An xDSL-Based Framework for Validation of Railway Models : Application to ERTMS/ETCS and EULYNX

Asfand Yar

▶ To cite this version:

Asfand Yar. An xDSL-Based Framework for Validation of Railway Models : Application to ERTMS/ETCS and EULYNX. Mathematical Software [cs.MS]. Université Grenoble Alpes [2020-..], 2023. English. NNT : 2023GRALM090 . tel-04675703

HAL Id: tel-04675703
https://theses.hal.science/tel-04675703v1

Submitted on 22 Aug 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Un cadre basé sur les DSL exécutables pour la validation des modèles ferroviaires : Application à ERTMS/ETCS et EULYNX

An xDSL-Based Framework for Validation of Railway Models: Application to ERTMS/ETCS and EULYNX

Présentée par :

Asfand YAR

Direction de thèse :

Yves LEDRU PROFESSEUR, Université Grenoble Alpes	Directeur de thèse
Akram IDANI MAITRE DE CONFERENCES, Université Grenoble Alpes	Co-directeur de thèse
Simon COLLART-DUTILLEUL DIRECTEUR DE RECHERCHE, Université Gustave Eiffel	Co-directeur de thèse

Rapporteurs :

Sophie Ebersold
PROFESSEURE, Université Toulouse - Jean Jaurès
Walter SCHÖN
PROFESSEUR, Université de Technologie de Compiègne, Sorbonne Universités, France

Thèse soutenue publiquement le **19 décembre 2023**, devant le jury composé de :

Yves LEDRU PROFESSEUR, Université Grenoble Alpes	Directeur de thèse
Akram IDANI MAITRE DE CONFERENCES, Université Grenoble Alpes	Co-directeur de thèse
Simon COLLART-DUTILLEUL DIRECTEUR DE RECHERCHE, Université Gustave Eiffel	Co-directeur de thèse
Amel Mammar PROFESSEURE, TELECOM SudParis	Examinatrice
Gwen Salaun PROFESSEUR, Université Grenoble Alpes **(President)**	Examinateur
Jérémie Christian Attiogbé PROFESSEUR DES UNIVERSITES, Nantes Université	Examinateur
Sophie Ebersold PROFESSEURE, Université Toulouse - Jean Jaurès	Rapporteure
Walter SCHÖN PROFESSEUR, Université de Technologie de Compiègne, Sorbonne Universités, France	Rapporteur

Acknowledgment

First and foremost, I have to thank my mother Safia and father Nisar Ahmed, who supported me emotionally and with their prayers entire all of my life. I also thank everyone in my family, brother Afras, my auntie, and sisters: Mahnoor and Sarika, for encouraging me in all of my pursuits and inspiring me to follow my dreams.

An immense thank you to my PhD supervisors: Akram IDANI, Yves LEDRU and Simon COLLART-DUTILLEUL for your support and guidance throughout the Ph.D. Akram in particular you've been a great supervisor and without your push, this Ph.D. would not be accomplished and thanks for believing in me. Thanks Yves, for all the timely administrative support and all the constructive feedback from start of Ph.D. to final review of the manuscript. Thanks Simon, for all the discussions, it was a great time working with you and specially your expertise knowledge of railway domain helped me alot.

I am also grateful to everyone in VASCO team, specially: Bahareh, Nicolas and German (all the coffee breaks that we had together). A separate special thanks to German for helping me out of many technical headaches.

To all my friends and colleagues, thank you for your encouragement during my moments of crises. I can not list all the names here, but you are always on my mind. Special thanks to all the jury members, for your interest, time, and willingness to be part of my thesis jury.

Abstract

Nowadays, software systems are the backbone on which the railway industry is based. They allow to reach a high level of automation which increases the global performance level for railway mechanisms and leads to new generations of systems like autonomous ones. Several efforts with important funding are hence devoted all around the world to the rigorous production of these software systems along their development process. In fact, railway systems must satisfy safety requirements in order to be certified and made operational. In this context, one of the major requirements addressed by most of the research works in this field is correctness, meaning that a railway system must be accident-free. This challenging requirement is well mastered today thanks to formal methods that allow defining a software system using mathematical notations assisted by automated reasoning tools like provers, constraint solvers, model-checkers, etc. Indeed, several successful stories of the application of formal methods in the railway industry can be cited, like the application of the B method for the Paris subway automation. Consequently, the application of formal methods is becoming vital for these systems. However, the formal notations are not easy to understand by all actors involved in the development process because they require good skills in mathematics. Amongst these actors, we meet territorial agents, jurists, certification authorities, etc., who don't have sufficient mathematical background. Even though a railway system is proved using formal methods, it does not guarantee it is correctly built and it might lead to errors because the initial formal specification might not describe the right model.

In order to overcome these challenges, this thesis proposes a formal xDSL-based framework that combines formal methods with domain-specific representations. Indeed, domain-specific representations help for validation and acceptance by domain experts because they are expected to be more meaningful (from a human point of view) than a formal specification, especially for actors who are not familiar with formal notations. Specific representations (textual or graphical) of domain concepts are omnipresent thanks to their ability to show standardized information with common knowledge about several railway mechanisms: tracks, rules, and interlocking systems. In our framework, the railway DSL is based on standard railway notations like EULYNX, and its semantics are defined using the B formal method. The framework also supports the execution of the railway DSL with dynamic semantics provided by existing proved ERTMS/ETCS B specifications. The proved B specifications are linked with the semantics of the DSL using a linkage machine written in B itself. The use of EULYNX and ERTMS/ETCS in the DSL-based framework makes the railway models conformant to standards. The overall execution in the framework and translation of DSL semantics into B is supported by Meeduse. It is a Language Workbench, dedicated to formally instrument DSLs using the B Method and it is built on Eclipse Modeling Framework and ProB, providing features for verification, animation, and debugging. Further, this thesis separates the concerns of a formal method expert from a domain expert and a model-driven engineering expert. The formal method experts will only be responsible for verifying their specifications and linking them with the formal semantics of the models using formal methods. Whereas the model-driven engineering expert defines the DSL, and domain experts design and validate their models expressed in the DSL.

Résumé

Aujourd'hui, les systèmes logiciels constituent l'épine dorsale sur laquelle repose l'industrie ferroviaire. Ils permettent d'atteindre un haut niveau d'automatisation qui augmente le niveau de performance global des mécanismes ferroviaires et conduit à de nouvelles générations de systèmes comme les systèmes autonomes. Plusieurs projets dotés de financements importants sont ainsi consacrés partout dans le monde à la production rigoureuse de ces systèmes logiciels tout au long de leur processus de développement. En effet, les systèmes ferroviaires doivent satisfaire à des exigences de sécurité pour être certifiés et rendus opérationnels. Dans ce contexte, l'une des exigences majeures abordées par la plupart des travaux de recherche dans ce domaine est la correction, ce qui signifie qu'un système ferroviaire ne doit pas engendrer des accidents. Cette exigence est aujourd'hui bien maîtrisée grâce aux méthodes formelles qui permettent de définir un système logiciel à l'aide de notations mathématiques assistées par des outils de raisonnement automatisé comme des prouveurs, des solveurs de contraintes, ou des model-checkers. En effet, on peut citer plusieurs exemples réussis d'application de méthodes formelles dans le domaine ferroviaire, comme l'application de la méthode B pour l'automatisation du métro parisien. Par conséquent, l'application de méthodes formelles devient vitale pour ces systèmes. Cependant, les notations formelles ne sont pas faciles à comprendre par tous les acteurs impliqués dans le processus de développement car elles nécessitent de bonnes compétences en mathématiques. Parmi ces acteurs, on rencontre des agents territoriaux, des juristes, des autorités de certification, etc., qui n'ont pas la formation mathématique nécessaire à la prise en main de ces méthodes. Cela pose des problèmes de validation car même si un système ferroviaire est prouvé à l'aide de méthodes formelles, il n'est pas garanti qu'il soit correct vis-à-vis des exigences métier.

Afin de surmonter ces défis, cette thèse propose un cadre formel basé sur des DSLs qui combine des méthodes formelles avec des représentations spécifiques du domaine. En effet, les représentations spécifiques à un domaine aident à la validation et à l'acceptation par les experts du domaine car elles sont censées être plus significatives (d'un point de vue humain) qu'une spécification formelle ; et ce, en particulier pour les acteurs qui ne sont pas familiers avec les notations formelles. Les représentations spécifiques (textuelles ou graphiques) des concepts du domaine sont très utilisées grâce à leur capacité à présenter des informations standardisées des mécanismes ferroviaires : voies, règles et systèmes d'enclenchement. Dans notre cadre, le DSL ferroviaire est basé sur des notations ferroviaires standards comme EULYNX, et sa sémantique est définie à l'aide de la méthode formelle B. Ce cadre prend également en charge l'exécution du DSL ferroviaire avec la sémantique dynamique fournie par des spécifications ERTMS/ETCS B ayant déjà été prouvées. Ces spécifications B sont connectées à la sémantique du DSL à l'aide d'une spécification de liaison, elle-même écrite en B. L'utilisation d'EULYNX et d'ERTMS/ETCS dans le DSL rend les modèles ferroviaires conformes aux normes. L'exécution et la traduction de la sémantique du DSL en B est prise en charge par Meeduse. Il s'agit d'un environnement dédié à l'instrumentation formelle des DSL à l'aide de la méthode B. Il est construit au dessus de Eclipse Modeling Framework et ProB, fournissant des fonctionnalités de vérification, d'animation et de déboggage. De plus, cette thèse sépare les préoccupations d'un expert en méthodes formelles de celles d'un expert de domaine et celles d'un expert en ingénierie dirigée par les modèles. Les experts en méthodes formelles seront uniquement chargés de vérifier leurs spécifications et de les relier à la sémantique formelle des modèles utilisant des méthodes formelles. Alors que l'expert en ingénierie dirigée par les modèles définit le DSL, et les experts du domaine conçoivent et valident leurs modèles exprimés dans le DSL.

Contents

List of Figures 10

List of Tables 11

List of Acronyms 12

Publications 14

1 Introduction 15
1.1 Context . . . 15
1.1.1 Validation . . . 16
1.1.2 Verification . . . 17
1.1.3 Standards . . . 17
1.2 Contribution . . . 19
1.2.1 Meeduse . . . 19
1.2.2 Proposed Framework . . . 20
1.3 Results . . . 21
1.4 Outline . . . 23
1.5 Résumé en français . . . 23

2 DSLs And Formal B Method 25
2.1 Domain Specific languages . . . 25
2.1.1 Static Semantics . . . 25
2.1.2 Dynamic Semantics . . . 27
2.2 Formal B Method . . . 28
2.2.1 Abstract Machine . . . 29
2.2.2 Refinements . . . 30
2.2.3 Inclusion . . . 32
2.3 Conclusion . . . 32
2.4 Résumé en français . . . 33

3 Railway Standards 34
3.1 ERTMS/ETCS . . . 34
3.2 UIC Standard documents . . . 37
3.3 EULYNX . . . 40
3.3.1 RSM Concepts . . . 41
3.3.2 ETCS Related Concepts . . . 43
3.3.3 Alignment between RSM and ERTMS/ETCS . . . 44
3.4 Conclusion . . . 44
3.5 Résumé en français . . . 44

4 State-Of-The-Art **45**
4.1 Introduction 45
4.2 Modeling 47
4.3 Structure vs Behavior 48
4.3.1 Structure 48
4.3.2 Behavior 49
4.3.3 Discussion 49
4.4 Standards 50
4.5 V & V (Verification & Validation) 50
4.5.1 Verification 50
4.5.2 Validation 53
4.5.3 Summary of V & V 53
4.6 Conclusion 54
4.7 Résumé en français 54

5 Visual Animation of B Specifications **56**
5.1 Introduction 56
5.2 Approach 57
5.2.1 The Lift Example 57
5.2.2 Proposed architecture 58
5.2.3 Illustration 59
5.3 Designing a domain-centric visual animation 59
5.3.1 The Lift DSL 59
5.3.2 Static Semantics 60
5.3.3 Linking B data structures 60
5.3.4 Initialization 63
5.3.5 Operations 64
5.3.6 Enhancements 65
5.4 Application to Scheduler Example 66
5.5 Discussion 69
5.6 Conclusion 69
5.7 Résumé en français 70

6 Validation of proved ERTMS/ETCS B specification **71**
6.1 Introduction 71
6.2 Towards an Iterative Formal Model-Driven Approach 72
6.3 An ERTMS/ETCS Hybrid Level 3 DSL 73
6.3.1 DSL version 0 (DSLv0) 73
6.3.2 Translation of the meta-model 73
6.3.3 Linkage Machines 74
6.3.4 Modeling and visual animation 76
6.4 Findings and Analysis 77
6.4.1 Next Iterations 77
6.4.2 Unexpected behaviors 80
6.4.3 Lessons learned 80
6.5 Conclusion 81
6.6 Résumé en français 82

7 Automatic Linkage Generation **83**
7.1 Introduction . . . 83
7.1.1 Pattern-Definition . . . 84
7.1.2 Pattern-Application . . . 85
7.2 Experimentation with the Tool . . . 86
7.2.1 Lift . . . 87
7.2.2 Scheduler . . . 90
7.2.3 ERTMS/ETCS . . . 92
7.3 Discussion . . . 95
7.4 Conclusion . . . 96
7.5 Résumé en français . . . 96

8 Application **97**
8.1 Introduction . . . 97
8.2 Methodology . . . 97
8.3 Alignment of Meta-models . . . 98
8.4 Translation of Meta-Model into B . . . 99
8.5 Linkage Invariants and Properties . . . 99
8.5.1 Mapping class SectionList to route (minTTD to maxTTD) . . . 99
8.5.2 Mapping class TvpSection to Ttds . . . 99
8.5.3 Mapping enumeration SectionVacancyTypes to set StateTTD . . . 101
8.5.4 Mapping TvpSection occupancy to Ttds occupancy (variable stateTTD) 101
8.5.5 Mapping class VirtualSubSection to Vss . . . 102
8.5.6 Other examples . . . 102
8.6 Linkage Operations . . . 103
8.7 Visualization . . . 104
8.7.1 Topology Design . . . 104
8.7.2 Tabular Route View . . . 104
8.7.3 State View . . . 106
8.7.4 ETCS Document's Style View . . . 107
8.8 Conclusion . . . 108
8.9 Résumé en français . . . 108

9 Conclusion and Perspectives **110**
9.1 Contribution . . . 110
9.2 Perspectives . . . 112
9.3 Résumé en français . . . 113

List of Figures

1.1 Use Case Diagram . . . 18
1.2 Meeduse Approach [60] . . . 20
1.3 xDSL-based Framework . . . 21

2.1 Four Meta-Levels of OMG [114] . . . 26
2.2 Petri-Net Meta-Model [44] . . . 26
2.3 Petri-Net Model [44] . . . 27
2.4 Algorithms to run Petri-Nets [44] . . . 27
2.5 Petri-Nets Model and Execution [44] . . . 28
2.6 Abstract Machine of Server [25] . . . 30
2.7 Development Process in B [35] . . . 31
2.8 Refinement of Abstract Server Machine [25] . . . 31
2.9 Counter Machine B model example with clause INCLUDES . . . 32

3.1 ETCS Level 1 diagram [32] . . . 35
3.2 ETCS Level 2 diagram [32] . . . 35
3.3 ERTMS level 2 and level 3 fixed block and moving block concepts [102] . . . 36
3.4 Hybrid level 3 section conventions [9] . . . 37
3.5 VSS State Machine [9] . . . 37
3.6 RTM Packages [68] . . . 38
3.7 Classes of Net Entity Package from RailTopoModel [68] . . . 39
3.8 Overview of RSM Packages [27] . . . 39
3.9 First-Level Classes from EULYNX subset . . . 41
3.10 RSM Classes from EULYNX subset . . . 41
3.11 ETCS Classes from EULYNX subset . . . 43

4.1 Screenshot of Rail-AiD editor . . . 48
4.2 Errors to be avoided . . . 51

5.1 Lift example [84] . . . 57
5.2 Proposed approach . . . 58
5.3 Visual animation in Meeduse . . . 59
5.4 Lift Meta-Model . . . 60
5.5 Structural part of machine $Lift_{static}$ . . . 61
5.6 Meeduse after the initialization . . . 63
5.7 Structural part of Existing Scheduler B Specification . . . 67
5.8 Scheduler Meta-Model . . . 67
5.9 Structural part of Scheduler B Specification . . . 68
5.10 Graphical animation of the Scheduler example . . . 68

6.1 The Iterative Architecture for the Case Study . . . 72
6.2 B data structure of existing abstract machine M0 . . . 74
6.3 DSLv0 Meta-Model . . . 74

6.4 Structural part of machine DSLv0.mch (without constants and variables) . 75
6.5 A model conforming to DSLv0 77
6.6 Animating DSLv0 using M0 77
6.7 Whole DSL Meta-Model 78
6.8 Assigning MA to Train 1 using DSLv3 79
6.9 Train Movement consuming MAs 80

7.1 Linkage Generation Methodology 83
7.2 Xtext grammar of Pattern-Definition 84
7.3 Pattern-Definition meta-model 85
7.4 User defined patterns 86
7.5 Xtext grammar of Pattern-Application 87
7.6 Pattern-Application meta-model 87
7.7 Pattern-Application textual editor 88
7.8 SingleValuedERefToSetElement Pattern 89
7.9 SingleValuedERefToSetElement Application 89
7.10 MultipleValuedERefToSet Pattern 89
7.11 MultipleValuedERefToSet Application 89
7.12 EnumTypeToBoolean Pattern 90
7.13 EnumTypeToBoolean Application 90
7.14 Scheduler Pattern Template 91
7.15 EClassToExtendedConstant Application 91
7.16 ReferenceToVariable Application 92
7.17 EClassToConstant Pattern 93
7.18 EClassToConstant Application 93
7.19 EnumToSet Pattern 93
7.20 EnumToSet Application 94
7.21 BoolAttributeToBoolVariable Pattern 94
7.22 BoolAttributeToBoolVariable Application 94
7.23 EnumTypeAttributeToSetValuedVariable Pattern 95
7.24 EnumTypeAttributeToSetValuedVariable Application 95

8.1 Merging EULYNX with existing ERTMS/ETCS B specifications 98
8.2 ETCS Data Meta-model 98
8.3 Structural part of machine eTCSData.mch (without constants and variables) 100
8.4 Operation trainEntering 103
8.5 Topology Design 104
8.6 Level 1 105
8.7 Level 2 105
8.8 Level 3 106
8.9 Level 4 (Assigning MA) 106
8.10 Level 4 (Train Movement) 106
8.11 States Legend 107
8.12 TTD States 107
8.13 VSS States of Route 1 107
8.14 ETCS Document's Style View of Route 1 108

9.1 Our xDSL-based framework for merging railway standard notations 110

List of Tables

4.1 Comparison of State-of-the-Art Approaches 46
4.2 Properties and the used formal techniques 51

7.1 Comparison of Defined and Applied Patterns 88

List of Acronyms

AFNOR	Association française de normalisation
CAD	Computer Aided Design
CNL	Controlled Natural Language
DSL	Domain Specific Language
DSML	Domain Specific Modelling Language
DP	Data Preparation
EMF	Eclipse Modeling Framework
EUAR	European Union Agency for Railways
ERTMS	European Rail Traffic Management System
ETCS	European Train Control System
FM	Formal Method
GMF	Graphical Modeling Framework
GPL	General Purpose Language
GPS	Global Positioning System
GNSS	Global Navigation Satellite System
GSM-R	Global System for Mobile communications - Railways
HL3	Hybrid Level 3
HTML	HyperText Markup Language
ISO	International Standard Organization
IDL	Interlocking Dynamic Language
ICL	Interlocking Configuration Language
IM	Infrastructure Manager
ITG	Interlocking Table Generator
LEU	Lineside Electronic Unit
MA	Movement Authority
MBSE	Model-Based System Engineering
MDE	Model-Driven Engineering
OMG	Object Management Group
RTM	RailTopoModel
RSM	RailSystemModel
RailML	Railway Markup Language
RBC	Radio Block Centre
Rail-AiD	Railway Infrastructure and Layout Aided Designer
RSL	RAISE Specification Language
SVG	Scalable Vector Graphics
SQL	Structured Query Language
SIL	Safety Integrity Level
TCL	Train Control Language
TIMS	Train Integrity Monitoring System
TTD	Trackside Train Detection
TSI	Technical Specifications for Interoperability

UIC	International Union of Railways
UML	Unified Modeling Language
V & V	Verification & Validation
VSS	Virtual Sub-Section
xDSL	eXecutable Domain Specific Language
xDSML	eXecutable DSML
XML	eXtensible Markup Language

Publications

1. **Asfand Yar**, Akram Idani, Yves Ledru, Simon Collart Dutilleul: Visual animation of B specifications using executable DSLs. MoDELS (Companion) 2022: 617-626
2. **Asfand Yar**, Akram Idani, Simon Collart Dutilleul: Merging Railway Standard Notations in a Formal DSL-Based Framework. ECSA (Companion) 2020: 411-419

Chapter 1

Introduction

In the railway industry, software systems are used to achieve a high level of automation [39, 108], and safety in this automation can be ensured through techniques of Formal Methods (FMs). FMs are used for the specification and verification. These techniques apply theorem proving or model-checking to ensure verification. Due to the safety-critical nature of railway systems, the use of FMs is a strong recommendation by CENELEC EN 50128 standard[1].

Though FMs prove the consistency of a railway system, it does not guarantee it is correctly built. The misunderstanding of the user requirements by FM experts may lead to errors. In order to avoid errors, validation is performed. Validation inspects whether specifications meet the user requirements. It has to be done by domain experts who require familiar notations. Domain Specific Languages (DSLs) are languages designed and tailored for a specific application domain [89]. In the railway domain, DSLs allow the design of readable railway models thanks to domain-specific notations, and such domain-specific models and their animation could be used for validation.

Verification and validation provide correctly built, proved railway systems. However, the railway systems must be certified by certification bodies. For certification of railway systems, standards are required in the design phase of railway systems. Several standards are documented by European and national authorities in the railway domain. These documents provide engineering rules and infrastructure guidelines and allow the establishment of common interfaces for railway systems to maintain compatibility among cross-border infrastructure objects. All the railway systems need to be aligned and conformant to such standards. In this chapter, we discuss the context and contribution of this thesis toward the validation of formal railway systems using DSLs that are conformant to standards.

1.1 Context

DSL technologies offer a lot of flexibility in designing a convenient system (either graphical or textual). The design of railway systems can be done using graphical syntax or textual syntax, and DSLs have been a choice for this purpose. There are several tools that propose DSLs to design railway systems, such as RaIL-AiD[2], SafeCap [65, 66, 67], OnTrack [76], etc. However, most of these tools are not formally defined, and hence they do not apply formal verification techniques. In railway systems, the design, simulation, and animation of models in DSLs help with the validation. Most DSLs allow the graphical design of railways models to avoid wrongful designs that lead to errors. But they do not facilitate railway experts to animate railway models visually. The visual animation of railway models makes the validation more comfortable as the railway experts are familiar with

[1]https://standards.globalspec.com/std/13113133/en-50129.

[2]Railway Infrastructure and Layout Aided Designer (https://www.rail-aid.com).

railway-centric notions, and apart from that, animated execution of models in DSL allows to trace errors.

The lack of either verification or validation techniques in DSLs always motivated computer scientists to explore the domain of railway DSLs, and previously several theses such as [108, 72, 121] have been accomplished in this domain. We can find many strategies and tools such as [110, 66, 73] for the verification and validation of railway systems, but the DSLs they provide are not directly derived from railway standards. In this context, our thesis aims to start by studying existing strategies and standards to propose a novel DSL-based solution to merge with railway standards while achieving validation. During this section, we discuss validation, verification, and standards in the context of this thesis.

1.1.1 Validation

Validation of a software system corresponds to the question: are we developing the right system? It can be done using testing, animation, simulation, and reviews. Validation can be seen in all the phases of the software development. Validation of a software system is checking whether it behaves as expected. For this purpose, the expected behavior must be properly documented, and the requirements must be properly identified. This process must involve an expert in the domain for which the software system is being developed. When the system's architecture is designed in the early phases of development, it can be checked against the identified requirements; for example, the UML diagrams of the system are designed based on the documented requirements. In software engineering, the development of the system can be seen in two ways: (i) software development using programming (writing code) e.g., agile methods, and (ii) model-driven software engineering, where the system is defined using models. Based on these two ways of development, validation can also be classified into two ways [104]: (i) Validation of software without models and (ii) Validation in model-driven engineering. In the first way, the code of software can be debugged or tested (dynamic checking). In the second way, a model of the system is checked against specific properties (called static checking by [104]), or its behavior is analyzed using animation or simulation (dynamic checking).

In the railway domain, different methods of validation are in practice. In companies like Alstom, test case scenarios are designed and then simulated using their simulators for validation[3]. In academia, there are also several researchers that provided approaches based on the validation of railway systems. James *et* al. [74] used methods like simulation and error injection where incorrect scenarios are introduced to check whether the errors can be identified or not. Vu *et* al. [115, 116] uses a static-checker for the validation of specified data and specifications to check whether these are well-formed or not. The industrial railway design tool Rail-AiD [28] favors the validation as it does not allow wrong designs that lead to errors. From the previously mentioned examples about validation in railways, it can be seen that validation in railways helps in error-free and well-formed railway models and scenarios.

With the aim of achieving validation, this thesis focuses on Model-Driven Engineering (MDE) paradigm, as it is supported by several tools (*e.g.* EMF, Xtext, Sirius, etc.) and approaches (*e.g.* transformation, code generation, etc.) that allow one to define graphical and textual domain-specific modeling languages (DSMLs). We particularly focus on executable DSMLs (xDSML) in order to allow domain experts to simulate domain-specific scenarios and validate the underlying behaviors.

[3] Private Communication

1.1.2 Verification

Verification in software systems provides confidence: are we developing the system right? Verification is part of FMs which provides a guarantee that the software meets its specification [34]. Some of the well-known examples of formal specification notation languages are Alloy [70], CASL [91], Lustre [58], CSP [47], LOTOS [17], B-Method [35], Z notation [36], Petri nets [53], RAISE [93], SPARK Ada [30], etc. To ensure that the system defined as mathematical notations is correct, FMs provide techniques like logic programming (a program written as a set of sentences in logical form), constraint solving (problems defined as questions that must satisfy a number of constraints or limitations), theorem proving (reasoning over proofs) and model-checking (checking whether a finite-state model of a system meets a given specification).

Case studies of FMs can be found almost in all safety-critical domains, from aerospace and embedded systems to railway domains and communication. The book Application of Formal Methods [46], published in the year 1995, contains a collection of articles by internationally renowned contributors from both academia and industry. The applications include STV (Single transferable vote) algorithm, AAMP5 microprocessor, real-time gate controller, rail traffic and signaling, operating system, communication system, AT&T switching system, aerospace system, etc. Especially regarding the aerospace systems, the NASA case study document [13] shows the use of different classes (techniques) of FMs, like theorem proving and model-checking. It illustrates how FMs can be used in a realistic avionics software development project.

In the railway field, the application of FMs is not a new paradigm. Formal techniques have been used in railway systems to verify different properties like safety, capacity, interlocking, deadlocks, security, topology, etc. International railway infrastructure and manufacturer companies like Alstom and Siemens Mobility and national rail service providers like SNCF, ProRail, etc., are using FMs. In fact, the survey [120] on FMs acknowledged transportation (including railway) as the first domain for the application of Formal Methods as well as for the development of real-world railway systems. Apart from being used in industry, the research involving the application of Formal Methods on railway systems is quite active in academia. Another survey [56] on FMs for railways shows that 68 % of the research papers regarding the use of FM in railways published after 2015 are authored in academia.

