays all necessary information to the driver, e.g., the current speed of the train or the current speed limit. The DMI also serves as an input device for the driver to enter information for setting up the OBU.
- *GSM-R* module provides the communication interface to the Radio Block Center (RBC) in the trackside subsystem.
- *Balise Transmission Module (BTM)*: The BTM reads the information stored in Eurobalises placed along the track as the train passes them. This information is essential for the OBU to compute the braking curves.
- *Odometry*: The odometry system estimates the current speed of the train and its position in the relation to the last reference location specified by the last Eurobalises that the train has passed.

Trackside Subsystem. The trackside subsystem consists of elements that are installed along the railway tracks. Some of the elements are listed in the following.

- *Eurobalises*: Eurobalises are reference beacons installed along the tracks. Each Eurobalise contains the information about its precise location and track characteristics such as gradient or friction.

- *Interlocking*: Interlocking systems are responsible for guiding trains safely through the network. Interlocking systems are explained more in Section 2.3.

- *Traffic Management System (TMS)*: The TMS is a centralised system that manages the overall traffic and interfaces with signalmen, timetabling systems, and traffic information systems.

- *Radio Block Center (RBC)*: The RBC manages trains running within the railway network under its control. It uses the information provided by the traffic management system, the OBU on the trains within the railway network under control, and the interlocking to calculate for each train Movement Authority (MA) dictating how far forward the train can go. RBC is also responsible for voice communication and data services.

A Typical Operation Scenario. In the following, a simplified version of a typical operation scenario in ETCS Level 2 is given in order to illustrate how different systems in ETCS Level 2 work together.

(1) The OBU of a train sends an MA request via GSM-R to the current RBC that the train is registered to. The MA asks for authorization to move further.

(2) The RBC, after receiving the MA request, consults the TMS about the timetable for the train based on the identifier registered for the train in order to figure out which route(s) shall be set (i.e., reserved) for the train.

(3) The RBC requests the interlocking to set the route(s) for the train as specified in the timetable.

(4) The interlocking checks the status of track sections, points and set them to appropriate states for the train to pass through safely. Afterwards, the interlocking notifies the RBC that the routes have been set.

(5) The RBC calculates the new MA and sends it to the train. The new MA dictates how far ahead the train can go, and the maximum speed allowed.

(6) The OBU, when receiving the new MA, calculates the new braking curve based on the new MA and information (e.g., location, track characteristics) it obtained from the Eurobalises. Then, the new information is displayed to the driver via the DMI. The new braking curve is also fed to the ATP to safe-guard the train.

2.3 Railway Interlocking Systems

A *railway interlocking system* (abbreviated as interlocking) is responsible for guiding trains safely through a given railway network. It is a vital part of any railway signalling system and has the highest safety integrity level (SIL4) according to the CENELEC 50128 standard [CEN12]. An interlocking system monitors the trackside elements of the railway network under its control, and set them to appropriate configuration so that trains can travel through the network without derailing or colliding to others [TVA09].

Interlockings are categorised into the following types based on their locking technologies [TVA09].

(1) *Mechanical interlockings* are the earliest type of interlockings. Their locking mechanism is purely mechanical using levers to operate different trackside elements to desired configuration.

(2) *Relay interlockings* consist of complex electrical circuits made up from electrical *relays*[‡] (electrically operated switches) arranged according to a certain interlocking logic. This interlocking logic ensures that once the trackside elements are locked in a certain configuration, all others conflicting configurations which may lead to collisions or derailments cannot be locked at the same time.

(3) *Electronic interlockings* (also known as solid state interlockings or computerised interlockings) are the latest type of interlockings where the locking logic is implemented by software rather than hard-wired circuits. Since electronic interlockings are developed after relay interlockings, the software implementation of locking mechanism often imitates the hard-wired circuits. Electronic interlockings provide also extra functionalities compared to other types of interlockings due to the resources available in software implementation.

There are three primary approaches toward interlocking implementation:

(a) *Route-based.* In a route-based interlocking, the railway network is divided into *fixed* fractions called routes [TVA09; Hax14]. A single centralised interlocking logic (either hard-wired by cables or implemented by software) controls the elements in the railway network layout and reserve routes for trains.

(b) *Communication-based.* Similarly to route-based approach, in a communication-based interlocking [EE04; PE09], there exists a centralised interlocking control logic. However, the railway network is not divided into fixed fractions. Instead, trains maintain constant communication with the interlocking logic and report their positions on the rail tracks. The interlocking logic ensures that every train under its control has an appropriate safe distance with other trains in front, or behind it. In other words, the interlocking logic maintains for every train under its control a *safety envelope*, similar to air traffic control.

[‡] `http://en.wikipedia.org/wiki/relay`

(c) *Geographical-based.* In a geographical-based interlocking, e.g., see [BF05; Fan12a; HP00; HPP14], the interlocking logic is distributed to the elements in the network layout. In order to reserve a fraction of the network for a train, messages are passed between the related elements following a predefined protocol. The goal of this protocol is to reach a consensus agreement on reserving the network fraction.

2.4 Interlockings and ETCS Level 2

Interlockings are not specified in ETCS standard. In principles, any relay interlockings or electronic interlockings can operate with ETCS-fitted system provided that they are compatible with ETCS. Traditionally, interlockings use colour light signals to convey permissions to drive forward to trains. On the contrary, in ETCS Level 2, these permissions are conveyed by movement authorities communicated via an RBC and a radio network to trains. To this end, the concept of *virtual signal* – signals that play the same role as traditional colour signals, but do not physically exist – is employed in the interface between ETCS and interlockings. Virtual signal concept is further explained in Section 2.5. Throughout this dissertation, if not mentioned otherwise, an interlocking shall be understood as an interlocking that is compatible with ETCS Level 2.

2.5 The new Danish Interlocking Systems

In this section, we introduce briefly the new Danish interlocking systems – electronic and route-based interlocking systems according to the categories presented in Section 2.3 – and the domain terminology. First, Section 2.5.1 describes different components of a specification of an interlocking system which is compatible with ETCS Level 2. Then, Section 2.5.2 explains the interlocking principles and the strict procedure that interlocking systems employ to ensure safety. Last, Section 2.5.3 explains the sequential release feature in the new Danish interlocking systems.

2.5.1 Specification of Interlocking Systems

The specification of a given interlocking system consists of two main components: (1) a railway network, and (2) a corresponding interlocking table.

Railway Networks. A railway network in ETCS Level 2 consists of a number of trackside elements of different types[§]: (a) linear sections, (b) points, and (c) marker boards.

[§]Here we only show types that are relevant to the work presented in this dissertation. Furthermore, for simplicity, we *do not* consider level crossings, derailers, moveable bridges.

Figure 2.2 shows an example layout of a railway network having six linear sections (`b10,t10,t12,t14,t20,b14`), two points (`t11,t13`), and eight marker boards (`mb10..mb21`).

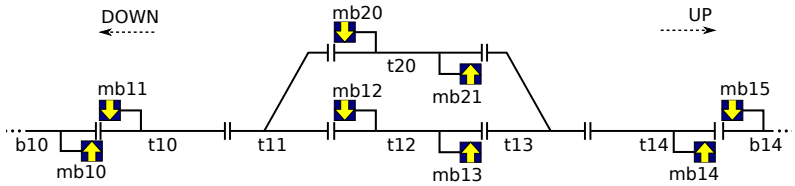


Figure 2.2: A railway network layout example

Different types of trackside elements are described in detail in the following.

(a) *Linear sections.* A *linear section* is a section (track segment) with up to two neighbours: one in the *up* end, and one in the *down* end. For example, the linear section `t12` in Figure 2.2 has `t13` and `t11` as neighbours at its up end and down end, respectively. In Danish railway's terminology, *up* and *down* denote the directions in which the distance from a reference location is *increasing* and *decreasing*, respectively. The reference location is the same for both up and down, e.g., an end of a railway line. For simplicity, in the examples and figures in the rest of this dissertation, the *up* (*down*) direction is assumed to be the left-to-right (right-to-left) direction, if it is not indicated otherwise.

(b) *Points.* A *point* can have up to three neighbours: one at the *stem*, one at the *plus* end, and one at the *minus* end, e.g., point `t11` in Figure 2.2 has `t10`, `t12`, and `t20` as neighbours at its stem, plus, and minus ends, respectively. The ends of a point are named so that the *stem* and *plus* ends form the straight (main) path through the point, and the *stem* and *minus* ends form the branching (siding) path through the point. A point can be switched between two positions: PLUS and MINUS[¶]. When a point is in the PLUS (MINUS) position, its *stem* end is connected to its *plus* (*minus*) end, thus traffic can run from its *stem* end to its *plus* (*minus*) end and vice versa. It is not possible for traffic to run from *plus* end to *minus* end and vice versa.

Linear sections and points are collectively called (train detection) *sections*, as they are provided with train detection equipments[‖] used by the interlocking system to detect the presence of trains on the sections. Note that sections are bidirectional by default, i.e., trains are allowed to travel in both directions (not at the same time) in a given section. For instance, trains can travel a linear section from its up end to its down end, and from its down end to its up end.

[¶]These terms are used in Denmark. In other countries, the terms NORMAL and REVERSE are used in place of PLUS and MINUS, respectively.

[‖]The equipments can be track circuits or axle counters, see [TVA09]. In case track circuits are used for train detection, sections are sometimes referred as track circuits.

In the scope of this dissertation, we do not distinguish between point machines – the mechanical/electrical machines that drive switching movements between positions of a point – and their associated detection sections. In practice, points are usually referred by the identifier of their point machines. These identifiers are different from the names of the associated detection sections. However, for simplicity, in our work, we refer to the points by the names of the detection sections that they lie on.

(c) *Marker boards.* Along each linear section, up to two *marker boards* (one for each direction) can be installed. A marker board can only be seen in one direction and is used as reference location (e.g., for the start and end of routes) for trains going in that direction. For example, in Figure 2.2, marker board `mb13` is installed along section `t12` for travel direction up.

Virtual Signals. As mentioned in Section 2.3, contrary to legacy systems, there are no physical signals in ETCS Level 2, but interlocking systems have a *virtual signal* associated with each marker board. Virtual signals play a similar role as physical signals in legacy systems: a virtual signal can be OPEN or CLOSED, respectively, allowing or disallowing traffic to pass the associated marker board. However, trains (more precisely train drivers) do not see the virtual signals, as opposed to physical signals. Instead, the aspect of virtual signals (OPEN or CLOSED) are communicated from the RBC to the onboard computer in the train via a radio network. For simplicity, the terms *virtual signals*, *signals*, and *marker boards* are used interchangeably throughout this dissertation.

Interlocking Tables. An interlocking system monitors constantly the status of track-side elements, and sets them to appropriate states in order to allow trains travelling safely through the railway network under control. The interlocking system grants a train the permission to drive on a fraction of the network layout, called a route, at a time.

A *route* is a path from a *source* signal to a *destination* signal (different from the source signal) in the given railway network. A route is called an *elementary route* if there are no signals that are located between its source signal and its destination signal, and that are intended for the same direction as the route. A *compound route* is a route created by concatenating multiple elementary routes so that permissions to drive on these elementary routes can be granted to a train at once.

In railway signalling terminology, *setting* a route denotes the process of allocating the resources – i.e., sections, points, signals – for the route, and then locking it exclusively for only one train when the resources are allocated. On the other hand, *releasing* a route denotes the process of releasing the resources that have been allocated for a route after they have been used by a train.

An *interlocking table* specifies the elementary routes in the given railway network and the conditions for setting these routes. The specification of a route r and conditions for setting r include the following information:

- $id(r)$ – the route's unique identifier,
- $src(r)$ – the source signal of r,
- $dst(r)$ – the destination signal r,
- $path(r)$ – the list of sections constituting r's path from $src(r)$ to $dst(r)$,
- $overlap(r)$ – a list of the sections in r's overlap, i.e., the buffer space after $dst(r)$ that would be used in case trains overshoot the route's path,
- $points(r)$ – a map from points** used by r to their required positions,
- $signals(r)$ – a set of protecting signals used for flank or front protection [TVA09] for the route, i.e., preventing other traffic from intervening with the traffic in the route, and
- $conflicts(r)$ – a set of conflicting routes which must not be set while r is set.

Table 2.3 shows an interlocking table example for the network shown in Figure 2.2. Each row of the table corresponds to a route specification. The column names are identical to the information of the route specifications that these columns contain. As an example, the first row in Table 2.3 specifies a route with id `1a`. The route goes from the source signal `mb10` to the destination signal `mb13` via three sections `t10`, `t11` and `t12` on its path, and has no overlap. It requires point `t11` (on its path) to be in PLUS position, and point `t13` (outside its path) to be in MINUS position (as a protecting point). The route has `mb11`, `mb12` and `mb20` as protecting signals, and it is in conflict with routes `1b`, `2a`, `2b`, `3`, `4`, `5a`, `5b`, `6b`, and `7`.

2.5.2 Interlocking Principles

In order to prevent hazardous situations, e.g., collision and derailment of trains, interlocking systems employ a classic principle:

A route is locked exclusively for use of one train at a time.

This is obtained by following a strict procedure for setting and releasing routes based on information in their interlocking tables. As an example, let us consider the following procedure for route `1a` specified in Table 2.3:

(0) Initially the route is *free*.

(1) The route is dispatched either manually by a signalman or automatically by a traffic management system. As a result, the route is *marked* as requested.

**These include points in the path and overlap, and points used for flank and front protection. For detail about flank and front protection, see [TVA09].

Table 2.3: An example of an interlocking table for the network layout in Figure 2.2. (p means PLUS, m means MINUS.)

id	src	dst	path	overlap	points	signals	conflicts
1a	mb10	mb13	t10;t11;t12		t11:p;t13:m	mb11;mb12;mb20	1b;2a;2b;3;4;5a;5b;6b;7
1b	mb10	mb13	t10;t11;t12		t11:p	mb11;mb12;mb15;mb20;mb21	1a;2a;2b;3;5a;5b;6a;6b;7;8
2a	mb10	mb21	t10;t11;t20		t11:m;t13:p	mb11;mb12;mb20	1a;1b;2b;3;5b;6a;6b;7;8
2b	mb10	mb21	t10;t11;t20		t11:m	mb11;mb12;mb13;mb15;mb20	1a;1b;2a;3;4;5a;5b;6a;6b;7
3	mb12	mb11	t11;t10		t11:p	mb10;mb20	1a;1b;2a;2b;5a;6b;7
4	mb13	mb14	t13;t14		t13:p	mb15;mb21	1a;2b;5a;5b;6a;6b;8
5a	mb15	mb12	t14;t13;t12		t11:m;t13:p	mb13;mb14;mb21	1a;1b;2b;3;4;5b;6a;6b;8
5b	mb15	mb12	t14;t13;t12		t13:p	mb10;mb13;mb14;mb20;mb21	1a;1b;2a;2b;4;5a;6a;6b;7;8
6a	mb15	mb20	t14;t13;t20		t11:p;t13:m	mb13;mb14;mb21	1b;2a;2b;4;5a;5b;6b;7;8
6b	mb15	mb20	t14;t13;t20		t13:m	mb10;mb12;mb13;mb14;mb21	1a;1b;2a;2b;3;4;5a;5b;6a;8
7	mb20	mb11	t11;t10		t11:m	mb10;mb12	1a;1b;2a;2b;3;5b;6a
8	mb21	mb14	t13;t14		t13:m	mb13;mb15	1b;2a;4;5a;5b;6a;6b

(2) The interlocking system checks the status of different track-side elements in the system to figure out whether it can start *allocating* resources for route `1a`, e.g., sections `t10`, `t11` and `t12` must be vacant, and the conflicting routes must not be allocated or locked. If so, the interlocking commands the protecting signals of the route – i.e., `mb11`, `mb12` and `mb20` – to change to CLOSED, and it commands points to switch to their required positions according to the route's specification – i.e., it commands `t11` to switch to PLUS, and `t13` to switch to MINUS.

(3) The interlocking system monitors constantly the status of the track-side elements. When the signals and points have changed their status as commanded in step (2), the route is *locked* and its source signal `mb10` is commanded to change to OPEN, allowing a train to enter the route.

(4) When the locked route is *occupied* – i.e., a train enters it, the source signal `mb10` is set to CLOSED preventing other trains from entering.

(5) The whole route is *released* (set back to *free*) when the train has finished using it – i.e., the train has passed `mb13`, or the train has come to a standstill in front of `mb13`.

(6) A route can be *cancelled* when it is in step (1), (2), or (3) if the route has not been occupied by a train. The interlocking will release the resources allocated for the route and set the route back to *free*.

2.5.3 Sequential Release

The new Danish interlocking systems employ *sequential release* (also known as sectional release) [TVA09, chap. 4]. This feature results in two major changes to the procedure explained above:

(a) With sequential release, the interlocking can release an element in a locked route as soon as the train has passed it, instead of waiting until the train has finished using the route and then releasing the route as a whole. Consequently, the capacity increases.

(b) As a direct result of (a), a route may be allocated – in step (2) of the procedure in Section 2.5.2 – while some of its conflicting routes are still in use by trains, instead of waiting for all of its conflicting routes to be released as specified in the procedure.

The advantage of sequential release is better illustrated by an extension of the example shown in Figure 2.2 where several linear sections reside between sections `t11` and `t12` on route `1a` in Table 2.3. For such an extended example, as soon as a train `T1` has left `t11` while going along route `1a`, `t11` can be released. As a consequence, route `7` (see Table 2.3) can already be *allocated* by another train `T2` (assuming that other allocation conditions are fulfilled), while `1a` is still used by `T1`. If sequential release

had not been employed, route `7` could first have been *allocated*, when route `1a` had been released (i.e., when `T1` had passed `mb13`, or it had come to a standstill in `t12` in front of `mb13`).

Note that the term *sequential release* may be used differently at different operational levels. (1) At a high operational level, sequential release is performed on compound routes: each single elementary route of a given compound route is released sequentially. In that sense, sequential release is a *route-wise* feature. (2) On the other hand, at a low operational level, sequential release is performed on elementary routes: each single element of an elementary route is released sequentially [TVA09, chap. 4]. In this case, sequential release is an *element-wise* feature. In the former usage of the term, sequential release is merely used for operational purposes so that part of a long compound route can be released sooner. Although elementary routes are released one by one, the release of each elementary route has to follow the release procedure for an elementary route. Therefore, if the procedures for releasing elementary routes are correct, then sequential release, in its former sense, does not have any influence on safety of the system. On the other hand, in the latter usage of the term, sequential release is more granular and it results in significant changes in the procedures for setting and releasing routes, consequently safety of the considered systems. Thus it poses extra challenges in proving safety properties of the systems. Furthermore, the latter offers better train throughput gain (which is the main purpose of sequential release) than the former, and if the sequential release is done correctly at the low operational level, it is not needed in the high operational level. Compound routes become simply operational shortcuts. In the work documented in this dissertation, we consider sequential release in its latter usage of the term. In order to include this feature into the models of interlocking systems, additional variables and transitions are required as described in Chapter 6. Therefore, the models become more complex with a higher level of concurrency. Consequently, the verification tasks are more challenging.

2.6 Product Line Characteristics

Railway interlocking systems have the characteristics of *product lines* [HP15]:

(a) There exist *generic applications* that are common and reusable for all systems; and

(b) Each individual system is produced by instantiating the generic applications with the concrete *configuration data* for that system.

As illustrated in Figure 2.4, for railway interlocking systems, the generic applications are generic interlocking logic and control algorithms, generic safety and functional properties that must be satisfied by a system. These generic applications are derived from railway signalling rules and know-how. The configuration data for an individual interlocking system is the concrete network layout under control and the corresponding interlocking table of the given interlocking. Each individual

interlocking system is constructed by instantiating the generic applications with the concrete configuration data for that interlocking system.

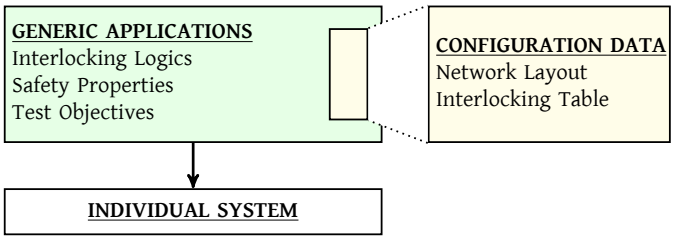


Figure 2.4: Illustration of the product line characteristics of interlocking systems

2.7 Formal Methods

Software and hardware systems play more and more important roles in our daily lives. As a result, they are growing in their functionalities and complexity. An error in such systems may cause catastrophic loss of money, time, or even human lives. A grand challenge in software engineering is to enable the development of systems that operate reliably despite the complexity. A way to address this grand challenge is to use formal methods. This section presents a brief introduction to formal methods.

Formal methods employ "mathematically based languages, techniques, and tools for specifying and verifying" software or hardware systems [CW96]. Formal methods are the mathematical foundation for software and hardware development. By building a mathematically rigorous model of complex systems, it is possible to verify thoroughly the properties of the systems. Formal methods are involved in different phases of software development life-cycle such as specification, verification, or implementation [CW96; Col15; Hax10].

(1) *Formal Specification.* Formal specification is the process of describing rigorously a system and its desired properties using one or more formal languages. Formal languages have a fixed grammar with mathematically defined syntax and semantics. Formal specification is basically the process of converting the understanding of a system in informal forms such as contractual documentation in natural languages into mathematical forms. Writing precisely down a formal description of the system is two-fold beneficial [CW96]. First, it provides a greater understanding of the considered system. The specification process allows uncovering flaws, ambiguities, inconsistencies, and incompleteness in the design. Second, the process results in an artefact that can be formally analysed, and serves as an unambiguous communication device between different parties involving in the development process.

(2) *Formal Verification.* Formal methods differ from conventional methods by emphasizing heavily on provability and correctness [Col15]. The formal specification of a system is basically a set of theorems about that system, thus it is possible to reason about them and prove that they are correct. However, it shall be emphasised that formal methods cannot guarantee that software is perfect [Hal90]. Formal methods cannot fix bad assumptions in the design. However, they can identify the errors and mistakes in reasoning and do so efficiently. Two most well-established approaches in formal verification are *theorem proving* [Duf91; Fit96] and *model checking* [CGP00; BK08]. The former is the process of finding a deduction proof from the description of a system to its desired properties expressed in a given formal system, which defines a set of axioms and inferences rules. The latter is dedicated to state-based systems and relies on building a finite model of a system and then explore exhaustively the state space of the model to prove the desired properties. The former is less automated because it requires guidance from human. On the other hand, the latter can be fully automated, but has a major challenge of the *state explosion problem* – the problem of exponential growth of the state space with the size of the considered system. One promising direction is to combine model checking and theorem proving, taking advantages of both approaches in formal verification [CW96].

(3) *Formal Implementation.* Once a model of a system is specified and verified, the system can be formally implemented by converting the specification into executable code. Executable code can be ensured to conform to the specification by object code verification, e.g., see [HPK11], using a verified compiler, e.g., CompCert[††] [Ler09] or [CO84], and verified hardware, or using other runtime verification techniques, e.g., see [LS09; Bar+04; Hav15].

Although formal methods research has been progressing since the sixties, they have been mainly used in critical application domains such as aviation, power, or transportation industries. One of the many reasons for such limited applicability is the misconception about formal methods pointed out in [Hal90; BH95; Col15]. Nevertheless, formal methods have gained more and more attention from the community. It has been shown that even applying formal methods correctly, even in the scope of a fraction of the development cycle, would greatly improve the understanding and quality of the resulting product [CW96; Woo+09]. Therefore, formal methods are strongly recommended by many industrial standards for safety/mission-critical hardware/software systems in aerospace, aviation, defence, railway, or finance domains. For example, in the railway domain, CENELEC 50128:2011 standard [CEN12] strongly recommends the use of formal methods in development and verification of systems with the highest safety requirement (SIL4).

[††] `http://compcert.inria.fr/`

2.8 Model Checking

Model checking is a well-established technique [CGP00; BK08] to verify that a system conforms to the specification of its intended behaviours. This is done by exploring the state space in an exhaustive, efficient, and highly automatic manner. However, the state space grows exponentially in the number of system components, which is referred to as the *state explosion problem*. Although there has been lot of advancement in efficient techniques for storing and exploring state space, the state explosion problem remains still a primary obstacle preventing model checking from being applied to complex systems in industrial applications.

Bounded Model Checking (BMC) remedies the state explosion problem by investigating model properties within the vicinity of a state, exploring only those states that are reachable by means of a bounded number of transition steps. Reachability is decided using satisfiability solving, hence exploring the whole state space is avoided [Bie+99a]. Theoretically, BMC can verify global properties – properties that hold in all reachable states of a system, e.g., invariants described in Section 2.11.1 – if the transition relation is unrolled for a sufficient number of steps, known as the *recurrence diameter* of the given transition system [Bie+99a; Bie+06]. In practice, the recurrence diameter of a given transition system is often too large, resulting in exhaustion of memory or unacceptable verification time, because the worst case complexity of the satisfiability solving grows exponentially in the number of unrolling steps. An alternative technique to verify global properties using BMC is to combine it with inductive reasoning, resulting in a technique called k-induction as explained in Section 2.11.2. Although it is not always feasible in practice to verify properties with BMC, it is getting more and more attention from research and industries for its excellent ability in finding bugs and automated test case generation.

2.9 Model-based Testing

Testing is a way to check the correctness of a system implementation by experimenting with it [Tre99]. The system implementation is exercised in a controlled environment. Based on the observed behaviours, a verdict about the correctness is concluded. In the work presented in this dissertation, we concentrate on *conformance testing* – i.e., testing to show the system implementation conforms to some specifications of the intended behaviours of the system. It is well-known that testing is theoretically not *complete*, as described by Dijkstra's famous statement [BR70]: *"Testing shows the presence, not the absence of bugs"*. Nevertheless, testing is a dominant and well-adopted technique in the industry for verification and validation of software systems. Testing is a mandatory activity in the development process of safety-critical software systems, e.g., see [CEN12].

2.9.1 Formal Verification and Testing

The relation between formal methods with testing is summarised in by Tretmans in [Tre99]:

(1) "formal methods and testing are a perfect couple";

(2) "testing and formal verification are both necessary";

(3) "a formal verified specification is a good starting point for testing";

(4) "formal testing is a good starting point for introducing formal methods in software development".

With regard to verification and validation, testing and formal verification are complementary to each other [Con+07; Tre99; Rus+04]. Although formal verification on a system has been performed on a model of a given system, testing is necessary to ensure that the system behaves correctly on the given hardware. Due to many problems such as unexpected delays, different interfaces, or limited memory space and other resources, an already verified software system may behave incorrectly on certain hardware. Additionally, the cost and efforts on testing would be reduced significantly if formal verification has been performed.

2.9.2 Testing Terminology

Testing activities are structured into test suites. A *test suite* consists of a sequence of test cases. A *test case* is a specification fragment covering a test requirement. A *test requirement* specifies expected behaviours of a system implementation, referred as a System Under Test (SUT) in testing terminology. The following information is associated with a test case:

(1) specification of inputs that stimulate the SUT so that it exhibits the behaviours under investigation;

(2) specification of expected behaviours of the SUT by means of outputs in response to the provided inputs; and

(3) references to related requirements that are tested by this inputs.

In testing of reactive systems, inputs usually consist of input traces. An *input trace* is a finite sequence of input vectors. An input vector is a value assignment for the input interfaces of the SUT. Each input vector stimulates the SUT at a certain time of the test execution. One or more test cases are executed by a *test procedure*. The outcome of the execution of a test case on a given SUT is called *test result*. A test procedure uses a *test oracle* to check a test result against the corresponding expected behaviours and produces a *test verdict* such as passed, failed, or inconclusive. The relation between test requirements, test cases, test procedures, and test results is *traceability data*. Figure 2.5 illustrates the relation between different terms in testing terminology.

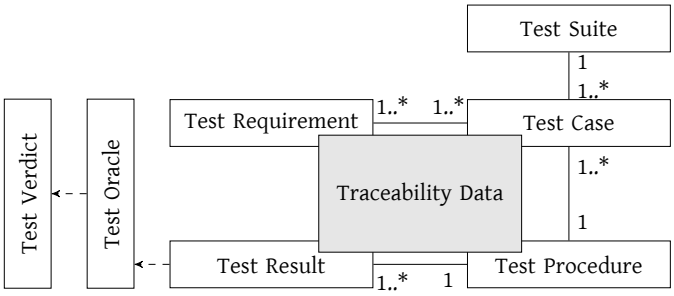


Figure 2.5: Testing terminology illustration

2.9.3 Model-based Testing

Model-based Testing (MBT) is an application of model-based design paradigm into testing. In MBT, there exists a *test model* specifying the correct behaviours of the given SUT. Once a test model in a suitable formalism – e.g., an input/output state transition system as described in Section 2.11 – such that desired requirements can be expressed by properties in a suitable logic – e.g., *Linear Temporal Logic (LTL)* [CGP00; BK08] – most of the remaining elements in MBT approach can be automatically generated [Bro+05; Pel13; FWA09]:

(a) Requirements can be identified automatically by exploring the test model or provided manually.

(b) Test cases can be automatically generated.

(c) Test oracles can be automatically generated from the test model.

(d) Test procedures can be automated created and executed.

(e) Traceability data can be automatically derived during above processes.

Due to its great potential of being rigorous and automated, MBT is getting adopted more and more widely in the industry, especially for safety/mission-critical systems. Furthermore, research in MBT is also very active. How to create a good test model, how to efficiently identify test requirements and generate test cases are among the questions needed to be answer [Bro+05; Pel13; FWA09].

2.10 Domain-specific Languages

Domain-specific Languages (DSLs) are languages that are *tailored to a particular domain*, in contrast to *General-purpose Languages (GPLs)*, that are applicable across domains and do not have specialised features for a particular domain. Many DSLs have been in use for years: BNF language for describing grammars [Knu64], DOT language for

describing graphs [Ell+01], R language for statistical computing and graphics [Tea00], regular expressions for pattern matching and text manipulation, or Cascading Style Sheets (CSS)[‡‡] just to name a few. DSLs are also widely used in domains other than computing: railway interlocking systems development, e.g., see [HP00; Mew10; Jam14], or material flow analysis modelling in waste management [ZB14].

A primary difference between DSLs and GPLs is the tradeoff between expressiveness and generality. *Expressiveness* here should be understood as *domain-wise expressiveness*, i.e., the ease to express various artefacts in a particular domain using the constructs defined by a language. Whereas *generality* means the ability to use the constructs defined by a language across different domains. While GPLs strive for generality, DSLs aim at expressiveness. A common feature of DSLs is that they incorporate the domain knowledge, i.e., concepts and notations, into the languages. Consequently, DSLs can express more efficiently the artefacts in their respective domains. Obviously, these artefacts can also be expressed by a GPL, but a domain-specific description offers certain advantages [MHS05].

(a) *Readability.* Having domain concepts and notations incorporated in the language makes a DSL description simpler and easier to understand, especially for users that are domain experts but are not familiar with software development. DSL descriptions can serve as a communication device between software engineers and their clients from the respective domain.

(b) *Reduce errors.* DSL restrictive constructs prevent certain erroneous descriptions from being made. Furthermore, as DSL descriptions are more readable, they are less prone to mistakes, especially when the descriptions become complex.

(c) *Productivity.* DSLs make the process of mapping domain knowledge to a description in the language straightforward. Consequently, this boosts the productivity in specification process, as it might be awkward and time-consuming to find an appropriate description in a GPL for a concept or notation in a given domain.

(d) *Reusability.* DSLs are developed with reusability in mind. The concepts and notations incorporated in a DSL can be reused for many applications in that particular domain.

(e) *Efficient analysis and verification.* DSLs offer the possibilities for more efficient and automated analysis, verification, optimisation, parallelization, and transformation. DSL restrictive constructs make it easier and more efficient to perform analyses and verification on DSL descriptions than on GPL descriptions.

On the other hand, DSLs have also drawbacks: they require a lot of development efforts, and the problems that they can express are limited. The problems that can be expressed by a DSL are restricted to its predefined constructs. Therefore, if an application in the domain requires extra constructs, the DSL has to be extended.

[‡‡] `http://www.w3.org/style/css`

Furthermore, contrary to GPLs, DSLs need to be developed for each domain of interest. The development process requires convergence of the knowledge of both language development and the respective domain. As the use of DSLs becomes more common, methodologies, frameworks, and tools for creating DSLs are developed. For a survey on when and how to create a DSL, see [MHS05]. Many GPLs nowadays have support for developing DSLs such as Ruby, Scala, or F#, just to name a few.

2.11 Mathematical Preliminaries

This section explains some mathematical preliminaries that are used in this dissertation, in particular Kripke structures, the k-induction scheme for proving invariants in a Kripke structure, and input/output state transitions systems [CGP00; BK08; Bro+05; Pel13].

Kripke structures are used to specify behavioural models of interlocking systems in our method, while k-induction scheme is used for verification of safety properties as explained in Chapter 6. Input/output state transitions systems are used to formalise test models in Chapter 7.

2.11.1 Kripke Structures

A *Kripke structure* $K \in \mathcal{K}$, where $\mathcal{K}$ is the domain of Kripke structures, is a five-tuple (S, I_0, R, L, AP) with state space S, a set of initial states $I_0 \subseteq S$, a total transition relation $R \subseteq S \times S$, and labelling function $L : S \to \mathcal{P}(AP)$, where AP is a set of atomic propositions and $\mathcal{P}(AP)$ is the power set of AP. The labelling function L maps a state s to the set $L(s)$ of atomic propositions that hold in s [CGP00; BK08].

Two states s and s' are said to be *consecutive* in K, if there is a transition from s to s', i.e., $(s, s') \in R$. A *path* in K is a finite or infinite sequence of consecutive states. A state s' is said to be *reachable* from another state s in K, if there exists a finite path $s \ldots s'$ starting in s and ending in s'. A state $s \in S$ is said to be *reachable* if it is reachable from an initial state $s_0 \in I_0$. The *reachable states* of K is the set of all reachable states.

In the context of this dissertation, the states of a Kripke structure are represented by valuation functions $s : V \to D$ over finite sets $V = \{v_0, \ldots, v_n\}$ of variables, where each variable $v_i \in V$ has an associated finite domain D_{v_i}. The range of a state s is $D = \bigcup_{v \in V} D_v$. The whole state space S is the set of all valuation functions $s : V \to D$ for which $s(v) \in D_v$ for all $v \in V$. The *equality* relation $(=)$ between states is defined by the equality of mathematical functions as follows: two states s and s' are equal – denoted by $s = s'$ – iff every variable $v \in V$ is evaluated to the same value in s and s', i.e.,

$$(s = s') \equiv \left(\bigwedge_{v \in V} s(v) = s'(v) \right) \tag{2.1}$$

For a proposition ϕ over free variables in V, we use $\phi(s)$ to denote the proposition obtained by replacing every occurrence of $v \in V$ in ϕ by the value $s(v)$. A proposition ϕ over free variables in V is said to *hold* in a state $s \in S$, denoted as $s \models \phi$, iff $\phi(s)$ holds. An *invariant* in K is a proposition that holds in all reachable states of K.

With the above definition, the set of initial states I_0 can be represented in propositional form as a proposition $\mathcal{I}$ over free variables in V such that

$$I_0 = \{s_0 \in S \mid \mathcal{I}(s_0)\} \tag{2.2}$$

The transition relation $R \subseteq S \times S$ can also be represented in propositional form as a proposition Φ over free variables in $V \cup V'$ such that

$$R = \{(s, s') \in S \times S \mid \Phi(s, s')\} \tag{2.3}$$

where $V' = \{v' \mid v \in V\}$ is a duplicate of V used to representing the next state, and $\Phi(s, s')$ is the proposition Φ with every occurrence of $v \in V$ replaced by the value $s(v)$, and every occurrence of $v' \in V'$ replaced by the value $s'(v)$.

A finite path $s_n.s_{n+1} \ldots s_{n+k-1}$ of length k through the model K, starting in an *arbitrary* state s_n (may or may not be reachable), is identified by a solution to the satisfiability of the following proposition.

$$\pi(s_n, \ldots, s_{n+k-1}) \equiv \bigwedge_{i=1}^{k-1} \Phi(s_{n+i-1}, s_{n+i}) \tag{2.4}$$

An *acyclic* path of length k is a finite path $s_n.s_{n+1} \ldots s_{n+k-1}$ in which there does not exist a pair of states in the path that are equal. Such a path is characterized by a solution to the satisfiability of the following proposition.

$$\pi^{=}(s_n, \ldots, s_{n+k-1}) \equiv \pi(s_n, \ldots, s_{n+k-1}) \wedge \bigwedge_{n \leq i < j \leq n+k-1} (s_i \neq s_j) \tag{2.5}$$

2.11.2 k-Induction

As explained in Section 2.8, although BMC can be used to verify global properties, it is often not feasible in practice due to many transition steps need to be unrolled to reach the recurrence diameter. To remedy this obstacle in verifying global properties, *k-induction* – a technique that combines BMC and inductive reasoning – is used.

A Complete k-Induction Scheme. In order to prove a proposition ϕ is an invariant in a Kripke structure K, a complete k-induction scheme [SSS00; MRS03] based on acyclic paths can be applied. The scheme consists of the following two steps.

(1) *Base Step*: prove that ϕ holds in every state of every acyclic path $s_0.s_1 \ldots s_{k-1}$ of length $k > 0$, starting from every initial state $s_0 \in I_0$, i.e., the following holds, where $\Rightarrow$ is logical implication.

$$(\mathcal{I}(s_0) \wedge \pi^{=}(s_0, \ldots, s_{k-1})) \Rightarrow \bigwedge_{i=0}^{k-1} \phi(s_i) \tag{2.6}$$

(2) *Induction Step*: prove that if ϕ holds in every state of an acyclic path $s_n.s_{n+1} \ldots s_{n+k-1}$ of length $k > 0$, starting from an arbitrary state s_n, then ϕ will also hold in every $(k+1)^{th}$ state s_{n+k}. In other words, the following holds

$$\left(\pi^{=}(s_n, \ldots, s_{n+k}) \wedge \bigwedge_{i=0}^{k-1} \phi(s_{n+i})\right) \Rightarrow \phi(s_{n+k}) \tag{2.7}$$

Transformation to Bounded Model Checking Problems. Both base step and induction step can be transformed to bounded model checking problems of finding witnesses for the violations. Violations of the base step are identified by the negation of Formula 2.6 as shown in the following.

$$\mathcal{I}(s_0) \wedge \pi^{=}(s_0, \ldots, s_{k-1}) \wedge \neg \bigwedge_{i=0}^{k-1} \phi(s_i) \tag{2.8}$$

A solution, if found, for Formula 2.8 identifies an execution of the system in which ϕ does not hold in at least one state within the vicinity of k transition steps from the initial state s_0. Likewise, violations of the induction step are identified by the negation of Formula 2.7 as shown in the following.

$$\pi^{=}(s_n, \ldots, s_{n+k}) \wedge \bigwedge_{i=0}^{k-1} \phi(s_{n+i}) \wedge \neg\phi(s_{n+k}) \tag{2.9}$$

A solution, if found, for Formula 2.9 shows an execution of length $(k+1)$ of the system where ϕ holds for the first k states, but not in the last one. If no violation is found for the base step or induction step, then ϕ is an invariant in K.

Strengthening Invariants. As pointed out in [MRS03], when ϕ is not strong enough to be inductive, counter-examples are found for the induction case. If ϕ is indeed an invariant in K, then these counter-examples are *spurious*, i.e., they start from an unreachable state and do not correspond to any actual run of the considered system. In order to make ϕ inductive, it is strengthened with an extra invariant ψ, i.e., one should prove $\phi \wedge \psi$ instead ϕ. ψ is called the *strengthening invariant*, which eliminates the spurious counter-examples.

2.11.3 Input/Output State Transition Systems

Kripke structures are used to model *closed* systems, in the sense that for every interface, both communication parties (readers and writers) are parts of the model. Consequently, there are no explicit notions of input or output: the whole system evolves according to the transition relation. However, in hardware/software integration testing, the SUT is stimulated by a test engine simulating the operational environment of the SUT and observing the SUT's reaction to the stimulations. Therefore, input/output state transition system semantics are employed in formalising test models. Input/output state transition system semantics is related to Kripke structure semantics but streamlined for testing purposes [Tre08; Tre96b; Pel13; HP15; Tre11; Tre96a] as explained in the following.

A *State Transition System (STS)* is a triple $TS = (S, I_0, R)$ with state space S, a set of initial states $I_0 \subseteq S$, and transition relation $R \subseteq S \times S$. For testing purposes, we need to stimulate the SUT via its certain input interfaces, and observe its reaction via its output interfaces. Therefore, we focus on STSs possessing the notion of variable valuations, input, and output. Additionally, the SUT may have some internal state variables that can not be observed during black-box testing. This focused class of STSs are referred as input/output state transition systems.

An *Input/Output State Transition System (IOSTS)* [Tre96b; HP15; Tre08] $TS \in \mathcal{TS}$, where $\mathcal{TS}$ is the domain of IOSTSs, is an STS where states $s \in S$ are valuation functions $s : V \rightarrow D$ as introduced in Section 2.11.1 for Kripke structures, except that the set of variables representing the state space can be partitioned into three disjoint sets of input variables $I = \{x_1, \ldots, x_t\}$, output variables $O = \{y_1, \ldots, y_q\}$, and (internal) model variables $M = \{m_1, \ldots, m_p\}$, that is

$$V = I \cup M \cup O \tag{2.10}$$

Furthermore, as in Kripke structures described in Section 2.11.1, the initial states I_0 and the transition relation R can also be presented in the propositional forms by a proposition $\mathcal{I}$ and Φ, respectively. Paths in an IOSTS are defined in the same way as in Equation 2.4 and Equation 2.5 for paths in a Kripke structure.

The *restriction* of a state s to variables from a set $U \subseteq V$, denoted by $s\mid_U$, is a function that has domain U and coincides with s on this domain. An IOSTS can be naturally extended to a Kripke structure by defining a set of atomic propositions AP as a subset of

$$\mathcal{A}(V) = \{p \mid p \text{ is an atomic proposition with free variables in } V\} \tag{2.11}$$

The labelling function L is then specified as a function mapping from a state $s \in S$ to $L(s)$ as specified in the following.

$$L(s) = \{p \in AP \mid p(s)\} \tag{2.12}$$

Since safety-critical systems are highly recommended to be deterministic, thus we only consider *deterministic* IOSTSs. An IOSTS is called deterministic if identical input

sequences lead to identical output sequences, i.e., for every pair of paths – similar to the paths in a Kripke structure defined in Section 2.11.1 – with equal length $s_0.s_1 \ldots s_n$ and $s_0.s'_1 \ldots s'_n$, the following holds [HP15].

$$\begin{aligned}&(s_0 \mid_I).(s_1 \mid_I) \ldots (s_n \mid_I) = (s_0 \mid_I).(s'_1 \mid_I) \ldots (s'_n \mid_I) \Rightarrow \\ &(s_0 \mid_O).(s_1 \mid_O) \ldots (s_n \mid_O) = (s_0 \mid_O).(s'_1 \mid_O) \ldots (s'_n \mid_O)\end{aligned} \tag{2.13}$$

Two states of an IOSTS are called *I/O-equivalent* if applying the same sequence of inputs to them results in the same sequence of outputs. Two IOSTSs are *I/O-equivalent* if their initial states are [HP15; Tre96b; Tre08].

Let D_I denotes the Cartesian product of the domain of input variables, i.e., $D_I = D_{x_1} \times \ldots \times D_{x_\iota}$; D_M and D_O are defined analogously. Using IOSTS semantics, a test case can be characterized by an input sequence $\iota \in D_I^*$ and a corresponding output sequence $\rho \in D_O^*$ of the same length as ι specifying the expected outputs when applying ι to the initial state.

CHAPTER 3

Method Overview

This chapter gives an overview of the main contribution of this dissertation: a holistic method – in other words, a recipe – for development, validation and verification of safety-critical railway control systems that have product line characteristics. First, the motivation is presented in Section 3.1. The elements of the method – the ingredients of the recipe – are given in Section 3.2, and Section 3.4 describes the development, verification, and validation work flow. The application of the proposed method to railway interlocking systems are presented in Section 3.7.

3.1 Motivation

As in other domains, software and hardware systems in the railway domain are growing in their complexity and criticality. It follows naturally that formal methods are strongly recommended by CENELEC 50128:2011 standard for railway applications with highest safety integrity level (SIL4). However, for many reasons, formal methods have not been widely employed in the industry. One among these reasons is that railway signalling engineers are not familiar with formal methods. Therefore, there is a need for a user-friendly interface to formal methods for domain engineers. Domain-specific approaches address this need by encapsulating the use of formal methods with familiar concepts and notions of the considered domain. Thus, in this dissertation, we propose a method that is a combination of domain-specific approaches and formal methods. Such combination would give a better understanding of the domain and better development and verification results.

3.2 Ingredients

In this dissertation we propose a method for efficient Verification and Validation (V&V) of safety-critical systems with the product line characteristics described in Section 2.6. The method is based on of the following elements.

(a) A *domain-specific language* for specifying *configuration data*.

(b) A *domain-specific language* for specifying *generic applications* (behavioural models, properties*, test objectives) that is built based on the DSL for configuration data.

(c) A DSL *specification editor* and *static checker* for configuration data.

(d) A DSL *specification editor* and *static checker* for generic applications.

(e) A *model generator* that takes a DSL specification of a generic behavioural model and a wellformed DSL specification of concrete configuration data as input, and produces a concrete behavioural model as output.

(f) A *property generator* that takes a DSL specification of generic properties and a wellformed DSL specification of concrete configuration data as input, and produces concrete properties as output.

(g) A *test generator* that takes a DSL specification of generic test objectives and a wellformed DSL specification of configuration data as input, and produces concrete test objectives as output.

(h) A *bounded model checker* that can perform k-induction.

(i) An MBT framework that supports automated test generation and execution.

3.3 Why Two Domain-specific Languages?

It is a *novelty* in our method to have *two* domain-specific languages (a) and (b) as explained in the following.

Domain-specific approach has been well-adopted to encapsulate the use of formal methods in formal development and verification of railway interlocking systems, e.g., see [JR14; Hax14; Mew10; Cao+11; Jam14]. In these works, there exists a single DSL which is mainly used for specifying the configuration data. Generic applications are specified in a GPL in which the configuration data is represented, showing how configuration data is manipulated and transformed into underlying formal models. There are few drawbacks with this approach:

*These include generic safety properties and generic strengthening invariants used for k-induction scheme described in Section 2.11.2

- Generic application specifications contain extra technical details due to the use of GPL constructs. Thus, they are not so readable and not easy to understand for domain engineers.
- Generic application specifications may be error-prone, as it is difficult for domain engineers to review the specifications.
- Generic applications are often difficult to change as changing them requires sufficient technical understanding.

To address these drawbacks, in our method, a DSL is dedicated for specifying generic applications besides a DSL for specifying configuration data. Having two domain-specific languages offers the following advantages:

- *Readability.* DSL descriptions of generic applications are more readable and easy for domain expert to understand as the tedious technical details are stripped out thanks to the dedicated DSL.
- *Reduce errors.* It is easier for domain engineers to review generic applications specifications.
- *Ease to use/change.* Domain engineers can change the generic applications with ease as the constructs are intuitive and familiar.
- *Different levels of abstraction.* The two DSLs offer two different levels of abstraction for different groups of users. While the DSL (a) is suitable for engineers from the customer side, e.g., Banedanmark, the DSL (b) is suitable for engineers from the supplier side, e.g., Thales. The customer-side engineers can focus on the configuration of each individual application, while it is assured that the generic component is the same across these individual applications. Specifications in these two DSLs can serve as unambiguous communication devices between customer-side engineers and supplier-side engineers.

3.4 Verification and Validation Flow

This section describes how the proposed method coordinates the ingredients in Section 3.2 in an efficient work flow for verification and validation of safety-critical systems with product line characteristics. The V&V flow consists of the following steps, as illustrated in Figure 3.1.

First, for a *product line*, the generic applications are created and validated by the following steps.

(1) A DSL specification of a generic behavioural model, generic properties, and generic test objectives are created in the language (b) using the specification editor described in (d).

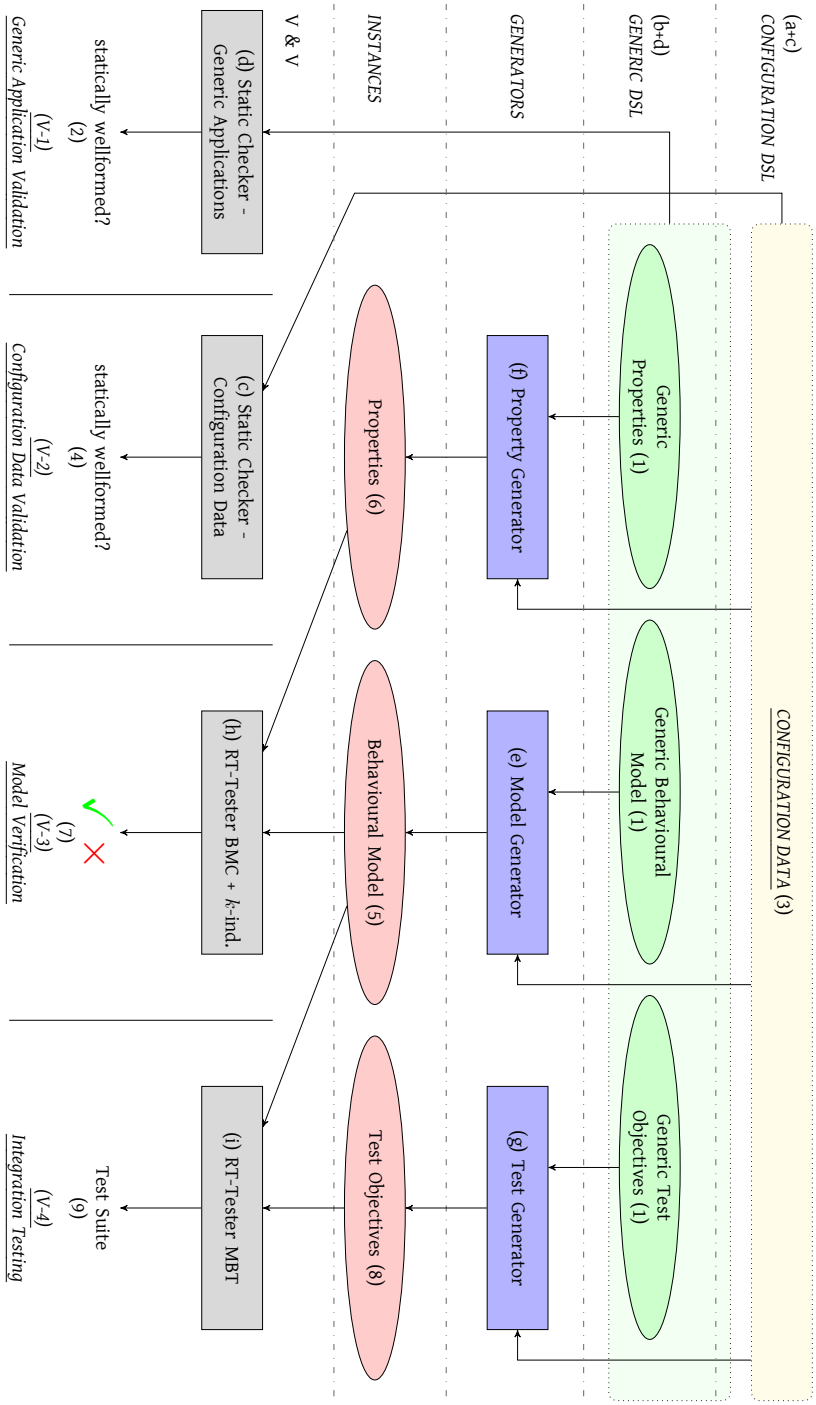


Figure 3.1: A method for V&V of systems with product line characteristics

(2) The static checker in (b) verifies that the created generic behavioural model, generic properties, and generic test objectives are statically wellformed according to the static semantics of the language (b).

Then, for each *individual system* in the above product line:

(3) A DSL specification of configuration data is created in the language (a) using the specification editor described in (c).

(4) The static checker in (c) verifies that the created configuration data is statically wellformed according to the static semantics of the language (a).

(5) Based on the dynamic semantics of the DSLs, the model generator (e) instantiates the generic model with the configuration data, resulting in a model instance.

(6) Similarly, the property generator (f) instantiates the generic properties (safety properties and strengthening invariants) with the given configuration data, resulting in concrete safety properties and strengthening invariants.

(7) The model instance generated in step (5) is then checked against the concrete properties using the combination of BMC and inductive reasoning supported by the bounded model checker (h). If the generated model does not satisfy the properties, counter-examples will be generated. The counter-examples are represented in the DSL level for debugging purposes.

(8) If the model instance generated in step (5) satisfies all the concrete properties, then the test generator (g) instantiates the generic test objectives with the given configuration data, resulting in concrete test objectives.

(9) The model instance generated in step (5) is used as the test model to generate a test suite for the concrete test objectives. The generated test suite can automatically be executed in the MBT framework (i) during integration testing. The test results are summarised into test reports that can be used as evidence for safety certification.

3.5 Remarks on Development Models

One may ask where software development comes into the diagram shown in Figure 3.1. The answer to this question varies depending on the organizational structure. A preferred organizational structure for SIL4 systems according to CENELEC 50128:2011 standards is shown in Figure 3.2. As can be seen, software development (performed by RQM, DES, IMP) and software V&V (performed by INT, TST, VER, VAL) are performed by different teams. Therefore, there are two possible scenarios: the same model is used for both development and V&V, or different models are used. In the former scenario, the software implementation may be derived (manually or automatically) from the verified behavioural model after step (7) in the flow described in Section 3.4. While in the latter scenario, both models are derived from the same

set of signalling rules and requirements, and the flow in Section 3.4 can be used for independent V&V activities.

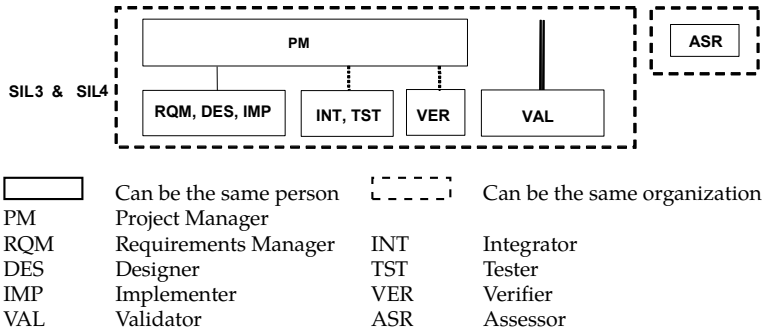


Figure 3.2: Preferred organizational structure for development of SIL3 & SIL4 systems according to CENELEC 50128:2011 [CEN12]

3.6 4-step Verification and Validation

With the V&V flow in Section 3.4, the verification and validation of an individual system are performed in the following 4-step approach.

VV-1 *Generic Application Validation*: The wellformedness of the generic applications is validated by the static checker in step (2).

VV-2 *Configuration Data Validation*: The wellformedness of the given configuration data is validated by the static checker in step (4).

VV-3 *Model Verification*: The safety properties are verified in step (7).

VV-4 *Integration Testing*: The conformance of the implementation of the system to the test model is checked in step (9).

This 4-step V&V approach allow revealing errors in the generic applications, the configuration data, the model instance, the concrete properties, or the concrete test objectives as early as possible in the development life-cycle. Consequently, this reduces the development cost. Additionally, these steps would allow isolating the errors, if there is any, hence making it easier for debugging.

3.7 Application to the Danish Interlocking Systems

For the case of the forthcoming Danish interlocking systems, the following V&V environment has been developed, following the recipe described in Section 3.2 and Section 3.4.

Ingredients. The ingredients for developing the Danish interlocking systems are shown in Figure 3.3. The ingredients here are suffixed with the numbering of their corresponding ingredients described in Section 3.2. For example, **DK:a** corresponds to (a) in Section 3.2. The ingredients are elaborated in the following.

DK:a *Interlocking Configuration Language (ICL)* – a DSL for specifying interlocking configuration data. Chapter 4 explains in detail the abstract syntax and semantics of ICL.

DK:b *Interlocking Dynamic Language (IDL)* – a DSL for specifying generic interlocking applications (behavioural models, safety properties, and test objectives). The syntax and semantics of IDL are elaborated in Chapter 5.

DK:c A *specification editor* and *static checker* for ICL. A graphical specification editor has been implemented by a master's student as an Eclipse[†] plug-in [Fol15]. The static semantics of ICL is presented in Chapter 4.

DK:d A *specification editor* and *static checker* for IDL. IDL has a textual concrete syntax which can be edited using any text editor. Due to the limited time frame, the static checker for IDL has not been fully implemented yet. At the current stage, only syntax and some sanity checks are performed. The sanity checks are done on-the-fly during generation of concrete behavioural model, safety properties, and test objectives as explained in the subsequent steps in this work flow.

DK:e A *model generator* that takes a wellformed generic interlocking model in IDL and wellformed interlocking configuration data in ICL as input, and produces a concrete interlocking model in the form of a Kripke structure as output. The generation is based on the semantics of ICL and IDL as described in Chapter 4 and Chapter 5.

DK:f A *property generator* that takes generic safety properties specified in IDL and wellformed interlocking configuration data as input, and produces concrete safety properties in the form of invariants in the Kripke structure generated in step **DK:e**. The generation is based on the semantics of ICL and IDL as described in Chapter 4 and Chapter 5.

DK:g A *test generator* that takes generic test objectives specified in IDL and wellformed interlocking configuration data in ICL as input, and produces concrete test objectives. The generation is based on the semantics of ICL and IDL as described in Chapter 4 and Chapter 5.

DK:h RT-Tester: an MBT framework and a *bounded model checker* that can perform k-induction [Pel13; Ver15]. RT-Tester has been selected because (1) it is an integrated model-based testing and BMC tool, and (2) its Satisfiability

[†] `https://eclipse.org`

Modulo Theories (SMT) solver also supports floating point arithmetic. The first property is crucial for us, because our objective is to complement the model verification with HW/SW integration tests. The second capability is vital, because we also plan to extend the model by real-time aspects, such as train velocity and braking curves.

DK:i RT-Tester is also chosen as the *MBT framework*.

V&V Flow. The V&V flow as described in Section 3.4 when adapted to the Danish interlocking systems results in the flow depicted in Figure 3.3. The steps are suffixed with the numbering of their corresponding steps in Section 3.4. For example, step **DK:3** corresponds to step (3) in Section 3.4. The steps are explained in the following.

First, for a product line of the Danish interlocking systems, the generic applications are created and validated by the following steps.

DK:1 The specification of a generic behavioural model, safety properties, and test objectives of the Danish interlocking systems in IDL are created. Chapter 6 describes the model and safety properties in detail, while the generic test objectives are presented in Chapter 7.

DK:2 The generic model, safety properties, and test objectives are syntactically checked during parsing. They are further checked on-the-fly during the generation process described in steps **DK:5** to **DK:8**.

Then, for each individual interlocking system:

DK:3 The specification of interlocking configuration data – consisting of a network layout and its corresponding interlocking table – in ICL is created. The specification is given in an eXtensible Markup Language (XML) representation [VHP14a]. Alternatively, the specification can be created in a graphical editor implemented by Foldager as an Eclipse plug-in in his master's thesis [Fol15]. As an option the user may not provide an interlocking table, but instead have an interlocking table created automatically from the network layout by an Interlocking Table Generator (ITG) [VHP14a].

DK:4 The static checker verifies whether the configuration data is statically well-formed according to the static semantics of ICL.

DK:5 The model generator instantiates the generic model created in step **DK:1** with the well-formed configuration data created in step **DK:3**. This results in a model instance in the form of a Kripke structure.

DK:6 Similarly, the property generator instantiates the generic safety properties. This results in concrete safety properties expressed as state invariants in the Kripke structure generated in step **DK:5**.

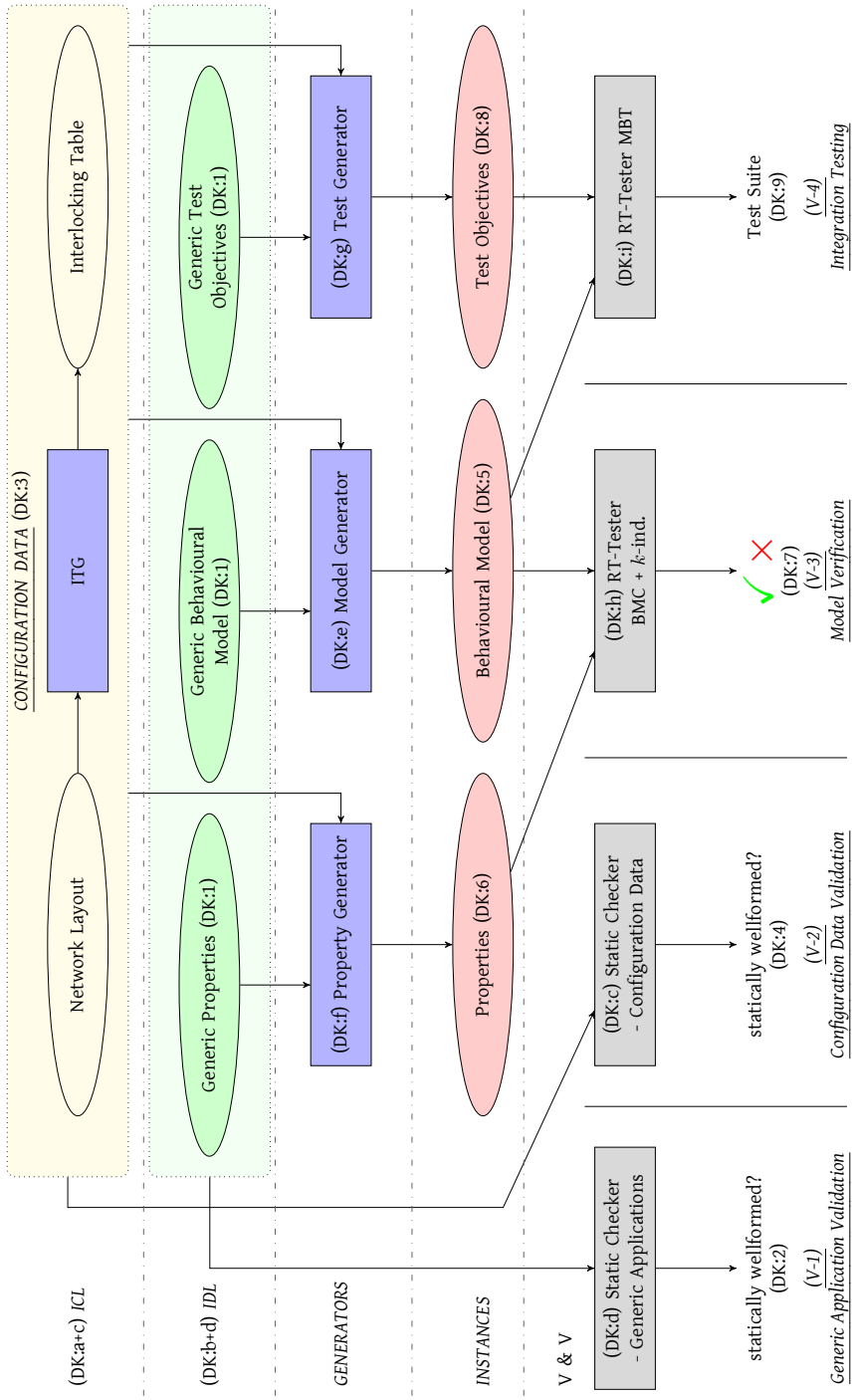


Figure 3.3: Method for V&V of the Danish interlocking systems

DK:7 The model instance generated in step **DK:5** is then checked against the concrete properties generated in step **DK:6** using a combination of BMC and inductive reasoning in RT-Tester. The bounded model checker in RT-Tester uses the SONOLAR SMT solver [PVL11] to compute counter-examples showing the violations of the base step or induction step of *k*-induction. If the model instance does not satisfy the properties, counter-examples will be generated. An interface for visualizing the counter-examples at the DSL level is integrated into the editor in Eclipse. Depending on whether the bugs are in the configuration data or the generic model, one shall go back to step **DK:3** or step **DK:1**, respectively, to fix the issues and start the process over. The verification strategy is presented in Chapter 6.

DK:8 If the generated model satisfies all safety properties in step **DK:7**, the test generator instantiates the generic test objectives with the configuration data, resulting in the concrete test objectives for the corresponding system.

DK:9 The model instance in step **DK:5** is used as test model for generating a test suite for the concrete test objectives generated in step **DK:8** using the RT-Tester framework. This test suite can then be used for integration testing. Chapter 7 explains the generation in detail.

3.8 Prototype Implementation

Table 3.4 lists the specification and implementation of a prototype of the associated toolchain of the method. The first column describes the name of the components, while the second column describes the technologies used for implementing that component. The last column describes the status of the components. A status of *New* means that the component has been developed as part of this PhD project. A status of *Reused* denotes that the component already existed (made by others), and was reused in this PhD project. A status of *Extend* denotes that the component has been developed as part of this PhD project by extending an existing framework with new features. A status of *MSc. student* indicates that the component was developed by a student whose master's thesis was co-supervised by the author.

Table 3.4: Prototype toolchain implementation

New	The component is newly implemented as part of this PhD project
Reused	The component already existed (made by others), and is reused in this PhD project
Extend	Extend the existing framework with new features
MSc. student	The component is developed by a master's thesis student that the author co-supervised

Component	Technologies	Status
ICL Specification	RSL	New
ICL	XML, C++	New
ICL Graphical Editor	EMF, GMF, Eclipse plugin	MSc. student [Fol15]
Static Checker	C++	New
IDL	BNF, Flex, Bison, C++	New
Model Generator	C++	New
Property Generator	C++	New
Test Generator	C++	New
k-induction	C++, RT-Tester	New
MBT framework	RT-Tester	Reused
Verification results visualisation	EMF, GMF, Eclipse plugin	MSc. student [Fol15]
Danish generic applications	IDL	New
Testing strategy for the Danish interlockings	IDL, C++	Extend

CHAPTER 4

A Domain-specific Language for Interlocking Configuration Data

This chapter presents a formal specification of *Interlocking Configuration Language (ICL)* – a DSL for describing the configuration data of interlocking systems that are compatible with ETCS Level 2. ICL is the ingredient **DK:a** – the *first* of the two DSLs described in Section 3.7. The second language – IDL – for specifying generic components will be elaborated in the next chapter, Chapter 5. The advantages of having two different DSLs has been explained in Section 3.3. Additionally, this chapter describes also an *Interlocking Table Generator (ITG)* that generates automatically a wellformed interlocking table from a wellformed railway network layout. This chapter elaborates in more detail the contribution published in [VHP14a] for which the authors have won the Best Paper award at the 10th Symposium on

Formal Methods for Automation and Safety in Railway and Automotive Systems – FORMS/FORMAT 2014, Braunschweig, Germany.