This thesis proposes to define the semantics of railway models using FM, and then existing proved specifications are linked with the model's semantics. This thesis aims to separate the concerns of a FM expert from a domain expert and a MDE expert. FM experts will only be responsible for verifying their specification and linking it with the formal semantics of the model using FM. Whereas the MDE expert defines the DSL, and domain experts design and validate their model. This separation of actors' concerns is depicted in Figure 1.1, illustrating the three actors and their activities (use cases). Dependencies between the use cases are also clearly mentioned in the Figure, which shows how the activities of actors are related to each other.

1.1.3 Standards

Standards refer to the common, agreed, and expert approaches to maintain quality and interoperability. These standards are managed by international, regional, and national bodies. ISO[4] (International Standard Organization) provides standards in many areas like quality, environment, health, energy, food, and IT security. Same as AFNOR[5] works as the

[4] https://www.iso.org/.

[5] Association française de normalisation https://www.afnor.org/.

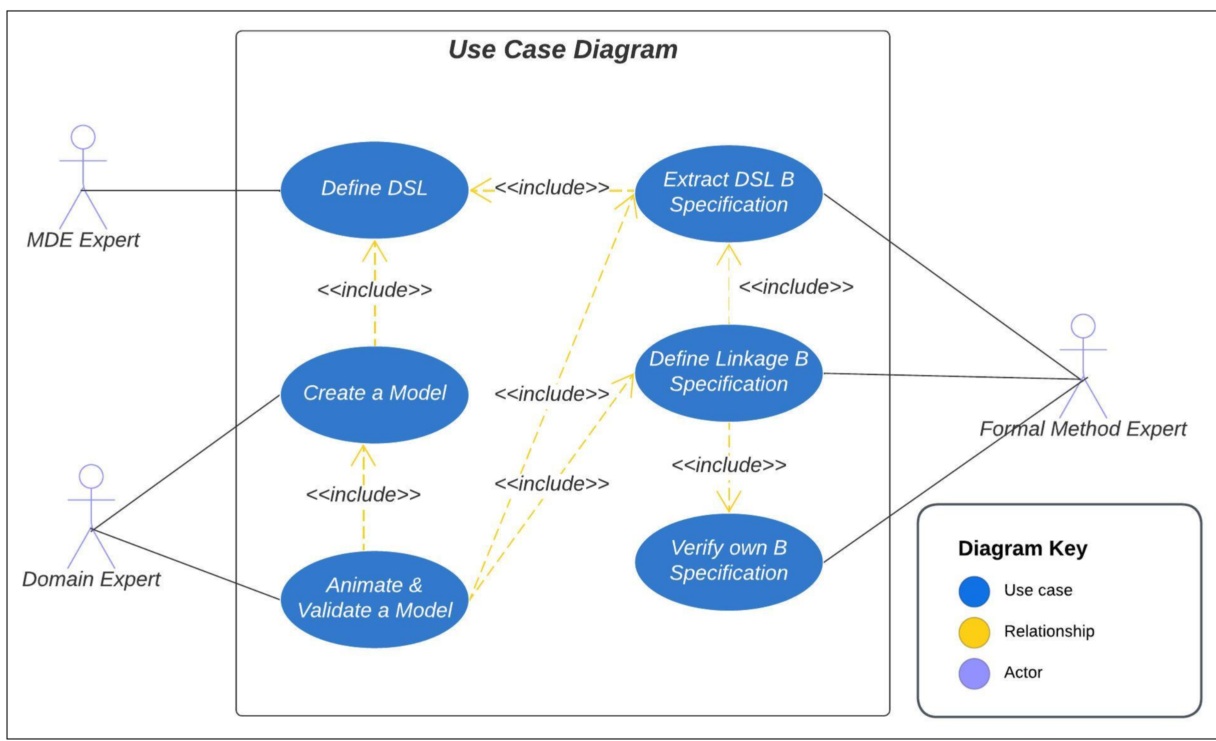


Figure 1.1: Use Case Diagram

body of standardization in France, maintaining good practices and effective solutions. In the railway domain, the EUAR[6] (European Union Agency for Railways) provides TSIs[7] which define the technical and operational standards to ensure the interoperability of railway systems of the European Union.

ERTMS/ETCS [32], European Rail Traffic Management System and European Train Control System, is a set of specifications by EUAR and European authorities. It is introduced as a standard for a common inter-operable platform for railway management and signaling system and is currently operational on many European railway lines. In the future, it is intended to replace all the national signaling systems in Europe. Currently, the use of ERTMS/ETCS is quite active in the railway domain; an industrial tool like Rail-AiD [28] used it for its data preparation, and academic researchers mentioned and illustrated the use of ERTMS/ETCS in their DSL-based approaches [74, 115, 63].

ERTMS/ETCS standard provides engineering and operational rules, but it does not provide support for the infrastructural design of railway systems. To support common infrastructure design in railway systems, infrastructure managers (IMs) came up with RailTopoModel (RTM) [68]. It is the International Railway Standard (IRS 30100[8]), a domain model whose aim is to optimize the communication between the various actors of the railway sector providing abstract concepts. Later RTM is re-branded as RailSystem-Model[9] (RSM). In this work, we pay particular attention to RailML [26] and EULYNX [11] to use RTM/RSM. RailML (Railway Mark-up Language) is an open XML (eXtensible Markup Language) based data exchange format for data interoperability of railway applications, and its infrastructure schema (containing concrete concepts) extends RTM. EULYNX is a European initiative by 13 Infrastructure Managers to standardize interfaces and elements of the signaling systems, and it also provides support for the logical concepts

[6]https://www.era.europa.eu/.

[7]Technical Specifications for Interoperability https://www.era.europa.eu/domains/technical-specifications-interoperability_en

[8]The IRS 30100 is the foundation for quick, unambiguous, and error-free data storage and data exchange inside and between business processes [68]

[9]https://rsm.uic.org/doc/rsm/rsm-1-2/

of ERTMS/ETCS.

In this thesis, we derive the static semantics of our DSL from RTM/RSM. We use meta-models provided by RailML or EULYNX (called common interfaces in this thesis), which makes the design of railway models conformant to RTM/RSM standards. We use ERTMS/ETCS formally proved specifications for the DSL's dynamic semantics. These proved specifications ensure that the underlying dynamic semantics are correctly derived from the ETCS document, ultimately achieving equivalence to standards.

1.2 Contribution

Several DSL-based approaches like [38], [43], [74], [75], [109], [115, 116], [65, 67], [76], etc. have been presented for railway systems which cover verification, validation and standards (detailed study is done in chapter 4). The mentioned existing approaches show their strength but the following areas of improvements and gaps were identified: (i) to have a DSL based on EULYNX and ERTMS together, (ii) a framework to work with any railway DSL and to provide executable models, and (iii) to use existing proved specifications for verification activities within a DSL-based framework.

To contribute in the domain and to cover the identified gaps, in this thesis, we provide an executable DSL-based framework, which allows the graphical design of railway models based on a railway common interface that follows RTM/RSM. The framework supports the execution of railway DSL with dynamic semantics provided by ERTMS/ETCS. Our framework shown in Figure 1.3 is built on Meeduse language Workbench [60], which first helps with the translation of railway models into FM and then the animation of domain-specific models (responsible for steps: Translation, Valuation, and synchronization including the overall Execution process). In this section, we discuss the components of our framework alongside Meeduse.

1.2.1 Meeduse

Meeduse [60, 64] is developed by the VASCO team at LIG (Grenoble Computer Science Laboratory). It can build proved DSLs and execute their dynamic semantics. It links three technological spaces: (i) EMF [52] (Model-driven engineering), (ii) B Method (proofs and refinements), and (iii) execution. Figure 1.2 shows the Meeduse approach, which is composed of two layers: The semantics layer and the Modeling layer. The semantics layer includes the component Translator, while the Modeling layer contains the components: Injector and Animator. The approach of Meeduse starts using the Translator component that translates Ecore[10] meta-models into B specifications as part of static semantics, generating Sets, Variables, Invariants, and basic Operations (constructors, destructors, getters, and setters). Users can make changes to strengthen some properties of the meta-models and can also include additional invariants (Note that AtelierB [2] can be used to ensure the proof of correctness of B specification with respect to included invariants). B operations can be introduced manually, but they must preserve the structural invariants.

Given a valid input model of the meta-model, the next step is to execute the behavior. The input model can be a graphical model, designed in EMF tools like Sirius, GMF, etc. (shown as Model Resource), or it can be a textual file given the Xtext grammar. The injector from the Modeling layer does valuation by injecting the instances of meta-models into the B specifications produced from the meta-models. Here, the valuation in B specification is done first by generating constants (with defined properties), and then

[10]Ecore (EMF-core) is a meta-model for describing models in Eclipse Modeling Framework, https://www.eclipse.org/modeling/emf/

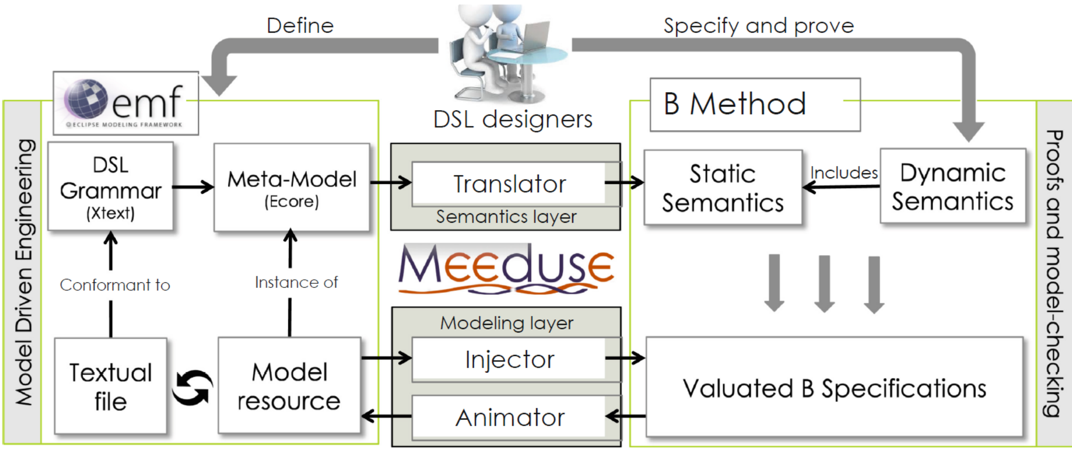


Figure 1.2: Meeduse Approach [60]

variables are initialized from the valid input model resource of the meta-model. The Animator component is supported by ProB model-checker [82]. It is asked to compute the initial state of B specification and the operations that can be animated from that state. ProB detects and stops the animation if the input model is wrong. Otherwise, given a correct input model, when a new state is reached, Meeduse translates it back to the model, resulting in domain-centric animation.

1.2.2 Proposed Framework

The top part of Figure 1.3 shows the DSLs Semantics layer of the framework. The semantics layer is divided into two more parts: static semantics and dynamic semantics. For the formal semantics, we use the B method. The choice of the B Method is motivated by several aspects; firstly, the B method is widely used in the railway field, and several success stories support this fact [80], for example, Meteor, the automated Paris subway. Secondly, a comparative study of several formal methods regarding their industrial suitability [88] has been done and rated B high for formal constructs like those applied in our thesis.

- Static Semantics: The static semantics deals with the structural schema of the DSL. In our approach, they are provided by meta-models based on RailML, EULYNX, and RailTopoModel. We use a subset of these meta-models and then Meeduse translates these into B specifications using its translator component.

- Dynamic Semantics: The dynamic semantics of a DSL deal with behavioral descriptions that make the DSL executable. In this thesis, we apply ERTMS/ETCS to introduce execution within the models provided by RailTopoModel, RailML, or EULYNX. Our objective is to use existing ERTMS/ETCS proved B specifications for the dynamic semantics definition. For this purpose, we create linkage B specifications in which we apply two mechanisms from the B method: refinement and inclusion. In the B method, refinements have two main principles: add requirements by going from abstract models to more concrete ones and prove the preservation of the abstract model invariants. The composition, such as inclusion, allows for breaking down the system by applying the separation of concerns principle. The linkage B specification includes the B specification translated from meta-models and refines/includes the existing ERTMS/ETCS B specifications.

The bottom part of Figure 1.3 shows the DSLs Execution layer of the framework. DSL execution is intended to perform early validation since the DSL is expected to be-

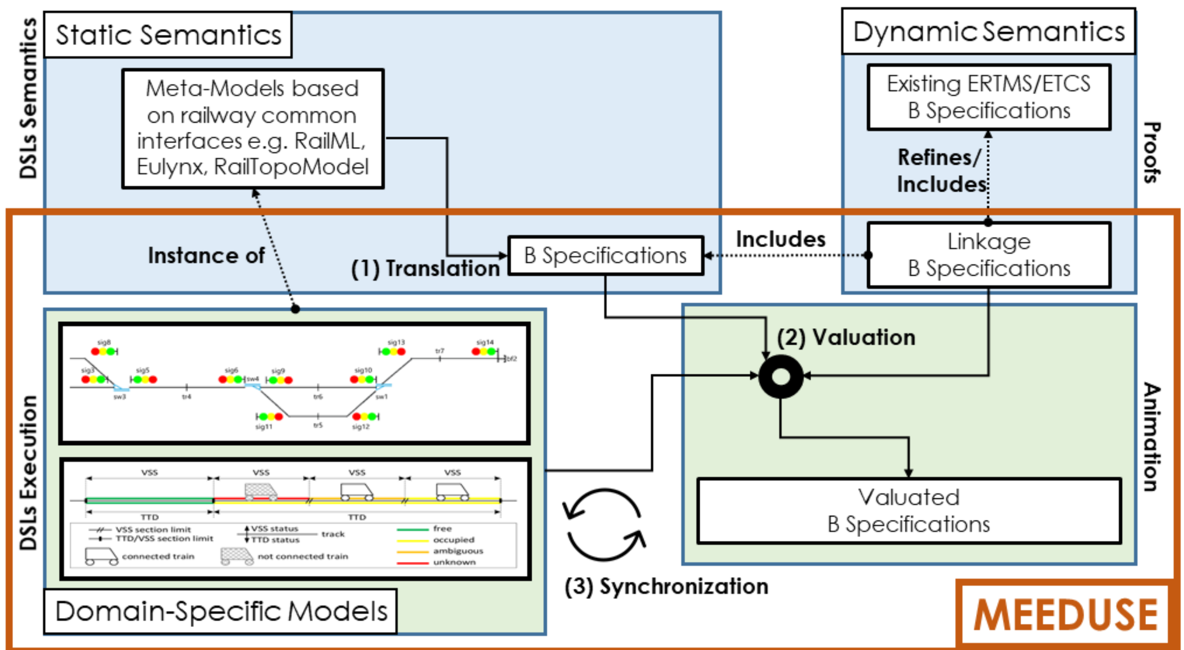


Figure 1.3: xDSL-based Framework

have as the target system should run. In our framework, domain-specific models are created as instances of the meta-models (RailML, EULYNX, or RailTopoModel/RSM). This allows the railway domain expert to design their railway models. In the next step, called (Valuation), Meeduse injects these models into B specifications issued from the meta-models. Thanks to the dynamic semantics, Meeduse can animate the design models making the DSL executable. All along the interactive animation, Meeduse synchronizes the current state of the B specifications with the models (Synchronisation), which results in a domain-centric visual animation. This execution allows the railway experts to validate the behavior of their models.

1.3 Results

The aim of this thesis is to provide an xDSL-based framework for the validation of railway models where the semantics of the DSLs are conformant to railway standards. For this purpose, we used railway infrastructural standards or interfaces like RailML, EULYNX, and RailTopoModel as static semantics of the DSL; and equipped the DSL with ERTMS/ETCS operations as dynamic semantics. We used the Meeduse language workbench for the animation of DSLs, which allows the railway expert to validate their standardized models. During this thesis, the following results are attained:

- **Visual Animation of B Specifications using DSLs [122]:** The first result of this thesis is to provide an approach to the visual animation of B Specifications using executable DSLs. Existing tools like BRAMA [107], AnimB[11], B-Motion Studio [78], B-Motion Web [79], and VisB [119]. have shown their strengths in practice, but still, there are some concerns that motivate our approach. First, visual animation provided by the existing tools requires specific skills such as in scripting languages (Flash, JavaScript, node.js) or in Scalable Vector Graphics (SVG files), etc. These technologies are not necessarily mastered by FM experts and can be cumbersome to

[11] AnimB: https://wiki.event-b.org/index.php/AnimB

learn and use. Second, mapping a specification to a domain representation is time-consuming and maybe error-prone, and thirdly, FM experts might not be familiar with the domain-specific notations.

In our approach, the FM expert links the existing B specification to the B specification of the DSL. For this purpose, an FM expert does not require other skills, but instead, mapping is done in B itself. FM experts also are not required to be familiar with domain-specific notations since the input models are provided by domain experts thanks to the DSL tool. We apply our approach to the examples of Lift, Scheduler, and the realistic example of ERTMS/ETCS.

- **Validation of existing proved ERTMS/ETCS Specification:** The second result of this thesis is finding an error while validating an existing proved B specification. The hypotheses mentioned earlier in this chapter *"Though FMs prove the consistency of a railway system, it does not guarantee it is correctly built"* are proved when we visually animated an existing formally proved ERTMS/ETCS specification for validation. ABZ'2018 [49] conference organizers defined a real-life case study issued from the railway domain with challenging safety requirements. The ABZ'2018 case study was related to the Hybrid ERTMS/ETCS Level 3 standard [9]. Many FM experts published their proved ERTMS/ETCS specifications at the conference. We select the proved ERTMS/ETCS specification of Amal Mammar [87] as the case study example for the application of our approach.

  During the validation of ERTMS/ETCS B specification, an error is identified where a track section appeared as "ambiguous" although a train was in the section, and the section should appear as "occupied." As per a railway expert's opinion, such an error might come from a misunderstanding of the specification. Since the FM expert fulfilled the requirements in the document for the case study, it might be possible that the requirements are not properly documented.

- **DSL-based Tool support for Linkage B Specifications:** This thesis also leads us to develop a tool to support the generation of linkage B specifications. The tool comprises two textual DSLs: Pattern-Definition (allows to define patterns of linkage B specification mappings) and Pattern-Application (to define the structure of linkage B specification and apply the defined patterns). Thanks to these DSLs, FM experts can create linkage B specifications in a (semi)-automated way, first by defining a catalog of generic reusable patterns, and then they can select the ones to apply. Having the definition of patterns and their application, a generator component automatically produces the linkage machine. We experimented with the tool on the case study of ERTMS/ETCS and examples like Lift and Scheduler.

- **Application to EULYNX:** The last result of this thesis is the application of our xDSL-based framework to the EULYNX. This helped us to validate an existing proved ERTMS/ETCS B specification using a formalized EULYNX-based DSL. The DSL provides different graphical views of a railway system with familiarized notations. In short, this result contributes to the following: (i) formalization of EULYNX, (ii) a DSL based on EULYNX, (iii) visualization of EULYNX, and (iv) merging EULYNX with ERTMS/ETCS operating rules enabling formalized calculation (previous-next) of sections and sub-sections of a railway route.

1.4 Outline

The dissertation consists of nine chapters. The current chapter gave the motivation and discussed the contribution of this thesis. In this section, we give the remaining outline of the eight chapters to be followed.

- In chapter 2, the technologies used in this thesis: DSLs and the formal B Method, are discussed in detail with examples. DSLs are languages that have been designed and tailored for a specific application domain, while B Method is Formal Method (FM) for specifying, designing, and coding software systems.

- The notion of ERTMS/ETCS and other railway standard notations like RTM, RSM, and EULYNX are presented in chapter 3. ERTMS/ETCS is introduced as a standard for a safe and common inter-operable platform for railway management and signaling system to be implemented in Europe. RTM and RSM are standards provided by UIC, International Union of Railways to supports various aspects of the railway industry, such as signaling, safety, standardization,etc. EULYNX is an European initiative to standardize elements and interfaces of railway signalling systems.

- Chapter 4 evaluates the state-of-the-art works and approaches related to this thesis. The approaches are compared on the criteria of their semantics, the syntax used, the standards followed, and V & V.

- Chapter 5 presents our approach of introducing linkage machines for the visual animation of B Specifications using executable DSLs. We apply our approach to a simple but well-known example of Lift as proof of concepts.

- In chapter 6, the iterative formal-driven approach for the validation of existing proved ERTMS/ETCS B specification is presented. We show how a DSL is developed step by step from the concepts of existing ERTMS/ETCS B specification and how DSL-based graphical animation is used to identify errors in order to support validation.

- The DSL-based tool to generate linkage machines in a (semi)-automated way alongside examples is presented in chapter 7. We provide two DSLs: Pattern-Definition (to define re-usable patterns) and Pattern-Application (to apply the defined patterns).

- Chapter 8 discusses and illustrates the application of our xDSL-based framework to the EULYNX. This application shows the formalization and visualization of EULYNX and how it supports the validation of ERTMS/ETCS proved B specification.

- Finally in chapter 9, we draw the conclusion of this thesis by recalling our DSL-based framework and briefly discuss the derived aspects. We also give some perspectives that can be achieved as the continuation of the work in this thesis.

1.5 Résumé en français

Dans l'industrie ferroviaire, les systèmes logiciels sont utilisés pour atteindre un haut niveau d'automatisation [39, 108], et la sûreté/sécurité de cette automatisation peut être assurée grâce à des techniques de méthodes formelles (FM). Les FM sont utilisées pour la spécification et la vérification. Ces techniques appliquent la preuve de théorèmes ou le model-checking pour mener à bien les activités de vérification. En raison de la nature

critique des systèmes ferroviaires, l'utilisation de FM est une forte recommandation de la norme CENELEC EN 50128[12]. Si les FM prouvent la cohérence d'un système ferroviaire, elles ne garantissent pas qu'il soit correct. Une mauvaise compréhension des besoins des utilisateurs par les experts FM peut conduire à des erreurs. D'où la nécessité de compléter les activités de vérification par des activités de validation. La validation a pour vocation de confronter les spécifications aux exigences des utilisateurs pour vérifier leur conformité. Cela doit être fait par des experts du domaine sur la base de notations qui leurs sont familières. Dans le cadre de cette thèse, nous proposons l'usage de l'ingénierie dirigée par les modèles (IDM) et des langages dédiés domaine (ou DSL) pour mener ces activités de validation. Les DSL sont des langages conçus pour un domaine d'application spécifique [89]. Dans le domaine ferroviaire, un DSL permet la conception de modèles qui se veulent lisibles grâce à des notations dédiées et bien connues par les experts ferroviaires. La vérification et la validation permettent de produire des systèmes avec un bon niveau de correction. Toutefois, les systèmes ferroviaires doivent aussi être certifiés par des organismes de certification. Pour ce faire, des normes sont requises dès la phase de conception du système. Celles-ci sont produites et documentées par des autorités européennes et/ou nationales. Ces documents fournissent des règles d'ingénierie et des lignes directrices en matière d'infrastructure et permettent l'établissement d'interfaces communes afin de maintenir la compatibilité entre les objets d'infrastructure transfrontaliers. Tous les systèmes ferroviaires doivent être conformes à ces normes. Dans ce chapitre, nous discutons du contexte et de la contribution de cette thèse à la validation de systèmes ferroviaires formels utilisant des DSLs conformes aux normes.

[12] https://standards.globalspec.com/std/13113133/en-50129.

Chapter 2

DSLs And Formal B Method

In chapter 1, we came across a few technologies used in this thesis. This thesis provides an xDSL-based framework for the validation of railway models. The semantics of the executable DSLs in the framework are defined into a Formal Method (FM) called B Method; and are conformant to railway standards. So in this chapter, we discuss the DSLs, and Formal B Method, presented in Sections 2.1 and 2.2, respectively.

2.1 Domain Specific languages

Languages in computer science are development systems that allow computer programmers to develop software applications and control the behavior of machines [99]. On the basis of usage, they are divided into two categories: General Purpose Languages (GPLs) and Domain Specific Languages (DSLs). GPLs support generality providing generic constructs to be used together for many applications. Common examples of GPL are Java, C++, C, and Python. DSLs are languages that have been designed and tailored for a specific application domain [89]. They support expressiveness such that language easily expresses various domain-specific notations using the constructs appearing in the language. Common examples of DSL are HTML, Excel, Latex, MATLAB, and SQL.

DSLs are defined in several ways; in one way, they can be defined through the notations based on Language Theory, and the other way is to use the Model-Driven Engineering (MDE) paradigm. This thesis provides a framework that allows domain experts to define their models using DSLs in the MDE paradigm. The MDE paradigm is a set of modeling concepts, and their relationships that are used to represent a specific domain [125]. It involves various activities such as meta-modeling, model transformation, model verification, validation, code generation, and model execution. Multiple tools are available for the definition of DSLs in the MDE, like Enterprise Architect [8], MetaSketch [95], MetaEdit+ [21], Microsoft DSL Tools [24], and Eclipse Modeling Framework (EMF) [52]. In MDE, DSLs are defined through static semantics (meta-modeling) and dynamic semantics (behavior).

2.1.1 Static Semantics

We define static semantics through meta-models. According to the definition, a meta-model is a set of rules, principles, and conventions for constructing a model. The instance of the model described through the meta-model depicts the real world. This relationship between the real world, model, and meta-model is defined into four meta-levels by OMG [22] because the meta-model itself is described through another meta-model (called meta-meta-model). Figure 2.1 shows the four meta-levels defined by OMG where the bottom level MO (Instances) is described by M1 (Model) and M1 itself is described through

M2 (Meta-Model). Then M2 level is described through M3, called Meta-Meta-Model. Finally, the top-level M3 describes itself. In this meta-level structure, each lower layer is an instance of the upper layer, and the last upper layer is an instance of itself.

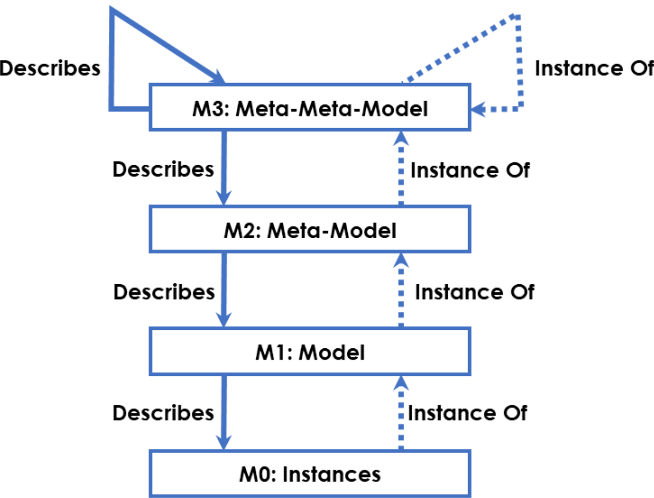


Figure 2.1: Four Meta-Levels of OMG [114]

Figure 2.2 shows the Petri-Net meta-model [44]. It contains three classes. **Net** is the root class which is composed of classes: **Place** and **Transition**. Class **Transition** has two associations with class Place: (i) input (one to many) and (ii) output (one to many). All classes have attribute *name* with type **EString**. Class **Place** also has an attribute called *tokens* of type **Integer**.