> [VHP14a] – Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "A Domain-Specific Language for Railway Interlocking Systems". In: *FORMS/FORMAT 2014 - 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*. Edited by Eckehard Schnieder and Géza Tarnai. Best paper award. Institute for Traffic Safety and Automation Engineering, Technische Universität Braunschweig., 2014, pages 200–209. ISBN: 978-3-9816886-6-5

The abstract syntax of ICL, its static semantics, and the ITG are formally specified in RSL [Gro92]. The abstract syntax is specified as data types in RSL while the static semantics and the ITG are specified as predicates over the data types specifying the abstract syntax. The full specification of ICL in RSL can be found in Appendix A, or can be obtained at `http://l.dtu.dk/3f52`.

The remainder of this chapter is organised as follows: the abstract syntax of ICL is presented in Section 4.1, while Section 4.2 describes the static semantics of ICL. Section 4.3 and Section 4.4 elaborate in detail how route protection and conflicting routes, respectively, are automatically checked in the static checker. Section 4.5 presents the ITG. The dynamic semantics is briefly presented in Section 4.6 and further explained in the next chapter along with the semantics of IDL. Section 4.7 explains how RSL specifications of ICL, its static checker, and the ITG can be executed directly, and the benefits of that. Section 4.8 and Section 4.9 describe implementation, and related work, respectively.

4.1 Abstract Syntax

The abstract syntax of the language is formally specified as data types in RSL [Gro92]. An excerpt of paper [VHP14a, Section 2] describing the abstract syntax of ICL is adopted and presented in the following.

A description of an interlocking system in ICL consists of a railway network layout and an interlocking table. In the following two subsections, the abstract syntaxes of network layouts and interlocking tables are specified as types in RSL.

4.1.1 Railway Network Layouts

A railway network layout* is a description of the topology of a railway network. Each element in a network layout is given a unique id. Therefore, we introduce types for ids of track sections and marker boards, respectively.

*In the remaining of this dissertation, the terms *network*, *network layout*, *railway network layout* are used interchangeably.

SecId = Id,
MbId = Id,
Id = **Text**

A network layout is represented as a record consisting of three maps – one for each kind of element – mapping the ids of the elements into geographical information about these elements.

NetworkLayout ::
  linears : SecId $\overset{m}{\rightarrow}$ Linear
  points : SecId $\overset{m}{\rightarrow}$ Point
  marker_boards : MbId $\overset{m}{\rightarrow}$ MarkerBoard

The information recorded about a linear section is composed of information about its neighboring sections and its length. A linear section may have up to two neighbors: one at the *down* end and at the *up* end[†].

Linear ::
  neighbors : LinearEnd $\overset{m}{\rightarrow}$ SecId
  length : Distance,
Direction == DOWN | UP,
LinearEnd = Direction,
Distance = **Nat**

For each point section similar information is recorded. In this case there are up to three neighboring sections: one at the *stem* end, one at the *plus* end, and one at the *minus* end. The length of a point section is the distance from the stem tip to the plus (or minus) tip.

Point ::
  neighbors : PointEnd $\overset{m}{\rightarrow}$ SecId
  length : Distance,
PointEnd == NB_STEM | NB_PLUS | NB_MINUS

The information recorded about a marker board includes: the id of the section along which it is placed, the travel direction (up or down) that it is intended for, and the distance from the location where it is placed to the tip of the section in the marker board's travel direction, as illustrated in Figure 4.1.

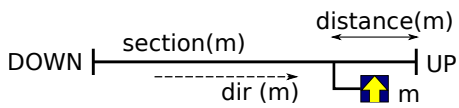


Figure 4.1: A marker board *m* and its associated geographical information.

[†]In Denmark *up* and *down* denote the directions in which the distance from a certain reference location is increasing and decreasing, respectively.

```
MarkerBoard ::
  section : SecId
  dir : Direction
  distance : Distance
```

4.1.2 Interlocking Tables

In order to guide trains safely through a railway network, the interlocking system reserves exclusively a fraction of the network, called a *route*, for a train at a time. Contrary to legacy systems, in ETCS Level 2, there are no physically signals (along the tracks), but *virtual signals*[‡]. A virtual signal is associated with a marker board, and has the same geographical information as the marker board. The aspects of virtual signals are used to calculate the movement authorities determining how far forward trains are allowed to move [ERT14]. A *route* is defined as a path from a *source* signal to (another) *destination* signal. Both signals are in the direction (up or down) of the route. A route is said to be *elementary* if there does not exist a signal which is placed between the source and the destination of the route and which has the same direction as the route.

An *interlocking table* specifies the elementary routes in a given network and the specification for setting these routes. A specification of a route includes: (i) the list of the sections in the *path* from the source to the destination, (ii) the list of the sections used as the *overlap*, (iii) a map from *points* used by the route to their required positions (`PLUS` or `MINUS`), (iv) a set of *protecting signals*, and (v) a set of routes that are in *conflict* with the route, and therefore must not be set at the same time as the given route.

An interlocking table is naturally represented as a map from the id of each route into a record containing the specification of the route.

```
InterlockingTable = RouteId m→ Route,
Route ::
  source : MbId
  dest : MbId
  path : SecId*
  overlap : SecId*
  points : SecId m→ PointPos
  signals : MbId-set
  conflicts : RouteId-set,
RouteId = Text,
PointPos == PLUS | MINUS
```

4.2 Static Semantics

The static semantics of the DSL presented in Section 4.1 are specified by predicates in RSL. An interlocking configuration data specified in ICL is wellformed if it satisfies

[‡]The term *virtual signal* is abbreviated to *signal* in the remaining of the dissertation.

all of the following.

I-01 Its network layout is wellformed.

I-02 Its interlocking table is wellformed w.r.t. its network layout.

```
is_wf : Interlocking → Bool
is_wf(ixl) ≡
  let n = track_layout(ixl), rt = interlocking_table(ixl) in
    /* I−01) network layout is wellformed */
    L.is_wf(n) ∧
    /* I−02) interlocking table is wellformed w.r.t. the network */
    is_wf_rt(rt, n) end
```

The conditions to be wellformed for network layouts and interlocking tables are explained in detail in subsequent subsections.

4.2.1 Railway Network Layouts

A network layout *n* is wellformed if all of the following requirements are fulfilled.

N-01 All *n*'s elements – linear sections, points, marker boards – have unique identifiers.

N-02 All linear sections in *n* are wellformed.

N-03 All points in *n* are wellformed.

N-04 All marker boards in *n* are wellformed.

N-05 Orientation is consistent in the network layout. This checks that starting from any border section, we must always go through linear sections in a consistent direction (up or down).

N-06 *n* has to be cycle-free (optional).

N-07 *n* has to satisfy assumptions about the boundary configuration (optional). Boundary configuration assumptions are discussed in further detail in subsequent paragraphs.

The following predicate in RSL checks if a network layout is wellformed.

```
is_wf : NetworkLayout → Bool
is_wf(n) ≡
  let ls = linears(n), ps = points(n), ms = marker_boards(n) in
    /* N−01) all elements have unique identifiers */
    unique_identifiers(n) ∧
    /* N−02) all linears are wellformed */
    (∀i : SecId • i ∈ ls ⇒ is_wf_l(i, n)) ∧
```

```
/* N−03) all points are wellformed */
(∀i : SecId • i ∈ ps ⇒ is_wf_p(i, n)) ∧
/* N−04) all marker boards are wellformed */
(∀i : MbId • i ∈ ms ⇒ is_wf_m(i, n)) ∧
/* N−05) the orientation is consistent */
orientation_is_correct (n) ∧
/* N−06) cycle−free ( optional ) */
no_cycles(n) ∧
/* N−07) boundary configuration assumption */
boundary_configuration(n) end
```

The wellformedness conditions for each kind of element are described in the subsequent paragraphs.

Linear Sections. A linear section *l* is wellformed if it satisfies all of the following requirements.

L-01 No self-neighbouring, i.e., *l* is not a neighbour of itself.

L-02 The linear section is not isolated, i.e., it must have at least a neighbour, and has at most two neighbours and its neighbours have to be distinct.

L-03 All neighbours exist and the neighbouring relationship is mutual, i.e., if *t* is a neighbour of *l*, then *l* must also be a neighbour of *t*.

L-04 The section has maximum two marker boards installed along it, one per direction (up, down), and they must be distinct (this is implied from the wellformedness of marker boards, cf. **M-02**).

L-05 The length of the section has to be at least the minimum length as required by engineering rules [Ban12]. This minimum length is specified by the constant MIN_SECTION_LENGTH in the RSL specification. This requirement is optional.

```
is_wf_l : SecId × NetworkLayout ~→ Bool
is_wf_l(i , n) ≡
  let l = get_linear(i , n), nbs = neighbors(l) in
    /* L−01) no self−neighboring */
    (i ∉ rng nbs) ∧
    /* L−02) 1 <= no. neighbors <= 2 and distinct */
    (card dom nbs ≥ 1) ∧ (card dom nbs = card rng nbs) ∧
    /* L−03) all neighbors exist and are mutual neighboring */
    (∀j : SecId •
        j ∈ rng nbs ⇒ s_exists(j , n) ∧ i ∈ get_neighbors(j, n)) ∧
    /* L−04) all signals are distinct and no. of signals <= 2 (i.e.
    max. one signal per direction , and they must be distinct ) −> implied
    from is_wf_m, thus dont need to check here */
    /* L−05) length is greater than minimum */
```

```
    (length(l) > MIN_SECTION_LENGTH) end
pre l_exists(i, n)
```

Points. A point section p is wellformed if it satisfies all of the following requirements.

P-01 No self-neighbouring, i.e., p is not a neighbour of itself.

P-02 A point must have three neighbours[§].

P-03 All neighbours of p are distinct.

P-04 Similarly to linear sections, all neighbours of a point section must exist, and the neighbouring relationship is mutual.

P-05 The length of the section has to be at least the minimum length as required by engineering rules [Ban12]. This requirement is optional.

```
is_wf_p : SecId × NetworkLayout ~→ Bool
is_wf_p(i, n) ≡
  let p = get_point(i, n), nbs = neighbors(p) in
    /* P−01) no self−neighboring */
    (i ∉ rng nbs) ∧
    /* P−02) no. neighbors == 3 (no border point) */
    (card dom nbs = 3) ∧
    /* P−03) neighbors are distinct */
    (card rng nbs = 3) ∧
    /* P−04) all neighbors exist and are mutual neighboring */
    (∀j : SecId •
       j ∈ rng nbs ⇒ s_exists(j, n) ∧ i ∈ get_neighbors(j, n)) ∧
    /* P−05) length is greater than minimum */
    (length(p) > MIN_SECTION_LENGTH) end
pre p_exists(i, n)
```

Marker Boards. Marker boards can be installed both along linear and point sections. However, for simplicity, we put a restriction that marker boards can only be installed along linear sections. A marker board installed along a point section can be converted to satisfy this restriction by adding an artificial linear section and moving the point there. A marker board m is wellformed if all of the following hold.

M-01 The section where the marker board installed along exists and is a linear section.

[§]For simplicity, we only allow linear sections to be at the border of the network, hence a point section has always three neighbours. If a point section is at the border, an artificial linear section can be added to ensure this property

M-02 There does not exist another marker board that is installed along the same the section and direction where *m* is installed.

M-03 The distance from the location where *m* is installed to the tip of the section that *m* is installed along in the travel direction that *m* is intended for has to be less than the length of the section itself.

```
is_wf_m : MbId × NetworkLayout ~→ Bool
is_wf_m(i, n) ≡
  let m = get_maker_board(i, n), si = section(m), d = dir(m) in
    /* M−01) the section exists and is a linear */
    l_exists (si , n) ∧
    /* M−02) there does not exist another marker board that is installed
    along the same section in the same direction */
    ¬ (∃j : MbId •
          j ∈ marker_boards(n) \ {i} ∧
          let
            m′ = get_maker_board(j, n), si′ = section(m′), d′ = dir(m′)
          in
            si′ = si ∧ d′ = d
          end) ∧
    /* M−03) distance is less than the section′s length */
    let l = get_linear(si , n) in distance(m) < length(l) end end
pre m_exists(i , n)
```

Boundary Configuration Assumptions. At the border of an interlocking system, the virtual signals and their associated marker boards are configured as shown in Figure 4.2. For example, for IXL1, signal `mb21` controls the entry movement from IXL2 to IXL1, while `mb12` controls the exiting movement from IXL1 to IXL2. The roles of `mb12` and `mb21` are exchanged for IXL2. Signal `mb12` is controlled by IXL2, while `mb21` is controlled by IXL1. For borders between an interlocked area and a non-interlocked area (e.g., a shunting area) the layout is similar, even though there are more details (e.g., both virtual signals are controlled by the interlocking and operations by shunting staff is involved as well). For borders between an ETCS interlocking and an external (non-ETCS) legacy interlocking, the configuration is more complicated. In our work, we assume that all layout has the configuration as shown in Figure 4.2 at their boundary sections.

```
boundary_configuration : NetworkLayout → Bool
boundary_configuration(n) ≡
 let bs = get_boundaries(n) in
    (∀i : SecId • i ∈ bs ⇒ boundary_configuration_l(i, n))
  end

boundary_configuration_l : SecId × NetworkLayout → Bool
```

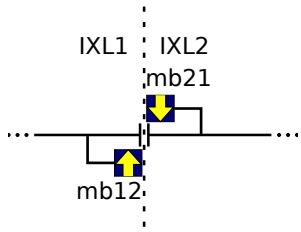


Figure 4.2: Configuration of marker boards at the boundary

```
boundary_configuration_l(i, n) ≡
  let l = get_linear(i, n), nbs = neighbors(l), j = hd rng nbs in
    l_exists (j, n) ∧
    if UP ∈ nbs
    then dom signals(i, n) = {UP} ∧ DOWN ∈ signals(j, n)
    else dom signals(i, n) = {DOWN} ∧ UP ∈ signals(j, n)
    end
  end
pre is_boundary(i, n)
```

Train movement across an interlocking border is handled transparently w.r.t. the boundary between two interlockings. Route setting, signalling and route release are done as within one interlocking, fulfilling the same functionality with a few simplifications. This is done by means of messages exchanged between the two interlockings (the protocol is an internal protocol part of the interlocking software). Such message exchange allows IXL1 to change the aspect of `mb12` although `mb12` is not controlled by IXL1, i.e., IXL1 can change the aspect of `mb12` by delegating the task to IXL2. For example, IXL1 can ask IXL2 to set `mb12` to CLOSED aspect in order to protect a route begins from `mb21` and goes into IXL1. At the abstract level, we can assume that IXL1 can control `mb12` as it can control `mb21` or any other virtual signals under its control. Other movements e.g., shunting movement is more simplified.

4.2.2 Interlocking Tables

An interlocking table *tb* is wellformed w.r.t. a network layout *n* if it satisfies all of the following.

T-01 Route identifiers are unique and differ from the identifiers of elements in *n*.

T-02 All route specifications are distinct.

T-03 All route specifications are wellformed.

T-04 A route is not conflicting with itself.

T-05 Conflicting is mutual: if r is in conflict with r' then r' is also in conflict with r.

T-06 All pairs of routes that are physically in conflict must be specified as conflicting in *tb*. This check is described in detail in Section 4.4.

```
is_wf_rt : InterlockingTable × L.NetworkLayout → Bool
is_wf_rt(tb, n) ≡
  /* T−01) routes identifiers are unique (ensure by the map) and differs
  from identifiers of elements in network layout */
  let
    js =
      (dom tb) ∩
     (dom L.linears(n) ∪ dom L.points(n) ∪
      dom L.marker_boards(n))
  in
    js = {}
  end ∧
  /* T−02) routes are distinct */
  (card dom tb = card rng tb) ∧
  /* T−03) all routes are wellformed */
  routes_are_wellformed(tb, n) ∧
  /* T−04) no self conflicting */
  no_self_conflicting (tb) ∧
  /* T−05) conflicts are mutual */
  conflicts_are_mutual(tb) ∧
  /* T−06) conflicting routes information is correct */
  conflicts_are_correct (tb, n) pre L.is_wf(n)
```

Routes. A route *r* is wellformed w.r.t. a network layout *n* if the following conditions hold.

R-01 The source signal and destination signal exist in *n* and their are intended for the same travel direction as *r*'s direction.

R-02 All protecting signals used by *r* exist in *n*, and the set of protecting signals does not contain the source and destination signals.

R-03 All points used by the route exist in *n*.

R-04 All elements in the route's path and overlap exist in *n*.

R-05 All points in the route's path and overlap must be listed with the same required positions in the map of required points $points(r)$.

R-06 The route's path must have minimum length of one (section).

R-07 The safety distance must be sufficient according engineering rules defined based on safety regulations. The safety distance is the total length of the overlap plus the distance from the destination signal to the tip of the section along

which it is installed. In Denmark, the safety distance must be at least 50m according to [Ban12], otherwise the last section in *r*'s path is a boundary section.

R-08 The source signal has to be installed in the same direction as *r* along the section preceding the first section of *r*'s path.

R-09 The destination signal has to be installed in the same direction as *r* along the last section of *r*'s path.

R-10 The route has to be an elementary route, i.e., there are no signals in the same direction as the route direction in between the source and destination signals.

R-11 The route's *full path*, i.e., including path and overlap, must be connected and acyclic.

R-12 The route must not go through a point via its plus-minus.

R-13 The route must have proper protection to prevent interfering traffic. The route protection needed for the route is automatically calculated from the network layout. Then we check if the route protection provided by the route specification in the interlocking table covers the calculated route protection. Section 4.3 explains the route protection calculation and coverage check in detail.

```
is_wf_r : Route × L.NetworkLayout → Bool
is_wf_r(r, n) ≡
  /* R−01) source and destination signals exist and agree on their
   direction */
  signals_exist_and_agree(r, n) ∧
  /* R−02) protecting signals exist , do not contain source and destination
  signals */
  protecting_signals_exist(r, n) ∧
  /* R−03) all points exist */
  points_exist(r, n) ∧
  /* R−04) all elements in the path and overlap exist */
  elems_in_path_and_ovs_exist(r, n) ∧
  /* R−05−−12) conditions on the path*/
  route_path_cnd(r, n) ∧
  /* R−13) the route has proper protection */
  has_proper_protection(r, n) pre L.is_wf(n)
```

4.3 Automatic Checking of Route Protection

This section describes how the route protection for a route can automatically be calculated from the network layout. Afterwards, it explains how the check for condition **R-13** in Section 4.2.2 is automatically performed. The check is formally specified in RSL function `has_proper_protection` in Appendix A.

4.3.1 Preliminaries

This section introduces briefly route protection and some preliminary concepts which are used in subsequent sections to explain the route protection calculation and checking process.

Route Protection. The purpose of route protection is to prevent interfering traffic from entering the route. The interfering traffic would collide with the traffic in the route, or derail due to a point is required to be in different positions by the interfering traffic and the traffic in the route.

There are two kinds of protection: flank protection and front protection [TVA09]. Flank protection prevent traffic from entering on the side of the route, while front protection prevent the incoming traffic from entering the route. The protection can be done using protecting signals and/or protecting points. In the former, protecting signals are set to CLOSED aspect, disallowing trains from entering the route. In the latter, protecting points are switched to a position that diverts traffic away from the route.

Figure 4.3 shows an example of flank protection using protecting points. The point $t20$ is switched to PLUS (straight) position, diverting traffic away from the route r. Figure 4.4 shows an example where flank protection for the route r is provided by

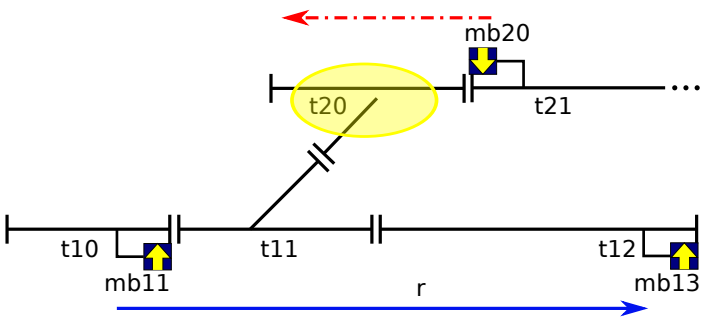


Figure 4.3: Flank protection by protecting point

the signal $mb20$ by setting $mb20$ to CLOSED aspect.

Figure 4.5 and Figure 4.6 show examples where front protection for route r is provided by a protecting point $t12$ or a protecting signal $mb12$, respectively.

Protection Transfer. Protection can be transferred from a protecting point to a protecting signal for operational reasons [TVA09]. Let us take an example in Figure 4.7 to illustrate protection transfer.

Figure 4.7 shows how protection is transferred from protecting point $t20$ to a protecting signal $mb20$. In order to prevent traffic from $t21$ and $t31$ from interfering with traffic in $r1$, $t20$ is required to PLUS (straight) position to divert the interfering

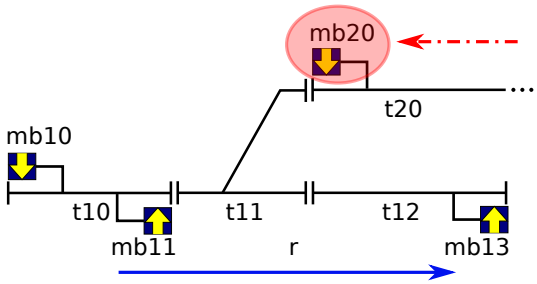


Figure 4.4: Flank protection by protecting signal

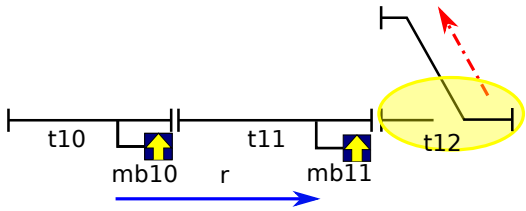


Figure 4.5: Front protection by protecting point

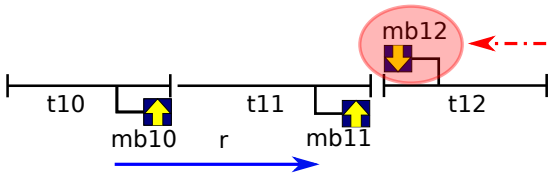


Figure 4.6: Front protection by protecting signal

traffic away. Similarly, $t20$ is required to be in MINUS (siding) position to divert interfering traffic away from route $r2$. Although $t20$ is the closest protection available for $r1$ and $r2$ in this case, there are many drawbacks in using $t20$ as protecting point:

(a) Due to different required position for $t20$, $r1$ and $r2$ are in conflict (cf. Section 4.4). Therefore, they cannot be set at the same time, even though that is clearly possible as seen in Figure 4.7.

(b) If the majority of traffic passes through $r1$ and $r2$, i.e., two tracks on the sides, and little traffic goes through the middle track ($t20, t21$), then the point $t20$ will constantly be switched from one position to another without actually having traffic passing through it. This results in extra maintenance for $t20$ and delay in setting $r1$ and $r2$.

In order to overcome these drawbacks, protection for $r1$ and $r2$ is transferred to the further protecting signal $mb20$. Consequently, both $r1$ and $r2$ can be set at the same time, and no constant switching of $t20$ is needed.

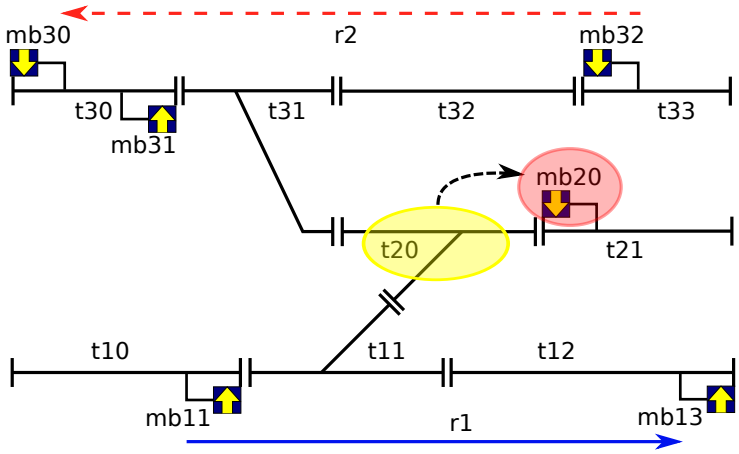


Figure 4.7: Protection transfer illustration

4.3.2 Protection Suites

A *protection suite* is a set of protecting signals and a map from protecting points to their required positions.

ProtectionSuite ::
  signals : MbId-**set**
  points : SecId $\underset{m}{\rightarrow}$ PointPos

For a given route r, the following function returns the associated protection suite.

protection : Route $\rightarrow$ ProtectionSuite
protection(r) $\equiv$
  mk_ProtectionSuite(signals(r), points(r) $\setminus$ **elems** path(r))

We defined the following operations and relations on protection suites.

(1) *Covered by*: A protection suite sa is said to be *covered by* another protection suite sb, denoted as $sa \subseteq sb$, if

  (a) sa's protecting signals are in the set of sb's protecting signals, and

  (b) all the protecting points in sa are also in sb, and they are required to be in the same positions in both sa and sb.

```
⊆ : ProtectionSuite × ProtectionSuite → Bool
sa ⊆ sb ≡
  signals(sa) ⊆ signals(sb) ∧
   let psa = points(sa), psb = points(sb) in
     (∀i : SecId • i ∈ psa ⇒ i ∈ psb ∧ psb(i) = psa(i))
   end
```

(2) *Conflicting*: Two protection suites *sa* and *sb* are said to be *conflicting*, denoted as *sa # sb*, if there exists a point *p* that is in both *sa* and *sb*, and *sa* and *sb* require *p* to be in different positions.

```
# : ProtectionSuite × ProtectionSuite → Bool
sa # sb ≡
  let psa = points(sa), psb = points(sb), cs = dom psa ∩ dom psb in
    (∃p : SecId • p ∈ cs ∧ psa(p) ≠ psb(p))
  end
```

(3) *Union*: The union of two protection suites *sa* and *sb*, denoted as *sa* ∪ *sb*, is a protection suite that contains the protecting signals and protecting points from both *sa* and *sb*. Note that it is a precondition that *sa* and *sb* are not conflicting, i.e., ¬ (sa # sb).

```
union : ProtectionSuite × ProtectionSuite → ProtectionSuite
sa ∪ sb ≡
  let
    sigs = signals(sa) ∪ signals(sb),
    psa = points(sa),
    psb = points(sb),
    cs = dom psa ∩ dom psb,
    ps =
      (psa \ cs) ∪
      [i ↦ psa(i) | i : SecId • i ∈ psa ∧ i ∉ cs] ∪
      [i ↦ psb(i) | i : SecId • i ∈ psb ∧ i ∉ cs]
  in
    mk_ProtectionSuite(sigs, ps)
  end
pre ¬ (sa # sb)
```

(4) *Subtraction*: The subtraction of a protection suite *sa* by a protection suite *sb*, denoted as *sa* \ *sb*, is specified as in the following.

```
\ : ProtectionSuite × ProtectionSuite → ProtectionSuite
sa \ sb ≡
  let
    sigs = signals(sa) \ signals(sb),
    psa = points(sa),
    psb = points(sb),
    ps =
```

```
    [i ↦ psa(i) |
     i : SecId • i ∈ psa ∧ (i ∉ psb ∨ psa(i) ≠ psb(i))]
in
  mk_ProtectionSuite(sigs, ps)
end
```

Basically, $sa \setminus sb$ returns a new protection suite that comprises the protecting signals in *sa* but not in *sb*, and points in *sa* that are not in *sb*, or are required to be in different positions than the ones required by *sb*. The subtraction is used to calculate an alternative protection suite in which protecting points are replaced by protecting signals.

4.3.3 Protection Calculation

Algorithm 1 shows the pseudo-code for the algorithm to find a protection suite that protects a section *i* from the traffic coming from a neighbouring section *j* in a network layout *n*. There are two cases:

(1) *j* is a linear section: if *j* has a signal *m* in the direction toward *i*, then a protection suite that has *m* as a protecting signal, and no protecting points is returned. Otherwise, *j* cannot provide protection for *i*, hence we continue looking for further protection, i.e., the protection for *j* from the section $k \neq i$ which is another neighbour of *j*. If we reach the border of the network before we find a protection suite, then an empty protection suite is returned.

(2) *j* is a point section: if *i* is plus (minus) neighbour of *j*, then a protecting suite that has no protecting signals and $[j \mapsto MINUS]$ ($[j \mapsto PLUS]$) as a protecting point is returned. On the other hand, if *i* is the stem neighbour of *j*, then *j* cannot provide protection for *i*, hence we continue looking for further protection, i.e., the protection for *j* from both *j*'s plus and minus neighbours. If these protection suites are found, the union of them will be returned as a protection suite for *i*.

The algorithm is formally specified in RSL by the function `find_protection` .

4.3.4 Protection Transfer Calculation

Algorithm 2 shows how protection transfer is calculated. Basically, for a given protecting point *p* of a route *r*, the algorithm calculates a protection suite *s* that protect *p* from traffic that is not coming from *r*. Since *p* is a protecting point of *r*, *p* can only be connected to *r* via its plus or minus end. Therefore, if *p* is connected to *r* via its plus (minus) end, then *s* is a protection suite protecting *p* from traffic from its stem and minus (plus) ends. If *s* is not empty and contains only protecting signals and no protecting points, then these signals can alternatively be used in the place of *p* to protect *r*. On the other hand, if *s* is empty or contains protecting points, then no protection transfer is possible for *p*. The algorithm is formally specified in RSL by the function `find_replacing_signals` .

Algorithm 1 Finding a protection suite for i to prevent traffic coming from j

```
function FIND_PROTECTION(i, j, n)
    if j is a linear section then
        if j has a signal m in the direction toward i then
            return ({m}, [])                                   ▷ m as a protecting signal
        else if j is not a boundary section then
            return FIND_PROTECTION(j, k, n) where k ∈ neighbors(j, n) ∧ k ≠ i
        else                                                   ▷ j is a boundary section
            return ({}, [])                                    ▷ an empty protecting suite
        end if
    else                                                       ▷ j is a point
        if i = plus(j, n) then
            return ({}, [j ↦ MINUS])                           ▷ j as a protecting point
        else if i = minus(j, n) then
            return ({}, [j ↦ PLUS])                            ▷ j as a protecting point
        else                                                   ▷ i = stem(j, n)
            FIND_PROTECTION(j, plus(j, n), n) ∪
            FIND_PROTECTION(j, minus(j, n), n)                 ▷ union two protection suites
        end if
    end if
end function
```

Algorithm 2 Calculate protection transfer for a point p

```
function FIND_REPLACING_SIGNALS(p, r, n)
    ps ← FIND_PROTECTION(p, stem(p, n), n)
    if j is connected to r via plus(p, n) then
        os ← FIND_PROTECTION(p, minus(p, n), n)
    else                                                       ▷ j is connected to r via minus(p, n)
        os ← FIND_PROTECTION(p, plus(p, n), n)
    end if
    if ps † os then                                            ▷ two protection suites are conflicting
        return {}                                              ▷ transfer is not possible
    end if
    s ← ps ∪ os                                                ▷ union two protection suites
    if points(s) = [] ∧ signals(s) ≠ {} then
        return signals(s)
    else
        return {}                                              ▷ transfer is not possible
    end if
end function
```

4.3.5 Route Protection Check

A given route r is protected properly by its protection suite specified in the interlocking table if all of the following hold.

PC-01 All the points in the full path of r, including its path and overlap, have flank protection. We do not need flank protection for linear sections as no traffic can enter the route from the side at a linear section.

PC-02 The last section of the full path of r has front protection to protect them from incoming traffic.

PC-03 All the signals installed along the sections in the full path of r in the opposite direction of r have to be in r's set of protecting signals.

It is trivial to check **PC-03**. The check for **PC-01** and **PC-02** are performed by the Algorithm 3. Let us assume that the section that needs to be protected is i, then the main steps of the algorithm are listed in the following.

(1) The protection suite $s_r = \texttt{protection}(r)$ is calculated from the specification of r in the interlocking table.

(2) A preliminary protection suite s for i is calculated from the network layout using Algorithm 1.

(3) If s is found, and $s \subseteq s_r$ then i is properly protected by the specification of r in the interlocking table.

(4) If there exists a protecting signal in s that is not in s_r, then i is not protected properly by the specification of r in the interlocking table.

(5) An alternative protecting suite s' is obtained by replacing the points that are in s but are not in s_r by their corresponding protecting signals as described in Algorithm 4. If $s' \subseteq s_r$, then i is properly protected by the specification of r in the interlocking table. Otherwise, i is not protected properly by the specification of r in the interlocking table. The check is finished.

4.4 Automatic Checking of Conflicting Routes

This section explains how condition **T-06** in Section 4.2.2 can automatically be performed. The check is formally specified in RSL function `conflicts_are_correct` in Appendix A. The following rules are used to determine whether two routes are in *conflict* from the network layout and compared with the route specifications.

CR-01 If two routes share one or more detection sections in their paths or overlap then they are in conflict.

Algorithm 3 Check if i is properly protected from traffic coming from j by r's specification

```
1:  function HAS_PROPER_PROTECTION(r, i, j, n)
2:      s_r ← protection(r)
3:      s ← FIND_PROTECTION(i, j, n)
4:      if s ⊆ s_r then                                  ▷ s is covered by s_r
5:          return true
6:      end if
7:      d ← (s \ s_r)                                    ▷ subtraction s by s_r
8:      if d = empty ∨ signals(d) ≠ {} then
9:          return false
10:     end if
11:     s' = FIND_ALT(s, points(d), r, n))               ▷ find an alternative suite s'
12:     return s' ≠ empty ∧ s' ⊆ s_r
13: end function
```

Algorithm 4 Find an alternative protection suite for s by replacing the points in ps with their transfer protecting signals

```
1:  function FIND_ALT(s, ps, r, n)
2:      ps' ← points(s) \ ps                             ▷ remove ps from s
3:      sigs' ← signals(s)
4:      for p ∈ ps do
5:          sigs ← FIND_REPLACING_SIGNALS(p, r, n)
6:          if sigs = {} then                            ▷ p cannot be transferred
7:              return empty
8:          else
9:              sigs' ← sigs' ∪ sigs
10:         end if
11:     end for
12:     return (sigs, ps')                               ▷ return the alternative protection suite
13: end function
```

CR-02 An exception to **CR-01**: two routes *r1*, *r2* are not in conflict if *all* of the following hold.

- *r1* and *r2* are two *concatenated routes*, i.e., the destination signal of *r1* is the source signal of *r2*, or the source signal of *r1* is the destination signal of *r2*, as shown in Figure 4.8. In such cases, *r2* is referred as a *next route* of *r1*, while *r1* is referred as a *previous route* of *r2*.
- *r1* and *r2* do not share any detection section in their paths[¶].

CR-03 If two routes share a point, they are in conflict, with the following exception.

[¶]Usually, concatenated routes do not share any detection in their paths, unless two routes form a cycle in the network.

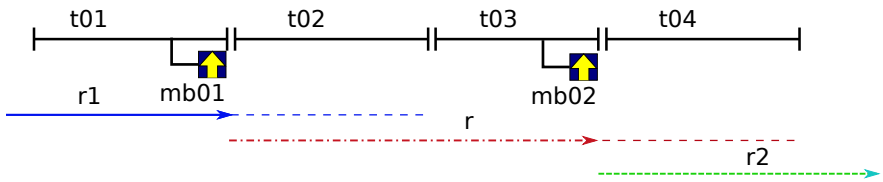


Figure 4.8: *Concatenated routes*: *r1* and *r2* are a *previous route* and a *next route* of *r*, respectively

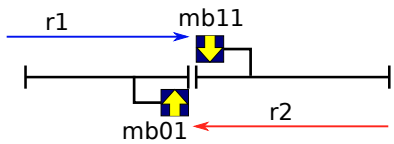


Figure 4.9: Two opposing routes *r*1 amd *r*2 that do not have overlap, and their destination signals are back-to-back

- If two routes require the shared point to be in the same position, and the shared point is a protecting point of *at least one* of the two routes (recall that the points used by a routes include points in the route's path, and protecting points which are outside of the route's path), then two routes are not in conflict.

CR-04 If two routes are in opposite directions, and their destination signals are back-to-back, then we have two cases:

(a) If *at least one* of the two routes has an overlap, then the routes share at least a detection section in their paths or overlap. Thus, they are in conflict according to **CR-01**.

(b) If *both* of the two routes have no overlap as depicted in Figure 4.9. In such case, the distances from the destination signals (e.g., *mb10* or *mb11* in Figure 4.9) to the joint of two detection sections shall be enough for the train to stop before the joint of two detection sections when it might accidentally overrun the destination signal. Therefore, it is not unsafe to set two routes *r1* and *r2* together, meaning that *r1* and *r2* are not in conflict[‖].

[‖] However, in practice, using both routes at the same time may cause a deadlock situation where neither of two trains can move forwards, at least one of them have to change direction, or reverse. Nevertheless, such two routes might be useful in some cases, e.g., in a station in which the track at a long platform is divided in two sections that can be occupied by two short trains, in order to accommodate more incoming trains.

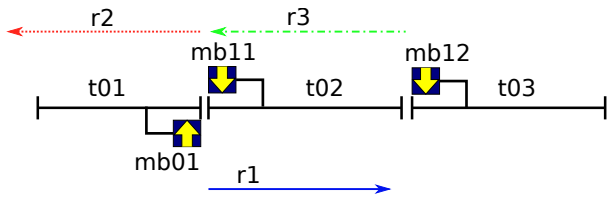


Figure 4.10: Two opposing routes *r*1 amd *r*2 whose source signals are back-to-back

CR-05 Two routes are in opposite directions, and their source signals are back-to-back as two routes *r1* and *r2* in Figure 4.10. In such cases, allowing them to be set at the same time will not result in any danger due to the restriction posed by other rules. Let us consider all possible cases:

(a) If a train is occupying *t02* and requesting *r2* to be set, then *r1* will not be set, because *t02* is not vacant.

(b) If *r2* is requested to be set for a train that is on *t03* then the train can approach *r2* only if *r3* is set. Since *r1* and *r3* are in conflict, due to **CR-01**, they will not be set together. Thus, we have two sub-cases:

- If *r3* is set then *r1* cannot be set.
- If *r1* is set then *r3* cannot be set; the train cannot approach *r2*, hence there is no danger if *r2* and *r1* are set together.

Thus, two routes in such situations are not in conflict.

CR-06 *Implicit conflicting routes*: In practice, two routes may be marked as conflicting routes due to operational reasons even though they are not really in conflict in the network track layout, based on the above rules. These routes are referred as *implicitly conflicting routes*. Implicitly conflicting routes cannot be calculated from the network layout.

The conditions for two routes to be in conflict are specified formally in RSL as follows

```
are_physically_in_conflict : Route × Route × L.NetworkLayout ~→ Bool
are_physically_in_conflict(r1, r2, n) ≡
  let
    path1 = elems path(r1),
    path2 = elems path(r2),
    ovs1 = elems overlap(r1),
    ovs2 = elems overlap(r2),
    secs1 = path1 ∪ ovs1,
    secs2 = path2 ∪ ovs2,
    ps1 = points(r1),
    ps2 = points(r2),
    protecting_points1 = dom ps1 \ secs1,
```

```
    protecting_points2 = dom ps2 \ secs2,
    shared_pps =
      (protecting_points1 ∩ dom ps2) ∪
     (protecting_points2 ∩ dom ps1)
  in
    /* not consecutive routes, overlapping paths or overlap */
    (¬ are_concatenated_routes(r1, r2, n) ∧ secs1 ∩ secs2 ≠ {}) ∨
    /* share a point, the shared point is a protecting point for at
    least one of the routes and two routes require the points in different
    positions */
    (∃i : SecId • i ∈ shared_pps ∧ ps1(i) ≠ ps2(i)) ∨
    /* entry signal of one route is a protecting
    * signal of the other route */
    (source(r1) ∈ signals(r2)) ∨ (source(r2) ∈ signals(r1))
  end
pre L.is_wf(n) ∧ is_wf_r(r1, n) ∧ is_wf_r(r2, n)

are_concatenated_routes : Route × Route × L.NetworkLayout ~→ Bool
are_concatenated_routes(r1, r2, n) ≡
  source(r1) = dest(r2) ∨ dest(r2) = source(r1)
pre L.is_wf(n) ∧ is_wf_r(r1, n) ∧ is_wf_r(r2, n)
```

4.5 Interlocking Table Generation

Given a wellformed network layout n, the ITG generates an interlocking table with all the possible routes that can be derived from the network layout. The generation is performed in two phases.

ITG-01 We construct a preliminary interlocking table containing all routes with their closest protection suite, i.e., protection transfer is not considered.

ITG-02 For each route r in the preliminary interlocking table constructed from **ITG-01**, we generate all alternative routes by transferring a subset of r's protecting points to protecting signals. In the end, we obtain an interlocking table with all possible routes in the given network layout.

These two phases are explained in detail in Section 4.5.1 and Section 4.5.2, respectively. The generation is formally specified by RSL function `mk_table` in Appendix A.

4.5.1 Preliminary Interlocking Table Construction

The basic idea of the algorithm for constructing the preliminary interlocking table is that for each marker board s in the given network layout n, we construct the specification of routes starting from s by performing a depth first search starting

with the first section right after s. At each section i in n that we traverse during the search, we collect the specification of the route related to that section. The collecting procedure depends on whether i is a linear section or a point as shown in Algorithm 5. The subsequent paragraphs explain the collecting procedure for linear sections and points sections, respectively. Preliminary interlocking construction is formally specified in RSL function `gen_routes` in Appendix A.

Algorithm 5 Collect specifications for a route r at a section i

```
1: function COLLECT_ROUTE(i, r, n)
2:     if i is a linear section then
3:         return COLLECT_ROUTE_ON_LINEAR(i, r, n)
4:     else                                                    ▷ i is a point
5:         return COLLECT_ROUTE_ON_POINT(i, r, n)
6:     end if
7: end function
```

Linear Sections. The collecting procedure on a linear section l is shown in Algorithm 6. If the destination signal of r has not been found, then we add l to the path, otherwise we add l to the overlap. If l has a signal *sig* in the same direction as s and the destination signal of r has not been found, we mark *sig* as destination signal. Signals installed along l in the opposite direction of r are added to the set of r's protecting signals. If the destination signal of r has been found and the safety distance is sufficient as specified in Section 4.2, then the route specification is returned. Otherwise we start a new iteration with the neighbour j of i which is not in r's path or overlap.

Points. The collecting procedure on a point p is shown in Algorithm 7. If the destination signal of r has not been found, then we add p to the path, otherwise we add p to the overlap. We have the following cases.

(1) p is connected to r's path by its plus (minus): we add $[i \mapsto PLUS]$ ($[i \mapsto MINUS]$) to $\texttt{points}(r)$. Using the procedure described in Section 4.3, we calculate the protection suite s_m (s_p) for p from its minus (plus) neighbour, and merge it with the r's protection suite. If the destination signal of r has been found, and the safety distance is sufficient, then we merge the protection suite s_s for p from its stem neighbour (front protection) and return the route specification. Otherwise, we start a new iteration with the stem neighbour of p.

(2) p is connected to the current route path by its stem: we add $[i \mapsto PLUS]$ ($[i \mapsto MINUS]$) to $\texttt{points}(r)$ and merge protection suites s_m (s_p) and s_s for p from its minus (plus) and stem neighbours, respectively, to r's protection suite. Then we start a new iteration with p's plus (minus) neighbour. Then we return the concatenated list of the routes returned by these two iterations (one for plus and one for minus).

Algorithm 6 Collect specification for a route r at a linear section l

```
function COLLECT_ROUTE_ON_LINEAR(l, r, n)
    sig ← the signal installed along l in the direction of r
    osig ← the signal installed along l in the opposite direction of r
    if osig exists then
        signals(r) ← signals(r) ∪ {osig}                    ▷ add osig as a protecting signal
    end if
    if dest(r) found then
        overlap(r) ← overlap(r) ⌢ ⟨l⟩
    else
        path(r) ← path(r) ⌢ ⟨l⟩
        if sig exists then
            dest(r) ← sig
        end if
    end if
    if l is a boundary section then
        if dest(r) assigned then
            return ⟨r⟩
        else
            return                                          ▷ ignore, no specification found
        end if
    else
        j ← l's neighbour where j ∉ path(r) ∪ overlap(r)
        dist ← distance(dest(r), n) + total length of overlap(r)        ▷ safety distance
        if dest(r) assigned and dist is sufficient then
            s ← FIND_PROTECTION(l, j, n)                    ▷ calculate front protection
            points(r) ← points(r) ∪ points(s)
            signals(r) ← signals(r) ∪ signals(s)
            return ⟨r⟩
        else
            return COLLECT_ROUTE(j, r, n)                   ▷ continue collecting
        end if
    end if
end function
```

4.5.2 Alternative Routes Generation

The procedure for generating alternative routes after obtaining the preliminary interlocking table from phase **ITG-01** is described in Algorithm 8. Alternative routes generation is formally specified in RSL function `mk_alt_routes` in Appendix A. For each route r in the preliminary interlocking table, the following procedure is performed.

(1) Find the set P_0 of r's protecting points.

(2) For each subset $P_i \subseteq P_0$, i.e., $P_i \in \mathcal{P}(P_0)$, we replace all protecting points in P_i with their protecting signals *sigs*. Then we replace P_i with *sigs* in r's protection suite to obtain an alternative route specification.

In the end, we will obtain an interlocking table with all possible routes in the given network layout.

4.6 Dynamic Semantics

The dynamic semantics of a description in ICL of interlocking configuration data describes the model of the behaviours in the form of a Kripke structure, a global property expressed in the form of a proposition that needs to be proved (conjunction of safety properties), and a number of test objectives for the interlocking system associated with the given configuration data. The denotational semantic function of ICL has the following signature.

$$[\![-]\!]^{\Sigma} : \Sigma \stackrel{\sim}{\to} \Delta \stackrel{\sim}{\to} (\mathcal{K} \times \mathit{Prop} \times \mathit{TestObj}^*)$$

where

- $[\![-]\!]^{\Sigma}$ is the semantic function for descriptions in ICL;
- Σ = Interlocking is the domain of interlocking configuration data in ICL where Interlocking is defined in Section 4.1;
- Δ is the domain of generic applications specified in IDL which will be described in detail in Chapter 5;
- $\mathcal{K}$ is the domain of Kripke structures defined in Section 2.11.1; and
- *Prop* and *TestObj* are the domains of propositions and test objectives, respectively. They will be explained further in Chapter 5.

The semantics of interlocking configuration data depends on the chosen generic applications in Δ. The semantic function $[\![-]\!]^{\Sigma}$ is defined in detail in Chapter 5.

Algorithm 7 Collect specification for a route r at a point p

```
1:  function COLLECT_ROUTE_ON_POINT(p, r, n)
2:      prev ← neighbour of p that is in r
3:      if dest(r) found then
4:          overlap(r) ← overlap(r) ⌢ ⟨p⟩
5:      else
6:          path(r) ← path(r) ⌢ ⟨p⟩
7:      end if
8:      dist ← distance(dest(r), n) + total length of overlap(r)        ▷ safety distance
9:      done ← dest(r) assigned and dist is sufficient
10:     s_p ← FIND_PROTECTION(p, plus(p, n), n)
11:     s_m ← FIND_PROTECTION(p, minus(p, n), n)
12:     s_s ← FIND_PROTECTION(p, stem(p, n), n)
13:     if prev = plus(p, n) then
14:         if done then
15:             points(r) ← points(r) ∪ points(s_s) ∪ points(s_m) ∪ [p ↦ PLUS]
16:             signals(r) ← signals(r) ∪ signals(s_s) ∪ signals(s_m)
17:             return ⟨r⟩
18:         else
19:             points(r) ← points(r) ∪ points(s_m) ∪ [p ↦ PLUS]
20:             signals(r) ← signals(r) ∪ signals(s_m)
21:             return COLLECT_ROUTE(stem(p), r, n)
22:         end if
23:     else if prev = minus(p, n) then
24:         if done then
25:             points(r) ← points(r) ∪ points(s_s) ∪ points(s_p) ∪ [p ↦ MINUS]
26:             signals(r) ← signals(r) ∪ signals(s_s) ∪ signals(s_p)
27:             return ⟨r⟩
28:         else
29:             points(r) ← points(r) ∪ points(s_p) ∪ [p ↦ MINUS]
30:             signals(r) ← signals(r) ∪ signals(s_p)
31:             return COLLECT_ROUTE(stem(p), r, n)
32:         end if
33:     else                                                            ▷ prev = stem(p, n)
34:         if done then return ⟨r⟩
35:         else
36:             r_p ← r
37:             points(r_p) ← points(r_p) ∪ points(s_m) ∪ [p ↦ PLUS]
38:             signals(r_p) ← signals(r_p) ∪ signals(s_m)
39:             r_m ← r
40:             points(r_m) ← points(r_m) ∪ points(s_p) ∪ [p ↦ MINUS]
41:             signals(r_m) ← signals(r_m) ∪ signals(s_p)
42:             return COLLECT_ROUTE(plus(p, n), r_p, n)⌢
43:                    COLLECT_ROUTE(minus(p, n), r_m, n)
44:         end if
45:     end if
46: end function
```

Algorithm 8 Make alternative route specifications for a route r

```
1:  function MK_ALT_ROUTES(r, n)
2:      s ← protection(r)                                   ▷ r's protection suite
3:      P_0 ← points(r)
4:      pool ← 𝒫(P_0)                                       ▷ powerset of points in s
5:      rs ← ⟨r⟩
6:      for P_i ∈ pool do
7:          sigs ← {}
8:          for p ∈ P_i do
9:              sigs ← sigs ∪ FIND_REPLACING_SIGNALS(p, r, n)
10:         end for
11:         r′ ← r                                          ▷ replace P_i in r with protecting signals
12:         points(r′) ← points(r′) \ P_i
13:         signals(r′) ← signals(r′) ∪ sigs
14:         rs ← rs ⌢ ⟨r′⟩
15:     end for
16:     return rs
17: end function
```

4.7 Executable RSL Specifications

The RSL specifications of ICL, its static checker, and the ITG can be executed using the RAISE `rsltc` tool [Geo08; Geo02; Geo+95; Geo04]. Executable specifications offer a number of advantages:

- A specification can be considered an early prototype which can be tested before implementing. The specification can later serve as the base for implementation.
- The specification is much more readable than the implementation. Therefore, a mistake in the specification is less likely to be overlooked than in the implementation.

As an example, it is explained in the following how the ITG can be executed. The static checker can be executed in a similar way. Let us consider an example of a typical network with two tracks as shown in Figure 2.2. This network can be specified in RSL as a value n of type NetworkLayout as shown in the following.

```
n : NetworkLayout = mk_NetworkLayout(
 ["t20" ↦ mk_Linear([UP ↦ "t13", DOWN ↦ "t11"], 100),
  "b14" ↦ mk_Linear([DOWN ↦ "t14"], 100),
  "t14" ↦ mk_Linear([UP ↦ "b14", DOWN ↦ "t13"], 100),
  "t12" ↦ mk_Linear([UP ↦ "t13", DOWN ↦ "t11"], 100),
  "t10" ↦ mk_Linear([DOWN ↦ "b10", UP ↦ "t11"], 100),
  "b10" ↦ mk_Linear([UP ↦ "t10"], 100)],
 ["t13" ↦
  mk_Point(
```

```
    [NB_MINUS ↦ "t20", NB_PLUS ↦ "t12", NB_STEM ↦ "t14"], 100),
  "t11" ↦
  mk_Point(
    [NB_MINUS ↦ "t20", NB_PLUS ↦ "t12", NB_STEM ↦ "t10"], 100)],
 ["mb21" ↦ mk_MarkerBoard("t20", UP, 50),
  "mb20" ↦ mk_MarkerBoard("t20", DOWN, 50),
  "mb15" ↦ mk_MarkerBoard("b14", DOWN, 50),
  "mb14" ↦ mk_MarkerBoard("t14", UP, 50),
  "mb13" ↦ mk_MarkerBoard("t12", UP, 50),
  "mb12" ↦ mk_MarkerBoard("t12", DOWN, 50),
  "mb11" ↦ mk_MarkerBoard("t10", DOWN, 50),
  "mb10" ↦ mk_MarkerBoard("b10", UP, 50)])
```

Executing the RSL term `mk_table(n)` results in a value of type InterlockingTable in RSL.

```
["01a" ↦ mk_Route("mb11", "mb13", ⟨"t10", t11", "t12"⟩, ⟨⟩,
            ["t13" ↦ MINUS, "t11" ↦ PLUS],
            {"mb11", "mb20", "mb12"}, {},
            {"01b", "02a", "02b", "03", "04", "05a", "05b", "06b", "07"}),
 ... content skipped ... ]
```

As it can be seen one of the generated routes has id `01a`, goes from `mb11` to `mb13` via two sections `t10,t11,t12`, and has no overlap. It requires point `t11` (on its path) to be in `PLUS` position and point `t13` (outside its path) to be in `MINUS` position (as a protecting point). The route has also `mb11, mb20, mb12` as protecting signals, and is in conflict with the routes `01b, 02a, 02b, 03, 04, 05a, 05b, 06b, 07`.

4.8 Implementation

ICL and the described ITG are implemented as a front-end of RT-Tester [Pel13; Ver15] for constructing and validating descriptions of networks and interlocking tables. Interlocking configuration data can be provided in XML format (which can be easily exported from computer-aided design tools). Errors are reported together with suggestions how to fix them, e.g., missing protecting signals, points, or conflicting routes can be suggested to be added to the table. A graphical editor for ICL has been developed as part of the master's thesis by Foldager [Fol15].

4.9 Related Work

Applications of formal methods to the railway domain have been investigated by numerous research groups. The ultimate goal is to produce methods for developing railway control systems efficiently while ensuring safety. A general overview of the trends can be found in [Fan14; FFM12; Fer+13; Fan12b].

DSLs for the railway domain have been proved to be efficient for describing interlocking data for other kinds of interlocking systems [HCD04; Mew10; Hax14;

JR14]. For example, Railway Control Systems Domain Language (RCSD) [Mew10] is a domain-specific language for specifying railway control systems. The language incorporates the domain knowledge into its semantics to validate the wellformedness of a given specification. Furthermore, Mewes proposed also a generic testing strategy that can exercise the important aspects of models in RCSD. Another DSL is proposed in [Jam14] for specifying interlocking configuration data which can then be translated to formal model in ModalCasl for verification. A more complete domain-specific framework for development and verification of railway control systems from the specification in a DSL to model generation and verification, and object code generation is presented in [Hax14]. Our work goes along the same line, but we have special focus on interlocking systems which are compatible with ETCS Level 2, e.g., we include notions such as marker boards and virtual signals instead of physical signals.

Several other research groups [Win+06; Win12; BFG05; HPK11; Cao+11; MY09; Jam+14] have also investigated interlocking systems having interlocking tables as design specifications. They also translate the interlocking tables into execution/design models which are then formally verified to satisfy high-level safety requirements. In some cases [Cao+11; MY09] that verification step is also used for data validation. However, inspired by the work in [HPK11], we follow a *4-step* approach for V&V as described in Section 3.6. The second step in this 4-step approach – *configuration data validation* – is performed by the static semantics checker (described in Section 4.2 to Section 4.4 of this chapter) in order to ensure the wellformedness of the interlocking configuration data that is used for instantiating behavioural models, safety properties, and test objectives which will be described in Chapter 6 and Chapter 7. Furthermore, in our method, there is a second DSL for specifying generic applications besides the DSL for specifying configuration data – ICL in the work presented here – as explained in Section 3.3. This second DSL would increase the readability, reduce errors, make it easy to change the generic applications, and provide different levels of abstraction suitable for different user groups. The second DSL – *Interlocking Dynamic Language (IDL)* – will be described in detail in Chapter 5.

A few ITGs have been proposed in previous research, e.g., in [MY09; Cao+11], but they have not been formally specified as our ITG. These ITGs generate tables having data similar to a subset of our data, also by traversing the given network layout. However, the ITG in [Cao+11] does not generate any data concerning flank and front protection. In [MY09], the ITG does not generate the collection of route conflicts and items for flank and front protection, but instead in a second phase (after the table generation) they employ model checking to derive these data which are then added manually. Our ITG is – to our best knowledge – the only ITG that is able provide completely automated generation of protecting points and signals directly from the network layout.

CHAPTER 5

A Domain-specific Language for Generic Interlocking Applications

This chapter specifies *Interlocking Dynamic Language (IDL)* – a DSL for specifying the generic applications of interlocking systems including generic behavioural models, safety properties, and test objectives. IDL is the ingredient **DK:b** – the *second* of the two DSLs described in Section 3.7. The first language – ICL for specifying configuration data – has been described in the previous chapter, Chapter 4. The advantages of having two different DSLs has been explained in Section 3.3.

Generic behavioural models, safety properties, and test objectives specified in IDL can be instantiated with configuration data specified in ICL presented in Chapter 4,

resulting in a concrete behavioural model of the interlocking system configured by the given configuration data, and its associated concrete safety properties and test objectives.

Unlike the specification of ICL in Chapter 4, in this chapter, the concrete syntax of IDL is presented instead of its abstract syntax as the concrete syntax of IDL is used in the next chapters, Chapter 6 and Chapter 7, to describe the generic behavioural model, generic safety properties, and generic test objectives for the case study of the forthcoming Danish interlocking systems. Furthermore, for simplicity, the denotational semantics of IDL in a RSL-like notation is given.

The remainder of the chapter is organised as follows. The syntax of IDL is described in Section 5.1. Section 5.2 presents the denotational semantics of IDL, which is then used to elaborate further the dynamic semantics of ICL that was briefly described in Section 4.6. Section 5.3 and Section 5.4 describe the implementation of IDL in our toolchain and some related work, respectively.

5.1 Syntax

This section describes the BNF grammar for the concrete syntax of IDL, and explains informally the meaning of different constructs in IDL. The syntax of IDL is inspired by a subset of RSL-SAL [PG07] and its extension in [Han15]. For simplicity, the following conventions are employed in the BNF grammar.

⟨*term-list*⟩ A term suffixed with *-list* denotes a sequence of ⟨*term*⟩ separated by a comma, i.e.,

⟨*term-list*⟩ ::= ⟨*term-list*⟩ **,** ⟨*term*⟩
| ⟨*term*⟩

⟨*term-string*⟩ A term suffixed with *-string* denotes a sequence of ⟨*term*⟩ separated by white-spaces, i.e.,

⟨*term-string*⟩ ::= ⟨*term-string*⟩ ⟨*term*⟩
| ⟨*term*⟩

⟨*[term]*⟩ A term enclosed by squared brackets denotes an optional appearance of ⟨*term*⟩, i.e.,

⟨*[term]*⟩ ::= ⟨*term*⟩ | ⟨*empty*⟩

⟨*empty*⟩ ::=

These conventions can be nested, e.g., ⟨*[term-string]*⟩ denotes an optional appearance of a sequence of ⟨*term*⟩ separated by white-spaces.

The full BNF grammar of the concrete syntax of IDL is listed in Appendix B. The subsequent subsections outline the essence of the language syntax.

5.1.1 Specification

A specification in IDL describes a generic behavioural model, generic safety properties, and generic test objectives of a product line of interlocking systems. This specification when instantiated with configuration data will result in a Kripke structure modelling the behaviours of the concrete interlocking system, a proposition expressing the concrete safety properties, and a list of concrete test objectives. This will be explained in detail in Section 5.2.

A specification in IDL has a name and consists of a sequence of declarations of different types:

- Encoding declarations
- Initial state declarations
- Module declarations
- Macro declarations
- Transition relation declarations
- Invariant declarations
- Test objective declarations

These different types of declarations will be explained in detail in subsequent subsections.

⟨*specification*⟩ ::= **kripke** ⟨*ident*⟩ ⟨*decl-string*⟩ **end**

⟨*decl*⟩ ::= ⟨*encoding-decl*⟩
| ⟨*macro-decl*⟩
| ⟨*initial-decl*⟩
| ⟨*transrel-decl*⟩
| ⟨*module-decl*⟩
| ⟨*invariant-decl*⟩
| ⟨*test-obj-decl*⟩

5.1.2 Encoding Declarations

An encoding declaration contains a list of encodings. An encoding ⟨*encoding*⟩ describes for an element of a given type (e.g., a section, a signal, or a route) which variables should be generated to represent the states of the element. The states of each element of a type ⟨*elem-type*⟩ are encoded by a number of variables. Each variable ⟨*variable*⟩ contains the following information:

(a) a symbol ⟨*symbol*⟩,

(b) a symbol type ⟨*sym-type*⟩ of input, local, or output,

(c) a target type ⟨*target-type*⟩,

(d) a range of its value from ⟨*low*⟩ to ⟨*high*⟩, and

(e) an initial value ⟨*ival*⟩.

Note that ⟨*sym-type*⟩ is used to tag a variable as an input, local, or output variable, so that we can consider a behavioural model as an IOSTS defined in Section 2.11 during test generation. This is explained further in Chapter 7. The ⟨*target-type*⟩ is bound to the primitive types supported by RT-Tester. If another bounded model checker or MBT framework is used, this shall be changed accordingly. Furthermore, ⟨*low*⟩ and ⟨*high*⟩ are also RT-Tester-specific: they help the bounded model checker running more effectively.

⟨*encoding-decl*⟩ ::= **encoding** ⟨*encoding-list*⟩

⟨*encoding*⟩ ::= ⟨*elem-type*⟩ **::** ⟨*variable-list*⟩

⟨*variable*⟩ ::= ⟨*symbol*⟩ → **[** ⟨*sym-type*⟩ **,** ⟨*target-type*⟩ **,** ⟨*ival*⟩ **,** ⟨*low*⟩ **,** ⟨*high*⟩ **]**

⟨*elem-type*⟩ ::= **Linear** | **Point** | **Section** | **Signal** | **Route**

⟨*sym-type*⟩ ::= **INPUT** | **LOCAL** | **OUTPUT**

⟨*target-type*⟩ ::= **"** ⟨*primitive-type*⟩ **"**

⟨*primitive-type*⟩ ::= int | unsigned int | long | unsigned long | long long
| unsigned long long | float | double | clock

⟨*ival*⟩ ::= ⟨*literal*⟩

⟨*low*⟩ ::= ⟨*literal*⟩

⟨*high*⟩ ::= ⟨*literal*⟩

As an example, the following encoding declaration specifies that a point p in the configuration data is encoded by the following variables:

- an input variable $p.POS$ that has initial value of 0, and its value range is $[0, 2]$;
- a local variable $p.MODE$ that has initial value of 0, and its value range is $[0, 2]$;
- an output variable $p.CMD$ that has initial value of 0, and its value range is $[0, 1]$; and

- three input variables $p.S2PM$, $p.P2S$, and $p.M2S$ that have initial value of 0, and their value range is $[0, 7]$. These three variables represent the occupancy status of the point p as explained further in Section 6.2.1.

All these variables have target type of "unsigned int".

encoding
Point::
POS → [**INPUT**,"unsigned int",0,0,2]
MODE → [**LOCAL**,"unsigned int",0,0,2]
CMD → [**OUTPUT**,"unsigned int",0,0,1]
S2PM → [**INPUT**,"unsigned int",0,0,7]
P2S → [**INPUT**,"unsigned int",0,0,7]
M2S → [**INPUT**,"unsigned int",0,0,7]

When a generic application containing this encoding declaration is instantiated with configuration data, for each point p in the configuration data, six variables $p.POS$, $p.MODE$, $p.CMD$, $p.S2PM$, $p.P2S$, and $p.M2S$ with the specified value domains will be generated in the resulting Kripke structure modelling the interlocking system associated with the given configuration data.

5.1.3 Macro Declarations

In order to facilitate the specification process, macros can be defined in a macro declaration. A macro is a shortcut for an expression. A macro has a name and zero or more parameters. If a macro has no parameters, the parentheses can be omitted from the macro definition.

⟨*macro-decl*⟩ ::= **macro** ⟨*macro-list*⟩

⟨*macro*⟩ ::= **def** ⟨*ident*⟩ **(** ⟨*[ident-list]*⟩ **)** **=** ⟨*expr*⟩
| **def** ⟨*ident*⟩ **=** ⟨*expr*⟩

For example, the macro vacant_point(p) shown in the following defines a shortcut to an expression denoting whether a point p is vacant, i.e., not occupied by a train, where $p.S2PM$, $p.P2S$, and $p.M2S$ are the variables representing the occupancy status of a point p as specified by the example encoding in Section 5.1.2.

macro
def vacant_point(p) = (p.S2PM + p.P2S + p.M2S = 0)

5.1.4 Initial State Declarations

An initial state declaration describes the proposition representing the initial states I of the resulting Kripke structure. In other words, it specifies a list of invariants that must hold in the initial states. Invariants are explained in detail in Section 5.1.7.

⟨*initial-decl*⟩ ::= **init** ⟨*invariant-list*⟩

An initial state declaration example is shown in the following. It specifies that in the initial states, all routes in the given interlocking table have to be in FREE mode.

init
[init_all_route_are_free]
(∀r : **Route** • r.MODE = FREE)

The above declaration, when instantiated with concrete configuration data, for each route in the configuration data, *r* will be replaced by the identifier of the route, resulting in a concrete invariant. The proposition representing initial states is then the conjunction of these concrete invariants.

Note that the initial state declaration is *optional* in the specification. If initial state declarations are not specified, the initial states of the target Kripke structure will consist of a single state where all the variables are set to their corresponding initial value as specified in the encoding declarations. On the other hand, if an initial state declaration is specified, then it will take the precedence, and the initial values in the encoding declaration will be ignored. If multiple initial state declarations are specified, the initial states of the target Kripke structure will be represented by the conjunction of all the propositions.

5.1.5 Transition Relation Declarations

There must be *exactly one* transition relation declaration in the specification. Otherwise, an error will be raised. The transition relation declaration, as the name suggests, describes the global transition relation of the resulting Kripke structure modelling the behaviours of interlocking systems.

⟨*transrel-decl*⟩ ::= **transrel** ⟨*transrel*⟩

⟨*transrel*⟩ ::= **[** ⟨*ident*⟩ **]** ⟨*simple-expr*⟩ ⟶ ⟨*next-expr*⟩
| **[** ⟨*ident*⟩ **]**
| **[** ⟨*ident*⟩ **(** ⟨*[expr-list]*⟩ **)** **]**
| ⟨*transrel*⟩ [=] ⟨*transrel*⟩
| ⟨*transrel*⟩ [>] ⟨*transrel*⟩
| **(** [=] ⟨*ident*⟩ **:** ⟨*elem-type*⟩ • ⟨*transrel*⟩ **)**

The transition relation is composed of transitions of different types:

(1) *Atomic transitions* of the following form

[name] guard ⟶ update

where *name* is a unique name to identify the transition, *guard* is a simple expression, and *update* is a next expression. Simple expressions and next expressions are explained in Section 5.1.9. When *guard* holds, then the transition can be taken,

consequently change the states as specified in *update*. An example of atomic transition is shown in the following.

$[\text{ctrl\_nocmd\_to\_dispatch}]\ (\text{r.CTRL} = \text{NOCMD}) \longrightarrow (\text{r.CTRL}' = \text{DISPATCH})$

The above transition means that, when a route *r* has no pending commands, then the route can be commanded to be dispatched.

When instantiated with specific configuration data, an atomic transition of the above form will be transformed into its corresponding propositional form as in the following.

$\text{guard} \wedge \text{update} \wedge \text{others\_unchanged}$

where *others_unchanged* is a proposition expressing that all variables other than the ones that are updated in proposition *update* are unchanged in the next state, i.e., their values in the next state are the same as their values in the current state.

(2) *Module application transitions* with or without parameters where module declarations are explained in Section 5.1.6. Some examples of module application transitions are shown in the following.

$[\text{moduleA}]$
$[=]$
$[\text{moduleB(p1,p2)}]$

The above module application transitions, when instantiated with a concrete configuration data, will be replaced by the actual transition relation of the modules with their actual parameters.

(3) *Quantified transitions* allow us to specify transitions over a class of elements of a given type, e.g.,

$([=]\ \text{r} : \textbf{Route} \bullet [\text{route\_marking}]\ \text{r.MODE} = \text{FREE} \longrightarrow \text{r.MODE}' = \text{MARKED})$

The above quantified transition, when instantiated with concrete configuration data, for each route in the configuration data, *r* will be replaced by the identifier of the route, resulting in a concrete transition.

(4) *Composition of transitions*: transitions can be combined by one of the following operators.

- Non-deterministic operator $[=]$: $\text{A}\ [=]\ \text{B}$ means that transition *A* has the same priority as transition *B*.
- Priority operator $[>]$: $\text{A}\ [>]\ \text{B}$ means that transition *A* has a higher priority than transition *B*. The priority operator $[>]$ is invented by Hansen in his master's thesis [Han15]. The operator has been adopted into IDL.

If two transitions having different priorities are enabled at the same time, the one with higher priority should be taken. On the other hand, transitions with the same priority should be chosen nondeterministically, if they are enabled at the same time. In propositional forms, if transitions represented by propositions Φ_a and Φ_b have the same priority, their combined behaviour is described by the proposition $\Phi_a \vee \Phi_b$. On the other hand, if the transitions represented by Φ_a has higher priority than the ones represented by Φ_b, then their combined behaviour is described by $\Phi_a \vee (\neg g_a \wedge \Phi_b)$, where g_a is the condition for at least one of the transitions represented by Φ_a to be enabled.

Note that [=] is left-associative, while [>] operator is right-associative, i.e.,

A [=] B [=] C ≡ ((A [=] B) [=] C)

A [>] B [>] C ≡ (A [>] (B [>] C))

5.1.6 Module Declarations

A module specifies the behaviours of a part of the given system such as the system under consideration or the environment. A module can be parameterised and instantiated multiple times in the global transition relation. Note that variables specified in the encoding declaration as described in Section 5.1.2 are shared among all modules.

⟨*module-decl*⟩ ::= **module** ⟨*ident*⟩ ⟨*transrel*⟩
| **module** ⟨*ident*⟩ **(** ⟨*[ident-list]*⟩ **)** ⟨*transrel*⟩

An excerpt from a module named *ET* specifying the behaviours of track elements is shown in the following.

```
/* ========================
 * TRACK ELEMENT TRANSITIONS
 * ========================*/
module ET
/* Start switching */
([=] p : Point •
  [point_switch_1] p.POS ≠ p.CMD ∧ p.POS ≠ INTER ⟶ p.POS′ = INTER)
[=]

/* Move in the commanded position */
([=] p : Point • [point_switch_2] p.POS = INTER ⟶ p.POS′ = p.CMD)
[=]
 ...
```

5.1.7 Invariant Declarations

An invariant declaration describes global properties that need to be proved in the Kripke structure. In case of the Danish interlocking systems, these properties are the safety properties and strengthening invariants as described in Chapter 6. An invariant is expressed by a simple expression, and is uniquely identified with a name. Simple expressions are explained further in Section 5.1.9.

⟨*invariant-decl*⟩ ::= **invariant** ⟨*invariant-list*⟩

⟨*invariant*⟩ ::= **[** ⟨*ident*⟩ **]** ⟨*simple-expr*⟩

Some examples of invariants are shown in the following.

```
invariant
/**
 * ============================
 * HIGH−LEVEL SAFETY PROPERTIES
 * ============================
 */
[no_head_to_head_collision_linear]
(∀l : Linear • (¬is_boundary_sec(l)) ⇒ (l.D2U * l.U2D = 0)),

[no_train_follows_another_collision_linear]
(∀l : Linear •
 (¬is_boundary_sec(l)) ⇒
 (l.D2U * (1 − (l.D2U & 1)) + l.U2D * (1 − (l.U2D & 1)) = 0)),
 ...
```

5.1.8 Test Objective Declarations

A test objective declaration describes a list of test objectives that need to be generated for a target system. LTL is used for specifying test objectives. The syntax for test objectives and LTL formulas are presented in the following.

⟨*test-obj-decl*⟩ ::= **test_obj** ⟨*test-obj-list*⟩

⟨*test-obj*⟩ ::= **[** ⟨*ident*⟩ **]** ⟨*ltl-formula*⟩
| **(** [=] ⟨*ident*⟩ **:** ⟨*elem-type*⟩ • ⟨*test-obj*⟩ **)**

Test cases can be one of the following types:

(1) *Elementary test objectives*: an elementary test objective has a unique name, and is described by an LTL formula over the free variables of the resulting Kripke structure, e.g.,

[TO_route_marked] **F** [r1.DSPL = MARKED]

(2) *Quantified test objectives*: A quantified test objective specifies a test objective for a class of elements, i.e., upon being instantiated by a given configuration, a quantified test objective will result in a number of test objectives, a test objective for each element of the specified element type, e.g.,

([=] r : **Route** • [TO_route_in_use] **F** [r.DSPL = OCCUPIED])

LTL Formulas. LTL formulas are used to specify requirements and test objectives. State formulas are enclosed in squared brackets. The supported LTL operators are *globally* **G**, *next* **X**, *finally* **F**, *until* **U**, and *exists* **E**. *Release* operator **R** is not supported. Logical operator *and* $\wedge$, *or* $\vee$, and *implication* $\Rightarrow$ are supported. The bounded semantics of LTL introduced by Biere et al. in [Bie+99a; Bie+06] is implemented in IDL. The same semantics is implemented in RT-Tester's bounded model checker and MBT framework.

⟨*ltl-formula*⟩ ::= **[** ⟨*simple-expr*⟩ **]**
| **G** ⟨*ltl-formula*⟩
| **X** ⟨*ltl-formula*⟩
| **F** ⟨*ltl-formula*⟩
| ¬ ⟨*ltl-formula*⟩
| **E** ⟨*ident*⟩ **:** ⟨*ltl-formula*⟩
| ⟨*ltl-formula*⟩ **U** ⟨*ltl-formula*⟩
| ⟨*ltl-formula*⟩ ∧ ⟨*ltl-formula*⟩
| ⟨*ltl-formula*⟩ ∨ ⟨*ltl-formula*⟩
| ⟨*ltl-formula*⟩ ⇒ ⟨*ltl-formula*⟩

5.1.9 Expressions

The syntax of an expression in IDL is shown in the following.

⟨*expr*⟩ ::= ⟨*ident*⟩
| ⟨*literal*⟩
| ⟨*elem-type*⟩
| ⟨*symbol-expr*⟩
| ⟨*uop*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*bop*⟩ ⟨*expr*⟩
| ⟨*macro-ex-expr*⟩
| ⟨*domain-expr*⟩
| ⟨*quantified-expr*⟩
| ⟨*if-then-else-expr*⟩
| ⟨*case-expr*⟩
| ⟨*let-expr*⟩
| ⟨*index-expr*⟩

There are different types of expressions supported by IDL as explained in the following.

(a) *Identifiers.* An identifier can contains any alphanumeric characters and underscores. Identifiers must start with a letter or an underscore and are not identical to any of the reserved keywords. Identifiers can be the identifiers of elements in interlocking configuration data, quantified variable in quantified expressions, parameters for macros, or local variables in the assignments of let expressions.

$\langle$*ident*$\rangle$ ::= [_a-zA-Z][_a-zA-Z0-9]*

(b) *Literals.* Literal natural values* can be specified in IDL in decimal, hexadecimal (prefixed with 0*x*), or binary (prefixed with 0*b*).

$\langle$*literal*$\rangle$ ::= [0-9]+ | 0b[01]+ | 0x[0-9a-fA-F]+

(c) *Type Expressions.* A type expression represents the set of identifiers of the elements of a certain type τ in a given interlocking configuration data. For instance, type expression **Linear** represents the set of linear sections in a given interlocking configuration data.

$\langle$*elem-type*$\rangle$ ::= **Linear** | **Point** | **Section** | **Signal** | **Route**

(d) *Prefix Expressions.* Logical negation operator $\neg$ is supported.

$\langle$*expr*$\rangle$ ::= $\langle$*uop*$\rangle$ $\langle$*expr*$\rangle$

$\langle$*uop*$\rangle$ ::= $\neg$

(e) *Infix Expressions.* The supported infix operators are listed in the following.

- *Comparison*: $\leq, <, >, \geq, =, \neq$
- *Logical*: and $\wedge$, or $\vee$, xor $\oplus$, implication $\Rightarrow$. Logical expressions are evaluated lazily.
- *Arithmetic*: addition $+$, subtraction $-$, multiplication $*$, division $/$, modulo %
- *Bit-wise*: bitwise and &, bitwise or |, arithmetic bit shift left $\ll$, arithmetic bit shift right $\gg$

Associativity and precedence of these operators are similar to their associativity and precedence in C/C++.

$\langle$*expr*$\rangle$ ::= $\langle$*expr*$\rangle$ $\langle$*bop*$\rangle$ $\langle$*expr*$\rangle$

*In the current version of the language, literals are only non-negative integers. However, they can be extended to contain floats or other types if needed.

$$
\begin{array}{lll}
\langle bop \rangle & ::= & \leq \mid < \mid > \mid \geq \mid = \mid \neq \\
 & \mid & \wedge \mid \vee \mid \oplus \mid \Rightarrow \\
 & \mid & + \mid - \mid * \mid / \mid \% \\
 & \mid & \& \mid \mid \mid \ll \mid \gg
\end{array}
$$

(f) *Symbol Access Expressions*: A symbol access expression describes a symbol variable in the resulting Kripke structure. A symbol access expression consists of three parts:

- the name of an element whose status the variable represents;
- the symbol; and
- the version of the variable. The version of a variable can be either current (specified by $\langle empty \rangle$) or next state (specified by $\langle ' \rangle$).

$$
\begin{array}{lll}
\langle symbol\text{-}expr \rangle & ::= & \langle expr \rangle \ \mathbf{.} \ \langle ident \rangle \ \langle version \rangle \\
\langle version \rangle & ::= & ' \mid \langle empty \rangle
\end{array}
$$

For example, r.MODE denotes the variable encoding the mode of the route r in the current state, while r.MODE$'$ denotes the variable encoding the mode of the route r in the next state.

(g) *Macro Expansion Expressions*: A macro expansion expression describes the application of a defined macro on the given parameters. Parentheses can be omitted if the macro does not have any parameters.

$$
\begin{array}{lll}
\langle macro\text{-}ex\text{-}expr \rangle & ::= & \langle ident \rangle \ \mathbf{(} \ \langle [expr\text{-}list] \rangle \ \mathbf{)} \\
 & \mid & \langle ident \rangle
\end{array}
$$

For example, vacant_point(p1) expands the macro example vacant_point(p) in Section 5.1.3 for point section $p1$ by replacing all occurrences of p in the expression defined by the macro by $p1$.

(h) *Domain Functions and Operators*: A number of domain functions and operators are supported by IDL to facilitate the access of data in the given interlocking configuration. For example, the domain function **down** when applied on a linear section l will return its neighbour in the down end in the given interlocking configuration data. The meaning of the domain functions and operators are listed in Section B.3.

$$
\begin{array}{lll}
\langle domain\text{-}expr \rangle & ::= & \langle domain\text{-}func \rangle \ \mathbf{(} \ \langle expr\text{-}list \rangle \ \mathbf{)} \\
 & \mid & \langle domain\text{-}uop \rangle \ \langle expr \rangle \\
 & \mid & \langle expr \rangle \ \langle domain\text{-}bop \rangle \ \langle expr \rangle
\end{array}
$$

⟨*domain-func*⟩ ::= **down** | **up** | **down_sig** | **up_sig**
| **stem** | **plus** | **minus**
| **dir** | **track**
| **src** | **dst** | **first** | **last**
| **path** | **overlap** | **points** | **signals** | **conflicts**
| **prev** | **next** | **prevs** | **nexts** | **req**
| **conn_end**
| **entry** | **exit**

⟨*domain-uop*⟩ ::= **elems** | **hd** | **tl** | **dom** | **rng** | **len**

⟨*domain-bop*⟩ ::= ∈ | ∪ | ∩ | \

(i) *Quantified Expressions*: A quantified expression allows us to reason over a class of elements such as linears or routes. There are three different types of quantified expressions: *all* ∀, *exists* ∃, and *unique* ∃!. Their meaning is the same as their mathematical meaning.

⟨*quantified-expr*⟩ ::= **(** ⟨*quan-op*⟩ ⟨*ident*⟩ **:** ⟨*elem-type*⟩ • ⟨*expr*⟩ **)**

⟨*quan-op*⟩ ::= ∀ | ∃ | ∃**!**

(j) *If-then-else Expressions*: Both the usual *if-then-else* expressions and their ternary forms are supported. Note that both *then* and *else* branches are required in an *if-then-else* expression.

⟨*if-then-else-expr*⟩ ::= **if** ⟨*expr*⟩ **then** ⟨*expr*⟩ **else** ⟨*expr*⟩ **end**
| ⟨*expr*⟩ **?** ⟨*expr*⟩ **:** ⟨*expr*⟩

(k) *Case Expressions*: A case expression specifies different outcomes for different cases. If the considered expression is evaluated to the same value as a condition, then the corresponding outcome will be returned and the evaluation stops. If no cases match, then the default branch denoted by the ⟨*wildcard*⟩ will be chosen.

⟨*case-expr*⟩ ::= **case** ⟨*expr*⟩ **of** ⟨*case-branch-list*⟩ **end**
| **case** ⟨*expr*⟩ **of** ⟨*case-default*⟩ **end**
| **case** ⟨*expr*⟩ **of** ⟨*case-branch-list*⟩ **,** ⟨*case-default*⟩ **end**

⟨*case-branch*⟩ ::= ⟨*expr*⟩ → ⟨*expr*⟩

⟨*case-default*⟩ ::= ⟨*wildcard*⟩ → ⟨*expr*⟩

⟨*wildcard*⟩ ::= _

(l) *Let Expressions*: A let expression allows us to assign some local variables to be used in an expression.

⟨*let-expr*⟩ ::= **let** ⟨*assign-list*⟩ **in** ⟨*expr*⟩ **end**

⟨*assign*⟩ ::= ⟨*ident*⟩ **=** ⟨*expr*⟩

(m) *Indexing Expressions*: An indexing expression allows us to access items of a list by indices, or items of a map by keys. The list index can be either a natural number, or a range of number, while a map index is an object. Lists are *0-indexed*, i.e., their indices start from 0.

⟨*index-expr*⟩ ::= ⟨*expr*⟩ **[** ⟨*expr*⟩ **]**
| ⟨*expr*⟩ **[** ⟨*expr*⟩ **:** ⟨*expr*⟩ **]**

Simple Expressions vs. Next Expressions. Simple expressions and next expressions have the same syntax, the only difference is that only variables representing the current state of the Kripke structure, i.e., variables in the set V introduce in Section 2.11.1, can appear in a simple expression. On the other hand, in a next expression, both variables presenting current state and next state, i.e., variables in the set $V \cup V'$, see Section 2.11.1, are allowed.

⟨*simple-expr*⟩ ::= ⟨*expr*⟩

⟨*next-expr*⟩ ::= ⟨*expr*⟩

5.1.10 Comments

Line comments and block comments in C-like style can be added anywhere in the specification.

5.2 Semantics

This section describes the formal semantics of specifications in IDL. In other words, it describes how a specification in IDL can be instantiated with a specification of interlocking configuration data to produce a concrete behavioural model in the form of a Kripke structure, concrete safety properties, and concrete test objectives associated with the behavioural model. The semantics of specifications in IDL is then used to elaborate further the semantics of specifications of configuration data in ICL which has been briefly introduced in Section 4.6.

5.2.1 Syntactic and Semantic Domains

We define the following syntactic domains.

- For brevity, Σ is used as an alias for Interlocking, the domain of interlocking configuration data in ICL as defined in Chapter 4, i.e., $\Sigma = \text{Interlocking}$.
- Δ is the domain of specifications in IDL.

The semantic domains used to describe the semantics of ICL and IDL are listed in the following.

- $\mathbb{N}$ is the domain of non-negative integers (i.e., natural numbers).
- $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ is the domain of boolean values.
- $\mathcal{V}$ is the domain of variables. A variable is a non-empty string of alphanumeric characters, or underscore, and starts by a letter or an underscore. Each variable $v \in \mathcal{V}$ is associated with a finite domain $D_v \subset \mathbb{N}$.
- $\mathcal{K}$ is the domain of Kripke structures defined in Section 2.11.1.
- $\mathcal{TS}$ is the domain of IOSTSes defined in Section 2.11.3.
- *Prop* is the domain of propositions over free variables in $\mathcal{V}$. In other words, *Prop* is the set of syntactic objects constructed by the following formation rules where $\langle Prop \rangle$ stands for any proposition, $\langle lit \rangle$ stands for a literal value in $\mathbb{N}$, and $\langle var \rangle$ stands for a variable in $\mathcal{V}$.

  $\langle Prop \rangle$ ::= $\langle lit \rangle$ | $\langle var \rangle$ | $\langle uop \rangle$ $\langle Prop \rangle$ | $\langle Prop \rangle$ $\langle bop \rangle$ $\langle Prop \rangle$

  $\langle uop \rangle$ ::= $\neg$

  $\langle bop \rangle$ ::= $\leq$ | $<$ | $>$ | $\geq$ | $=$ | $\neq$
  | $\wedge$ | $\vee$ | $\oplus$ | $\Rightarrow$
  | $+$ | $-$ | $*$ | $/$ | %
  | & | \| | $\ll$ | $\gg$

  $\langle var \rangle$ ::= [_a-zA-Z][_a-zA-Z0-9]*

  $\langle lit \rangle$ ::= [0-9]+ | 0b[01]+ | 0x[0-9a-fA-F]+

  Note that the conversion between integral values and boolean values are done implicitly in *Prop* as explained in Section 5.2.2. The meaning of prefix operators $\langle uop \rangle$ and infix operators $\langle bop \rangle$ are similar to the meaning of the prefix operators and infix operators in IDL, see Section 5.1.9.

- *TestObj* is the domain of concrete test objectives. A concrete test objective is expressed by an LTL formula over propositions in *Prop*. In other words, *TestObj* is the set of syntactic objects constructed by the following formation rules where $\langle TestObj \rangle$ stands for any test objective, and $\langle Prop \rangle$ stands for a proposition in *Prop* defined above. LTL state formulas are enclosed in squared brackets. **G**, **X**, **F**, **E**, **U**, $\neg$, $\wedge$, $\vee$, $\Rightarrow$ are LTL and logical operators as explained in Section 5.1.8.

  $\langle TestObj \rangle$ ::= **[** $\langle prop \rangle$ **]**
  | **G** $\langle TestObj \rangle$
  | **X** $\langle TestObj \rangle$

$$
\begin{array}{ll}
| & \mathbf{F}\ \langle \mathit{TestObj} \rangle \\
| & \neg\ \langle \mathit{TestObj} \rangle \\
| & \mathbf{E}\ \langle \mathit{ident} \rangle : \langle \mathit{TestObj} \rangle \\
| & \langle \mathit{TestObj} \rangle\ \mathbf{U}\ \langle \mathit{TestObj} \rangle \\
| & \langle \mathit{TestObj} \rangle \wedge \langle \mathit{TestObj} \rangle \\
| & \langle \mathit{TestObj} \rangle \vee \langle \mathit{TestObj} \rangle \\
| & \langle \mathit{TestObj} \rangle \Rightarrow \langle \mathit{TestObj} \rangle
\end{array}
$$

- Section $=$ Linear $\cup$ Point is the domain of detection sections where Linear and Point are defined in Section 4.1.
- Elem $=$ Section $\cup$ MarkerBoard $\cup$ Route is the domain of interlocking elements where markerboard and route are defined in Section 4.1.
- Val is the domain of denotational values of expressions.

  $$\mathrm{Val} = \texttt{Id} \cup \texttt{Id-set} \cup \texttt{Id}^* \cup (\texttt{Id} \underset{m}{\rightarrow} \mathbb{N}) \cup \mathbb{N}\texttt{-set} \cup \mathbb{N} \cup \mathit{Prop}$$

  where S-`set` is an RSL-like notion denoting the powerset of a set S as explained in Section 5.2.2. Syntactically, *Prop* is a superset of `Id`, so the union absorbs `Id`. However, `Id` is still shown in the above formula for readability.

5.2.2 Auxiliary Functions and Notions

In order to facilitate the specification of the semantics of IDL, we define the following functions and notions.

(1) S-`set` is an RSL-like notion denoting the powerset of a set S.

(2) *Term replacement*: $e[t/i]$ denotes a term (expression/transition relation/test case) where all the appearances of i in the term e is replaced by the term t.

(3) *Conversion between booleans and naturals*: The conversion is done implicitly as in C/C++ in the subsequent subsections by two functions:

- $\eta : \mathbb{B} \rightarrow \mathbb{N}$ converts a boolean value $b \in \mathbb{B}$ to a natural value $\eta(b)$

  $$\eta(b) \triangleq \begin{cases} 1 & \text{if } b = \texttt{true} \\ 0 & \text{otherwise} \end{cases}$$

- $\beta : \mathbb{N} \rightarrow \mathbb{B}$ converts a natural value $n \in \mathbb{N}$ to a boolean value $\beta(b)$.

  $$\beta(n) \triangleq n \neq 0$$

(4) $\kappa : \mathcal{TS} \rightarrow \mathcal{K}$ is a function that transforms an IOSTS to a Kripke structure. Such transformation has been explained in Section 2.11.

(5) *written* : *Prop* $\rightarrow$ $\mathcal{V}$-`set` is a function that returns the set of variables that are written to in a proposition ϕ, i.e., their values in the next state are updated in ϕ.

(6) RSL builtin functions and functions that are defined in the specification of ICL in Chapter 4, when referred in this section, are typed in regular monospace font face, e.g., `hd` or `down`. The following RSL functions are used [Gro92].

- `dom` returns the domain of a given map.
- `rng` returns the range of a given map.
- `elems` returns the set of elements of a given list.
- `hd` returns the first element of a given list, or a random element from a given set.
- `tl` returns the remaining of a given list after removing the first element.
- `len` returns the length of a given list.
- `card` returns the cardinality of a given set.

Functions that are defined in the specification of ICL can be found in Chapter 4 and Appendix A.

(7) $elems : \Sigma \stackrel{\sim}{\rightarrow} (\texttt{Id} \underset{m}{\rightarrow} \text{Elem})$ is a function that returns a map from identifiers to the corresponding interlocking elements.

$$elems(\sigma) \triangleq \texttt{linears}(N) \cup \texttt{points}(N) \cup \texttt{marker\_boards}(N) \cup I$$

where $N = \texttt{track\_layout}(\sigma)$, and $I = \texttt{interlocking\_table}(\sigma)$. Functions `linears`, `points`, `marker_boards`, `track_layout`, `interlocking_table` are defined in the RSL specification of ICL in Section 4.1.

(8) $dom : \Sigma \stackrel{\sim}{\rightarrow} \texttt{Id-set}$ returns the set of identifiers of all elements in a given interlocking configuration data σ.

$$dom(\sigma) \triangleq \texttt{dom}\ elems(\sigma)$$

(9) *Membership operator* denotes whether an element with id i belongs to a given interlocking configuration data σ.

$$\in : \texttt{Id} \times \Sigma \rightarrow \mathbb{B}$$

$$i \in \sigma \triangleq i \in dom(\sigma)$$

(10) *Subscript operator* returns an element with id i from a given wellformed interlocking data σ. Note that subscript operator is injective because identifiers of elements in σ are unique as described in the wellformedness conditions of interlocking configuration data in Chapter 4.

$$[-] : \Sigma \times \texttt{Id} \stackrel{\sim}{\rightarrow} \text{Elem}$$

$$\sigma[i] \triangleq elems(\sigma)[i] \qquad \text{if } i \in \sigma$$

(11) *Type function*: $\Gamma(e)$ where $e \in \sigma$ returns the element type, e.g., **Linear** or **Point**, of the given element e. For example, $\Gamma(l)$ where l is a linear section will give **Linear**.

(12) *Restriction*: $\sigma \mid_\tau$ where τ is an element type, e.g., **Linear** or **Point**, denotes the set of identifiers of the elements in the given configuration data σ which have the element type of τ. For example, $\sigma \mid_{\mathbf{Linear}}$ is the set of identifiers of all linear sections in σ.

$$\sigma \mid_\tau \triangleq \{i \mid i \in dom(\sigma) \wedge \Gamma(\sigma[i]) = \tau\}$$

5.2.3 Semantics Overview

IDL Specification Semantics. A specification $\delta = (E_\delta, I_\delta, R_\delta, P_\delta, TC_\delta) \in \Delta$ with encoding declaration E_δ, initial declaration I_δ, transition relation R_δ, invariant declaration P_δ, and test case declaration TC_δ when instantiated with configuration data $\sigma \in \Sigma$, will result in a corresponding IOSTS TS modelling the behaviours of the interlocking systems, a safety invariant ϕ, and a list of symbolic test cases TC.

$$[\![-]\!]^\Delta : \Delta \overset{\sim}{\to} \Sigma \overset{\sim}{\to} (\mathcal{K} \times Prop \times TestObj^*)$$
$$[\![\delta]\!]^\Delta \sigma \triangleq (\kappa(TS), \phi, TC)$$

where

- κ is the function that transforms an IOSTS to a Kripke structure described in Section 5.2.2.
- $TS = (S, I, R) \in \mathcal{TS}$ is an IOSTS with
  - the state space S represented by the set of variables $V = [\![E_\delta]\!]^\mathcal{V}\sigma$;
  - the set of initial states $I = \{s \mid ([\![I_\delta]\!]^\mathcal{I}\sigma)(s)\}$ if I_δ is specified in δ, otherwise $I = \{s_0\}$ where s_0 is the state in which $v = v_0$ for all variable $v \in V$ where v_0 is the initial value of v as specified in the encoding E_δ; and
  - the transition relation $R = \{(s, s') \mid ([\![R_\delta]\!]^\mathcal{R}\sigma)(s, s')\}$.
- $\phi = [\![P_\delta]\!]^\mathcal{I}\sigma \in Prop$ is a proposition over free variables in V.
- $TC = [\![TC_\delta]\!]^\mathcal{T}\sigma \in TestObj^*$ is the list of symbolic test cases in TS.

The semantic functions $[\![-]\!]^\mathcal{V}$, $[\![-]\!]^\mathcal{R}$, $[\![-]\!]^\mathcal{I}$, and $[\![-]\!]^\mathcal{T}$ of encodings, transition relation, invariants, and test cases, respectively, are explained in detail in the subsequent subsections.

ICL Specification Semantics. Then the semantics of a given interlocking configuration data σ corresponding to a given generic model δ is derived as in the following.

$$[\![\sigma]\!]^{\Sigma}\delta \triangleq [\![\delta]\!]^{\Delta}\sigma \tag{5.1}$$

Throughout the rest of this chapter, for readability, syntactic objects in IDL are represented by their corresponding formation rules in BNF from which these objects are constructed. For example, an *if-then-else* expression in IDL is represented in the semantics by its corresponding formation rule as in the following.

if c **then** t **else** e **end**

5.2.4 Encoding Semantics

The semantic function $[\![-]\!]^{\mathcal{V}}$ of encodings allow us to instantiate the encoding declaration $E_\delta \in \texttt{Enc}$ of a generic specification δ with a given interlocking configuration σ, resulting in a set of variables $V \in \mathcal{V}\texttt{-set}$ representing the state space.

$$[\![-]\!]^{\mathcal{V}} : \texttt{Enc} \tilde{\to} \Sigma \tilde{\to} \mathcal{V}\texttt{-set}$$

$$[\![\tau :: s_1 \to [t_1, v_1, i_1, l_1, h_1], \ldots, s_n \to [t_n, v_n, i_n, l_n, h_n]]\!]^{\mathcal{V}}\sigma \triangleq \bigcup_{e \in [\![\tau]\!]^{\mathcal{E}}\sigma} \{\mathrm{var}(e, s_1, t_1, v_1, i_1, l_1, h_1), \ldots, \mathrm{var}(e, s_n, t_n, v_n, i_n, l_n, h_n)\}$$

$$[\![e_1, \ldots, e_n]\!]^{\mathcal{V}}\sigma \triangleq [\![e_1]\!]^{\mathcal{V}}\sigma \cup \cdots \cup [\![e_n]\!]^{\mathcal{V}}\sigma$$

where $\mathrm{var}(e, s, t, v, i, l, h)$ returns a variable of symbol type t, target type v, corresponding to the symbol s of the element e that has the low bound of l and high bound of h and initial value of i. It can formally be defined as in the following.

$$\mathrm{var}(e, s, t, v, i, l, h) \triangleq v \in \mathcal{V}$$

where

- $v = e \frown \zeta \frown s$ with ζ is a predefined delimiter, e.g., an underscore;
- $D_v = \{l..h\} \subset \mathbb{N}$; and
- $v_0 = i$.

5.2.5 Expression Semantics

The semantic function $[\![-]\!]^{\mathcal{E}}$ maps an expression in **Expr** to its denotation, i.e., a value in Val, as defined in the following.

$$[\![-]\!]^{\mathcal{E}} : \texttt{Expr} \tilde{\to} \Sigma \tilde{\to} \mathrm{Val}$$

The semantics of different types of expressions are described in detail in the following paragraphs.

Identifiers.

$[\![v]\!]^{\mathcal{E}}\sigma \triangleq v \in dom(\sigma)$ if $v \in \sigma$

Literals.

$[\![n]\!]^{\mathcal{E}}\sigma \triangleq n \in \mathbb{N}$

Type Expressions.

$[\![\tau]\!]^{\mathcal{E}}\sigma \triangleq \sigma|_{\tau} \in \texttt{Id-set}$

Symbol Access Expressions.

$[\![o.sym\ v]\!]^{\mathcal{E}}\sigma \triangleq ([\![o]\!]^{\mathcal{E}}\sigma) \frown \zeta \frown sym \frown v \in Prop$ if $[\![o]\!]^{\mathcal{E}}\sigma \in \sigma$

where ζ is the predefined delimiter described in Section 6.2.1.

Prefix Expressions.

$[\![\texttt{uop}\ e]\!]^{\mathcal{E}}\sigma \triangleq \texttt{uop}\ [\![e]\!]^{\mathcal{E}}\sigma \in Prop$ if $[\![e]\!]^{\mathcal{E}}\sigma \in Prop$

Infix Expressions.

$[\![e_1\ \texttt{bop}\ e_2]\!]^{\mathcal{E}}\sigma \triangleq [\![e_1]\!]^{\mathcal{E}}\sigma\ \texttt{bop}\ [\![e_2]\!]^{\mathcal{E}}\sigma \in Prop$ if $[\![e_1]\!]^{\mathcal{E}}\sigma \in Prop$ and $[\![e_2]\!]^{\mathcal{E}}\sigma \in Prop$

Macro Expansion Expressions.

$[\![m(e_1,\ldots,e_n)]\!]^{\mathcal{E}}\sigma \triangleq [\![\texttt{expr}(m)[[\![e_1]\!]^{\mathcal{E}}\sigma/p_1,\ldots,[\![e_n]\!]^{\mathcal{E}}\sigma/p_n]]\!]^{\mathcal{E}}\sigma$

where $\texttt{expr}(m)$ returns the expression defined by the macro m which has $p_1,\ldots,p_n$ as formal parameters.

Domain Functions and Operators. The semantics of domain functions and operators in IDL are given in the following. The functions in IDL are in roman bold face, e.g., **down**, while the functions in monospace face, e.g., `down`, denotes the corresponding functions defined in the ICL in Chapter 4 or predefined functions in RSL.

$[\![\mathbf{elems}\ e_1]\!]^{\mathcal{E}}\sigma \triangleq \texttt{elems}\ [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set}$ if $[\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id}^*$

$[\![\mathbf{dom}\ e_1]\!]^{\mathcal{E}}\sigma \triangleq \texttt{dom}\ [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set}$ if $[\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id} \underset{m}{\rightarrow} \mathbb{N}$

$[\![\mathbf{rng}\ e_1]\!]^{\mathcal{E}}\sigma \triangleq \texttt{rng}\ [\![e_1]\!]^{\mathcal{E}}\sigma \in \mathbb{N}\texttt{-set}$ if $[\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id} \underset{m}{\rightarrow} \mathbb{N}$

$$[\![\mathbf{hd}\ e_1]\!]^{\mathcal{E}}\sigma \triangleq \texttt{hd}\ [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in (\mathbb{N}\texttt{-set} \cup \texttt{Id-set}) \text{ and } \texttt{card}\ [\![e_1]\!]^{\mathcal{E}}\sigma > 0$$
$$\text{or } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id}^* \text{ and } \texttt{len}\ [\![e_1]\!]^{\mathcal{E}}\sigma > 0$$

$$[\![\mathbf{tl}\ e_1]\!]^{\mathcal{E}}\sigma \triangleq \texttt{tl}\ [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id}^*$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in (\mathbb{N}\texttt{-set} \cup \texttt{Id-set}) \text{ and } \texttt{card}\ [\![e_1]\!]^{\mathcal{E}}\sigma > 0$$
$$\text{or } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id}^* \text{ and } \texttt{len}\ [\![e_1]\!]^{\mathcal{E}}\sigma > 0$$

$$[\![\mathbf{len}\ e_1]\!]^{\mathcal{E}}\sigma \triangleq \begin{cases} \texttt{len}\ [\![e_1]\!]^{\mathcal{E}}\sigma \in \mathbb{N} & \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id}^* \\ \texttt{card}\ [\![e_1]\!]^{\mathcal{E}}\sigma \in \mathbb{N} & \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set} \\ \texttt{card dom}\ [\![e_1]\!]^{\mathcal{E}}\sigma \in \mathbb{N} & \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id} \xrightarrow{m} \mathbb{N} \end{cases}$$

$$[\![\mathbf{down}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{down}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Section}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Linear}}$$
$$[\![\mathbf{up}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{up}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Section}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Linear}}$$
$$[\![\mathbf{down\_sig}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{down\_sig}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{MarkerBoard}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Linear}}$$
$$[\![\mathbf{up\_sig}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{up\_sig}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{MarkerBoard}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Linear}}$$

$$[\![\mathbf{stem}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{stem}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Section}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Point}}$$
$$[\![\mathbf{plus}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{plus}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Section}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Point}}$$
$$[\![\mathbf{minus}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{minus}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Section}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Point}}$$

$$[\![\mathbf{dir}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{dir}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \mathbb{N} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{MarkerBoard}}$$
$$[\![\mathbf{track}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{track}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Linear}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{MarkerBoard}}$$

$$[\![\mathbf{src}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{src}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{MarkerBoard}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$
$$[\![\mathbf{dst}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{dst}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{MarkerBoard}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$
$$[\![\mathbf{first}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{first}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Section}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$
$$[\![\mathbf{last}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{last}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Section}} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$
$$[\![\mathbf{path}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{path}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in (\sigma\,|_{\mathbf{Section}})^* \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$
$$[\![\mathbf{overlap}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{overlap}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in (\sigma\,|_{\mathbf{Section}})^* \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$
$$[\![\mathbf{points}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{points}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in \sigma\,|_{\mathbf{Point}} \xrightarrow{m} \mathbb{N} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$
$$[\![\mathbf{signals}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{signals}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in (\sigma\,|_{\mathbf{MarkerBoard}})\texttt{-set} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$
$$[\![\mathbf{conflicts}(e_1)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{conflicts}([\![e_1]\!]^{\mathcal{E}}\sigma, \sigma) \in (\sigma\,|_{\mathbf{Route}})\texttt{-set} \qquad \text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\mathbf{Route}}$$

$$[\![\mathbf{prev}(e_1,e_2)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{prev}([\![e_1]\!]^{\mathcal{E}}\sigma,[\![e_2]\!]^{\mathcal{E}}\sigma,\sigma) \in \sigma\,|_{\textbf{Section}}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Route}} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Section}}$$
$$[\![\mathbf{next}(e_1,e_2)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{next}([\![e_1]\!]^{\mathcal{E}}\sigma,[\![e_2]\!]^{\mathcal{E}}\sigma,\sigma) \in \sigma\,|_{\textbf{Section}}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Route}} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Section}}$$
$$[\![\mathbf{prevs}(e_1,e_2)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{prevs}([\![e_1]\!]^{\mathcal{E}}\sigma,[\![e_2]\!]^{\mathcal{E}}\sigma,\sigma) \in (\sigma\,|_{\textbf{Section}})^*$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Route}} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Section}}$$
$$[\![\mathbf{nexts}(e_1,e_2)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{nexts}([\![e_1]\!]^{\mathcal{E}}\sigma,[\![e_2]\!]^{\mathcal{E}}\sigma,\sigma) \in (\sigma\,|_{\textbf{Section}})^*$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Route}} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Section}}$$
$$[\![\mathbf{entry}(e_1,e_2)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{entry}([\![e_1]\!]^{\mathcal{E}}\sigma,[\![e_2]\!]^{\mathcal{E}}\sigma,\sigma) \in \mathbb{N}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Route}} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Section}}$$
$$[\![\mathbf{exit}(e_1,e_2)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{exit}([\![e_1]\!]^{\mathcal{E}}\sigma,[\![e_2]\!]^{\mathcal{E}}\sigma,\sigma) \in \mathbb{N}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Route}} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Section}}$$
$$[\![\mathbf{req}(e_1,e_2)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{req}([\![e_1]\!]^{\mathcal{E}}\sigma,[\![e_2]\!]^{\mathcal{E}}\sigma,\sigma) \in \mathbb{N}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Route}} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Point}}$$

$$[\![\mathbf{conn\_end}(e_1,e_2)]\!]^{\mathcal{E}}\sigma \triangleq \texttt{conn\_end}([\![e_1]\!]^{\mathcal{E}}\sigma,[\![e_2]\!]^{\mathcal{E}}\sigma,\sigma) \in \mathbb{N}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Section}} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \sigma\,|_{\textbf{Section}}$$

$$[\![e_1 \in e_2]\!]^{\mathcal{E}}\sigma \triangleq ([\![e_1]\!]^{\mathcal{E}}\sigma \in [\![e_2]\!]^{\mathcal{E}}\sigma) \in \mathbb{B}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set}$$
$$[\![e_1 \cup e_2]\!]^{\mathcal{E}}\sigma \triangleq ([\![e_1]\!]^{\mathcal{E}}\sigma \cup [\![e_2]\!]^{\mathcal{E}}\sigma) \in \texttt{Id-set}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set}$$
$$[\![e_1 \cap e_2]\!]^{\mathcal{E}}\sigma \triangleq ([\![e_1]\!]^{\mathcal{E}}\sigma \cap [\![e_2]\!]^{\mathcal{E}}\sigma) \in \texttt{Id-set}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set}$$
$$[\![e_1 \setminus e_2]\!]^{\mathcal{E}}\sigma \triangleq ([\![e_1]\!]^{\mathcal{E}}\sigma \setminus [\![e_2]\!]^{\mathcal{E}}\sigma) \in \texttt{Id-set}$$
$$\text{if } [\![e_1]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set} \text{ and } [\![e_2]\!]^{\mathcal{E}}\sigma \in \texttt{Id-set}$$

Quantified Expressions.

$$[\![\forall\, v : \tau \bullet p]\!]^{\mathcal{E}}\sigma \triangleq \bigwedge_{c \in [\![\tau]\!]^{\mathcal{E}}\sigma} [\![p[c/v]]\!]^{\mathcal{E}}\sigma \in \mathbb{B} \qquad \text{if } [\![p[c/v]]\!]^{\mathcal{E}}\sigma \in \mathbb{B} \text{ for all } c \in [\![\tau]\!]^{\mathcal{E}}\sigma$$
$$[\![\exists\, v : \tau \bullet p]\!]^{\mathcal{E}}\sigma \triangleq \bigvee_{c \in [\![\tau]\!]^{\mathcal{E}}\sigma} [\![p[c/v]]\!]^{\mathcal{E}}\sigma \in \mathbb{B} \qquad \text{if } [\![p[c/v]]\!]^{\mathcal{E}}\sigma \in \mathbb{B} \text{ for all } c \in [\![\tau]\!]^{\mathcal{E}}\sigma$$

$$[\![\exists! v : \tau \bullet p]\!]^{\mathcal{E}}\sigma \triangleq \sum_{c \in [\![\tau]\!]^{\mathcal{E}}\sigma} \eta([\![p[c/v]]\!]^{\mathcal{E}}\sigma) = 1 \in \mathbb{B} \quad \text{if } [\![p[c/v]]\!]^{\mathcal{E}}\sigma \in \mathbb{B} \text{ for all } c \in [\![\tau]\!]^{\mathcal{E}}\sigma$$

If-then-else Expressions.

$$[\![\textbf{if } c \textbf{ then } t \textbf{ else } e \textbf{ end}]\!]^{\mathcal{E}}\sigma \triangleq \begin{cases} [\![t]\!]^{\mathcal{E}}\sigma & \text{if } [\![c]\!]^{\mathcal{E}}\sigma \\ [\![e]\!]^{\mathcal{E}}\sigma & \text{otherwise} \end{cases} \qquad \text{if } [\![c]\!]^{\mathcal{E}}\sigma \in \mathbb{B}$$

Let Expressions.

$$[\![\textbf{let } p_1 = e_1, \ldots, p_n = e_n \textbf{ in } e \textbf{ end}]\!]^{\mathcal{E}}\sigma \triangleq [\![e[[\![e_1]\!]^{\mathcal{E}}\sigma/p_1, \ldots, [\![e_n]\!]^{\mathcal{E}}\sigma/p_n]]\!]^{\mathcal{E}}\sigma$$

Case Expressions.

$$[\![\textbf{case } e \textbf{ of } c_1 \to e_1, \ldots, c_n \to e_n, \_ \to e_d \textbf{ end}]\!]^{\mathcal{E}}\sigma \triangleq \begin{cases} [\![e_1]\!]^{\mathcal{E}}\sigma & \text{if } [\![e]\!]^{\mathcal{E}}\sigma = [\![c_1]\!]^{\mathcal{E}}\sigma \\ \ldots & \\ [\![e_n]\!]^{\mathcal{E}}\sigma & \text{if } [\![e]\!]^{\mathcal{E}}\sigma = [\![c_n]\!]^{\mathcal{E}}\sigma \\ [\![e_d]\!]^{\mathcal{E}}\sigma & \text{otherwise} \end{cases}$$

List Indexing Expressions. List indexing expressions allow us to extract a single element from a list of elements by an index, or a list of elements using a range of indices.

$$[\![e[i]]\!]^{\mathcal{E}}\sigma \triangleq [\![e]\!]^{\mathcal{E}}\sigma([\![i]\!]^{\mathcal{E}}\sigma) \in \texttt{Id} \qquad \text{if } [\![e]\!]^{\mathcal{E}}\sigma \in \texttt{Id}^* \text{ and } [\![i]\!]^{\mathcal{E}}\sigma \in \mathbb{N}$$

$$[\![e[i:j]]\!]^{\mathcal{E}}\sigma \triangleq \langle [\![e]\!]^{\mathcal{E}}\sigma(k) \mid [\![i]\!]^{\mathcal{E}}\sigma \leq k < [\![j]\!]^{\mathcal{E}}\sigma \rangle \in \texttt{Id}^*$$
$$\text{if } [\![e]\!]^{\mathcal{E}}\sigma \in \texttt{Id}^*, [\![i]\!]^{\mathcal{E}}\sigma \in \mathbb{N}, \text{ and } [\![j]\!]^{\mathcal{E}}\sigma \in \mathbb{N}$$

Map Indexing Expressions.

$$[\![e[i]]\!]^{\mathcal{E}}\sigma \triangleq [\![e]\!]^{\mathcal{E}}\sigma[[\![i]\!]^{\mathcal{E}}\sigma] \in \mathbb{N} \qquad \text{if } [\![e]\!]^{\mathcal{E}}\sigma \in \texttt{Id} \underset{m}{\rightharpoonup} \mathbb{N} \text{ and } [\![i]\!]^{\mathcal{E}}\sigma \in [\![e]\!]^{\mathcal{E}}\sigma$$

5.2.6 Transition Relation Semantics

The semantic function $[\![-]\!]^{\mathcal{R}}$ returns a proposition corresponding to a transition relation specified in IDL instantiated in an environment σ.

$$[\![-]\!]^{\mathcal{R}} : \texttt{TransRel} \overset{\sim}{\to} \Sigma \overset{\sim}{\to} \mathit{Prop}$$

$$[\![g \longrightarrow u]\!]^{\mathcal{R}}\sigma \triangleq [\![g]\!]^{\mathcal{E}}\sigma \wedge [\![u]\!]^{\mathcal{E}}\sigma \wedge \bigwedge_{v \in \mathcal{V} \setminus \mathit{written}([\![u]\!]^{\mathcal{E}}\sigma)} (v' = v) \qquad \text{if } [\![g]\!]^{\mathcal{E}}\sigma, [\![u]\!]^{\mathcal{E}}\sigma \in \mathit{Prop}$$

$$[\![l\ [=]\ r]\!]^{\mathcal{R}}\sigma \triangleq [\![l]\!]^{\mathcal{R}}\sigma \vee [\![r]\!]^{\mathcal{R}}\sigma$$
$$[\![l\ [>]\ r]\!]^{\mathcal{R}}\sigma \triangleq [\![l]\!]^{\mathcal{R}}\sigma \vee (\neg[\![\mathrm{guard}(l)]\!]^{\mathcal{E}}\sigma \vee [\![r]\!]^{\mathcal{R}}\sigma) \qquad \text{if } [\![\mathrm{guard}(l)]\!]^{\mathcal{E}}\sigma \in \mathit{Prop}$$

$$[\![[=]v : \tau \bullet tr]\!]^{\mathcal{R}}\sigma \triangleq \bigvee_{c \in [\![\tau]\!]^{\mathcal{E}}\sigma} [\![tr[c/v]]\!]^{\mathcal{R}}\sigma$$

$$[\![M(e_1, \ldots, e_n)]\!]^{\mathcal{R}}\sigma \triangleq [\![\mathrm{transrel}(M)[[\![e_1]\!]^{\mathcal{E}}\sigma/p_1, \ldots, [\![e_n]\!]^{\mathcal{E}}\sigma/p_n]]\!]^{\mathcal{E}}\sigma$$

where

$\mathit{written}([\![u]\!]^{\mathcal{E}}\sigma)$ returns a set of variables that are written to in $[\![u]\!]^{\mathcal{E}}\sigma$ as defined in Section 5.2.2;

$\mathrm{guard}(l)$ is the disjunction of the guard of all atomic transitions in l;

$\mathrm{transrel}(M)$ returns the transition relation of the module M, which has $p_1, \ldots, p_n$ as formal parameters.

5.2.7 Test Case Semantics.

In order to specify the semantics of test cases specified in IDL, we first describe the semantics of LTL formulas.

LTL Formulas.

$$[\![-]\!]^{\mathcal{F}} : \texttt{LTLFormula} \xrightarrow{\sim} \Sigma \xrightarrow{\sim} \mathit{TestObj}$$

$$[\![[a]]\!]^{\mathcal{F}}\sigma \triangleq [[\![a]\!]^{\mathcal{E}}\sigma] \qquad \text{if } [\![a]\!]^{\mathcal{E}}\sigma \in \mathit{Prop}$$
$$[\![\mathbf{G}\, f]\!]^{\mathcal{F}} \triangleq \mathbf{G}\ [\![f]\!]^{\mathcal{F}}\sigma$$
$$[\![\mathbf{X}\, f]\!]^{\mathcal{F}} \triangleq \mathbf{X}\ [\![f]\!]^{\mathcal{F}}\sigma$$
$$[\![\mathbf{F}\, f]\!]^{\mathcal{F}} \triangleq \mathbf{F}\ [\![f]\!]^{\mathcal{F}}\sigma$$
$$[\![\neg f]\!]^{\mathcal{F}} \triangleq \neg[\![f]\!]^{\mathcal{F}}\sigma$$
$$[\![\mathbf{E}\ v :\ f]\!]^{\mathcal{F}} \triangleq \mathbf{E}\ v : [\![f]\!]^{\mathcal{F}}\sigma$$
$$[\![f_1\ \mathbf{U}\ f_2]\!]^{\mathcal{F}} \triangleq [\![f_1]\!]^{\mathcal{F}}\sigma\ \mathbf{U}\ [\![f_1]\!]^{\mathcal{F}}\sigma$$
$$[\![f_1 \wedge f_2]\!]^{\mathcal{F}} \triangleq [\![f_1]\!]^{\mathcal{F}}\sigma \wedge [\![f_1]\!]^{\mathcal{F}}\sigma$$
$$[\![f_1 \vee f_2]\!]^{\mathcal{F}} \triangleq [\![f_1]\!]^{\mathcal{F}}\sigma \vee [\![f_1]\!]^{\mathcal{F}}\sigma$$
$$[\![f_1 \Rightarrow f_2]\!]^{\mathcal{F}} \triangleq [\![f_1]\!]^{\mathcal{F}}\sigma \Rightarrow [\![f_1]\!]^{\mathcal{F}}\sigma$$

Test Cases.

$$[\![-]\!]^{\mathcal{T}} : \texttt{TestCase} \overset{\sim}{\to} \Sigma \overset{\sim}{\to} \mathit{TestObj}^*$$

$$[\![f]\!]^{\mathcal{T}}\sigma \triangleq \langle [\![f]\!]^{\mathcal{F}}\sigma \rangle$$

$$[\![tc_1, \ldots, tc_n]\!]^{\mathcal{T}}\sigma \triangleq [\![tc_1]\!]^{\mathcal{T}}\sigma \frown \ldots \frown [\![tc_n]\!]^{\mathcal{T}}\sigma$$

$$\begin{aligned}[\![[=]\ v : \tau \bullet tc]\!]^{\mathcal{T}}\sigma \triangleq\ & [\![tc[c_1/v]]\!]^{\mathcal{T}}\sigma \frown \ldots \frown [\![tc[c_n/v]]\!]^{\mathcal{T}}\sigma \\ & \text{where } \{c_1, \ldots, c_n\} = [\![\tau]\!]^{\mathcal{E}}\sigma\end{aligned}$$

5.2.8 Invariant Semantics

$$[\![-]\!]^{\mathcal{I}} : \texttt{Invariant} \overset{\sim}{\to} \Sigma \overset{\sim}{\to} \mathit{Prop}$$

$$\begin{aligned}
&[\![e]\!]^{\mathcal{I}}\sigma \triangleq [\![e]\!]^{\mathcal{E}}\sigma && \text{if } [\![e]\!]^{\mathcal{E}}\sigma \in \mathit{Prop} \\
&[\![e_1, \ldots, e_n]\!]^{\mathcal{I}}\sigma \triangleq [\![e_1]\!]^{\mathcal{I}}\sigma \wedge \cdots \wedge [\![e_n]\!]^{\mathcal{I}}\sigma \\
&[\![[=]\ v : \tau \bullet e]\!]^{\mathcal{I}}\sigma \triangleq \bigwedge_{c \in [\![\tau]\!]^{\mathcal{E}}\sigma} [\![e[c/v]]\!]^{\mathcal{I}}\sigma
\end{aligned}$$

5.3 Implementation

A parser for IDL is implemented in our toolchain using Flex and GNU Bison. The semantics of IDL is implemented quite straightforward in C++ and integrated into RT-Tester. The implementation inherits the implementation of ICL described in Chapter 4.

5.4 Related Work

The idea with IDL is similar to Object Constraint Language (OCL)[†] and Schematron[‡]. OCL is a language for specifying constraints and object query expressions on any Meta-Object Facility (MOF) meta-model, including Unified Modelling Language (UML) models. Schematron is an ISO/IEC standard, a language for making assertions about the presence or absence of patterns in XML documents. Although Schematron is a markup language, the way it manipulates XML elements is similar to

[†] `http://www.omg.org/spec/OCL/`
[‡] `http://www.schematron.com/`

the way IDL works on different elements of interlocking configuration data specified in ICL. IDL differs itself from languages like OCL or Schematron by its narrow focus on a specific class of railway applications. Furthermore, IDL provides a more simplified and user-friendly interface in order to support the users coming from different areas of expertise other than computer science.

European Railway Interlocking Specification (EURIS) [BMS93; Dij+98] is a domain-specific, modular language/method for specifying interlocking logics. A specification of an interlocking system in EURIS is constructed by interconnecting generic building blocks representing different types of elements such as signals or points. These building blocks exchange telegrams in order to reach a consensus about reserving a fraction of a railway network for a train. Though both EURIS and IDL allow describing the generic behaviours of interlocking systems, EURIS is more appropriate for specifying geographical-based interlockings, while IDL is more appropriate for route-based interlockings.

Furthermore, having IDL as a second DSL for specifying generic applications in our method beside ICL – a DSL for specifying configuration data presented in Chapter 4 – offers a number of advantages as elaborated in Section 3.3. This is a novelty as it has not been done – to the best of the author's knowledge – in any previously proposed methods for specification, verification and validation of railway interlocking systems.

CHAPTER 6

Formal Modelling and Verification of the Danish Interlocking Systems

This chapter elaborates how the forthcoming Danish interlocking systems are modelled and verified by applying our proposed method as outlined in Section 3.7. First, a generic behavioural model, and generic high-level safety properties created in step **DK:1** of the V&V flow for the forthcoming Danish interlocking systems are described. The generic test objectives for the forthcoming Danish interlocking systems will be described in Chapter 7. The generic behavioural model and generic safety properties are formalised in IDL – the DSL for describing generic interlocking applications – presented in Chapter 5. The model generator and the property generator (ingredients **DK:e** and **DK:f**, respectively, introduced in Section 3.7) instantiate generic behavioural model and safety properties, respectively, with concrete configuration data specified in ICL – the DSL for describing interlocking configuration data – presented in Chapter 4. Following the semantics of ICL and IDL described in Chapter 4 and Chapter 5, this instantiation results in a concrete behavioural model in the form of a Kripke structure, and concrete safety properties in the form of invariants in the resulting Kripke structure. A verification strategy using BMC and inductive reasoning is used to verify the satisfiability of concrete

safety properties on concrete systems as described in step **DK:7** of the V&V flow in Section 3.7. Experimental results on the Early Deployment Line (EDL) of the Danish Signalling Programme and comparison with nuXmv are also presented. This chapter elaborates in more detail the contribution published in [VHP15] and in a journal article [VHPss] which is under review process.

> [VHP15] – Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release". English. In: *Formal Techniques for Safety-Critical Systems*. Edited by Cyrille Artho and Peter Csaba Ölveczky. Volume 476. Communications in Computer and Information Science. Springer International Publishing, 2015, pages 223–238. ISBN: 978-3-319-17580-5. DOI: `10.1007/978-3-319-17581-2_15`. URL: `http://dx.doi.org/10.1007/978-3-319-17581-2_15`

> [VHPss] – Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release". In: *Science of Computer Programming - Special Issue: Formal Techniques for Safety-Critical Systems* (Under minor revision process)

The remainder of the chapter is organised as follows. First, some modelling assumptions are listed in Section 6.1. Section 6.2 specifies in detail a generic behavioural model of the forthcoming Danish interlocking systems in IDL. The generic high-level safety properties corresponding to this generic behavioural model are specified in Section 6.3. Section 6.4 presents our verification strategy which is a combination of BMC and inductive reasoning. The experimental results of our method and toolchain on the EDL of the Danish Signalling Programme are presented in Section 6.5. Section 6.6 compares these results with other verification techniques. Section 6.7 concludes the chapter with some related work.

6.1 Modelling Assumptions

This section summarises the assumptions that have been made in our model. The assumptions describe cases that we *do not* consider in our model.

MA-01 We do not consider shunting movements* in our model.

MA-02 When a train stops at a closed signal at the end of a route, the whole train lies within the last detection sections before the signal, called the *destination area* of the route. For simplicity, we consider only destination area of length one (detection section) in our model. This implies that trains are not longer than the length of the last detection section of a route.

*Shunting movements are movements (usually manual and not supervised by interlocking systems) of trains at low speed in the parking/maintenance area or at the platforms in stations in order to couple/decouple trains or move trains in/out of the parking/maintenance area [TVA09].

MA-03 Communication channels are perfect: communication between the interlocking system and trains, detection sections, points is perfect, i.e., there is no message loss, and the delay is admissible.

MA-04 There is no faulty equipments, e.g., no faulty point machines, or faulty detection sections.

6.2 Generic Behavioural Model

This section describes a generic behavioural model of the new Danish interlocking systems created in step **DK:1** of the development and V&V flow described in Chapter 3. The generic model is specified in IDL, which has been presented in Chapter 5. Through the rest of this chapter, for readability, named constants and their corresponding integral values are used interchangeably. We use the notation *name(integral-value)* to mean that *name* is the name of a constant having the value *integral-value*. For instance, PLUS(0) denotes a constant PLUS having the value of 0. Furthermore, integral values prefixed with $0b$ are written in their binary forms, e.g., $0b110$ is the binary value 110 (which is 6 in decimal).

The remainder of this section is organised as in the following. First, the state space of the Kripke structure modelling the forthcoming Danish interlocking systems are presented in Section 6.2.1. The initial state and the transition relation are elaborated in Section 6.2.2 and Section 6.2.3, respectively. Section 6.2.4 to Section 6.2.7 describe the detail of different parts of the transition relation.

6.2.1 State Space

As described in Chapter 2, the state space S is the set of all valuation functions $s : V \to \bigcup_{v \in V} D_v$ for which $s(v) \in D_v$ for all $v \in V$, where V is a set of variables used to represent the status of different components, such as track elements and routes, in the given interlocking system. Each variable $v \in V$ has an associated finite domain $D_v \subset \mathbb{N}_0$. The subsequent paragraphs present the set of variables V and their associated value domains D_v.

Occupancy Status. As mentioned in Chapter 2, trains can travel in any physically possible direction on a section. Thus, for each section in the network, we use, a variable for each physically possible travel direction to keep track of the occupancy status of that section in that possible travel direction.

For each linear section l there are two variables recording its occupancy status.

- $l.U2D$ – recording the occupancy status of l in the direction from its up end to its down end, and
- $l.D2U$ – recording the occupancy status of l in the direction from its down end to its up end.

Similarly, for each point p, there are three variables recording its occupancy status.

- $p.S2PM$ – recording the occupancy status from its stem end to its plus/minus end,
- $p.P2S$ – recording the occupancy status from its plus end to its stem end, and
- $p.M2S$ – recording the occupancy status from its minus end to its stem end.

The movement of trains through a given network are reflected by state transitions of the occupancy status variables of the sections in the network. This is specified in detail in Section 6.2.7 below.

The occupancy status of a section in a given travel direction is encoded using the three least significant bits `HTO` of a non-negative integer variable as shown in Figure 6.1. The value `1` of the bits `H`, `T`, `O` indicate: (`H`) the head of the train is within the section, (`T`) the tail of the train is within the section, and (`O`) the section is occupied, respectively. Note that in some cases the encoding contains redundant information, e.g., a section is obviously occupied when the head of a train is inside the section. However, the `O` bit is necessary for modelling the case where a train is occupying the section, but the head and the tail are outside the section, as it is the case for section $t3$ in Figure 6.2.

This occupancy status encoding offers two advantages: (a) the encoding can cover the case where a train occupies more than one section (e.g., when it is crossing the joint between two sections), and (b) the safety properties can be expressed efficiently using arithmetic operations on integer variables as shown in Section 6.3.

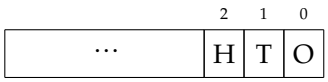


Figure 6.1: A variable recording occupancy status of a section.

Figure 6.2 shows some examples of how different values of occupancy status variables reflect how trains occupy sections. Let us assume that the trains are traveling in the direction down-to-up (left-to-right in the figure). Section $t1$ is occupied by the whole train, thus its occupancy status variable $t1.D2U$ is $0b111$ indicating that all three bits H, T, and O are set. $t2$ is occupied by the tail of a train, thus its occupancy status variable is $t2.D2U = 0b011$, $t3$ is occupied without a head or a tail of a train in it – i.e., the train is long enough to cover the whole section, thus $t3.D2U = 0b001$. Lastly, $t4$ is occupied by a head of a train, therefore its occupancy status variable is $t4.D2U = 0b101$.

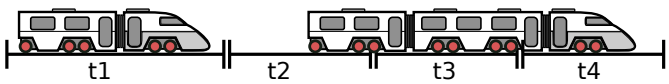


Figure 6.2: Examples of the values of occupancy status variables.

A section is vacant when all of its occupancy status variables are evaluated to zero. For example, a linear section l is vacant when both $l.D2U$ and $l.U2D$ are evaluated to zero. Since all occupancy status variables are non-negative, we can define the following macro in IDL to determine whether a linear section l is vacant.

def vacant_linear(l) =
(l.D2U + l.U2D = 0)

An analogous macro can be defined for a point p as in the following.

def vacant_point(p) =
(p.S2PM + p.P2S + p.M2S = 0)

Grouping these two macros together forms the following macro in IDL for determining whether a section e is vacant.

def vacant(e) =
(e ∈ **Linear**) ? vacant_linear(e) : vacant_point(e)

Lockable Elements. In order to accommodate sequential release feature described in Section 2.5 in our model, we consider a linear or point section as a *lockable element*. For each lockable element e in the network layout, there are two variables for representing the status of e:

- $e.MODE$ – recording the mode of the element, and
- $e.PREV$ – recording whether the previous section in the route has been released.

The possible values of $e.MODE$ are: AVAIL(0) (the element is not *exclusively* locked by a route, or used by any train), EXLCK(1) (the element is *exclusively* locked for a route), or USED(2) (the element has been used, i.e., occupied, by a train after it was *exclusively* locked for a route). The possible values of $e.PREV$ are: PENDING(0) (the previous section in the same route has not been released) and RELEASED(1) (the previous section in the same route has been released).

Point Positions. For each point p in the network layout, there are two variables:

- $p.POS$ – for representing the actual position of the point, and
- $p.CMD$ – for representing the point position commanded by the interlocking.

The possible values of $p.POS$ are: PLUS(0), MINUS(1), and INTER(2) – the position where the point is switching from plus (minus) to minus (plus). The possible values of $p.CMD$ are only PLUS(0) and MINUS(1), as the interlocking *cannot* command a point to switch to the INTER position.

Signal Aspects. For each virtual signal s in the network layout, there are two variables:

- $s.ACT$ – for representing the actual aspect of the signal as "seen" by the train, and
- $s.CMD$ – for representing the aspect of the signal as commanded by the interlocking system.

The possible values of these variables are: OPEN(1) and CLOSED(0). The values of these two variables may differ due to the delay in the communication between the interlocking system and the onboard computers in the trains.

Routes. For each route r in the interlocking table, there are three variables recording its status.

- $r.CTRL$ – for representing the current command for the route.
- $r.MODE$ – for representing the current mode of the route, and
- $r.DSPL$ – for representing the current mode of the route displayed to the output interfaces.

The following commands can be issued (manually by a signalman or automatically by a traffic management system) for a route: NOCMD(0), DISPATCH(1), or CANCEL(2). A route can be in one of the following modes: FREE(0), MARKED(1), ALLOCATING(2), LOCKED(3), or OCCUPIED(4).

Specification in IDL. The state space is specified in by the following encoding declaration in IDL.

encoding

Linear::
D2U → [**INPUT**,"unsigned int",0,0,7]
U2D → [**INPUT**,"unsigned int",0,0,7]
MODE → [**LOCAL**,"unsigned int",0,0,2]
PREV → [**LOCAL**,"unsigned int",0,0,1],

Point::
S2PM → [**INPUT**,"unsigned int",0,0,7]
P2S → [**INPUT**,"unsigned int",0,0,7]
M2S → [**INPUT**,"unsigned int",0,0,7]
CMD → [**OUTPUT**,"unsigned int",0,0,1]
POS → [**INPUT**,"unsigned int",0,0,2]
MODE → [**LOCAL**,"unsigned int",0,0,2]
PREV → [**LOCAL**,"unsigned int",0,0,1],

Signal::
ACT → [**INPUT**,"unsigned int",0,0,1]
CMD → [**OUTPUT**,"unsigned int",0,0,1],

Route::
CTRL → [**INPUT**,"unsigned int",0,0,2]
MODE → [**LOCAL**,"unsigned int",0,0,4]
DSPL → [**OUTPUT**,"unsigned int",0,0,4]

6.2.2 Initial State

The initial state s_0 is the state in which all sections are in AVAIL mode and vacant (i.e., there are no trains in the network), the commanded and actual aspects of all signals are CLOSED, all routes are in FREE mode, and the commanded and actual positions of all points are PLUS. Our encodings in Section 6.2.1 are deliberately chosen so that s_0 is the state in which all variables are evaluated to 0. In our generic behavioural model for the Danish interlocking systems, the initial state declaration is omitted from the IDL specification. Instead, the initial values from the encoding declaration are assigned to variables in the initial state.