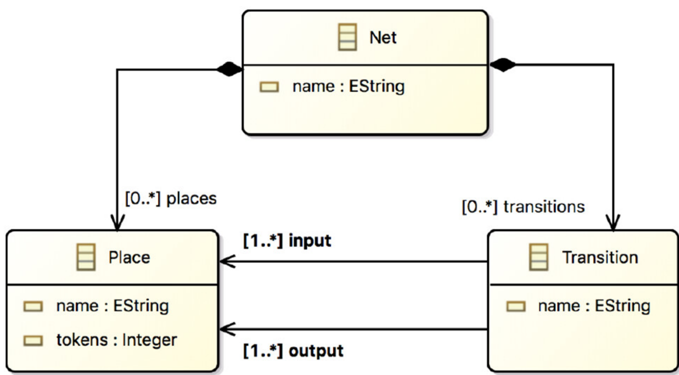


Figure 2.2: Petri-Net Meta-Model [44]

In this thesis, we use EMF, a framework containing tools for creating and manipulating models based on a structured data format called Ecore [19]. EMF supports various MDE activities such as meta-modeling, model transformation, code generation, and model execution [71], and it provides an Ecore meta-meta-model[1] for the definition of meta-models. The meta-models in EMF can be described through Ecore class diagrams. For modeling, there are several EMF-based tools, like Sirius [7], EuGENia [10], and GMF [14], which provide the graphical instantiation of meta-models. One can graphically model an instance of Ecore meta-models using one of these tools. Figure 2.3 [44] shows one graphical model of the Petri-Net meta-model from Figure 2.2. In the model, a white circle shows an instance of class Place and a vertical black bar represents an instance of class Transition. The model contains five places (p1, p2,....) and three transitions (t1, t2, and t3). The black dots in the circle places show the tokens.

[1] https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

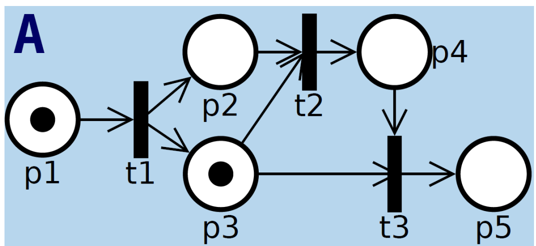


Figure 2.3: Petri-Net Model [44]

2.1.2 Dynamic Semantics

In general programming languages (GPL), the execution refers to the step when the semantics are translated into machine-readable byte-code and the byte-code is interpreted. In MDE, the execution of models is defined by means of dynamic semantics, operational semantics, execution semantics or behavioral semantics. All these terms refer to the same notion, which is the description of the model's behavior, and we use the term dynamic semantics all along this thesis for this notion. The Petri-Nets, State-Chart diagrams, and Sequence diagrams provide graphical descriptions of behaviors for the execution of DSLs.

Figure 2.4, taken from [44], shows two algorithms that can be used as dynamic semantics to run Petri-Nets. The algorithms contain five actions: run, isEnabled, fire, addToken, and removeToken. The algorithms take a Net object as input and use run to execute it. It chooses an enabled transition (which satisfies the enabledness [2] property 'isEnabled') non-deterministically from the set of transitions and then calls fire on the enabled transition. For each input place of the enabled transition, the algorithm removes a token (removeToken). It also adds a token (addToken) for each output place of enabled transition.

Algorithm 1: run

Input:
n : the Net object to run

[1] **begin**
[2] $t_{\text{enabled}} :\in \{t \in n.\texttt{transitions} \mid \mathit{isEnabled}(t)\}$
[3] **while** $t_{\text{enabled}} \neq \mathit{null}$ **do**
[4] $\mathit{fire}(t_{\text{enabled}})$
[5] $t_{\text{enabled}} :\in \{t \in n.\texttt{transitions} \mid \mathit{isEnabled}(t)\}$

Algorithm 2: fire

Input:
t : the Transition object to fire

[1] **begin**
[2] **foreach** $p \in t.\texttt{input}$ **do**
[3] $\mathit{removeToken}(p)$
[4] **foreach** $p \in t.\texttt{output}$ **do**
[5] $\mathit{addToken}(p)$

Figure 2.4: Algorithms to run Petri-Nets [44]

Figure 2.5 shows different states of the Petri-Net model [44] executed by the above-mentioned algorithm. Model state A is the initial state where only two places p1 and p3 contain tokens. At this state, fire is triggered on enabled transition t1, for which input place is p1 and output places are p2 and p3. After the trigger of fire on t1, the changes in model state B can be seen where input place p1 of transition t1 is empty, and output places got the tokens. From this state (B), fire is triggered on enabled transition t2. The change can be seen in model state C where the tokens are removed from the input places

[2] A transition is enabled if all its input places have at least one token

of t2, and a token is added into the output place p4. From model state C, fire is triggered on transition t3. This fire action removes tokens from both input places p3 and p4 of transition t3 and adds a token into place p5, as shown in the model state D.

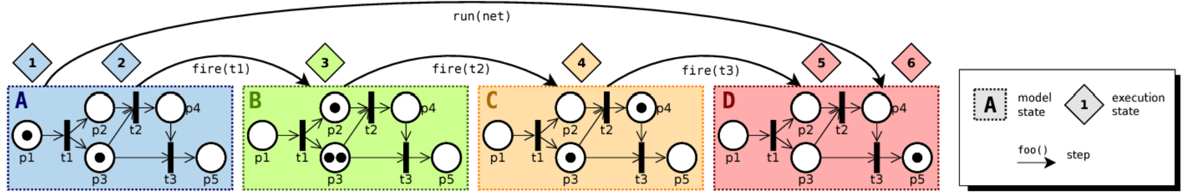


Figure 2.5: Petri-Nets Model and Execution [44]

There are two kinds of approaches that support executable DSLs: the in-line approach and the translational approach. In in-line approach, behavior is weaved into meta-models using an action language, which is used to specify the bodies of operations which are in the meta-model [48]. In the translational approach, first, the meta-model is translated into formal language semantics using model transformation. Then the operations defined in this formal semantics are used for the execution semantics.

In this thesis, we use Meeduse Language Workbench for the execution of the DSL. Using a translational approach, the EMF meta-model is translated into B Method. Then B operations serving as dynamic semantics are used to support the execution of the DSL. We use Eclipse Sirius for the graphical model, and the states of B operations are synchronized with the states of the models, assuming that they are equivalent. Computing the list of enabled operations changing the current state of the model results in domain-specific animation.

2.2 Formal B Method

According to NASA [34] *"Formal Methods" refers to mathematically rigorous techniques and tools for the specification, design, and verification of software and hardware systems.* In this thesis, we use B Method (developed by J.R. Abrial), which is a Formal Method (FM) for specifying, designing, and coding software systems [35]. It is based on Set Theory[3] and First-order Logic [37] to specify separate versions of the system created during the development process. B Method provides the formalization of railway models by giving semantics to the DSL.

The choice of the B method is motivated by several aspects. First, the comparative study of several formal methods regarding their industrial suitability [88] rated B high regarding formal constructs related to the railway domain. Secondly, B Method is widely used in the railway field, and there are several success stories [80] like Paris Metro lines 1 and 14. It has also been used in the development of the tram control system of Guangzhou Huangpu Line 1 in China [94]. Big companies working in the railway domain, like Alstom and Siemens Mobility, are using it to fulfill the recommendation of CENELEC EN 50128[4] for the development of SIL4 [3] compliant components. It covers areas like speed supervision (checking speed limits), auto-pilots for metros (metro trains without drivers), train detection systems (locating trains on the railway lines), etc.

B Method provides the mechanism for the proof of program correctness. It is done through theorem proving where logical formulas generated from different constructs of

[3] https://plato.stanford.edu/entries/set-theory/

[4] Recommends the use of formal methods in safety-critical railway systems. https://standards.globalspec.com/std/2023439/afnor-nf-en-50128

a B machine, called proof obligations, ensure the correctness claim for a given property, i.e., invariant property. Alongside theorem proving, model-checking is also used in the B Method in order to check whether a finite-state model of a system meets a given specification.

There are a few tools developed to support the B Method. The known ones are ProB [83] (a B animator and model-checker), AtelierB [2] (enabling the operational use of the B method and also with proof assitance), B-Toolkit [31] (a suite of fully integrated tools designed to support a rigorous or formal development of software systems using the B-Method), and Rodin [12] (an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof).

The development cycle of B involves three kinds of components: an abstract machine, refinements, and implementation. Abstract machine is the first version of the system providing a structured development. Refinement turns the abstract machine into a more concrete one by adding details about the system. After multiple refinements, when a deterministic version of the system is achieved, it is called implementation. It is the last version of the system and can be coded in a programming language (like C, C++, ADA, etc.) for compilation as an executable program. During this thesis, implementation component is not covered; we use only abstract machines and refinements.

2.2.1 Abstract Machine

Abstract machine is also called a B machine or abstract specification, or high-level specification, and it is comprised of three parts: header, static and dynamic.

```
Structure of an abstract machine
 MACHINE
    HEADER PART
       Name of the machine
       Parameters
       Constraints
    STATIC PART
       Sets
       Constants
       Properties of the constants
       Variables (state)
       Invariant (Properties of the variables)
    DYNAMIC PART
       Initialization of Variables
       Operations
 END
```

- **Header Part** contains the machine's name, its parameters, and the corresponding constraints of these parameters.

- **Static Part**, also known as the declarative part. It includes sets, constants, variables (states of the machine), and the corresponding properties of constants and variables. The properties of variables are called invariants and are expressed in the first-order predicate logic.

- **Dynamic Part** contains the initialization (of variables) and operations. Operations modify the states of the machine, satisfying the invariant properties.

Figure 2.6 illustrates the notions of an abstract machine mentioned above. It shows an abstract machine of a server (machine *Server_Abstract*) [25] which defines a set of processes called *Process* and a variable *logged_in* (which is a subset of *Process* allowing clients to log in). The machine has two operations: *LogIn*, and *LogOut*. Operation *LogIn* assigns a process to the *logged_in* provided it is not already assigned to it. Operation *LogOut* removes an assigned process from the *logged_in*.

MACHINE
  Server_Abstract
SETS
  $Process = \{p1, p2\}$
VARIABLES
  logged_in
INVARIANT
  $logged_in \in \mathcal{P}\ (Process)$
INITIALISATION
  $logged_in := \emptyset$
OPERATIONS
  LogIn(pp) =
    PRE $pp \in Process \wedge pp \notin logged_in$ **THEN**
      $logged_in := logged_in \cup \{pp\}$
    END;
  LogOut(pp) =
    PRE $pp \in Process \wedge pp \in logged_in$ **THEN**
      $logged_in := logged_in - \{pp\}$
    END
END

Figure 2.6: Abstract Machine of Server [25]

2.2.2 Refinements

A refinement must satisfy the properties and invariants of the refined machine as well as its own properties and invariants. Figure 2.7, shows this formal development process in B where an abstract machine as the preliminary design is refined by detailed design refinements. It is to be noted that the development process in B can follow successive refinements to make the system more concrete step by step. It is done by refining the previous refinement and similarly doing the proofs.

During the refinement, the header part of the B specification contains two keywords: REFINEMENT and REFINES. The former is the name of the refinement component, and the latter is the name of the refined component, which looks as follows:

Header part of refined machine
REFINEMENT refinement
REFINES abstractmachine

Refinement of the Server example (Refinement *Server_Refinement*) is illustrated in Figure 2.8, where machine *Server_Abstract* is refined first by data refinement with the addition of a new set called *SessionID* and a new variable called *session* which is a partial function ($\nrightarrow$) from *Process* to *SessionID*. In this refinement, variable set *logged_in* is redefined as the domain of variable *session* (**dom**(*session*)). Machine *Server_Abstract* is

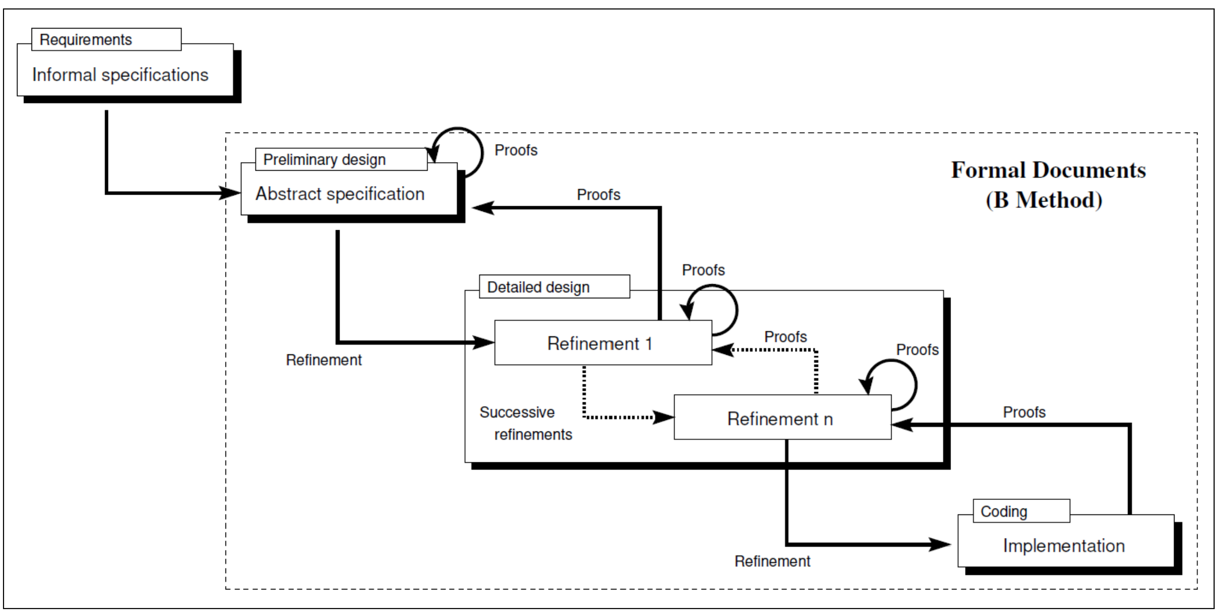


Figure 2.7: Development Process in B [35]

REFINEMENT *Server_Refinement*
REFINES *Server_Abstract*
SETS
$SessionID = \{sid1, sid2, sid3\}$
VARIABLES
session
INVARIANT
$session \in Process \rightarrowtail\!\!\!\!\!\!+ \; SessionID \wedge$
$logged_in = \mathbf{dom}(session)$
INITIALISATION
$session := \emptyset$
OPERATIONS
$\mathbf{LogIn}(pp) =$
$\mathbf{PRE}\ pp \in Process \wedge pp \notin \mathbf{dom}(session)\ \mathbf{THEN}$
$\mathbf{ANY}\ sid\ \mathbf{WHERE}\ sid \in SessionID \wedge sid \notin \mathbf{ran}(session)$
THEN
$session := session \cup \{pp \mapsto sid\}$
END
END;
$\mathbf{LogOut}(pp) =$
$\mathbf{PRE}\ pp \in Process \wedge pp \in \mathbf{dom}(session)\ \mathbf{THEN}$
$session := \{pp\} \lhd\!\!\!- \; session$
END
END

Figure 2.8: Refinement of Abstract Server Machine [25]

then refined by operation refinement. Operations *LogIn*, and *LogOut* are redefined with respect to *session* instead of *logged_in*. During the operation *LogIn*, a process is included in the *session* such that it is not already within the domain of *session* and further the *SessionID* associated with *Process* is not in the range of *session*. Operation *LogOut* subtracts a *Process* from the *session*, which is within its domain.

2.2.3 Inclusion

Another mechanism of the B Method that we use in this thesis is inclusion. It allows to break down the system by applying the separation of concerns principle and includes other machines in a given machine. Figure 2.9 shows an example where a B model is composed of two abstract machines that are related to each other via the inclusion mechanism. The left side of the figure defines a counter machine (called *CounterMachine*). It has a set *COUNTER* and variable *Counter* which is defined as a member of the powerset of set *COUNTER*. *Counter* is linked to a value (an integer) with one basic setter operation setValue, which sets the value of the counter. The right side of the figure shows the machine *CounterOperations* with operations: **increase**, and **decrease**, which increase and decrease the value of the counter, respectively. This Machine also provides two constants: *MAXvalue* and *MINvalue* which are integers to define the maximum and minimum of the counter's value. Note that using the Inclusion mechanism, the variables of the first abstract machine, in this case, *CounterMachine*, can be read by the other machine (*CounterOperations*) but their modification is only possible via operation call.

```
MACHINE
  CounterMachine
SETS
  COUNTER={counter}
VARIABLES
  Counter,
  value
INVARIANT
  Counter ∈ P (COUNTER)   ∧
  value ∈ Counter ⇸ Z
INITIALISATION
  Counter:= {counter} ||
  value:= {counter ↦ 1}
OPERATIONS
  setValue(aCounter,aValue) =
  PRE   aCounter ∈ Counter ∧
     aValue ∈ Z
  THEN
     value := ({aCounter} ⩤ value)
       ∪ {(aCounter ↦ aValue)}
  END
END
```

```
MACHINE
  CounterOperations
INCLUDES
  CounterMachine
CONSTANTS
  MAXvalue,
  MINvalue
PROPERTIES
  MAXvalue ∈ Z ∧ MAXvalue=10 ∧
  MINvalue ∈ Z ∧ MINvalue=0
OPERATIONS
  increase = PRE
     value(counter) < MAXvalue
  THEN
     setValue(counter, value(counter)+1)
  END;
  decrease = PRE
     value(counter) > MINvalue
  THEN
     setValue(counter, value(counter)-1)
  END
END
```

Figure 2.9: Counter Machine B model example with clause INCLUDES

2.3 Conclusion

In this chapter, we studied the technologies used in this thesis from the computer science discipline point of view: DSLs and the B Method. The former helped us to understand the rules and ways behind defining it, while the latter helped us to understand its structure, components, and usage. DSLs are defined using the MDE paradigm, and MDE is more convenient to model the railway notions and their relationships. Another important point about the MDE paradigm is the support for validation activities. The verification is done using Formal Methods (FMs) and as per CENELEC EN 50128, the use of FMs is a strong

recommendation in railway safety critical systems. We use the formal B Method in this thesis as it is widely used in the railway field and has several success stories in the domain. In the next chapter (3), we explore railway standard notions that are useful in this thesis.

2.4 Résumé en français

Cette thèse fournit un cadre basé sur des DSL pour la validation des modèles ferroviaires. La sémantique des DSL exécutables y est définie dans une méthode formelle (FM) appelée méthode B. Ainsi, dans ce chapitre, nous discutons des DSL et de la méthode formelle B. Les DSL sont définis à l'aide du paradigme MDE, et MDE est bien adapté pour modéliser les notions ferroviaires et leurs relations. Un autre point important du paradigme MDE est la prise en charge des activités de validation. La vérification est effectuée à l'aide de méthodes formelles (FM) et conformément à la norme CENELEC EN 50128, l'utilisation de FM est fortement recommandée dans les systèmes critiques pour la sécurité ferroviaire. Nous utilisons la méthode formelle B dans cette thèse car elle est largement utilisée dans le domaine ferroviaire et compte plusieurs réussites dans le domaine.

Chapter 3

Railway Standards

This thesis provides an xDSL-based framework, which allows the graphical design of railway models based on railway standards like RTM/RSM and EULYNX. The framework supports the execution of railway DSL with dynamic semantics provided by ERTMS/ETCS. We discuss thesis points in Sections 3.1, 3.2, and 3.3.

3.1 ERTMS/ETCS

ERTMS stands for European Rail Traffic Management System, and ETCS stands for European Train Control System (It can be called simply ERTMS or ETCS). It is introduced as a standard for a safe and common inter-operable platform for railway management and signaling system to be implemented in Europe. It is intended to be adopted in all European countries and to replace their national signaling systems [32].

In ETCS, the safe train behavior is made sure by the concept of Movement Authority (MA). MA defines how and when a train is allowed to enter a specific section of the track. The function of MA does not allow two trains to be on one specific section at the same time. ETCS also looks after the concerns regarding safe driving which includes emergency brakes in case of an accident ahead or change in the track condition. It also assigns speed limits to the train depending on the traffic and curvedness of the line to avoid train collisions and derailments respectively.

Moreover, ERTMS/ETCS is classified into different functional levels according to the equipment used and operating modes. The levels are ETCS level 1, level 2, and level 3. The application of ETCS enables the trackside equipment to transmit information to the train. In ETCS level 1, this information is transmitted by Balise (also called Eurobalise), which is an equipment placed alongside the track and connected to the signaling system. In level 2 and 3, this information is transmitted by GSM-R[1] radio network. The ERTMS/ETCS level 1, level 2 and level 3 are discussed below:

- ETCS level 1: It is a cab signaling system[2] and can be easily implemented on existing national signaling systems. In this level, data is exchanged from track to train using balise. The signals alongside the track are necessary at this level. The balise gets the signal information and sends them to train as MA with data of route. Figure 3.1 illustrates the cab signaling system of ETCS level 1 (taken from [32]) where a signal is connected to LEU (Lineside Electronic Unit). LEU sends the appropriate information (called a telegram) to the balise, which in turn sends it to the onboard

[1] GSM-R is the mobile communication system used in the railway sector.

[2] Cab signaling system is a railway safety approach where track status and information is communicated to the driver's compartment.

system. Train detection and train integrity[3] checks are performed by the trackside equipment beyond the scope of ERTMS.

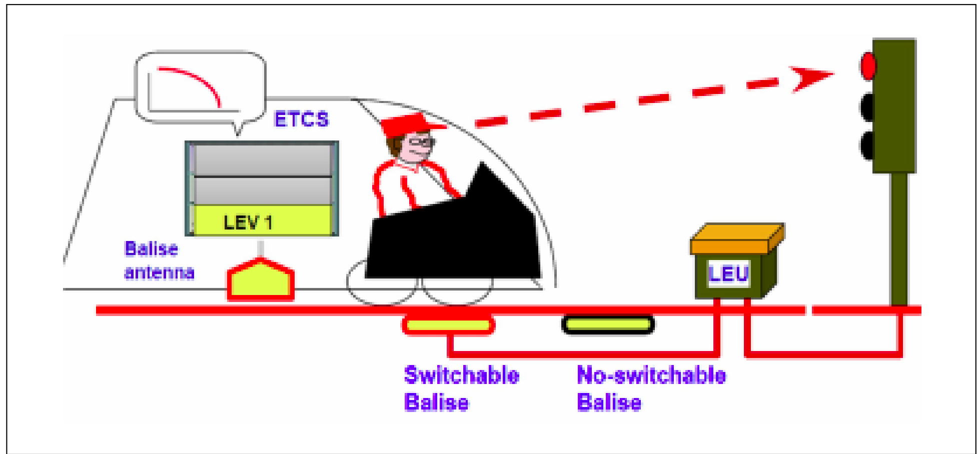


Figure 3.1: ETCS Level 1 diagram [32]

- ETCS level 2: In this level, a radio-based system is used to display the cab's signaling, and MAs. The train continuously sends its data to the Radio Block Centre (called RBC) to report its exact position and its direction using GSM-R. RBCs are radio centers alongside the route, which control the train movements in their covered area and send MA to train with speed information and data regarding the route. The trains are equipped with sensors which determine their position between two balises. At this level, signals are optional, and tracks are divided into fixed blocks. The operational diagram of ETCS Level 2 (taken from [32]) is shown in Figure 3.2 where the two-way communication between the train antenna and RBC is illustrated. Based on the train direction and position, RBC sends the information to interlocking, which is the arrangement of switches, points, signals, and other arrangements to set the train route. At this level also, train detection and train integrity checks are performed by the trackside equipment beyond the scope of ERTMS/ETCS.

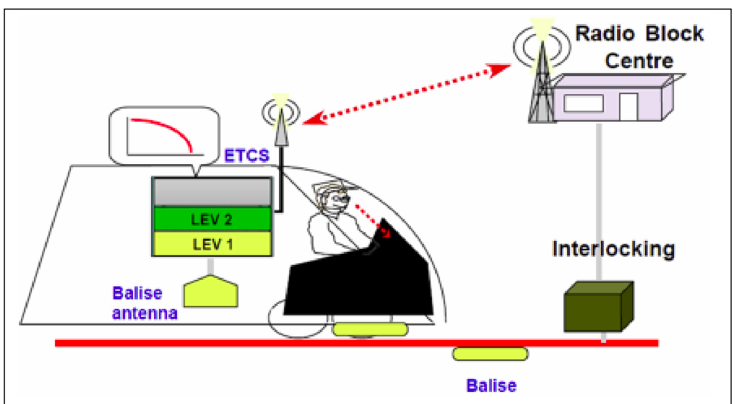


Figure 3.2: ETCS Level 2 diagram [32]

- ETCS level 3: This level is built around a full radio-based system without the involvement of any trackside equipment like balise. The position of trains is sent to RBC using GSM-R, and in return, RBC calculates the smallest possible distances

[3]Train integrity is the level of belief in the train being complete and not having left coaches or wagons behind. (https://www.era.europa.eu/content/set-specifications-3-etcs-b3-r2-gsm-r-b1_en)

between trains at any time. As there is no trackside equipment, trains have to guarantee their integrity. The tracks are split into moving blocks instead of fixed blocks. Virtual balise is a concept replacing non swichable physical balises by a functionnaly equivalent components. It can be implemented usign GNSS[4] technology, but some industrial solution are based on mecanical odometry. Moreover, the moving block technology applied by metros is not using satellite postitioning. The virtual balise concept allows building hybrid ERTMS3 designs, but can be used in other industrial context. Diagram from [102], illustrating the concepts of fixed block and moving block is shown in Figure 3.3 where in level 2, tracks are divided into fixed blocks called sections and the communicated Train 2 has authority to move until an assigned section. The moving block in level 3 is illustrated without any sections. The MA of Train 2 based on its actual distance from the train ahead, and this distance is calculated continuously. In level 3, train detection and integrity checks are within the scope of the ERTMS/ETCS.

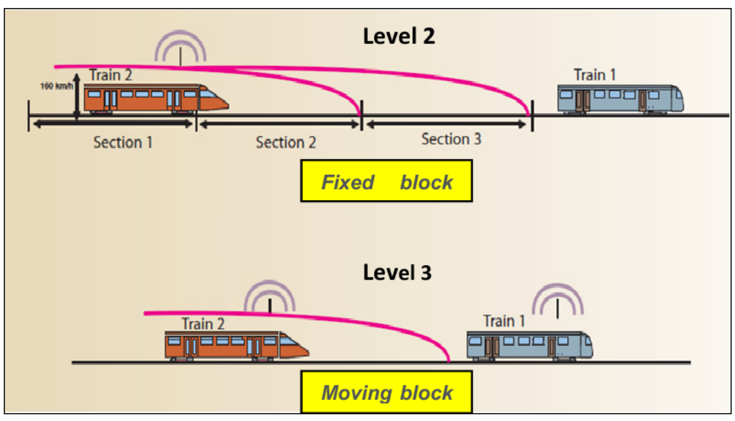


Figure 3.3: ERTMS level 2 and level 3 fixed block and moving block concepts [102]

ERTMS/ETCS levels 1 and 2 are already operational. However, ETCS Level 3 is still in the design and standardization phase. In this thesis, we use a case study following Hybrid ERTMS/ETCS Level 3 [9]. Hybrid level 3 uses fixed virtual blocks for the separation of trains. The trains are fitted with a train integrity monitoring system (TIMS) which reports their position and integrity. A limited installation of trackside train detection equipment is used for the separation of trains that are without TIMS. Also, in this level, trains which do not report their integrity can still be authorized to run on railway lines. A line (track) is divided into sections known as Trackside Train Detection (abbreviated as TTD). A TTD is further divided into subsections called Virtual Sub-Section (VSS). The ERTMS hybrid level 3 section conventions are illustrated in Figure 3.4 with TTDs and VSSs. A TTD can be free or occupied, and a VSS can be free, occupied, ambiguous, or unknown. A section is free when there is no train located. The occupied state on TDD is achieved when a train is located, and for VSS, it is achieved when a train position is reported from the trackside and there is no other train located in the rear of this train on the same VSS. A VSS is ambiguous when the trackside has information from a position report that a train is located on the VSS, and the trackside does not have any surety that no other train is located in the rear of this train on the same VSS. A VSS is unknown when there is no information about a train position report from the trackside, but it is not certain that the VSS is free. These states are illustrated in Figure 3.4 where the first VSS is free as there is no train located on it. As per ETCS rules, if all the VSSs of a

[4]Global Navigation Satellite System. https://www.euspa.europa.eu/european-space/eu-space-programme/what-gnss

TTD are free, then the TDD is free, and if a VSS is occupied, ambiguous, or unknown, then TTD is occupied.