One can also use the following initial declaration in IDL to specify the initial states.

init

[initial_state_linear]
$(\forall l : \textbf{Linear} \bullet \text{vacant}(l) \wedge l.\text{MODE} = \text{AVAIL} \wedge l.\text{PREV} = \text{PENDING}),$

[initial_state_point]
$(\forall p : \textbf{Linear} \bullet$
$(\text{vacant}(p) \wedge p.\text{MODE} = \text{AVAIL} \wedge p.\text{PREV} = \text{PENDING}) \wedge$
$(p.\text{CMD} = \text{PLUS} \wedge p.\text{POS} = \text{PLUS})),$

[initial_state_signal]
$(\forall s : \textbf{Signal} \bullet s.\text{CMD} = \text{CLOSED} \wedge s.\text{ACT} = \text{CLOSED}),$

[initial_state_route]
$(\forall r : \textbf{Route} \bullet r.\text{CTRL} = \text{NOCMD} \wedge r.\text{MODE} = \text{FREE} \wedge r.\text{DISP} = \text{FREE})$

6.2.3 Transition Relation

This section describes the generic transition relation modelling the behaviours of the Danish interlocking systems. The generic transition relation is specified in IDL using the following principles.

(a) Atomic events are specified in IDL using atomic transitions as described in Section 5.1.

(b) Different types of events/transitions are assigned priorities because different types of events in interlocking systems occur at significantly different speed, and we prefer to avoid introducing time into the model.

(c) Transitions for atomic events are combined according to their relative priorities. If two transitions have different priorities then they are combined using the prioritized choice operator $[>]$. On the other hand, if two transitions have the same priority, they are combined using the non-deterministic choice $[=]$. These operators and their meaning have been explained in Chapter 5.

Overview of Transitions and their Priorities. We group the transitions of atomic events of an interlocking system into four types, such that the transitions of each type have the same priority and they can be combined using the $[=]$ operator to represent the combined behaviour of the transitions of that type. For readability, each type of transitions is specified by a module in the IDL specification. The types and their corresponding modules are as follows:

(0) DP – modelling route dispatching transitions. This module is described in detail in Section 6.2.4.

(1) SUT – modelling interlocking controller transitions, e.g., setting the mode of a route. This module is explained in Section 6.2.5.

(2) ET – modelling track element transitions, e.g., switching a point, or communicating a signal aspect to a train. This module is explained in Section 6.2.6.

(3) TM – modelling train movement transitions. This module is explained in Section 6.2.7.

Transitions of type (0) are not prioritized, i.e., they can be taken nondeterministically whenever they are enabled, independently from other transitions. On the other hand, transitions of types (1), (2), and (3) are prioritized in the descending order that they appear in the list, i.e., transitions of type (1) have the highest priority and transitions of type (3) have the lowest. This priority of transitions is based on the intuition that in practice, the events in the interlocking controller occur at significantly higher speed than the ones occurring in a track element. For example, a cycle of a route controller occurs within a hundred milliseconds, while switching a point from one position to another may take a few seconds. An analogous argument applies to events related to track elements compared to events related to train movements. For instance, switching a point may take a few seconds, while it may take a train minutes to pass a section.

With this grouping, SUT models the behaviours of the system under consideration – the interlocking controller; while DP , ET , and TM model the behaviours of the operational environment that interacts with the interlocking controller. The interlocking system is a closed system that evolves according to the combined transition relation as specified in IDL as in the following.

[DP] [=] ([SUT] [>] [ET] [>] [TM])

The route dispatching transitions DP is given the same priority as the priority given to the combined transition relation specifying all the other transitions in order to allow routes to be dispatched arbitrarily. If route dispatching transitions were given higher priority than the one given to any of the other transitions, all routes which could be dispatched would have to be dispatched before interlocking controller, track elements, or trains could make any transitions. On the other hand, if route dispatching were given lower priority than any of the other transitions, then a route could not be dispatched, if another route is processed by the interlocking controller, or a track element or a train could make a transition.

In the subsequent subsections, we explain in detail of the transitions of each type. For readability, the informal explanation of a transition is described first, followed by the formal specification in IDL.

6.2.4 Route Dispatching Transitions

The route dispatching transitions of a given interlocking system, specified by the module DP, model how routes are dispatched and cancelled as the results of manual requests from signalmen or automatic requests from a traffic management system. Figure 6.3 shows different control commands for a route *r*, i.e., the different values of the variable r.CTRL, and the transitions from one command to another. In the initial state, there are no commands for *r*, i.e., r.CTRL = NOCMD. Signalmen or traffic management systems can then command to dispatch *r*, i.e., r.CTRL = DISPATCH. When the command has been acknowledged by the interlocking controller, and the route has been dispatched accordingly, r.CTRL is set back to NOCMD. Cancellation of a route can be commanded at any time, unless a cancellation has been issued. Once cancellation command is issued, the command remains until the route becomes free.

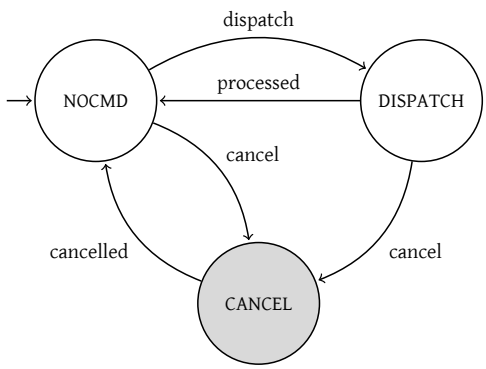


Figure 6.3: Route dispatching transitions: how the value of r.CTRL is changed.

Route Dispatch. A route *r* can be dispatched arbitrarily whenever its mode is FREE and has not been already dispatched. This means that multiple routes can be dispatched before any of them are processed by the interlocking controller, and routes can be dispatched when other routes are being processed. Upon dispatching, the route's control command changes from NOCMD to DISPATCH.

Guard A route *r* can be dispatched if all of the following hold

- Its mode is FREE(0).
- It has not been already dispatched, i.e., its control command is NOCMD(0).

Update

- The variable r.CTRL is set to DISPATCH(1).

The transition is specified in IDL as in the following.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route} \bullet \\
\ [\mathrm{ctrl\_nocmd\_to\_dispatch}]\ (\mathrm{r.CTRL} = \mathrm{NOCMD} \wedge \mathrm{r.DSPL} = \mathrm{FREE}) \longrightarrow \\
\ (\mathrm{r.CTRL}' = \mathrm{DISPATCH}))
\end{array}
$$

Once the route is dispatched and it is processed by the interlocking controller the control command for the route can be reset.

Guard

- The route's current command is DISPATCH(1)
- The route's mode is not FREE(0)

Update

- The route's control command is set to NOCMD(0)

The transition is specified in IDL as in the following.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route} \bullet \\
\ [\mathrm{ctrl\_dispatch\_to\_nocmd}]\ (\mathrm{r.CTRL} = \mathrm{DISPATCH} \wedge \mathrm{r.DSPL} \neq \mathrm{FREE}) \longrightarrow \\
\ (\mathrm{r.CTRL}' = \mathrm{NOCMD}))
\end{array}
$$

Route Cancellation. A route *r* that is processed by the interlocking controller can be commanded to be cancelled if the route is not used yet.

Guard

- *r*'s mode is one of the following: MARKED(1), ALLOCATING(2), or LOCKED(3)
- There is not a pending cancellation command for the same route, i.e., its control command is not CANCEL(2)

Update

- r's control command is changed to CANCEL(2)

$$
\begin{array}{l}
([=]\ \mathtt{r}\ :\ \textbf{Route} \bullet \\
\ [\,\text{ctrl\_to\_cancel}\,] \\
\ (\mathrm{r}.\mathrm{CTRL} \neq \mathrm{CANCEL} \wedge \\
\quad (\mathrm{r}.\mathrm{DSPL} = \mathrm{MARKED} \vee \mathrm{r.DSPL} = \mathrm{ALLOCATING} \vee \mathrm{r.DSPL} = \mathrm{LOCKED})) \\
\longrightarrow (\mathrm{r.CTRL}' = \mathrm{CANCEL}))
\end{array}
$$

The cancellation command is reset when the route's mode is back to FREE(0) and its source signal's actual aspect is CLOSED(0).

Guard

- r's control command is CANCEL(2)
- r's mode is FREE(0) and its source signal's actual aspect is CLOSED(0)

Update

- r's control command is reset to NOCMD(0)

$$
\begin{array}{l}
([=]\ \mathtt{r}\ :\ \textbf{Route} \bullet \\
\ [\text{ctrl\_cancel\_to\_nocmd}] \\
\ (\mathrm{r}.\mathrm{CTRL} = \mathrm{CANCEL} \wedge \mathrm{r.DSPL} = \mathrm{FREE} \wedge \mathbf{src}(\mathrm{r}).\mathrm{ACT} = \mathrm{CLOSED}) \\
\longrightarrow (\mathrm{r.CTRL}' = \mathrm{NOCMD}))
\end{array}
$$

6.2.5 Interlocking Controller Transitions

The interlocking controller transitions of a given interlocking system, represented by SUT, model the internal behaviours of the interlocking controller. In particular, they model how the status of routes and lockable elements change in response to the changes in the environment. The subsequent paragraphs describe the transitions in detail.

Life-cycle of a Route. Figure 6.4 shows the life-cycle of a route, i.e., its different modes and the transitions from one mode to another. The life-cycle shown in Figure 6.4 reflects the procedure for setting and sequentially releasing a route r as described in Section 2.5. The transitions labelled (1), (2), (3), (4), (5c), and (6) in Figure 6.4 correspond to items (1) – (6) in the procedure presented in Section 2.5.2 for setting and releasing a route. Transitions (5a) and (5b) model the sequential release that can take place while the route stays in OCCUPIED mode: as the train moves along the route, its elements are used (5a) when the train enters them, and then they are released sequentially (5b) as soon as the train has passed them. Transition (2) is adapted to sequential release: allocating resources for a given route r is now also allowed even if a conflicting route r' is in the OCCUPIED mode, given that the

elements shared between r and r' have been sequentially released. The transitions are elaborated in the subsequent paragraphs.

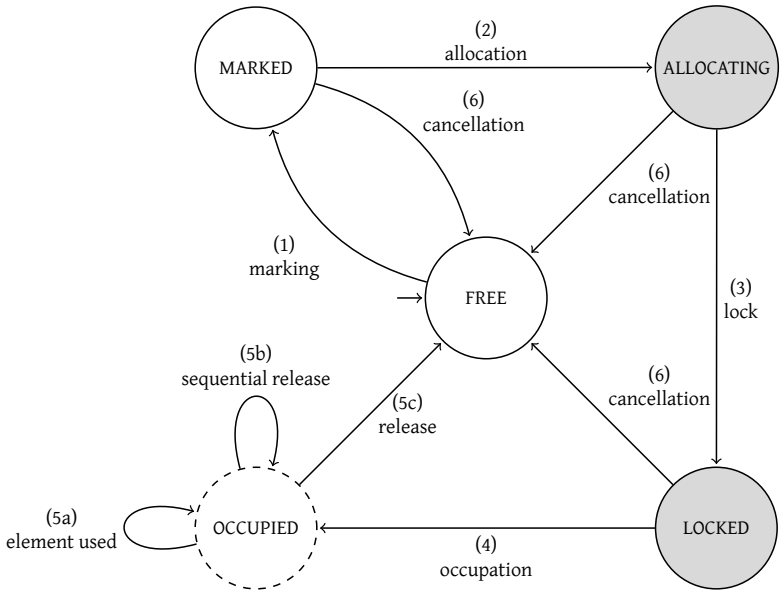


Figure 6.4: Life-cycle for route r, showing how the value of $\mathrm{r.MODE}$ is changed.

Route Marking. A route r is marked as requested when its control command is DISPATCH and it is in FREE mode.

Guard

- r's control command is DISPATCH(1)
- r's mode is FREE(0)

Update

- r's mode is changed to MARKED(1)

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route} \bullet \\
\ [\mathrm{route\_marking}]\ (\mathrm{r.CTRL} = \mathrm{DISPATCH} \wedge \mathrm{r.MODE} = \mathrm{FREE}) \longrightarrow \\
\ (\mathrm{r.MODE}' = \mathrm{MARKED} \wedge \mathrm{r.DSPL}' = \mathrm{MARKED}))
\end{array}
$$

Route Allocation.

Guard A route *r* can be allocated if

- It is in MARKED(1) mode.
- None of the conflicting routes are in ALLOCATING(2) or LOCKED(3) modes.
- All detection sections in the route's path and overlap are vacant†.
- All elements in *r*'s path are in AVAIL(0) mode.
- None of the elements in *r*'s overlap is in USED(2) mode‡.
- All protecting points are in AVAIL(0) mode or they are already in the positions as required by the route *r*.

Update

- The mode of the route is updated to ALLOCATING(2) and
- The interlocking starts allocating the resources for *r* by
  - commanding points to switch to required position
  - commanding protecting signals to switch to CLOSED(0)
  - lock *exclusively* all elements in *r*'s path, settings their modes to EXLCK(1)

```
([=] r : Route •
 [route_allocating]
 (r.MODE = MARKED) ∧
 /* none of the conflicting routes in ALLOCATING(2) or LOCKED(3) modes
  */
 (∀cr : Route •
  (cr ∈ conflicts (r)) ⇒ (cr.MODE ≠ ALLOCATING ∧ cr.MODE ≠ LOCKED)) ∧
 /* all detection sections in the path and overlap are vacant */
 (∀e : Section •
  e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e)) ∧
 /* all elements in route's path are in AVAIL mode */
 (∀e : Section • e ∈ path(r) ⇒ (e.MODE = AVAIL)) ∧
 /* all elements in route's overlap are not in USED mode */
 (∀e : Section • e ∈ overlap(r) ⇒ (e.MODE ≠ USED)) ∧
 /* all protecting points are in AVAIL mode,
  * or are already in the correct position */
 (∀e : Point •
  e ∈ (dom points(r) \ elems path(r)) ⇒
  (e.MODE = AVAIL ∨ e.POS = req(r,e)))
 ⟶
```

†This check is needed as the route and its elements may all have been released, but the train is in the standstill position on the last detection section of the route, at the closed signal. In such cases, the route *must not* be allocated.

‡The elements in the overlap can be in EXLCK(1) mode for a successive route.

$(r.\mathrm{MODE}' = \mathrm{ALLOCATING}) \wedge (r.\mathrm{DSPL}' = \mathrm{ALLOCATING}) \wedge$
/* *command points* */
$(\forall p : \mathbf{Point} \bullet p \in \mathbf{points}(r) \Rightarrow (p.\mathrm{CMD}' = \mathbf{req}(r,p))) \wedge$
/* *command signals* */
$(\forall s : \mathbf{Signal} \bullet s \in \mathbf{signals}(r) \Rightarrow (s.\mathrm{CMD}' = \mathrm{CLOSED})) \wedge$
/* *lock exclusively all elements in the route's path* */
$(\forall e : \mathbf{Section} \bullet e \in \mathbf{path}(r) \Rightarrow (e.\mathrm{MODE}' = \mathrm{EXLCK})))$

Route Lock.

Guard A route *r* can be locked if the following conditions are fulfilled:

- The route's mode is ALLOCATING(2)
- The route is *allocated*
  - All protecting signals' actual aspect are CLOSED(0)
  - All points' actual positions are as required by *r*
  - All elements in the *r*'s path and overlap are vacant
  - All elements in *r*'s path are locked exclusively for *r*

Update

- the route is set to LOCKED(3) mode, and
- its source signal is commanded to be OPEN(1).

$([=]\ r : \mathbf{Route} \bullet$
[route_lock]
$r.\mathrm{MODE} = \mathrm{ALLOCATING} \wedge$
/* *protecting signals' actual aspects are as required* */
$(\forall s : \mathbf{Signal} \bullet s \in \mathbf{signals}(r) \Rightarrow (s.\mathrm{ACT} = \mathrm{CLOSED})) \wedge$
/* *points' actual positions are as required* */
$(\forall p : \mathbf{Point} \bullet p \in \mathbf{points}(r) \Rightarrow (p.\mathrm{POS} = \mathbf{req}(r,p))) \wedge$
/* *all detection sections in the path and overlap are vacant* */
$(\forall e : \mathbf{Section} \bullet$
$e \in (\mathbf{elems\ path}(r) \cup \mathbf{elems\ overlap}(r)) \Rightarrow \mathrm{vacant}(e)) \wedge$
/* *all elements in the route's path are locked exclusively* */
$(\forall e : \mathbf{Section} \bullet e \in \mathbf{path}(r) \Rightarrow (e.\mathrm{MODE} = \mathrm{EXLCK}))$
$\longrightarrow r.\mathrm{MODE}' = \mathrm{LOCKED} \wedge r.\mathrm{DSPL}' = \mathrm{LOCKED} \wedge \mathbf{src}(r).\mathrm{CMD}' = \mathrm{OPEN})$

Route Occupation.

Guard The route *r* is occupied if

- the route is in LOCKED(3) mode
- the route is first occupied by the train, i.e., the first detection section right after the source signal becomes occupied.

Update

- The route's mode is set to OCCUPIED(4)
- The first element's mode is set to USED(2)
- The source signal is also commanded to be CLOSED(0)[§].

```
([=] r : Route •
 [route_in_use]
 /* the first element of the route is occupied */
 let e = first(r) in
   r.MODE = LOCKED ∧ ¬vacant(e)
 end
 ⟶
 r.MODE′ = OCCUPIED ∧ r.DSPL′ = OCCUPIED ∧ src(r).CMD′ = CLOSED ∧
  first(r).MODE′ = USED)
```

Sequential Release. Figure 6.5 depicts the life-cycle of a lockable element e within the network controlled by a given interlocking system. Each node in the diagram in Figure 6.5 is labelled with the following information about the status of the element e, the information changes in each transition is marked red.

- vacant(e) – indicating whether the element is vacant where vacant(e) is a formula over occupancy status variables of e as shown in Section 6.2.1;
- its current mode – i.e., the value of $e.MODE$ as described in Section 6.2.1; and
- the value of the $e.PREV$ variable indicating whether the previous element $prev(r, e)$ of e in the route r has been released

The life-cycle of a lockable element can be summarised as in the following where the numbering of the items corresponds to the labels of transitions in Figure 6.5.

(0) An element e is initially in a state where it is vacant, in AVAIL(0) mode, and its $PREV$ variable is PENDING(0).

(1) When the interlocking controller is allocating a route r whose path contains e, e.MODE is set to EXLCK(1), meaning that e is locked exclusively for r.

(2) The element becomes occupied, i.e., $\neg vacant(e)$, as a train enters.

(3) The interlocking controller detects the change in the occupancy status of e, consequently sets e.MODE to USED(2).

[§]In practice, the interlocking has a "memory circuit" which ensures that the source signal will not be opened again for the same route until the route is released. However, in our model, the source signal is only set to OPEN in the transition from ALLOCATING to LOCKED, and is never set again when the route is in the USED mode. Additionally, trains are not allowed to reverse in our model, thus this guarantees that the source signal will not be open again for the same route until the route is released.

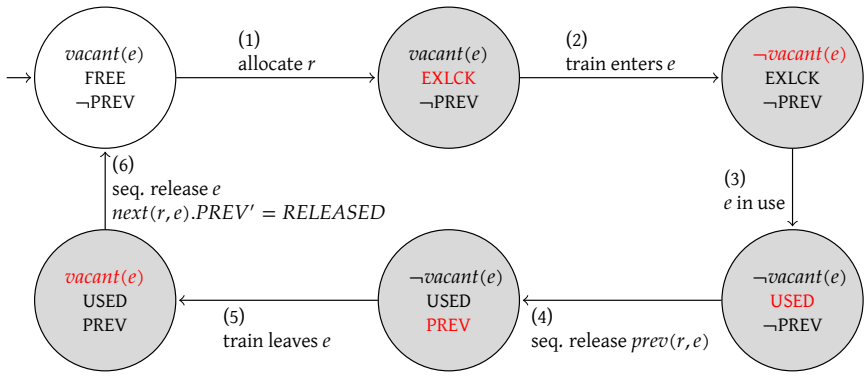


Figure 6.5: A life-cycle of a lockable element e in a route r

(4) When the train leaves the previous element $prev(r,e)$ of e in the route r, $prev(r,e)$ is released, and it informs e by setting the variable $e.PREV$ to RELEASED(1).

(5) When the train leaves e, the element becomes vacant again.

(6) The interlocking controller again detects the change, releases e and informs the next element $next(r,e)$ in the same route by setting $next(r,e).PREV$ to RELEASED(1).

(1) has been specified in route allocation transition. (2) and (5) will be specified in train movement transitions in Section 6.2.7. (3) and (6) are specified by (SR-1) and (SR-2), respectively, in the following. (4) is specified for $prev(r,e)$ analogously to (SR-2).

(SR-1) *Element in Use*: For each element e in the route's path (except the first element) we have the following transition:

Guard an element e is in use when

- The route is in OCCUPIED(4) mode
- The element is in EXLCK(1) mode
- the train starts occupying the element
- the previous element of e in r, $prev(r,e)$, is in USED(2) mode.
- if e is a point, its actual position should be as required by r
- if e is not the last element of the route, i.e., $e \neq last(r)$, the next element of e should be in EXLCK(1) mode.

Update

- the element changes to USED(2) mode

```
([=] r : Route •
 ([=] e : Section •
  [element_in_use]
  e ∈ path(r) ∧ e ≠ first(r) ∧ prev(r,e).MODE = USED ∧
  r.MODE = OCCUPIED ∧ e.MODE = EXLCK ∧ not vacant(e) ∧
  ((e ∈ Point) ⇒ e.POS = req(r,e)) ∧
  (e ≠ last(r) ⇒ next(r,e).MODE = EXLCK)
  ⟶ e.MODE′ = USED))
```

(SR-2) *Sequential Release of Elements*: For each element *e* that is in the route's path and is not the last element, we have the following transitions:

Guard an element *e* is released when

- the route is in OCCUPIED(4) mode
- the element is in USED(2) mode
- the train has passed the element, i.e., *e* is vacant
- the previous element of *e* in *r* must have been released if *e* is not the first element in the route *r*
- the next element of *e* in *r*:
  - is in USED(2) mode
  - must have PREV = PENDING(0).
  - if it is a point, it should be in correct position as required by *r*
  - it should have the tail of the train on it[¶], i.e., the T bit of the HTO variable is 1.
- if *e* is a point, it must be in the correct position as required by the route[‖]

Update

- the element *e*'s mode changes to AVAIL(0) and e.PREV is reset to PENDING(0).
- inform the next element of *e* that it can be released whenever the train passes it, i.e., set the next element's PREV to RELEASED(1).

```
([=] r : Route •
 ([=] e : Section •
  [sequential_release_e]
  e ∈ path(r) ∧ e ≠ last(r) ∧ r.MODE = OCCUPIED ∧
  e.MODE = USED ∧ vacant(e) ∧ (e ≠ first(r) ⇒ e.PREV = RELEASED) ∧
  let nx = next(r,e) in
    nx.PREV = PENDING ∧ nx.MODE = USED ∧ (_T_(hto(nx,r,0)) ≠ 0) ∧
    ((nx ∈ Point) ⇒ nx.POS = req(r,nx))
  end ∧ ((e ∈ Point) ⇒ e.POS = req(r,e))
  ⟶ e.MODE′ = FREE ∧ e.PREV′ = PENDING ∧ next(r,e).PREV′ = RELEASED))
```

¶This to make sure we are informing the next element correctly.

‖This condition ensures we are releasing the element of the right route.

Route Release. With sequential release, the route r is released when the last element $e = last(r)$ of the route is released. The route can be released by one of the following transitions.

(RR-1) *Sequential Release the Last Element.* When r's next route is locked and the train moves towards the first section of the next route. The current route r and its last section e will be released when the train has completely left e.

Guard

- the route r is in OCCUPIED(4) mode
- the element e is in USED(2) mode
- the train has passed the element e, i.e., it is vacant
- the previous element of e in r must have been released if e is not the first element.
- if e is a point, it must be in the position required by r.

Update

- the element e's mode changes to AVAIL(0), and its PREV variable is reset to PENDING.
- the route is released, i.e., its mode is set to FREE(0)

```
([=] r : Route •
 [sequential_release_last_elem]
 r.MODE = OCCUPIED ∧
 let e = last(r) in
   e.MODE = USED ∧ vacant(e) ∧ (e ≠ first(r) ⇒ e.PREV = RELEASED) ∧
   ((e ∈ Point) ⇒ e.POS = req(r,e))
 end
 ⟶
 last(r).MODE′ = AVAIL ∧ last(r).PREV′ = PENDING ∧ r.MODE′ = FREE ∧
 r.DSPL′ = FREE)
```

(RR-2) *Pseudo Timer.* Alternatively, when the train occupies the last section $e = last(r)$ of the route r, a timer is started. The last section e and the route r are released when the timer expires. The timer ensures that the train has come to a standstill in front of the destination signal[**]. We do not have time in our model, thus we just release the last section when the train comes to standstill within the last section.

Guard

- the route r is in OCCUPIED(4) mode
- the last element e is in USED(2) mode
- the whole train is in the last section e, i.e., the HTO variable has all three bits set.

[**] in this case, the whole train is contained within the last detection section, cf. Assumption **MA-02** in Section 6.1

- the destination signal's actual aspect is CLOSED(0)
- the previous element of *e* in *r* has been released if e is not the first element of *r*.

Update

- the element *e* is released, i.e., its mode changes to AVAIL(0) and its PREV variable is set to PENDING(0).
- the route *r* is released, i.e., its mode is set to FREE(0)

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route} \bullet \\
\ [\text{release\_last\_elem\_pseudo\_timer}] \\
\ \mathrm{r.MODE} = \mathrm{OCCUPIED} \wedge \\
\ \textbf{let}\ \ \mathrm{e} = \textbf{last}(\mathrm{r})\ \ \textbf{in} \\
\quad \mathrm{e.MODE} = \mathrm{USED} \wedge \mathrm{hto(e,r,0)} = \mathrm{0b111} \wedge \textbf{dst}(\mathrm{r}).\mathrm{ACT} = \mathrm{CLOSED} \wedge \\
\quad (\mathrm{e} \neq \textbf{first}(\mathrm{r}) \Rightarrow \mathrm{e.PREV} = \mathrm{RELEASED}) \\
\ \textbf{end} \\
\longrightarrow \\
\ \textbf{last}(\mathrm{r}).\mathrm{MODE}' = \mathrm{AVAIL} \wedge \textbf{last}(\mathrm{r}).\mathrm{PREV}' = \mathrm{PENDING} \wedge \mathrm{r.MODE}' = \mathrm{FREE} \wedge \\
\ \mathrm{r.DSPL}' = \mathrm{FREE})
\end{array}
$$

Route Cancellation.

Guard

- *r*'s control command is CANCEL(2)
- One of the following holds:
  - *r*'s mode is MARKED(1)
  - *r*'s mode is ALLOCATING(2), and all the points *p* required by *r* are not switching, i.e., p.POS = p.CMD
  - *r*'s mode is LOCKED(3)[††], and all sections in *r*'s path and overlap are vacant.

Update

- *r*'s mode is set to FREE(0)
- close *r*'s source signal if *r*'s mode was LOCKED(3)
- cancel all commands to points if *r*'s mode was ALLOCATING(2)
- unlock all elements in *r*'s path if *r*'s mode was ALLOCATING(2) or LOCKED(3)

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route} \bullet \\
\ [\text{cancel\_marked\_route}]\ (\mathrm{r.CTRL} = \mathrm{CANCEL} \wedge \mathrm{r.MODE} = \mathrm{MARKED}) \longrightarrow \\
\ (\mathrm{r.MODE}' = \mathrm{FREE} \wedge \mathrm{r.DSPL}' = \mathrm{FREE}))
\end{array}
$$

[††] this implies all the points required by *r* are not switching

```
([=] r : Route •
 [cancel_allocating_route]
 (r.CTRL = CANCEL ∧ r.MODE = ALLOCATING ∧
  /* no points are switching, if there is a point switching, we wait until the
   * next cycle when the point is already done switching and then cancel the
   * route */
  (∀p : Point • p ∈ points(r) ⇒ p.POS = req(r,p)))
 ⟶
 /* free the route */
 (r.MODE′ = FREE) ∧ (r.DSPL′ = FREE) ∧
 /* canceling the command to points, we don't canceling the protecting point
  * because it may be used by other routes */
 (∀p : Point • p ∈ path(r) ⇒ (p.CMD′ = p.POS)) ∧
 /* and unlock all sections in the route's path */
 (∀e : Section • e ∈ path(r) ⇒ (e.MODE′ = AVAIL)))

([=] r : Route •
 [cancel_locked_route]
 (r.CTRL = CANCEL ∧ r.MODE = LOCKED ∧
  /* can only be canceled if the route has not been used, i.e., all the
   * route's path and overlap are still vacant */
  (∀e : Section •
   e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e)))
 ⟶
 /* free the route */
 (r.MODE′ = FREE) ∧ (r.DSPL′ = FREE) ∧
 /* close the source signal */
 (src(r).CMD′ = CLOSED) ∧
 /* and unlock all sections in the route's path */
 (∀e : Section • e ∈ path(r) ⇒ (e.MODE′ = AVAIL)))
```

6.2.6 Track Element Transitions

The track elements transitions of a given interlocking system, represented by ET, models the behaviour of track-side elements in the railway network under control, i.e., how points are switched and how signal aspects are communicated from the interlocking controller to the onboard computer in a train.

Switching Points. A point p can be switched if it is commanded to be switched to a position $p.CMD$ that is different from its current position $p.POS$. The point switching process occurs in two steps, first the point moves from its current position to the *intermediate* position, and then the point is switched from the *intermediate* position to the requested position. Figure 6.6 illustrates the point switching steps from PLUS position to MINUS position.

For each point p, there are two transition rules for switching the point as described in the following.

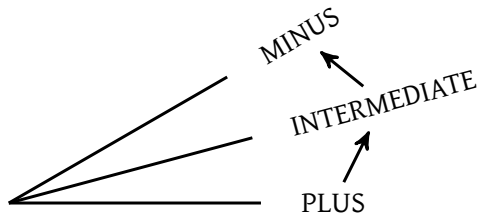


Figure 6.6: Point switching steps from PLUS to MINUS

(PS-1) The point moves from its current position to the intermediate (INTER) position, in other words it is switching toward a new position

Guard

- the commanded position is different from the actual position of the point, and
- the actual position of the point is not INTER(2).

Update

- The point's actual position changes to INTER(2)

$([=]\ \mathrm{p} : \textbf{Point} \bullet$
$[\mathrm{point\_switch\_1}]\ \mathrm{p.POS} \neq \mathrm{p.CMD} \wedge \mathrm{p.POS} \neq \mathrm{INTER} \longrightarrow \mathrm{p.POS}' = \mathrm{INTER})$

(PS-2) The point is completely switched to the commanded position

Guard

- The point is in INTER(2) position[‡‡]

Update

- The actual position of the point is set to the position commanded by the interlocking

$([=]\ \mathrm{p} : \textbf{Point} \bullet\ [\mathrm{point\_switch\_2}]\ \mathrm{p.POS} = \mathrm{INTER} \longrightarrow \mathrm{p.POS}' = \mathrm{p.CMD})$

Communicating Signal Aspects. For each signal s, there is a transition rule for changing the setting of the signal:

$([=]\ \mathrm{s} : \textbf{Signal} \bullet$
$[\mathrm{communicate\_signal\_aspect}]\ \mathrm{s.ACT} \neq \mathrm{s.CMD} \longrightarrow \mathrm{s.ACT}' = \mathrm{s.CMD})$

It states that whenever the actual aspect $s.ACT$ of the signal s differs from its commanded aspect $s.CMD$, the actual aspect of the signal is updated to the commanded aspect.

[‡‡]This guard shall be stronger in practice in order to prevent the case where the point might be in INTERMEDIATE position because of a failure. However, we do not consider failures yet in our model, thus the guard here is adequate.

6.2.7 Train Movements Transitions

Trains are not explicitly specified in our model, in the sense that there are no explicit train objects. Instead, train movements and related aspects are implicitly modelled via the occupancy status of sections, inspired by the "rubber-band" model described in [AT12]. This implicit model is advantageous compared to the explicit one, because it can model arbitrary numbers of trains of arbitrary length. Therefore, we do not have to investigate how many trains we should put in the model and how long trains should be for the safety proof to be sound. Additionally, in the implicit model of train movements, train length – in terms of numbers of sections that a train occupies – may vary as trains move. This variation reflects the actual view of interlocking systems of the train length: although trains have fixed geometric length, their length – in terms of the number of sections that they occupy – as seen by the interlocking systems is not fixed.

Trains in our model are assumed to be well-behaved, meaning that all of the following hold.

TA-01 Trains always move according to the actual settings of the physical railway network under consideration, e.g., if the actual position is PLUS (MINUS) then trains only move from stem end of the point to the plus (minus) end of the point and vice versa.

TA-02 Trains always stop in front of a signal whose actual aspect is CLOSED, and they only proceed when authorized by a signal whose actual aspect is OPEN.

TA-03 Trains do not "fly" – i.e., they do not move from a section t_a to a further section t_b without making a continuous path through the intermediate sections between t_a and t_b.

TA-04 Trains can change their travel direction but do not reverse.

These assumptions are well justified for the following reasons. First, in ETCS Level 2, the sophisticated onboard computer systems in a fitted train ensure that the train would not go further than what it is authorized to move forward and that the train does not accidentally reverse [ERT14, chap. 3]. An emergency brake will be triggered to bring a train to a full-stop if the train violates its movement authority. Second, the primary causes of train flying as seen by interlocking systems are failures in the train detection systems (also known as Track Occupancy Detection systems) which detect whether a section is vacant or occupied by a train [TVA09]. New interlocking systems use axle counters which are far more reliable than track circuits used by legacy systems for train detection. Thus, for simplicity at the high level design, failures in train detection systems are not included in the model, hence eliminating the possibilities of having flying trains.

The movements of a train as seen by interlocking systems are like an "elastic band" as illustrated in Figure 6.7. The band extends to the next section in front of the train when its head starts occupying the next section – movement from (a) to (b)

in Figure 6.7. The band shrinks when the train leaves the section occupied by its tail – movement from (b) to (c) in Figure 6.7.

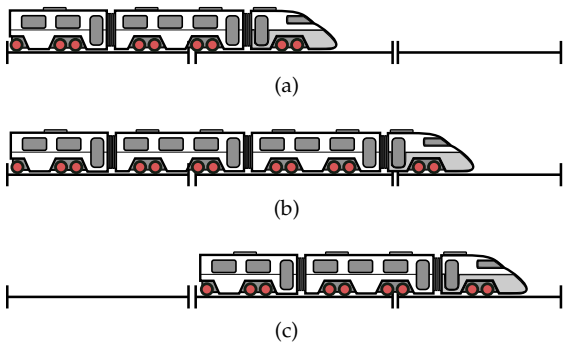


Figure 6.7: "Elastic band" train movements.

Train movements in the "elastic band" model are categorized into the following types: (1) head movements, (2) tail movements, (3) change direction movements (only on linear sections), (4) entering interlocked area movements, and (5) leaving interlocked area movements. The *interlocked area* is a network or the fraction of a network under control of the considered interlocking system. The head movements and tail movements respectively model the extensions and shrinking shown in Figure 6.7. The movements of type (3) model the case where a train changes its travel direction. Note that change direction movements differ from reversing movements: in a change direction movement, the driver has to move from the front driver cab to the back driver cab and drives the train forward (w.r.t. the direction that the driver cab is facing), while in a reversing movement the driver stays in the front driver cab and drives the train backward [ERT14, Subset 026, Sect. 5.12-13]. Change direction movements are allowed in our model of train movements, while reversing movements are not allowed, as stated in assumption **TA-04**. Note that the interlocking controller cannot distinguish between change direction movements and reversing movements because they result in the same changes in occupancy statuses of sections (the only way the interlocking controller "sees" the train movements). Therefore the interlocking controller deals with them in the same way. It is the responsibility of the onboard system to ensure that the train does not reverse, unless explicitly authorized [ERT14, Subset 026, Sect. 3.14-15]. Thus, handling reversing movements is beyond the scope of interlocking systems. The movements of type (4) and (5) account for the movements at the boundary (see Section 4.2 for the boundary configuration) of the network under control. In other words, they model how trains are put into the network and taken out of the network under consideration. The subsequent paragraphs describe in details the movements of each type.

In the subsequent paragraphs, the macro names in the text in the guards and updates – e.g., head_leaves – are hyperlinks, they are linked to their respective

specification in Appendix D. One can follow the link to see the full specification of the macros.

Head Movements. A head movement can occur on a non-boundary (see Section 4.2) section e if it is occupied by a train and the head of the train is in the section, i.e., the H and O bits of the variable representing the occupancy status of e are set (see Section 6.2.1). If e is a linear section and has a signal s intended for the same direction as the direction of the train, then the actual aspect of s has to be OPEN. The effect of the head movement is illustrated in Figure 6.8: the head of the train leaves e, effectively toggles the H bit of e's occupancy status variable; and then the head of the train enters the next section in the travel direction – in this case $up(e)$, consequently toggles the H and O bits of the occupancy status variable of $up(e)$.

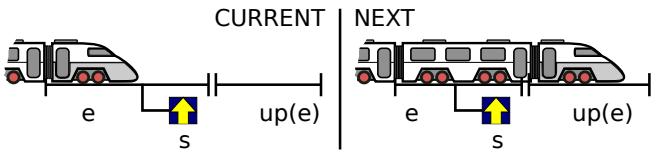


Figure 6.8: Example of head movement on section e from down to up.

The conditions and effects on occupancy status variables can be modelled efficiently by bit-wise AND (&), OR (|), and XOR ($\oplus$) operators and arithmetic shift left ($\ll$) and arithmetic shift right ($\gg$) operators. The following proposition models the head movement transition for a linear section.

Guard

- e is not a boundary section, and its neighbouring section in the travel direction is not a boundary section. The movements at boundary sections are specified by entering/leaving interlocked area movements explained in the subsequent paragraphs. The conditions determining boundary sections are specified by is_boundary_sec_up and is_boundary_sec_down macros. Detail about these macros can be found in Appendix D.
- The head of the train has to be in the current section e, i.e., the H and O bits of the occupancy status variable is on. This is specified by occupied_with_head macro.
- if there is signal mounted along e in the direction of travel, it has to be in OPEN(1) aspect.

Update

- the `H` bit is toggled in current section (specified by head_leaves macro), and
- the `H` and `O` bits are toggled for the next section (specified by head_enters_next macro).

([=] l : **Linear** •
[head_movement_linear_up]
up(l) $\wedge$ $\neg$is_boundary_sec_up(**up**(l)) $\wedge$ $\neg$is_boundary_sec_down(l) $\wedge$
occupied_with_head(l.D2U) $\wedge$ ($\neg$**up_sig**(l) $\vee$ **up_sig**(l).ACT = OPEN)
$\longrightarrow$ head_leaves(l.D2U,l.D2U$'$) $\wedge$ head_enters_next(**up**(l),l))

([=] l : **Linear** •
[head_movement_linear_down]
down(l) $\wedge$ $\neg$is_boundary_sec_down(**down**(l)) $\wedge$ $\neg$is_boundary_sec_up(l) $\wedge$
occupied_with_head(l.U2D) $\wedge$ ($\neg$**down_sig**(l) $\vee$ **down_sig**(l).ACT = OPEN)
$\longrightarrow$ head_leaves(l.U2D,l.U2D$'$) $\wedge$ head_enters_next(**down**(l),l))

If e is a point, then depending on the point's position, the corresponding next section will be updated:

- If the train is travelling from stem end toward plus/minus ends and the point's position is PLUS (MINUS) then the next section is the neighboring section connected to e's plus (minus) end.
- If the train is travelling from plus/minus ends toward stem end, then the next section is the neighboring section connected to e's stem end.

([=] p : **Point** •
[head_movement_point_stem_to_plus] occupied_with_head(p.S2PM) $\wedge$ p.POS = PLUS
$\longrightarrow$ head_leaves(p.S2PM,p.S2PM$'$) $\wedge$ head_enters_next(**plus**(p),p))

([=] p : **Point** •
[head_movement_point_stem_to_minus]
occupied_with_head(p.S2PM) $\wedge$ p.POS = MINUS
$\longrightarrow$ head_leaves(p.S2PM,p.S2PM$'$) $\wedge$ head_enters_next(**minus**(p),p))

([=] p : **Point** •
[head_movement_point_plus_to_stem] occupied_with_head(p.P2S) $\wedge$ p.POS = PLUS
$\longrightarrow$ head_leaves(p.P2S,p.P2S$'$) $\wedge$ head_enters_next(**stem**(p),p))

([=] p : **Point** •
[head_movement_point_minus_to_stem]
occupied_with_head(p.M2S) $\wedge$ p.POS = MINUS
$\longrightarrow$ head_leaves(p.M2S,p.M2S$'$) $\wedge$ head_enters_next(**stem**(p),p))

Tail Movements. A tail movement can occur on a non-boundary section e if it is occupied by a train and the tail of the train is within the section, while the head of the train is not in e. This means that e's occupancy status variable has the T and O bits set, and the H bit unset, i.e., its value is $0b011$. Figure 6.9 illustrates the effect a tail movement: the tail of the train vacates e, hence resetting e's occupancy status variable to 0; then the tail of the train moves to the next section in the travel direction – in this case $up(e)$, consequently toggles the T bit of the occupancy status variables of $up(e)$.

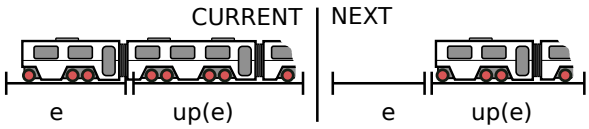


Figure 6.9: Example of tail movement on section *e* from down to up.

Guard

- *e* is not a boundary section, and its neighbouring section in the travel direction is not a boundary section. Conditions for boundary sections have been explained in the above paragraph about head movements.
- The tail of the train is in the current section, while the head is not. This is specified by occupied_with_only_tail macro.

Update

- The current section is set to vacant for current section (specified by tail_leaves macro), and
- The bit T of the next section is toggled (specified by tail_enters_next macro).

([=] l : **Linear** •
[tail_movement_linear_up]
up(l) $\wedge$ $\neg$is_boundary_sec_up(**up**(l)) $\wedge$ $\neg$is_boundary_sec_down(l) $\wedge$
occupied_with_only_tail(l.D2U)
$\longrightarrow$ tail_leaves(l.D2U,l.D2U$'$) $\wedge$ tail_enters_next(**up**(l), l))

([=] l : **Linear** •
[tail_movement_linear_down]
down(l) $\wedge$ $\neg$is_boundary_sec_down(**down**(l)) $\wedge$ $\neg$is_boundary_sec_up(l) $\wedge$
occupied_with_only_tail(l.U2D)
$\longrightarrow$ tail_leaves(l.U2D,l.U2D$'$) $\wedge$ tail_enters_next(**down**(l),l))

As with head movements, if *e* is a point, then tail movements on it will depend on *e*'s physical position. If *e* is a point, then depending on the point's position, the corresponding next section will be updated, analogously to the head movements.

([=] p : **Point** •
[tail_movement_point_stem_to_plus]
occupied_with_only_tail(p.S2PM) $\wedge$ p.POS = PLUS
$\longrightarrow$ tail_leaves(p.S2PM,p.S2PM$'$) $\wedge$ tail_enters_next(**plus**(p),p))

([=] p : **Point** •
[tail_movement_point_stem_to_minus]
occupied_with_only_tail(p.S2PM) $\wedge$ p.POS = MINUS
$\longrightarrow$ tail_leaves(p.S2PM,p.S2PM$'$) $\wedge$ tail_enters_next(**minus**(p),p))

$([=]\ \mathrm{p} : \textbf{Point} \bullet$
$[\mathrm{tail\_movement\_point\_plus\_to\_stem}]$
$\mathrm{occupied\_with\_only\_tail(p.P2S)} \wedge \mathrm{p.POS} = \mathrm{PLUS}$
$\longrightarrow \mathrm{tail\_leaves(p.P2S,p.P2S')} \wedge \mathrm{tail\_enters\_next}(\textbf{stem}(\mathrm{p}),\mathrm{p}))$

$([=]\ \mathrm{p} : \textbf{Point} \bullet$
$[\mathrm{tail\_movement\_point\_minus\_to\_stem}]$
$\mathrm{occupied\_with\_only\_tail(p.M2S)} \wedge \mathrm{p.POS} = \mathrm{MINUS}$
$\longrightarrow \mathrm{tail\_leaves(p.M2S,p.M2S)} \wedge \mathrm{tail\_enters\_next}(\textbf{stem}(\mathrm{p}),\mathrm{p}))$

Change Direction Movements. Change direction movements allow trains to change their travel direction to the opposite. As specified in [ERT14, pages 5.12-13], a change direction movement has to follow a strict procedure that requires the train to have reached the end of its movement authority and be at a stand-still position. For simplicity, it is assumed that a train is only allowed to change its travel direction on a linear section which has signals intended for both up and down directions as shown in Figure 6.10. A train can change its direction on a section e when the signal the train is facing has the actual aspect of CLOSED, and the whole train is inside e. These conditions ensure that no further movements can be made forward by the train, i.e., the train is in a stand-still position. The effect of the movement is straight-forward: the values of the occupancy status variables $e.D2U$ and $e.U2D$ are swapped. Note that we do not check the setting of $s1$ in the condition, however the existence of $s1$ is essential, since the interlocking controller should use it to prevent the unauthorized movements of the train after it has changed the direction.

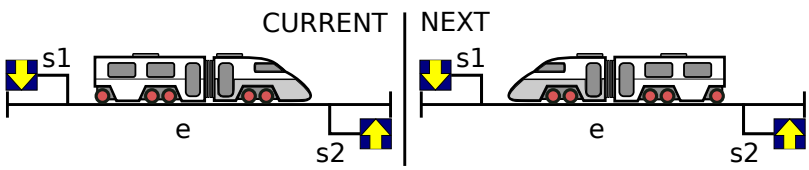


Figure 6.10: Change direction from up to down. It is analogous from down to up.

Guard

- e must have signals intended for both up and down directions.
- The whole train has to be within the section.
- The signal in the current travel direction is closed.

Update

- Swap the values of the `HTO` variables between two directions. This is specified by swap_up_down_vars macro.

([=] l : **Linear** •
[change_direction_up_to_down]
down_sig(l) ∧ **up_sig**(l) ∧ l.D2U = 0b111 ∧ **up_sig**(l).ACT = CLOSED
⟶ swap_up_down_vars(l))

([=] l : **Linear** •
[change_direction_down_to_up]
down_sig(l) ∧ **up_sig**(l) ∧ l.U2D = 0b111 ∧ **down_sig**(l).ACT = CLOSED
⟶ swap_up_down_vars(l))

Entering Interlocked Area Movements. A train enters an interlocked area by two steps: (a) first the head of the train enters the interlocked area, then (b) the tail enters the interlocked area as shown in Figure 6.11 and Figure 6.12, respectively. The details about these movements are explained in the following.

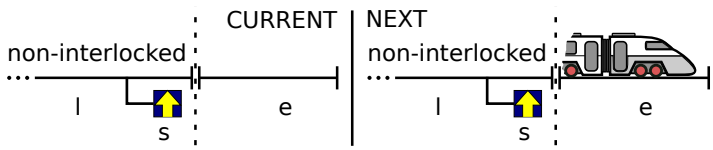


Figure 6.11: The head of a train enters interlocked area.

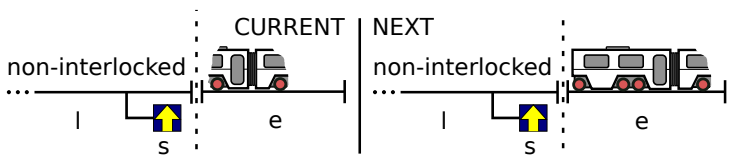


Figure 6.12: The tail of a train enters interlocked area.

(a) The head of a train can enter the section e at the boundary of the interlocked area when the actual aspect of signal s is OPEN, as shown in Figure 6.11. Consequently, the head of the train is simply "put" on e – i.e., the H and O bits of e's occupancy status variable are toggled. The following proposition expresses the transition for the example shown in Figure 6.11.

Guard

- The signal s, which controls the entry movement into interlocked area, is open

Update

- Toggle the H and O bits of the occupancy status variable of *e* by making an xor with $0b101 = 5$. In other words, we "put" the head of the train on the section. This is specified by head_enters macro.

```
([=] l : Linear •
 [enter_interlocked_area_head_from_down]
 is_boundary_sec_down(l) ∧ up_sig(l).ACT = OPEN
 ⟶
 let e = up(l) in
   head_enters(e.D2U,e.D2U′)
 end)
```

```
([=] l : Linear •
 [enter_interlocked_area_head_from_up]
 is_boundary_sec_up(l) ∧ down_sig(l).ACT = OPEN
 ⟶
 let e = down(l) in
   head_enters(e.U2D,e.U2D′)
 end)
```

(b) The tail of the train can enter the boundary section *e* after the head has entered the section. The condition is that *e* is occupied by a train without its tail in the section, as shown in the left of Figure 6.12. If this condition holds, then the tail of the train will be "put" into the network – i.e., the T bit of *e* occupancy status variable is toggled – as illustrated in the right of Figure 6.12. The following proposition expresses the transition for the example in Figure 6.12.

Guard

- The section *e* is occupied without a tail (specified by occupied_without_tail macro) in the direction coming from the non-interlocked area.

Update

- Toggle the T bit of the occupancy status variable of *e* by making an xor with $0b010 = 2$. This is specified by tail_enters macro.

```
([=] l : Linear •
 [enter_interlocked_area_tail_from_down]
 is_boundary_sec_down(l) ∧
 let e = up(l) in
   occupied_without_tail(e.D2U)
 end
 ⟶
 let e = up(l) in
   tail_enters (e.D2U,e.D2U′)
 end)
```

```
([=] l : Linear •
 [enter_interlocked_area_tail_from_up]
 is_boundary_sec_up(l) ∧
 let e = down(l) in
   occupied_without_tail(e.U2D)
 end
 ⟶
 let e = down(l) in
   tail_enters (e.U2D,e.U2D′)
 end)
```

Leaving Interlocked Area Movements. Similar to entering interlocked area movements, a train leaves an interlocked area in two steps: (a) first the head of the train leaves the interlocked area, then (b) the tail leaves the interlocked area as illustrated by Figure 6.13 and Figure 6.14, respectively. These movements are further elaborated below.

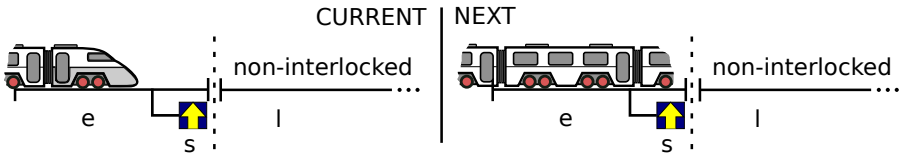


Figure 6.13: The head of a train leaves the interlocked area.

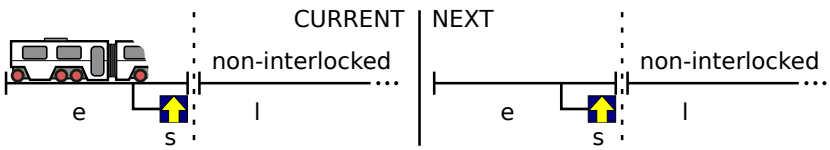


Figure 6.14: The tail of a train leaves the interlocked area.

(a) The train can leave the interlocked area at a boundary section e if the train occupies e in the direction toward the non-interlocked area and the head of the train is inside e. As the effect, the head of the train is removed from e – i.e., the H bit of e's occupancy status variable is toggled. The following proposition specifies the transition for the movement shown in Figure 6.13.

Guard

- The section e is occupied with the head inside the section. This is specified by occupied_with_head macro.

Update

- Toggle the H bit of e's occupancy status variable (specified by head_leaves macro).

```
([=] l : Linear •
 [leave_interlocked_area_head_to_down]
 is_boundary_sec_down(l) ∧
 let e = up(l) in
   occupied_with_head(e.U2D)
 end
 ⟶
 let e = up(l) in
   head_leaves(e.U2D,e.U2D′)
 end)
```

```
([=] l : Linear •
 [leave_interlocked_area_head_to_up]
 is_boundary_sec_up(l) ∧
 let e = down(l) in
   occupied_with_head(e.D2U)
 end
 ⟶
 let e = down(l) in
   head_leaves(e.D2U,e.D2U′)
 end)
```

(b) The tail of a train can leave the interlocked area at a boundary section e if e is occupied in the direction toward the non-interlocked area, the tail of the train is in the section, and the head of the train is not in e. The train is removed from the interlocked area – i.e., e's occupancy status variable is reset to 0 – as the effect of the transition. The movement in Figure 6.14 is modelled by the following proposition.

Guard

- The section e is occupied with only the tail inside the section. This is specified by occupied_with_only_tail macro.

Update

- Reset e's occupancy status variable to 0. This is specified by tail_leaves macro.

```
([=] l : Linear •
 [leave_interlocked_area_tail_to_down]
 is_boundary_sec_down(l) ∧
 let e = up(l) in
   occupied_with_only_tail(e.U2D)
 end
```

```
⟶
let e = up(l) in
  tail_leaves (e.U2D,e.U2D′)
end)

([=] l : Linear •
 [leave_interlocked_area_tail_to_up]
 is_boundary_sec_up(l) ∧
 let e = down(l) in
   occupied_with_only_tail(e.D2U)
 end
⟶
 let e = down(l) in
   tail_leaves (e.D2U,e.D2U′)
 end)
```

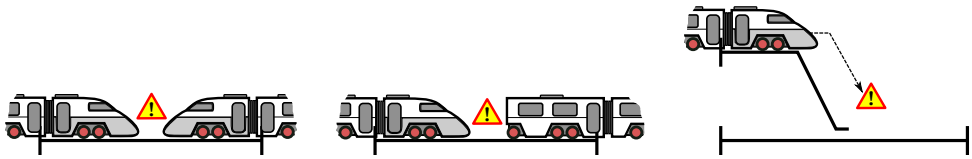
Note that in Figure 6.13, the signal s is not controlled by the considered interlocking system, but the neighboring one. Therefore, in our model, we do not check the setting of s for the movement shown in Figure 6.13. Intuitively, this is an over-approximation because trains would move more freely than they are supposed to. This over-approximation does not jeopardize the soundness of the safety proof of the considered interlocking. Moreover, if we want to verify the safety of the combined model of two interlockings, the train movement model can easily be restricted to allow trains to pass s only when s is OPEN. Thus, the safety of the combined model can be inferred from the safety of the two element interlocking models. A formal proof for such composition may be made, however it is out of the scope of this work.

6.3 Generic High-level Safety Properties

Interlocking systems must at least guarantee the high-level safety properties of no collisions and no derailments. These properties can be expressed as invariants over the occupancy status variables of linear and point sections in the given network. Basically, an interlocking system is safe if no hazardous situations occur on any linear or point sections at any time. The invariants ruling out hazards of different types on a section shown in Figure 6.15 are explained in the following paragraphs. Note that all occupancy status variables are non-negative as described in Section 6.2.1. As a consequence, all components in the formulas in the following paragraphs are non-negative.

Head-to-head collision on a section. A head-to-head collision occurs on a linear section l, when two trains running in opposite directions meet in l. The following invariant rules out such cases.

```
[no_head_to_head_collisions_linear]
(∀l : Linear • (¬is_boundary_sec(l)) ⇒ (l.D2U * l.U2D = 0))
```



(a) Head-to-head collision (b) Head-to-tail collision (c) Derailment on a point

Figure 6.15: Hazardous situations on a section

Since both $l.D2U$ and $l.U2D$ are non-negative, the above formula ensures that l is never occupied in both direction up and down (i.e., when $l.D2U > 0$ and $l.U2D > 0$).

A head-to-head collision occurs on a point p, when at least two trains running in two different directions – in the three possible travel directions on p: stem-to-plus/minus, plus-to-stem, and minus-to-stem – meet in p. Such cases can be ruled out by the following invariant.

[no_head_to_head_collisions_point]
(∀p : **Point** • p.M2S * p.S2PM + p.P2S * p.S2PM + p.P2S * p.M2S = 0)

Head-to-tail collision on a section. A head-to-tail collision occurs on a section e when a train T_2 enters e while it is already occupied by another train T_1 travelling in the same direction. Although two trains may never collide if e is long enough to accommodate both of them, or if T_1 travels at higher speed than T_2 does, these cases are still considered as collisions. These situations are reflected by the values of the occupancy status variables of e. For example, when e is occupied by a train T_1 in the direction from down to up, $l.D2U$ will have its O bit set. If another train T_2 enters e in the same direction, the H and O bits of $l.D2U$ will be toggled according to the train movement model described in Section 6.2.7, resulting in $l.D2U$ having O bit unset, while at least one of its H or T bits are set. Therefore, a head-to-tail collision on e in the direction up is detected by a violation of one of the following formulas where & is *bit-wise and* operator.

[no_head_to_tail_collisions_linear]
(∀l : **Linear** •
(¬is_boundary_sec(l)) ⇒
(l.D2U * (1 − (l.D2U & 1)) + l.U2D * (1 − (l.U2D & 1)) = 0))

[no_head_to_tail_collisions_point]
(∀p : **Point** •
p.S2PM * (1 − (p.S2PM & 1)) + p.P2S * (1 − (p.P2S & 1)) +
p.M2S * (1 − (p.M2S & 1)) = 0)

Derailment on a point. A derailment occurs when a train traverses a point p which is not locked in the correct position for the travel direction of the train. Such cases are ruled out by the following invariant in IDL where $\gg$ is arithmetic shift right operator.

[no_derailments]
(∀p : **Point** •
p.POS * p.P2S + (1 − (p.POS & 1)) * p.M2S + (p.POS ≫ 1) * p.S2PM = 0)

The above invariant rules out the following cases: (a) a train is entering a point from its plus end ($p.P2S > 0$) while the point is in not in the plus position ($p.POS > 0$); (b) a train is entering a point from its minus end ($p.M2S > 0$) while the point is not in the minus position ($1 - (p.POS \ \& \ 1) > 0$); and (c) a train is entering a point from its stem end ($p.S2PM > 0$) while the point is in the intermediate position ($(p.POS \gg 1) > 0$).

6.4 Verification Strategy

This section explains how the third step in the 4-step V&V approach used in our method, step **VV-3**, is performed on the case studies for the forthcoming Danish interlocking systems. A verification strategy which is a combination of BMC and inductive reasoning is employed. The similar strategy has successfully been used in [HPK11].

For a given interlocking system, the generic behavioural model described in Section 6.2 and generic safety properties described in Section 6.3 are instantiated with the concrete configuration data in step **DK:5** and **DK:6** of the V&V flow described in Section 3.7, respectively. Following the semantics of ICL and IDL described in Section 5.2, this instantiation results in a concrete behavioural model of the given interlocking systems in the form of a Kripke structure K, and concrete safety properties ϕ in the form of an invariant in K. To prove that the safety property ϕ is indeed an invariant in K, in step **DK:7**, we employ the k-induction scheme described in Section 2.11.2 in our verification strategy. As mentioned in Section 2.11.2, it is necessary to strengthen the safety property ϕ with another invariant ψ in order to eliminate spurious counter-examples. Then instead of proving that ϕ is an invariant in K, we prove that $\phi \wedge \psi$ is an invariant in K. ψ is a conjunction of a number of propositions restricting the starting state of the induction step. The list of all strengthening invariants and their associated propositions can be found in Appendix E. An example of such propositions is given in the following.

Train Integrity. Some states are not feasible and are not reachable in the model of a given interlocking system because the combinations of the values of the occupancy status variables of sections (see Section 6.2.1) in such states reflect situations that are not physically possible. Examples of infeasible states are shown in Figure 6.16. A section e is occupied by a train $T1$, and the head of the train is not in e; while the next section e' of e in the travel direction is occupied by another train $T2$ as shown in

Figure 6.16a, or e' is vacant as shown in Figure 6.16b. These situations are physically impossible, because they imply that the train $T1$ does not have a head.

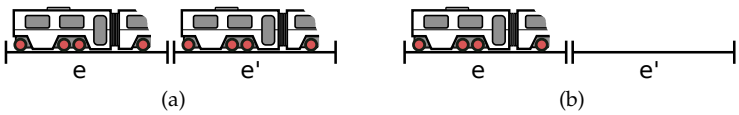


Figure 6.16: Examples of infeasible states ×

In order to eliminate such states, we strengthen the safety properties with *train integrity* invariants. As illustrated intuitively in Figure 6.17, train integrity invariants, as the name suggests, ensure the integrity of all the trains (modelled implicitly by the occupancy status variables of sections) in the model, i.e., every train is a unified object with a head and a tail, except for trains that are entering or leaving the considered network. The idea is simple: *starting from a section e that is occupied by the head (tail) of a train, if we search backward (forward) – w.r.t. the train's travel direction – we should find a unified train without gap*. In other words, we should eventually find the tail (head) of the same train somewhere in one of the previous (next) sections of e – w.r.t. to the train's direction – or the boundary of the interlocked area, before we find the head (tail) of another train, or a vacant section.

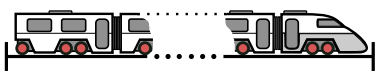


Figure 6.17: Train integrity invariant illustration ✓

The train integrity invariants can be formalized as a conjunction of formulas over the track occupancy variables. For each possible travel direction on a non-boundary section e, there is a formula expressing the constraints between the occupancy status variable of e and the occupancy status variable of the next section e' in the same travel direction. The pattern of such a formula depends on which type of section (linear or point) the current section e and the next section e' are. For instance, for travel direction *up* and a linear section e that has another linear section e' as the neighbor in travel direction *up* – i.e., $e' = up(e)$, the formula will take the following form:

$$(e.D2U \ \& \ 0b101) = 0b001 \Leftrightarrow (e'.D2U \ \& \ 0b011) = 0b001 \tag{6.1}$$

This formula expresses that section e is occupied by a train in direction *up* (the O bit of $e.D2U$ is 1) without the head of the train being on the section (the H bit of $e.D2U$ is 0), iff section e' is also occupied by a train in direction *up* (the O bit of $e'.D2U$ is 1) without the tail of the train being on the section (the T bit of $s'.D2U$ is 0). It can easily be seen that Equation 6.1 rules out the situations shown in Figure 6.16. Equation 6.1 shows the expressiveness of our state encodings allowing properties to be efficiently and compactly formulated.

6.5 Experiments with Our Toolchain

The verification strategy presented in Section 6.4 has been implemented in the prototype toolchain described in Section 3.8. We have used the toolchain to verify successfully the safety properties for model instances (cases) of a number of railway networks as shown in Table 6.18. The cases in Table 6.18 are listed in the approximate order of increasing complexity. The first seven cases are made-up networks inspired by the typical examples used in other studies about formal verification of railway interlocking systems [HBK10; Win12; Jam+14; HPP14]. The network layouts of these cases are shown in Appendix C. For example, *Tiny* is the most simple case with just a single straight track and two routes, while *Lyngby* is inspired by the layout of Lyngby station in Denmark, with three tracks and 24 routes. The last three cases are real networks. Gadstrup-Havdrup (Gt-Hd) and Køge are extracted from the Early Deployment Line (EDL) in the Danish Signalling Programme. The EDL is the first regional line in Denmark to be commissioned in the Danish Signalling Programme. The line spreads over 55 kilometers from Roskilde station to Næstved station. There are in total eight stations in the EDL ranging from simple stations similar to the one shown in Figure 2.2 (named *Mini* in Table 6.18), to complex stations such as Køge.

Table 6.18: Verification results for different networks using simple induction ($k = 1$). Time is measured in seconds, memory usage is measured in MB.

	Linears	Points	Signals	Routes	$log_{10}(\|S\|)$	Time	Memory
Tiny	3	0	4	2	14	2	27
Toy	6	1	6	4	31	3	104
Twist	8	2	8	8	50	9	206
Fork	9	2	8	6	48	9	202
Cross	8	2	8	10	53	15	232
Mini	6	2	8	12	52	19	231
Lyngby	11	6	14	24	108	373	1143
Gt-hd	21	5	24	33	152	399	1865
Køge	57	23	60	73	419	7928	12881
EDL	110	39	126	179	863	38934	40150

In our first trials of verifying the models, we used simple induction (k-induction with $k = 1$) with only safety properties. As discussed in Section 2.11, we got spurious counter-examples because the safety properties are not strong enough to be inductive. In order to remedy the issues, we tried two different approaches: (1) increasing k, and (2) strengthening the invariant to be verified (in this case, the safety properties). It turned out that the verification time increased significantly as k increased in the former approach, making it impossible to verify even the small networks. In the latter approach, we were able to derive strengthening properties ψ (see Section 2.11) for which the verification could be done just using simple induction. According to De Moura et al. [MRS03], any k-induction proof can be reduced to a

simple induction proof with invariant strengthening. Note, however, that in some applications, k-induction has been shown to be advantageous, see, for example, [HPK11]. Table 6.18 shows the results of the final verification using the invariant strengthening approach. Each row of the table lists the size of a network in terms of the number of linear sections, points, signals, and routes in the configuration, and the approximate number of possible states ($|S|$) in the corresponding model instance. For brevity, the approximate number of states are represented in their common logarithm values ($log_{10}(|S|)$); for example, *tiny* case has approximately 10^{14} possible states. The two last columns show the approximate accumulated verification time (in seconds) and memory usage (in MB). All experiments have been performed on a machine with Intel(R) Xeon(R) CPU E5-2667 0 @ 2.90GHz, 64GB RAM, CentOS 6.6, Linux 2.6.32-504.8.1.el6.x86_64 kernel.