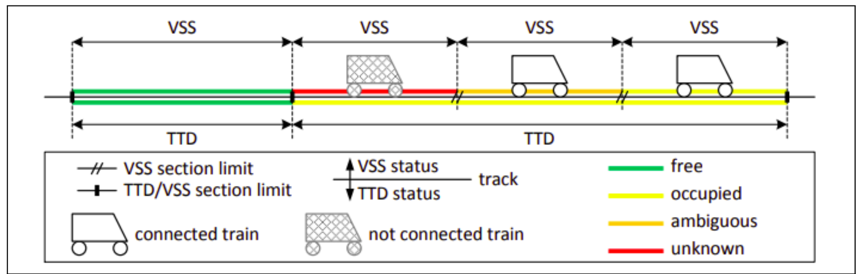


Figure 3.4: Hybrid level 3 section conventions [9]

Figure 3.5 shows VSS state transitions defined and explained in Hybrid ERTMS/ETCS Level 3 Principles document [9]. These transitions are based on reported train information and information from the trackside. For example, the transition from a free state to an occupied takes place when a train (integrated) is reported from the trackside on VSS; and similarly, from an occupied state to a free state takes place when an integrated train reports that it has left the VSS. Another example is the transition from an occupied state to an ambiguous state when a train loses its integrity or does not report integrity. More detailed information on VSS state machine transitions is available in document [9].

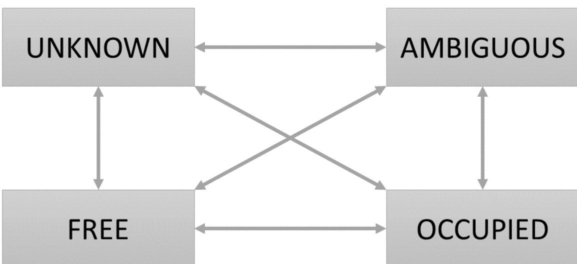


Figure 3.5: VSS State Machine [9]

In this thesis, we define the dynamic semantics of the DSL using existing ERTMS/ETCS specifications that have been proved. The ABZ 2018 conference [49] invited contributions regarding the formal modeling of ERTMS/ETCS level 3, and Mammar et al. [87] provided a B model of the hybrid ERTMS/ETCS level 3 standard. We re-use this model in our approach as a classical-B artifact which consists of four components: an abstract machine (M0) and three refinements (M1, M2, and M3).

3.2 UIC Standard documents

UIC, International Union of Railways [33] is an international body for rail transport. It supports many aspects of the railway industry, such as signaling, safety, standardization, etc. In this section, we briefly cover two standard documents provided by UIC.

RailTopoModel (RTM) [68] is an International Railway Standard (IRS 30100), developed with the contribution of several railway infrastructure managers and industrial companies. It defines and describes the structure of a railway network together with the physical installations that it manages. The generic description of the railway topology is considered the basic part of the RailTopoModel; it applies two modeling principles: topology of the railway network and multilevel architecture [124]. RailTopoModel is based on a UML class diagram divided into four packages shown in Figure 3.6. The packages

are Base, Topology, Net Entity (Note: Net is the short term used for the network in RailTopoModel), and Positioning System:

Package	Colour code	Main element(s)
Base	Grey Network	Network, LevelNetwork
Topology	Yellow *NetElement*	NetElement, Relation, CompositionNetElement
Positioning Systems	Green PositioningSystem	PositioningSystem, IntrinsicCoordinate
Net Entities	Light blue NetEntity	LocatedNetEntity, EntityLocation

Figure 3.6: RTM Packages [68]

- Base: The base package is centered around the Network and LevelNetwork classes. The class LevelNetwork provides different levels of network abstractions like macro, meso, and micro. The class Network in the base package contains all the network resources (i.e., Net Elements, Net Entities, and Associated Positioning System) issued from the other packages in the model.
- Topology: The classes in the Topology package define networks (logical paths) like Net Elements and their relations.
- Positioning System: The Positioning Systems package in the RailTopoModel specifies the network topology's positioning system (intrinsic, linear, or geographic). The coordinates of Net Elements and Net Entities are also defined using the Positioning System.
- Net Entity: The Net Entity package defines the installations of entities on the Topology. This package can be extended by concrete concepts like Tracks, bridges, and Interlocking Net Entities (i.e., railway signals).

The complete information about the four packages and other specifications of RailTopoModel is available in the IRS 30100 document [68]. RailTopoModel is not a usable format as it is a logical object model, and it cannot be used as a data format as it is [113]. For using RailTopoModel, we need to extend it by introducing more concrete concepts that someone can see in RailML [26] and EULYNX [11]. Figure 3.7 shows the Net Entity package of RailTopoModel, where concrete concepts are illustrated as shaded classes which are not part of the generic RTM model.

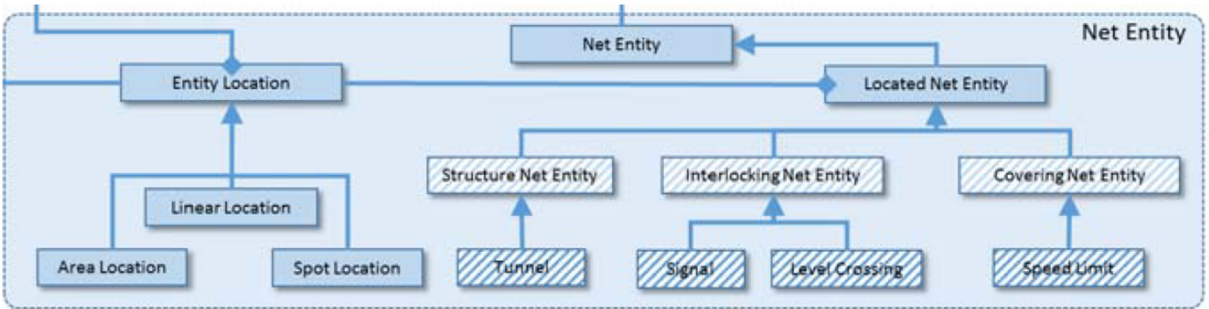


Figure 3.7: Classes of Net Entity Package from RailTopoModel [68]

In 2021, RTM was re-branded as RailSystemModel (RSM) [27]. RSM gives a global perspective of railway systems and corresponding operations to support and facilitate railway sector development. It is a cross-domain inter-operable, project-independent, and implementable model for expert projects like, i.e., EULYNX [11], IFC Rail [15], etc. The re-branded RSM is composed of two sets of packages containing abstract concepts called high-level classes. The two packages are Infrastructure and Common, shown in Figure 3.8.

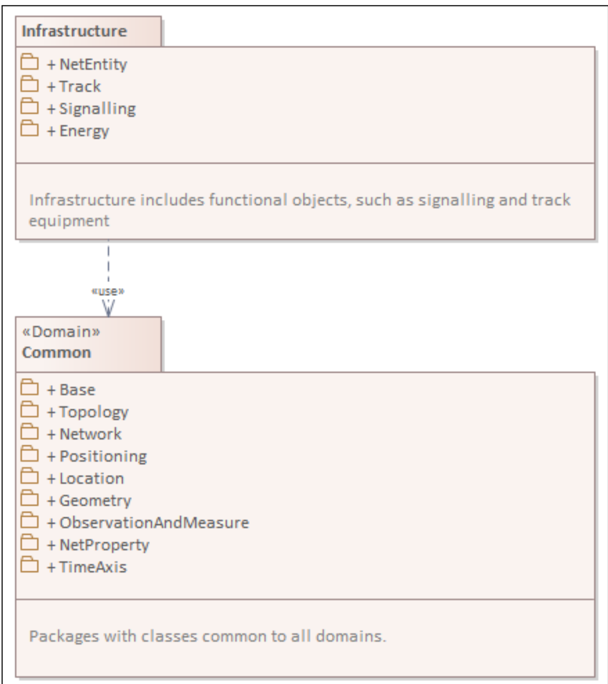


Figure 3.8: Overview of RSM Packages [27]

- Infrastructure Packages: The packages in this set are NetEntity (same as RTM), Track, Signalling and Energy. These packages include classes for functional objects like signalling equipment, track work and location of resources in the network.

- Common Packages: It includes packages with classes common to all domains like: Base, Topology, Positioning from predecessor RTM and newly introduced Location, Geometry, etc.

3.3 EULYNX

As an European initiative by fourteen Infrastructure Managers (IMs), EULYNX [11] is developed to standardize elements and interfaces of railway signalling systems. It defines an internationally standardized signaling system and regularly publishes specification documents.

EULYNX provides a data preparation (DP) model which defines a format for IMs and contractors to exchange information regarding signalling and data. The DP model is currently published as XML schemata, UML (XMI), C-sharp and HTML web view. The model contains ETCS concepts and is built in a model-based system engineering (MBSE) paradigm. EULYNX DP is aligned with RSM and can be considered as an extended concrete model of RSM. The introduction of EULYNX DP brought a strengthening of cooperation among the European IMs. Leveraging from RSM, it provides conformance to standardization and inter-operability of railway systems. DP is composed of three kinds of packages, which are generic, signaling, and several national implementations:

- Generic package: It is a single package that defines container classes from packaging, geo information, project management, project area, and common configuration.
- Signalling packages: These are packages that define classes having a function in signaling. Such packages include Level_crossing, Platform, Signal, Track, Train_detection, ETCS, Route, etc.
- National implementations: There are currently fourteen national IMs involved in EULYNX, but to date, EULYNX only provides specialized classes for six national IMs. The data of each IM is included in a separate package containing national and ETCS concepts. These national implementations include DB Netz, Network Rail, ProRail, Rete Ferroviaria Italiana (RFI), SNCF Réseau, and Trafikverket (TRV).

In this thesis, we have the option to use RailML or EULYNX for the meta-model of railway DSL. We chose the EULYNX, and our choice is motivated by many points. First of all, EULYNX, being standard, is accepted by multiple IMs. Secondly, it is aligned with RSM and the integration of ETCS concepts. Thirdly, it is also provided as an UML model, which can be easily adapted as an Ecore class diagram for DSL meta-model, and Meeduse translates it into the formal B Method. We use a subset from the EULYNX containing selected ETCS and RSM concepts. The subset is small and specific to the ERTMS/ETCS hybrid level 3 case study as compared to the whole EULYNX model.

Figure 3.9 shows the first-level classes from our subset of EULYNX that we use in this thesis. First of all, we defined a class DocumentRoot as a root class to contain the two high-level container classes of EULYNX: RSMEntities (Container of RSM concepts) and DataPrepEntities (Container for objects in the EULYNX DP). In the subset, we use three kinds of entities that are defined classes associated with class DataPrepEntities and can be further extended as general signaling concepts or concepts from a national domain, e.g. NR, ProRail, DB, SNCF, etc. The first one is the Abstract class AssetAndState, which can be extended as a class of pair (asset and its state).

The second one is the Abstract class TrackAsset, which can be a physical or virtual element that is relevant to signaling. This class TrackAsset is specialized by class TvpSection, which is a virtual track asset, and it corresponds to the section of track that must be proven vacant for safe train operations.

The abstract class RouteBodyProperty is the third one which is the base class for additional route body properties, and in this subset, we use the class SectionList (List of sections included in the route body) to specialize it. SectionList has a one-to-many

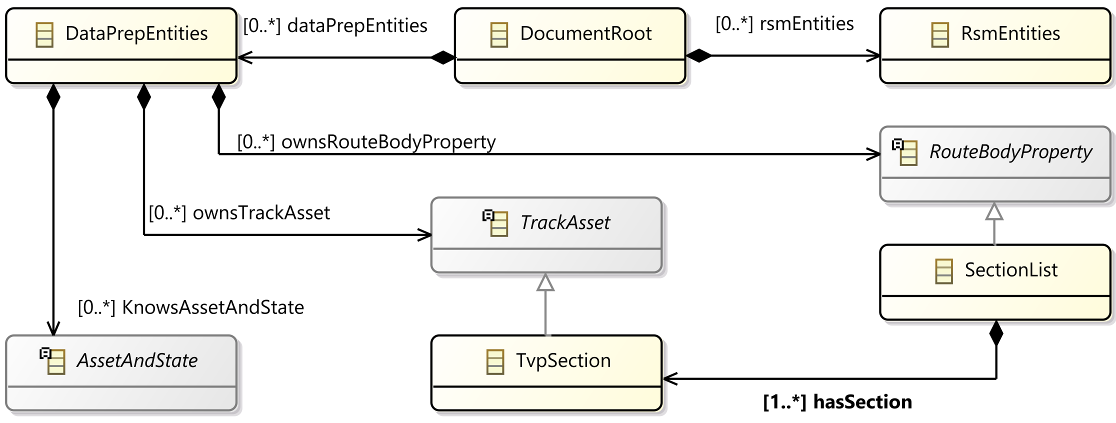


Figure 3.9: First-Level Classes from EULYNX subset

association to class TvpSection called hasSection which includes at least one or more track sections in the route.

3.3.1 RSM Concepts

Figure 3.10 shows the ecore class diagram of RSM concepts from EULYNX subset. These classes provide abstract infrastructural concepts from RSM common packages and can be aligned with concrete topological concepts of railway use cases such as ETCS. In the following, we discuss these RSM classes of the EULYNX subset.

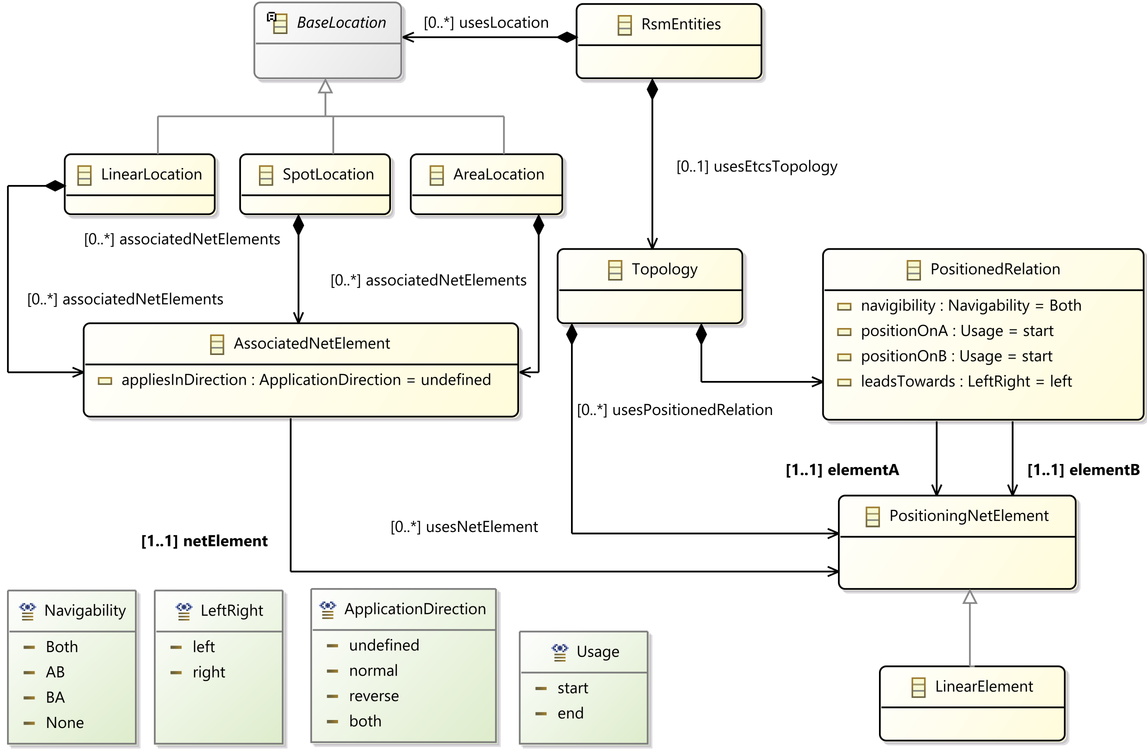


Figure 3.10: RSM Classes from EULYNX subset

3.3.1.1 RsmEntities

It is the container for entities defined in RSM namespace. Its main subject is topology and topography, and many associations from this class are linked with class Topology. In the thesis, we only use the association for the ETCS topology called usesEtcs, which is a zero-to-one (0 to 1) association with class Topology. Many other associations also originates from RsmEntities class to other classes like Unit, BaseLocation, Signal, RouteBody, etc, but we only use the association usesLocation to BaseLocation class.

3.3.1.2 Topology

It is the class for topological components where users can define separate containers for rail, cabling, energy, and other topologies. There are two associations that originated from this class: (i) usesNetElement to class PositioningNetElement and, (ii) usesPositionedRelation to class PositionedRelation.

3.3.1.3 PositioningNetElement

In RSM, NetElement defines nodes in the connectivity graph representing the topological network; and a PositioningNetElement is a topological element connected and oriented in the network by means of oriented relations. In this subset, we use LinearElement to extend the PositioningNetElement, which is a high-level abstract class for linear entities like track, line, or route.

3.3.1.4 PositionedRelation

In RSM, class Relation is the base class for defining edges in the connectivity graph representing the topological network. In this thesis, we use the class PositionedRelation, which extends the class Relation. PositionedRelation defines an oriented relation between two PositioningNetElements with associations: elementA and elementB. This class has four attributes. Attribute navigability indicates the direction of possible travel between elements A and B. Attributes: positionOnA and positionOnB show whether the relation is with the start or end of elementA and elementB respectively. Attribute leadsTowards shows the positioned relation connected either to the left or right branch.

3.3.1.5 BaseLocation

It is an abstract class that provides basic information about the location in the network topology information. The location is an abstraction in topology that is used by a concrete concept and has no link with information about the position on Earth. It contains three kinds of locations: class SpotLocation, LinearLocation, and AreaLocation. All three classes have a zero-to-many association to class AssociatedNetElement called associatedNetElements.

3.3.1.6 AssociatedNetElement

This class associates a location to an element of the topology and has an association called netElement towards the class PositioningNetElement. Class has an attribute appliesInDirection which Indicates the direction in which the function of the object along the linear element applies. It is not used when irrelevant or without meaning to the context of the object.

3.3.2 ETCS Related Concepts

ETCS related concepts in EULYNX are included in the signaling domain as well as national implementations. We choose those concepts that are useful for our application during this thesis. ETCS classes from the subset are shown in Figure 3.11 and discussed below.

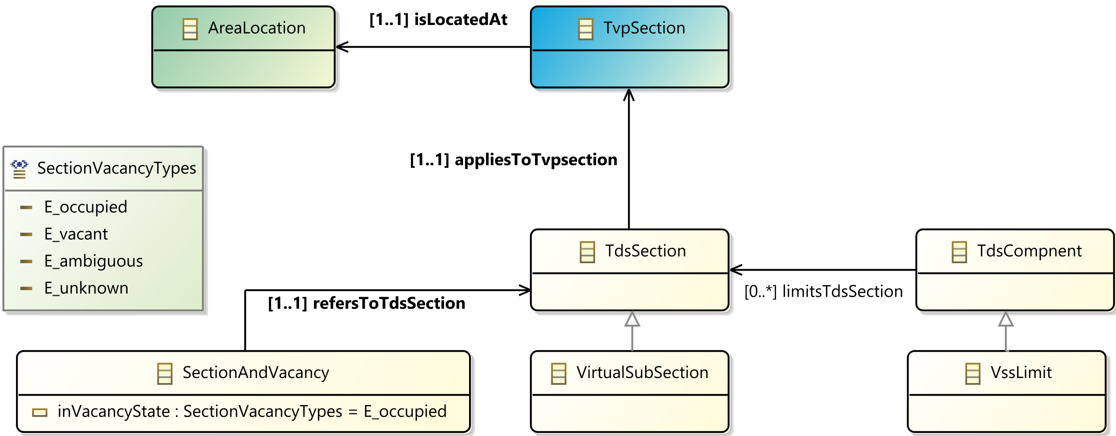


Figure 3.11: ETCS Classes from EULYNX subset

3.3.2.1 TdsSection & VirtualSubSection

TdsSection is a logical section of the train detection system and a virtual track asset. This section is associated with the class TvpSection using the association appliesToTvpSection. We relate the TdsSection to a TTD (Trackside Train Detection) of ERTMS/ETCS. Class TdsSection is extended by class VirtualSubSection (or VSS in ETCS). A VSS is a subdivision of TTD used for positioning trains by means of reported position and length.

3.3.2.2 SectionAndVacancy

Class SectionAndVacancy extends class AssetAndState and is a pair: section and vacancy. It expresses whether a train detection section is vacant or occupied and is associated with TdsSection using association refersToTdsSection. It corresponds to states of TTDs and VSSs in ERTMS/ETCS. In EULYNX, it has an attribute of enumeration type called SectionVacancyTypes with values occupied and vacant. EULYNX allows us to extend this list with extra states. We added two more states (ambiguous and unknown) for ERTMS HL3 and tagged the values with *E_* to be distinguished in our Ecore class diagram. Note that the values *E_ambiguous* and *E_unknown* are only additions from us in the subset as per the documentation of EULYNX.

3.3.2.3 TdsComponent & VssLimit

TdsComponent is a line-side physical track asset of a train detection installation, typically a rail joint or axle counter head. It is located at a TVPSection boundary spot such that the train detection can safely detect train vacancy. The TdsComponent is a delimiter to TdsSection and is associated with it using association limitsTdsSection. It is extended by class VssLimit, which is used to define the limit of a VSS.

3.3.3 Alignment between RSM and ERTMS/ETCS

In EULYNX, the concepts from its DP signaling domain and other national implementation are aligned with RSM using references. From our EULYNX subset, one example of this referencing is illustrated in Figure 3.11. We already discussed that class TdsSection corresponds to a TTD from ERTMS/ETCS, and it is associated one-to-one with class TvpSection using association appliesToTvpSection. Then the class TvpSection is associated one-to-one (using association isLocatedAt) with the class AreaLocation of RSM. This referencing between TvpSection and AreaLocation allows to extend the RSM with a functional signaling object (TTD of ERTMS/ETCS). EULYNX model provides many such referencing for the alignment between RSM and other signaling domains.

3.4 Conclusion

In this chapter, we discussed the railway standard initiatives related to the context of this thesis. First, we explored ERTMS/ETCS, which is a standardized European train control and signaling system. Then we discussed the UIC standard initiatives like RTM and RSM. We also discussed EULYNX, which is an initiative by Infrastructure Managers (IMs) to standardize elements and interfaces of railway signaling systems. EULYNX signaling domain is aligned with RSM, and it also provides packages for other national implementations. In this thesis, we use a subset of EULYNX, which is also illustrated and discussed during in this chapter. After getting an understanding of the technologies and notions mentioned in chapter 2 and the current chapter, we check state-of-the-art works and tools related to the idea of this thesis in the next chapter (4).

3.5 Résumé en français

Cette thèse fournit un cadre basé sur des DSL, qui permet la conception graphique de modèles ferroviaires basés sur des normes ferroviaires telles que RTM/RSM et EULYNX. Ce cadre prend en charge l'exécution du DSL ferroviaire avec une sémantique dynamique fournie par ERTMS/ETCS. ERTMS signifie European Rail Traffic Management System et ETCS signifie European Train Control System. ERTMS/ETCS est présenté comme une norme pour une plate-forme interopérable sûre et commune aux systèmes de gestion et de signalisation ferroviaire en Europe. Il est destiné à être adopté dans tous les pays européens et à remplacer leurs systèmes de signalisation nationaux [32]. Dans cette thèse, nous définissons la sémantique dynamique du DSL en utilisant des spécifications formelles de ERTMS/ETCS. La conférence ABZ 2018 [49] a fait un appel à contributions concernant la modélisation formelle de ERTMS/ETCS niveau 3, et Mammar et Al. [87] a fourni un modèle B de cette norme (dans sa version hybride ERTMS/ETCS). Nous réutilisons ce modèle B dans notre approche comme un artefact B-classique composé de quatre composants : une machine abstraite (M0) et trois raffinements (M1, M2 et M3). EULYNX [11] est une initiative européenne de quatorze gestionnaires d'infrastructure (GI), développée pour normaliser les éléments et les interfaces des systèmes de signalisation ferroviaire. Il définit un système de signalisation normalisé au niveau international et publie régulièrement des documents de spécifications. Dans cette thèse, nous utilisons un sous-ensemble d'EULYNX, qui est également illustré et discuté dans ce chapitre.

Chapter 4

State-Of-The-Art

Previously, in chapters 2 and 3, we discussed the notions that are useful in this thesis. We got familiar with the computer technologies like DSLs (including static semantics and dynamic semantics) and B Method, which are used for modeling and verification, respectively. We also discussed the railway standards like ERTMS/ETCS and EULYNX. In this chapter, we evaluate the state-of-the-art works and DSL tools that support the modeling and verification of railway systems and the use of railway standards in their approaches.

4.1 Introduction

In the industrial context of railway signaling, CENELEC [4] norms are considered, especially 50128[1] and 50129[2]. CENELEC 50128 recommends the use of FMs for critical software development, while in CENELEC 50129, a graphical description of the system, structured specification, and formal or semi-formal specification are highly recommended. As mentioned in chapter 1, our proposed framework mixes FMs and DSLs where railway models are graphically illustrated using domain-specific notations, and B Method formalizes these railway models. We believe that the use of the B Method and graphical domain-specific notations justify our work's position with CENELEC 50128 and CENELEC 50129 norms, respectively.

In the last decade, several approaches like [38], [43], [74], [75], [109], [115, 116] and tools such as RaIL-AiD [28], SafeCap [65, 67], OnTrack [76], have been presented for railway systems ranging from designing railway topologies to interlocking systems. In this chapter, we analyze and evaluate these works based on the following questions:

- Is the approach DSL based?
- Does the approach provide a graphical representation?
- Does it follow ERTMS/ETCS and RailML/EULYNX?
- Does it provide verification and validation?

Table 4.1 compares the state-of-the-art approaches, where they are first compared for their semantics (static and dynamic). Our second criteria for the comparison is the syntax for modeling of railway system provided by the approach. The syntax is distinguished as either graphical or textual syntax. Railway standards are the next criteria where we compare the state-of-the-art on their coverage of RailML/EULYNX and ERTMS/ETCS.