We also injected errors into models. Counter examples for these were normally found in relatively short time. This appears to be a general trend when dealing with interlocking systems [JR11]. In a few cases, it took long time to find counter examples. Such examples usually represent very subtle errors in the model or the configuration data, which may be easily overlooked by inspection.

6.6 Comparison with other Techniques

In order to study how other invariant checking techniques, such as Binary Decision Diagram (BDD)-based [Bur+92], CounterExample-Guided Abstraction Refinement (CEGAR) [Cla+03], Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3), or Cone of Influence (COI) techniques perform on our models compared to our toolchain, we translated our model into the input language of nuXmv.

nuXmv is "a new symbolic model checker for the analysis of synchronous finite-state and infinite-state systems" [Cav+14]. It extends one of the most popular open-source model checkers, NuSMV2 [Cim+02], with support for various state-of-the-art abstraction techniques. The subsequent paragraphs report the translation, techniques, configurations, settings, results, and discussion of our benchmarking experiments.

Translation. In order to ensure a faithful translation as the ground for our comparison, the following principles are used for translating the behavioural model and safety properties (generated in step **DK:5** and step **DK:6**, respectively, of the flow described in Section 3.7) to the input language [Boz+14] of nuXmv.

- Variables representing the state space as described in Section 6.2.1 are translated to their corresponding primitive types in nuXmv, i.e., they are translated to `unsigned word` [Boz+14].
- The transition relation of the Kripke structure representing behavioural model is translated to a `TRANS` constraint [Boz+14] in nuXmv.

- Safety properties and strengthening invariants are translated to nuXmv's invariant specifications `INVARSPEC` [Boz+14].

Techniques. We tried to verify the safety properties in the nuXmv translation of the concrete models using various invariant checking techniques [Boz+14] supported by nuXmv as listed in the following. Further detail about these techniques can be found in nuXmv user manual [Boz+14].

(1) `check_invar`: BDD-based invariant checking using reachability analysis

(2) `check_invar_bmc_inc`: induction using a SAT solver, incremental algorithm

(3) `check_invar_cegar_predabs`: performs CEGAR [Cla+03] loop

(4) `check_invar_ic3`: checking invariant using IC3 [Bra11] engine

(5) `check_invar_inc_coi_bdd`: BDD-based incremental COI [Bie+99b] invariant checking

(6) `msat_check_invar_inc_coi`: SMT-based incremental COI invariant checking

Configurations. Each technique is run on three different configurations of the properties to be verified denoted by the suffixes appended to the name of the technique as described in the following.

(a) `all`: the technique is used to verify the conjunction of all properties including both safety properties described in Section 6.3 and strengthening invariants described in Section 6.4.

(b) `safety`: the technique is used to verify the conjunction of all safety properties (strengthening invariants are not included).

(c) `safety_individually`: the technique is used to verify each safety property individually (strengthening invariants are not included).

For example, `check_invar_all` means that `check_invar` is used to verify the conjunction of all properties, including both safety properties and strengthening invariants.

These configurations allow us to study whether other techniques can verify the safety properties efficiently without strengthening invariants, which are required for a successful verification in our toolchain. Note that some techniques work only with a subset of the above configurations. In such cases, only results of working configurations are reported.

Settings. All experiments are run with NuXmv 1.0.0 on the same machine specified in Section 6.5, where the evaluation with our toolchain in RT-Tester has been performed. Each technique is run on networks with increasing complexity as listed in Table 6.18. Running time and memory usage are profiled during the experiments. *Running time threshold* is 24 hours, and *memory usage threshold* is 54GB. These thresholds are chosen based on the following factors: the computation capacity of the machine where the experiments were run, the verification results of our toolchain as shown in Table 6.18, and the limit amount of time we had for experiments (experiments should not be run forever). If an experiment exceeds at least one of these two thresholds and produces no conclusion so far, it will be *terminated*. Once an experiment is terminated, no further experiments will be run for the same technique because the technique would simply reach the thresholds again for more complex networks.

Results. Table 6.19 shows the running time and memory usage collected from the benchmarking experiments where `rt_tester` denotes our toolchain implemented in RT-Tester. For each technique, the running time (in seconds) of *successful* experiments are listed in the first row (with white background and underlined), while the memory usage (in MB) of successful experiments are listed in the second row (with green background). Experiments that were *terminated* are marked in *yellow* or *red* background denoting that they have exceeded the running time threshold or the memory threshold, respectively, without producing any conclusion. The value "–" denotes the experiments that were not run because the experiments with the same technique on smaller cases have been terminated due to too long running time or memory exhaustion. Few techniques, e.g., `check_invar_bmc_inc`, have running time and memory usage of zero for `tiny` case because the experiments ran too quick for the profiler to measure.

Discussion. As seen in Table 6.19, among all of the techniques used in the benchmarking experiments, only our toolchain (`rt_tester`) was able to deliver a conclusion for the largest case, the EDL, within the running time and memory limits. All other techniques exceeded either the running time threshold or the memory threshold without delivering a conclusion for the EDL case, hence they were terminated.

As discussed in Chapter 2, checking techniques based on complete model representations also failed for the NuXmv tool. The BDD-based invariant checking technique – `check_invar` – did not perform well on our models. It exceeded the running time threshold for small cases on all of the three configurations.

BDD-based incremental COI technique – `check_invar_inc_coi_bdd` – did not perform better compared to its BDD-based peer – `check_invar`. It exceeded the running time threshold for small cases on all three configurations.

CEGAR technique – `check_invar_cegar_predabs` – did not perform well on our models either. When run with configuration `all`, it exceeded the memory threshold while refining the abstraction, even for the smallest case. When run with

Table 6.19: Benchmarking results – verification time and memory usage

n	verification succeeded, took *n* seconds	n	running time threshold exceeded,		memory threshold exceeded,	seg.	segmentation
m	and used *m* MB of memory		terminated, no conclusion	m	terminated, no conclusion	fault	fault
-	canceled because its previous one has been terminated						

	Tiny	Toy	Twist	Fork	Cross	Mini	Lyngby	Gt-hd	Køge	EDL
`rt_tester_all`	2	3	9	9	15	19	373	399	7928	38934
	27	104	206	202	232	231	1143	1865	12881	40150
(1a) `check_invar_all`	1	6685	86401	-	-	-	-	-	-	-
	48	724		-	-	-	-	-	-	-
(1b) `check_invar_safety`	2	5664	86401	-	-	-	-	-	-	-
	48	694		-	-	-	-	-	-	-
(1c) `check_invar_safety_individually`	1	3657	86401	-	-	-	-	-	-	-
	48	691		-	-	-	-	-	-	-
(2a) `check_invar_bmc_inc_all`	1	1	3	2	3	3	18	22	230	
	0	73	165	141	190	197	999	2168	31280	55304
(3a) `check_invar_cegar_predabs_all`		-	-	-	-	-	-	-	-	-
	55297	-	-	-	-	-	-	-	-	-
(3b) `check_invar_cegar_predabs_safety`	seg. fault	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	-	-	-
(3c) `check_invar_cegar_predabs_safety_individually`	seg. fault	-	-	-	-	-	-	-	-	-
		-	-	-	-	-	-	-	-	-
(4a) `check_invar_ic3_all`	1	3	8	7	12	15	1456	2143		-
	60	142	332	293	409	424	2433	4554	55313	-
(4b) `check_invar_ic3_safety`	2	48	621	590	1111	86401	-	-	-	-
	60	139	317	279	363		-	-	-	-
(4c) `check_invar_ic3_safety_individually`	2	108	2613	1621	86401	-	-	-	-	-
	59	136	381	307		-	-	-	-	-
(5a) `check_invar_inc_coi_bdd_all`	1	6159	86401	-	-	-	-	-	-	-
	47	728		-	-	-	-	-	-	-
(5b) `check_invar_inc_coi_bdd_safety`	1	23947	86401	-	-	-	-	-	-	-
	59	699		-	-	-	-	-	-	-
(5c) `check_invar_inc_coi_bdd_safety_individually`	1	9741	86401	-	-	-	-	-	-	-
	48	1067		-	-	-	-	-	-	-
(6a) `msat_check_invar_inc_coi_all`	1	1	4	4	7	7	110	181	41736	
	0	82	176	155	208	214	1135	2381	37792	55331

configuration `safety`, or configuration `safety_individually`, it gave a segmentation fault error as shown in Table 6.19.

IC3 technique – `check_invar_ic3` – was not able to handle the large cases: it exceeded the memory threshold when run on Køge with configuration `all`, and exceeded the running time threshold on even smaller cases when run with configuration `safety` or configuration `safety_individually`.

Induction using an SMT solver – `msat_check_invar_inc_coi` – performed better than our toolchain for small cases when running on configuration `all`. For large cases, e.g., Køge, it used more memory and time than our toolchain. Consequently, it exceeded the memory threshold without delivering a conclusion for the EDL case.

When running induction using an SMT solver – `msat_check_invar_inc_coi` – or induction using a SAT solver in nuXmv – `check_invar_bmc_inc` – on configuration `safety` or configuration `safety_individually`, the verification time increased significantly as a number of unrolled steps k increased, similarly to the case with our toolchain where we tried to increase k instead of strengthening the properties as described in Section 6.5. Consequently, both two techniques exceeded the running time threshold even for the smallest case without delivering an answer. Thus these results are not shown in Table 6.19. It appears to be a general trend that increasing k in the k-induction scheme does not work well for our models.

It is evident that when running on configuration `all`, induction using a SAT solver in nuXmv – `check_invar_bmc_inc` – outperformed our toolchain for cases up to Køge as far as the running time is concerned. On the other hand, it used increasingly more memory compared to our toolchain as the size of cases grows. As a consequence, the technique exceeded the memory threshold without delivering a conclusion for the EDL case. Based on the rate of increasing memory usage of `check_invar_bmc_inc` and our toolchain, we would expect the technique to use – for the EDL case – at least three times the amount of memory it used for verifying Køge case, i.e., at least 93GB. This prediction is based on: (1) our toolchain has slower rate of increasing memory usage for large cases, and (2) for EDL, our toolchain uses approximately three times the amount of memory it used for verifying Køge. Therefore our memory threshold is not the limiting factor preventing the technique from succeeding on the EDL case in our benchmarking experiments. The analogous arguments are applied for `msat_check_invar_inc_coi`.

6.7 Related Work

Railway domain has been identified as a *grand challenge* for computing science and transport engineering [Bjø04]. Bjørner analysed the results and trends in formal techniques and tools for development of software for transportation systems especially railways in [Bjø03]. The author pointed out two major problems for future research: (1) the need of integrating different techniques and tools for better result, (2) and the lack of domain understandings. The author proposed a joint research and development project between the software engineering researchers and transport

engineers to establish a conceptual model for railway system to which any actual railway system would be precisely described.

In recent years, the railway domain has become one of the most promising application domains of formal methods. Several research groups have investigated how formal methods would help producing efficiently more robust railway control systems [Win+06; Win12; BFG05; HPK11; Cao+11; MY09; Jam+14]. An overview of recent trends, challenges, and lesson learned can be found in [Fan14; FFM12; Fer+13; Fan12b], and recommendations and best-practices for efficient development and verification of safe railway control systems are summarized in [HP13a; Fer+13]. Reconfigurable systems and automated, formal verification are among these recommendations that we have followed in our method.

Using formal methods to verify safety properties of interlocking systems is investigated by numerous research groups. Different approaches are used to obtain automatically a formal model of an interlocking system from its specification or implementation, e.g.:

(a) Extract a formal model from the low level implementation of the interlocking controller in Ladder Logic Diagrams (LLD), e.g., see [Bon13; Bon+13; Jam+13; Fer+10; JR11; KMS09];

(b) Derive a formal model from the relay circuit diagrams of a relay-based interlocking system, e.g., see [Hax14; Hax12; HKB11; AT12]; or

(c) Derive a formal model from the given network layout and the corresponding interlocking table for route-based interlocking systems, e.g., see [VHP15; Jam14; JR14; Jam+14; Win+06; Win12].

Model checking is one of the most promising techniques for verifying safety properties of interlocking systems thanks to its capability to be fully automated. Unfortunately, due to the state explosion problem, the technique is only able to verify applications of small size as elaborated in [Fer+10]. Several techniques have been proposed in order to push the applicability bounds toward industrial size. Winter et al. suggest using ordering strategies optimized for interlocking models [Win12; Win+06]. A number of high-level abstractions for reducing the complexity of interlocking models are presented in [Jam+14]. In [Fan12a], Fantechi et al. suggest a distributed interlocking model whose verification can be divided into small tasks and verified in parallel. SAT-based model checking and slicing techniques are used in [JR11]. In order to remedy the problem with state space explosion in the global model checking approach, a strategy that is a combination of BMC and inductive reasoning has successfully used for some other applications in previous studies, e.g., see [HPK11; HPP14]. In the current work, we employed a similar strategy that is a combination of SMT-based BMC with inductive reasoning. This strategy allows us to verify safety properties without having to explore the whole state space, therefore we were able to push the bounds even further to handle larger networks of industrial size. For example, only small, simple networks – smaller than *Lyngby* case shown in

Table 6.18 – can be verified in [Jam+14]. As another example, the largest network can be verified with an optimized variable ordering strategy in [Win12] has 41 routes, 9 points, 19 signals, and 31 sections. This is smaller compared to the Køge case, and much smaller compared to the EDL case in the experiments with our toolchain shown in Table 6.18. Of course this is a rough comparison, a more thorough benchmarking taking into account the differences in national regulation and rules is left for future work.

As an alternative to the model checking approach, theorem proving based techniques have also shown success in the railway domain, e.g., see [HP00], but these provide a lesser degree of automation.

There exist also commercial solutions for formal verification of interlocking systems, e.g., Prover iLock™ offered by Prover® Technology[§§]. However, they are *proprietary*, hence very limited information about the underlying verification method is publicly available.

Although sequential release has been used in some interlocking systems, we have not found any published formal models of interlocking systems that integrate this feature. In [TRN02], the conditions for elements to be unlocked and reused in sequential releases are pre-computed and specified in the interlocking tables. In our approach, sequential release is integrated into the behavioural model rather than into the configuration data. This reduces the complexity of the configuration data and makes interlocking configuration data relatively independent from the chosen interlocking approaches.

[§§] `http://www.prover.com/`

CHAPTER 7

Model-based Testing of Railway Interlocking Systems

This chapter presents a domain-specific strategy for model-based testing of railway interlocking systems used in step **VV-4** in the 4-step verification and validation approach presented in Section 3.6. As discussed in Section 2.9, testing and formal verification are two complementary activities in the software development cycle. Although formal verification on a system has been performed, testing is necessary to ensure that the system implementation behaves correctly on the given hardware.

The domain-specific testing strategy presented in this chapter is an extension of the testing strategy used by Mewes in [Mew10, Sect. 5.2]. The generic requirements identified for interlocking systems in [Mew10] are adopted and reused. On the other hand, a domain-specific approach and an SMT solver are used for test generation instead of implementing test generation directly as in [Mew10]. Furthermore, in addition to the generic requirements identified in [Mew10], generic requirements related to sequential release are also identified in the testing strategy presented here.

The remainder of this chapter is organised as follows. First, Section 7.1 motivates the choice of a domain-specific testing strategy for interlocking systems. Then, the proposed domain-specific testing strategy is presented in Section 7.2. The generic requirements identified for the forthcoming Danish interlocking systems are described in Section 7.3. Section 7.4 and Section 7.5 present some experimental results and related work, respectively.

7.1 Why Domain-specific Testing Strategy?

As discussed in [HP15], Dijkstra's famous statement [BR70] – *"Testing shows the presence, not the absence of bugs"* – needs to be clarified. It is indeed possible to *prove* the absence of bugs by testing, as long as certain hypotheses of the true behaviours can be assumed to be valid. Such hypotheses can be formalised by fault models. A *fault model* is a three-tuple $\mathcal{F} = (\mathcal{S}, \sqsubseteq, \mathcal{D})$ where $\mathcal{S}$ is the *reference model* – i.e., a test model in our case, $\mathcal{D}$ is a *fault domain* [PYB96], and $\sqsubseteq$ is a *conformance relation* specifying whether a model $\mathcal{S}' \in \mathcal{D}$ conforms to the reference model $\mathcal{S}$, denoted by $\mathcal{S}' \sqsubseteq \mathcal{S}$ [PYB96; HP15]. If such a fault model $\mathcal{F}$ can be defined, it is possible to produce a complete test suite with respect to $\mathcal{F}$. A test suite is *complete with respect to a fault model* $\mathcal{F}$ if it satisfies the following two conditions [HP14]:

- *Sound.* If a model $\mathcal{S}' \sqsubseteq \mathcal{S}$ then $\mathcal{S}'$ will pass all the tests in the test suite.
- *Exhaustive.* If a model $\mathcal{S}' \not\sqsubseteq \mathcal{S}$ then $\mathcal{S}'$ will fail at least one test in the test suite.

In other words, a complete test suite will uncover all the faulty implementations, while it does not reject any correct implementations in the given fault model. Even if the SUT lies outside the fault domain, a complete test suite with respect to a given fault domain has been proved to have significantly higher test strength compared to other approaches [HP14; HHP15]. Throughout the rest of this chapter, a complete test suite with respect to a fault domain $\mathcal{F}$ is abbreviated as a complete test suite.

Although the above result is very interesting in theory, the application of this in practice is limited, and often infeasible. As the size of the state space of a system grows, the number of test cases in a complete test suite becomes enormous. Therefore, it may not be feasible to generate and execute the whole complete test suite within a reasonable time frame.

In order to remedy the problem, a smaller test suite, which is a subset of the complete test suite, needs to be selected such that the smaller test suite is as close to being complete as possible. The smaller test suite can be chosen using different plausible heuristics. Such heuristic selection is referred as a *testing strategy*. Here we name a few common strategies:

(a) *Random testing strategy.* In this approach, test cases are created by generating random values as input to the SUT [Ham94]. Consequently, random test cases are chosen from a complete test suite. Although this process can be highly automated, it results in a rather weak test suite as the creation of test cases is not guided by the test model. As a consequence, random testing is often used as a baseline for evaluating the strength of a test suite [HHP15; HP13b].

(b) *Domain-specific strategy.* In a domain-specific approach, a smaller test suite is chosen using domain knowledge and experience. The intuition is that domain engineers would have experience on where, when, and how a system would fail. Such insight would let us decide which test cases are more important than

others. This would at least ensure that the most important parts of a system are exercised.

(c) *Equivalence class testing strategy.* In equivalence class testing, a complete test suite is partitioned into disjoint equivalence classes. An *equivalence class* is a set of test cases for which the system behaves the same way, i.e., the behaviours of the system are not exercised more or less if only one test case in an equivalence class is tested, or all the test cases in the equivalence class are tested. If such a partition can be made, then it is possible to produce much a smaller test suite by selecting one test case from each equivalence class. This smaller test case would be as complete as the original test suite. The formal justification and application of model-based equivalence class testing to ETCS onboard systems can be found in [HP13b; Bra+14b]. Inspiration for applying equivalence class testing to railway interlocking systems is discussed in [HP15].

As elaborated in Section 2.6, railway interlocking systems have product line characteristics: they share common generic applications, while each individual system is obtained by instantiating the generic applications with concrete configuration data. Naturally, requirements for interlocking systems exist at a generic level and can be instantiated for each concrete system with a concrete configuration data. This suggests a generic and domain-specific testing strategy for interlocking systems. This testing strategy is employed in our method, and it falls into category (b) described above. An equivalence class testing strategy for interlocking systems will also be investigated in future work based on the inspiration described in [HP15].

7.2 Domain-specific Testing Strategy

This section describes the proposed domain-specific testing strategy. The basic idea is that generic requirements and test objectives should be identified for a product line, e.g., the Danish interlocking systems. Then for each individual interlocking system, these test objectives are instantiated with concrete configuration data, resulting in concrete test objectives. Eventually, a test suite is generated for the given system from the concrete test objectives and a test model which is a refinement of the verified model generated in step **DK:5** of the V&V flow. This test suite ensures that applicable requirements are exercised for the given system.

The subsequent subsections elaborate the testing strategy in detail. First, Section 7.2.1 describes a simplified version of the test generation and execution process using an SMT solver in RT-Tester – a reference MBT framework [Pel13]. Then Section 7.2.2 shows how the process is extended with our generic, domain-specific testing strategy.

7.2.1 Model-based Testing Using an SMT Solver

Figure 7.1 shows a simplified test generation and execution process in an MBT framework using an SMT solver. The process involves the following steps:

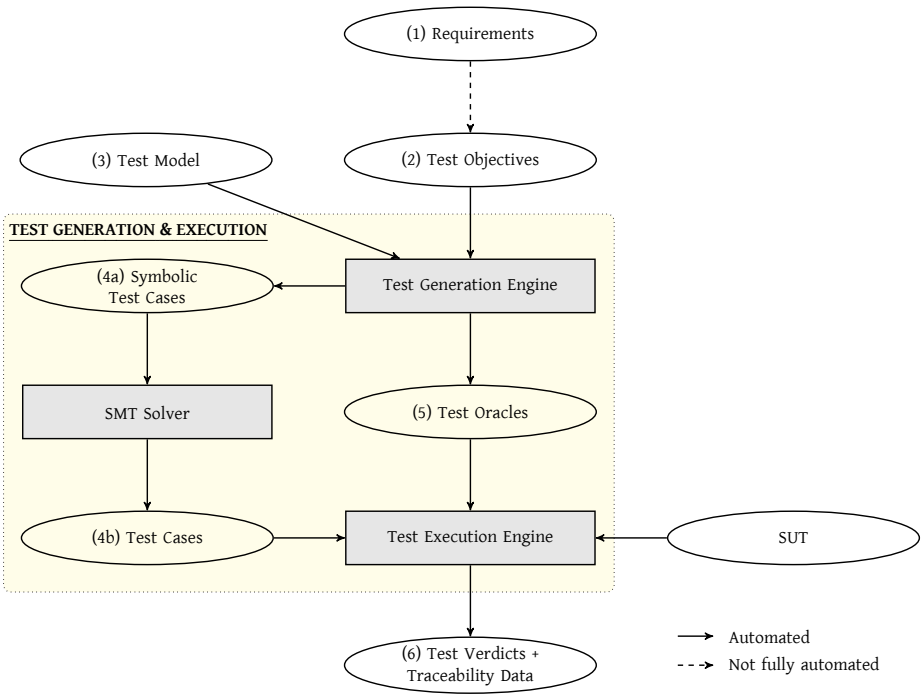


Figure 7.1: Model-based testing using an SMT solver

(1) The process starts with requirements expressed in an appropriate logic. We use LTL for this purpose in the work presented in this dissertation. These requirements may have been identified automatically from the test model or provided manually.

(2) For each requirement, a *test objective* – i.e., an LTL formula characterizing the test cases contributing to the requirement – is derived. Note that there is no general rules for deriving a test objective from an arbitrary requirement. However, rules exist for requirements expressed in certain forms. For example, for a requirement expressed by the LTL formula of the form $\mathbf{G}(\alpha \Rightarrow \beta)$ the test objective is $\mathbf{F}(\alpha)$, as we would like to check that β holds whenever α holds, thus it is not interesting when α does not hold. Note that the concept of test objectives is quite similar to the concept of *trap properties* in test generation with model checkers [FWA09; Bro+05]. The difference here is that *witnesses* of a test objective are used to derive test cases, while *counter-examples* of a trap property are used to derive test cases. This is due to the fact that model checkers are meant to prove properties, while SMT solvers try to find satisfiability assignments for properties.

(3) A test model formalised as an IOSTS is created.

(4) A test case for a given test objective ϕ shall be derived from a finite path $s_0.s_1 \ldots s_k$ in the test model starting from the current model state s_0, and satisfying ϕ. Note that s_0 is not necessary to be an initial state of the test model, it can also be, e.g., the last state visited by the previous test case in the test procedure when we run test cases continuously without resetting the SUT. The calculation of a test case from a test objective and a test model is done as in the following.

   (a) A test generation engine transforms the test objective ϕ into a *symbolic test case* – a proposition characterising paths in the test model starting from s_0 and satisfying ϕ. This results in a symbolic test case of the following form.

$$tc \equiv J(s_0) \wedge \pi(s_0, \ldots, s_k) \wedge g(s_0, \ldots, s_k) \tag{7.1}$$

   where

   - $J(s_0)$ is a proposition characterising the current model state s_0;
   - $\pi(s_0, \ldots, s_k)$ is the proposition characterising a path from s_0 to s_k in the test model as defined in Equation 2.4; and
   - $g(s_0, \ldots, s_k)$ is the translation of ϕ on a path from s_0 to s_k using the bounded semantics of LTL formulas described in [Bie+06; Bie+99a].

   (b) Equation 7.1 is then solved by an SMT solver, the SONOLAR solver [PVL11] in our case. A solution of Equation 7.1 represents a path starting from s_0 in the test model and satisfying ϕ. The states in the path are then applied the restriction operation described in Section 2.11.3 to input variables, resulting in the sequence of inputs for a test case for ϕ.

(5) The test generation engine generates also test oracles from the test model.

(6) Test cases generated in step (4) are executed on a SUT, the results are checked by the test oracle generated in step (5) to produce the test verdicts. The traceability data is also derived automatically during test generation and execution [HP15].

7.2.2 Extended Test Generation and Execution Process

Figure 7.2 shows the test generation and execution process in our method. It is a more detailed version of the test generation and execution shown in Figure 3.3. The test generation and execution in Figure 7.2 extends the process described in Section 7.2.1 with a generic, domain-specific strategy. The generation and execution from step (4) onward is unchanged compared to the process described in Section 7.2.1. The main difference in the extended process is how the test model and test objectives are obtained as explained in the following.

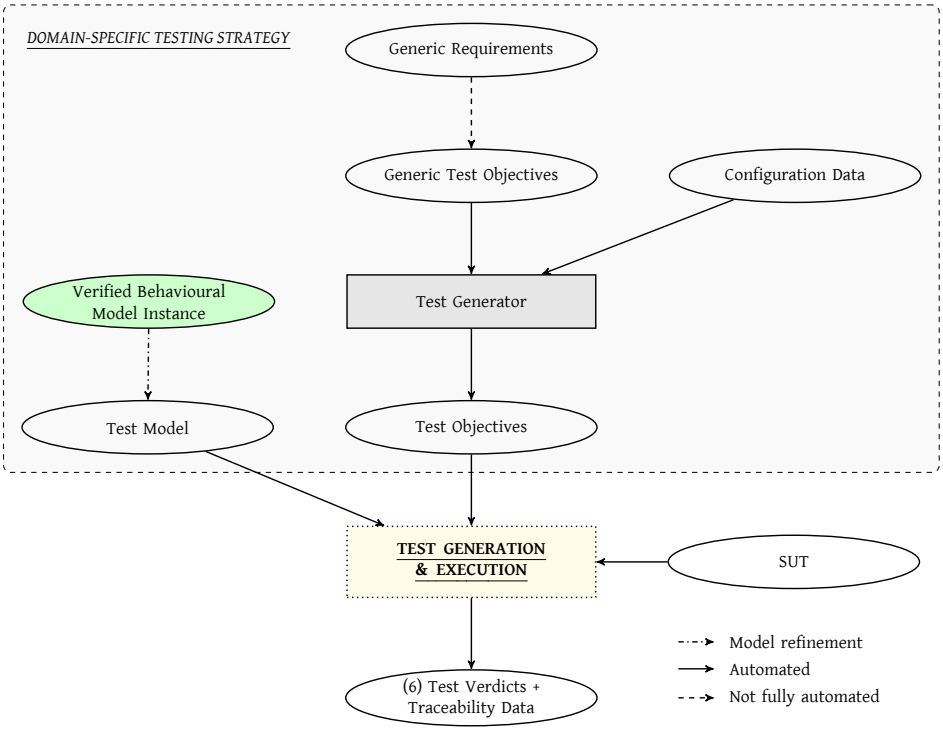


Figure 7.2: Test generation and execution process. The yellow rectangle represents the part of the process framed in the yellow rectangle in Figure 7.1.

Test Objectives. For a given system, test objectives are generated from generic requirements by the following procedure, see Figure 7.2.

(a) For a product line, *generic requirements* are identified. The generic requirements identified for the Danish interlocking systems are presented in Section 7.3. The generic requirements are specified in IDL described in Chapter 5.

(b) For each identified generic requirement, a *generic test objective* characterising test cases contributing to the given generic requirement is derived. This derivation is similar to the step (2) described in Section 7.2.1, except that the requirements and their corresponding test objectives are generic. The generic test objectives are also specified in IDL.

(c) A *test generator* – ingredient **DK:g** in the method described in Chapter 3 – instantiates the generic test objectives with a wellformed configuration data,

resulting in concrete test objectives. This is the step **DK:8** in the V&V flow described in Section 3.7.

Test Model. One of the challenges in MBT is how to obtain a correct test model. As elaborated in Chapter 6, using BMC and inductive reasoning, we can verify the safety properties of a model instance K generated in step **DK:5** of the V&V flow described in Section 3.7. Therefore, it is sensible to use the verified model instance as the test model for test generation process. However, the model K is in the form of a Kripke structure, which is not suitable for test generation yet due to the following reasons.

- K is a closed system evolving according to its transition relation. Thus there are no explicit notions of input and output. As explained in Section 2.11.3, in hardware/software integration testing, we need explicit notions of input and output in the test model in order to specify how we stimulate the SUT and the expected outcome. Therefore, we need to introduce the notion of input and output into our verified model instance.

- K is a non-deterministic safe over-approximation of the behaviours of the system. As the chosen MBT framework has not yet supported test generation and execution using a non-deterministic model, we need to refine K to a deterministic model TS and use TS as the test model. As a consequence, the resulting test suite will only show the conformance of the implementation to TS.

The following rules are used to refine the verified model instance K in a form of a Kripke structure into a deterministic IOSTS TS that is suitable to use as test model.

- *Introduce notions of input and output.* The variables representing state space as described in Chapter 6 are divided into three disjoint sets of input, local, and output variables. This allows us to consider the behavioural model instance as an IOSTS TS. Note that the Kripke structure obtained by extending this IOSTS is the same as K. The division of variables into disjoint sets are merely for distinguishing the SUT – in this case, the interlocking controller – from its surrounding environment for testing purposes.

- *Refine to a deterministic model.* The following rules are used to refine the model instance to a deterministic one. Note that this refinement is necessary at the moment because the support for non-determinism in RT-Tester is currently under development. Once non-determinism is supported, this refinement step is not needed.

  - Transitions in the transition relation described in Chapter 6 that have the same priority are taken in the order that they appear in the specification.
  - Transitions in the transition relation that have different priorities are taken according to their priorities, i.e., transitions with a higher priority are taken first.

– Transitions that are generated from a quantified transition in the transition relation are taken in the alphanumeric order of the values of the quantified variable.

7.3 Generic Requirements for the Danish Interlocking Systems

This section explains the generic requirements for the forthcoming Danish interlocking systems identified in our domain-specific testing strategy. The requirements are specified in IDL – the language for describing generic applications presented in Chapter 5. The syntax is similar to LTL formulas, with some extra constructs for describing generic requirements, see Chapter 5. Most of the requirements have one of the forms shown in Table 7.3 (neglecting the quantification for specifying the generic part) from which the generic test objectives can be derived. Requirements of the third form (the third row in Table 7.3) are safety requirements. They have been proved in Chapter 6 using BMC and inductive reasoning. Note that it is not trivial to obtain a test objective for a requirement of this form. Simple technique would result in test cases containing only an initial state [FWA09]. Other techniques need to be used to derive test cases for requirements of this form, see [Bro+05; FWA09].

Table 7.3: Generic requirements and their corresponding generic test objectives

Requirements	Test Objectives
$\mathbf{G}\ \mathbf{F}\ \phi$	$\mathbf{F}\ \phi$
$\mathbf{G}\ (\phi \Rightarrow \psi)$	$\mathbf{F}\ \phi$
$\mathbf{G}\ \phi$	(*)

(*) safety requirements, proved in Chapter 6

For readability, the generic requirements for the Danish interlocking systems are grouped into three different categories: (1) functional requirements, (2) safety requirements, and (3) sequential release related requirements. These categories of requirements are not necessarily disjoint: a requirement may belong to more than one group. For example, the sequential release related requirement **RR-05** may be also considered as a safety requirement. Therefore, the grouping is mainly for readability. Note that some of the requirements are also derived from functional requirements document for the forthcoming Danish interlocking systems. Due to a confidentiality agreement, these requirements are not included in this thesis. The subsequent subsections explain in detail each category of the generic requirements.

7.3.1 Functional Requirements

The requirements in this category mainly deal with the functionalities of the systems. In other words, these requirements check whether a given system perform what it is supposed to do.

FR-01 Every route *r* specified in the interlocking table can be dispatched.

$([=]\ \mathrm{r}\ :\ \mathbf{Route} \bullet [\mathrm{FR\_route\_marked}]\ \mathbf{G}\ \mathbf{F}\ [\mathrm{r.DSPL} = \mathrm{MARKED}])$

FR-02 Every route *r* specified in the interlocking table can be allocated.

$([=]\ \mathrm{r}\ :\ \mathbf{Route} \bullet [\mathrm{FR\_route\_allocating}]\ \mathbf{G}\ \mathbf{F}\ [\mathrm{r.DSPL} = \mathrm{ALLOCATING}])$

FR-03 Every route *r* specified in the interlocking table can be locked.

$([=]\ \mathrm{r}\ :\ \mathbf{Route} \bullet [\mathrm{FR\_route\_locked}]\ \mathbf{G}\ \mathbf{F}\ [\mathrm{r.DSPL} = \mathrm{LOCKED}])$

FR-04 It shall be possible to use each route *r* specified in the interlocking table, i.e., trains can travel on the specified route.

$([=]\ \mathrm{r}\ :\ \mathbf{Route} \bullet [\mathrm{FR\_route\_in\_use}]\ \mathbf{G}\ \mathbf{F}\ [\mathrm{r.DSPL} = \mathrm{OCCUPIED}])$

FR-05 It shall be possible to cancel each route *r* specified in the interlocking table when *r* is in MARKED mode.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \mathbf{Route} \bullet \\
\ [\mathrm{FR\_cancel\_marked\_route}] \\
\ \mathbf{G}([\mathrm{r.CTRL} = \mathrm{CANCEL} \wedge \mathrm{r.DSPL} = \mathrm{MARKED}] \Rightarrow \mathbf{F}\ [\mathrm{r.DSPL} = \mathrm{FREE}]))
\end{array}
$$

FR-06 It shall be possible to cancel each route *r* specified in the interlocking table when *r* is in ALLOCATING mode.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \mathbf{Route} \bullet \\
\ [\mathrm{FR\_cancel\_allocating\_route}] \\
\ \mathbf{G}([\mathrm{r.CTRL} = \mathrm{CANCEL} \wedge \mathrm{r.DSPL} = \mathrm{ALLOCATING}] \Rightarrow \\
\ \ \mathbf{F}\ [\mathit{/*\ the\ route\ get\ free\ */} \\
\ \ \ (\mathrm{r.DSPL}' = \mathrm{FREE}) \wedge \\
\ \ \ \mathit{/*\ points\ in\ the\ path\ stop\ moving\ */} \\
\ \ \ (\forall \mathrm{p} : \mathbf{Point} \bullet \mathrm{p} \in \mathbf{path}(\mathrm{r}) \Rightarrow (\mathrm{p.CMD} = \mathrm{p.POS})) \wedge \\
\ \ \ \mathit{/*\ all\ sections\ in\ the\ path\ get\ unlocked\ */} \\
\ \ \ (\forall \mathrm{e} : \mathbf{Section} \bullet \mathrm{e} \in \mathbf{path}(\mathrm{r}) \Rightarrow (\mathrm{e.MODE} = \mathrm{AVAIL}))]))
\end{array}
$$

FR-07 It shall be possible to cancel each route *r* specified in the interlocking table when *r* is in LOCKED mode, and no trains have entered *r* yet.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \mathbf{Route} \bullet \\
\ [\mathrm{FR\_cancel\_locked\_route}] \\
\ \mathbf{G}([\mathrm{r.CTRL} = \mathrm{CANCEL} \wedge \mathrm{r.DSPL} = \mathrm{ALLOCATING} \wedge \\
\ \ \ \mathit{/*\ the\ route\ has\ not\ been\ used,\ i.e.,\ all\ the\ route's\ path\ and} \\
\ \ \ \ \mathit{*\ overlap\ are\ still\ vacant\ */} \\
\ \ \ (\forall \mathrm{e} : \mathbf{Section} \bullet \\
\ \ \ \ \mathrm{e} \in (\mathbf{elems\ path}(\mathrm{r}) \cup \mathbf{elems\ overlap}(\mathrm{r})) \Rightarrow \mathrm{vacant}(\mathrm{e}))] \Rightarrow \\
\ \ \mathbf{F}\ [\mathit{/*\ the\ route\ get\ free\ */} \\
\ \ \ (\mathrm{r.DSPL} = \mathrm{FREE}) \wedge
\end{array}
$$

/* *close the source signal* */
(**src**(r).CMD = CLOSED) ∧
/* *all sections in the path get unlocked* */
(∀e : **Section** • e ∈ **path**(r) ⇒ (e.MODE = AVAIL))]))

FR-08 Each pair of a given route *r* and another route *other* that is not in conflict with *r*, can be allocated concurrently.

([=] r : **Route** •
([=] other : **Route** •
[FR_non_conflicting_routes_allocating]
[other ≠ r ∧ other ∉ **conflicts**(r)] ∧
G F [(r.DSPL = ALLOCATING ∧ other.DSPL = ALLOCATING)]))

FR-09 Each pair of a given route *r* and another route *other* that is not in conflict with *r*, can be locked concurrently.

([=] r : **Route** •
([=] other : **Route** •
[FR_non_conflicting_routes_locked]
[other ≠ r ∧ other ∉ **conflicts**(r)] ∧
G F [(r.DSPL = LOCKED ∧ other.DSPL = LOCKED)]))

FR-10 Each pair of a given route *r* and another route *other* that is not in conflict with *r*, can be used concurrently.

([=] r : **Route** •
([=] other : **Route** •
[FR_non_conflicting_routes_used]
[other ≠ r ∧ other ∉ **conflicts**(r)] ∧
G F [(r.DSPL = OCCUPIED ∧ other.DSPL = OCCUPIED)]))

7.3.2 Safety Requirements

These requirements show that different unsafe situations must not occur. Note that all these requirements (except **SR-04**) have been proved in the Chapter 6 using BMC and inductive reasoning. The purpose of testing these requirements is to demonstrate that the implementation conforms to the safety of the test model. Requirements **SR-01** to **SR-03** are high-level safety properties that have been formalised in Section 6.3. Requirements **SR-05** to **SR-12** are low-level safety properties as they are specific to the Danish interlocking systems. These low-level safety properties have also been used as strengthening invariants, see Appendix E, to prove the high-level safety properties using BMC and inductive reasoning as described in Section 6.4.

Requirement **SR-04** does not need to be handled in the context analysed in this thesis, since we are checking the safe control decisions of the interlocking controller. According to the ETCS standard, **SR-04** is delegated to the EVCs –

the standardised onboard computers – in the train. The train speed is closely supervised by a number of standard modules in the EVC. An emergency brake would automatically be triggered to stop the train if it went over the allowed speed. An example of such modules is the *ceiling speed monitoring* module [ERT14, Subset 026, Sect. 3.13]. A *complete* test suite for this ceiling speed monitoring module can be automatically generated using equivalence class testing strategy as elaborated in [Bra+14b; Bra+14a].

SR-01 Head-to-head collisions can never occur, see Section 6.3.

SR-02 Head-to-tail collisions can never occur, see Section 6.3.

SR-03 Derailment due to erroneous point positions can never occur, see Section 6.3.

SR-04 Derailment due to overspeeding can never occur.

SR-05 Whenever a route *r* is allocated, all routes that are in conflict with *r* must not be allocated or locked.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route} \bullet \\
\ [\mathrm{SR\_conflicting\_routes\_allocating}] \\
\ \mathbf{G}\ [(\mathrm{r}.\mathrm{DSPL} = \mathrm{ALLOCATING}) \Rightarrow \\
\ \ (\forall \mathrm{other}\ :\ \textbf{Route} \bullet \\
\ \ \ \mathrm{other} \in \textbf{conflicts}\,(\mathrm{r})\ \Rightarrow \\
\ \ \ \mathrm{other}.\mathrm{DSPL} \neq \mathrm{ALLOCATING} \wedge \mathrm{other}.\mathrm{DSPL} \neq \mathrm{LOCKED})])
\end{array}
$$

SR-06 Whenever a route *r* is locked, all routes that are in conflict with *r* must not be allocated or locked.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route} \bullet \\
\ [\mathrm{SR\_conflicting\_routes\_locked}] \\
\ \mathbf{G}\ [(\mathrm{r}.\mathrm{DSPL} = \mathrm{LOCKED}) \Rightarrow \\
\ \ (\forall \mathrm{other}\ :\ \textbf{Route} \bullet \\
\ \ \ \mathrm{other} \in \textbf{conflicts}\,(\mathrm{r})\ \Rightarrow \\
\ \ \ \mathrm{other}.\mathrm{DSPL} \neq \mathrm{ALLOCATING} \wedge \mathrm{other}.\mathrm{DSPL} \neq \mathrm{LOCKED})])
\end{array}
$$

SR-07 Whenever a route *r* is in ALLOCATING mode, then all of the following must hold.

- All points are commanded to the correct positions as required by *r*.
- All *r*'s protecting signals are commanded to CLOSED.
- All sections in *r*'s path are locked exclusively.
- All sections in *r*'s path and overlap are vacant.

```
([=] r : Route •
 [SR_route_allocating]
 G [(r.DSPL = ALLOCATING) ⇒
  ((∀p : Point • p ∈ points(r) ⇒ (p.CMD = req(r,p))) ∧
   /* protecting signals are commanded in correct aspects */
   (∀s : Signal • s ∈ signals(r) ⇒ (s.CMD = CLOSED)) ∧
   /* all lockable elements in the path are EXLCK(1) */
   (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK)) ∧
   /* all sections have to be vacant */
   (∀e : Section •
    e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e)))])
```

SR-08 Whenever a route *r* is in the LOCKED mode, then all of the following must hold.

- All points used by *r* have their actual positions as required by *r*, and cannot move.
- All *r*'s protecting signals have their actual aspects of CLOSED, and cannot change their aspects.
- All sections in *r*'s path are locked exclusively.
- All sections in *r*'s path and overlap are vacant, except the first section of *r*'s path. The exception of the first section represents the case where a train just enters the route, and the interlocking controller has not reacted to the change yet.
- The source signal of *r* is commanded to an OPEN aspect.

```
([=] r : Route •
 [SR_route_locked]
 G [(r.DSPL = LOCKED) ⇒
  let fst = first(r) in
    /* points are in correct positions */
    (∀p : Point •
     p ∈ points(r) ⇒ (p.POS = req(r,p) ∧ p.POS = p.CMD)) ∧
    /* protecting signals are in correct aspects */
    (∀s : Signal •
     s ∈ signals(r) ⇒ (s.ACT = CLOSED ∧ s.ACT = s.CMD)) ∧
    /* all lockable elements in the path are EXLCK(1) */
    (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK)) ∧
    /* all sections except the first one have to be vacant */
    (∀e : Section •
     (e ≠ fst ∧ e ∈ (elems path(r) ∪ elems overlap(r))) ⇒
     vacant(e)) ∧
    /* first section is vacant or occupied by head of the train */
    (vacant(fst) ∨ hto(fst,r) = 5) ∧
    /* entry signal is commanded to be open */
    src(r).CMD = OPEN
  end])
```

SR-09 Whenever a point *p* is occupied or in use, it must not be commanded to move.

```
([=] p : Point •
 [SR_not_commanding_used_point_to_move]
 G [(¬vacant(p) ∨ p.MODE = USED) ⇒ p.POS = p.CMD])
```

SR-10 Whenever a point *p* is commanded to change its position, the point must not in USED mode and the point is commanded as part of the allocating process of a route.

```
([=] p : Point •
 [SR_point_only_cmd_when_alloc_a_route]
 [(∃r : Route • p ∈ points(r))] ⇒
 G [(p.CMD ≠ p.POS) ⇒
  (p.MODE ≠ USED ∧
   (∃r : Route • p ∈ points(r) ∧ r.DSPL = ALLOCATING) ∧
   (∀r : Route • p ∈ points(r) ⇒ r.DSPL ≠ LOCKED))])
```

SR-11 If a signal *s* is commanded to change its aspect to OPEN, then exactly one route among the routes that have *s* as its source signal has to be in the LOCKED mode.

```
([=] s : Signal •
 [SR_signal_cmd_open_cnd]
 [(∃r : Route • s = src(r))] ⇒
 G [(s.CMD = OPEN) ⇒ (∃!r : Route • src(r) = s ∧ r.DSPL = LOCKED)])
```

SR-12 If the actual aspect of a signal *s* is OPEN, then one of the following must hold.

(1) At least one of the routes that have *s* as the source signal has been cancelled, but *s* has not been closed as commanded yet.

(2) Exactly one route *r* among the routes that have *s* as the source signal such that all sections in *r*'s path, except the first one, are exclusively locked for *r* and vacant, and all sections in *r*'s overlap are vacant. Additionally, one of the following holds:

   (a) *r* is LOCKED mode, and there is no command to change the aspect of *s*, i.e., *s.CMD* is also OPEN, or a train has entered the route, but the interlocking controller has not reacted to that event yet.

   (b) *r* is in OCCUPIED mode, a train has entered the route, and *s* has been commanded to be closed, but the actual aspect has not changed yet.

```
([=] s : Signal •
 [SR_signal_act_open_cnd]
 [(∃r : Route • s = src(r))] ⇒
 /* whenever the signal's actual aspect is OPEN */
 G [(s.ACT = OPEN) ⇒
  /* there exists a route r that has s as its source signal. The route has
   * been cancelled, but the signal has not been closed as commanded yet
```

```
 */
((∃r : Route •
  s = src(r) ∧ r.DSPL = FREE ∧ s.CMD = CLOSED ∧ r.CTRL = CANCEL) ∨
 /* OR there is exactly one route r that has s as the source signal */
 (∃!r : Route •
  (src(r) = s) ∧
  /* r is locked */
  (((r.DSPL = LOCKED ∧
     (s.CMD = OPEN ∨ H__(hto(first(r),r)) ≠ 0)) ∨
    /* a train has entered the route, the interlocking
     * controller has reacted, but the signal has not yet
     */
    (r.DSPL = OCCUPIED ∧ s.CMD = CLOSED ∧
     H__(hto(first(r), r)) ≠ 0)) ∧
   /* all sections in the route's path, except the first one, are
    * exclusively locked and vacant */
   (∀e : Section •
    (e ∈ path(r) ∧ e ≠ first(r)) ⇒
    (vacant(e) ∧ e.MODE = EXLCK)) ∧
   /* all sections in the route's overlap are vacant */
   (∀e : Section • e ∈ overlap(r) ⇒ vacant(e)))))])
```

7.3.3 Sequential Release Related Requirements

The requirements listed in the following concern the sequential release feature. The requirements are roughly divided into two subgroups: functional requirements or safety requirements.

Functional Requirements. The following requirements are more concerning about functionalities of the given system.

RR-01 It shall be possible for two routes that are in conflict to be used at the same time, given that they do not share the last section in their paths, and there exists at least one point that is required to be in two different positions by two routes.

```
([=] r : Route •
 ([=] other : Route •
  [RR_conflicting_route_used_same_time]
  [other ∈ conflicts(r) ∧
   (∃p : Point •
    p ∈ dom points(r) ∩ dom points(other) ∧
    req(r,p) = MINUS ∧ req(other,p) = PLUS ∧
    last(r) ≠ last(other))] ∧
  G F([r.DSPL = OCCUPIED ∧ other.DSPL = OCCUPIED])))
```

RR-02 The sections of a route shall automatically be released sequentially following the passage of a train.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route}\ \bullet \\
\ ([=]\ \mathrm{e}\ :\ \textbf{Section}\ \bullet \\
\quad [\mathrm{RR\_elem\_released\_following\_train\_passage}] \\
\quad [\mathrm{e} \in \textbf{path}(\mathrm{r})\ \wedge \mathrm{e} \neq \textbf{last}(\mathrm{r})]\ \wedge \\
\quad \mathbf{G}([(\mathrm{r}.\mathrm{MODE} = \mathrm{OCCUPIED} \wedge \mathrm{e}.\mathrm{MODE} = \mathrm{USED} \wedge \mathrm{vacant}(\mathrm{e}) \wedge \\
\qquad (\mathrm{e} \neq \textbf{first}(\mathrm{r}) \Rightarrow \mathrm{e}.\mathrm{PREV} = \mathrm{RELEASED}))] \Rightarrow \\
\quad \mathbf{F}\ [\mathrm{e}.\mathrm{MODE} = \mathrm{AVAIL} \wedge \mathrm{r}.\mathrm{MODE} = \mathrm{OCCUPIED} \wedge \textbf{next}(\mathrm{r},\mathrm{e}).\mathrm{MODE} = \mathrm{USED}])))
\end{array}
$$

Safety Requirements. The following requirements are more concerning about safety of the given system.

RR-03 A section of a route is not released sequentially until its previous section in the same route has been released.

$$
\begin{array}{l}
([=]\ \mathrm{r}\ :\ \textbf{Route}\ \bullet \\
\ ([=]\ \mathrm{e}\ :\ \textbf{Section}\ \bullet \\
\quad [\mathrm{RR\_elem\_released\_after\_prev\_has\_been\_released}] \\
\quad [\mathrm{e} \in \textbf{path}(\mathrm{r})\ \wedge \mathrm{e} \neq \textbf{first}(\mathrm{r})]\ \wedge \\
\quad \mathbf{G}([\mathrm{r}.\mathrm{MODE} = \mathrm{OCCUPIED} \wedge \mathrm{e}.\mathrm{MODE} = \mathrm{USED} \wedge \mathrm{e}.\mathrm{PREV} = \mathrm{PENDING}] \Rightarrow \\
\quad \mathbf{X}([\mathrm{e}.\mathrm{MODE} \neq \mathrm{FREE}]\ \mathbf{U}\ [\mathrm{e}.\mathrm{PREV} = \mathrm{RELEASED}]))))
\end{array}
$$

RR-04 Whenever a section *e* is occupied, it should be exclusively locked or used for a route.

$$
\begin{array}{l}
([=]\ \mathrm{e}\ :\ \textbf{Linear}\ \bullet \\
\ [\mathrm{RR\_linear\_occupied\_implies\_exlck\_or\_used}] \\
\ [(\forall \mathrm{r}\ :\ \textbf{Route}\ \bullet\ \mathrm{e} \neq \textbf{last}(\mathrm{r}))\ \wedge \textbf{down}(\mathrm{e}) \wedge \textbf{up}(\mathrm{e})] \Rightarrow \\
\ \mathbf{G}\ [\neg \mathrm{vacant}(\mathrm{e}) \Rightarrow (\mathrm{e}.\mathrm{MODE} = \mathrm{EXLCK} \vee \mathrm{e}.\mathrm{MODE} = \mathrm{USED})])
\end{array}
$$

$$
\begin{array}{l}
([=]\ \mathrm{e}\ :\ \textbf{Point}\ \bullet \\
\ [\mathrm{RR\_point\_occupied\_implies\_exlck\_or\_used}] \\
\ [(\forall \mathrm{r}\ :\ \textbf{Route}\ \bullet\ \mathrm{e} \neq \textbf{last}(\mathrm{r}))] \Rightarrow \\
\ \mathbf{G}\ [\neg \mathrm{vacant}(\mathrm{e}) \Rightarrow (\mathrm{e}.\mathrm{MODE} = \mathrm{EXLCK} \vee \mathrm{e}.\mathrm{MODE} = \mathrm{USED})])
\end{array}
$$

RR-05 If a section *e* is in EXLCK or USED mode, it shall be locked/used by exactly one route. In Chapter 6, this requirement has been proved as strengthening invariants, see Section E.5.

7.4 Experiments

This section presents the implementation of the domain-specific testing strategy described in Section 7.2 in our toolchain in RT-Tester framework [Pel13; Ver15].

Demonstrative SUT Generation. Due to confidentiality issues, we did not have an actual implementation of an interlocking system from our industrial partners. Instead, a demonstrative version of the SUT in C++ is generated from the verified model instance using the following transformation rules:

(1) A transition of the form $g \longrightarrow u$ is transformed into an *if-then* statement `if` g `then` u.

(2) Transitions with the same priority are transformed in the order they appear in the specification.

(3) Transitions with different priority are transformed in the order of their priority, i.e., transitions with the highest priority are transformed first, and transitions with lowest priority are transformed last.

(4) Quantified transitions are transformed in a way such that the resulting concrete transitions appear in the alphanumeric order of the values of the quantified variable.

(5) The results of the transformation are put into a while loop. Once the *then* branch of one of the *if-then* statement is executed, the remaining statements are ignored and a new loop begins.

Obviously, the generated SUT will pass all the generated test cases. In realistic settings, the demonstrative SUT shall be replaced with the real SUT. The demonstrative SUT can also be used for simulation in order to check the quality of the test suite before execute it on in the realistic settings. This would reduce the testing cost, as debugging and fixing bugs in the test suite in the realistic settings are expensive [BF14].

Automated Test Execution and Report. Thanks to the existing MBT framework in RT-Tester, the generated test cases can be executed automatically. The results of test cases are tagged with the name of the transitions where they are observed, and the requirements that they cover. All these results are documented in an automatically generated test report including the traceability data back to the requirements and the test model.

7.5 Related Work

In the railway domain, there is a trend to shift toward using model-based software development and formal methods in their development process. One of the most prominent successes is the application of MBT which has been investigated by many research groups and enterprises, especially railway manufacturers. However, the transition from code-based process to a model-based one encounters numerous challenges as pointed out in [Fer+13] such as limited support of modelling languages

and industrial tools, multiple formalisms, and difficulties in integrating model-based development into the existing processes used by railway system manufacturers. The article also summarised a number of lessons learned in the process of adopting model-based development and formal methods in a multinational railway manufacturer. These lessons can serve as a useful guideline for any railway system manufacturers in transitioning to model-based development.

Ferrari et al. [Fer+11] reports the experience of applying successfully a two-phase V&V process in a railway signalling manufacturer. The two-phase V&V process resulted in the cost reduction of about 70 percent. In the first phase, MBT was used to exercise the functional behaviours of models and code in order to check that the generated code conformed to the models. In the second phase, abstract interpretation was used to detect runtime errors. The MBT phase consists of two steps: (a) a back-to-back testing on both the Simulink model and the generated code to check for equivalence, and (b) an additional evaluation to ensure the absence of errors introduced by the model-to-code translation. The unit requirements are encoded into a Stateflow model. Then the test data is derived using simulation and the assessment of the correctness is done manually instead of automatically due to the low level of abstraction of the model – a necessity to generate code.

Bonacchi et al. [BF14] developed a framework that is able to extract a model from an implementation of an interlocking system expressed in LLD. A pre-planned test suite provided by railway signalling engineers is then used to perform a software-in-the-loop testing on the model in order to find bugs and defects in the planned test suite. This reduces the time spending on debugging while performing hardware-in-the-loop testing on the target.

In [Bon+12] testing in the railway domain is investigated from a different angle where a graphical user interface software is tested. The work focus on testing equipment configuration tools that produce configurations for different safety-critical equipments. These tools have certain impact on the safety of the equipments that load the resulting configurations. The authors proposed a testing strategy tailored for such configuration tools with a convenient trade-off between high coverage of the input domain and the feasibility within a time constraint.

In [Cal+06], the TTCN-3 testing language is applied for testing interlocking systems. The requirements, derived from a railway signalling manufacturer and CENELEC standards, are defined as scenarios for general infrastructure. Then these scenarios are mapped to concrete scenarios based on the actual conditions of the given railway networks. Next, the TTCN-3 test cases reflecting the concrete scenarios are automatically executed on a simulation of the interlocking software. It is not mentioned in the paper whether the concrete scenarios and TTCN-3 are derived automatically for different railway networks.

In [Mew10], a domain-specific testing strategy for railway interlocking systems is proposed. A number of generic requirements have been identified. The testing strategy presented in this chapter is an extension of the work by Mewes [Mew10] in the following aspects:

- The generic requirements are specified using a DSL tailored for interlocking systems;
- In addition to generic requirements identified in [Mew10], generic requirements related to sequential release are also identified as presented in Section 7.3.3;
- An SMT solver was used to calculate data for test cases from the test model and test objectives, instead of implementing the test case generation directly as in [Mew10].

Equivalence class testing has been investigated as a technique to produce complete test suite for railway applications. A successful application of the technique on the *ceiling speed monitoring* module in the onboard computers in trains are reported in [Bra+14b; Bra+14a]. Application of equivalence class testing for interlocking systems are a ongoing research topic. A discussion of an initial idea can be found in [HP15].

CHAPTER 8

Conclusion

This chapter concludes the dissertation with a short summary and evaluation of the presented research. First, Section 8.1 sums up the novelties of the work and its contributions with respect to different disciplines. Afterwards, Section 8.2 evaluates the limitations of the work. Section 8.3 closes the chapter with some suggestions for future work.

8.1 Contributions and Novelties

The work presented in this dissertation focuses primarily on safety-critical software systems that have product line characteristics – i.e., these systems share common, generic components, while each individual system is constructed from the generic components by instantiating with the configuration data specific for that individual system. The work involves multiple disciplines: railway control systems, formal methods, domain-specific approaches, verification and validation of safety-critical systems, and model-based testing. Naturally, the work contributes to the advancement in all these areas. The contributions are highlighted in the following.

- *A holistic and formal method.* The work puts together both new and existing languages, tools, and techniques in the involving disciplines to produce a holistic and formal method for verification and validation of safety-critical systems with product line characteristics.

- *A fruitful combination of formal methods and domain-specific approaches.* The proposed method takes advantages of both formal methods and domain-specific approaches to provide a more efficient development process for safety-critical software systems. While the former offers mathematically rigorous specification, verification and validation, the latter encapsulates the complex mathematical foundation of the former, make it more accessible for end-users and prevents human errors. Furthermore, the method features a 4-step verification and validation approach integrated naturally into the development life cycle, allowing errors to be discovered as early as possible in the development process.

- *An extra step in using domain-specific approaches.* The use of *two domain-specific languages* – one for specifying configuration data and one for specifying generic applications – fits perfectly to systems with product line characteristics. These two domain-specific languages provide easy-to-use interfaces, and an appropriate level of abstraction such that one can focus on a single part of the system at a time. Errors, if exist, can be located efficiently. It is a novelty to have a second DSL for specifying generic applications instead of specifying generic applications in a GPL like in previous studies [JR14; Hax14; Mew10; Cao+11; Jam14]. To the best of the author's knowledge, this has not been done in any previously published methods for specification, verification and validation of railway interlocking systems.

- *Application of the method to the forthcoming Danish interlocking systems.* The application of the method to the case of the forthcoming Danish interlocking systems has shown the applicability of the method. The forth coming Danish interlocking systems exhibit the product line characteristics: while the common railway signalling rules, know-how, and safety requirements are shared between systems, each individual system has its own configuration data – the railway network layout under its control and the corresponding interlocking table. Furthermore, the forthcoming Danish interlocking systems have two extra important features: (1) they are ETCS Level 2 compatible; and (2) they feature sequential release. These two features pose extra challenges in the verification and validation tasks. Following the recipe of the proposed method, ICL – a DSL for specifying interlocking configuration data – and IDL – a DSL for specifying the generic behavioural models, safety properties, and test objectives – are developed based on the concepts and notions of the interlocking domain. The generic parts are then instantiated with the concrete configuration data for a given interlocking system, resulting in a concrete behavioural model, concrete safety properties, and concrete test objectives. The 4-step verification and validation approach is used to address the challenging verification and validation tasks.

- *Novel, formal generic applications for the forthcoming Danish interlocking systems.* The formal generic applications (generic behavioural model, safety properties, and test objectives) presented in this dissertation for the forthcoming Danish interlocking systems contain a number of novelties as listed in the following.

  - By introducing the concept of virtual signals, ETCS Level 2 compatible interlocking systems – in which there is no physical signals along the tracks – can be formalised in a similar way as conventional interlocking systems with physical signals.
  - The model is the first formal model of interlocking systems that treats sequential release in full detail, to the best of the author's knowledge.
  - An innovative encoding for occupancy status of sections allows properties (safety properties and strengthening invariants) and transition relation, especially train movement transitions, to be formalised in a succinct way. Further-

more, the encoding can be handled efficiently by the SMT solver used in our method, hence improving the verification performance.

- *Developed a prototype toolchain.* A toolchain has been developed in RT-Tester framework for the case studies of the forthcoming Danish interlocking systems.
- *Pushed the applicability bound of formal methods in verifying railway interlocking systems further.* Experimental results on different case studies including the early deployment line of the Danish Signalling Programme have shown that the 4-step verification and validation approach offered by the method can scale to the systems of industrial size despite the complexity of the sequential release feature in the systems. A comparison with other verification techniques has shown that our verification and validation approach out-performed other techniques for industrial size cases. The method is capable of verifying safety properties of the design of railway interlocking systems of a size that has not been achieved before – to the best of the author's knowledge – in previously published studies, e.g., [Win12; Jam+14].
- *An extension of a generic, domain-specific model-based testing strategy for railway interlocking systems.* The work extends the generic, domain-specific MBT testing strategy in [Mew10] with a DSL for specifying requirements and test objectives, a test generation process using an SMT solver, and requirements related to sequential release. This results in a testing strategy which can exercise the important aspects of an implementation of a given interlocking system. Although the testing strategy is specialised for the forthcoming Danish interlocking systems, it may also be adapted to suit other interlocking systems.

The success with the case studies of the forthcoming Danish interlocking systems *confirms our hypothesis* laid out in Chapter 1 that a proper combination of formal methods and domain-specific approaches leads to a more efficient verification and validation of safety-critical software systems. The applicability of the proposed method to other domains than railway control systems is also promising.

8.2 Limitations

Due to the limited time and resources during the project, there is a number of aspects that have not been addressed yet in the work presented in this dissertation as listed in the following.

- For simplicity, some aspects of the forthcoming Danish interlocking systems have not been incorporated in the model such as level crossings or failures handling.
- The static checker for IDL has not been fully implemented in our toolchain.
- The semantics of non-deterministic behaviours have not been considered in our MBT testing strategy. Deterministic behaviours are selected for simplicity. Supporting non-deterministic behaviours is an on-going research topic.

8.3 Directions for Future Work

Besides addressing the limitations listed in Section 8.2, the following directions can be investigated further in future.

- *Decomposition.* Although our verification and validation approach scales nicely to applications of industrial size, the verification and validation performance could be improved by investigating a framework for decomposing large systems into smaller ones. For the case of railway interlocking systems, the Cover Abstraction techniques proposed by James et al. [Jam+14] can serve as inspiration.
- *Refinement.* A framework for refining the generic models could be beneficial. Such a refinement framework would allow adding further detail into an already verified abstract model. Ultimately, the model would reach a sufficient level of detail to enable code generation directly from the model.
- *Non-deterministic and Equivalence Class Testing Strategies.* Theories and tools should be developed for addressing the non-deterministic behaviours in the test model, and application of equivalence class testing strategy for a complete test suite.
- *Technology Transfer.* In order to adopt the method in the industry, it should be investigated how to integrate the method in the existing development cycles and tools used in the railway industry. Although the V&V flow of the method fits nicely in different phases of the software development cycle, it is necessary to define a roadmap and develop industrial tools in order to have a seamless introduction of the method in the existing development process.
- *Applications in other Domains.* Although the method was developed based on the needs in verification of railway interlocking systems, it may also be used in other domains for applications with similar characteristics, e.g., see [ZB14].

APPENDIX A

Formal Specification of ICL in RSL

This appendix lists the formal specification of ICL described in Chapter 4. Section A.1 presents some common types and values. The abstract syntaxes and static checkers for network layouts and interlocking tables described in Chapter 4 are specified in Section A.2 and Section A.3, respectively. Section A.4 presents the specification of the Interlocking Table Generator (ITG) described in Section 4.5. The specification is formalised in the RAISE Specification Language (RSL).