[1] https://standards.globalspec.com/std/2023439/afnor-nf-en-50128
[2] https://standards.globalspec.com/std/10280790/dsf-fpren-50129

Table 4.1: Comparison of State-of-the-Art Approaches

Approaches	Semantics		Syntax		Standards		V & V
	Static	Dynamic	Graphical	Textual	RailML/EULYNX	ERTMS/ETCS	
Bjørnar *et* al. [38, 86] RailCOMPLETE [16]	✓		✓	✓	✓		P
Chiappini *et* al. [50]	✓	✓	UML-based	✓		✓	P
Rail-AiD [28]	✓		✓		✓	P	P
SafeCap [29, 65, 67]	✓	✓	✓				✓
James *et* al. [75] OnTrack [76]	✓		✓				P
James *et* al. [74]	✓	✓	✓	✓		P	✓
Vu *et* al. [115, 116]	✓	✓		✓		P	✓
Svendsen *et* al. [109]	✓	✓	✓				✓
Idani *et* al. [62, 63]	✓	✓	✓	✓		P	✓

Yes= ✓
Partially covered= P

Though RailML has been used by some of the approaches, most of the European infrastructure managers do not agree with how it has been managed and do not consider it as standard. Note that we do not use RailML in this thesis. The last criteria for comparison are V & V (verification and validation). We included nine state-of-the-art approaches in this comparison, which are briefed and listed below:

- Bjørnar *et* al. [38] provides a way to extract railway models from CAD (Computer Aided Design) railway designs. Their tool named RailCOMPLETE [16] is integrated with industrial CAD software where they use RailML as the basis of representation for designing railway infrastructure. They use logic programming methods for the formalization of railway layout and interlocking.

- Chiappini *et* al. [50] provides a methodology for the formalization and validation of the ETCS specifications going from the informal analysis of the requirements, to their formalization and validation. They also formalized a subset of the ETCS specification.

- Rail-AiD [28] (Railway Infrastructure and Layout Aided Designer) is a framework that provides an editor for modeling railway signaling systems. The tool allows the design of railway infrastructure topology based on RailML and provides the corresponding data in tabular form. Also supports the import and export of railway designs in an interchangeable format. The tool does not provide verification activities.

- SafeCap [29, 65, 67] is an Eclipse-based toolset for modeling, simulation, and formal verification of railway networks. The toolset includes a graphical editor for creating and editing railway models, a simulator for testing the models, and a model-checker for verifying the safety properties of models.

- James *et* al. [75] provides a methodology for encapsulating formal methods within DSLs. They created OnTrack [76] tool, which allows the modeling and verification of railway schemes (models).

- James *et* al. [74] provides a way to formulate the safety properties of ERTMS/ETCS level 2 in Real-Time Maude [97]. This work also used the DSL tool OnTrack [76] for the topological design of railway models.

- Vu *et* al. [115, 116] present domain-specific languages: IDL (Interlocking Dynamic Language) and ICL (Interlocking Configuration Language) for defining railway models and properties. They also provide a way to verify their defined models.

- Svendsen *et* al. [109] provides an automatic way to generate train station models. Their approach uses the Train Control Language (TCL) [55], which is a domain-specific modeling language for modeling train stations and generating configuration code for controlling the signaling system on the station.

- Idani *et* al. [62, 63] provides a tool-based domain-specific approach for the modeling and verification of railway systems by combining MDE and FMs. The Eclipse-based approach allows the definition of the graphical design of railway DSLs, and then the B Method is used to define its underlying operational semantics and to guarantee the correctness of the model's behavior with respect to its safety properties.

4.2 Modeling

By modeling, we refer to the design of the system. The modeling can be done using graphical syntax or textual syntax, and the state-of-the-art approaches in Table 4.1 are compared on both of these criteria. The syntax criteria column in the table shows the modeling perspective of the state-of-the-art approaches for designing railway systems. The check-mark ✓(Yes) in sub-columns defines modeling perspective either as graphical or textual. 7 out of 9 evaluated approaches used graphical syntax for modeling. We can also find two different approaches in both industry and academia to model railway systems: UML-based and DSL based. The UML diagrams are UML (Unified Modeling Language) [96] based visual representations of the system. UML-based approaches are followed in railway giant companies like ALSTOM, and other consultants, where railway system requirements are defined using UML activity diagrams, and specific scenarios are illustrated in sequence diagrams. During our state-of-the-art study, we found that the methodology provided by Chiappini *et* al. [50] defines the railway concepts, behaviors, and scenarios using UML diagrams. The UML class diagram is used to represent the concepts and relationships between them. Then they extended the UML model with a textual language called Controlled Natural Language (CNL). It combines mathematical and English expressions to define constraints and other properties of the entities in the model. Though using English expressions can be easier for railway experts but combining them with mathematical expressions can be tricky. Also working with UML diagrams, railway experts need specialized knowledge of UML concepts.

To overcome such shortcomings, DSL-based approaches are there. They allow experts to model railway systems using the knowledge they have about their domain. DSL-based tools OnTrack [76] and SafeCap [29] are built using GMF [14], which allows the design of graphical railway models. The difference between them is that the Bjørner's DSL [40] is adopted as a meta-model for OnTrack. Another DSL tool, TCL (Train Control Language) [55], is also developed using GMF editor. The domain-specific approach for railway systems by Idani *et* al. [62] is based on EMF [52], and for the graphical editor, they used Eclipse Sirius[7], which can provide different views of a railway model.

Bjørnar *et* al. [38] uses RailCOMPLETE [16], allowing experts to design railway models. Like RailCOMPLETE, Rail-AiD [28] also supports the graphical modeling of railway signaling and systems. Figure 4.1 is the screenshot of the Rail-AiD editor where the left side shows the palette of objects to be drawn in the canvas (black screen in the middle) and the right hand of the screenshot shows the properties of a selected object. The table over the canvas shows the data of the drawn model. The DSL tools like RailCOMPLETE and Rail-AiD are built using general programming languages like Java and Java-script.

Apart from graphical railway DSL-based approaches, we also found some approaches that use textual syntax for specifying models or defining constraints and properties for a

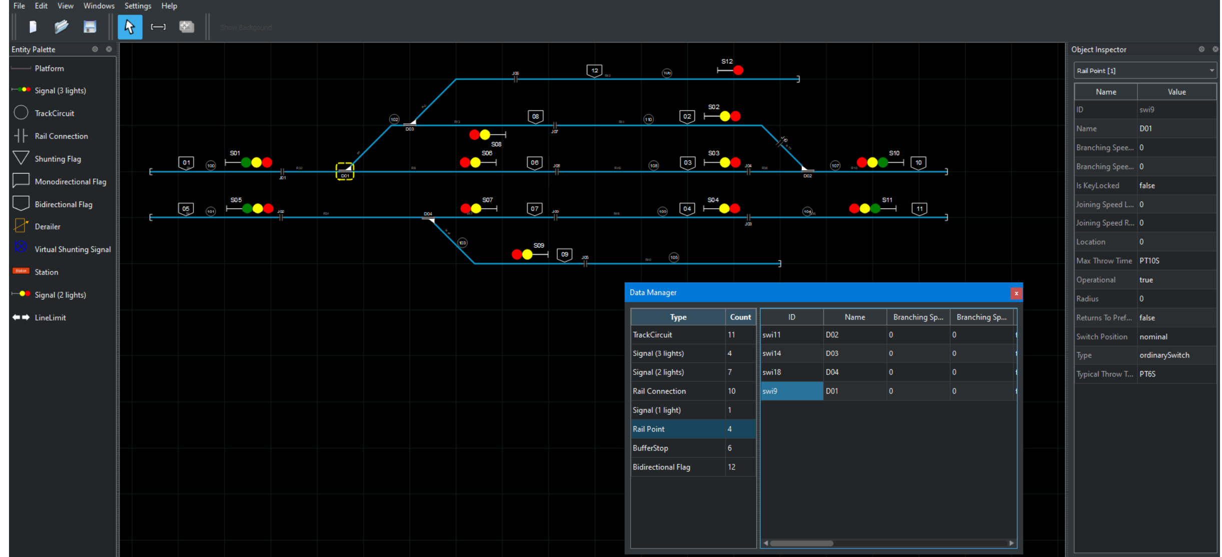


Figure 4.1: Screenshot of Rail-AiD editor

graphically defined model. For example, the work in Bjørnar *et* al. [38] is extended with the inclusion of a domain-specific controlled natural language called RailCNL [86]. It allows experts to write properties and constraints in a natural text language for a model designed in RailCOMPLETE. The two textual DSLs of Vu *et* al. [115, 116] are IDL and ICL. The former is a textual DSL used for specifying behavioral models and properties of the railway interlocking system (switches and points between tracks). Then, the latter is used to specify the interlocking system's configuration data (objects of the model).

Summary: The graphical models can be more understandable as they contain realistic domain notations which are easier to be understood by domain expert and they also help in the system's animation. We observed that 5 out of 7 graphical syntax-based approaches used the Eclipse platform (GMF/ EMF) for the DSL design. Tools like OnTrack, SafeCap, and TCL use GMF, while the approach by Idani *et* al. [62, 63] uses EMF with Eclipse Sirius. In this thesis, we follow the approach of Idani *et* al. [62, 63] for graphical modeling. This approach allows us to define railway models with different views using Sirius. Another reason behind using Sirius is the support for defining conditional styles with an OCL [6] like syntax.

4.3 Structure vs Behavior

4.3.1 Structure

We refer to the structure as static semantics. During the study of state-of-the-art, we found that every approach provides a structure to define the railway system. In Chiappini *et* al. [50], the structure of the system is defined with concepts like train, track, train position, onboard system, and trackside systems using UML diagrams. In the approach provided by Vu *et* al. [115, 116], railway systems are referred to as railway network layouts where tracks, signals, switches, marker boards, level-crossings are defined in RSL (RAISE Specification Language) [100]. Bjørnar *et* al. [38, 86] uses RailML [26] elements for the structure of the railway designs. Their design in the paper includes tracks, switches, and signals. The designs of railway topologies in Rail-AiD [28] are also based on the RailML. In James *et* al. [74] just a railway station model has been designed with tracks, marker boards, and switches using Maude [51].

James *et* al. [75] integrates OnTrack [76] where they use Dine Bjørner's DSL [41]

for designing railway models. The DSL is well-known in the railway community and contains concepts of a railway network with stations, lines, tracks, routes, etc. TCL[55] in Svendsen *et* al. [109] follows a meta-model containing the structure of railway system with main concepts like station, track, signal, switch, etc. The DSL of SafeCap [67] contains the concepts to define a railway system. The common concepts in their DSL are track, line, route switch (called point), section, and train detection circuit (called ambit). The DSL in Idani *et* al. [62] is dedicated to the railway topologies and signaling systems containing train, track, axle counter (a trackside train detection equipment), and Movement Authority while DSL in Idani *et* al. [63] follows the ERTMS/ETCS 3 concepts like Trackside and virtual block.

4.3.2 Behavior

We refer to the behavior as dynamic semantics, which corresponds to the provided execution. In the state-of-the-art approaches, we found that it is achieved using different ways, such as model execution, executing scenarios, and data (instance) generation. In Chiappini *et* al. [50], state machines are used to model the behavior of the executable methods of a class, and sequence diagrams are used to model scenarios requirements such as the exchange of messages between two trains and trackside radio devices. In the approach of James *et* al. [74], the designed railway model is executed using the commands of real-time Maude, which shows the behavior of the train moving on the corresponding track with the defined movement authority.

The behavior in Vu *et* al. [115, 116] does not provide train operations but includes an Interlocking Table Generator (ITG) which generates interlocking tables from the elements in the network layout. For generating the interlocking table, their RSL specification contains the execution command: make table *(mk_table(n))*. In Svendsen *et* al. [109], operational semantics are added to identify the unambiguous behavior of the model. Their approach generates train station models based on user-specified properties and the train operations are defined in TCL [55] to move it into or out of the generated station model.

SafeCap [29] tool is programmed in a way that one can include trains in their designed railway topology and using train operations, one can simulate it. Their DSL is not formally defined, but a B model is exported based on the defined railway model. Then the train operations are introduced in the B model (B machine) alongside the information of static aspects designed in the editor. As per our knowledge, these actions are performed outside the tool. On the contrary, the DSL of Idani *et* al. [62, 63] is translated into B specification using Meeduse [64] and then operations for safe train movement (train moving within movement authority) are manually defined in B specification. These formally defined operations are executed within the tool, and their behavior can be observed graphically.

4.3.3 Discussion

The semantics column of Table 4.1 shows whether the approaches provide structure (static) and behavior (dynamic) or not. In the current state-of-the-art study, we found that all approaches (either DSL or not) provide the structure and have the same concepts in the structure of their railway topology, like tracks, switches (points), signals, etc. Some of them (SafeCap [65], Idani *et* al. [62, 63], and Vu *et* al. [115, 116]) have their own way of defining their structure while others follow some specific provided structures like RailML [26] or Dine Bjørner's DSL [41]. Regarding the behavior, we looked at how the execution is provided by the tools. Most of the tools use general programming for the execution, which includes generating models and train movement operations. James *et* al. [74] and Idani *et* al. [62, 63] use formal methods (FM) to define the train movement

operations where former uses Maude and later uses the B method. In this thesis, we are intended to provide an executable DSL. Regarding the structure, we will use a specific provided structure (a subset from EULYNX), and for the behavior, we complement the work of Idani *et* al. [62, 63]. This allows us to execute formally defined train operations on the structure provided by EULYNX.

4.4 Standards

The Standards column in Table 4.1 is further classified into two sub-columns: RailML/ EULYNX and ERTMS/ETCS. Here, we put the criteria for using RailML alongside EULYNX as both of them can be used in the same way to provide the structure of the railway system. RailML has the ability to implement RTM while EULYNX includes the RSM. From state-of-the-art, we found that none of the approaches use EULYNX. The railway infrastructure models in Bjørnar *et* al. [38] complies with the RailML. The models designed in Rail-AiD [28] tool conform to the RailML. In Rail-AiD, RailML file formats can be exported and imported to and from other tools.

Regarding the ERTMS/ETCS, it is followed by Rail-AiD [28] just for track-side data preparation. Rail-AiD did not use other applications of ERTMS/ETCS like train movement, route calculations, RBC handovers, etc. The railway concepts in Chiappini *et* al. [50] are taken from ERTMS/ETCS. The approach is applied to a large piece of subset 26 of ETCS specifications[3]. The railway system which is modeled textually in James *et* al. [74] follows ERTMS/ETCS level 2. In Vu *et* al. [115, 116], the interlocking tables generated are compatible with ERTMS/ETCS 2. Idani *et* al. [63] used a part of concepts from ERTMS/ETCS level 3.

Summary: The RailML is just used by Bjørnar *et* al. [38] and Rail-AiD [28] and EULYNX is considered by none of the other approaches mentioned in this state-of-the-art. ERTMS/ETCS contains a wide range of specifications and concepts for the design of a railway system. Following all the dimensions of ERTMS/ETCS by an approach is hardly possible. From the state-of-the-art, only Chiappini *et* al. covered a large subset of ERTMS/ETCS and all other approaches which followed ERTMS/ETCS only covered it partially and are marked (P) in the Table 4.1. In this thesis, we use EULYNX to provide the structure of the railway system in our DSL, and we will use ERTMS/ETCS as a way to provide the operational rules within the structure.

4.5 V & V (Verification & Validation)

We checked the availability of verification and validation in the current survey of state-of-the-art approaches. In this sub-section, first, we will go through verification and then see the validation part later.

4.5.1 Verification

Verification is a way to prove or disprove the correctness of an algorithm in a system with respect to specific properties, using formal methods [57]. In the railway domain, formal techniques like logic programming, constraint solving, theorem proving, and model-checking have been used to verify safety and other properties. The evaluated approaches in this state-of-the-art alongside their used techniques are shown in Table 4.2 that verify properties like interlocking, topology, safety, and capacity. First, we briefly discuss these properties before going ahead.

[3]https://www.era.europa.eu/content/set-specifications-3-etcs-b3-r2-gsm-r-b1_en

Approaches or Tools	Formal Techniques and Tools	Properties
Bjørnar *et* al. [38, 86]	Datalog (Logic Programming)	Interlocking Topology
James *et* al. [75], OnTrack [76]	SPASS, eProver (Theorem Proving)	Safety
Vu *et* al. [115, 116]	RSL, RT-Tester, LTL (Model-checking)	Safety Interlocking
James *et* al. [74]	Maude, Maude LTL model-checker (Model-checking)	Safety
Svendsen *et* al. [109]	Alloy, Alloy analyzer (Theorem Proving)	Capacity
SafeCap [29, 65, 67]	B-method, Event-B, SMT-LIB compliant SMT solver, ProB (Constraint solving, Model-checking)	Topology Safety Capacity
Idani *et* al. [62, 63]	B-method Atelier B, ProB (Proofs, Model-checking)	Safety

Table 4.2: Properties and the used formal techniques

Safety is ensured by avoiding accidents: collisions (train conflicts on the track blocks, i.e., front-to-front[4] or rear-to-front[5]) and derailment (changes in switch position configurations while train movement; and also the excessive speed).

The topology of a railway network is verified to ensure static well-formedness. Static well-formedness avoids errors, like two branches of a switch are connected to the same track; or one end of a track is connected to two different switches. Figure 4.2 illustrates two anomalies that can be avoided by static well-formedness. In the first one, the left and right branches of *Switch01* are connected to the same track: *Track01*. In the second one, the left end and right end of a track *Track02* have the same switch *Switch02* which leads to a loop.

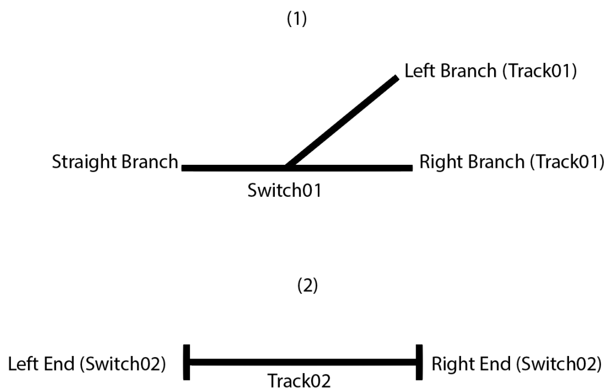


Figure 4.2: Errors to be avoided

The interlocking table of a railway model is verified to check whether the switch positions and signal states(lights) are correct for all the routes in the interlocking table.

[4]Front of 1st train colliding with the front of 2nd train.
[5]Rear of 1st train colliding with the front of the 2nd train.

Capacity properties are verified to check whether the railway model satisfies a number of trains at the moment. During the thesis, we will consider the verification of safety properties. The static well-formedness of the topology, Interlocking, and capacity are not addressed and are out of the scope of this thesis.

Below is the list of approaches (from Table 4.2) and their verification methods, briefly summarized:

- In Bjørnar *et* al. [38], two kinds of properties are verified: (I) Topology and the objects (elements) in it; and (ii) Interlocking. They used Datalog [112] (a declarative logic programming) for the verification.
- James *et* al. [75] wanted to verify a safety property in their specification from ETCS rules such that a track is not assigned to two movement authorities (MAs) at the same time (to avoid MA overlapping). They used SPASS [118] and eProver [106] theorem prover over the specification.
- In Vu *et* al. [115, 116], safety and well-formedness in interlocking is ensured. They used a static checker to check that the specification follows the well-formedness rules. Regarding the safety properties, first, they are expressed as formulae in the temporal logic LTL [59] and then used as input to the bounded model checker of RT-Tester [98].
- In the approach of James *et* al. [74], safety is ensured in a ERTMS/ETCS model. First, the model is defined using Maude and then Maude LTL model-checker [54] is used to verify that the model is collision free. They defined an invariant which is that *the two trains never fall below a minimum threshold*.
- The approach provided by Svendsen *et* al. [109] verifies the capacity of a generated railway model (station). The property is the requirement for the station to handle a number of trains simultaneously. The formal specification is defined using Alloy [70] and then extended with user-defined properties. Alloy analyzer is used to generate and analyze new models until the capacity property is satisfied.
- SafeCap [29, 65, 67] covers and satisfies many of the properties in the railway domain: static well-formedness properties for the definition of railway topology and operations to ensure safety in topology. It also provides support to reason about the capacity of the railway model. The approach uses constraint solving for the verification of static properties, taking Event-B as input. Verification conditions are derived from the definition of DSL schema by an automated tool and translated into input notation of an SMT-LIB [101] compliant SMT solver. If any property is not satisfied using SMT-solver, the tool uses ProB model-checker (which takes the classical B of the schema and the control table as inputs). ProB reports the sequence of steps that are not satisfied.
- In Idani *et* al. [62, 63], the safety property is ensured such that the railway model is accident-free. The translated B specification of the model is extended with safety properties and invariants. The approach uses Meeduse tool, [64] where the ProB model-checker is incorporated which allows the user to verify the safety during the animation. The B specification can also be checked for proofs, where Atelier B can be used to guarantee the correctness of the model's behavior with respect to the defined properties and invariants.

In Table 4.2, 5 out of 7 approaches verified safety properties. These approaches ensured safety properties by avoiding two trains on the same section or maintaining a threshold

distance between any two trains. The first step for verification in the above-mentioned approaches is almost the same: the formalization of semantics (defining the railway system in the formal specification). The difference is the use of different formal techniques, and even if the formal technique is the same but its application is different. Vu *et* al. [115, 116], James *et* al. [74], Svendsen *et* al. [109], and Idani *et* al. [62, 63], the static semantics are either defined or translated into a formal language (i.e., RSL, Maude, Alloy, B-method, etc.) and then, underlying formal techniques are applied for theorem proving or model-checking.

4SECURail [5] (a European project that aims to improve railway security and interoperability by using formal methods and cyber security techniques.) suggests using the B-method in their model-based development methodology, where model-checking is proposed for formal verification of the safety properties. From the discussed approaches for the verification of railway systems, it can be seen that model-checking has been used by most of the works. The approaches by SafeCap [29] and Idani *et* al. [62, 63] are using the B-method and model-checking (also ProB) for formal verification, which shows the similarity with the suggestion of 4SECURail.

4.5.2 Validation

Validation is to check that the system meets the user requirements. In contrast with verification, which is mostly done on the developer's end, validation is done mainly by the user. The validation can be done by reviews, simulation, animation, and testing. OpenETCS[6] [23] suggests using formal language and model-checking to inject a fault in the model and analyze the effect. James *et* al. [74] used this technique of error injection (using incorrect scenarios in the scheme plan to check that the errors can be identified or not) and also simulation.

Vu *et* al. [115, 116] uses a static-checker for the validation of specified data and interlocking specifications to check whether these are well-formed or not. The graphical design of railway models in Rail-AiD [28] favors the validation as the tool does not allow wrong designs that lead to errors.

The approaches of SafeCap [29] and Svendsen *et* al. [109] did not mention the validation explicitly, but their design and simulation of railway models favor validation. The work of Chiappini *et* al. [50] is based on a three-phase approach that goes from the informal analysis of the requirements, their formalization, and validation. The validation process is achieved by formalizing a realistic ETCS subset where different scenarios are checked against the requirements.

Idani *et* al. [62, 63] provides a graphical DSL for the railway system where the user can define models and validate specific scenarios in this model by animation. The animation is accomplished by triggering operations that were initially defined in the B specification of DSL.

4.5.3 Summary of V & V

We analyze state-of-the-art approaches for their application towards verification and validation. In the verification part, the approaches are listed, and their verification methodologies are discussed. We observed that the current trend is formalizing the system and verifying the properties with formal techniques using underlying tools. Regarding the validation, 7 out of 9 approaches favored it. EMF-based DSL approaches achieved validation mostly through simulation and animation. In Table 4.1, the V & V column is either

[6] A project to develop an integrated framework for modeling, development, validation, and testing of ERTMS/ETCS.

filled with P (Partially covered) or available ✓. All those approaches that achieved both verification and validation are marked as ✓. While those that either achieved verification or validation are marked as P.

During this thesis, we use the B method for the formalization of the DSL. First DSL meta-model is defined in EMF, and then the translation from EMF to B is done using B4MSecure [61], and this is written in Java. This transformation can be translated into B using Meeduse and can be verified, but this work is out of the scope of this thesis. In fact, this thesis uses existing proved B specifications and links them with the formal specification of the DSL. This linkage is written in B itself and contains linkage (gluing) properties and invariants, and we ensure that these gluing properties and invariants must satisfy the properties and invariants of the existing B specifications.

Most importantly, this thesis is focused on validation rather than verification. Our approach provides the graphical model for the existing proved B specifications using DSLs. Then these proved specifications are animated with their own defined operations (synced with states of DSL), and a railway expert can validate the proved B specification against the requirements.

4.6 Conclusion

In this chapter, we studied state-of-the-art approaches and compared them on the bases of the criteria: semantics, syntax, standards, and V & V. We found that most of the approaches provide graphical representation for illustrating railway topology. DSL-based approaches and tools: OnTrack, SafeCap, TCL, and Idani *et* al. [62, 63, 64] are built using eclipse EMF/ GMF, which shows the support of EMF-based tools for developing DSLs in model-driven engineering paradigm.

Regarding the standards, the support for RailML is only present in Bjørnar *et* al. [38, 86] and Rail-AiD tool. ERTMS/ETCS specifications are followed by James *et* al. [74], Vu *et* al. [115, 116], Idani *et* al. [63] and Chiappini *et* al. [50]. As far as verification is concerned, most of the approaches are focused on the verification of the safety properties. Safety properties are verified by ensuring that the railway model is free from collisions and derailment.

In this thesis, we provide a graphical DSL using EMF where the structure of the DSL is based on standard railway notations. We use the Meeduse tool for the formalization of the standardized DSL into B. Our approach facilitates the validation of existing ERTMS/ETCS proved B specifications using such a DSL. This validation is done using visual animation with the help of linkage specifications which are written in B itself. In the next chapter, we will discuss and illustrate the approach where linkage specifications are used to link existing B specifications with DSLs ultimately visually animating the B specifications.

4.7 Résumé en français

Dans ce chapitre, nous évaluons les travaux de l'état de l'art et les outils qui supportent aussi bien la modélisation que la vérification des systèmes ferroviaires. Nous évaluons également l'utilisation des normes ferroviaires dans ces travaux. Au cours de la dernière décennie, plusieurs approches comme [38], [43], [74], [75], [109], [115, 116] et des outils comme RaIL -AiD [28], SafeCap [65, 67], OnTrack [76], ont été proposés pour les systèmes ferroviaires allant de la conception de topologies ferroviaires aux systèmes d'enclenchements. Dans ce chapitre, nous analysons et évaluons ces travaux en nous basant sur les questions suivantes :

- L'approche est-elle basée sur un DSL ?
- L'approche fournit-elle une représentation graphique ?
- Est-ce qu'elle utilise ERTMS/ETCS et RailML/EULYNX ?
- Assure-t-elle la vérification et la validation ?

- Nous avons constaté que la plupart des approches fournissent une représentation graphique pour illustrer la topologie ferroviaire. Les approches et outils basés sur des DSLs, tels que OnTrack, SafeCap, TCL sont construits à l'aide d'Eclipse EMF/GMF. Cela montre l'utilité des outils basés sur EMF pour le développement de DSLs en IDM.

Concernant les standards, le support de RailML n'est présent que dans Bjørnar *et* al. [38, 86] et l'outil Rail-AiD. Les spécifications ERTMS/ETCS sont supportées par James *et* al. [74], Vu *et* al. [115, 116], Idani *et* al. [63] et Chiappini *et* al. [50]. En ce qui concerne la vérification, la plupart des approches se concentrent sur la vérification des propriétés de sécurité comme l'absence de collisions et de déraillements.