A.1 Common Types and Values

```
object T :
  class
    type
      Id = Text,
      SecId = Id, -- section id
      MbId = Id, -- markerboard id
      RouteId = Id, -- route id
      Direction == UP | DOWN,
      LinearEnd = Direction, -- correspond to direction
      PointEnd == NB_STEM | NB_PLUS | NB_MINUS,
      PointPos == PLUS | MINUS,
      Distance = Nat,
      T = Text

    value
      MINOR : Text = "abcdefghijklmnopqrstuvxyz",

      nat : Direction → Nat
      nat(d) ≡
        case d of
          DOWN → 0,
```

```
    UP → 1
  end,

nat : PointEnd → Nat
nat(e) ≡
  case e of
    NB_PLUS → 0,
    NB_MINUS → 1,
    NB_STEM → 2
  end,

nat : PointPos → Nat
nat(p) ≡
  case p of
    PLUS → 0,
    MINUS → 1
  end,

/* ASCII code lookup table , for specifying string comparison */
ASCII_CODE : Char →m Nat =
  [′0′ ↦ 48, ′1′ ↦ 49, ′2′ ↦ 50, ′3′ ↦ 51, ′4′ ↦ 52, ′5′ ↦ 53,
   ′6′ ↦ 54, ′7′ ↦ 55, ′8′ ↦ 56, ′9′ ↦ 57, ′A′ ↦ 65, ′B′ ↦ 66,
   ′C′ ↦ 67, ′D′ ↦ 68, ′E′ ↦ 69, ′F′ ↦ 70, ′G′ ↦ 71, ′H′ ↦ 72,
   ′I′ ↦ 73, ′J′ ↦ 74, ′K′ ↦ 75, ′L′ ↦ 76, ′M′ ↦ 77, ′N′ ↦ 78,
   ′O′ ↦ 79, ′P′ ↦ 80, ′Q′ ↦ 81, ′R′ ↦ 82, ′S′ ↦ 83, ′T′ ↦ 84,
   ′U′ ↦ 85, ′V′ ↦ 86, ′X′ ↦ 88, ′Y′ ↦ 89, ′Z′ ↦ 90, ′a′ ↦ 97,
   ′b′ ↦ 98, ′c′ ↦ 99, ′d′ ↦ 100, ′e′ ↦ 101, ′f′ ↦ 102, ′g′ ↦ 103,
   ′h′ ↦ 104, ′i′ ↦ 105, ′j′ ↦ 106, ′k′ ↦ 107, ′l′ ↦ 108,
   ′m′ ↦ 109, ′n′ ↦ 110, ′o′ ↦ 111, ′p′ ↦ 112, ′q′ ↦ 113,
   ′r′ ↦ 114, ′s′ ↦ 115, ′t′ ↦ 116, ′u′ ↦ 117, ′v′ ↦ 118,
   ′x′ ↦ 120, ′y′ ↦ 121, ′z′ ↦ 122, ′_′ ↦ 95],

/**
* REMARK:
* ==============================================
The specification of the following functions are for illustration
purpose. These functions need to be specified explicitly in order
to be translated to SML for running the spec. In practice , these
functions can be implemented differently to ensure the performance.
*/
/* compare two string s1,s2
* s1 = s2 −> 0
* s1 > s2 −> 1
* s1 < s2 −> −1 */
compare : Text × Text → Int
compare(s1, s2) ≡
  if s1 = ⟨⟩ ∧ s2 = ⟨⟩ then 0
```

```
  else
    if s1 = ⟨⟩ then −1
    else
      if s2 = ⟨⟩ then 1
      else
        let c1 = ASCII_CODE(hd s1), c2 = ASCII_CODE(hd s2) in
          if c1 = c2 then compare(tl s1, tl s2)
          else if c1 > c2 then 1 else −1 end
          end
        end
      end
    end
  end,

sum : Int* → Int
sum(ls) ≡
  if ls = ⟨⟩ then 0 else let i = hd ls in i + sum(tl ls) end end,

/* get the opposite direction */
− : Direction → Direction
 − d ≡
 case d of
    UP → DOWN,
    DOWN → UP
  end,

/**
* RouteId has the form of [MAJOR][MINOR]=[\d+][a−z]
*/
/* make the major id of a route */
mk_major_id : Nat → RouteId
mk_major_id(i) ≡
 case i of
    1 → "01",
    2 → "02",
    3 → "03",
    4 → "04",
    5 → "05",
    6 → "06",
    7 → "07",
    8 → "08",
    9 → "09",
    10 → "09",
    11 → "11",
    12 → "12",
    13 → "13",
    14 → "14",
```

```
    15 → "15",
    16 → "16",
    17 → "17",
    18 → "18",
    19 → "19"
  end,

/* make the minor id of a route, i.e. 01 -> 01a, 01b ... */
mk_minor_id : RouteId × Nat → RouteId
mk_minor_id(major, i) ≡ major⌢⟨MINOR(i)⟩,

/* Add an item e to all the sets in s */
add_elem : T × (T-set)-set → (T-set)-set
add_elem(e, s) ≡ {{e} ∪ x | x : T-set • x ∈ s},

/**
/* abstract specification of powerset function */
powerset_abs : T-set → (T-set)-set
powerset_abs(s) ≡ {i | i : T-set • i ⊆
s},
*/
/* generate the powerset of a Text−set */
powerset : T-set → (T-set)-set
powerset(s) ≡
  if s = {} then {{}}
  else
    let
      e = hd s,
      rs = s \ {e},
      pwrs = powerset(rs),
      px = add_elem(e, pwrs)
    in
      pwrs ∪ px
    end
  end,

/* flatten a (Text−set)−set to a Text−set which includes all items
*/
flatten : (T-set)-set → T-set
flatten (ss) ≡
  if ss = {} then {}
  else let s = hd ss in s ∪ flatten (ss \ {s}) end
  end,

/* convert a set to a list */
to_list : T-set → T*
```

```
    to_list (s) ≡
      if s = {} then ⟨⟩
      else let i = hd s in ⟨i⟩ ⌢ to_list (s \ {i}) end
      end,

    index : T* × T → Nat
    index(ls, e) ≡
      let fs = {i | i : Nat • i ∈ inds ls ∧ ls(i) = e} in hd fs end
    pre e ∈ ls,

    /* sort a list of text ascendingly */
    sort : Text* → Text*
    sort(l) ≡
      if l = ⟨⟩ then ⟨⟩
      else
        let i = hd l in
          sort(⟨j | j in tl l • (compare(j, i) ≤ 0)⟩) ⌢ ⟨i⟩ ⌢
          sort(⟨j | j in tl l • (compare(j, i) > 0)⟩)
        end
      end
  end
```

A.2 Railway Network Layouts

```
T

scheme NetworkLayout =
  with T in
  class
    type
      /**
      * Abstract syntax of a railway network layout
      */
      Linear :: neighbors : LinearEnd →m SecId length : Distance,
      Point :: neighbors : PointEnd →m SecId length : Distance,
      MarkerBoard :: section : SecId dir : LinearEnd distance : Distance,
      /*NOT−SML−TRANSLATABLE*/
      Section = Linear | Point,
      NetworkLayout ::
        linears : SecId →m Linear
        points : SecId →m Point
        marker_boards : MbId →m MarkerBoard

    value
      /**
      * CONSTANTS
      */
```

```
  MIN_SECTION_LENGTH : Distance = 0

value
  /**
  * wellformedness conditions for a network layout
  */

  is_wf : NetworkLayout → Bool
  is_wf(n) ≡
    let ls = linears(n), ps = points(n), ms = marker_boards(n) in
      /* N−01) all elements have unique identifiers */
      unique_identifiers(n) ∧
      /* N−02) all linears are wellformed */
      (∀i : SecId • i ∈ ls ⇒ is_wf_l(i, n)) ∧
      /* N−03) all points are wellformed */
      (∀i : SecId • i ∈ ps ⇒ is_wf_p(i, n)) ∧
      /* N−04) all marker boards are wellformed */
      (∀i : MbId • i ∈ ms ⇒ is_wf_m(i, n)) ∧
      /* N−05) the orientation is consistent */
      orientation_is_correct (n) ∧
      /* N−06) cycle−free ( optional ) */
      no_cycles(n) ∧
      /* N−07) boundary configuration assumption */
      boundary_configuration(n) end,

  /* boundary assumption */
  boundary_configuration : NetworkLayout → Bool
  boundary_configuration(n) ≡
  let bs = get_boundaries(n) in
      (∀i : SecId • i ∈ bs ⇒ boundary_configuration_l(i, n))
    end,

  boundary_configuration_l : SecId × NetworkLayout → Bool
  boundary_configuration_l(i, n) ≡
    let l = get_linear(i, n), nbs = neighbors(l), j = hd rng nbs in
      l_exists (j, n) ∧
      if UP ∈ nbs
      then dom signals(i, n) = {UP} ∧ DOWN ∈ signals(j, n)
      else dom signals(i, n) = {DOWN} ∧ UP ∈ signals(j, n)
      end
    end
  pre is_boundary(i, n),

  /* linears and points are not overlaping */
  unique_identifiers : NetworkLayout → Bool
  unique_identifiers(n) ≡
    let
```

```
    ls = dom linears(n), ps = dom points(n), ms = dom marker_boards(n)
  in
    card (ls ∪ ps ∪ ms) = card ls + card ps + card ms
  end,

/* wellformed linear section */
is_wf_l : SecId × NetworkLayout ~→ Bool
is_wf_l(i, n) ≡
  let l = get_linear(i, n), nbs = neighbors(l) in
    /* L−01) no self−neighboring */
    (i ∉ rng nbs) ∧
    /* L−02) 1 <= no. neighbors <= 2 and distinct */
    (card dom nbs ≥ 1) ∧ (card dom nbs = card rng nbs) ∧
    /* L−03) all neighbors exist and are mutual neighboring */
    (∀j : SecId •
        j ∈ rng nbs ⇒ s_exists(j, n) ∧ i ∈ get_neighbors(j, n)) ∧
    /* L−04) all signals are distinct and no. of signals <= 2 (i.e.
    max. one signal per direction , and they must be distinct ) −> implied
    from is_wf_m, thus dont need to check here */
    /* L−05) length is greater than minimum */
    (length(l) > MIN_SECTION_LENGTH) end
pre l_exists (i, n),

/* wellformed point section */
is_wf_p : SecId × NetworkLayout ~→ Bool
is_wf_p(i, n) ≡
  let p = get_point(i, n), nbs = neighbors(p) in
    /* P−01) no self−neighboring */
    (i ∉ rng nbs) ∧
    /* P−02) no. neighbors == 3 (no border point) */
    (card dom nbs = 3) ∧
    /* P−03) neighbors are distinct */
    (card rng nbs = 3) ∧
    /* P−04) all neighbors exist and are mutual neighboring */
    (∀j : SecId •
        j ∈ rng nbs ⇒ s_exists(j, n) ∧ i ∈ get_neighbors(j, n)) ∧
    /* P−05) length is greater than minimum */
    (length(p) > MIN_SECTION_LENGTH) end
pre p_exists(i, n),

/* wellformed marker board */
is_wf_m : MbId × NetworkLayout ~→ Bool
is_wf_m(i, n) ≡
  let m = get_maker_board(i, n), si = section(m), d = dir(m) in
    /* M−01) the section exists and is a linear */
    l_exists (si, n) ∧
    /* M−02) there does not exist another marker board that is installed
```

```
    along the same section in the same direction */
    ¬ (∃j : MbId •
          j ∈ marker_boards(n) \ {i} ∧
          let
            m′ = get_maker_board(j, n), si′ = section(m′), d′ = dir(m′)
          in
            si′ = si ∧ d′ = d
          end) ∧
    /* M−03) distance is less than the section′s length */
    let l = get_linear(si, n) in distance(m) < length(l) end end
pre m_exists(i, n),

/* check whether the orientiation of a network layout is correct,
i.e. starting from a border section, we traverse the network, then
we should always go in the same direction */
orientation_is_correct : NetworkLayout ~→ Bool
orientation_is_correct(n) ≡
  let
    nols =
      {i |
       i : SecId •
         i ∈ linears(n) ∧ DOWN ∉ neighbors(get_linear(i, n))},
    nors =
      {i |
       i : SecId •
         i ∈ linears(n) ∧ UP ∉ neighbors(get_linear(i, n))}
  in
    (∀i : SecId •
       i ∈ nols ⇒
         let l = get_linear(i, n) in
           check_orientation(up(l), i, UP, n)
         end) ∧
    (∀i : SecId •
       i ∈ nors ⇒
         let l = get_linear(i, n) in
           check_orientation(down(l), i, DOWN, n)
         end)
  end,

check_orientation :
  SecId × SecId × Direction × NetworkLayout ~→ Bool
check_orientation(i, prev, d, n) ≡
  if l_exists(i, n)
  then
    let l = get_linear(i, n), nbs = neighbors(l) in
      (prev ∈ rng nbs) ∧ (get_l_end_by_nb_id(i, prev, n) ≠ d) ∧
      (∀j : SecId •
```

```
            j ∈ (rng nbs) \ {prev} ⇒ check_orientation(j, i , d, n))
        end
      else
        let
          p = get_point(i, n),
          nbs = neighbors(p),
          side = get_p_end_by_nb_id(i, prev, n)
        in
          (prev ∈ rng nbs) ∧
          case side of
            NB_STEM →
              check_orientation(plus(p), i , d, n) ∧
              check_orientation(minus(p), i , d, n),
            _ → check_orientation(stem(p), i, d, n)
          end
        end
      end
    pre s_exists(i , n),

    /* no cycles in the network */
    no_cycles : NetworkLayout → Bool
    no_cycles(n) ≡ let ts = sections(n) in ¬ has_cycle(ts , {}, n) end,

    /* detect whether a network has cycles using
    * graph search , more efficient algorithms such
    * as union−find can be used in the implementation
    */
    has_cycle : SecId-set × SecId-set × NetworkLayout → Bool
    has_cycle(queue, visited , n) ≡
      if queue = {} then false
      else
        let
          s = hd queue,
          rest = queue \ {s},
          nbs = get_neighbors(s, n),
          next = rest ∪ (nbs \ visited)
        in
          s ∈ visited ∨ has_cycle(next, visited ∪ {s}, n)
        end
      end

  value
    /**
    * Getters
    */
    get_linear : SecId × NetworkLayout ~→ Linear
    get_linear(i , n) ≡ linears(n)(i) pre l_exists(i , n),
```

```
get_point : SecId × NetworkLayout ~→ Point
get_point(i, n) ≡ points(n)(i) pre p_exists(i, n),

get_boundaries : NetworkLayout → SecId-set
get_boundaries(n) ≡
 let ls = linears(n) in
    {i | i : SecId • i ∈ ls ∧ is_boundary(i, n)}
  end
pre (∀i : SecId • i ∈ linears(n) ⇒ is_wf_l(i, n)),

/*NOT−SML−TRANSLATABLE*/
get_section : SecId × NetworkLayout ~→ Section
get_section(i, n) ≡
  if l_exists(i, n) then get_linear(i, n) else get_point(i, n) end
pre s_exists(i, n),

/* get the signals associated with a linear
* section */
signals : SecId × NetworkLayout ~→ (LinearEnd ~m→ MbId)
signals(i, n) ≡
  let mbs = marker_boards(n) in
    [dir(get_maker_board(mi, n)) ↦ mi |
     mi : MbId • mi ∈ mbs ∧ section(get_maker_board(mi, n)) = i]
  end
pre l_exists(i, n),

/* get the signal in the down direction of
* a linear section */
dsig : SecId × NetworkLayout ~→ MbId
dsig(i, n) ≡ let sigs = signals(i, n) in sigs(DOWN) end
pre l_exists(i, n) ∧ DOWN ∈ signals(i, n),

/* get the signal in the up direction of a
* linear section */
usig : SecId × NetworkLayout ~→ MbId
usig(i, n) ≡ let sigs = signals(i, n) in sigs(UP) end
pre l_exists(i, n) ∧ UP ∈ signals(i, n),

/* get the "down" neighbor of a linear section */
down : Linear ~→ SecId
down(l) ≡ let nbs = neighbors(l) in nbs(DOWN) end
pre DOWN ∈ neighbors(l),

/* get the "up" neighbor of a linear section */
up : Linear ~→ SecId
```

```
up(l) ≡ let nbs = neighbors(l) in nbs(UP) end pre UP ∈ neighbors(l),

/* get the stem neighbor of a point section */
stem : Point ~→ SecId
stem(p) ≡ let nbs = neighbors(p) in nbs(NB_STEM) end
pre NB_STEM ∈ neighbors(p),

/* get the plus neighbor of a point section */
plus : Point ~→ SecId
plus(p) ≡ let nbs = neighbors(p) in nbs(NB_PLUS) end
pre NB_PLUS ∈ neighbors(p),

/* get the minus neighbor of a point section */
minus : Point ~→ SecId
minus(p) ≡ let nbs = neighbors(p) in nbs(NB_MINUS) end
pre NB_MINUS ∈ neighbors(p),

/* get length of a section */
get_length : SecId × NetworkLayout ~→ Distance
get_length(i, n) ≡
  if l_exists(i, n) then let s = get_linear(i, n) in length(s) end
  else let s = get_point(i, n) in length(s) end
  end
pre s_exists(i, n),

/* get the neighbors of a given section */
get_neighbors : SecId × NetworkLayout ~→ SecId-set
get_neighbors(i, n) ≡
  if p_exists(i, n) then rng neighbors(get_point(i, n))
  else rng neighbors(get_linear(i, n))
  end
pre s_exists(i, n),

/* get the linear end correspondingto a given neighbor id (s) of
the linear section with id (i) */
get_l_end_by_nb_id : SecId × SecId × NetworkLayout ~→ LinearEnd
get_l_end_by_nb_id(i, s, n) ≡
  let l = get_linear(i, n), nbs = neighbors(l) in
    hd {j | j : LinearEnd • j ∈ nbs ∧ nbs(j) = s}
  end
pre l_exists(i, n) ∧ s ∈ rng neighbors(get_linear(i, n)),

/* get the point end correspondingto a given neighbor id (s) of
the point section with id (i) */
get_p_end_by_nb_id : SecId × SecId × NetworkLayout ~→ PointEnd
get_p_end_by_nb_id(i, s, n) ≡
```

```
      let p = get_point(i, n), nbs = neighbors(p) in
        hd {j | j : PointEnd • j ∈ nbs ∧ nbs(j) = s}
      end
    pre p_exists(i, n) ∧ s ∈ rng neighbors(get_point(i, n)),

    get_maker_board : MbId × NetworkLayout ~→ MarkerBoard
    get_maker_board(i, n) ≡ marker_boards(n)(i) pre m_exists(i, n),

    sections : NetworkLayout ~→ SecId-set
    sections(n) ≡ dom linears(n) ∪ dom points(n)

  value
    /**
    * Auxiliary functions
    */
    l_exists : SecId × NetworkLayout → Bool
    l_exists(i, n) ≡ i ∈ linears(n),

    p_exists : SecId × NetworkLayout → Bool
    p_exists(i, n) ≡ i ∈ points(n),

    s_exists : SecId × NetworkLayout → Bool
    s_exists(i, n) ≡ l_exists(i, n) ∨ p_exists(i, n),

    m_exists : MbId × NetworkLayout → Bool
    m_exists(i, n) ≡ i ∈ marker_boards(n),

    is_boundary : SecId × NetworkLayout → Bool
    is_boundary(i, n) ≡
      l_exists(i, n) ∧
      let l = get_linear(i, n) in card dom neighbors(l) = 1 end,

    /* neighboring is reflexive */
    are_neighbors : SecId × SecId × NetworkLayout ~→ Bool
    are_neighbors(i, j, n) ≡
      let nbsi = get_neighbors(i, n), nbsj = get_neighbors(j, n) in
        i ∈ nbsj ∧ j ∈ nbsi
      end
    pre s_exists(i, n) ∧ s_exists(j, n)
  end
```

A.3 Interlocking Tables

```
NetworkLayout

scheme Interlocking =
  with T in
```

```
class
  object L : NetworkLayout

  type
    /**
    * Abstract syntax of an interlocking table
    */
    Route ::
      source : MbId
      dest : MbId
      path : SecId*
      overlap : SecId*
      points : SecId m→ PointPos
      signals : MbId-set
      conflicts : RouteId-set,
    InterlockingTable = RouteId m→ Route,
    Interlocking ::
      track_layout : L.NetworkLayout
      interlocking_table : InterlockingTable

  value
    /**
    * WELLFORMED INTERLOCKING CONFIGURATION
    */
    is_wf : Interlocking → Bool
    is_wf(ixl) ≡
      let n = track_layout(ixl), rt = interlocking_table(ixl) in
        /* I−01) network layout is wellformed */
        L.is_wf(n) ∧
        /* I−02) interlocking table is wellformed w.r.t. the network */
        is_wf_rt(rt, n) end

  type
    /**
    * Auxiliary types
    */
    /* a protection suite for an element of a route, or a route */
    ProtectionSuite :: signals : MbId-set points : SecId m→ PointPos

  value
    /**
    * CONSTANTS
    */
    MIN_SAFETY_DISTANCE : Distance = 50,
    /* an empty protection suite */
    empty : ProtectionSuite = mk_ProtectionSuite({}, [])
```

```
value
  /**
  * Wellformed route table
  */
  is_wf_rt : InterlockingTable × L.NetworkLayout → Bool
  is_wf_rt(tb, n) ≡
    /* T−01) routes identifiers are unique (ensure by the map) and differs
    from identifiers of elements in network layout */
    let
      js =
        (dom tb) ∩
       (dom L.linears(n) ∪ dom L.points(n) ∪
        dom L.marker_boards(n))
    in
      js = {}
    end ∧
    /* T−02) routes are distinct */
    (card dom tb = card rng tb) ∧
    /* T−03) all routes are wellformed */
    routes_are_wellformed(tb, n) ∧
    /* T−04) no self conflicting */
    no_self_conflicting (tb) ∧
    /* T−05) conflicts are mutual */
    conflicts_are_mutual(tb) ∧
    /* T−06) conflicting routes information is correct */
    conflicts_are_correct (tb, n) pre L.is_wf(n),

  routes_are_wellformed : InterlockingTable × L.NetworkLayout → Bool
  routes_are_wellformed(tb, n) ≡
   (∀r : Route • r ∈ rng tb ⇒ is_wf_r(r, n))
  pre L.is_wf(n),

  no_self_conflicting : InterlockingTable ~→ Bool
  no_self_conflicting (tb) ≡
    (∀i : RouteId •
       i ∈ tb ⇒
         let r = get_route(i, tb), cs = conflicts (r) in
           /* no self − conflicting */
           (i ∉ cs) end),

  conflicts_are_mutual : InterlockingTable ~→ Bool
  conflicts_are_mutual(tb) ≡
    (∀i : RouteId •
       i ∈ tb ⇒
         let r = get_route(i, tb), cs = conflicts (r) in
           (∀j : RouteId •
              j ∈ cs ⇒
```

```
                /* conflict relation is symmetric */
                r_exists(j, tb) ∧
                let r1 = get_route(j, tb), cs1 = conflicts(r1) in
                  i ∈ cs1
                end)
        end),


  conflicts_are_correct : InterlockingTable × L.NetworkLayout ~→ Bool
  conflicts_are_correct(tb, n) ≡
    (∀i : RouteId •
       i ∈ tb ⇒
         let r = get_route(i, tb), cs = conflicts(r) in
           /* routes are physically in conflict are marked */
           (∀j : RouteId •
              j ∈ tb \ {i} ⇒
                let r1 = get_route(j, tb) in
                  are_physically_in_conflict(r, r1, n) ⇒
                    marked_as_being_in_conflict(i, j, tb)
                end) end)
  pre L.is_wf(n) ∧ routes_are_wellformed(tb, n)

value
  /**
  * Auxiliary functions
  */
  /* whether two routes are in conflict based on the information about
  the network layout */
  are_physically_in_conflict : Route × Route × L.NetworkLayout ~→ Bool
  are_physically_in_conflict(r1, r2, n) ≡
    let
      path1 = elems path(r1),
      path2 = elems path(r2),
      ovs1 = elems overlap(r1),
      ovs2 = elems overlap(r2),
      secs1 = path1 ∪ ovs1,
      secs2 = path2 ∪ ovs2,
      ps1 = points(r1),
      ps2 = points(r2),
      protecting_points1 = dom ps1 \ secs1,
      protecting_points2 = dom ps2 \ secs2,
      shared_pps =
        (protecting_points1 ∩ dom ps2) ∪
       (protecting_points2 ∩ dom ps1)
    in
      /* not consecutive routes, overlapping paths or overlap */
      (¬ are_concatenated_routes(r1, r2, n) ∧ secs1 ∩ secs2 ≠ {}) ∨
```

```
      /* share a point, the shared point is a protecting point for at
      least one of the routes and two routes require the points in different
      positions */
      (∃i : SecId • i ∈ shared_pps ∧ ps1(i) ≠ ps2(i)) ∨
      /* entry signal of one route is a protecting
      * signal of the other route */
      (source(r1) ∈ signals(r2)) ∨ (source(r2) ∈ signals(r1))
    end
  pre L.is_wf(n) ∧ is_wf_r(r1, n) ∧ is_wf_r(r2, n),

  /* whether two routes are concatenated routes */
  are_concatenated_routes : Route × Route × L.NetworkLayout ~→ Bool
  are_concatenated_routes(r1, r2, n) ≡
    source(r1) = dest(r2) ∨ dest(r2) = source(r1)
  pre L.is_wf(n) ∧ is_wf_r(r1, n) ∧ is_wf_r(r2, n),

  /* check whether two routes are marked as being in conflict in their
   specification */
  marked_as_being_in_conflict :
    RouteId × RouteId × InterlockingTable ~→ Bool
  marked_as_being_in_conflict(i, j, tb) ≡
    let
      r1 = get_route(i, tb),
      r2 = get_route(j, tb),
      cs1 = conflicts(r1),
      cs2 = conflicts(r2)
    in
      i ∈ cs2 ∧ j ∈ cs1
    end
  pre r_exists(i, tb) ∧ r_exists(j, tb)

value
  /**
  * Getters
  */
  /* get the first detection section of a route */
  first : Route ~→ SecId
  first(r) ≡ hd path(r) pre len path(r) > 0,

  /* get the last detectin section of a route */
  last : Route ~→ SecId
  last(r) ≡ let p = path(r) in p(len p) end pre len path(r) > 0,

  prev : Route × SecId ~→ SecId
  prev(r, i) ≡ let ps = path(r), j = index(ps, i) − 1 in ps(j) end
  pre i ∈ path(r) ∧ i ≠ first(r),
```

```
prevs : Route × SecId ~→ SecId*
prevs(r, i) ≡
  let ps = path(r), j = index(ps, i) − 1 in
    ⟨ps(i) | i in ⟨0 .. j⟩⟩
  end
pre i ∈ path(r),

next : Route × SecId ~→ SecId
next(r, i) ≡ let ps = path(r), j = index(ps, i) + 1 in ps(j) end
pre i ∈ path(r) ∧ i ≠ last(r),

nexts : Route × SecId ~→ SecId*
nexts(r, i) ≡
  let ps = path(r), j = index(ps, i) + 1 in
    ⟨ps(i) | i in ⟨j .. len (ps)⟩⟩
  end
pre i ∈ path(r),

get_route : RouteId × InterlockingTable ~→ Route
get_route(i, tb) ≡ tb(i) pre r_exists(i, tb),

get_route : RouteId × Interlocking′ ~→ Route
get_route(i, ixl) ≡ get_route(i, interlocking_table(ixl))
pre r_exists(i, interlocking_table(ixl)),

r_exists : RouteId × InterlockingTable → Bool
r_exists(i, tb) ≡ i ∈ tb,

r_exists : RouteId × Interlocking′ → Bool
r_exists(i, ixl) ≡ r_exists(i, interlocking_table(ixl))

type
  −− define a subtype of wellformed interlocking configuration data
  −− so we don't need to add wellformedness check as pre conditions
  −− for functions used for defining semantics of IDL
  Interlocking′ = {| ixl : Interlocking • is_wf(ixl) |}

value
  /**
  * FUNCTIONS FOR THE SPECIFICATION OF IDL SEMANTICS
  * ================================================
  * Some of these functions are not SML translatable,
  * comment out when translating to SML
  */
```

```
down : SecId × Interlocking′ ~→ SecId
down(i, ixl) ≡
  let n = track_layout(ixl), l = L.get_linear(i, n) in L.down(l) end
pre
  let n = track_layout(ixl) in
    L.l_exists(i, n) ∧ DOWN ∈ L.neighbors(L.get_linear(i, n))
  end,

up : SecId × Interlocking′ ~→ SecId
up(i, ixl) ≡
  let n = track_layout(ixl), l = L.get_linear(i, n) in L.up(l) end
pre
  let n = track_layout(ixl) in
    L.l_exists(i, n) ∧ UP ∈ L.neighbors(L.get_linear(i, n))
  end,

down_sig : SecId × Interlocking′ ~→ MbId
down_sig(i, ixl) ≡
  let n = track_layout(ixl), mid = L.dsig(i, n) in mid end
pre
  let n = track_layout(ixl) in
    L.l_exists(i, n) ∧ DOWN ∈ L.signals(i, n)
  end,

up_sig : SecId × Interlocking′ ~→ MbId
up_sig(i, ixl) ≡
  let n = track_layout(ixl), mid = L.usig(i, n) in mid end
pre
  let n = track_layout(ixl) in
    L.l_exists(i, n) ∧ UP ∈ L.signals(i, n)
  end,

stem : SecId × Interlocking′ ~→ SecId
stem(i, ixl) ≡
  let n = track_layout(ixl), p = L.get_point(i, n) in L.stem(p) end
pre L.p_exists(i, track_layout(ixl)),

plus : SecId × Interlocking′ ~→ SecId
plus(i, ixl) ≡
  let n = track_layout(ixl), p = L.get_point(i, n) in L.plus(p) end
pre L.p_exists(i, track_layout(ixl)),
```

```
minus : SecId × Interlocking′ ~→ SecId
minus(i, ixl) ≡
  let n = track_layout(ixl), p = L.get_point(i, n) in L.minus(p) end
pre L.p_exists(i, track_layout(ixl)),

dir : MbId × Interlocking′ → Nat
dir(i, ixl) ≡
  let n = track_layout(ixl), m = L.get_maker_board(i, n) in
    nat(L.dir(m))
  end
pre L.m_exists(i, track_layout(ixl)),

track : MbId × Interlocking′ ~→ SecId
track(i, ixl) ≡
  let n = track_layout(ixl), m = L.get_maker_board(i, n) in
    L.section(m)
  end
pre L.m_exists(i, track_layout(ixl)),

src : RouteId × Interlocking′ ~→ MbId
src(i, ixl) ≡ let r = get_route(i, ixl) in source(r) end
pre r_exists(i, ixl),

dst : RouteId × Interlocking′ ~→ MbId
dst(i, ixl) ≡ let r = get_route(i, ixl) in dest(r) end
pre r_exists(i, ixl),

first : RouteId × Interlocking′ ~→ SecId
first(i, ixl) ≡ let r = get_route(i, ixl) in first(r) end
pre r_exists(i, ixl),

last : RouteId × Interlocking′ ~→ SecId
last(i, ixl) ≡ let r = get_route(i, ixl) in last(r) end
pre r_exists(i, ixl),

path : RouteId × Interlocking′ ~→ SecId*
path(i, ixl) ≡ let r = get_route(i, ixl) in path(r) end
```

```
pre r_exists(i, ixl),

overlap : RouteId × Interlocking′ ~→ SecId*
overlap(i, ixl) ≡ let r = get_route(i, ixl) in overlap(r) end
pre r_exists(i, ixl),

points : RouteId × Interlocking′ ~→ (SecId →m Nat)
points(i, ixl) ≡
  let r = get_route(i, ixl), ps = points(r) in
    [k ↦ nat(ps(k)) | k : SecId • k ∈ ps]
  end
pre r_exists(i, ixl),

signals : RouteId × Interlocking′ ~→ MbId-set
signals(i, ixl) ≡ let r = get_route(i, ixl) in signals(r) end
pre r_exists(i, ixl),

conflicts : RouteId × Interlocking′ ~→ RouteId-set
conflicts(i, ixl) ≡ let r = get_route(i, ixl) in conflicts(r) end
pre r_exists(i, ixl),

prev : RouteId × SecId × Interlocking′ ~→ SecId
prev(ri, i, ixl) ≡ let r = get_route(ri, ixl) in prev(r, i) end
pre
  r_exists(ri, ixl) ∧
  let r = get_route(ri, ixl) in i ∈ path(r) ∧ i ≠ first(r) end,

prevs : RouteId × SecId × Interlocking′ ~→ SecId*
prevs(ri, i, ixl) ≡ let r = get_route(ri, ixl) in prevs(r, i) end
pre
  r_exists(ri, ixl) ∧ let r = get_route(ri, ixl) in i ∈ path(r) end,

next : RouteId × SecId × Interlocking′ ~→ SecId
next(ri, i, ixl) ≡ let r = get_route(ri, ixl) in next(r, i) end
pre
  r_exists(ri, ixl) ∧
  let r = get_route(ri, ixl) in i ∈ path(r) ∧ i ≠ last(r) end,
```

```
nexts : RouteId × SecId × Interlocking′ ~→ SecId*
nexts(ri, i, ixl) ≡ let r = get_route(ri, ixl) in nexts(r, i) end
pre
  r_exists(ri, ixl) ∧ let r = get_route(ri, ixl) in i ∈ path(r) end,


conn_end : SecId × SecId × Interlocking′ ~→ Nat
conn_end(i, j, ixl) ≡
  let n = track_layout(ixl) in
    if L.l_exists(i, n) then nat(L.get_l_end_by_nb_id(i, j, n))
    else nat(L.get_p_end_by_nb_id(i, j, n))
    end
  end
pre L.are_neighbors(i, j, track_layout(ixl)),


entry : RouteId × SecId × Interlocking′ ~→ Nat
entry(ri, i, ixl) ≡
  let r = get_route(ri, ixl) in
    if i = first(r)
    then
      let
        m = L.get_maker_board(source(r), track_layout(ixl)),
        j = L.section(m)
      in
        conn_end(i, j, ixl)
      end
    else conn_end(i, prev(r, i), ixl)
    end
  end
pre r_exists(ri, ixl) ∧ i ∈ path(get_route(ri, ixl)),


exit : RouteId × SecId × Interlocking′ ~→ Nat
exit(ri, i, ixl) ≡
  let r = get_route(ri, ixl) in
    if i = last(r)
    then
      let m = L.get_maker_board(dest(r), track_layout(ixl)) in
        nat(L.dir(m))
      end
    else conn_end(i, next(r, i), ixl)
    end
  end
pre r_exists(ri, ixl) ∧ i ∈ path(get_route(ri, ixl)),
```

```
  req : RouteId × SecId × Interlocking′ ~→ Nat
  req(ri, i, ixl) ≡ let r = get_route(ri, ixl) in nat(points(r)(i)) end
  pre r_exists(ri, ixl) ∧ i ∈ points(get_route(ri, ixl))

value
  /**
  * Wellformed route
  */
  is_wf_r : Route × L.NetworkLayout → Bool
  is_wf_r(r, n) ≡
    /* R−01) source and destination signals exist and agree on their
    direction */
    signals_exist_and_agree(r, n) ∧
    /* R−02) protecting signals exist, do not contain source and destination
    signals */
    protecting_signals_exist(r, n) ∧
    /* R−03) all points exist */
    points_exist(r, n) ∧
    /* R−04) all elements in the path and overlap exist */
    elems_in_path_and_ovs_exist(r, n) ∧
    /* R−05−−12) conditions on the path*/
    route_path_cnd(r, n) ∧
    /* R−13) the route has proper protection */
    has_proper_protection(r, n) pre L.is_wf(n),

  /* all the elements in the route's path and overlap exist */
  elems_in_path_and_ovs_exist : Route × L.NetworkLayout → Bool
  elems_in_path_and_ovs_exist(r, n) ≡
   let es = elems path(r) ∪ elems overlap(r), ps = points(r) in
      (∀i : SecId •
         i ∈ es ⇒ L.l_exists(i, n) ∨ (L.p_exists(i, n) ∧ i ∈ ps))
    end
  pre L.is_wf(n),

  /* source and destination signals exist and their directions agree
  with each other */
  signals_exist_and_agree : Route × L.NetworkLayout → Bool
  signals_exist_and_agree(r, n) ≡
    let si = source(r), di = dest(r) in
      /* source exists */
      L.m_exists(si, n) ∧
      /* destination exists */
      L.m_exists(di, n) ∧
      /* source and destination's direction agrees */
      let s = L.get_maker_board(si, n), d = L.get_maker_board(di, n) in
        L.dir(s) = L.dir(d)
      end end
```

```
pre L.is_wf(n),

/* all the points of the route exist */
points_exist : Route × L.NetworkLayout → Bool
points_exist(r, n) ≡
  (∀i : SecId • i ∈ points(r) ⇒ L.p_exists(i, n))
pre L.is_wf(n),

/* all the protecting signals of the route exist */
protecting_signals_exist : Route × L.NetworkLayout → Bool
protecting_signals_exist(r, n) ≡
  let sigs = signals(r) in
    (∀i : MbId • i ∈ sigs ⇒ L.m_exists(i, n)) ∧
    source(r) ∉ sigs ∧ dest(r) ∉ sigs
  end
pre L.is_wf(n),

/* the conditions on the route's full path */
route_path_cnd : Route × L.NetworkLayout ~→ Bool
route_path_cnd(r, n) ≡
  let
    ps = path(r),
    ovs = overlap(r),
    pps = points(r),
    s = L.get_maker_board(source(r), n),
    d = L.get_maker_board(dest(r), n),
    dir = L.dir(d),
    fst = L.section(s),
    lst = L.section(d),
    l = L.get_linear(lst, n),
    l_nbs = L.neighbors(l),
    rp = ⟨fst⟩ ⌢ ps ⌢ ovs,
    le = rp(len rp),
    safety_dx = L.distance(d) + sum(⟨L.get_length(o, n) | o in ovs⟩)
  in
    /* R−06) the route's path has at least length 1 */
    (len ps ≥ 1) ∧
    /* R−07) if the safety distance is less then minimum then the last
    section of the path is a border section */
    (safety_dx ≥ MIN_SAFETY_DISTANCE ∨ dir ∉ l_nbs) ∧
    /* R−09) the last section is where the exit signal is */
    (lst = ps(len ps)) ∧
    /* R−10) no signal in the middle of the path, going in the same
    direction */
    (∀i : Nat •
      i ∈ ⟨1 .. (len ps − 1)⟩ ⇒
        (L.l_exists(ps(i), n) ⇒
```

```
            let sigs = L.signals(ps(i), n) in L.dir(s) ∉ sigs end)) ∧
    /* R−11) the whole path is acyclic */
    (len rp = card (elems rp)) ∧
    /* R−11) the whole path is connected */
    (∀i : Nat •
       i ∈ ⟨1 .. (len rp − 1)⟩ ⇒
         L.are_neighbors(rp(i), rp(i + 1), n)) ∧
    /* R−05,12) the path MUST NOT go through a point via PLUS−MINUS
    and the position should be specified in the route's point setting
    */
    (∀i : Nat •
       i ∈ ⟨2 .. (len rp − 1)⟩ ⇒
         (L.p_exists(rp(i), n) ⇒
            let
              p = L.get_point(rp(i), n),
              nbs = L.neighbors(p),
              r_nbs =
                [e ↦ nbs(e) |
                 e : PointEnd •
                   e ∈ nbs ∧ nbs(e) ∈ {rp(i − 1), rp(i + 1)}]
            in
              dom nbs ≠ {NB_PLUS, NB_MINUS} ∧ rp(i) ∈ pps ∧
              if (NB_PLUS ∈ r_nbs) then pps(rp(i)) = PLUS
              else pps(rp(i)) = MINUS
              end
            end)) ∧
    /* R−05) the last section of the route if is a point, should be
     specified with correct position in the route's points */
    (L.p_exists(le, n) ⇒
       let ld = L.get_p_end_by_nb_id(le, rp(len rp − 1), n) in
         le ∈ pps ∧
         case ld of
           NB_PLUS → pps(le) = PLUS,
           NB_MINUS → pps(le) = MINUS,
           _ → true
         end
       end)
  end
pre
  L.is_wf(n) ∧ signals_exist_and_agree(r, n) ∧ points_exist(r, n) ∧
  elems_in_path_and_ovs_exist(r, n),

/* if a route has proper protection */

has_proper_protection : Route × L.NetworkLayout ~→ Bool
has_proper_protection(r, n) ≡
  let
```

```
    px = path(r),
    ovs = overlap(r),
    s = L.get_maker_board(source(r), n),
    d = L.dir(s),
    fst = L.section(s),
    rp = ⟨fst⟩ ⌢ px ⌢ ovs,
    lst = rp(len rp)
  in
    (∀i : Nat •
       i ∈ ⟨2 .. (len rp − 1)⟩ ⇒
         (L.p_exists(rp(i), n) ⇒
            let
              p = L.get_point(rp(i), n),
              j = hd (rng L.neighbors(p) \ {rp(i − 1), rp(i + 1)})
            in
              has_proper_protection(r, rp(i), j, n)
            end)) ∧
    /* handle the last segment separately this includes both flank and
    front protection */
    (L.p_exists(lst, n) ⇒
       let
         prev = rp(len rp − 1),
         p = L.get_point(lst, n),
         nbs = L.neighbors(p),
         nbx = rng nbs \ {prev}
       in
         (∀j : SecId •
            j ∈ nbx ⇒ has_proper_protection(r, lst, j, n))
       end) ∧
    /* signals going in the opposite direction mounted in the path or
    overlap must be in the protecting signals set */
    (∀i : Nat •
       i ∈ ⟨2 .. (len rp)⟩ ⇒
         (L.l_exists (rp(i), n) ⇒
            let sigs = L.signals(rp(i), n), osigs = rng (sigs \ {d}) in
              osigs ⊆ signals(r)
            end))
  end
pre L.is_wf(n) ∧ route_path_cnd(r, n),

/* if a section i is protected by section j */
has_proper_protection :
  Route × SecId × SecId × L.NetworkLayout ~→ Bool
has_proper_protection(r, i, j, n) ≡
  let
    sigs = signals(r), ps = points(r), suite = find_protection(i, j, n)
  in
```

```
      covered_by(suite, r, n)
    end
  pre
    L.is_wf(n) ∧ route_path_cnd(r, n) ∧ L.s_exists(i, n) ∧
    L.s_exists(j, n),

  /* if the route's protection suite contain the given protection
  suite */
  covered_by : ProtectionSuite × Route × L.NetworkLayout → Bool
  covered_by(s, r, n) ≡
    /* the suite is already covered by the route's protection suite
    */
    (s ⊆ protection(r)) ∨
    let
      diff = s \ protection(r), sigs = signals(diff), ps = points(diff)
    in
      /* if we have extra protecting signals, they cannot be transferred,
      i.e., sigs ~= {} => false */
      sigs = {} ∧
      /* there should be more protecting points to transfer */
      ps ≠ [] ∧
      /* find an alternative protecting suite */
      let s′ = find_alt(s, dom ps, r, n) in
        /* alternatives found */
        (s′ ≠ empty) ∧
        /* check if alternative suite is covered by the route's suite */
        covered_by(s′, r, n) end
    end pre L.is_wf(n)

value
  /**
  * Auxiliary functions for protection search
  */
  /* covered by relation between protection suites, sa <<= sb denotes
  sa is covered by sb */
  ⊆ : ProtectionSuite × ProtectionSuite → Bool
  sa ⊆ sb ≡
    signals(sa) ⊆ signals(sb) ∧
    let psa = points(sa), psb = points(sb) in
      (∀i : SecId • i ∈ psa ⇒ i ∈ psb ∧ psb(i) = psa(i))
    end,

  /* protection suite subtraction */
  \ : ProtectionSuite × ProtectionSuite → ProtectionSuite
  sa \ sb ≡
    let
      sigs = signals(sa) \ signals(sb),
```

```
    psa = points(sa),
    psb = points(sb),
    ps =
      [i ↦ psa(i) |
       i : SecId • i ∈ psa ∧ (i ∉ psb ∨ psa(i) ≠ psb(i))]
  in
    mk_ProtectionSuite(sigs, ps)
  end,

/* protection suite union */
∪ : ProtectionSuite × ProtectionSuite → ProtectionSuite
sa ∪ sb ≡
  let
    sigs = signals(sa) ∪ signals(sb),
    psa = points(sa),
    psb = points(sb),
    cs = dom psa ∩ dom psb,
    ps =
      (psa \ cs) ∪
      [i ↦ psa(i) | i : SecId • i ∈ psa ∧ i ∉ cs] ∪
      [i ↦ psb(i) | i : SecId • i ∈ psb ∧ i ∉ cs]
  in
    mk_ProtectionSuite(sigs, ps)
  end
pre ¬ (sa # sb),

/* protection suite conflicting */
# : ProtectionSuite × ProtectionSuite → Bool
sa # sb ≡
  let psa = points(sa), psb = points(sb), cs = dom psa ∩ dom psb in
    (∃p : SecId • p ∈ cs ∧ psa(p) ≠ psb(p))
  end,

protection : Route → ProtectionSuite
protection(r) ≡
  mk_ProtectionSuite(signals(r), points(r) \ elems path(r)),

/* find the set of replacing signals for a given protecting point
*/

find_replacing_signals : SecId × Route × L.NetworkLayout ~→ MbId-set
find_replacing_signals(i, r, n) ≡
  let
    p = L.get_point(i, n),
    nbs = L.neighbors(p),
    others =
      {s |
```

```
        s : PointEnd •
          s ∈ nbs ∧
          nbs(s) ∉ (elems path(r) ∪ elems overlap(r))},
    s_stem = find_protection(i, L.stem(p), n),
    s_other =
      find_protection(
          i, if NB_PLUS ∈ others then L.plus(p) else L.minus(p) end, n),
    s_ps = points(s_stem) ∪ points(s_other),
    s_sigs = signals(s_stem) ∪ signals(s_other)
  in
    if s_sigs ≠ {} ∧ s_ps = [] then s_sigs else {} end
  end
pre L.is_wf(n) ∧ L.p_exists(i, n),

/* find an alternative protection suite by replacing a set of points
by their replacing signals */
find_alt :
  ProtectionSuite × SecId-set × Route × L.NetworkLayout ~→
    ProtectionSuite
find_alt(s, ps, r, n) ≡
  let sigs = signals(s), s_ps = points(s) in
    if ps = {} then s
    else
      let
        i = hd ps,
        a_sigs = find_replacing_signals(i, r, n),
        s_ps = points(s)
      in
        if a_sigs ≠ {}
        then
          find_alt(
              mk_ProtectionSuite(sigs ∪ a_sigs, s_ps \ {i}),
              ps \ {i}, r, n)
        else
          /* cannot replace the points with signals */
          empty
        end
      end
    end
  end
pre
  L.is_wf(n) ∧
  let sigs = signals(s), s_ps = points(s) in
    (∀m : MbId • m ∈ sigs ⇒ L.m_exists(m, n)) ∧
    (∀i : MbId • i ∈ s_ps ⇒ L.p_exists(i, n)) ∧ ps ⊆ dom s_ps
  end,
```

```
/* get protecting signal for i, preventing traffic from j to go
toward i */

find_protection : SecId × SecId × L.NetworkLayout ~→ ProtectionSuite
find_protection(i, j, n) ≡
  if L.l_exists (j, n)
  then
    let
      l = L.get_linear(j, n),
      nbs = L.neighbors(l),
      sigs = L.signals(j, n),
      d = L.get_l_end_by_nb_id(j, i, n)
    in
      if d ∉ sigs
      then
        /* if we haven't reached the border */
        if (−d) ∈ nbs then find_protection(j, nbs(−d), n)
        else empty
        end
      else mk_ProtectionSuite({sigs(d)}, [])
      end
    end
  else
    let p = L.get_point(j, n), e = L.get_p_end_by_nb_id(j, i, n) in
      case e of
        NB_PLUS → mk_ProtectionSuite({}, [j ↦ MINUS]),
        NB_MINUS → mk_ProtectionSuite({}, [j ↦ PLUS]),
        NB_STEM →
          let
            p_s = find_protection(j, L.plus(p), n),
            m_s = find_protection(j, L.minus(p), n)
          in
            if (p_s = empty) ∨ (m_s = empty) ∨ p_s # m_s
            then
              /* get rubbish information, return empty */
              empty
            else p_s ∪ m_s
            end
          end
      end
    end
  end
pre L.is_wf(n) ∧ L.are_neighbors(i, j, n)
end
```

A.4 Interlocking Table Generator

Interlocking

```
scheme InterlockingTableGenerator =
  with T in
  class
    object I : Interlocking

    type
      -- define a subtype of wellformed network layouts
      NetworkLayout′ = {| n : I.L.NetworkLayout • I.L.is_wf(n) |}
    -- /**
    -- * ABSTRACT SPECIFICATION
    -- * ======================
    -- * abstract specification for an algorithm
    -- that
    -- generate an interlocking table from a given
    -- wellformed network layout
    -- */
    -- value
    --  mk_table_abs :
    --    I.L.NetworkLayout -> I. InterlockingTable
    --  mk_table_abs(n) as tb post
    --    -- a generated table is a well-formed
    -- one
    --    I.is_wf_rt(tb, n) /\
    --    -- and it's the largest
    --    (all tb′ : I. InterlockingTable :-
    --        rng tb′ <<= rng tb)

    value
      -- initial route with its source at s
      init_r : MbId → I.Route
      init_r (s) ≡ I.mk_Route(s, "", ⟨⟩, ⟨⟩, [], {}, {})

    value
      /**
      * GENERATE INTERLOCKING TABLE FROM A LAYOUT
      */
      -- generate the interlocking table for a given
      -- network layout

      mk_table : NetworkLayout′ → I.InterlockingTable
      mk_table(n) ≡
        let rs = gen_routes(n) in mark_conflicts(assign_id(rs, 1, n), n) end,

      -- are we collecting the overlap, i.e. already
      -- found the destination signal
```

```
is_collecting_ovs : I.Route × NetworkLayout′ → Bool
is_collecting_ovs(r, n) ≡ I.L.m_exists(I.dest(r), n),

−− last collected section in the route
last_collected_sec : I.Route × NetworkLayout′ → SecId
last_collected_sec(r, n) ≡
  let
    ps = I.path(r),
    ovs = I.overlap(r),
    src = I.L.get_maker_board(I.source(r), n),
    fst = I.L.section(src)
  in
    if ps = ⟨⟩ ∧ ovs = ⟨⟩ then fst
    else if ovs ≠ ⟨⟩ then ovs(len ovs) else ps(len ps) end
    end
  end,

−− collecting route information on a linear
−− section
collect_route_on_linear :
  SecId × I.Route × NetworkLayout′ ~→ I.Route*
collect_route_on_linear(i, r, n) ≡
  let
    isOvs = is_collecting_ovs(r, n),
    prev = last_collected_sec(r, n),
    l = I.L.get_linear(i, n),
    nbs = I.L.neighbors(l),
    sigs = I.L.signals(i, n),
    d = (−I.L.get_l_end_by_nb_id(i, prev, n)),
    has_sig = d ∈ sigs,
    dsig = if isOvs ∨ ¬ has_sig then I.dest(r) else sigs(d) end,
    opposing_sigs = rng (sigs \ {d}),
    new_ovs = if isOvs then I.overlap(r) ⌢ ⟨i⟩ else I.overlap(r) end,
    new_path = if isOvs then I.path(r) else I.path(r) ⌢ ⟨i⟩ end,
    ovs_len = sum(⟨I.L.get_length(o, n) | o in new_ovs⟩)
  in
    if
      d ∉ nbs −−reached the border
    then
      if isOvs ∨ has_sig
      then
        ⟨I.mk_Route(
            I.source(r), dsig, new_path, new_ovs, I.points(r),
            I.signals(r) ∪ opposing_sigs, I. conflicts (r))⟩
      else ⟨⟩
      end
    else
```

```
        let
          j = nbs(d),
          safety_dx =
            if has_sig ∨ isOvs
            then I.L.distance(I.L.get_maker_board(dsig, n))
            else 0
            end,
          done =
            (isOvs ∧ (ovs_len + safety_dx) ≥ I.MIN_SAFETY_DISTANCE) ∨
            (has_sig ∧ safety_dx ≥ I.MIN_SAFETY_DISTANCE),
          front_prot =
            if done then I.find_protection(i, j, n) else I.empty end,
          new_r =
            I.mk_Route(
                I.source(r), dsig, new_path, new_ovs,
                I.points(r) ∪ I.points(front_prot),
                I.signals(r) ∪ opposing_sigs ∪
              I.signals(front_prot), I.conflicts(r))
        in
          if done then ⟨new_r⟩ else collect_route(j, new_r, n) end
        end
      end
    end
  pre I.L.l_exists(i, n),

  -- collecting route information on a point
  -- section
  collect_route_on_point :
    SecId × I.Route × NetworkLayout′ ~→ I.Route*
  collect_route_on_point(i, r, n) ≡
    let
      isOvs = is_collecting_ovs(r, n),
      prev = last_collected_sec(r, n),
      p = I.L.get_point(i, n),
      nbs = I.L.neighbors(p),
      new_ovs = if isOvs then I.overlap(r) ⌢ ⟨i⟩ else I.overlap(r) end,
      new_path = if isOvs then I.path(r) else I.path(r) ⌢ ⟨i⟩ end,
      ovs_len = sum(⟨I.L.get_length(o, n) | o in new_ovs⟩),
      done = isOvs ∧ ovs_len ≥ I.MIN_SAFETY_DISTANCE,
      side = I.L.get_p_end_by_nb_id(i, prev, n)
    in
      case side of
        NB_STEM →
          let
            plus = I.L.plus(p),
            prot_plus = I.find_protection(i, plus, n),
            minus = I.L.minus(p),
```

```
    prot_minus = I.find_protection(i, minus, n)
  in
    if done
    then
      let
        done_s_r =
          I.mk_Route(
              I.source(r), I.dest(r), new_path, new_ovs,
              I.points(r) ∪ I.points(prot_plus) ∪
              I.points(prot_minus),
              I.signals(r) ∪ I.signals(prot_plus) ∪
              I.signals(prot_minus), I. conflicts (r))
      in
        ⟨done_s_r⟩
      end
    else
      let
        plus_r =
          I.mk_Route(
              I.source(r), I.dest(r), new_path, new_ovs,
              I.points(r) ∪ [i ↦ PLUS] ∪
           I.points(prot_minus),
              I.signals(r) ∪ I.signals(prot_minus),
              I. conflicts (r)),
        minus_r =
          I.mk_Route(
              I.source(r), I.dest(r), new_path, new_ovs,
              I.points(r) ∪ [i ↦ MINUS] ∪
          I.points(prot_plus),
              I.signals(r) ∪ I.signals(prot_plus),
              I. conflicts (r))
      in
        collect_route(plus, plus_r, n) ⌢
        collect_route(minus, minus_r, n)
      end
    end
  end,
_ →
  let
    pos_side = if side = NB_PLUS then PLUS else MINUS end,
    prot_side =
      if side = NB_PLUS then I.find_protection(i, I.L.minus(p), n)
      else I.find_protection(i, I.L.plus(p), n)
      end
  in
    if done
    then
```

```
            let
              prot_stem = I.find_protection(i, I.L.stem(p), n),
              done_r =
                I.mk_Route(
                    I.source(r), I.dest(r), new_path, new_ovs,
                    I.points(r) ∪ [i ↦ pos_side] ∪
                  I.points(prot_side) ∪ I.points(prot_stem),
                    I.signals(r) ∪ I.signals(prot_side) ∪
                    I.signals(prot_stem), I. conflicts (r))
            in
              ⟨done_r⟩
            end
          else
            let
              stem_r =
                I.mk_Route(
                    I.source(r), I.dest(r), new_path, new_ovs,
                    I.points(r) ∪ [i ↦ pos_side] ∪
                  I.points(prot_side),
                    I.signals(r) ∪ I.signals(prot_side),
                    I. conflicts (r))
            in
              collect_route(I.L.stem(p), stem_r, n)
            end
          end
        end
    end
  end
pre I.L.p_exists(i, n),

-- collecting the information for a route
collect_route : SecId × I.Route × NetworkLayout′ ~→ I.Route*
collect_route(i, r, n) ≡
  if I.L.l_exists (i, n) then collect_route_on_linear(i, r, n)
  else collect_route_on_point(i, r, n)
  end
pre I.L.s_exists(i, n),

-- for a given signal set, generate all routes
-- starting from a signal in the set
gen_route_m : MbId* × NetworkLayout′ ~→ I.Route*
gen_route_m(ms, n) ≡
 if ms = ⟨⟩ then ⟨⟩
  else
    let
      mi = hd ms,
      m = I.L.get_maker_board(mi, n),
```

```
        i = I.L.section(m),
        l = I.L.get_linear(i, n),
        nbs = I.L.neighbors(l),
        s = I.L.dir(m),
        lrs = gen_route_m(tl ms, n)
      in
        if
          -- if we do not reach the border yet
          s ∈ nbs
        then
          let j = nbs(s), r = init_r(mi) in
            collect_route(j, r, n) ⌢ lrs
          end
        else lrs
        end
      end
    end
  pre (∀m : MbId • m ∈ ms ⇒ I.L.m_exists(m, n)),

  -- generate the primary set of all elementary
  -- routes for a given network
  -- ==============================================
  -- Remarks: The sorting doesn't have anything
  -- todo with the generation, it just makes
  -- sure the route are assigned ids in order
  -- of their source signals

  gen_routes : NetworkLayout′ → I.Route*
  gen_routes(n) ≡
    let sigs = sort( to_list (dom I.L.marker_boards(n))) in
      gen_route_m(sigs, n)
    end,

  -- make an alternative route by replacing the
  -- set of protecting point rp with protecting
  -- signals
  mk_alt :
    I.Route × SecId-set × (SecId →m MbId-set) × NetworkLayout′ ~→
      I.Route
  mk_alt(r, rp, sigmap, n) ≡
    let
      new_sigs = flatten(rng (sigmap / rp)),
      sigs = I.signals(r) ∪ new_sigs,
      ps = I.points(r) \ rp
    in
      I.mk_Route(
        I.source(r), I.dest(r), I.path(r), I.overlap(r), ps, sigs,
```

```
        I. conflicts (r))
  end
pre I.is_wf_r(r, n),

-- make an alternative route by replacing a
-- set from the powerset of protecting point
-- ss with protecting signals
mk_alternatives :
  I.Route × (SecId-set)-set × (SecId ~m~> MbId-set) × NetworkLayout′ ~→
    I.Route*
mk_alternatives(r, ss, sigmap, n) ≡
  if ss = {} then ⟨⟩
  else
    let s = hd ss, rs = mk_alternatives(r, ss \ {s}, sigmap, n) in
      ⟨mk_alt(r, s, sigmap, n)⟩ ⌢ rs
    end
  end
pre
  I.is_wf_r(r, n) ∧
  let ps = I.points(r) in
    (∀s : SecId-set •
       s ∈ ss ⇒
         (∀i : SecId • i ∈ s ⇒ i ∈ ps ∧ i ∈ sigmap))
  end,

-- Find all alternative routes of a given route
-- by replacing a subset of the set of protecting
-- points of the route with protecting signals

mk_alt_routes : I.Route × NetworkLayout′ → I.Route*
mk_alt_routes(r, n) ≡
  let
    ps = I.points(r),
    po = elems I.path(r) ∪ elems I.overlap(r),
    reps_init =
      [i ↦ I.find_replacing_signals(i, r, n) |
       i : SecId • i ∈ ps \ po],
    sigmap =
      [i ↦ reps_init(i) |
       i : SecId • i ∈ reps_init ∧ reps_init(i) ≠ {}],
    pws = powerset(dom sigmap)
  in
    mk_alternatives(r, pws, sigmap, n)
  end
pre I.is_wf_r(r, n),

-- assign id for the routes in the set, and
```

```
-- transform the set to a map
assign_id : I.Route* × Nat × NetworkLayout′ → I.InterlockingTable
assign_id(rs, c, n) ≡
  if rs = ⟨⟩ then []
  else
    let
      r = hd rs,
      id = mk_major_id(c),
      alts = mk_alt_routes(r, n),
      m =
        if alts = ⟨r⟩ then [id ↦ r] else mk_minor_map(alts, id, 1) end
    in
      assign_id(tl rs, c + 1, n) ∪ m
    end
  end,

-- make a map of alternative routes with minor
-- ids
mk_minor_map : I.Route* × RouteId × Nat → I.InterlockingTable
mk_minor_map(rs, major, c) ≡
if rs = ⟨⟩ then []
  else
    let r = hd rs, id = mk_minor_id(major, c) in
      [id ↦ r] ∪ mk_minor_map(tl rs, major, c + 1)
    end
  end,

-- mark pairs of routes that are physically
-- in conflict as conflicting routes
mark_conflicts :
  I.InterlockingTable × NetworkLayout′ → I.InterlockingTable
mark_conflicts(rt, n) ≡
  let ids = dom rt in
    [i ↦
     let
       r = rt(i),
       cs =
         {j |
          j : RouteId •
            j ∈ ids \ {i} ∧
            I.are_physically_in_conflict(r, rt(j), n)}
     in
       I.mk_Route(
          I.source(r), I.dest(r), I.path(r), I.overlap(r), I.points(r),
          I.signals(r), I.conflicts(r) ∪ cs)
     end | i : RouteId • i ∈ ids]
  end
```

end

APPENDIX B

Interlocking Dynamic Language – IDL

This appendix presents some extra information in order to facilitate the specification of IDL in Chapter 5. Section B.1 shows the complete BNF grammar of IDL. Section B.2 to Section B.3 explain the meaning of some operators, domain functions in IDL, respectively.

B.1 BNF Grammar

⟨*specification*⟩ ::= **kripke** ⟨*ident*⟩ ⟨*decl-string*⟩ **end**

⟨*decl*⟩ ::= ⟨*encoding-decl*⟩
| ⟨*macro-decl*⟩
| ⟨*initial-decl*⟩
| ⟨*module-decl*⟩
| ⟨*transrel-decl*⟩
| ⟨*invariant-decl*⟩
| ⟨*test-obj-decl*⟩

⟨*encoding-decl*⟩ ::= **encoding** ⟨*encoding-list*⟩

⟨*encoding*⟩ ::= ⟨*elem-type*⟩ **::** ⟨*variable-list*⟩

⟨*variable*⟩ ::= ⟨*symbol*⟩ → **[** ⟨*sym-type*⟩ **,** ⟨*target-type*⟩ **,** ⟨*ival*⟩ **,** ⟨*low*⟩ **,** ⟨*high*⟩ **]**

⟨*target-type*⟩ ::= **"** ⟨*primitive-type*⟩ **"**

⟨*primitive-type*⟩ ::= int | unsigned int | long | unsigned long | long long
| unsigned long long | float | double | clock

⟨*ival*⟩ ::= ⟨*literal*⟩

⟨*low*⟩ ::= ⟨*literal*⟩

⟨*high*⟩ ::= ⟨*literal*⟩

⟨*macro-decl*⟩ ::= **macro** ⟨*macro-list*⟩

⟨*macro*⟩ ::= **def** ⟨*ident*⟩ **(** ⟨*[ident-list]*⟩ **) =** ⟨*expr*⟩
| **def** ⟨*ident*⟩ **=** ⟨*expr*⟩

⟨*initial-decl*⟩ ::= **init** ⟨*invariant-list*⟩

⟨*module-decl*⟩ ::= **module** ⟨*ident*⟩ ⟨*transrel*⟩
| **module** ⟨*ident*⟩ **(** ⟨*[ident-list]*⟩ **)** ⟨*transrel*⟩

⟨*invariant-decl*⟩ ::= **invariant** ⟨*invariant-list*⟩

⟨*invariant*⟩ ::= **[** ⟨*ident*⟩ **]** ⟨*simple-expr*⟩

⟨*transrel-decl*⟩ ::= **transrel** ⟨*transrel*⟩

⟨*transrel*⟩ ::= **[** ⟨*ident*⟩ **]** ⟨*simple-expr*⟩ $\longrightarrow$ ⟨*next-expr*⟩
| **[** ⟨*ident*⟩ **]**
| **[** ⟨*ident*⟩ **(** ⟨*[expr-list]*⟩ **)]**
| ⟨*transrel*⟩ [=] ⟨*transrel*⟩
| ⟨*transrel*⟩ [>] ⟨*transrel*⟩

⟨*test-obj-decl*⟩ ::= **test_obj** ⟨*test-obj-list*⟩

⟨*test-obj*⟩ ::= **[** ⟨*ident*⟩ **]** ⟨*ltl-formula*⟩
| **(** [=] ⟨*ident*⟩ **:** ⟨*elem-type*⟩ • ⟨*test-obj*⟩ **)**

⟨*ltl-formula*⟩ ::= **[** ⟨*simple-expr*⟩ **]**
| **G** ⟨*ltl-formula*⟩
| **X** ⟨*ltl-formula*⟩
| **F** ⟨*ltl-formula*⟩
| $\neg$ ⟨*ltl-formula*⟩
| **E** ⟨*ident*⟩ **:** ⟨*ltl-formula*⟩
| ⟨*ltl-formula*⟩ **U** ⟨*ltl-formula*⟩
| ⟨*ltl-formula*⟩ $\wedge$ ⟨*ltl-formula*⟩
| ⟨*ltl-formula*⟩ $\vee$ ⟨*ltl-formula*⟩
| ⟨*ltl-formula*⟩ $\Rightarrow$ ⟨*ltl-formula*⟩

⟨*simple-expr*⟩ ::= ⟨*expr*⟩

⟨*next-expr*⟩ ::= ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*ident*⟩
| ⟨*literal*⟩
| ⟨*elem-type*⟩
| ⟨*symbol-expr*⟩
| ⟨*uop*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*bop*⟩ ⟨*expr*⟩
| ⟨*macro-ex-expr*⟩
| ⟨*domain-expr*⟩
| ⟨*quantified-expr*⟩
| ⟨*if-then-else-expr*⟩
| ⟨*case-expr*⟩
| ⟨*let-expr*⟩
| ⟨*index-expr*⟩

⟨*symbol-expr*⟩ ::= ⟨*expr*⟩ **.** ⟨*ident*⟩ ⟨*version*⟩

⟨*macro-ex-expr*⟩ ::= ⟨*ident*⟩ **(** ⟨*[expr-list]*⟩ **)**

⟨*domain-expr*⟩ ::= ⟨*domain-func*⟩ **(** ⟨*expr-list*⟩ **)**
| ⟨*domain-uop*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*domain-bop*⟩ ⟨*expr*⟩

⟨*quantified-expr*⟩ ::= **(** ⟨*quan-op*⟩ ⟨*ident*⟩ **:** ⟨*elem-type*⟩ **•** ⟨*expr*⟩ **)**

⟨*quan-op*⟩ ::= ∀ | ∃ | ∃**!**

⟨*if-then-else-expr*⟩ ::= **if** ⟨*expr*⟩ **then** ⟨*expr*⟩ **else** ⟨*expr*⟩ **end**
| ⟨*expr*⟩ **?** ⟨*expr*⟩ **:** ⟨*expr*⟩

⟨*case-expr*⟩ ::= **case** ⟨*expr*⟩ **of** ⟨*case-branch-list*⟩ **end**
| **case** ⟨*expr*⟩ **of** ⟨*case-default*⟩ **end**
| **case** ⟨*expr*⟩ **of** ⟨*case-branch-list*⟩ **,** ⟨*case-default*⟩ **end**

⟨*case-branch*⟩ ::= ⟨*expr*⟩ → ⟨*expr*⟩

⟨*case-default*⟩ ::= ⟨*wildcard*⟩ → ⟨*expr*⟩

⟨*let-expr*⟩ ::= **let** ⟨*assign-list*⟩ **in** ⟨*expr*⟩ **end**

⟨*assign*⟩ ::= ⟨*ident*⟩ **=** ⟨*expr*⟩

⟨*index-expr*⟩ ::= ⟨*expr*⟩ **[** ⟨*expr*⟩ **]**
| ⟨*expr*⟩ **[** ⟨*expr*⟩ **:** ⟨*expr*⟩ **]**

⟨*elem-type*⟩ ::= **Linear** | **Point** | **Section** | **Signal** | **Route**

⟨*sym-type*⟩ ::= **INPUT** | **LOCAL** | **OUTPUT**

⟨*uop*⟩ ::= ¬

⟨*bop*⟩ ::= ≤ | < | > | ≥ | = | ≠
| ∧ | ∨ | ⊕ | ⇒
| + | − | ∗ | / | %
| & | | | ≪ | ≫

⟨*domain-func*⟩ ::= **down** | **up** | **down_sig** | **up_sig**
| **stem** | **plus** | **minus**
| **dir** | **track**
| **src** | **dst** | **first** | **last**
| **path** | **overlap** | **points** | **signals** | **conflicts**
| **prev** | **next** | **prevs** | **nexts** | **req**
| **conn_end**
| **entry** | **exit**

⟨*domain-uop*⟩ ::= **elems** | **hd** | **tl** | **dom** | **rng** | **len**

⟨*domain-bop*⟩ ::= ∈ | ∪ | ∩ | \

⟨*version*⟩ ::= ′ | ⟨*empty*⟩

⟨*wildcard*⟩ ::= _

⟨*ident*⟩ ::= [_a-zA-Z][_a-zA-Z0-9]*

⟨*literal*⟩ ::= [0-9]+ | 0b[01]+ | 0x[0-9a-fA-F]+

B.2 Operators and Their Meaning

The meaning of a number of selected operators are explained in the following.

$v\prime$	value of v in the next state
$o.VAR$	variable VAR of the object o
$[=]$	non-deterministic choice
$[>]$	prioritized choice
$\gg$	arithmetic shift right
$\ll$	arithmetic shift left
ˆ	bitwise xor
&	bitwise and
|	bitwise or
$\wedge$	logical and
$\vee$	logical or
$\oplus$	logical xor
$\Rightarrow$	logical implication
elems	return the set of the items in a list
hd	return the head item of a list
tl	return the tail of a list
dom	return the set of domain items of a map
rng	return the set of value items of a map
$ls[i]$	return the ith item of the list ls (0-index)
$ls[i:j]$	return the items in the range i^{th} to j^{th} of list ls
$m[k]$	return the value corresponding to the key k in the map m

B.3 Domain Functions

The meaning of domain functions are explained in the following. Each function explanation is a row in the table. (1) The first column shows the applications of IDL domain function. (2) The second column shows their semantics of by RSL function applications (of RSL functions defined on the abstract syntax for ICL) where σ is the interlocking configuration data under consideration. Note that the function name in this column is linked to its respective RSL function in Appendix A. One can go to the respective RSL function by following the hyper link. (3) The third column is the brief informal description of the function applications.

Functions on Linear Sections.

IDL Syntax	Meaning in RSL	Description
up(l)	`up`(l,σ)	the neighboring section at l's up end
down(l)	`down`(l,σ)	the neighboring section at l's down end
up_sig(l)	`up_sig`(l,σ)	the signal intended for the up direction
down_sig(l)	`down_sig`(l,σ)	the signal intended for the down direction

Functions on Points.

IDL Syntax	Meaning in RSL	Description
stem(p)	`stem`(p,σ)	the neighboring section at p's stem end
plus(p)	`plus`(p,σ)	the neighboring section at p's plus end
minus(p)	`minus`(p,σ)	the neighboring section at p's minus end

Functions on Sections.

IDL Syntax	Meaning in RSL	Description
conn_end(e,n)	`conn_end`(e,n,σ)	returns the end of e that is connected to n

Functions on Signals.

IDL Syntax	Meaning in RSL	Description
dir(s)	`dir`(s,σ)	the direction s is intended for
track(s)	`track`(s,σ)	the section where s is installed along

Functions on Routes.

IDL Syntax	Meaning in RSL	Description
src(r)	`src`(r,σ)	the source signal of the route r
dst(r)	`dst`(r,σ)	the destination signal of the route r
path(r)	`path`(r,σ)	the list of sections in r's path
overlap(r)	`overlap`(r,σ)	the list of sections in r's overlap
points(r)	`points`(r,σ)	the map of points used by r to their required positions
signals(r)	`signals`(r,σ)	the set of r's protecting signals
conflicts(r)	`conflicts`(r,σ)	the set of r's conflicting routes
first(r)	`first`(r,σ)	the first section of r's path
last(r)	`last`(r,σ)	the last section of r's path
entry(r,e)	`entry`(r,e,σ)	the end of e from which r enters e
exit(r,e)	`exit`(r,e,σ)	the end of e to which r exits e
next(r,e)	`next`(r,e,σ)	the next element of e in route r
prev(r,e)	`prev`(r,e,σ)	the previous element of e in route r
nexts(r,e)	`nexts`(r,e,σ)	the next elements of e in route r
prevs(r,e)	`prevs`(r,e,σ)	the previous elements of e in route r
req(r,p)	`req`(r,p,σ)	the position of p as required by r

APPENDIX C

Cases for Experiments with Our Toolchain

This appendix lists the network layout of made-up cases that were used for the experiments with our toolchain as described in Section 6.5. The cases extracted from the Early Deployment Line (EDL) of the Danish Signalling Programme are not listed here due to the non-disclosure agreement (NDA) with our industrial partners.

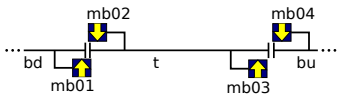


Figure C.1: Tiny

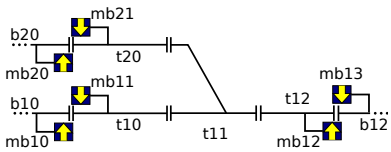


Figure C.2: Toy

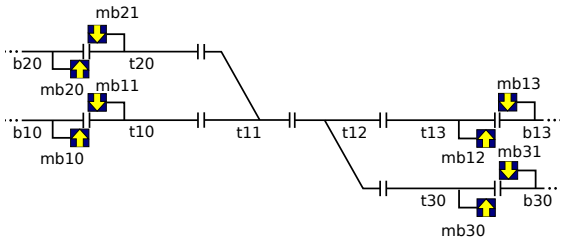


Figure C.3: Twist

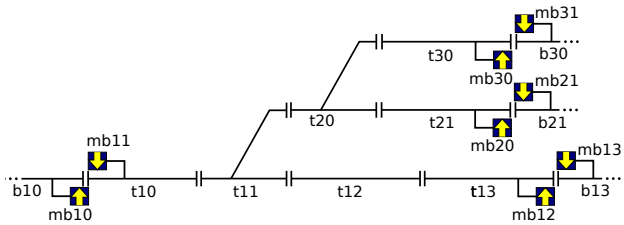


Figure C.4: Fork

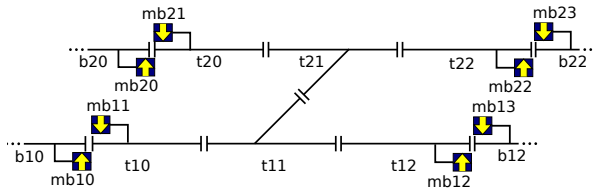


Figure C.5: Cross

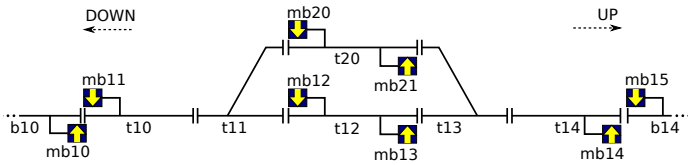


Figure C.6: Mini

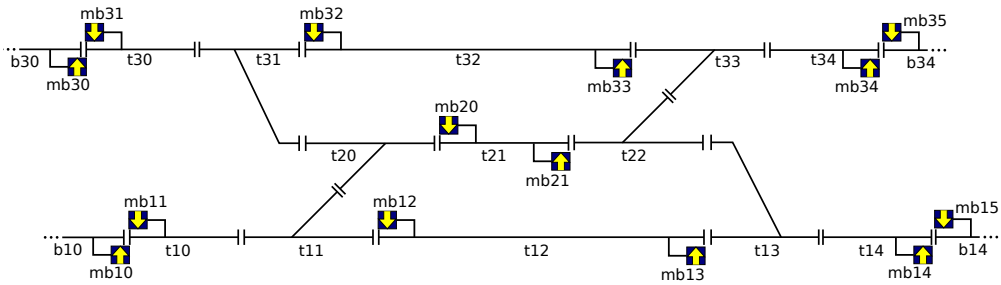


Figure C.7: Lyngby

APPENDIX D

Generic Applications of the Danish Interlocking Systems

This appendix lists the full specification of the generic applications (behavioural model, properties, and test objectives) for the forthcoming Danish interlocking systems in IDL as described in Chapter 6. First, Section D.1 explains briefly some selected macros that are used in the specification. The full specification in IDL is then given in Section D.2.

D.1 Selected Macros

Some selected macros and their objectives are explained briefly in the following. All macros used for our generic applications for the Danish interlocking systems are defined in Section D.2.

- vacant(e) : whether *e* is vacant
- H__(hto) : get the H bit of the occupancy status variable *hto*
- _T_(hto) : get the T bit of the occupancy status variable *hto*
- __O(hto) : get the O bit of the occupancy status variable *hto*
- route_entry_hto(e,r,v) : get the occupancy status variable of *e* in the direction of *r* at version *v*
- neighbor_hto(e,n,v) : get the occupancy status variable of *e* in the direction coming from *n*
- hto(e,x,v) : a generic version of two above macros
- is_boundary_sec_down(e) : whether *e* is a boundary section in the down direction
- is_boundary_sec_up(e) : whether *e* is a boundary section in the up direction
- occupied_with_head(hto) : whether a section is occupied with the head of a train in it

- occupied_without_tail(hto) : whether a section is occupied without the tail of a train in it
- occupied_without_head(hto) : whether a section is occupied without the head of a train in it
- occupied_with_only_tail(hto) : whether a section is occupied with only the tail of a train in it
- head_enters(hto,hto′) : toggle the H and O bits of the occupancy variable *hto* to model the head of a train entering a section
- head_leaves(hto,hto′) : toggle the H bit of the occupancy variable *hto* to model the head of a train leaving a section
- tail_enters (hto,hto′) : toggle the T bit of the occupancy variable *hto* to model the tail of a train entering a section
- tail_leaves (hto,hto′) : set the occupancy variable *hto* to 0 to model the tail of a train leaving a section
- can_turn_around_at(e) : whether trains can change direction at *e*

D.2 Full Specification in IDL

```
/* ==========================================================
 *      File : $Name: dk_interlocking .kr $
 *  Created : $Date: 2014-04-10 12:22:23 $
 *    Author: $Author: Linh H. Vu<lvho@dtu.dk> $
 * ==========================================================
 * Description : Generic behavioral model of the forthcoming
 * Danish interlocking systems
 * ==========================================================
 * Route control states :
 * ====================
 * 0 : NOCMD
 * 1 : DISPATCH
 * 2 : CANCEL
 * Possible transitions :
 * NOCMD -> DISPATCH/CANCEL
 * DISPATCH -> NOCMD/CANCEL
 * CANCEL -> NOCMD
 * Route states :
 * ============
 * 0 : FREE
 * 1 : MARKED
 * 2 : ALLOCATING
```

```
 * 3 : LOCKED
 * 4 : OCCUPIED
 * Signals aspect :
 * ===============
 * 0 : CLOSED
 * 1 : OPEN
 * Point positions :
 * ================
 * 0 : PLUS
 * 1 : MINUS
 * 2 : INTER
 * Element modes:
 * ==============
 * 0 : AVAIL
 * 1 : EXLCK
 * 2 : USED
 */
kripke dk_interlocking

encoding

/**
 * D2U: occupancy status for direction from down to up
 * U2D: occupancy status for direction from up to down
 * MODE: current mode of the element
 * PREV: whether the previous element in the same route has been
 * released
 */
Linear::
  D2U → [INPUT,"unsigned int",0,0,7]
  U2D → [INPUT,"unsigned int",0,0,7]
  MODE → [LOCAL,"unsigned int",0,0,2]
  PREV → [LOCAL,"unsigned int",0,0,1],
/**
 * S2PM: occupancy status for direction from stem to plus/minus
 * P2S: occupancy status for direction from plus to stem
 * M2S: occupancy status for direction from minus to stem
 * CMD: commanded position of the point
 * POS: actual position of the point
 * MODE: current mode of the element
 * PREV: whether the previous element in the same route has been
 * released
 */
Point::
  S2PM → [INPUT,"unsigned int",0,0,7]
  P2S → [INPUT,"unsigned int",0,0,7]
  M2S → [INPUT,"unsigned int",0,0,7]
```

```
  CMD → [OUTPUT,"unsigned int",0,0,1]
  POS → [INPUT,"unsigned int",0,0,2]
  MODE → [LOCAL,"unsigned int",0,0,2]
  PREV → [LOCAL,"unsigned int",0,0,1],
/**
 * ACT: actual aspect of the signal
 * CMD: commanded aspect of the signal
 */
Signal::
  ACT → [INPUT,"unsigned int",0,0,1]
  CMD → [OUTPUT,"unsigned int",0,0,1],
/**
 * CTRL: control command for the route
 * MODE: current mode of the route ( internal )
 * DSPL: current mode of the route (shown externally )
 */
Route::
  CTRL → [INPUT,"unsigned int",0,0,2]
  MODE → [LOCAL,"unsigned int",0,0,4]
  DSPL → [OUTPUT,"unsigned int",0,0,4]

// init
// /**
//  * Linear sections are vacant and available , their PREV variables are unset
//  */
// [ initial_state_linear ]
// ( all l : Linear :- vacant(l) ∧ l.MODE = AVAIL ∧ l.PREV = PENDING),
// /**
//  * Points are vacant and available , their PREV variables are unset, and their
//  * positions are PLUS
//  */
// [ initial_state_point ]
// ( all p : Linear :-
//  (vacant(p) ∧ p.MODE = AVAIL ∧ p.PREV = PENDING) ∧
//  (p.CMD = PLUS ∧ p.POS = PLUS)),
// /**
//  * Signals are CLOSED
//  */
// [ initial_state_signal ]
// ( all s : Signal :- s.CMD = CLOSED ∧ s.ACT = CLOSED),
// /**
//  * Routes are FREE and no pending commands
//  */
// [ initial_state_route ]
// ( all r : Route :- r.CTRL = NOCMD ∧ r.MODE = FREE ∧ r.DSPL = FREE)
/**
 * ========================================
```

```
 * TRANSITION RELATION OF THE WHOLE SYSTEM
 * ========================================
 */
transrel

[DP] [=] ([SUT] [>] [ET] [>] [TM])

/**
 * ==============================
 * ROUTE DISPATCHING TRANSITIONS
 * ==============================
 * i.e. communication with TMS, controlling route dispatching/canceling
 */

module DP

/**
 * DISPATCH ROUTES
 * ===============
 * if a route has not been dispatched, then it can be dispatched
 */
([=] r : Route •
 [ctrl_nocmd_to_dispatch] (r.CTRL = NOCMD ∧ r.DSPL = FREE) ⟶
 (r.CTRL′ = DISPATCH))
[=]

/**
 * DISPATCH ORDERED HAS BEEN SERVED
 * ================================
 */
([=] r : Route •
 [ctrl_dispatch_to_nocmd] (r.CTRL = DISPATCH ∧ r.DSPL ≠ FREE) ⟶
 (r.CTRL′ = NOCMD))
[=]

/**
 * CANCELING ROUTES
 * ================
 * a route can be canceled if it is not already been commanded to be
 * canceled
 */
([=] r : Route •
 [ctrl_to_cancel]
 (r.CTRL ≠ CANCEL ∧
  (r.DSPL = MARKED ∨ r.DSPL = ALLOCATING ∨ r.DSPL = LOCKED))
 ⟶ (r.CTRL′ = CANCEL))
[=]
```

```
/**
 * CANCELING ORDER HAS BEEN SERVED
 * ================================
 * finish route canceling
 */
([=] r : Route •
 [ctrl_cancel_to_nocmd]
 (r.CTRL = CANCEL ∧ r.DSPL = FREE ∧ src(r).ACT = CLOSED)
 ⟶ (r.CTRL′ = NOCMD))

/**
 * ========================
 * INTERLOCKING TRANSITIONS
 * ========================
 */

module SUT

/**
 * BEGIN: HANDLING ROUTE CANCELING
 * ================================
 */
/* a marked route can be canceled at anytime */
([=] r : Route •
 [cancel_marked_route] (r.CTRL = CANCEL ∧ r.MODE = MARKED) ⟶
 (r.MODE′ = FREE ∧ r.DSPL′ = FREE))
[=]

/* a route in allocating mode can be canceled at anytime */
([=] r : Route •
 [cancel_allocating_route]
 (r.CTRL = CANCEL ∧ r.MODE = ALLOCATING ∧
  /* no points are switching, if there is a point switching, we wait until the
   * next cycle when the point is already done switching and then cancel the
   * route */
  (∀p : Point • p ∈ points(r) ⇒ p.POS = req(r,p)))
 ⟶
 /* free the route */
 (r.MODE′ = FREE) ∧ (r.DSPL′ = FREE) ∧
 /* canceling the command to points, we don't canceling the protecting point
  * because it may be used by other routes */
 (∀p : Point • p ∈ path(r) ⇒ (p.CMD′ = p.POS)) ∧
 /* and unlock all sections in the route's path */
 (∀e : Section • e ∈ path(r) ⇒ (e.MODE′ = AVAIL)))
[=]
```

```
/* a locked route can only be canceled if they are not used yet we don't need to
 * check if any points are switching in here as when the route is in ALLOCATING
 * mode, because the route in LOCKED mode implies that all points have been
 * switched to a proper position as requested */
([=] r : Route •
 [cancel_locked_route]
 (r.CTRL = CANCEL ∧ r.MODE = LOCKED ∧
  /* can only be canceled if the route has not been used, i.e., all the
   * route's path and overlap are still vacant */
  (∀e : Section •
   e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e)))
 ⟶
 /* free the route */
 (r.MODE′ = FREE) ∧ (r.DSPL′ = FREE) ∧
 /* close the source signal */
 (src(r).CMD′ = CLOSED) ∧
 /* and unlock all sections in the route's path */
 (∀e : Section • e ∈ path(r) ⇒ (e.MODE′ = AVAIL)))
[=]

/**
 * END: HANDLING ROUTE CANCELING
 * ============================
 */
/**
 * ROUTE MARKING
 * =============
 */
([=] r : Route •
 [route_marking] (r.CTRL = DISPATCH ∧ r.MODE = FREE) ⟶
 (r.MODE′ = MARKED ∧ r.DSPL′ = MARKED))
[=]

/*
 * ROUTE ALLOCATING
 * ================
 */
([=] r : Route •
 [route_allocating]
 (r.MODE = MARKED) ∧
 /* none of the conflicting routes in ALLOCATING(2) or LOCKED(3) modes
  */
 (∀cr : Route •
  (cr ∈ conflicts(r)) ⇒ (cr.MODE ≠ ALLOCATING ∧ cr.MODE ≠ LOCKED)) ∧
 /* all detection sections in the path and overlap are vacant */
 (∀e : Section •
  e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e)) ∧
```

```
 /* all elements in route's path are in AVAIL mode */
 (∀e : Section • e ∈ path(r) ⇒ (e.MODE = AVAIL)) ∧
 /* all elements in route's overlap are not in USED mode */
 (∀e : Section • e ∈ overlap(r) ⇒ (e.MODE ≠ USED)) ∧
 /* all protecting points are in AVAIL mode,
  * or are already in the correct position */
 (∀e : Point •
  e ∈ (dom points(r) \ elems path(r)) ⇒
  (e.MODE = AVAIL ∨ e.POS = req(r,e)))
 ⟶
 (r.MODE′ = ALLOCATING) ∧ (r.DSPL′ = ALLOCATING) ∧
 /* command points */
 (∀p : Point • p ∈ points(r) ⇒ (p.CMD′ = req(r,p))) ∧
 /* command signals */
 (∀s : Signal • s ∈ signals(r) ⇒ (s.CMD′ = CLOSED)) ∧
 /* lock exclusively all elements in the route's path */
 (∀e : Section • e ∈ path(r) ⇒ (e.MODE′ = EXLCK)))
[=]

/**
 * ROUTE LOCKING
 * =============
 */
([=] r : Route •
 [route_lock]
 r.MODE = ALLOCATING ∧
 /* protecting signals' actual aspects are as required */
 (∀s : Signal • s ∈ signals(r) ⇒ (s.ACT = CLOSED)) ∧
 /* points' actual positions are as required */
 (∀p : Point • p ∈ points(r) ⇒ (p.POS = req(r,p))) ∧
 /* all detection sections in the path and overlap are vacant */
 (∀e : Section •
  e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e)) ∧
 /* all elements in the route's path are locked exclusively */
 (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK))
 ⟶ r.MODE′ = LOCKED ∧ r.DSPL′ = LOCKED ∧ src(r).CMD′ = OPEN)
[=]

/**
 * ROUTE IN USE + FIRST ELEMENT IN USE
 * ===================================
 */
([=] r : Route •
 [route_in_use]
 /* the first element of the route is occupied */
 let e = first(r) in
   r.MODE = LOCKED ∧ ¬vacant(e)
```

```
 end
 ⟶
 r.MODE′ = OCCUPIED ∧ r.DSPL′ = OCCUPIED ∧ src(r).CMD′ = CLOSED ∧
  first (r). MODE′ = USED)
[=]

/**
 * ELEMENT IN USE
 * ==============
 * Excluding the first element
 */
([=] r : Route •
 ([=] e : Section •
  [element_in_use]
  e ∈ path(r) ∧ e ≠ first (r) ∧ prev(r,e). MODE = USED ∧
  r.MODE = OCCUPIED ∧ e.MODE = EXLCK ∧ not vacant(e) ∧
  ((e ∈ Point) ⇒ e.POS = req(r,e)) ∧
  (e ≠ last(r) ⇒ next(r,e).MODE = EXLCK)
  ⟶ e.MODE′ = USED))
[=]

/**
 * SEQUENTIAL RELEASE OF ELEMENTS
 * ==============================
 * Except the last element
 */
([=] r : Route •
 ([=] e : Section •
  [sequential_release_e]
  e ∈ path(r) ∧ e ≠ last(r) ∧ r.MODE = OCCUPIED ∧
  e.MODE = USED ∧ vacant(e) ∧ (e ≠ first(r) ⇒ e.PREV = RELEASED) ∧
  let nx = next(r,e) in
    nx.PREV = PENDING ∧ nx.MODE = USED ∧ (_T_(hto(nx,r,0)) ≠ 0) ∧
    ((nx ∈ Point) ⇒ nx.POS = req(r,nx))
  end ∧ ((e ∈ Point) ⇒ e.POS = req(r,e))
  ⟶ e.MODE′ = FREE ∧ e.PREV′ = PENDING ∧ next(r,e).PREV′ = RELEASED))
[=]

/**
 * SEQUENTIAL RELEASE OF THE LAST ELEMENT + ROUTE
 * ==============================================
 * – It is the last element, release the route
 */
([=] r : Route •
 [sequential_release_last_elem]
 r.MODE = OCCUPIED ∧
 let e = last(r) in
```

```
   e.MODE = USED ∧ vacant(e) ∧ (e ≠ first(r) ⇒ e.PREV = RELEASED) ∧
   ((e ∈ Point) ⇒ e.POS = req(r,e))
 end
 ⟶
 last(r).MODE′ = AVAIL ∧ last(r).PREV′ = PENDING ∧ r.MODE′ = FREE ∧
 r.DSPL′ = FREE)
[=]

/**
 * RELEASE THE LAST ELEMENT AND THE ROUTE
 * ======================================
 */
([=] r : Route •
 [release_last_elem_pseudo_timer]
 r.MODE = OCCUPIED ∧
 let e = last(r) in
   e.MODE = USED ∧ hto(e,r,0) = 0b111 ∧ dst(r).ACT = CLOSED ∧
   (e ≠ first(r) ⇒ e.PREV = RELEASED)
 end
 ⟶
 last(r).MODE′ = AVAIL ∧ last(r).PREV′ = PENDING ∧ r.MODE′ = FREE ∧
 r.DSPL′ = FREE)

/**
 * ==========================
 * TRACK ELEMENT TRANSITIONS
 * ==========================
 */

module ET

/**
 * POINT SWITCHING
 * ===============
 */
/* Start switching */
([=] p : Point •
 [point_switch_1] p.POS ≠ p.CMD ∧ p.POS ≠ INTER ⟶ p.POS′ = INTER)
[=]

/* Move in the commanded position */
([=] p : Point • [point_switch_2] p.POS = INTER ⟶ p.POS′ = p.CMD)
[=]

/**
 * COMMUNICATE SIGNAL ASPECTS TO TRAINS
 * ====================================
```

```
 */
([=] s : Signal •
 [communicate_signal_aspect] s.ACT ≠ s.CMD ⟶ s.ACT′ = s.CMD)

/**
 * ==========================
 * TRAIN MOVEMENT TRANSITIONS
 * ==========================
 */

module TM

/**
 * HEAD MOVEMENT ON LINEAR SECTIONS
 * ================================
 */
/* from down to up */
([=] l : Linear •
 [head_movement_linear_up]
 up(l) ∧ ¬is_boundary_sec_up(up(l)) ∧ ¬is_boundary_sec_down(l) ∧
 occupied_with_head(l.D2U) ∧ (¬up_sig(l) ∨ up_sig(l).ACT = OPEN)
 ⟶ head_leaves(l.D2U,l.D2U′) ∧ head_enters_next(up(l),l))
[=]

/* from up to down */
([=] l : Linear •
 [head_movement_linear_down]
 down(l) ∧ ¬is_boundary_sec_down(down(l)) ∧ ¬is_boundary_sec_up(l) ∧
 occupied_with_head(l.U2D) ∧ (¬down_sig(l) ∨ down_sig(l).ACT = OPEN)
 ⟶ head_leaves(l.U2D,l.U2D′) ∧ head_enters_next(down(l),l))
[=]

/**
 * TAIL MOVEMENT ON LINEAR SECTIONS
 * ================================
 */
/* from down to up */
([=] l : Linear •
 [tail_movement_linear_up]
 up(l) ∧ ¬is_boundary_sec_up(up(l)) ∧ ¬is_boundary_sec_down(l) ∧
 occupied_with_only_tail(l.D2U)
 ⟶ tail_leaves(l.D2U,l.D2U′) ∧ tail_enters_next(up(l),l))
[=]

/* from up to down */
([=] l : Linear •
 [tail_movement_linear_down]
```

```
 down(l) ∧ ¬is_boundary_sec_down(down(l)) ∧ ¬is_boundary_sec_up(l) ∧
 occupied_with_only_tail(l.U2D)
 ⟶ tail_leaves(l.U2D,l.U2D′) ∧ tail_enters_next(down(l),l))
[=]

/**
 * HEAD MOVEMENT ON POINT SECTIONS
 * ================================
 */
/* from stem toward plus */
([=] p : Point •
 [head_movement_point_stem_to_plus] occupied_with_head(p.S2PM) ∧ p.POS = PLUS
 ⟶ head_leaves(p.S2PM,p.S2PM′) ∧ head_enters_next(plus(p),p))
[=]

/* from stem toward minus */
([=] p : Point •
 [head_movement_point_stem_to_minus]
 occupied_with_head(p.S2PM) ∧ p.POS = MINUS
 ⟶ head_leaves(p.S2PM,p.S2PM′) ∧ head_enters_next(minus(p),p))
[=]

/* from plus toward stem */
([=] p : Point •
 [head_movement_point_plus_to_stem] occupied_with_head(p.P2S) ∧ p.POS = PLUS
 ⟶ head_leaves(p.P2S,p.P2S′) ∧ head_enters_next(stem(p),p))
[=]

/* from minus toward stem */
([=] p : Point •
 [head_movement_point_minus_to_stem]
 occupied_with_head(p.M2S) ∧ p.POS = MINUS
 ⟶ head_leaves(p.M2S,p.M2S′) ∧ head_enters_next(stem(p),p))
[=]

/**
 * TAIL MOVEMENT ON POINT SECTIONS
 * ================================
 */
/* from stem toward plus */
([=] p : Point •
 [tail_movement_point_stem_to_plus]
 occupied_with_only_tail(p.S2PM) ∧ p.POS = PLUS
 ⟶ tail_leaves(p.S2PM,p.S2PM′) ∧ tail_enters_next(plus(p),p))
[=]

/* from stem toward minus */
```

```
([=] p : Point •
 [tail_movement_point_stem_to_minus]
 occupied_with_only_tail(p.S2PM) ∧ p.POS = MINUS
 ⟶ tail_leaves(p.S2PM,p.S2PM′) ∧ tail_enters_next(minus(p),p))
[=]

/* from plus toward stem */
([=] p : Point •
 [tail_movement_point_plus_to_stem]
 occupied_with_only_tail(p.P2S) ∧ p.POS = PLUS
 ⟶ tail_leaves(p.P2S,p.P2S′) ∧ tail_enters_next(stem(p),p))
[=]

/* from minus toward stem */
([=] p : Point •
 [tail_movement_point_minus_to_stem]
 occupied_with_only_tail(p.M2S) ∧ p.POS = MINUS
 ⟶ tail_leaves(p.M2S,p.M2S) ∧ tail_enters_next(stem(p),p))
[=]

/**
 * CHANGE DIRECTION (ONLY ON LINEAR SECTIONS)
 * ==========================================
 */
/* change direction from going upward to downward */
([=] l : Linear •
 [change_direction_up_to_down]
 down_sig(l) ∧ up_sig(l) ∧ l.D2U = 0b111 ∧ up_sig(l).ACT = CLOSED
 ⟶ swap_up_down_vars(l))
[=]

/* change direction from going downward to upward */
([=] l : Linear •
 [change_direction_down_to_up]
 down_sig(l) ∧ up_sig(l) ∧ l.U2D = 0b111 ∧ down_sig(l).ACT = CLOSED
 ⟶ swap_up_down_vars(l))
[=]

/**
 * ENTER INTERLOCKED AREA (ONLY ON LINEAR SECTIONS)
 * ================================================
 */
/**
 * from the down side if down is the border of interlocked area
 * <-- down      up -->
 * ////------|--||------
 *         l    []     up(l)
```

```
 *                sig
 */
/* head enters */
([=] l : Linear •
 [enter_interlocked_area_head_from_down]
 is_boundary_sec_down(l) ∧ up_sig(l).ACT = OPEN
 ⟶
 let e = up(l) in
   head_enters(e.D2U,e.D2U′)
 end)
[=]

/* tail enters */
([=] l : Linear •
 [enter_interlocked_area_tail_from_down]
 is_boundary_sec_down(l) ∧
 let e = up(l) in
   occupied_without_tail(e.D2U)
 end
 ⟶
 let e = up(l) in
    tail_enters (e.D2U,e.D2U′)
 end)
[=]

/**
 * from the up if the up is the border of interlocked area
 * <-- down        up -->
 *              sig
 * down(l)     []    l
 * ---------||--|------////
 */
/* head enters */
([=] l : Linear •
 [enter_interlocked_area_head_from_up]
 is_boundary_sec_up(l) ∧ down_sig(l).ACT = OPEN
 ⟶
 let e = down(l) in
   head_enters(e.U2D,e.U2D′)
 end)
[=]

/* tail enters */
([=] l : Linear •
 [enter_interlocked_area_tail_from_up]
 is_boundary_sec_up(l) ∧
 let e = down(l) in
```

```
   occupied_without_tail(e.U2D)
 end
 ⟶
 let e = down(l) in
   tail_enters (e.U2D,e.U2D′)
 end)
[=]

/**
 * LEAVE INTERLOCKED AREA (ONLY ON LINEAR SECTIONS)
 * ================================================
 * We don't care the aspect of the exit signal since the exiting signal
 * is not controled by this interlocking
 */
/**
 * leave downward if the down is border
 * <-- down     up -->
 * ////---------||------
 *        l        up(l)
 */
/* head leaves */
([=] l : Linear •
 [leave_interlocked_area_head_to_down]
 is_boundary_sec_down(l) ∧
 let e = up(l) in
   occupied_with_head(e.U2D)
 end
 ⟶
 let e = up(l) in
   head_leaves(e.U2D,e.U2D′)
 end)
[=]

/* tail leaves */
([=] l : Linear •
 [leave_interlocked_area_tail_to_down]
 is_boundary_sec_down(l) ∧
 let e = up(l) in
   occupied_with_only_tail(e.U2D)
 end
 ⟶
 let e = up(l) in
   tail_leaves (e.U2D,e.U2D′)
 end)
[=]

/**
```

```
 * leave upward if the up is border
 * <-- down      up -->
 * down(l)         l
 * ---------||-------////
 */
/* head leaves */
([=] l : Linear •
 [leave_interlocked_area_head_to_up]
 is_boundary_sec_up(l) ∧
 let e = down(l) in
   occupied_with_head(e.D2U)
 end
 ⟶
 let e = down(l) in
   head_leaves(e.D2U,e.D2U′)
 end)
[=]

/* tail leaves */
([=] l : Linear •
 [leave_interlocked_area_tail_to_up]
 is_boundary_sec_up(l) ∧
 let e = down(l) in
   occupied_with_only_tail(e.D2U)
 end
 ⟶
 let e = down(l) in
   tail_leaves (e.D2U,e.D2U′)
 end)

invariant

/**
 * ============================
 * HIGH-LEVEL SAFETY PROPERTIES
 * ============================
 */
[no_head_to_head_collisions_linear]
(∀l : Linear • (¬is_boundary_sec(l)) ⇒ (l.D2U * l.U2D = 0)),

[no_head_to_tail_collisions_linear]
(∀l : Linear •
 (¬is_boundary_sec(l)) ⇒
 (l.D2U * (1 − (l.D2U & 1)) + l.U2D * (1 − (l.U2D & 1)) = 0)),

[no_head_to_head_collisions_point]
(∀p : Point • p.M2S * p.S2PM + p.P2S * p.S2PM + p.P2S * p.M2S = 0),
```

```
[no_head_to_tail_collisions_point]
(∀p : Point •
 p.S2PM * (1 − (p.S2PM & 1)) + p.P2S * (1 − (p.P2S & 1)) +
 p.M2S * (1 − (p.M2S & 1)) = 0),

[no_derailments]
(∀p : Point •
 p.POS * p.P2S + (1 − (p.POS & 1)) * p.M2S + (p.POS ≫ 1) * p.S2PM = 0),

/**
 *
 * ========================
 * STRENGTHENING INVARIANTS
 * ========================
 */
/**
 * TRAIN INTEGRITY CONDITIONS
 * ==========================
 */
/* linear section , up direction */
[train_integrity_linear_up]
(∀l : Linear •
 (up(l) ∧ ¬is_boundary_sec_up(up(l)) ∧ ¬is_boundary_sec_down(l)) ⇒
 ((occupied_without_head(l.D2U) ⇒ occupied_without_tail(hto(up(l),l))) ∧
  (occupied_without_tail(hto(up(l),l)) ⇒ occupied_without_head(l.D2U)))),

/* linear section , down direction */
[train_integrity_linear_down]
(∀l : Linear •
 (down(l) ∧ ¬is_boundary_sec_down(down(l)) ∧ ¬is_boundary_sec_up(l)) ⇒
 ((occupied_without_head(l.U2D) ⇒ occupied_without_tail(hto(down(l),l))) ∧
  (occupied_without_tail(hto(down(l),l)) ⇒ occupied_without_head(l.U2D)))),

/* point section , from stem toward plus/minus */
[train_integrity_point_stem_to_plusminus]
(∀p : Point •
 (occupied_without_head(p.S2PM) ⇒
  ((occupied_without_tail(hto(plus(p),p)) ∧ p.POS = PLUS) ⊕
  (occupied_without_tail(hto(minus(p),p)) ∧ p.POS = MINUS))) ∧
 (occupied_without_tail(hto(plus(p),p)) ⇒
  (occupied_without_head(p.S2PM) ∧ p.POS = PLUS)) ∧
 (occupied_without_tail(hto(minus(p),p)) ⇒
  (occupied_without_head(p.S2PM) ∧ p.POS = MINUS)) ∧
 ¬(occupied_without_tail(hto(plus(p),p)) ∧
  occupied_without_tail(hto(minus(p),p)))),
```

```
/* point section , from plus/minus toward stem */
[train_integrity_point_plusminus_to_stem]
(∀p : Point •
 (occupied_without_head(p.P2S) ⇒
  (occupied_without_tail(hto(stem(p),p)) ∧ p.POS = PLUS)) ∧
 (occupied_without_head(p.M2S) ⇒
  (occupied_without_tail(hto(stem(p),p)) ∧ p.POS = MINUS)) ∧
 (occupied_without_tail(hto(stem(p),p)) ⇒
  ((occupied_without_head(p.P2S) ∧ p.POS = PLUS) ⊕
  (occupied_without_head(p.M2S) ∧ p.POS = MINUS)))),

/* \ ~ (occupied_without_head(p.P2S) /\ occupied_without_head(p.M2S))), ->
 * covered by point
 * safety prop */
/**
 * CONDITIONS ON ROUTES
 * ====================
 */
[mode_and_display_are_identical]
(∀r : Route • r.MODE = r.DSPL),

[conflicting_routes_are_not_set_together]
(∀r : Route •
 (r.MODE = ALLOCATING ∨ r.MODE = LOCKED) ⇒
 (∀cr : Route •
  cr ∈ conflicts(r) ⇒ (cr.MODE ≠ ALLOCATING ∧ cr.MODE ≠ LOCKED))),

[route_allocating_cnd]
(∀r : Route •
 (r.MODE = ALLOCATING) ⇒
 /* points are commanded in correct positions */
 ((∀p : Point • p ∈ points(r) ⇒ (p.CMD = req(r,p))) ∧
  /* protecting signals are commanded in correct aspects */
  (∀s : Signal • s ∈ signals(r) ⇒ (s.CMD = CLOSED)) ∧
  /* all lockable elements in the path are EXLCK(1) */
  (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK)) ∧
  /* all sections have to be vacant */
  (∀e : Section •
   e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e)))),

[route_lock_cnd]
(∀r : Route •
 (r.MODE = LOCKED) ⇒
 let fst = first(r) in
   /* points are in correct positions */
   (∀p : Point •
    p ∈ points(r) ⇒ (p.POS = req(r,p) ∧ p.POS = p.CMD)) ∧
```

```
    /* protecting signals are in correct aspects */
    (∀s : Signal •
     s ∈ signals(r) ⇒ (s.ACT = CLOSED ∧ s.ACT = s.CMD)) ∧
    /* all lockable elements in the path are EXLCK(1) */
    (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK)) ∧
    /* all sections except the first one have to be vacant */
    (∀e : Section •
     (e ≠ fst ∧ e ∈ (elems path(r) ∪ elems overlap(r))) ⇒
     vacant(e)) ∧
    /* first section is vacant or occupied by head of the train */
    (vacant(fst) ∨ hto(fst,r) = 5) ∧
    /* entry signal is commanded to be open */
    src(r).CMD = OPEN
  end),

[route_used_cnd_a]
(∀r : Route • r.MODE = OCCUPIED ⇒ (last(r).MODE ≠ AVAIL)),

[route_used_cnd_b]
(∀r : Route •
 r.MODE = OCCUPIED ⇒
 (count_fwd___O(r) ∨
  let fst = first(r),
      lst = last(r) in
    ((lst ≠ fst) ? lst.PREV : (lst.MODE = USED)) ∧ vacant(last(r))
  end)),

[route_used_cnd_c]
(∀r : Route • r.MODE = OCCUPIED ⇒ linear_chunk_cnd(r,last(r))),

[route_in_use_last_sec_is_free_in_opposite_dir]
(∀r : Route • r.MODE = OCCUPIED ⇒ bwd_hto(last(r),r,0) = 0),

[routes_share_last_not_used_at_same_time]
(∀r : Route •
 r.MODE = OCCUPIED ⇒
 (∀opr : Route •
  (opr ≠ r ∧ last(r) = last(opr)) ⇒ (opr.MODE ≠ OCCUPIED))),

[first_entry_cnd]
(∀r : Route •
 ¬(¬vacant(first(r)) ∧ src(r).CMD = OPEN ∧ src(r).ACT = CLOSED)),

/**
 * CONDITIONS ON SIGNALS
 * ======================
 */
```

```
[entry_signal_closed_when_tail_in_first]
(∀r : Route • _T_(hto(first(r), r)) ⇒ src(r).ACT = CLOSED),

[signal_cmd_open_cnd]
(∀s : Signal •
 (∃r : Route • s = src(r)) ⇒
 (s.CMD = OPEN ⇒ (∃!r : Route • src(r) = s ∧ r.MODE = LOCKED))),

[signal_act_open_cnd]
(∀s : Signal •
 (∃r : Route • s = src(r)) ⇒
 (s.ACT = OPEN ⇒
  ((∃r : Route •
    s = src(r) ∧ r.MODE = FREE ∧ s.CMD = CLOSED ∧ r.CTRL = CANCEL) ∨
   (∃!r : Route •
    (src(r) = s) ∧
    (((r.MODE = LOCKED ∧
       (s.CMD = OPEN ∨ H__(hto(first(r),r)) ≠ 0)) ∨
      (r.MODE = OCCUPIED ∧ s.CMD = CLOSED ∧
       H__(hto(first(r), r)) ≠ 0)) ∧
     (∀e : Section •
      (e ∈ path(r) ∧ e ≠ first(r)) ⇒
      (vacant(e) ∧ e.MODE = EXLCK)) ∧
     (∀e : Section • e ∈ overlap(r) ⇒ vacant(e )))))),

/**
 * POINT SWITCHING CONDITIONS
 * ==========================
 */
[not_commanding_occupied_point_to_move]
(∀p : Point • (¬vacant(p) ∨ p.MODE = USED) ⇒ p.POS = p.CMD),

/* protecting point can be in FREE mode */
[point_only_cmd_when_alloc_a_route]
(∀e : Point •
 (∃r : Route • e ∈ points(r)) ⇒
 (e.CMD ≠ e.POS ⇒
  (e.MODE ≠ USED ∧
   (∃r : Route • e ∈ points(r) ∧ r.MODE = ALLOCATING)))),

/**
 * GROUND UNOCCUPIED ELEMENTS
 * ======================
 */
/* if a signal is not an entry signal of any route, then it is always
 * CLOSED */
[ground_unused_signal]
```

```
(∀s : Signal •
 (∀r : Route • src(r) ≠ s) ⇒ (s.ACT = CLOSED ∧ s.CMD = CLOSED)),

/* ground unused segments */
[ground_unused_linear_to_down]
(∀l : Linear •
 /* a section is not used by any routes in the direction up */
 (¬is_boundary_sec(l) ∧
  (∀r : Route • l ∈ path(r) ⇒ entry(r,l) ≠ UP) ∧
  /* trains cannot turn around here */
  ¬(can_turn_around_at(l) ∧
   (∃r : Route • l ∈ path(r) ∧ entry(r,l) = DOWN))) ⇒
 (l.U2D = 0)),

[ground_unused_linear_to_up]
(∀l : Linear •
 /* a section is not used by any routes in the up direction up */
 (¬is_boundary_sec(l) ∧
  (∀r : Route • l ∈ path(r) ⇒ entry(r,l) ≠ DOWN) ∧
  /* trains cannot turn around here */
  ¬(can_turn_around_at(l) ∧
   (∃r : Route • l ∈ path(r) ∧ entry(r,l) = UP))) ⇒ (l.D2U = 0)),

[ground_unused_point_p]
(∀p : Point •
 (¬(∃r : Route •
   p ∈ (dom points(r) ∩ elems path(r)) ∧ entry(r,p) = PLUS)) ⇒
 (p.P2S = 0)),

[ground_unused_point_m]
(∀p : Point •
 (¬(∃r : Route •
   p ∈ (dom points(r) ∩ elems path(r)) ∧ entry(r,p) = MINUS)) ⇒
 (p.M2S = 0)),

[ground_unused_point_s]
(∀p : Point •
 (¬(∃r : Route •
   p ∈ (dom points(r) ∩ elems path(r)) ∧ entry(r,p) = STEM)) ⇒
 (p.S2PM = 0)),

/**
 * Ground boundary sections
 */
[ground_unused_boundary_linear]
(∀l : Linear • is_boundary_sec(l) ⇒ (l.U2D = 0 ∧ l.D2U = 0)),
```

```
/**
 * CONDITIONS ON ELEMENTS
 * ======================
 */
[element_prev_variable]
(∀e : Section • is_boundary_sec(e) ∨ (e.PREV = RELEASED ⇒ e.MODE = USED)),

[occupied_implies_exlck_or_used_point]
(∀e : Point •
 (∀r : Route • e ≠ last(r)) ⇒
 (¬vacant(e) ⇒ (e.MODE = EXLCK ∨ e.MODE = USED))),

[occupied_implies_exlck_or_used_linear]
(∀e : Linear •
 ((∀r : Route • e ≠ last(r)) ∧ down(e) ∧ up(e)) ⇒
 (¬vacant(e) ⇒ (e.MODE = EXLCK ∨ e.MODE = USED))),

[point_chunk_cnd_stem_plus]
(∀e : Point •
 let s = stem(e),
     p = plus(e),
     s_b = hto(stem(e),e,0),
     p_b = hto(plus(e),e,0),
     s_t = e.S2PM,
     p_t = e.P2S in
   (¬can_turn_around_at(s) ∧ ¬can_turn_around_at(p)) ⇒
   ((e.POS = PLUS) ⇒
    ((p_t + s_b) * (s_t + p_b * (¬p.PREV ∧ p.MODE = USED)) = 0))
 end),

[point_chunk_cnd_stem_minus]
(∀e : Point •
 let s = stem(e),
     m = minus(e),
     s_b = hto(stem(e),e,0),
     m_b = hto(minus(e),e,0),
     s_t = e.S2PM,
     m_t = e.M2S in
   (¬can_turn_around_at(s) ∧ ¬can_turn_around_at(m)) ⇒
   ((e.POS = MINUS) ⇒
    ((m_t + s_b) * (s_t + m_b * (¬m.PREV ∧ m.MODE = USED)) = 0))
 end),

[hto_vs_mode]
(∀r : Route •
 (∀e : Section •
  (e ∈ path(r) ∧ e ≠ last(r)) ⇒
```

```
  ((hto(e,r,0) = 1 ⇒ e.MODE = USED) ∧
   (hto(e,r,0) = 3 ⇒ e.MODE = USED) ∧
   (hto(e,r,0) = 5 ⇒ e.MODE = EXLCK ∨ e.MODE = USED) ∧
   (hto(e,r,0) = 7 ⇒ e.MODE = USED)))),

[hto_vs_mode_last_down]
(∀r : Route •
 (entry(r, last(r)) = 0) ⇒
 let e = last(r) in
   (e.D2U = 1 ⇒ e.MODE = USED) ∧
   (e.D2U = 3 ⇒ e.MODE = AVAIL ∨ e.MODE = USED) ∧
   (e.D2U = 5 ⇒ e.MODE = EXLCK ∨ e.MODE = USED) ∧
   (e.D2U = 7 ⇒ e.MODE = AVAIL ∨ e.MODE = USED) ∧
   ((r.MODE = OCCUPIED ∧ e.MODE = USED ∧ e.PREV = RELEASED ∧ e.D2U = 0) ⇒
    (e.U2D = 0 ∧
     (¬up(e) ∨ is_boundary_sec_up(up(e)) ∨ _T_(hto(up(e),e)) = 1)))
 end),

[hto_vs_mode_last_up]
(∀r : Route •
 (entry(r, last(r)) = 1) ⇒
 let e = last(r) in
   (e.U2D = 1 ⇒ e.MODE = USED) ∧
   (e.U2D = 3 ⇒ e.MODE = AVAIL ∨ e.MODE = USED) ∧
   (e.U2D = 5 ⇒ e.MODE = EXLCK ∨ e.MODE = USED) ∧
   (e.U2D = 7 ⇒ e.MODE = AVAIL ∨ e.MODE = USED) ∧
   ((r.MODE = OCCUPIED ∧ e.MODE = USED ∧ e.PREV = RELEASED ∧ e.U2D = 0) ⇒
    (e.D2U = 0 ∧
     (¬down(e) ∨ is_boundary_sec_down(down(e)) ∨ _T_(hto(down(e),e)) = 1)))
 end),

[elem_locked_by_only_one_route]
(∀e : Section •
 (∃r : Route • e ∈ path(r)) ⇒
 (e.MODE = EXLCK ⇒
  (∃!r : Route •
   e ∈ path(r) ∧
   let lst = last(r),
       fst = first(r) in
     /* route is allocating, imply all elems locked */
     ((r.MODE = ALLOCATING) ∨
      /* route is locked, imply all elems locked */
      (r.MODE = LOCKED) ∨
      /* route in use, the 1st elem is never locked when
       * the route is in use */
      ((e ≠ fst) ∧ r.MODE = OCCUPIED ∧
       ((e ∈ Point) ⇒ (e.POS = req(r,e) ∧ e.CMD = req(r,e))) ∧
```

```
      /* next elems in the same route are locked and vacant */
      (∀ne : Section •
       ne ∈ nexts(r,e) ⇒
       (ne.MODE = EXLCK ∧ vacant(ne) ∧
        ((ne ∈ Point) ⇒
         (ne.POS = req(r,ne) ∧ ne.CMD = req(r,ne))))) ∧
      /* the head of the train is in e
       * then prev is occupied without a head
       * we have e ~= fst , thus pv always exists */
      let pv = prev(r,e) in
        (hto(e,r) = 5 ∧ pv.MODE = USED ∧
         ((pv ∈ Point) ⇒ pv.POS = req(r,pv)) ∧
         occupied_without_head(hto(pv,r))) ∨
        /* e is vacant, then the prev is locked or used */
        (vacant(e) ∧ (pv.MODE = EXLCK ∨ pv.MODE = USED) ∧
         ((pv ∈ Point) ⇒ (pv.POS = req(r,pv) ∧ pv.CMD = req(r,pv))))
      end))
  end))),

[elem_used_by_only_one_route]
(∀e : Section •
 (∃r : Route • e ∈ path(r)) ⇒
 (e.MODE = USED ⇒
  (∃!r : Route •
   e ∈ path(r) ∧
   let fst = first(r),
       lst = last(r) in
     r.MODE = OCCUPIED ∧ ((e ∈ Point) ⇒ e.POS = req(r,e)) ∧
     (e ≠ fst ⇒
      let pv = prev(r,e) in
        /* the prev has been released */
        e.PREV ∨
        /* the prev is used without a head
         * or is going to be released */
        (((pv ∈ Point) ⇒ pv.POS = req(r,pv)) ∧ pv.MODE = USED ∧
         (occupied_without_head(hto(pv,r)) ∨
          (vacant(pv) ∧ _T_(hto(e,r)) = 1 ∧ (pv ≠ fst ⇒ pv.PREV))))
      end) ∧
     (e ≠ lst ⇒
      let nx = next(r,e) in
        ¬nx.PREV ∧
        /* this includes nx */
        (∀p : Point •
         p ∈ nexts(r,e) ⇒ (p.POS = req(r,p) ∧ p.CMD = req(r,p))) ∧
        ((nx.MODE = EXLCK ∧
          ((hto(nx,r) = 5 ∧ occupied_without_head(hto(e,r))) ∨
           (vacant(nx) ∧ H__(hto(e,r)) = 1)) ∧
```

```
        (∀ne : Section •
         (ne ≠ nx ∧ ne ∈ nexts(r,e)) ⇒ (ne.MODE = EXLCK))) ∨
       (nx.MODE = USED ∧
        ((occupied_without_head(hto(e,r)) ∧
          occupied_without_tail(hto(nx,r))) ∨
         (vacant(e) ∧ _T_(hto(nx,r)) = 1 ∧ (e ≠ fst ⇒ e.PREV)))))
    end)
  end)))

test_obj

([=] r : Route •
 ([=] other : Route •
  [FR_non_conflicting_routes_allocating]
  [other ≠ r ∧ other ∉ conflicts(r)] ∧
  G F [(r.DSPL = ALLOCATING ∧ other.DSPL = ALLOCATING)])),

([=] r : Route •
 ([=] other : Route •
  [FR_non_conflicting_routes_locked]
  [other ≠ r ∧ other ∉ conflicts(r)] ∧
  G F [(r.DSPL = LOCKED ∧ other.DSPL = LOCKED)])),

([=] r : Route •
 ([=] other : Route •
  [FR_non_conflicting_routes_used]
  [other ≠ r ∧ other ∉ conflicts(r)] ∧
  G F [(r.DSPL = OCCUPIED ∧ other.DSPL = OCCUPIED)])),

([=] r : Route • [FR_route_marked] G F [r.DSPL = MARKED]),

([=] r : Route • [FR_route_in_use] G F [r.DSPL = OCCUPIED]),

([=] r : Route • [FR_route_allocating] G F [r.DSPL = ALLOCATING]),

([=] r : Route • [FR_route_locked] G F [r.DSPL = LOCKED]),

([=] r : Route •
 [FR_cancel_marked_route]
 G([r.CTRL = CANCEL ∧ r.DSPL = MARKED] ⇒ F [r.DSPL = FREE])),

([=] r : Route •
 [FR_cancel_allocating_route]
 G([r.CTRL = CANCEL ∧ r.DSPL = ALLOCATING] ⇒
  F [/* the route get free */
   (r.DSPL′ = FREE) ∧
   /* points in the path stop moving */
```

```
    (∀p : Point • p ∈ path(r) ⇒ (p.CMD = p.POS)) ∧
    /* all sections in the path get unlocked */
    (∀e : Section • e ∈ path(r) ⇒ (e.MODE = AVAIL))])),

([=] r : Route •
 [FR_cancel_locked_route]
 G([r.CTRL = CANCEL ∧ r.DSPL = ALLOCATING ∧
    /* the route has not been used, i.e., all the route's path and
     * overlap are still vacant */
    (∀e : Section •
     e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e))] ⇒
  F [/* the route get free */
    (r.DSPL = FREE) ∧
    /* close the source signal */
    (src(r).CMD = CLOSED) ∧
    /* all sections in the path get unlocked */
    (∀e : Section • e ∈ path(r) ⇒ (e.MODE = AVAIL))])),

([=] e : Linear •
 [RR_linear_occupied_implies_exlck_or_used]
 [(∀r : Route • e ≠ last(r)) ∧ down(e) ∧ up(e)] ⇒
 G [¬vacant(e) ⇒ (e.MODE = EXLCK ∨ e.MODE = USED)]),

([=] e : Point •
 [RR_point_occupied_implies_exlck_or_used]
 [(∀r : Route • e ≠ last(r))] ⇒
 G [¬vacant(e) ⇒ (e.MODE = EXLCK ∨ e.MODE = USED)]),

/**
 * GENERIC REQUIREMENTS TAKEN FROM THE PAPER
 * Anne Haxthausen \& Jan Peleska. Efficient Development and Verification of
 * Safe Railway Control Software. Nova Science Publishers Inc., 2013.
 * [[ bib : Haxthausen13Efficient ]] [[ papers : Haxthausen13Efficient ]]
 * ==========================================================================
 */
/**
 * It shall be possible to allocate/lock each specified route
 * ([=] r : Route :- G(F(r.DSPL = ALLOCATING))
 */
([=] r : Route • [tcgen_route_allocating] G(F([r.DSPL = ALLOCATING]))),

/**
 * ([=] r : Route :- G(F(r.DSPL = LOCKED))
 */
([=] r : Route • [tcgen_route_locked] F([r.DSPL = LOCKED])),

/**
```

```
 * It shall be possible to use each specified route
 * ([=] r : Route :- G(F(r.DSPL = OCCUPIED))
 */
([=] r : Route • [tcgen_route_in_use] F([r.DSPL = OCCUPIED])),

/**
 * Non-conflicting routes may be allocated in a concurrent way
 */
([=] r : Route •
 ([=] other : Route •
  [tcgen_non_conflicting_routes_req]
  [other ≠ r ∧ other ∉ conflicts(r)] ∧
  G F [(r.DSPL = LOCKED ∧ other.DSPL = LOCKED)])),

([=] r : Route •
 ([=] other : Route •
  [tcgen_non_conflicting_routes]
  [other ≠ r ∧ other ∉ conflicts(r)] ∧
  F([r.DSPL = LOCKED ∧ other.DSPL = LOCKED]))),

/**
 * Conflicting routes which can be used at the same time
 */
([=] r : Route •
 ([=] other : Route •
  [RR_conflicting_route_used_same_time]
  [other ∈ conflicts(r) ∧
   (∃p : Point •
    p ∈ dom points(r) ∩ dom points(other) ∧
    req(r,p) = MINUS ∧ req(other,p) = PLUS ∧
    last(r) ≠ last(other))] ∧
  G F([r.DSPL = OCCUPIED ∧ other.DSPL = OCCUPIED]))),

/**
 * Allocation to a train is only granted after the points are locked in the
 * positions required for the requested route
 */
// ([=] r : Route :- [tcgen_open_signal] F([src(r).CMD = OPEN])),
([=] r : Route •
 [tcgen_open_signal_req]
 G([src(r).CMD = OPEN] ⇒
  [(∀s : Signal • s ∈ signals(r) ⇒ s.ACT = CLOSED) ∧
   (∀p : Point • p ∈ points(r) ⇒ p.CMD = req(r,p))])),

/* conflicting routes */
([=] r : Route •
 [SR_conflicting_routes_allocating]
```

```
 G [(r.DSPL = ALLOCATING) ⇒
  (∀other : Route •
   other ∈ conflicts (r) ⇒
   other.DSPL ≠ ALLOCATING ∧ other.DSPL ≠ LOCKED)]),

([=] r : Route •
 [SR_conflicting_routes_locked]
 G [(r.DSPL = LOCKED) ⇒
  (∀other : Route •
   other ∈ conflicts (r) ⇒
   other.DSPL ≠ ALLOCATING ∧ other.DSPL ≠ LOCKED)]),

/* route allocating */
([=] r : Route •
 [SR_route_allocating]
 G [(r.DSPL = ALLOCATING) ⇒
  ((∀p : Point • p ∈ points(r) ⇒ (p.CMD = req(r,p))) ∧
   /* protecting signals are commanded in correct aspects */
   (∀s : Signal • s ∈ signals(r) ⇒ (s.CMD = CLOSED)) ∧
   /* all lockable elements in the path are EXLCK(1) */
   (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK)) ∧
   /* all sections have to be vacant */
   (∀e : Section •
    e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e)))] ),

/* route locked */
([=] r : Route •
 [SR_route_locked]
 G [(r.DSPL = LOCKED) ⇒
  let fst = first (r) in
    /* points are in correct positions */
    (∀p : Point •
     p ∈ points(r) ⇒ (p.POS = req(r,p) ∧ p.POS = p.CMD)) ∧
    /* protecting signals are in correct aspects */
    (∀s : Signal •
     s ∈ signals(r) ⇒ (s.ACT = CLOSED ∧ s.ACT = s.CMD)) ∧
    /* all lockable elements in the path are EXLCK(1) */
    (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK)) ∧
    /* all sections except the first one have to be vacant */
    (∀e : Section •
     (e ≠ fst ∧ e ∈ (elems path(r) ∪ elems overlap(r))) ⇒
     vacant(e)) ∧
    /* first section is vacant or occupied by head of the train */
    (vacant(fst) ∨ hto(fst,r) = 5) ∧
    /* entry signal is commanded to be open */
    src(r).CMD = OPEN
  end]),
```

```
([=] r : Route • [TC_route_locked] F [r.DSPL = LOCKED]),

/* not commanding used points to move */
([=] p : Point •
 [SR_not_commanding_used_point_to_move]
 G [(¬vacant(p) ∨ p.MODE = USED) ⇒ p.POS = p.CMD]),

([=] p : Point •
 [SR_point_only_cmd_when_alloc_a_route]
 [(∃r : Route • p ∈ points(r))] ⇒
 G [(p.CMD ≠ p.POS) ⇒
  (p.MODE ≠ USED ∧
   (∃r : Route • p ∈ points(r) ∧ r.DSPL = ALLOCATING) ∧
   (∀r : Route • p ∈ points(r) ⇒ r.DSPL ≠ LOCKED))]),

/* signal commanded open */
([=] s : Signal •
 [SR_signal_cmd_open_cnd]
 [(∃r : Route • s = src(r))] ⇒
 G [(s.CMD = OPEN) ⇒ (∃!r : Route • src(r) = s ∧ r.DSPL = LOCKED)]),

/* signal actual open */
([=] s : Signal •
 [SR_signal_act_open_cnd]
 [(∃r : Route • s = src(r))] ⇒
 /* whenever the signal's actual aspect is OPEN */
 G [(s.ACT = OPEN) ⇒
  /* there exists a route r that has s as its source signal. The route has
   * been cancelled, but the signal has not been closed as commanded yet
   */
  ((∃r : Route •
    s = src(r) ∧ r.DSPL = FREE ∧ s.CMD = CLOSED ∧ r.CTRL = CANCEL) ∨
   /* OR there is exactly one route r that has s as the source signal */
   (∃!r : Route •
    (src(r) = s) ∧
    /* r is locked */
    (((r.DSPL = LOCKED ∧
       (s.CMD = OPEN ∨ H__(hto(first(r),r)) ≠ 0)) ∨
      /* a train has entered the route, the interlocking
       * controller has reacted, but the signal has not yet
       */
      (r.DSPL = OCCUPIED ∧ s.CMD = CLOSED ∧
       H__(hto(first(r), r)) ≠ 0)) ∧
     /* all sections in the route's path, except the first one, are
      * exclusively locked and vacant */
     (∀e : Section •
```

```
      (e ∈ path(r) ∧ e ≠ first(r)) ⇒
      (vacant(e) ∧ e.MODE = EXLCK)) ∧
     /* all sections in the route's overlap are vacant */
     (∀e : Section • e ∈ overlap(r) ⇒ vacant(e)))))] ),

/* simplified */
([=] r : Route •
 ([=] e : Section •
  [RR_elem_released_following_train_passage]
  [e ∈ path(r) ∧ e ≠ last(r)] ∧
  G([(r.MODE = OCCUPIED ∧ e.MODE = USED ∧ vacant(e) ∧
     (e ≠ first(r) ⇒ e.PREV = RELEASED))] ⇒
   F [e.MODE = AVAIL ∧ r.MODE = OCCUPIED ∧ next(r,e).MODE = USED]))),

/**
 * only release an element when its previous element in the same route has been
 * released
 */
([=] r : Route •
 ([=] e : Section •
  [RR_elem_released_after_prev_has_been_released]
  [e ∈ path(r) ∧ e ≠ first(r)] ∧
  G([r.MODE = OCCUPIED ∧ e.MODE = USED ∧ e.PREV = PENDING] ⇒
   X([e.MODE ≠ FREE] U [e.PREV = RELEASED]))))

macro

/**
 * NAMED CONSTANTS
 */
/* control modes */
def NOCMD = 0,

def DISPATCH = 1,

def CANCEL = 2,

/* Route modes */
def FREE = 0,

def MARKED = 1,

def ALLOCATING = 2,

def LOCKED = 3,

def OCCUPIED = 4,
```

```
/* Signals aspect */
def CLOSED = 0,

def OPEN = 1,

/* Point positions and neighbor sides */
def PLUS = 0,

def MINUS = 1,

def STEM = 2,

def INTER = 2,

/* Element modes */
def AVAIL = 0,

def EXLCK = 1,

def USED = 2,

/* prev variable values */
def PENDING = 0,

def RELEASED = 1,

/* DIRECTION */
def DOWN = 0,

def UP = 1,

/*
 * Perform function f on a linear section
 * e: the linear element
 * p: the neighbor end
 * 0: down end
 * 1: up end
 * f: function to perform, takes two parameters: current value, next
 * value
 */
def do_linear(e,p,f) =
case p of
  DOWN → f(e.D2U,e.D2U′),
  UP → f(e.U2D,e.U2D′)
end,
```

```
/*
 * Perform function f on a point section
 * e: the point element
 * p: the neighbor end
 * 0: plus end
 * 1: minus end
 * 2: stem end
 * f: function to perform, takes two parameters: current value, next
 * value
 */
def do_point(e,p,f) =
case p of
  PLUS → f(e.P2S,e.P2S′),
  MINUS → f(e.M2S,e.M2S′),
  STEM → f(e.S2PM,e.S2PM′)
end,

/*
 * Perform function f on a section. The function will decide whether to
 * perform
 * do_linear or do_point based on the type of e.
 * e: the linear element
 * p: the neighbor end
 * f: function to perform, takes two parameters: current value, next
 * value
 */
def do_sec(e,p,f) =
(e ∈ Linear) ? do_linear(e,p,f) : do_point(e,p,f),

/*
 * A function returns HTO variable at version 0
 */
def _hto0(v0,v1) =
v0,

/*
 * A function returns HTO variable at version 0
 */
def _hto1(v0,v1) =
v1,

/*
 * Return the HTO variable of the element e in the direction that the route r
 * enters e
 */

def route_entry_hto(e,r,v) =
```

```
case v of
  0 → do_sec(e,entry(r,e),_hto0),
  1 → do_sec(e,entry(r,e),_hto1)
end,

def bwd_hto(e,r,v) =
case v of
  0 → do_sec(e,exit(r,e),_hto0),
  1 → do_sec(e,exit(r,e),_hto1)
end,

/*
 * return the HTO variable of e which encodes traffic coming from n
 */

def neighbor_hto(e,n,v) =
case v of
  0 → do_sec(e,conn_end(e,n),_hto0),
  1 → do_sec(e,conn_end(e,n),_hto1)
end,

/*
 * Return the HTO variable of the element e based on either :
 * – the end that neighbor x is connected to e
 * – the direction that route x enters e
 */

def hto(e,x,v) =
(x ∈ Route) ? route_entry_hto(e,x,v) : neighbor_hto(e,x,v),

/*
 * An overload of hto(x,e,v), returns HTO variable at version 0
 */
def hto(e,x) =
hto(e,x,0),

/*
 * Return H bit of a HTO variable
 */

def H__(hto) =
(hto ≫ 2),

/*
 * Return T bit of HTO variable
 */
```

```
def _T_(hto) =
((hto & 2) ≫ 1),

/*
 * Return O bit of HTO variable
 */

def __O(hto) =
(hto & 1),

/*
 * A formula encodes whether a linear section l is vacant
 */
def vacant_linear(l) =
(l.D2U + l.U2D = 0),

/*
 * A formula encodes whether a point section p is vacant
 */

def vacant_point(p) =
(p.S2PM + p.P2S + p.M2S = 0),

/*
 * A formula encodes whether a section e is vacant
 */

def vacant(e) =
(e ∈ Linear) ? vacant_linear(e) : vacant_point(e),

/*
 * Toggle the H and O bit of a section
 */

def head_enters(hto0,hto1) =
hto1 = (hto0 ^ 5),


def head_enters_next(nx,curr) =
let hto0 = hto(nx,curr,0),
    hto1 = hto(nx,curr,1) in
  head_enters(hto0,hto1)
end,

/*
 * Toggle the H bit of a section
 */
```

```
def head_leaves(hto0,hto1) =
hto1 = (hto0^ 4),

/*
 * Toggle the T bit of a section
 */

def tail_enters (hto0,hto1) =
hto1 = (hto0^ 2),


def tail_enters_next (nx,curr) =
let hto0 = hto(nx,curr ,0),
    hto1 = hto(nx,curr,1) in
  tail_enters (hto0,hto1)
end,

/*
 * Tail leaves
 */

def tail_leaves (hto0,hto1) =
hto1 = 0,

/**
 * Swap values of occupancy status variables in a linear section
 */

def swap_up_down_vars(l) =
l.D2U′ = l.U2D ∧ l.U2D′ = l.D2U,

/**
 * ==================================
 * Macros for strenthening properties
 * ==================================
 */
/*
 * These macros are used for specifying strengthening invariants
 */
/*
 * Occupied without a tail : 101 001
 */

def occupied_without_tail(hto) =
(hto & 3) = 1,
```

```
/*
 * Occupied without a head: 001 011
 */

def occupied_without_head(hto) =
(hto & 5) = 1,

/*
 * Occupied with a head: 111 101
 */

def occupied_with_head(hto) =
(hto & 5) = 5,

/*
 * Occupied with only the tail : 011
 */

def occupied_with_only_tail(hto) =
hto = 3,

/*
 * If trains can change direction at a section
 */

def can_turn_around_at(e) =
(e ∈ Linear) ∧ down_sig(e) ∧ up_sig(e),

/* Apply function f to HTO variables (in the same direction as r's) of elements
 * in the path of r, start from the last section toward the first section,
 * until a point which has different position than required by r is met
 */
def count_fwd_bit(r,f) =
count_fwd_bit(r,last(r), f),

/*
 * Apply function f to HTO variables (in the same direction as r's) of elements
 * in the path of r, start from e toward the first section, until a point which
 * has different position than required by r is met
 */
def count_fwd_bit(r,e,f) =
let fst = first(r) in
  if (e ∈ Linear)
  then
    (e = fst) ? f(hto(e,r,0)) : (f(hto(e,r,0)) + count_fwd_bit(r,prev(r,e),f))
  else
    (e.POS = req(r,e)) ∧
```

```
    ((e = fst) ? f(hto(e,r,0))
     : (f(hto(e,r,0)) + count_fwd_bit(r,prev(r,e),f)))
  end
end,

/*
 * Apply function f to HTO variables (in the opposite direction of r's) of
 * elements in the path of r, start from the last section toward the first
 * section, until a point which has different position than required by r is
 * met
 */
def count_bwd_bit(r,f) =
count_bwd_bit(r,last(r),f),

/*
 * Apply function f to HTO variables (in the opposite direction of r's) of
 * elements in the path of r, start from e toward the first section, until a
 * point which has different position than required by r is met
 */
def count_bwd_bit(r,e,f) =
let fst = first(r) in
  if (e ∈ Linear)
  then
    let val = (entry(r,e) = DOWN) ? f(e.U2D) : f(e.D2U) in
      (e = fst) ? val : (val + count_bwd_bit(r,prev(r,e),f))
    end
  else
    (e.POS = req(r,e)) ∧
    let val = (entry(r,e) ≠ STEM) ? f(e.S2PM)
              : ((req(r,e) = PLUS) ? f(e.P2S) : f(e.M2S)) in
      (e = fst) ? val : (val + count_bwd_bit(r,prev(r,e),f))
    end
  end
end,

/*
 * Search backward from e for a point
 */
def check_point(r,e) =
let fst = first(r) in
  if (e = fst)
  then
    e
  else
    let pv = prev(r,e) in
      (pv ∈ Point) ? e : check_point(r,pv)
    end
```

```
  end
end,

/*
 * This formula ensures each chunk of linear sections between two points of a
 * route only is occupied in one direction at a time
 */
def linear_chunk_cnd(r,e) =
let fst = first(r),
    cp = check_point(r,e) in
  ((cp = e) ? 1
   : (count_chunk_fwd_bit(r,e,cp,__O) * count_chunk_bwd_bit(r,e,cp,__O) =
      0)) ∧ ((cp ≠ fst) ⇒ linear_chunk_cnd(r,prev(r,cp)))
end,

/*
 * Apply function f to HTO variables (in the same direction of r's) of elements
 * in the path of r, start from e toward stop
 */
def count_chunk_fwd_bit(r,e,stop,f) =
(e = stop) ? f(hto(e,r,0))
: (f(hto(e,r,0)) + count_chunk_fwd_bit(r,prev(r,e),stop,f)),

/*
 * Apply function f to HTO variables (in the opposite direction of r's) of
 * elements in the path of r, start from e toward stop
 */
def count_chunk_bwd_bit(r,e,stop,f) =
if (e ∈ Linear)
then
  let val = (entry(r,e) = DOWN) ? f(e.U2D) : f(e.D2U) in
    (e = stop) ? val : (val + count_chunk_bwd_bit(r,prev(r,e),stop,f))
  end
else
  let val = (entry(r,e) ≠ STEM) ? f(e.S2PM)
            : ((req(r,e) = PLUS) ? f(e.P2S) : f(e.M2S)) in
    (e = stop) ? val : (val + count_chunk_bwd_bit(r,prev(r,e),stop,f))
  end
end,

/*
 * Count O bit of forward HTO variables
 */
def count_fwd___O(r) =
count_fwd_bit(r,__O),

/* check if e is a boundary section */
```

```
def is_boundary_sec(e) =
(is_boundary_sec_down(e) ∨ is_boundary_sec_up(e)),

/* check if e is a boundary section in the downward direction */

def is_boundary_sec_down(e) =
(e ∈ Linear) ∧ ¬down(e) ∧ up_sig(e) ∧ ¬down_sig(e) ∧
let us = up(e) in
  (us ∈ Linear) ∧ down_sig(us)
end,

/* check if e is a boundary section in the upward direction */

def is_boundary_sec_up(e) =
(e ∈ Linear) ∧ ¬up(e) ∧ down_sig(e) ∧ ¬up_sig(e) ∧
let ds = down(e) in
  (ds ∈ Linear) ∧ up_sig(ds)
end

end
```

APPENDIX E

Strengthening Invariants

This appendix lists all strengthening invariants that are used in our verification strategy for verifying safety properties of the forthcoming Danish interlocking systems. Note that some of these strengthening invariants are also safety properties at a lower level. Further detail can be found in the full specification of the generic applications for the forthcoming Danish interlocking systems in Appendix D.