Chapter 5

Visual Animation of B Specifications

In the previous chapter, we studied state-of-the-art railway DSL-based approaches and compared them on different criteria, and we found that most of the approaches provide graphical (visual) syntax of the railway model. In this thesis, we also provide the graphical syntax of the DSL, and our objective is to use existing B specifications to provide the dynamic semantics (behavior) of the DSL to support the animation. Visual animation is an interesting way to do validation which inspects whether formal specifications meet the user requirements. Validation of safety-critical systems is an essential task because misunderstanding user requirements may lead to erroneous implementations. This task is known to be complex, especially for formal specifications, because of their mathematical notations. In fact, stakeholders who are not trained in formal methods (FMs), such as domain experts, find these notations difficult to understand. To address validation issues, several formal tools [78, 85, 117] provide graphical animation techniques. The intention is to exhibit a visual representation of data and behaviors of a formal specification, making it more readable for domain experts. In this chapter, we provide an approach where B specifications written independently of Meeduse team, are visually animated using DSLs.

5.1 Introduction

This thesis is more focused on validation of specifications than the verification of specifications. The validation is achieved through visual animation using DSL where Meeduse workbench applies B to execute the DSL. As per our knowledge, this technique has not been investigated before, mainly because, apart from Meeduse, none of the existing language workbenches provide execution features that are built on a well-established FM such as B. This claim is attested by the survey of Iung *et al.,* [69] in which the formal dimension is completely absent. This survey shows that among the 59 discussed language workbenches, only 9 provide support for verification, which is only done by testing.

Our approach in this thesis to visually animate B specifications, complement tools such as BRAMA [107], AnimB[1], B-Motion Studio [78], B-Motion Web [79], and VisB [119]. Existing tools have shown their strengths in practice, but still some concerns remain, which motivates our current approach. First, visual animation often requires specific skills such as in scripting languages (Flash, JavaScript, node.js), or in Scalable Vector Graphics (SVG files), etc. These technologies are not necessarily mastered by the formal methods expert and can be cumbersome to learn and use. In [77], Krings and Körner observed that "*When errors occur, it is not clear in which layer the cause is located in: is it an error in the B model? Is an SVG file broken? Is the config file incorrect? Is there a bug in the JavaScript code? As some errors are not reported, development can be cumbersome if one is not an*

[1] AnimB: https://wiki.event-b.org/index.php/AnimB

expert in all technologies". Second, mapping a specification to a domain representation is time-consuming and maybe error prone. Often this is done by a formal methods expert who is trying to document his own specification. Unfortunately, if the process of validation reveals some misunderstandings, not only the formal specifications must be reworked, but also the visual representations and the underlying mapping. Furthermore, the formal methods expert might not be familiar with the domain-specific notations.

To deal with the aforementioned concerns, we propose a new technique built on domain-specific languages (DSLs). In our approach, the domain-specific representations are designed using a DSL tool that is created in EMF [52]. The challenge is, therefore, to make the bridge between the DSL and the formal specification. To this purpose, we use Meeduse for the execution capabilities. Our approach favors the separation of concerns principle: the formal methods expert is responsible for the development of B specifications, and the domain expert provides useful input models thanks to the DSL tool. However, our approach involves another actor, the MDE expert, who is responsible for the development of the DSL tool.

5.2 Approach

This thesis is based on the application to the railway domain, but in this chapter, we illustrate the application of our approach to the example of Lift as proof of concept. We select the Lift as it is well-known in formal verification and validation [92].

5.2.1 The Lift Example

Leuschel *et* al., in [84] used many examples for the visualization of B specifications, where we found the example of Lift. Several rules about the Lift specification have been proved; for example, the lift may not move with doors open. The structural part of this specification is given in Figure 5.1 (named $\text{Lift}_{\text{existing}}$).

MACHINE
   $\mathit{Lift}_{\text{existing}}$
CONCRETE_CONSTANTS
   $\mathit{groundf}, \mathit{topf}$
PROPERTIES
   $\mathit{topf} \in \text{NAT} \wedge \mathit{groundf} \in \text{NAT} \wedge (\mathit{groundf} < \mathit{topf})$
CONCRETE_VARIABLES
   $\mathit{call_buttons}, \mathit{cur_floor}, \mathit{direction_up}, \mathit{door_open}$
INVARIANT
   $\mathit{cur_floor} \in (\mathit{groundf} \,..\, \mathit{topf})$
   $\wedge\ \mathit{door_open} \in \textbf{BOOL}$
   $\wedge\ \mathit{call_buttons} \in \text{Pow}(\mathit{groundf} \,..\, \mathit{topf})$
   $\wedge\ \mathit{direction_up} \in \textbf{BOOL}$
   $\wedge\ (\mathit{door_open} = \textbf{TRUE} \Rightarrow \mathit{cur_floor} \in \mathit{call_buttons})$
INITIALISATION
   $\mathit{cur_floor} := \text{groundf}$
   $||\ \mathit{door_open} := \textbf{FALSE}$
   $||\ \mathit{call_buttons} := \emptyset$
   $||\ \mathit{direction_up} := \textbf{TRUE}$

Figure 5.1: Lift example [84]

It represents one lift that moves between the ground floor (constant groundf) and the top floor (constant topf). Both groundf and topf are natural numbers, such that groundf is less than topf. Variable cur_floor gives the current position of the lift among the floors. Variables direction_up and door_open are booleans, representing whether the direction of the lift is up and whether the door of the lift is open respectively. The call buttons are represented with variable call_buttons, a set of natural numbers, without any distinction between the internal and external buttons of the lift cabin. The specification contains seven B operations that are not presented here for space reasons: move_up, move_down, reverse_lift_up, reverse_lift_down, open_door, close_door and push_call_button. We choose the lift specification because it is a pedagogical example used in the B-Book [35] and referenced in several papers [84, 119]. Furthermore, the example gathers several B data structures allowing us to illustrate various concepts of our approach. Note: The complete existing lift B specification including all the resources of this thesis are available in Meeduse Git Repository [20].

5.2.2 Proposed architecture

Our architecture is built on Meeduse Language Workbench (discussed in 1.2.1), which translates the semantics of DSL into a B specification as part of static semantics (structure). Then it requires to provide a B specification of the dynamic semantics (behavior). We propose to use an existing one that is already proved correct and provided by B method experts. Our approach is illustrated in Figure 5.2, which redefines the dynamic semantics part from Figure 1.2. Note that this approach for visual animation of B specification is part of our DSL-based Framework illustrated in Figure 1.3.

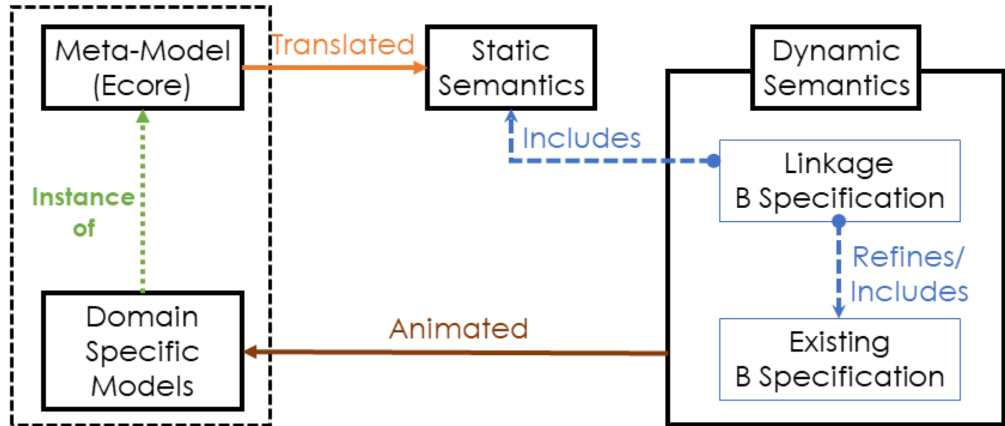


Figure 5.2: Proposed approach

Since the static semantics (B_{static}) of the DSL is written in B, therefore the remaining part of the approach is to link the existing B specification (B_{existing}) to B_{static}. To this purpose, we add a linkage B specification (B_{linkage}) along with B_{existing}. The linkage B specification (B_{linkage}) maps the concepts of B_{existing} to B_{static}. We propose two techniques: RefinementInclusion and InclusionInclusion. In the RefinementInclusion technique, B_{linkage} refines B_{existing} and includes B_{static}. In the InclusionInclusion technique, B_{linkage} includes both B_{existing} and B_{static}.

Note that B_{linkage} applies B invariants to map the variables of B_{static} to those issued from B_{existing}. By animating B_{linkage}, the states of both B_{existing} and B_{static} are modified, and ProB verifies that the mapping is preserved all along the animation. The task of Meeduse is to update the input model resource during the animation conforming to the state of B_{static}, which produces the expected visual animation.

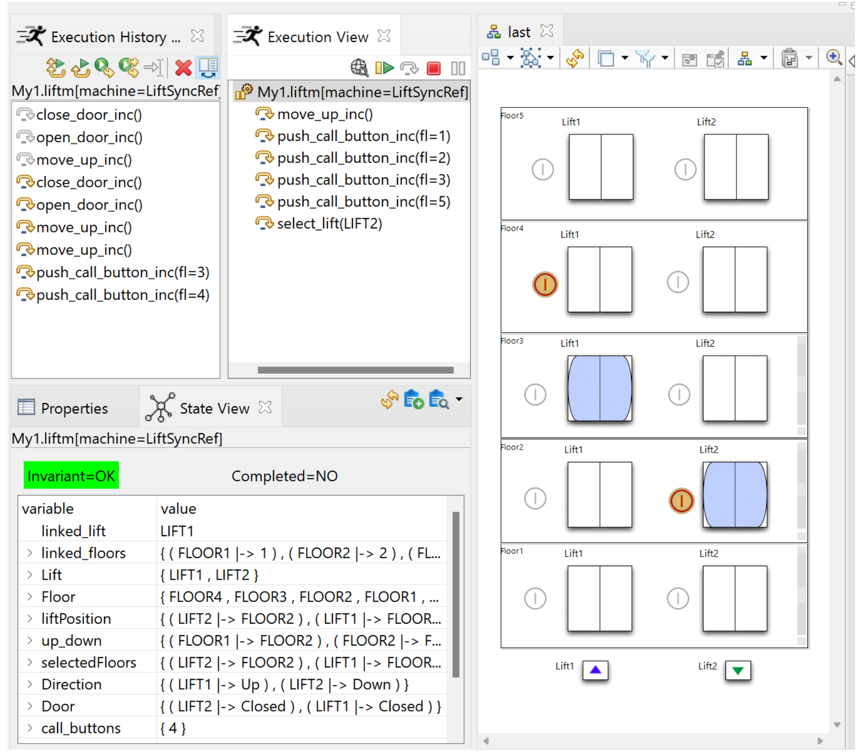


Figure 5.3: Visual animation in Meeduse

5.2.3 Illustration

Figure 5.3 shows a screenshot of Meeduse while animating an EMF-based DSL that represents the Lift example. The domain representation on the right-hand side of the figure can be realized by a domain expert using the modeling tool. The latter allows one to define buildings with several lifts. In this illustration, the model editor is developed with Sirius [7]. The other views are for interactive animation. The execution view shows the candidate operations computed by ProB from B_{linkage} , and the state view shows the current valuations of the various B variables. The execution history records the operation calls that led to the current state. After every animation step, the model resource is updated by Meeduse, and Sirius automatically renders the graphical model without the need for any specific implementation effort.

5.3 Designing a domain-centric visual animation

The previous section presented and illustrated a quick overview of the main concepts of our approach. In this section, we show, using the Lift example, how a domain-centric visual animation can be instrumented in Meeduse.

5.3.1 The Lift DSL

Figure 5.4 gives the meta-model of our Lift DSL. The three main concepts of the DSL are buildings (class Building), floors (class Floor) and lifts (class Lift). Each floor has at most one upper floor (association up) and one lower floor (association down). Association liftPosition gives the position of a given lift, and selectedFloors refers to the set of selected floors from a given lift. Each floor is composed of zero or many cabins (association cabins); by "cabin" we mean the interface for a lift at a given floor. The concept of cabin is useful for our DSL because it manages the graphical representation: the visual aspect of a cabin changes depending on the current position of the lift and the door state. Class Lift has an

attribute Direction that can be either Up or Down. Finally, attribute Door represents the door, which can be either Closed or Open. The model presented on the right side of Figure 5.3 is an instance of this meta-model. It gathers two lifts (Lift1 and Lift2) and five floors (Floor1, Floor2, Floor3, Floor4, and Floor5). Each floor has two cabins, corresponding to the two lifts. The arrow symbols at the bottom of the model show the direction of the lift. The direction of Lift1 is up while the direction of the Lift2 is down. The doors of both lifts are closed. The active button beside each door shows the selected floor for a given lift.

Note that the definition of the DSL is done independently from the existing B specification that one would like to visualize. The objective is to focus on the domain concepts and their relationships and build a DSL modeler that is useful for domain experts. For example, there is no explicit notion of call buttons in the DSL meta-model. This concept is somehow similar to the association selectedFloors from Lift to Floor.

5.3.2 Static Semantics

The extraction of B_{static} from the DSL is done by Meeduse, resulting in a B machine that covers the structure of the meta-model and gives several basic operations such as constructors, destructors, getters, and setters. Figure 5.5 gives the structural part of this machine (named $Lift_{static}$).

In the generated B specification, we can notice that there are no data structures generated for the Building and Cabin classes. These concepts are useful for the lift DSL but we do not require them in our B specification. In fact, Meeduse features an annotation mechanism that allows the user to select the concepts to be translated. We generated seven variables together with their typing invariants: Lift, Floor, liftPosition, up_down, selectedFloors, Direction, and Door. For these variables, corresponding invariants, initialization, and basic operations (setters, getters, constructors, deconstructors) are generated. The complete generated specification of $Lift_{static}$ including initialisation and operations, can also be found in Git repository [20].

5.3.3 Linking B data structures

Figures 5.1 and 5.5 gave respectively the existing machine ($Lift_{existing}$) and the static semantics of the lift DSL ($Lift_{static}$). In order to apply Meeduse for visual animation, one

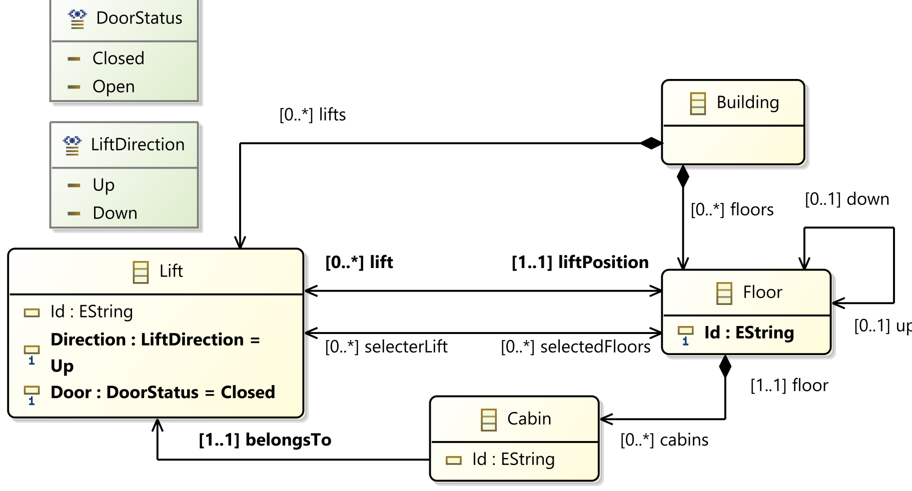


Figure 5.4: Lift Meta-Model

MACHINE
$\text{Lift}_{\text{static}}$
SETS
$LiftDirection = \{Up, Down\}$;
$DoorStatus = \{Closed, Open\}$;
$LIFT$; $FLOOR$
VARIABLES
$Lift$,
$Floor$,
$liftPosition$,
up_down,
$selectedFloors$,
$Direction$,
$Door$
INVARIANT
$Lift \in \text{Pow}\ (LIFT) \wedge$
$Floor \in \text{Pow}\ (FLOOR) \wedge$
$liftPosition \in Lift \rightarrow Floor \wedge$
$up_down \in Floor \rightarrowtail\!\!\!\!\twoheadrightarrow Floor \wedge$
$selectedFloors \in Lift \leftrightarrow Floor \wedge$
$Direction \in Lift \rightarrow LiftDirection \wedge$
$Door \in Lift \rightarrow DoorStatus$

Figure 5.5: Structural part of machine $\text{Lift}_{\text{static}}$

needs to create a third machine ($\text{Lift}_{\text{linkage}}$) that maps B data of both machines.

5.3.3.1 Mapping a class to a machine.

Machine $\text{Lift}_{\text{existing}}$ contains invariants and B operations to move a simple lift between the floors safely without any error. In machine $\text{Lift}_{\text{static}}$ multiple objects of class Lift can be created (abstract set LIFT and variable Lift). In this case, we consider that $\text{Lift}_{\text{existing}}$ will control only one instance of class Lift. Thus, we introduce in machine $\text{Lift}_{\text{linkage}}$ variable *linked_lift* to select the lift object that will be controlled by the existing machine:

Listing 1: EClass to BMachine

VARIABLE: *linked_lift*
INVARIANT: $linked_lift \in Lift$
INITIALISATION: $linked_lift :\in Lift$

5.3.3.2 Mapping a class to a set.

In $\text{Lift}_{\text{existing}}$ floors are represented with a set of natural numbers ($groundf\ ..\ topf$), and in $\text{Lift}_{\text{static}}$ they are defined with a dedicated variable representing class Floor. The mapping is then done as follows:

Listing 2: EClass to BSet

```
VARIABLE: linked_floors
INVARIANT: linked_floors ∈ Floor ⤖ groundf .. topf
INITIALISATION:
1.  ANY mapFloors WHERE
2.     mapFloors ∈ Floor ⤖ groundf .. topf ∧
3.      ∀ (f1,f2).(f1 ∈ Floor ∧ f2 ∈ Floor ∧ (f1 ↦ f2)
4.      ∈ up_down⇒ mapFloors(f2) = mapFloors(f1) + 1)
5.  THEN
6.     linked_floors := mapFloors
7.  END
```

The relation *linked_floors* is a total bijection meaning that every floor in the DSL must be linked to one value from (*groundf .. topf*) and vice-versa. The initialisation is done such that if a floor f2 is associated to a floor f1 via relation up_down then the value of mapFloors(f2) is equal to mapFloors(f1) plus one.

5.3.3.3 Mapping a single-valued reference to a set element.

In $\text{Lift}_{\text{existing}}$ the current floor of the lift is defined with variable *cur_floor*, which is an element of set (*groundf. .topf*). This concept is defined in the DSL with reference liftPosition, whose source and target classes have already been mapped with the rules above. The source has been mapped to a machine and the target to a set. To map *cur_floor* to reference liftPosition, we introduce the following invariant:

Listing 3: Single-valued EReference to a set element

```
INVARIANT:
   linked_floors(liftPosition(linked_lift)) = cur_floor
```

5.3.3.4 Mapping a multi-valued reference to a set.

Variable *call_buttons* of machine $\text{Lift}_{\text{existing}}$ is the set of pressed buttons that allows one to select the floors to which the lift is to be moved. This concept is represented in the DSL via reference selectedFloors. This leads to the following invariant:

Listing 4: Multi-valued EReference to a set

```
INVARIANT:
 linked_floors[selectedFloors[{linked_lift}]] = call_buttons
```

5.3.3.5 Mapping an enumeration to Boolean values.

In $\text{Lift}_{\text{existing}}$ the Boolean variable *door_open* defines the status of the lift door (false if the door is closed and true if it is open). This concept is represented in the DSL via attribute Door of class Lift, whose type is enumeration DoorStatus. The same case happens for attribute Direction and variable *direction_up*. Mapping these concepts to each other introduces additional invariants to $\text{Lift}_{\text{linkage}}$:

Listing 5: EnumType to Boolean values

```
INVARIANT:
 (Door(linked_lift) = Closed ⇔ door_open = FALSE)
 ∧ (Door(linked_lift) = Open ⇔ door_open = TRUE)
 ∧ (Direction(linked_lift) = Down ⇔ direction_up = FALSE)
 ∧ (Direction(linked_lift) = Up ⇔ direction_up = TRUE)
```

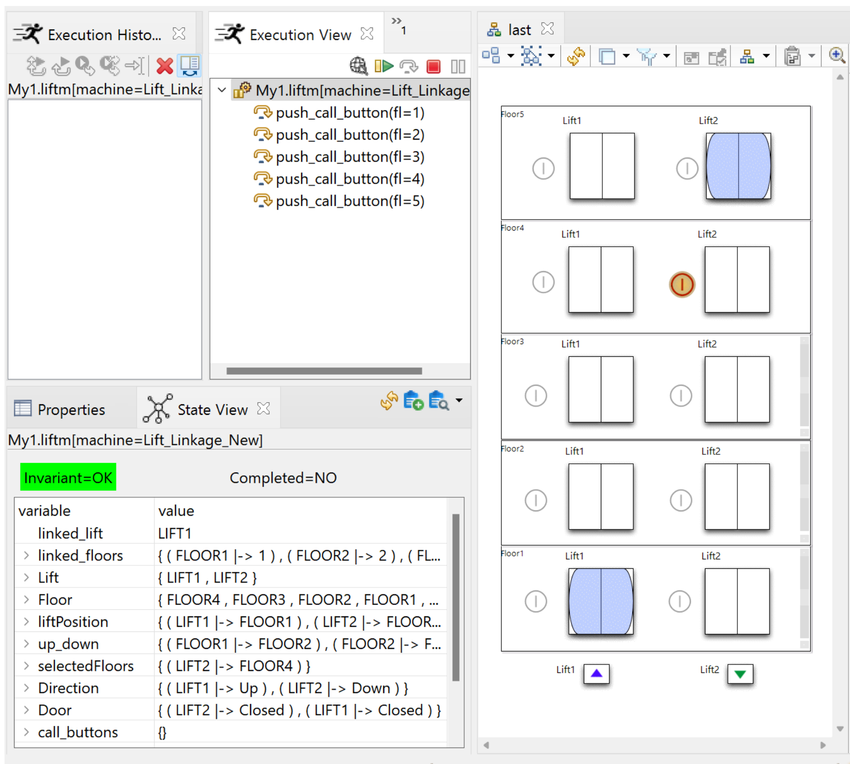


Figure 5.6: Meeduse after the initialization

5.3.4 Initialization

Once the data structures of both machines have been mapped to each other, using additional variables and invariants, we need to think about the initialization and the operations in order to keep the conformance of both models all along the animation. Machine $\text{Lift}_{\text{linkage}}$ includes machine $\text{Lift}_{\text{static}}$, which allows it to use the basic operations provided by the latter to update the state of its variables. $\text{Lift}_{\text{linkage}}$ includes or refines (depending on the strategy chosen by the user) machine $\text{Lift}_{\text{existing}}$. In both cases, the constants of $\text{Lift}_{\text{existing}}$ are visible and can be read by $\text{Lift}_{\text{linkage}}$. The initialization of the model is done as follows:

Listing 6: Initializing the model

PROPERTIES:

$\quad \mathbf{card}(groundf \,..\, topf) = \mathbf{card}(FLOOR)$

INITIALISATION:

$\quad SetDirection(linked_lift, Up);$

$\quad SetDoor(linked_lift, Closed);$

$\quad UnsetSelectedFloors(linked_lift)\ ;$

$\quad SetLiftPosition(linked_lift,\ linked_floors^{-1}(groundf))$

Constants *groundf* and *topf* are not assigned to values in the existing specification. However, the visual animation starts from a model built in the DSL tool and in which buildings and floors have already been created. Thus, the PROPERTIES clause indicates to ProB to choose any valuation of these constants that respects the existing number of objects defined in the DSL model. Regarding the INITIALISATION clause, in addition to the initialisation of the linkage variables (Cf. listings 1 and 2), it updates the DSL model such that it starts in a state that is conformant to the initialisation of machine $\text{Lift}_{\text{existing}}$ given in Figure 5.1. To this purpose we simply call basic operations provided by $\text{Lift}_{\text{static}}$

(*i.e.* `SetDirection`, `SetDoor`, `UnsetSelectedFloors` and `SetLiftPosition`). Figure 5.6 is a screenshot of Meeduse after the initialization.

The state view on the bottom left side of the figure shows that the linked_lift is LIFT1. Note that ProB gives the two possible initialisations of linked_lift (LIFT1 and LIFT2), and the user may select the one to be controlled with $\text{Lift}_{\text{existing}}$. In this case, we selected LIFT1. Besides, the DSL model deals with five floors, which are all mapped via linked_floors satisfying the underlying invariant. The liftPosition of LIFT1 is set to FLOOR1 because machine $\text{Lift}_{\text{existing}}$ starts with a lift that is at the ground floor. Figure 5.6 also shows the LIFT2, where its position is on FLOOR5 and it has been called on FLOOR4 (selectedFloors). Note at this stage, the specification only deals with linked_lift LIFT1.

5.3.5 Operations

We propose two strategies to build the dynamic semantics of the DSL by means of a linkage machine: (1) RefinementInclusion and (2) InclusionInclusion. In both strategies, the static semantics machine is included in the linkage machine.

5.3.5.1 RefinementInclusion.

In this strategy, the existing machine is refined by introducing the linkage variables and invariants, which suggests the refinement of the operations too. In Listing 7 below, we give the refinement of operation *move_up*. We copied, from $\text{Lift}_{\text{existing}}$, the body of the operation (from line `2.` to line `9.`) and we introduced a call to operation *setLiftPosition* (line `10.`). This basic setter of relation `position` is provided by $\text{Lift}_{\text{static}}$; its task is to change the position of the lift conforming to the linkage. The objective is to preserve the invariant of Listing 3, since only variable *cur_floor* is modified by operation *move_up*.

Listing 7: Refining operation *move_up*

```
OPERATIONS:
1.  move_up =
2.   SELECT
3.    door_open = FALSE
4.     ∧ cur_floor < topf
5.     ∧ direction_up = TRUE
6.     ∧ ∃ cc.((cc ∈ Z ) ∧ ((cc ∈ Z ) ∧ (cc > cur_floor) ∧ (cc ∈ call_buttons)))
7.     ∧ (cur_floor ∉ call_buttons)
8.   THEN
9.    cur_floor := ((cur_floor)+(1)) ;
10. SetLiftPosition(linked_lift,linked_floors⁻¹(cur_floor))
12. END
```

5.3.5.2 InclusionInclusion

In this strategy, the existing machine is included in the linkage machine so that its operations can be used without a need to copy its body like in the RefinementInclusion strategy. The idea is to synchronize the states of both machines during the animation. For every operation of the existing machine, we create a synchronization operation that manages the evolution of the two machines. For example, operation *move_up_inc* below calls both *move_up* and *SetLiftPosition*.

Listing 8: Synchronizing models

```
OPERATIONS:
move_up_inc =
 BEGIN
  move_up ;
  SetLiftPosition(linked_lift, linked_floors ⁻¹(cur_floor))
 END
```

Both strategies have their advantages. The InclusionInclusion provides a lightweight approach to the usage of operations. It is practical when the DSL or the existing machine is not yet finished and may be changed. The RefinementInclusion is better if one would like to continue the refinement process such that the variables from the existing machine are completely redefined using variables of the static semantics machine. This technique may be useful for producing step-by-step the dynamic semantics of a DSL from an existing machine that has already been proved correct.

5.3.6 Enhancements

The initialization provided in listing 6 initializes $\text{Lift}_{\text{static}}$ based on the initial state of $\text{Lift}_{\text{existing}}$. This approach is reasonable as far as the objective is visual animation − since we do not introduce any modification to the existing specification, we just visualize it. Nevertheless, the limitation is that every time we switch from one lift to another (*e.g.* from LIFT1 to LIFT2) we must animate the initialization with a different lift. The selected new lift is, therefore, re-initialized, which brings it back to the ground floor. A better solution would be to use $\text{Lift}_{\text{existing}}$ as a controller which can switch between the two lifts. To this purpose, we need to initialize $\text{Lift}_{\text{existing}}$ from $\text{Lift}_{\text{static}}$ and introduce an operation that allows one to select on-the-fly any lift without re-initializing it. The setter (setValues) of listing 9 is introduced within $\text{Lift}_{\text{existing}}$ in order to update its variables conforming to its invariants. This kind of setter can be generated automatically: every parameter is assigned to a variable and the precondition applies the invariants to the parameters.

Listing 9: Adding a setter to $\text{B}_{\text{existing}}$

```
OPERATIONS:
setValues
(call_buttons_,cur_floor_, direction_up_,door_open_) =
 PRE
  cur_floor_ ∈ groundf. .topf
   ∧ door_open_ ∈ BOOL
   ∧ call_buttons_ ∈ Pow (groundf .. topf)
   ∧ direction_up_ ∈ BOOL
   ∧ (door_open_ = TRUE ⇒ cur_floor_ ∈ call_buttons_)
 THEN
  cur_floor := cur_floor_
   || door_open := door_open_
   || call_buttons := call_buttons_
   || direction_up := direction_up_
 END
```

To initialize $\text{Lift}_{\text{existing}}$ from $\text{Lift}_{\text{static}}$ we define the substitution of listing 10 and we use it in the initialization clause of $\text{Lift}_{\text{linkage}}$. In this case, we are applying the Inclusion/Inclusion approach. Definition update_existing calls setter setValues based on the linkage invariants. Once the lift and the floors are mapped, this definition finds a valuation to the variables of $\text{Lift}_{\text{existing}}$ making them compatible with the valuations of $\text{Lift}_{\text{static}}$.

Listing 10: Definition for updating $B_{existing}$

```
DEFINITIONS:
update_existing ==
 ANY cur_floor_, door_open_, direction_up_, call_buttons_
 WHERE
  cur_floor_ = linked_floors(liftPosition(linked_lift))
   ∧ (Door(linked_lift) = Closed ⇒ door_open_ = FALSE)
   ∧ (Door(linked_lift) = Open ⇒ door_open_ = TRUE)
   ∧ (Direction(linked_lift) = Down ⇒ direction_up_ = FALSE)
   ∧ (Direction(linked_lift) = Up ⇒ direction_up_ = TRUE)
   ∧ linked_floors[selectedFloors[{linked_lift}]] = call_buttons_
 THEN
  setValues(call_buttons_, cur_floor_,direction_up_,door_open_)
 END
```

Finally, in order to switch between lifts at any time during the animation, we introduce the following operation select_lift. It selects a different lift, updates variable *linked_lift* and sequentially applies substitution update_existing. An illustration of this operation is provided in Figure 5.3. By running operation select_lift(LIFT2) one can control LIFT2 without bringing it back to the ground floor, and later continue with LIFT1. Note that operation select_lift could apply a scheduling approach and provide an optimized technique (not to re-initialize the machine) when switching between lifts.

Listing 11: change mapping on-the-fly

```
OPERATIONS:
select_lift =
    ANY lift WHERE
        lift ∈ Lift ∧ lift ≠ linked_lift
    THEN
        linked_lift := lift ; update_existing
    END
```

5.4 Application to Scheduler Example

We also applied our approach to the *Scheduler* example discussed in [84, 81]. The structural part of the existing Scheduler B specification is presented in Figure 5.7. The machine involves a set of processes called PID. It contains three processes (process1, process2,and process3). We have three variables: active, ready, and waiting, which are defined as sets of processes. Further, the predicates defined in invariants make sure that (i) a process cannot be in states ready and waiting at the same moment, (ii) a process cannot be in states active and ready at the same moment, (iii) a process cannot be in states active and waiting at the same moment, (iv) only one or none of the processes can be active at the moment, (v) and state active contains a process if and only if state ready has a process.

The meta-model of Scheduler is shown in Figure 5.8, which contains two classes: SchedulerClass (root) and ProcessClass. The class ProcessClass has three attributes: Number (integer), Status (enumeration : StatusEnum), and PreviousStatus (enumeration : StatusEnum). Enumeration type StatusEnum has four values: Waiting, Ready, Active, and Deleted.

Using Meeduse, we generated the Static B specification of the scheduler. The generated specification includes three Sets and six variables (alongside their invariants and empty

MACHINE
  Scheduler_Existing
SETS
  $PID = \{process1, process2, process3\}$

CONCRETE_VARIABLES *active, ready, waiting*
INVARIANT
$$active \in \mathcal{P}(PID) \land ready \in \mathcal{P}(PID) \land waiting \in \mathcal{P}(PID) \land$$
$$active \subseteq PID \land ready \subseteq PID \land waiting \subseteq PID \land$$
$$(ready \cap waiting) = \emptyset \land$$
$$active \cap (ready \cup waiting) = \emptyset \land$$
$$\mathbf{card}(active) \leq 1 \land$$
$$((active = \emptyset) \Rightarrow (ready = \emptyset))$$
END

Figure 5.7: Structural part of Existing Scheduler B Specification

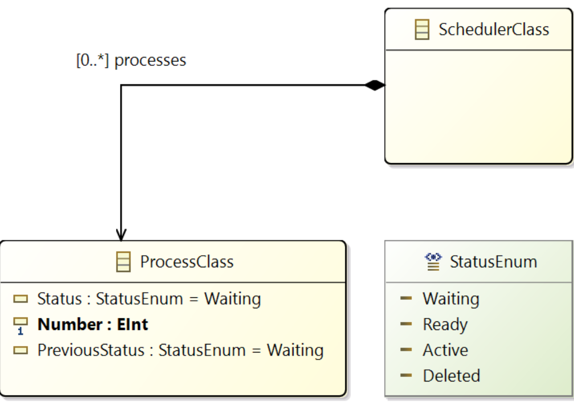


Figure 5.8: Scheduler Meta-Model

initialization) and operations (setters, getters, etc.). Figure 5.9 shows the generated B Specification with Sets (StatusEnum, SCHEDULERCLASS, and PROCESSCLASS), Variables (SchedulerClass, ProcessClass, Processes, Status, Number, and PreviousStatus) and corresponding invariants.

This scheduler example helped us to explore a new mapping rule. The three variables of existing: active, ready, waiting which are defined as sets of processes need to be mapped with similar concepts in DSL. We defined these concepts in the DSL via attribute Status, whose type is enumeration StatusEnum. Mapping enumerations to sets introduces the following invariants in the linkage machine:

Listing 12: EnumType to a Set

INVARIANT:
$$linked_process[\mathbf{dom}(Status \rhd \{Waiting\})] = waiting$$
$$\land\ linked_process[\mathbf{dom}(Status \rhd \{Ready\})] = ready$$
$$\land\ linked_process[\mathbf{dom}(Status \rhd \{Active\})] = active$$

Note: In the above listing, *linked_process* is a constant where process PROCESSCLASS is total bijection to process PID in existing machine ($PROCESSCLASS \rightarrowtail\!\!\!\!\twoheadrightarrow PID$), meaning each process in the DSL is linked to a process in the existing machine.

Figure 5.10 applies state/ transition diagrams to visualize the behavior of processes managed by the Scheduler example. This model introduces three processes (one state/

MACHINE
  scheduler
SETS
  $StatusEnum = \{Waiting, Ready, Active, Deleted\}$;
  $SCHEDULERCLASS$;
  $PROCESSCLASS$
ABSTRACT_VARIABLES
  $SchedulerClass$, $ProcessClass$, $Processes$, $Status$, $Number$, $PreviousStatus$
INVARIANT
  $SchedulerClass \in \text{Pow}\ (SCHEDULERCLASS) \wedge$
  $ProcessClass \in \text{Pow}\ (PROCESSCLASS) \wedge$
  $Processes \in ProcessClass \mapsto SchedulerClass \wedge$
  $Status \in ProcessClass \mapsto StatusEnum \wedge$
  $Number \in ProcessClass \rightarrow \mathcal{Z} \wedge$
  $PreviousStatus \in ProcessClass \mapsto StatusEnum$
END

Figure 5.9: Structural part of Scheduler B Specification

transition diagram per process), and deals with three states: Waiting, Ready and Active. Transitions refer to the various operations of the formal specification, which are: ready, active and swap. The highlighted transition shows the next enabled operations and the highlighted state shows the current state of the process. The concepts PreviousStatus and Delated from the DSL are not presented in the graphical animation.

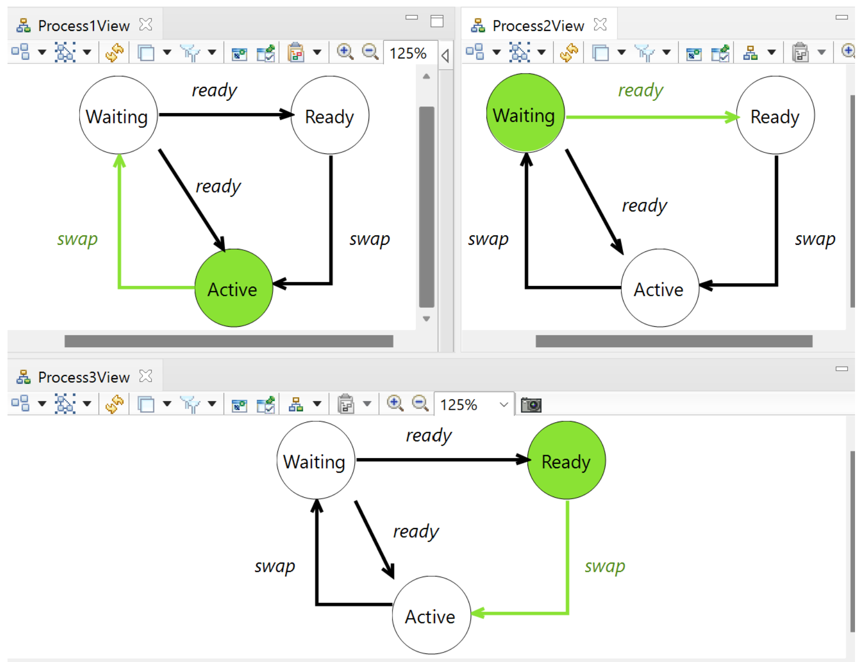


Figure 5.10: Graphical animation of the Scheduler example

Process1 showed in *Process1View* is in state Active whereas the animatable operation from this state is swap. *Process2View* and *Process3View* illustrate Process2 and Process3 whose current states are respectively Waiting and Ready. These state/transition diagrams emphasize on some properties such as deadlock freedom, or the non-blocking property. Indeed, one can see that after being active a process does not stop the system. In fact, the animation of operation swap puts Process1 in state waiting and activates one ready

process. In this case, it activates Process3. As for the Lift, all the B specifications of the Scheduler can be found in the GIT repository[2].

5.5 Discussion

Looking into related works, mixing FMs and DSLs has been addressed in several works [42, 103, 123, 90], but often with the intention to provide formal semantics to a DSL and hence be able to reason about its correctness. In fact, the strength of FMs originates from their precision and the availability of automated reasoning tools. Zalila *et al.*, in [123] suggest the use of an additional DSL to feedback formal verification results. The authors propose a new language (called FeVeReL) that allows the designer to implement the verification result feedback from the formal level to the DSL level. The approach is close to visual animation, but the use of additional languages may weaken the acceptance of the underlying technique because it requires new skills. The work of Tikhonova [111] starts from a DSL meta-model, generates Event-B specifications, but applies classical visual animation using BMotion Studio to the formal specifications in order to understand the behavior of the DSL. The limitation is that the DSL syntax must be reworked in BMotion Studio, which requires additional verifications to check the compatibility between the initial DSL syntax and the graphical visualization.

5.6 Conclusion

This chapter presented our approach for the visual animation of B specifications using an MDE paradigm built on DSLs. Our objective was to ensure validation thanks to domain-centric notations that are expected to be more comprehensible for domain experts than formal specifications. We used Meeduse, a language workbench dedicated to formally instrumenting the static semantics (structure) and dynamic semantics (behavior) of a DSL by means of B models.

We provided a linkage B specification which maps the B models managed by Meeduse and the B specification to visualize. Two strategies are provided and discussed in this chapter: (*i*) RefinementInclusion, and (*ii*) InclusionInclusion. The examples mentioned in this chapter are covered to show the viability of our technique. Writing the linkage B specification is easier when the DSL is based on the same concepts as the existing B specification. DSLs can also be used to visualize specifications that are defined with several refinement levels.

Besides, we showed that our approach also favors the reuse of existing specifications as dynamic semantics of a DSL. It can be done with minor modifications of the existing specification (*e.g.* by adding operations such as setValues and update_existing in the Lift example). The existing lift specification is used for one lift, whereas our approach allows us to use the same existing specification for multiple instances of lifts in our DSL. In the Lift example, we can add more lifts and floors but in the Scheduler example, we are limited to one scheduler, and the user can not add more schedulers. This limitation in the scheduler does not allow us to call it as DSL since a DSL is more flexible to add multiple objects of a class. In the Scheduler example, the number of states is also fixed which is three, but the number of processes can be increased.

Lift and Scheduler are used as proof of concepts, showing our approach's applicability. In the next chapter (Chapter 6), We show how our approach can be applied to a realistic railway system based on the formal B specification of [87], the existing B specification is

[2]https://github.com/meeduse/Samples/tree/main/Scheduler

composed of 4 components (1 machine and 3 refinements). We apply our approach in a formal model-driven way where railway DSL is built iteratively with each refinement of B specification. This allows us to evaluate the feasibility and scalability of our technique and explore various other mapping rules.

5.7 Résumé en français

La validation des systèmes critiques est une tâche essentielle car une mauvaise compréhension des exigences des utilisateurs peut conduire à des implémentations erronées. Cette tâche est connue pour être complexe, notamment pour les spécifications formelles, du fait de leurs notations mathématiques. En fait, les parties prenantes qui ne sont pas formées aux méthodes formelles (FM), comme les experts du domaine, ont du mal à comprendre ces notations. Pour résoudre les problèmes de validation, plusieurs outils formels [78, 85, 117] fournissent des techniques d'animation graphique. L'intention est de proposer une représentation visuelle des données et des comportements d'une spécification formelle, la rendant plus lisible pour les experts du domaine. Dans ce chapitre, nous proposons une approche où les spécifications B écrites indépendamment de l'équipe Meeduse sont animées visuellement à l'aide de DSL. Nous illustrons l'application de notre approche à l'exemple de l'ascenseur (Lift) comme preuve de concept. Nous sélectionnons ce modèle car il est bien connu en matière de vérification et de validation formelles [92].

Nous avons fourni une spécification B de liaison qui connecte les modèles B gérés par Meeduse et la spécification B à visualiser. Deux stratégies sont fournies et discutées dans ce chapitre : (*i*) Refinement/Inclusion et (*ii*) Inclusion/Inclusion. Les exemples mentionnés dans ce chapitre montrent la viabilité de notre technique. L'écriture de la spécification B de liaison est plus facile lorsque le DSL est basé sur les mêmes concepts que la spécification B existante. Les DSLs peuvent également être utilisés pour visualiser des spécifications définies avec plusieurs niveaux de raffinement.

Par ailleurs, nous avons montré que notre approche favorise également la réutilisation de spécifications existantes comme sémantique dynamique d'un DSL. Cela peut être réalisé avec des modifications mineures de la spécification existante (*par exemple* en ajoutant des opérations telles que setValues et update_existing dans l'exemple Lift). La spécification d'ascenseur existante est utilisée pour un seul ascenseur, alors que notre approche nous permet d'utiliser la même spécification existante pour plusieurs instances d'ascenseurs dans notre DSL. Dans l'exemple Ascenseur, nous pouvons ajouter plus d'ascenseurs et d'étages, mais dans l'exemple du Scheduler, nous sommes limités à un seul Scheduler et l'utilisateur ne peut pas en ajouter d'autres. Dans cet exemple, le nombre d'états est fixe.

Chapter 6

Validation of proved ERTMS/ETCS B specification

In the previous chapter, we presented our approach to how B specifications are visually animated using DSLs. We illustrated our approach with an example of Lift as proof of concepts. In this chapter, we apply our approach to an existing proved ERTMS/ETCS B specification. Actually, many works provide solutions to verification issues of ERTMS/ETCS using formal methods, but the validation of the resulting formal models has not been investigated. So apart from visual animation, this chapter also deals with the challenge of validation of proved ERTMS/ETCS B specifications. We present an iterative formal model-driven approach that helps to validate step-by-step a real formal specification of ERTMS/ETCS hybrid level 3. The approach introduces Domain-Specific Languages (DSLs) to help system experts understand existing specifications that are already proved.

6.1 Introduction

The ERTMS/ETCS has gained significant attention in industry and academia, and because of the underlying safety requirements, the formal methods community has investigated numerous techniques for its modeling and verification. For instance, the call for solutions of the ABZ'2018 conference [49] has resulted in several realistic applications of formal methods to ERTMS/ETCS. Most of these applications deal with verification concerns without providing insights showing whether their formal models are valid and conform to the requirements. Some approaches propose translations from graphical models (*e.g.* UML, KAOS) into formal specifications, but as the transformation is not proved correct, the resulting formal models still need to be validated. In chapter 5, we presented an approach for visual animation of B specifications using DSLs. In this chapter, we extend our approach with an iterative model-driven technique. Considering an existing B specification of an ERTMS hybrid level 3 system provided by a formal methods expert [87]. We propose to build incrementally a meta-model of the system which defines the abstract syntax of a DSL. The system is analyzed at each level of abstraction, and the corresponding requirements are simulated. Links are built from the current incremental stage of the model and the corresponding provided B component. This simulation allows railway experts to validate the formal models and compare their understandings.

6.2 Towards an Iterative Formal Model-Driven Approach

Mammar *et* al. [87] provides an EVENT- B model of the hybrid ERTMS/ETCS level 3 standard. In this model, a railway track (a line to run a train) is divided into sections known as Trackside Train Detection (abbreviated as TTD). A TTD is further divided into subsections called Virtual Sub-Section (VSS). An ERTMS train can be fitted with TIMS, which is Train Integrity Monitoring System that reports its position and integrity to the train supervisor, which is the system controller. ERTMS trains without TIMS can only report their front position, while non-ERTMS trains do not report their position at all to the supervisor. Train's VSS occupation is also determined by the TIMS. In the ERTMS/ETCS, Movement Authority (MA) is the permission assigned to the train to move on specific sections or subsections. In this model, the supervisor assigns the MA (containing VSSs) and sends it to the ERTMS train. A train cannot go beyond the VSS specified in the MA in order to avoid accidents (collisions).

This model allows the trains to be connected or disconnected. A connected train regularly reports its integrity and position to the supervisor. The concept of timer is introduced which disconnects a connected train after a specified time. Note that, in this use case, all trains move in the same direction on the same track, and MA is chosen non-deterministically in order to avoid collisions. We re-use this model in our approach as a classical-B artifact which consists of four components: an abstract machine (M0) and three refinements (M1, M2, and M3).

Our approach discussed in chapter 5 introduces a linkage machine that allows the usage of the B machine as dynamic semantics of the DSL. This approach is extended here and applied to a realistic case study. We incrementally build our DSL layer and use refinements and inclusions to make the connection between every increment of the DSL and the considered B model. The proposed approach is applied to each abstract/refinement component as depicted in Figure 6.1.

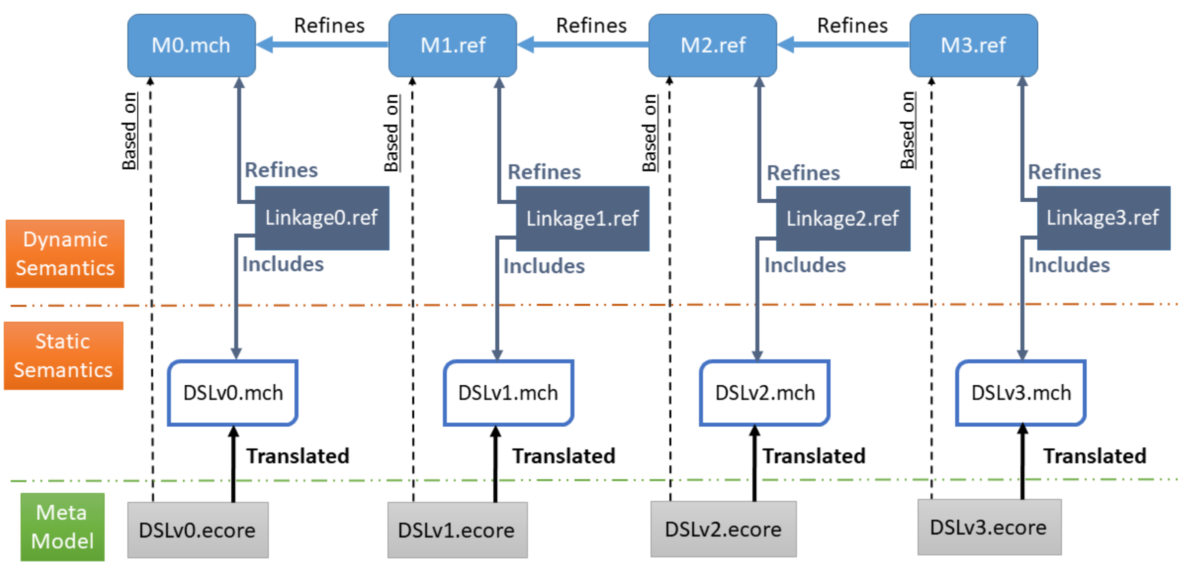


Figure 6.1: The Iterative Architecture for the Case Study

First, we develop a DSL meta-model (DSLv0.ecore) based on the initial abstract component (M0.mch) of the model. Then we provide a linkage machine (Linkage0.ref) that refines the M0.mch and includes the translated static semantics of DSLv0.mch (translation from ecore to B is done using Meeduse). In the next iteration, we update the DSL based

on refinement M1.ref (which is a refinement of the M0.mch machine), and the resulting DSL becomes DSLv1.ecore. In this iteration, another linkage machine (Linkage1.ref) is introduced, which refines the existing refinement M1.ref and includes the translated static semantics DSLv1.mch. The same step is repeated until the final refinement M3. At each iteration, we are able to visually animate the existing component using the corresponding version of the DSL, thanks to the execution of the linkage machine in Meeduse. This graphical animation allows the domain expert to check that the verified built specification (existing model) captures the right requirements. The complete B specification of the existing model, the ones generated from the DSL and linkage machines, can be found in the Meeduse git repository[1]. The mechanisms of linking B data structures, the initialization, and the operations of linkage machines are already discussed in chapter 5.

6.3 An ERTMS/ETCS Hybrid Level 3 DSL

In a Model-Driven Architecture, a DSL is built from a meta-model. We propose to incrementally create this meta-model based on the existing formal B model, such that each concept in the meta-model corresponds to a concept in the B model. From each version of the meta-model Meeduse generates B static semantics, and the dynamic semantics refer to the corresponding component of the B model. Figure 6.7 (presented later) shows the whole meta-model where concepts of each refinement are defined using different colors.

6.3.1 DSL version 0 (DSLv0)

DSLv0 is built based on abstract machine M0.mch of the existing model. Figure 6.2 shows the B data structure of M0.mch with its sets, constants and variables. Initialization and operations are not shown here due to space limitation. Machine M0.mch allows free movement of trains on TTDs and collisions are possible at this level. This level contains operations: trainSupervisor, trainEntering, trainMovingInSameTTD, trainMovingFrontNextTTD, trainMovingRearNextTTD, trainExiting, trainConnect, trainDisconnect, and TimerExpiration.

Figure 6.3 shows the meta-model of DSLv0 where class Railway is the root class. It is composed of class Trackside and class Train. Class Trackside has the attribute TrackStatus which can have the value Free or Occupied from the enumeration Status. Each Trackside has 0 to 2 previous tracksides (association previous) and 0 to 2 next tracksides (association next).

Class Train has two attributes in addition to its identifier: kindOfTrain and Connected. Attribute kindOfTrain refers to three kinds of trains (enumeration KindOfTrains): ERTMS, NoERTMS and TIMSERTMS. An TIMSERTMS train is equipped with Train Integrity Management System contrary to simple ERTMS trains. Note ERTMS trains equipped with TIMS communicate their location with the supervisor. Attribute Connected is a Boolean that shows the connection of a train with the supervisor. Each train has a head and a tail whose positions are defined with references front and rear to class Trackside.

6.3.2 Translation of the meta-model

In order to provide the static semantics as B specifications, Meeduse generates machine DSLv0.mch from DSLv0.ecore, the above ECore meta-model. Figure 6.4 shows the generated Sets, Properties of the Constants, and the related typing invariants. For more details about this translation we refer the reader to [64, 60].

[1] https://github.com/meeduse/Samples/tree/main/ETCSLevel3

MACHINE
$M0$
SETS
$StateTTD = \{freeT, occupiedT\}$;
$TRAINS$;
$TrainKind = \{TimErtms, Ertms, NoErtms\}$
CONSTANTS
$Ttds, minTTD, maxTTD, trainKind, Trains, Cars$
VARIABLES
$stateTTD$, $trainOccupationTTDFront$, $trainOccupationTTDRear$, $isConnected$, $supervisor$
INVARIANT
$stateTTD \in Ttds \rightarrow StateTTD$
$\wedge\ trainOccupationTTDFront \in TRAINS \nrightarrow Ttds$
$\wedge\ trainOccupationTTDRear \in \mathbf{dom}(trainOccupationTTDFront) \rightarrow Ttds$
$\wedge\ \forall\ tr\ .\ (tr \in \mathbf{dom}(trainOccupationTTDFront)$
$\Rightarrow trainOccupationTTDRear(tr) \leq trainOccupationTTDFront(tr))$
$\wedge\ isConnected \in trainKind^{-1}[\{Ertms, TimErtms\}] \rightarrow \mathbf{BOOL}$
$\wedge\ supervisor \in \mathbf{BOOL}$

Figure 6.2: B data structure of existing abstract machine M0

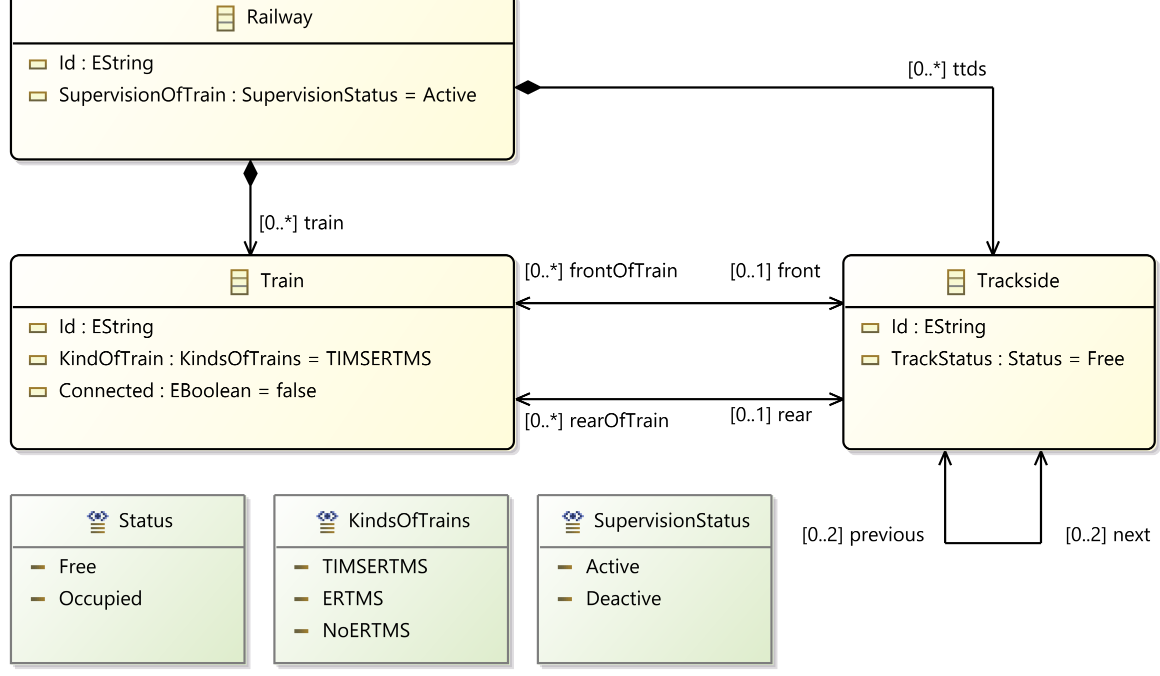


Figure 6.3: DSLv0 Meta-Model

6.3.3 Linkage Machines

Based on the approach of Figure 6.1, linkage B machines are created at each iteration. These machines link the existing B specification (*e.g.* machine M0), which we would like to animate, to the DSL components (*e.g.* machine DSLv0). It refines the existing B model and includes the machine issued from the meta-model of the DSL. The idea is to map concepts from the DSL to concepts from the existing B model. In this sub-section, we give an overview of the mappings used in this first refinement level of our case study.