E.1 Train Integrity

The occupancy status encoding described in Section 6.2.1 allows us to distinguish different situations in which:

(a) a section is vacant,

(b) a section is occupied by the whole train

(c) a section is occupied by just the head of the train

(d) a section is occupied by just the tail of the train, or

(e) the train spreads over the section, i.e., when the train is longer than the section.

Figure E.1 shows the values of the occupancy status variable of the sections corresponding to different cases.

Some spurious counter-examples given by the model checker start from a state where the train integrity is violated. An example of such cases is shown in Figure 6.16b where the train of the right has only the head.

In order to eliminate such spurious cases, we introduce the *train integrity* invariant. For a linear section l that is not a boundary section and its neighbouring section in the considered travel direction is not a boundary section, the following property must hold

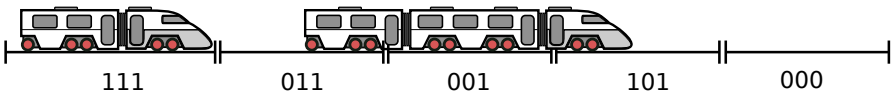


Figure E.1: The values of the occupancy status variable of sections in different situations. The labels under each section denotes the value of the occupancy status variable for that section.

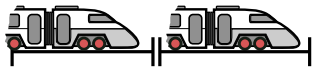


Figure E.2: A spurious state where the train on the right has only the head

TI-L-1 *l* is occupied by a train in the direction *up* (*down*) without the head being on *l*, if and only if *up(l)* (*down(l)*) is occupied by a train in the same direction without the tail being on *up(l)* (*down(l)*).

For a boundary section*, when the previous section or next section cannot be determined – because the considered boundary section does not have those neighbouring sections – we consider that the train integrity properties hold for that section. The *train integrity* invariant for linear sections is specified in detailed in the following.

[train_integrity_linear_up]
(∀l : **Linear** •
(**up**(l) ∧ ¬is_boundary_sec_up(**up**(l)) ∧ ¬is_boundary_sec_down(l)) ⇒
((occupied_without_head(l.D2U) ⇒ occupied_without_tail(hto(**up**(l),l))) ∧
(occupied_without_tail(hto(**up**(l),l)) ⇒ occupied_without_head(l.D2U))))

[train_integrity_linear_down]
(∀l : **Linear** •
(**down**(l) ∧ ¬is_boundary_sec_down(**down**(l)) ∧ ¬is_boundary_sec_up(l)) ⇒
((occupied_without_head(l.U2D) ⇒ occupied_without_tail(hto(**down**(l),l))) ∧
(occupied_without_tail(hto(**down**(l),l)) ⇒ occupied_without_head(l.U2D))))

For a point section *p*, the position of the point is taken into account. For the travel direction from the stem end toward plus/minus ends, the following must hold:

TI-P-1 If *p* is occupied by a train in the travel direction coming from its stem end and the head of the train is not on *p*, then *only one* of the following must hold:

(TI-P-1a) *p*'s actual position is PLUS(0), and *plus(p)* is occupied by a train in the direction coming from *p* and the tail of the train is not on *plus(p)*.

*Only linear sections can be boundary sections, see Chapter 4

(TI-P-1b) *p*'s actual position is MINUS(1), and *minus(p)* is occupied by a train in the direction coming from *p* and the tail of the train is not on *minus(p)*.

TI-P-2 If *plus(p)* (*minus(p)*) is occupied by a train in the travel direction coming from *p* and the tail of the train is not on *plus(p)* (*minus(p)*), then *p*'s actual position must be PLUS(0) (MINUS(0)) and *p* must be occupied by a train in the travel direction coming from its stem end and the head of the train is not on *p*.

TI-P-3 It is not the case that both of the following hold:

(TI-P-3a) *plus(p)* is occupied by a train in the travel direction coming from *p* and the tail of the train is not on *plus(p)*.

(TI-P-3b) *minus(p)* is occupied by a train in the travel direction coming from *p* and the tail of the train is not on *minus(p)*.

```
[train_integrity_point_stem_to_plusminus]
(∀p : Point •
 (occupied_without_head(p.S2PM) ⇒
  ((occupied_without_tail(hto(plus(p),p)) ∧ p.POS = PLUS) ⊕
  (occupied_without_tail(hto(minus(p),p)) ∧ p.POS = MINUS))) ∧
 (occupied_without_tail(hto(plus(p),p)) ⇒
  (occupied_without_head(p.S2PM) ∧ p.POS = PLUS)) ∧
 (occupied_without_tail(hto(minus(p),p)) ⇒
  (occupied_without_head(p.S2PM) ∧ p.POS = MINUS)) ∧
 ¬(occupied_without_tail(hto(plus(p),p)) ∧
  occupied_without_tail(hto(minus(p),p))))
```

For travel direction from the plus/minus ends toward the stem end, the following must hold:

TI-P-4 If *p* is occupied by a train in the travel direction coming from its plus (minus) end and the head of the train is not on *p*, then *p*'s actual position must be PLUS(0) (MINUS(0)) and *stem(p)* must be occupied by a train in the travel direction coming from *p* and the tail of the train is not on *stem(p)*.

TI-P-5 If *stem(p)* is occupied by a train in the travel direction coming from *p* and the tail of the train is not on *stem(p)*, then *only one* of the following must hold:

(TI-P-5a) *p*'s actual position is PLUS(0), and *p* is occupied by a train in the direction coming from its plus end and the head of the train is not on *p*.

(TI-P-5b) *p*'s actual position is MINUS(1), and *p* is occupied by a train in the direction coming from its minus end and the head of the train is not on *p*.

```
[train_integrity_point_plusminus_to_stem]
(∀p : Point •
 (occupied_without_head(p.P2S) ⇒
```

```
  (occupied_without_tail(hto(stem(p),p)) ∧ p.POS = PLUS)) ∧
 (occupied_without_head(p.M2S) ⇒
  (occupied_without_tail(hto(stem(p),p)) ∧ p.POS = MINUS)) ∧
 (occupied_without_tail(hto(stem(p),p)) ⇒
  ((occupied_without_head(p.P2S) ∧ p.POS = PLUS) ⊕
  (occupied_without_head(p.M2S) ∧ p.POS = MINUS))))
```

E.2 Invariants on Routes

SI-R-1 If a route is in ALLOCATING or LOCKED mode, then all of its conflicting routes must not be in ALLOCATING or LOCKED mode.

```
[conflicting_routes_are_not_set_together]
(∀r : Route •
 (r.MODE = ALLOCATING ∨ r.MODE = LOCKED) ⇒
 (∀cr : Route •
  cr ∈ conflicts (r) ⇒ (cr.MODE ≠ ALLOCATING ∧ cr.MODE ≠ LOCKED)))
```

SI-R-2 If a route *r* is in ALLOCATING mode, then all of the following hold

- all points used by *r* are commanded into correct position as required by *r*
- all protecting signals are commanded to be CLOSED
- all sections in *r*'s path are in EXLCK mode
- all sections in *r*'s path and overlap are vacant

```
[route_allocating_cnd]
(∀r : Route •
 (r.MODE = ALLOCATING) ⇒
 /* points are commanded in correct positions */
 ((∀p : Point • p ∈ points(r) ⇒ (p.CMD = req(r,p))) ∧
  /* protecting signals are commanded in correct aspects */
  (∀s : Signal • s ∈ signals(r) ⇒ (s.CMD = CLOSED)) ∧
  /* all lockable elements in the path are EXLCK(1) */
  (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK)) ∧
  /* all sections have to be vacant */
  (∀e : Section •
   e ∈ (elems path(r) ∪ elems overlap(r)) ⇒ vacant(e))))
```

SI-R-3 If a route *r* is in LOCKED mode, the all of the following hold

- the actual positions of all points *e* used by *r* are as required, and there is no potential change, i.e., $e.POS = e.CMD$
- the actual aspect of all protecting signals *e* are as required, and there is no potential change, i.e., $e.ACT = e.CMD$

- all sections in r's path are in EXLCK mode
- all sections in r's path and overlap are vacant, except the first section, which must be vacant or occupied by the head of the train.
- the source signal is commanded to be switched to OPEN

```
[route_lock_cnd]
(∀r : Route •
 (r.MODE = LOCKED) ⇒
 let fst = first(r) in
   /* points are in correct positions */
   (∀p : Point •
    p ∈ points(r) ⇒ (p.POS = req(r,p) ∧ p.POS = p.CMD)) ∧
   /* protecting signals are in correct aspects */
   (∀s : Signal •
    s ∈ signals(r) ⇒ (s.ACT = CLOSED ∧ s.ACT = s.CMD)) ∧
   /* all lockable elements in the path are EXLCK(1) */
   (∀e : Section • e ∈ path(r) ⇒ (e.MODE = EXLCK)) ∧
   /* all sections except the first one have to be vacant */
   (∀e : Section •
    (e ≠ fst ∧ e ∈ (elems path(r) ∪ elems overlap(r))) ⇒
    vacant(e)) ∧
   /* first section is vacant or occupied by head of the train */
   (vacant(fst) ∨ hto(fst,r) = 5) ∧
   /* entry signal is commanded to be open */
   src(r).CMD = OPEN
 end)
```

SI-R-4 If a route r is in OCCUPIED mode, all of the following hold

- the last section of the route is not in FREE mode
- the route is occupied by a train (i.e., starting from the last section, if we go in the opposite direction of the r's direction, then we should meet a section which has the O bit set), or the train has just moved pass the last section to the next route, i.e., the last section *lst* is vacant and $lst.PREV = 1$
- if we divide the route into chunks of linear sections, two chunks are separated by a point, the for each chunk, the following hold: if a chunk is occupied in one direction (up or down) then the other direction must be vacant. This condition eliminates two cases:
  - two trains are moving head-on, which will eventually result in a collision
  - two trains are back-to-back, which is infeasible within a chunk
- if a route is in use, then the last section of the route must be free in the opposite direction w.r.t. the direction of the route.

```
[route_used_cnd_a]
(∀r : Route • r.MODE = OCCUPIED ⇒ (last(r).MODE ≠ AVAIL))
```

```
[route_used_cnd_b]
(∀r : Route •
 r.MODE = OCCUPIED ⇒
 (count_fwd___O(r) ∨
  let fst = first(r),
      lst = last(r) in
    ((lst ≠ fst) ? lst.PREV : (lst.MODE = USED)) ∧ vacant(last(r))
  end))

[route_used_cnd_c]
(∀r : Route • r.MODE = OCCUPIED ⇒ linear_chunk_cnd(r,last(r)))

[route_in_use_last_sec_is_free_in_opposite_dir]
(∀r : Route • r.MODE = OCCUPIED ⇒ bwd_hto(last(r),r,0) = 0)
```

SI-R-5 Two routes that share the last section must not be occupied at the same time.

```
[routes_share_last_not_used_at_same_time]
(∀r : Route •
 r.MODE = OCCUPIED ⇒
 (∀opr : Route •
  (opr ≠ r ∧ last(r) = last(opr)) ⇒ (opr.MODE ≠ OCCUPIED)))
```

SI-R-6 It is never the case that the first section of a route is not vacant, while the source signal is commanded to be open and its actual aspect is closed.

```
[first_entry_cnd]
(∀r : Route •
 ¬(¬vacant(first(r)) ∧ src(r).CMD = OPEN ∧ src(r).ACT = CLOSED))
```

SI-R-7 The current mode of a route displayed to the output interfaces must faithfully represents the current mode of the route.

```
[mode_and_display_are_identical]
(∀r : Route • r.MODE = r.DSPL)
```

E.3 Invariants on Signals

SI-S-1 If a signal *s* is commanded to be opened, then there is exactly one route *r* that is in LOCKED mode, and *s* is *r*'s source signal

```
[signal_cmd_open_cnd]
(∀s : Signal •
 (∃r : Route • s = src(r)) ⇒
 (s.CMD = OPEN ⇒ (∃!r : Route • src(r) = s ∧ r.MODE = LOCKED)))
```

SI-S-2 If a signal s's actual aspect is OPEN, then one of the following holds:

- There is a route that was just canceled, i.e., its control command is CANCEL(2), its mode is FREE(0), and s is commanded to be closed.
- There is exactly one route r that have e as source signal and one of the following holds
  - r is in LOCKED mode, and e is commanded to be opened or the train has occupied the first section
  - r is in OCCUPIED mode, e is commanded to be CLOSED, the train occupies the first section, all sections in r's path, except the first section, are in EXLOCK mode and vacant, and all sections in r's overlap are vacant.

```
[signal_act_open_cnd]
(∀s : Signal •
 (∃r : Route • s = src(r)) ⇒
 (s.ACT = OPEN ⇒
  ((∃r : Route •
    s = src(r) ∧ r.MODE = FREE ∧ s.CMD = CLOSED ∧ r.CTRL = CANCEL) ∨
   (∃!r : Route •
    (src(r) = s) ∧
    (((r.MODE = LOCKED ∧
       (s.CMD = OPEN ∨ H__(hto(first(r),r)) ≠ 0)) ∨
      (r.MODE = OCCUPIED ∧ s.CMD = CLOSED ∧
       H__(hto(first(r), r)) ≠ 0)) ∧
     (∀e : Section •
      (e ∈ path(r) ∧ e ≠ first(r)) ⇒
      (vacant(e) ∧ e.MODE = EXLCK)) ∧
     (∀e : Section • e ∈ overlap(r) ⇒ vacant(e)))))))
```

SI-S-3 When the tail of the train is in the first section, the source signal's actual aspect must be CLOSED

```
[entry_signal_closed_when_tail_in_first]
(∀r : Route • _T_(hto(first(r), r)) ⇒ src(r).ACT = CLOSED)
```

E.4 Invariants on Points

SI-P-1 When a point p is not vacant or is in USED mode, it must not have any potential to move, i.e., $p.POS = p.CMD$

```
[not_commanding_occupied_point_to_move]
(∀p : Point • (¬vacant(p) ∨ p.MODE = USED) ⇒ p.POS = p.CMD)
```

SI-P-2 When a point p is going to switch, i.e., $p.POS \neq p.CMD$, it must not be in USED mode, and there exists a route r that requires p and r is in ALLOCATING mode.

```
[point_only_cmd_when_alloc_a_route]
(∀e : Point •
 (∃r : Route • e ∈ points(r)) ⇒
 (e.CMD ≠ e.POS ⇒
  (e.MODE ≠ USED ∧
   (∃r : Route • e ∈ points(r) ∧ r.MODE = ALLOCATING))))
```

SI-P-3 For a point p, if a train can change direction neither in the neighbouring section at its stem end, nor in the neighbouring section at its plus (minus) end, then if the actual position of p is PLUS(0), only one of the following holds:

- p is occupied by a train in the direction coming from its plus (minus) end, or *stem(p)* is occupied by a train in the direction coming from p.
- p is occupied by a train in the direction coming from its stem end, or *plus(p)* (*minus(p)*) is occupied by a train in the direction coming from p and *plus(p)*'s (*minus(p)*'s) mode is USED(2) and *plus(p)*'s (*minus(p)*'s) PREV variable is not set.

```
[point_chunk_cnd_stem_plus]
(∀e : Point •
 let s = stem(e),
     p = plus(e),
     s_b = hto(stem(e),e,0),
     p_b = hto(plus(e),e,0),
     s_t = e.S2PM,
     p_t = e.P2S in
   (¬can_turn_around_at(s) ∧ ¬can_turn_around_at(p)) ⇒
   ((e.POS = PLUS) ⇒
    ((p_t + s_b) * (s_t + p_b * (¬p.PREV ∧ p.MODE = USED)) = 0))
 end)

[point_chunk_cnd_stem_minus]
(∀e : Point •
 let s = stem(e),
     m = minus(e),
     s_b = hto(stem(e),e,0),
     m_b = hto(minus(e),e,0),
     s_t = e.S2PM,
     m_t = e.M2S in
   (¬can_turn_around_at(s) ∧ ¬can_turn_around_at(m)) ⇒
   ((e.POS = MINUS) ⇒
    ((m_t + s_b) * (s_t + m_b * (¬m.PREV ∧ m.MODE = USED)) = 0))
 end)
```

E.5 Invariants on Elements

SI-E-1 If an element *e* is in EXLCK mode, it must be locked for exactly one route *r* that has *e* in its path and satisfies one of the following

- *r* is in ALLOCATING mode
- *r* is in LOCKED mode
- *r* is in OCCUPIED mode, and all the following hold
  - if *e* is a point, it must be in correct position as required by *r*
  - all *e*'s next sections in *r* must be in EXLCK mode, vacant, and in correct position required by *r* if the section is a point
  - *e*'s previous section in *r* is in EXLCK or USED mode, and in correct position if it is a point

```
[elem_locked_by_only_one_route]
(∀e : Section •
 (∃r : Route • e ∈ path(r)) ⇒
 (e.MODE = EXLCK ⇒
  (∃!r : Route •
   e ∈ path(r) ∧
   let lst = last(r),
       fst = first(r) in
     /* route is allocating, imply all elems locked */
     ((r.MODE = ALLOCATING) ∨
      /* route is locked, imply all elems locked */
      (r.MODE = LOCKED) ∨
      /* route in use, the 1st elem is never locked when
       * the route is in use */
      ((e ≠ fst) ∧ r.MODE = OCCUPIED ∧
       ((e ∈ Point) ⇒ (e.POS = req(r,e) ∧ e.CMD = req(r,e))) ∧
       /* next elems in the same route are locked and vacant */
       (∀ne : Section •
        ne ∈ nexts(r,e) ⇒
        (ne.MODE = EXLCK ∧ vacant(ne) ∧
         ((ne ∈ Point) ⇒
          (ne.POS = req(r,ne) ∧ ne.CMD = req(r,ne))))) ∧
       /* the head of the train is in e
        * then prev is occupied without a head
        * we have e ~= fst, thus pv always exists */
       let pv = prev(r,e) in
         (hto(e,r) = 5 ∧ pv.MODE = USED ∧
          ((pv ∈ Point) ⇒ pv.POS = req(r,pv)) ∧
          occupied_without_head(hto(pv,r))) ∨
         /* e is vacant, then the prev is locked or used */
         (vacant(e) ∧ (pv.MODE = EXLCK ∨ pv.MODE = USED) ∧
          ((pv ∈ Point) ⇒ (pv.POS = req(r,pv) ∧ pv.CMD = req(r,pv))))
```

```
      end))
   end)))
```

SI-E-2 If an element *e* is in USED mode, it must be used by exactly one route *r* that has *e* in its path and all of the following hold

- *r* is in OCCUPIED mode
- if *e* is a point, it must be in correct position as required by *r*
- *e*'s previous section *pv* in *r* has been released, or still locked for *r* and the train is still in *pv* or just left *pv* to *e*.
- all *e*'s next sections *nx* satisfy the following
  - if *nx* is a point, it must be in position as required by *r*
  - one of the following holds
    * *nx* is in EXLCK mode, and *e*'s next sections in *r* are in EXLCK mode
    * *nx* is in USED mode, and the train occupies both *e* and *nx* or the train occupies only *nx* and *e* is going to be released.

```
[elem_used_by_only_one_route]
(∀e : Section •
 (∃r : Route • e ∈ path(r)) ⇒
 (e.MODE = USED ⇒
  (∃!r : Route •
   e ∈ path(r) ∧
   let fst = first(r),
       lst = last(r) in
     r.MODE = OCCUPIED ∧ ((e ∈ Point) ⇒ e.POS = req(r,e)) ∧
     (e ≠ fst ⇒
      let pv = prev(r,e) in
        /* the prev has been released */
        e.PREV ∨
        /* the prev is used without a head
         * or is going to be released */
        (((pv ∈ Point) ⇒ pv.POS = req(r,pv)) ∧ pv.MODE = USED ∧
         (occupied_without_head(hto(pv,r)) ∨
          (vacant(pv) ∧ _T_(hto(e,r)) = 1 ∧ (pv ≠ fst ⇒ pv.PREV))))
      end) ∧
     (e ≠ lst ⇒
      let nx = next(r,e) in
        ¬nx.PREV ∧
        /* this includes nx */
        (∀p : Point •
         p ∈ nexts(r,e) ⇒ (p.POS = req(r,p) ∧ p.CMD = req(r,p))) ∧
        ((nx.MODE = EXLCK ∧
          ((hto(nx,r) = 5 ∧ occupied_without_head(hto(e,r))) ∨
           (vacant(nx) ∧ H__(hto(e,r)) = 1)) ∧
```

```
        (∀ne : Section •
         (ne ≠ nx ∧ ne ∈ nexts(r,e)) ⇒ (ne.MODE = EXLCK))) ∨
       (nx.MODE = USED ∧
        ((occupied_without_head(hto(e,r)) ∧
          occupied_without_tail(hto(nx,r))) ∨
         (vacant(e) ∧ _T_(hto(nx,r)) = 1 ∧ (e ≠ fst ⇒ e.PREV)))))
    end)
  end)))
```

SI-E-3 If a non-boundary element e has $e.PREV = 1$ then it must be in USED mode

```
[element_prev_variable]
(∀e : Section • is_boundary_sec(e) ∨ (e.PREV = RELEASED ⇒ e.MODE = USED))
```

SI-E-4 If an element e is occupied, then it must be in EXLCK or USED mode.

```
[occupied_implies_exlck_or_used_point]
(∀e : Point •
 (∀r : Route • e ≠ last(r)) ⇒
 (¬vacant(e) ⇒ (e.MODE = EXLCK ∨ e.MODE = USED)))

[occupied_implies_exlck_or_used_linear]
(∀e : Linear •
 ((∀r : Route • e ≠ last(r)) ∧ down(e) ∧ up(e)) ⇒
 (¬vacant(e) ⇒ (e.MODE = EXLCK ∨ e.MODE = USED)))
```

E.6 Ground Unused Elements

SI-G-1 If a signal is not a source signal of any route, then it remains closed

```
[ground_unused_signal]
(∀s : Signal •
 (∀r : Route • src(r) ≠ s) ⇒ (s.ACT = CLOSED ∧ s.CMD = CLOSED))
```

SI-G-2 For a travel direction of a section, the section remains vacant if the section is not in the path of any route going in the same direction, and the section is not a boundary section, and there are no routes entering the section from the opposite direction and the train may turn around in the section.

```
[ground_unused_linear_to_down]
(∀l : Linear •
 /* a section is not used by any routes in the direction up */
 (¬is_boundary_sec(l) ∧
  (∀r : Route • l ∈ path(r) ⇒ entry(r,l) ≠ UP) ∧
  /* trains cannot turn around here */
  ¬(can_turn_around_at(l) ∧
```

$(\exists r : \textbf{Route} \bullet l \in \textbf{path}(r) \wedge \textbf{entry}(r,l) = \text{DOWN}))) \Rightarrow$
$(l.\text{U2D} = 0))$

[ground_unused_linear_to_up]
$(\forall l : \textbf{Linear} \bullet$
/* *a section is not used by any routes in the up direction up* */
$(\neg \text{is\_boundary\_sec}(l) \wedge$
$(\forall r : \textbf{Route} \bullet l \in \textbf{path}(r) \Rightarrow \textbf{entry}(r,l) \neq \text{DOWN}) \wedge$
/* *trains cannot turn around here* */
$\neg(\text{can\_turn\_around\_at}(l) \wedge$
$(\exists r : \textbf{Route} \bullet l \in \textbf{path}(r) \wedge \textbf{entry}(r,l) = \text{UP}))) \Rightarrow (l.\text{D2U} = 0))$

[ground_unused_point_p]
$(\forall p : \textbf{Point} \bullet$
$(\neg(\exists r : \textbf{Route} \bullet$
$p \in (\textbf{dom points}(r) \cap \textbf{elems path}(r)) \wedge \textbf{entry}(r,p) = \text{PLUS})) \Rightarrow$
$(p.\text{P2S} = 0))$

[ground_unused_point_m]
$(\forall p : \textbf{Point} \bullet$
$(\neg(\exists r : \textbf{Route} \bullet$
$p \in (\textbf{dom points}(r) \cap \textbf{elems path}(r)) \wedge \textbf{entry}(r,p) = \text{MINUS})) \Rightarrow$
$(p.\text{M2S} = 0))$

[ground_unused_point_s]
$(\forall p : \textbf{Point} \bullet$
$(\neg(\exists r : \textbf{Route} \bullet$
$p \in (\textbf{dom points}(r) \cap \textbf{elems path}(r)) \wedge \textbf{entry}(r,p) = \text{STEM})) \Rightarrow$
$(p.\text{S2PM} = 0))$

SI-G-3 All boundary sections remain vacant since the interlocking under consideration does not control them, and they are initialized vacant.

[ground_unused_boundary_linear]
$(\forall l : \textbf{Linear} \bullet \text{is\_boundary\_sec}(l) \Rightarrow (l.\text{U2D} = 0 \wedge l.\text{D2U} = 0))$

Bibliography

[AT12] Morten Aanæs and Hoang Phuong Thai. "Modelling and Verification of Relay Interlocking Systems". Series: IMM-MSC-2012-14. Master's thesis. Technical University of Denmark, DTU Informatics, 2012. URL: http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6262.

[Ban10] Banedanmark. *The Signaling Programme - A Total Renewal of the Danish Signaling Infrastruture*. 2010. ISBN: 978-87-90682-04-0.

[Ban12] Banedanmark. "Appendix 3.3 - Common Engineering Rules". In: *Fjernbane Infrastructure East* (2012).

[Ban14] Banedanmark. *A Programme for Total Replacement*. 2014. URL: http://uk.bane.dk/visArtikel_eng.asp?artikelID=6090.

[Bar+04] Howard Barringer et al. "Rule-Based Runtime Verification". In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*. Edited by Bernhard Steffen and Giorgio Levi. Volume 2937. Lecture Notes in Computer Science. Springer, 2004, pages 44–57. ISBN: 3-540-20803-8. DOI: 10.1007/978-3-540-24622-0_5. URL: http://dx.doi.org/10.1007/978-3-540-24622-0_5.

[BF05] Michele Banci and Alessandro Fantechi. "Geographical Versus Functional Modelling by Statecharts of Interlocking Systems". In: *Electr. Notes Theor. Comput. Sci.* 133 (2005), pages 3–19. DOI: 10.1016/j.entcs.2004.08.055. URL: http://dx.doi.org/10.1016/j.entcs.2004.08.055.

[BF14] Andrea Bonacchi and Alessandro Fantechi. "Validation of Interlocking Systems by Testing their Models". In: *9th International Conference on the Quality of Information and Communications Technology, QUATIC 2014, Guimaraes, Portugal, September 23-26, 2014*. IEEE, 2014, pages 226–229. ISBN: 978-1-4799-6132-0. DOI: 10.1109/QUATIC.2014.37. URL: http://dx.doi.org/10.1109/QUATIC.2014.37.

[BFG05] Michele Banci, Alessandro Fantechi, and S Ginesi. "Some Experiences on Formal Specification of Railway Interlocking Systems using Statecharts". In: *Train International Workshop at SEFM*. 2005.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. "Seven More Myths of Formal Methods". In: *IEEE Software* 12.4 (1995), pages 34–41. DOI: 10.1109/52.391826. URL: http://doi.ieeecomputersociety.org/10.1109/52.391826.

[Bie+06] Armin Biere et al. "Linear Encodings of Bounded LTL Model Checking". In: *Logical Methods in Computer Science* 2.5 (November 2006). arXiv: cs/0611029. ISSN: 18605974. DOI: 10.2168/LMCS-2(5:5)2006. URL: http://arxiv.org/abs/cs/0611029 (visited on June 26, 2014).

[Bie+99a] Armin Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. Edited by Rance Cleaveland. Volume 1579. Lecture Notes in Computer Science. Springer, 1999, pages 193–207. ISBN: 3-540-65703-7. DOI: 10.1007/3-540-49059-0_14. URL: http://dx.doi.org/10.1007/3-540-49059-0_14.

[Bie+99b] Armin Biere et al. "Verifying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*. Edited by Nicolas Halbwachs and Doron Peled. Volume 1633. Lecture Notes in Computer Science. Springer, 1999, pages 60–71. ISBN: 3-540-66202-2. DOI: 10.1007/3-540-48683-6_8. URL: http://dx.doi.org/10.1007/3-540-48683-6_8.

[Bjø03] D. Bjørner. "New Results and Trends in Formal Techniques and Tools for the Development of Software for Transportation Systems-A Review". In: *FORMS2003: Symposium on Formal Methods for Railway Operation and Control Systems. LHarmattan Hongrie. Conf. held at Techn. Univ. of Budapest, Hungary. Editors: G. Tarnai and E. Schnieder, Germany*. 2003.

[Bjø04] Dines Bjørner. "TRain: The Railway domain - A "Grand Challenge" for Computing Science & Transportation Engineering". In: *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*. Edited by René Jacquart. Volume 156. IFIP. Kluwer/Springer, 2004, pages 607–611. ISBN: 1-4020-8156-1. DOI: 10.1007/978-1-4020-8157-6_58. URL: http://dx.doi.org/10.1007/978-1-4020-8157-6_58.

[BK08] Christel. Baier and Joost-Pieter. Katoen. *Principles of model checking*. MIT Press, 2008, 975 s. ISBN: 9780262026499.

[BMS93] J Berger, P Middelraad, and AJ Smith. "EURIS, European railway interlocking specification". In: *UIC, Commission A* 7 (1993), page 16.

[Bon+12] Andrea Bonacchi et al. "A GUI Testability Problem: A Case Study in the Railway Signaling Domain". In: *8th International Conference on the Quality of Information and Communications Technology, QUATIC 2012, Lisbon, Portugal, 2-6 September 2012, Proceedings*. Edited by João Pascoal Faria, Alberto Rodrigues da Silva, and Ricardo Jorge Machado. IEEE Computer Society, 2012, pages 103–107. ISBN: 978-0-7695-4777-0. DOI: 10.

1109/QUATIC.2012.10. URL: http://dx.doi.org/10.1109/QUATIC.2012.10.

[Bon+13] Andrea Bonacchi et al. "Validation of Railway Interlocking Systems by Formal Verification, A Case Study". In: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*. Edited by Steve Counsell and Manuel Núñez. Volume 8368. Lecture Notes in Computer Science. Springer, 2013, pages 237–252. ISBN: 978-3-319-05031-7. DOI: 10.1007/978-3-319-05032-4_18. URL: http://dx.doi.org/10.1007/978-3-319-05032-4_18.

[Bon13] Andrea Bonacchi. "Formal safety proof: a real case study in a railway interlocking system". In: *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. Edited by Mauro Pezzè and Mark Harman. ACM, 2013, pages 378–381. ISBN: 978-1-4503-2159-4. DOI: 10.1145/2483760.2492398. URL: http://doi.acm.org/10.1145/2483760.2492398.

[Boz+14] Marco Bozzano et al. *nuXmv 1.0 User Manual*. 2014. URL: https://nuxmv.fbk.eu/.

[BR70] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[Bra+14a] Cécile Braunstein et al. *A SysML Test Model and Test Suite for the ETCS Ceiling Speed Monitor*. Technical report. Embedded Systems Testing Benchmarks Site, 2014. URL: http://www.mbt-benchmarks.org.

[Bra+14b] Cécile Braunstein et al. "Complete Model-Based Equivalence Class Testing for the ETCS Ceiling Speed Monitor". In: *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*. Edited by Stephan Merz and Jun Pang. Volume 8829. Lecture Notes in Computer Science. Springer, 2014, pages 380–395. ISBN: 978-3-319-11736-2. DOI: 10.1007/978-3-319-11737-9_25. URL: http://dx.doi.org/10.1007/978-3-319-11737-9_25.

[Bra11] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. Edited by Ranjit Jhala and David A. Schmidt. Volume 6538. Lecture Notes in Computer Science. Springer, 2011, pages 70–87. ISBN: 978-3-642-18274-7. DOI: 10.1007/978-3-642-18275-4_7. URL: http://dx.doi.org/10.1007/978-3-642-18275-4_7.

[Bro+05] Manfred Broy et al., editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*. Volume 3472. Lecture Notes in Computer Science. Springer, 2005. ISBN: 3-540-26278-4.

[Bur+92] Jerry R. Burch et al. "Symbolic Model Checking: 10^{20} States and Beyond". In: *Inf. Comput.* 98.2 (1992), pages 142–170. DOI: 10.1016/0890-5401(92)90017-A. URL: http://dx.doi.org/10.1016/0890-5401(92)90017-A.

[Cal+06] Jens R. Calame et al. "TTCN-3 Testing of Hoorn-Kersenboogerd Railway Interlocking". In: *Proceedings of the Canadian Conference on Electrical and Computer Engineering, CCECE 2006, May 7-10, 2006, Ottawa Congress Centre, Ottawa, Canada*. IEEE, 2006, pages 620–623. DOI: 10.1109/CCECE.2006.277762. URL: http://dx.doi.org/10.1109/CCECE.2006.277762.

[Cao+11] Yan Cao et al. "Automatic Generation and Verification of Interlocking Tables Based on Domain Specific Language for Computer Based Interlocking Systems (DSL-CBI)". In: *Proceedings of the IEEE International Conference on Computer Science and Automation Engineering (CSAE 2011)*. IEEE, 2011, pages 511–515.

[Cav+14] Roberto Cavada et al. "The NuXmv Symbolic Model Checker". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Edited by Armin Biere and Roderick Bloem. Volume 8559. Lecture Notes in Computer Science. Springer, 2014, pages 334–342. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9_22. URL: http://dx.doi.org/10.1007/978-3-319-08867-9_22.

[CEN12] E.N. CENELEC. "50128 - Railway Applications-Communication, Signalling and Processing Systems-Software for Railway Control and Protection Systems". In: *Book EN 50128 Railway Application-Communications, signalling and processing systems-Software for railway control and protection systems* (2012).

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT Press, 2000, 314 s. ISBN: 0262032708, 9780262032704.

[Cim+02] Alessandro Cimatti et al. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. Edited by Ed Brinksma and Kim Guldstrand Larsen. Volume 2404. Lecture Notes in Computer Science. Springer, 2002, pages 359–364. ISBN: 3-540-43997-8. DOI: 10.1007/3-540-45657-0_29. URL: http://dx.doi.org/10.1007/3-540-45657-0_29.

[Cla+03] Edmund M. Clarke et al. "Counterexample-guided Abstraction Refinement for Symbolic Model Checking". In: *J. ACM* 50.5 (2003), pages 752–794. DOI: 10.1145/876638.876643. URL: http://doi.acm.org/10.1145/876638.876643.

[CN14] Steve Counsell and Manuel Núñez, editors. *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*. Volume 8368. Lecture Notes in Computer Science. Springer, 2014. ISBN: 978-3-319-05031-7. DOI: 10.1007/978-3-319-05032-4. URL: http://dx.doi.org/10.1007/978-3-319-05032-4.

[CO84] G. B. Clemmensen and Ole N. Oest. "Formal Specification and Development of an Ada Compiler - A VDM Case Study". In: *Proceedings, 7th International Conference on Software Engineering, Orlando, Florida, USA, March 26-29, 1984*. Edited by Terry A. Straeter, William E. Howden, and Jean-Claude Rault. IEEE Computer Society, 1984, pages 430–440. ISBN: 0-8186-0528-6. URL: http://dl.acm.org/citation.cfm?id=802002.

[Col15] Michael Collins. *Formal Methods*. 2015. URL: http://users.ece.cmu.edu/~koopman/des_s99/formal_methods/.

[Con+07] Camille Constant et al. "Integrating formal verification and conformance testing for reactive systems". In: *IEEE Trans. Software Eng.* 33.8 (2007), pages 558–574. DOI: 10.1109/TSE.2007.70707. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70707.

[CW96] Edmund M. Clarke and Jeannette M. Wing. "Formal Methods: State of the Art and Future Directions". In: *ACM Comput. Surv.* 28.4 (1996), pages 626–643. DOI: 10.1145/242223.242257. URL: http://doi.acm.org/10.1145/242223.242257.

[Dij+98] Fokko van Dijk et al. "EURIS, a Specification Method for Distributed Interlockings". In: *Computer Safety, Reliability and Security, 17th International Conference, SAFECOMP'98, Heidelberg, Germany, October 5-7, 1998, Proceedings*. Edited by Wolfgang D. Ehrenberger. Volume 1516. Lecture Notes in Computer Science. Springer, 1998, pages 296–305. ISBN: 3-540-65110-1. DOI: 10.1007/3-540-49646-7_23. URL: http://dx.doi.org/10.1007/3-540-49646-7_23.

[Duf91] David A. Duffy. *Principles of automated theorem proving*. Wiley professional computing. Wiley, 1991. ISBN: 978-0-471-92784-6.

[EE04] Institute of Electrical and Electronics Engineers. "IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements". In: *IEEE Std 1474.1-2004 (Revision of IEEE Std 1474.1-1999)* (2004), pages 1–45. DOI: 10.1109/IEEESTD.2004.95746.

[Ell+01] John Ellson et al. "Graphviz - Open Source Graph Drawing Tools". In: *Graph Drawing*. 2001, pages 483–484. DOI: 10.1007/3-540-45848-4_57. URL: http://dx.doi.org/10.1007/3-540-45848-4_57.

[ERT14] ERTMS. *Annex A for ETCS Baseline 3 and GSM-R Baseline 0*. European Railway Agency, April 2014. URL: http://www.era.europa.eu/Document-Register/Pages/New-Annex-A-for-ETCS-Baseline-3-and-GSM-R-Baseline-0.aspx.

[Eur01] European Commission. "Commission Decision 2001/260/EC of 21 March 2001 on the basic parameters of the command-control and signalling sub- system of the trans-European high-speed rail system referred to as 'ERTMS characteristics' in Annex II(3) to Directive 96/48/EC". In: *Official Journal of the European Union* L.93 (2001), pages 53–56.

[Eur96] European Council. "Council Directive 96/48/EC of 23 July 1996 on the interoperability of the trans-European high-speed rail system". In: *Official Journal of the European Communities* L.235 (1996), pages 6–24.

[Fan12a] Alessandro Fantechi. "Distributing the Challenge of Model Checking Interlocking Control Tables". In: *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*. Edited by Tiziana Margaria and Bernhard Steffen. Volume 7610. Lecture Notes in Computer Science. Springer, 2012, pages 276–289. ISBN: 978-3-642-34031-4. DOI: 10.1007/978-3-642-34032-1. URL: http://dx.doi.org/10.1007/978-3-642-34032-1.

[Fan12b] Alessandro Fantechi. "The Role of Formal Methods in Software Development for Railway Applications". In: *Railway Safety, Reliability, and Security: Technologies and Systems Engineering* (2012). Edited by IGI Global, pages 282–297. DOI: 10.4018/978-1-4666-1643-1.ch012. URL: http://www.igi-global.com/chapter/role-formal-methods-software-development/66676.

[Fan14] Alessandro Fantechi. "Twenty-Five Years of Formal Methods and Railways: What Next?" In: *Software Engineering and Formal Methods*. Edited by Steve Counsell and Manuel Núñez. Volume 8368. Lecture Notes in Computer Science. Springer, 2014, pages 167–183.

[Fer+10] Alessio Ferrari et al. "Model Checking Interlocking Control Tables". In: *FORMS/FORMAT 2010 – Formal Methods for Automation and Safety in Railway and Automotive Systems*. Edited by Eckehard Schnieder and Géza Tarnai. Springer, 2010, pages 107–115. ISBN: 978-3-642-14260-4.

[Fer+11] Alessio Ferrari et al. "Adoption of Model-Based Testing and Abstract Interpretation by a Railway Signalling Manufacturer". In: *IJERTCS* 2.2 (2011), pages 42–61. DOI: 10.4018/jertcs.2011040103. URL: http://dx.doi.org/10.4018/jertcs.2011040103.

[Fer+13] Alessio Ferrari et al. "Model-Based Development and Formal Methods in the Railway Industry". In: *IEEE Software* 30.3 (2013), pages 28–34. DOI: 10.1109/MS.2013.44. URL: http://doi.ieeecomputersociety.org/10.1109/MS.2013.44.

[FFM12] Alessandro Fantechi, Wan Fokkink, and Angelo Morzenti. "Some Trends in Formal Methods Applications to Railway Signaling". In: *Formal Methods for Industrial Critical Systems: A Survey of Applications*. John Wiley Sons, Inc., 2012, pages 61–84. ISBN: 9781118459898. DOI: 10.1002/9781118459898.ch4. URL: http://dx.doi.org/10.1002/9781118459898.ch4.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving, Second Edition*. Graduate Texts in Computer Science. Springer, 1996. ISBN: 978-1-4612-7515-2. DOI: 10.1007/978-1-4612-2360-3. URL: http://dx.doi.org/10.1007/978-1-4612-2360-3.

[Fol15] Andreas Foldager. "A Graphical Domain-specific Language for Railway Interlocking Systems". Master's thesis. Technical University of Denmark, DTU Compute, 2015.

[FWA09] Gordon Fraser, Franz Wotawa, and Paul Ammann. "Testing with model checkers: a survey". In: *Softw. Test., Verif. Reliab.* 19.3 (2009), pages 215–261. DOI: 10.1002/stvr.402. URL: http://dx.doi.org/10.1002/stvr.402.

[Geo+95] Chris W George et al. *The RAISE Method*. 1995.

[Geo02] Chris George. "The Development of the RAISE Tools". In: *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*. Edited by Bernhard K. Aichernig and T. S. E. Maibaum. Volume 2757. Lecture Notes in Computer Science. Springer, 2002, pages 49–64. ISBN: 3-540-20527-6. DOI: 10.1007/978-3-540-40007-3_4. URL: http://dx.doi.org/10.1007/978-3-540-40007-3_4.

[Geo04] Chris George. "Tutorial on the RAISE Language, Method and Tools". In: *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*. Edited by Jim Davies, Wolfram Schulte, and Michael Barnett. Volume 3308. Lecture Notes in Computer Science. Springer, 2004, pages 3–4. ISBN: 3-540-23841-7. DOI: 10.1007/978-3-540-30482-1_2. URL: http://dx.doi.org/10.1007/978-3-540-30482-1_2.

[Geo08] Chris George. "RAISE tool user guide". In: *UNU/IIST, Macau, Tech. Rep* 227 (2008).

[Gro92] RAISE Language Group. *The RAISE Specification Language*. Prentice Hall, 1992.

[Hal90] Anthony Hall. "Seven Myths of Formal Methods". In: *IEEE Software* 7.5 (1990), pages 11–19. DOI: 10.1109/52.57887. URL: http://doi.ieeecomputersociety.org/10.1109/52.57887.

[Ham94] Richard Hamlet. "Random testing". In: *Encyclopedia of software Engineering* (1994).

[Han15] Jacob Bøgelund Hansen. "A Formal Specification Language for Generic Railway Control Systems". Master's thesis. Technical University of Denmark, DTU Compute, 2015.

[Hav15] Klaus Havelund. "Rule-based runtime verification revisited". In: *STTT* 17.2 (2015), pages 143–170. DOI: 10.1007/s10009-014-0309-2. URL: http://dx.doi.org/10.1007/s10009-014-0309-2.

[Hax10] Anne E Haxthausen. "An Introduction to Formal Methods for the Development of Safety-critical Applications". In: *Technical University of Denmark* (2010). URL: http://www2.imm.dtu.dk/courses/02263/F15/Files/FormalMethodsNoteTS.pdf.

[Hax12] Anne Elisabeth Haxthausen. "Automated Generation of Safety Requirements from Railway Interlocking Tables". In: *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*. Edited by Tiziana Margaria and Bernhard Steffen. Volume 7610. Lecture Notes in Computer Science. Springer, 2012, pages 261–275. ISBN: 978-3-642-34031-4. DOI: 10.1007/978-3-642-34032-1_25. URL: http://dx.doi.org/10.1007/978-3-642-34032-1_25.

[Hax14] Anne Elisabeth Haxthausen. "Automated generation of formal safety conditions from railway interlocking tables". In: *STTT* 16.6 (2014), pages 713–726. DOI: 10.1007/s10009-013-0295-9. URL: http://dx.doi.org/10.1007/s10009-013-0295-9.

[HBK10] Anne E. Haxthausen, Marie Le Bliguet, and Andreas A. Kjær. "Modelling and Verification of Relay Interlocking Systems". In: *15th Monterey Workshop: Foundations of Computer Software, Future Trends and Techniques for Development*. Edited by Christine Choppy and Oleg Sokolsky. Lecture Notes in Computer Science 6028. Invited paper. Springer, 2010, pages 141–153.

[HCD04] Anne E. Haxthausen, Nikolaj Christensen, and Rasmus Dyhrberg. "From Domain Model to Domain-specific Language for Railway Control Systems". In: *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004), Braunschweig*. 2004. URL: http://www2.imm.dtu.dk/pubdb/p.php?3439.

[HHP15] Felix Hübner, Wen-ling Huang, and Jan Peleska. "Experimental Evaluation of a Novel Equivalence Class Partition Testing Strategy". In: *Tests and Proofs - 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015. Proceedings*. Edited by Jasmin Christian Blanchette and Nikolai Kosmatov. Volume 9154. Lecture Notes in Computer Science. Springer, 2015, pages 155–172. ISBN: 978-3-319-21214-2. DOI: 10.1007/978-3-319-21215-9_10. URL: http://dx.doi.org/10.1007/978-3-319-21215-9_10.

[HKB11] Anne Elisabeth Haxthausen, Andreas A. Kjær, and Marie Le Bliguet. "Formal Development of a Tool for Automated Modelling and Verification of Relay Interlocking Systems". In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Edited by Michael J. Butler and Wolfram Schulte. Volume 6664. Lecture Notes in Computer Science. Springer, 2011, pages 118–132. ISBN: 978-3-642-21436-3. DOI: 10.1007/978-3-642-21437-0_11. URL: http://dx.doi.org/10.1007/978-3-642-21437-0_11.

[HP00] Anne E. Haxthausen and Jan Peleska. "Formal Development and Verification of a Distributed Railway Control Systems". In: *IEEE Transactions on Software Engineering*. Volume 26. 8. IEEE, 2000, pages 687–701.

[HP13a] Anne E. Haxthausen and Jan Peleska. "Efficient Development and Verification of Safe Railway Control Software". In: *Railways: Types, Design and Safety Issues*. Nova Science Publishers, Inc., 2013, pages 127–148.

[HP13b] Wen-ling Huang and Jan Peleska. "Exhaustive Model-Based Equivalence Class Testing". In: *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*. Edited by Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich. Volume 8254. Lecture Notes in Computer Science. Springer, 2013, pages 49–64. ISBN: 978-3-642-41706-1. DOI: 10.1007/978-3-642-41707-8_4. URL: http://dx.doi.org/10.1007/978-3-642-41707-8_4.

[HP14] Wen-ling Huang and Jan Peleska. "Complete model-based equivalence class testing". In: *International Journal on Software Tools for Technology Transfer* (2014), pages 1–19.

[HP15] Anne Elisabeth Haxthausen and Jan Peleska. "Model Checking and Model-Based Testing in the Railway Domain". In: *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*. Edited by Rolf Drechsler and Ulrich Kühne. Springer, 2015, pages 82–121. ISBN: 978-3-658-09993-0. DOI: 10.1007/978-3-658-09994-7_4. URL: http://dx.doi.org/10.1007/978-3-658-09994-7_4.

[HPK11] Anne Elisabeth Haxthausen, Jan Peleska, and Sebastian Kinder. "A formal approach for the construction and verification of railway control systems". In: *Formal Aspects of Computing* 23.2 (2011), pages 191–219. DOI: 10.1007/s00165-009-0143-6. URL: http://dx.doi.org/10.1007/s00165-009-0143-6.

[HPP14] Anne E. Haxthausen, Jan Peleska, and Ralf Pinger. "Applied Bounded Model Checking for Interlocking System Designs". In: *Software Engineering and Formal Methods*. Edited by Steve Counsell and Manuel Núñez. Volume 8368. Lecture Notes in Computer Science. Springer, 2014, pages 205–220.

[Jam+13] Phillip James et al. "Verification of Solid State Interlocking Programs". In: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*. Edited by Steve Counsell and Manuel Núñez. Volume 8368. Lecture Notes in Computer Science. Springer, 2013, pages 253–268. ISBN: 978-3-319-05031-7. DOI: 10.1007/978-3-319-05032-4_19. URL: http://dx.doi.org/10.1007/978-3-319-05032-4_19.

[Jam+14] Philip James et al. "Verification of Scheme Plans Using CSP||B". In: *Software Engineering and Formal Methods*. Edited by Steve Counsell and Manuel Núñez. Volume 8368. Lecture Notes in Computer Science. Springer, 2014, pages 189–204.

[Jam14] Phillip James. "Designing Domain Specific Languages for Verification and Applications to the Railway Domain". PhD thesis. Ph. D. thesis, Swansea University, 2014.

[JR11] Phillip James and Markus Roggenbach. "Automatically Verifying Railway Interlockings Using SAT-based Model Checking". In: *Electronic Communications of the EASST*. Volume 35. EASST, 2011.

[JR14] Phillip James and Markus Roggenbach. "Encapsulating Formal Methods within Domain Specific Languages: A Solution for Verifying Railway Scheme Plans". In: *Mathematics in Computer Science* 8.1 (2014), pages 11–38. DOI: 10.1007/s11786-014-0174-0. URL: http://dx.doi.org/10.1007/s11786-014-0174-0.

[KMS09] Karim Kanso, Faron Moller, and Anton Setzer. "Automated Verification of Signalling Principles in Railway Interlocking Systems". In: *Electr. Notes Theor. Comput. Sci.* 250.2 (2009), pages 19–31. DOI: 10.1016/j.entcs.2009.08.015. URL: http://dx.doi.org/10.1016/j.entcs.2009.08.015.

[Knu64] Donald E. Knuth. "backus normal form vs. Backus Naur form". In: *Commun. ACM* 7.12 (1964), pages 735–736. DOI: 10.1145/355588.365140. URL: http://doi.acm.org/10.1145/355588.365140.

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pages 107–115. DOI: 10.1145/1538788.1538814. URL: http://doi.acm.org/10.1145/1538788.1538814.

[LS09] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *J. Log. Algebr. Program.* 78.5 (2009), pages 293–303. DOI: 10.1016/j.jlap.2008.08.004. URL: http://dx.doi.org/10.1016/j.jlap.2008.08.004.

[Mew10] Kirsten Mewes. *Domain-specific Modelling of Railway Control Systems with Integrated Verification and Validation*. Verlag Dr. Hut, 2010.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Comput. Surv.* 37.4 (2005), pages 316–344. DOI: 10.1145/1118890.1118892. URL: http://doi.acm.org/10.1145/1118890.1118892.

[MRS03] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. "Bounded Model Checking and Induction: From Refutation to Verification". In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Edited by Warren A. Hunt Jr. and Fabio Somenzi. Volume 2725. Lecture Notes in Computer Science. Springer, 2003, pages 14–26. ISBN: 3-540-40524-0. DOI: 10.1007/978-3-540-45069-6_2. URL: http://dx.doi.org/10.1007/978-3-540-45069-6_2.

[MS12] Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*. Volume 7610. Lecture Notes in Computer Science. Springer, 2012. ISBN: 978-3-642-34031-4. DOI: 10.1007/978-3-642-34032-1. URL: http://dx.doi.org/10.1007/978-3-642-34032-1.

[MY09] Ahmad Mirabadi and Mohammad Bemani Yazdi. "Automatic Generation and Verification of Railway Interlocking Control Tables Using FSM and NuSMV". In: *Transportation Problems* (2009), pages 103–110.

[PE09] R.D. Pascoe and T.N. Eichorn. "What is communication-based train control?" In: *Vehicular Technology Magazine, IEEE* 4.4 (December 2009), pages 16–21. ISSN: 1556-6072. DOI: 10.1109/MVT.2009.934665.

[Pel13] Jan Peleska. "Industrial-Strength Model-Based Testing - State of the Art and Current Challenges". In: *Proceedings 8th Workshop on Model-Based Testing, Rome, Italy*. Edited by Alexander K. Petrenko and Holger Schlingloff. Volume 111. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2013, pages 3–28.

[PG07] Juan Ignacio Perna and Chris George. "Model Checking RAISE Applicative Specifications". In: *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*. Edited by M. Hinchey and T. Margaria. IEEE Computer Society, 2007, pages 257–268. ISBN: 978-0-7695-2884-7. DOI: 10.1109/SEFM.2007.25. URL: http://doi.ieeecomputersociety.org/10.1109/SEFM.2007.25.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. "Automated Test Case Generation with SMT-Solving and Abstract Interpretation". In: *NASA Formal Methods*. Edited by Mihaela Gheorghiu Bobaru et al. Volume 6617. Lecture Notes in Computer Science. Springer, 2011, pages 298–312. ISBN: 978-3-642-20397-8.

[PYB96] Alexandre Petrenko, Nina Yevtushenko, and Gregor von Bochmann. "Fault Models for Testing in Context". In: *Formal Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI, Kaiserslautern, Germany, 8-11 October 1996*. Edited by Reinhard Gotzhein and Jan Bredereke. Volume 69. IFIP Conference Proceedings. Chapman & Hall, 1996, pages 163–178. ISBN: 0-412-79490-X.

[Rus+04] Vlad Rusu et al. "From Safety Verification to Safety Testing". In: *Testing of Communicating Systems, 16th IFIP International Conerence, TestCom 2004, Oxford, UK, March 17-19, 2004, Proceedings*. Edited by Roland Groz and Robert M. Hierons. Volume 2978. Lecture Notes in Computer Science. Springer, 2004, pages 160–176. ISBN: 3-540-21219-1. DOI: 10.1007/978-3-540-24704-3_11. URL: http://dx.doi.org/10.1007/978-3-540-24704-3_11.

[Sni15] Aleksander Sniady. "Communication Technologies Support to Railway Infrastructure and Operations". PhD thesis. 2015. DOI: 10.11581/DTU:00000010.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*. Edited by Warren A. Hunt Jr. and Steven D. Johnson. Volume 1954. Lecture Notes in Computer Science. Springer, 2000, pages 108–125. ISBN: 3-540-41219-0. DOI: 10.1007/3-540-40922-X_8. URL: http://dx.doi.org/10.1007/3-540-40922-X_8.

[Sys15] The European Rail Traffic Management System. *ERTMS Deployment Outside Europe – ERTMS As A Global Standard*. 2015. URL: http://www.ertms.net/wp-content/uploads/2014/09/ERTMS_Factsheet_7_ERTMS_deployment_outside_Europe.pdf.

[Tea00] R Core Team. *R Language Definition*. 2000.

[Tre08] Jan Tretmans. "Model Based Testing with Labelled Transition Systems". In: *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. Edited by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Volume 4949. Lecture Notes in Computer Science. Springer, 2008, pages 1–38. ISBN: 978-3-540-78916-1. DOI: 10.1007/978-3-540-78917-8_1. URL: http://dx.doi.org/10.1007/978-3-540-78917-8_1.

[Tre11] Jan Tretmans. "Model-Based Testing and Some Steps towards Test-Based Modelling". In: *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. Edited by Marco Bernardo and Valérie Is-

sarny. Volume 6659. Lecture Notes in Computer Science. Springer, 2011, pages 297–326. ISBN: 978-3-642-21454-7. DOI: 10.1007/978-3-642-21455-4_9. URL: http://dx.doi.org/10.1007/978-3-642-21455-4_9.

[Tre96a] Jan Tretmans. "Conformance testing with labelled transition systems: Implementation relations and test generation". In: *Computer Networks and {ISDN} Systems* 29.1 (1996). Protocol Testing, pages 49–79. ISSN: 0169-7552. DOI: http://dx.doi.org/10.1016/S0169-7552(96)00017-7. URL: http://www.sciencedirect.com/science/article/pii/S0169755296000177.

[Tre96b] Jan Tretmans. "Test Generation with Inputs, Outputs and Repetitive Quiescence". In: *Software - Concepts and Tools* 17.3 (1996), pages 103–120.

[Tre99] Jan Tretmans. "Testing Concurrent Systems: A Formal Approach". In: *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*. Edited by Jos C. M. Baeten and Sjouke Mauw. Volume 1664. Lecture Notes in Computer Science. Springer, 1999, pages 46–65. ISBN: 3-540-66425-4. DOI: 10.1007/3-540-48320-9_6. URL: http://dx.doi.org/10.1007/3-540-48320-9_6.

[TRN02] David Tombs, Neil Robinson, and George Nikandros. "Signalling Control Table Generation and Verification". In: *CORE 2002: Cost Efficient Railways through Engineering*. Railway Technical Society of Australasia/Rail Track Association of Australia, 2002, page 415.

[TVA09] Gregor Theeg, Sergeĭ Valentinovich Vlasenko, and Enrico Anders. *Railway Signalling & Interlocking: International Compendium*. Eurailpress, 2009.

[Ver15] Verified Systems International GmbH. *RT-Tester Model-Based Test Case and Test Data Generator - RTT-MBT - User Manual*. Verified Systems International GmbH, 2015.

[VHP14a] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "A Domain-Specific Language for Railway Interlocking Systems". In: *FORMS/FORMAT 2014 - 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*. Edited by Eckehard Schnieder and Géza Tarnai. Best paper award. Institute for Traffic Safety and Automation Engineering, Technische Universität Braunschweig., 2014, pages 200–209. ISBN: 978-3-9816886-6-5.

[VHP14b] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "Formal Verification of the Danish Interlocking Systems". In: *AVoCS 2014 Pre-proceedings*. University of Twente, 2014.

[VHP15] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release". English. In: *Formal Techniques for Safety-Critical Systems*. Edited by Cyrille Artho and Peter Csaba Ölveczky. Volume 476. Communications in Computer and Information Science. Springer International Publishing, 2015,

pages 223–238. ISBN: 978-3-319-17580-5. DOI: 10.1007/978-3-319-17581-2_15. URL: http://dx.doi.org/10.1007/978-3-319-17581-2_15.

[VHPss] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. "Formal Modeling and Verification of Interlocking Systems Featuring Sequential Release". In: *Science of Computer Programming - Special Issue: Formal Techniques for Safety-Critical Systems* (Under minor revision process).

[Win+06] K. Winter et al. "Tool Support for Checking Railway Interlocking Designs". In: *Proceedings of the 10th Australian workshop on Safety Critical Systems and Software - Volume 55*. SCS '05. Sydney, Australia: Australian Computer Society, Inc., 2006, pages 101–107. ISBN: 1-920-68237-6. URL: http://dl.acm.org/citation.cfm?id=1151816.1151827.

[Win12] Kirsten Winter. "Optimising Ordering Strategies for Symbolic Model Checking of Railway Interlockings". In: *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*. Edited by Tiziana Margaria and Bernhard Steffen. Volume 7610. Lecture Notes in Computer Science. Springer, 2012, pages 246–260. ISBN: 978-3-642-34031-4. DOI: 10.1007/978-3-642-34032-1. URL: http://dx.doi.org/10.1007/978-3-642-34032-1.

[Woo+09] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM Comput. Surv.* 41.4 (2009). DOI: 10.1145/1592434.1592436. URL: http://doi.acm.org/10.1145/1592434.1592436.

[ZB14] Bahram Zarrin and Hubert Baumeister. "Design of a Domain-Specific Language for Material Flow Analysis using Microsoft DSL tools: An Experience Paper". In: *Proceedings of the 14th Workshop on Domain-Specific Modeling (DSM '14)*. 2014, pages 23–28. ISBN: 978-1-4503-2156-3. DOI: 10.1145/2688447.2688452.