Machine
  DSLv0
SETS
  $Status = \{Free, Occupied\};$
  $KindsOfTrains = \{TIMSERTMS, ERTMS, NoERTMS\};$
  $SupervisionStatus = \{Active, Deactive\};$
  $RAILWAY; TRACKSIDE; TRAIN;$
VARIABLES [...]
CONSTANTS [...]
PROPERTIES
  $Railway \in \text{Pow } (RAILWAY) \wedge$
  $Trackside \in \text{Pow } (TRACKSIDE) \wedge$
  $Train \in \text{Pow } (TRAIN) \wedge$
  $KindOfTrain \in Train \nrightarrow KindsOfTrains \wedge$
  $previous_next \in Trackside \leftrightarrow Trackside \wedge$
INVARIANT
  $TrainFront \in Train \nrightarrow Trackside \wedge$
  $TrainRear \in Train \nrightarrow Trackside \wedge$
  $SupervisionOfTrain \in Railway \nrightarrow SupervisionStatus \wedge$
  $TrackStatus \in Trackside \nrightarrow Status \wedge$
  $Connected \in Train \nrightarrow \textbf{BOOL} \wedge$
  $\forall\ thePrevious.(\ thePrevious \in \textbf{ran}(previous_next)$
    $\Rightarrow \textbf{card}(previous_next^{-1}[\{thePrevious\}]) \leq 2) \wedge$
  $\forall\ theNext.(\ theNext \in \textbf{dom}(previous_next) \Rightarrow$
    $\Rightarrow \textbf{card}(previous_next[\{theNext\}]) \leq 2)$

Figure 6.4: Structural part of machine DSLv0.mch (without constants and variables)

6.3.3.1 Rule 1: EClass to BMachine.

In the DSL, class Railway is a root class that contains all other classes. We consider that this class and machine M0 represent the same concept, which is the railway system. Thus, we introduce a constant *Linked_Railway* in the linkage machine, which will be helpful later while mapping the underlying concepts.

CONSTANTS: *Linked_Railway*
PROPERTIES: $Linked_Railway \in Railway$

6.3.3.2 Rule 2: Enumeration to BSet.

In machine M0, StateTTD is a set containing values freeT and occupiedT. A similar concept in the DSL is the enumeration Status with the values: Free and Occupied. So, the mapping between an enumeration and a set is done as follows:

CONSTANTS: *Linked_Status*
PROPERTIES: $Linked_Status = \{\ freeT \mapsto Free,\ occupiedT \mapsto Occupied\}$

6.3.3.3 Rule 3: EClass to BSet.

In the existing model, constant Ttds is set from constant minTTD to constant maxTTD. The Ttds concept is similar to class Trackside from the DSL. Trains is a finite set of TRAINS, and the similar concept from the DSL is class Train. Mapping these sets and classes is done as follows:

CONSTANTS: $Linked_Trackside$, $Linked_Trains$
PROPERTIES:
$Linked_Trackside \in Trackside \rightarrowtail\!\!\!\!\twoheadrightarrow Ttds \wedge$
$Linked_Trains \in Train \rightarrowtail\!\!\!\!\twoheadrightarrow Trains$

6.3.3.4 Rule 4: boolean EAttribute to boolean BVariable .

Attribute isConnected of class Train is a boolean attribute. A similar concept in the existing model is variable Connected. To map these two concepts, we introduce the following invariant:

INVARIANT: $Connected = (Linked_Trains \ ; \ isConnected)$

6.3.3.5 Rule 5: EAttribute (EnumType) to BVariable (SetValued).

stateTTD is a variable in the existing model which is typed as a set-value from set StateTTD. In the DSL, TrackStatus is an enumeration attribute in the class Trackside. We used composition relation (;) in this mapping as:

INVARIANT: $TrackStatus = (Linked_Trackside \ ; \ stateTTD \ ; \ Linked_Status)$

6.3.3.6 Rule 6: Attribute (EnumType) to a Boolean variable.

The boolean variable supervisor defines the status of the controller (false if not active and true if it is active). In the DSL, the same concept is defined using an attribute SupervisionOfTrain (enumeration SupervisionStatus) in class Railway. Mapping between enumeration values and the boolean values introduces the following invariant:

INVARIANT:
$(supervisor = \mathbf{TRUE} \Rightarrow SupervisionOfTrain(Linked_Railway) = Active)$
$\wedge \ (supervisor = \mathbf{FALSE} \Rightarrow SupervisionOfTrain(Linked_Railway) = Deactive)$

6.3.3.7 Rule 7: Single valuated EReference to BVariable.

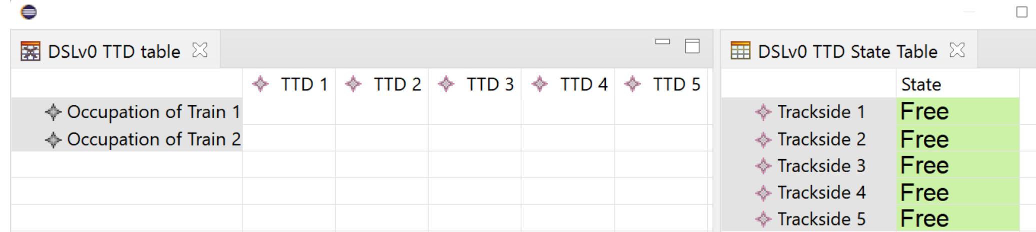
Variable trainOccupationTTDFront of the existing model defines the occupation of the train's front on a Ttd. In the DSL, this concept is defined with reference TrainFront from class Train to class Trackside. We introduce the following invariant to map both concepts:

INVARIANT:
$\mathbf{dom}(TrainFront) = Linked_Trains^{-1} \ [\mathbf{dom}(trainOccupationTTDFront)]$

6.3.4 Modeling and visual animation

Figure 6.5 is a graphical model conforming to DSLv0. It features two TIMS trains (Train 1, Train 2) and five tracks (Trackside 1..5). The state of each track is represented in the right-hand side of the figure. In this model all tracks are set to Free; they are not occupied by any of the two trains. Actually, the model of Figure 6.5 is drawn by the domain expert to describe an initial state.

Once the linkage machine between DSLv0 and M0 is created, the model in Figure 6.5 can be animated in Meeduse. The tool assigns values to all the B data (variables and



DSLv0 TTD table

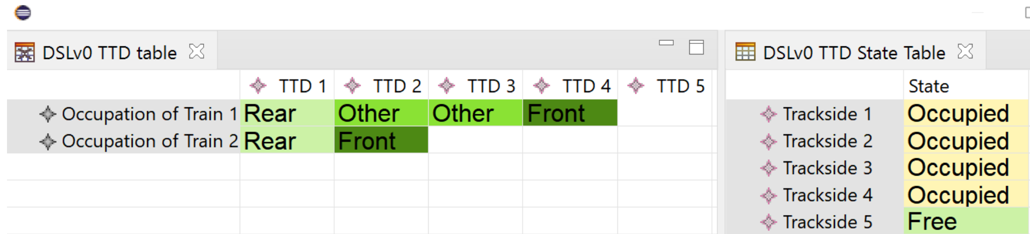
	TTD 1	TTD 2	TTD 3	TTD 4	TTD 5
Occupation of Train 1					
Occupation of Train 2					

DSLv0 TTD State Table

	State
Trackside 1	Free
Trackside 2	Free
Trackside 3	Free
Trackside 4	Free
Trackside 5	Free

Figure 6.5: A model conforming to DSLv0

constants), initializes the machine, and applies ProB for animation. In a classical animation of B specifications, the B method expert has to complete by hand the specifications with valuations. Here, it is the domain expert who created two instances of Train and five instances of Trackside; and then the tool automatically produces the valuated machine together with its initialisation. Figure 6.6 shows the movement of trains in the graphical representation by triggering the operations of M0.mch mentioned in Section 6.3.1.



DSLv0 TTD table

	TTD 1	TTD 2	TTD 3	TTD 4	TTD 5
Occupation of Train 1	Rear	Other	Other	Front	
Occupation of Train 2	Rear	Front			

DSLv0 TTD State Table

	State
Trackside 1	Occupied
Trackside 2	Occupied
Trackside 3	Occupied
Trackside 4	Occupied
Trackside 5	Free

Figure 6.6: Animating DSLv0 using M0

The Front of Train 1 occupies TTD 4 while its Rear occupies TTD 1. The Other shows the occupancy of the train on TTDs that are between Front and Rear. For Train 2, the Front is positioned at TTD 2 and Rear at TTD 1. Apart from Trackside 5, all other tracksides are occupied. At this stage, someone can observe the collision between Train 1 and Train 2 as both occupy the TTD 1 and TTD 2. Actually, the safety property regarding the non-collision is only satisfied in the M3 refinement level of the existing model. We chose not to represent other concepts defined in the meta-model such as supervision status, train connect/disconnect, because they are not directly linked to possible collisions.

6.4 Findings and Analysis

The complete meta-model of our DSL is shown in Figure 6.7. Concepts of every refinement level are presented using different colors. Yellow classes and black associations show DSLv0. Associations in brown color are introduced during DSLv1. Class **VirtualBlock** and associations represented in light blue are from DSLv2. Finally, attributes and associations in purple represent DSLv3.

6.4.1 Next Iterations

6.4.1.1 DSLv1.

This version is an update of DSLv0 by introducing two new concepts introduced in refinement M1 of the existing specification. Since M1 refines M0, all concepts from M0 are already included in meta-model of DSLv1. Associations frontTrackLocation and rearTrackLocation, illustrated with brown color in Figure 6.7 are added in DSLv1. The frontTrackLocation shows the front location of a train that is communicated to the controller (su-

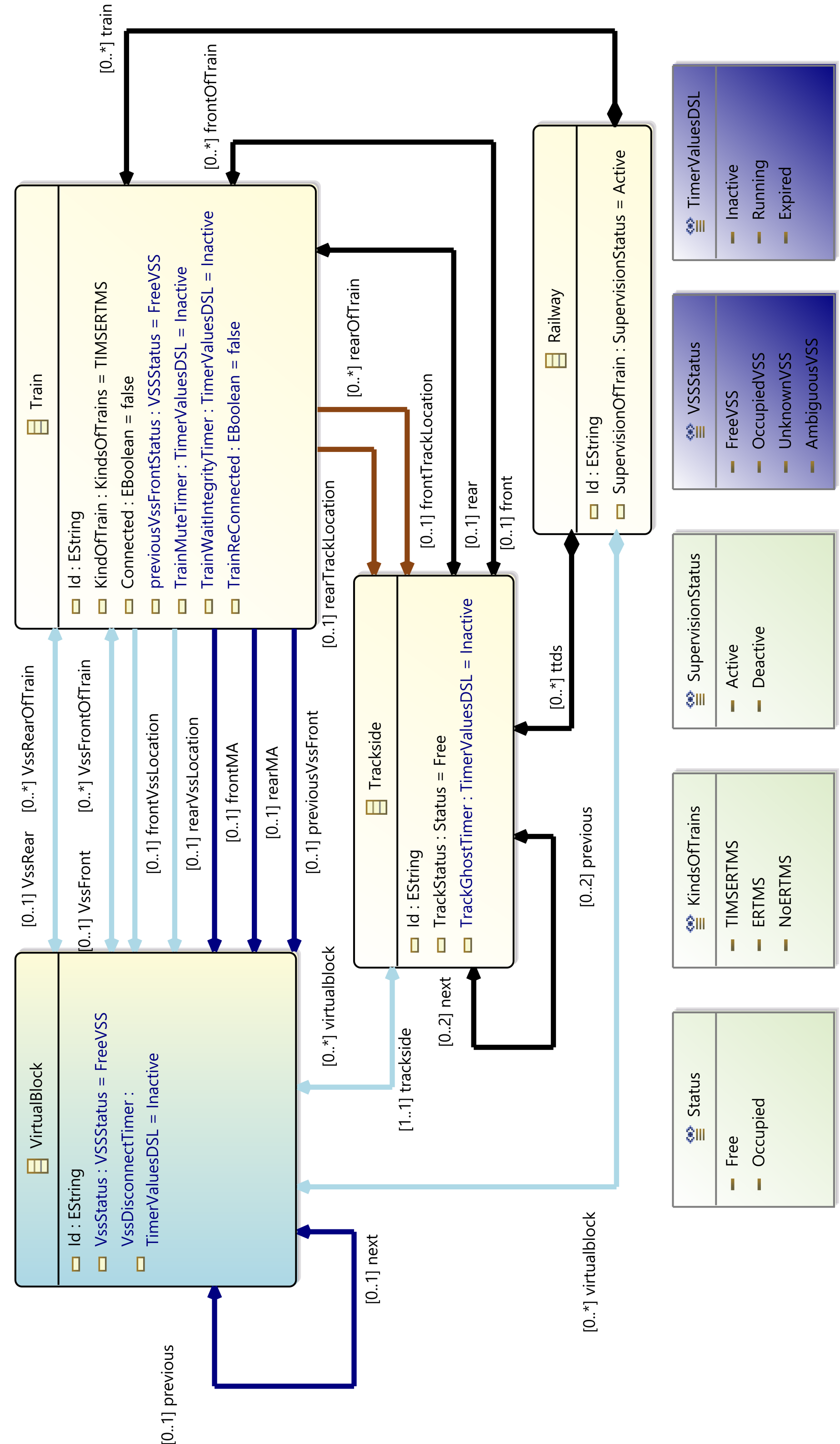


Figure 6.7: Whole DSL Meta-Model

pervisor) and rearTrackLocation is the communicated location of the train's rear. The inclusion of these concepts updates the graphical representation with Train's location information, as known by the supervisor.

6.4.1.2 DSLv2.

Concepts of DSLv2 are shown in light blue in Figure 6.7. Class VirtualBlock has been introduced. It is linked to class Railway using the composition relation. At this level, no attributes are included in class VirtualBlock except the Id. A link between class Trackside and VirtualBlock is created where a track side can have many virtual blocks (association virtualblock), and in the opposite each virtual block is associated with at-least one track (association trackside). The other introduced associations in this level are those from class Train to class VirtualBlock, which are VssFront (front of train on virtual block), VssRear (rear of train on virtual block), frontVssLocation (front of train on virtual block communicated to the controller), and rearVssLocation (rear of train on virtual block communicated to the controller). These additional concepts lead to the creation of a new representation containing virtual blocks (VSSs) instead of tracks (TTDs).

6.4.1.3 DSLv3.

Concepts of DSLv3 are shown in violet in Figure 6.7. Enumeration VSSStatus gives the state of a virtual block. It can be free (FreeVSS), occupied (OccupiedVSS), unknown (UnknownVSS), or ambiguous (AmbiguousVSS). Enumeration TimerValuesDSL includes values: Inactive, Running, and Expired, which are the states of a timer. The associations: previous and next from VirtualBlock to VirtualBlock represents the connexions between virtual blocks. The concept of movement authority (MA) is defined by associations: frontMA (MA for train's head) and rearMA (MA for train's rear) from class Train to class VirtualBlock. Association previousVssFront is used to store the value of a previous VSS for a train's front. We introduced the four timer concepts in the DSL as defined in the existing ERTMS/ETCS HL3 specifications. The timers related to the trains are defined in class Train as TrainMuteTimer and TrainWaitIntegrityTimer. The timer related to VSS is VssDisconnectTimer, which is introduced inside class VirtualBlock, while timer TrackGhostTimer which is related to TTD, is included in class Trackside. The VssStatus attribute in class VirtualBlock shows the status of each virtual block using enumeration VSSStatus. Attribute previousVssFrontStatus stores the status of the previous VSS for the train's front. Finally, attribute TrainReConnected is a Boolean and gets value TRUE when a train is connected back after disconnecting.

Figure 6.8 is a graphical representation conforming to DSLv3. The figure represents eleven VSSs and two TIMS trains, in addition to train's location, occupation, and MA. The right-hand side represents the states of the VSSs. Before assigning MA, train supervisor calculates the VSSs and sets the state of VSSs to free if there is no train. Once the calculation of VSSs is done, a train can be connected and can be assigned an MA. In Figure 6.8 VSSs are free and movements authorities are assigned to Train1 from VSS 1 till VSS 5 . Rear of MA is the "start of authority" and Front of MA is "End of Authority" (EoA).

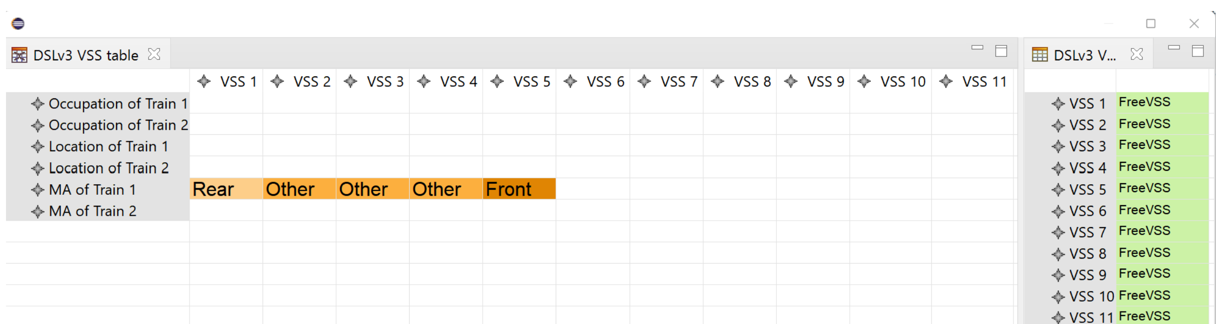


Figure 6.8: Assigning MA to Train 1 using DSLv3

6.4.2 Unexpected behaviors

Figure 6.9 shows that Train1 has entered and reached EoA. In a normal case, it should be possible to assign a new MAs to train 1 to move on the next VSSs, but the animation tells us that this is not possible. The only possible allowed operations are disconnection and connection of Train1. This problem is a deadlock and was identified during the animation of normal scenarios from ERTMS/ETCS. In Figure 6.9, a user can observe that some VSSs remain concerned by MAs even after a train has consumed them. It can be seen that VSS 1, 2, and 3 have been released, but still they are concerned by the MAs of Train1. This behavior can be considered as a problem from the domain expert's point of view. Actually, it reveals some limits (misunderstandings of the requirements) of the existing B specifications.

Another unexpected behavior that we observed is that a VSS never gets the state OccupiedVSS. According to ERTMS/ETCS, when the train's front or rear is over a VSS then the VSS is considered to be occupied. The right side of Figure 6.9 shows that VSS 4 and VSS 5 are set to AmbiguousVSS, which should be instead OccupiedVSS as both VSS host the rear and the front of Train1 respectively. In order to test whether this problem is coming from the linkage machine or the existing machine, we analyzed the states of VSSs in the existing machine. We came to the conclusion that the problem is located in the existing machine, and this information was communicated to the authors of the existing machine. Note that the authors of the existing model already mentioned in their paper that proof obligations related to VSS state machine were found ambiguous (non-deterministic) and were not discharged.

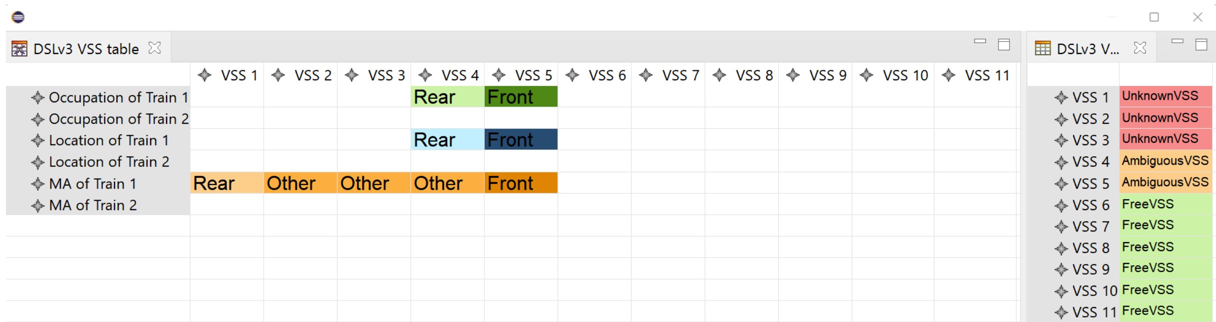


Figure 6.9: Train Movement consuming MAs

6.4.3 Lessons learned

6.4.3.1 Lessons learned from a formal methods expert's point of view.

Amel Mammar is a professor at Télécom SudParis School in Évry, France. She has published several research papers on topics such as intrusion detection systems, software vulnerabilities, and formal methods. Since, we used the existing B specification from her paper [87], we got here got here point of view regarding this case study.

Amel Mammar: "The existing animation tools (Brama, AnimeB, etc.) do not offer a useful graphical view for the model's animation. Indeed, ProB and AnimB, for instance, only display the values of the variables at each animation step leaving the task of analyzing and explaining them to the user. This is definitely not exploitable for complex systems with several variables like ERTMS 3 system. So, the approach introduced in this work permits overcoming the drawbacks of the existing tools by providing a useful graphical view of the animation, permitting users a better understanding of their systems and helping them detect errors and bugs. Among others, this approach permits us to

detect, for instance, that a VSS never becomes occupied, while this went unnoticed under Rodin and ProB."

6.4.3.2 Lessons learned from a railway expert's point of view.

Simon Collart-Dutilleul is a Senior researcher and head of the ERTMS task force of IFSTTAR (The French institute of science and technology for transport, spatial planning, development and networks). He is one of the supervisors supervising this thesis. He gave his point of view on this case study.

Simon Collart-Dutilleul: "Among the three unexpected behaviors, one is to be pointed out at first: VSS never gets the state OccupiedVSS. This is a problem because when a train is on a VSS, this VSS must be occupied. Particularly when the train is connected and sends its position to the control center supervising train movement. In a first analysis, it is surprising that such an evidence is not fulfilled. Analyzing deeper, this is not so surprising. A railway norm is written by railway experts for railway experts. It means that some evidences are not recalled to mind: this is an implicit requirement that everybody is supposed to know. The paper of Mammar [87], clearly says that in their opinion, the specification is ambiguous.

From a methodological point of view, DSL is run using various scenarios generating behaviors that often surprise a railway expert. Two trains in the same location are correct in the first level because the non-collision mechanisms will be implemented in a later step. The simulation analysis by an expert helps to define the semantic limits of a given DSL, but the more interesting part comes when a misunderstanding of the specification is identified. The OccupiedVSS value that is never reached is a good demonstration. Even the authors of the code never identified the problem before. It shows that the basic specification errors happen and are really difficult to detect without a graphical animation."

6.5 Conclusion

In this chapter, we showed how a DSL can be used to validate an existing formal B specification through an industrial case study (ERTMS 3/ETCS). We define links (mappings) between the concepts defined in our DSLs and those used in the B specification (based on our approach discussed in chapter 5 previously). The existing proved B specification used in this work is provided in Event-B originally, since our approach uses classical-B, the B specification has been first rewritten with respect to the classical-B language syntax, and then all the operations have been re-proved under AtelierB by original author of the existing B specification. During this case study, some specific scenarios were used to illustrate how a graphical animations can be used to illustrate domain concepts and how experts (domain and formal methods) can collaborate to ensure that safety requirements are met. The work in this chapter results in, firstly, the creation of DSL from the existing B model and, secondly, proving the hypotheses *"Though FMs prove the consistency of a railway system, it does not guarantee it is correctly built"*.

In the previous chapter (5) and current chapter, the linkage B specifications are produced manually but systematically. In the next chapter (7), we present the tool to produce linkage B specifications in (semi)-automatic way.

6.6 Résumé en français

Dans le chapitre précédent, nous avons présenté comment les spécifications B sont animées visuellement à l'aide de DSLs. Nous avons illustré notre approche avec un exemple (le Lift) en tant que preuve de concept. Dans ce chapitre, nous appliquons notre approche à une spécification B d'ERTMS/ETCS existante et déjà prouvée. Actuellement, de nombreux travaux proposent des solutions aux problèmes de vérification d'ERTMS/ETCS en utilisant des méthodes formelles, mais la validation des modèles formels résultants n'a pas été étudiée. Ainsi, outre l'animation visuelle, ce chapitre traite également du défi de la validation des spécifications d'ERTMS/ETCS. Nous présentons une approche formelle itérative qui permet de valider étape par étape une spécification formelle réaliste de ERTMS/ETCS (niveau 3 hybride).

Nous proposons de construire progressivement un méta-modèle du DSL qui définit sa syntaxe abstraite. Le système est analysé à chaque niveau d'abstraction et les exigences correspondantes sont simulées par animation. Les liens sont construits à partir de l'étape actuelle du modèle et du composant B fourni. Cette simulation permet aux experts ferroviaires de valider les modèles formels et de comparer leurs compréhensions. Les travaux de ce chapitre aboutissent, d'une part, à la création d'un DSL à partir du modèle B existant et, d'autre part, à la démonstration de la propriété suivante *"Bien que les FM prouvent la cohérence d'un système ferroviaire, ils ne garantissent pas que ce dernier est correctement construit"*.

Chapter 7

Automatic Linkage Generation

Previously in chapters 5 and 6, we showed the applicability and scalability of our approach. On the one hand, it is used for visual animation of B specifications. On the other hand, we successfully illustrated how the domain expert identifies the unexpected behavior of a proved system. During the application of our approach, the linkage specifications are created manually but systematically. In this chapter, we introduce a DSL-based tool to automate the extraction of the linkage specification.

7.1 Introduction

To generate the linkage machines, we introduce an approach built on the definition of generation patterns that the user defines, and that can be applied (and reused) for various specifications and models. First, we propose a textual DSL, called Pattern-Definition, that allows one to define patterns of mappings. Secondly, to apply the patterns defined with Pattern-Definition, another DSL is proposed. We call the latter Pattern-Application. Figure 7.1 shows the underlying approach. Pattern-Definition and Pattern-Application are textual DSLs conforming to their corresponding XText grammars. Thanks to these DSLs, the user first proposes a catalog of generic reusable patterns and then he/she can select the ones to apply. Having the definition of patterns and their application, the component Linkage Generator automatically produces the linkage machine. This component is built using Acceleo [1], an Eclipse-based model-to-text transformation tool.

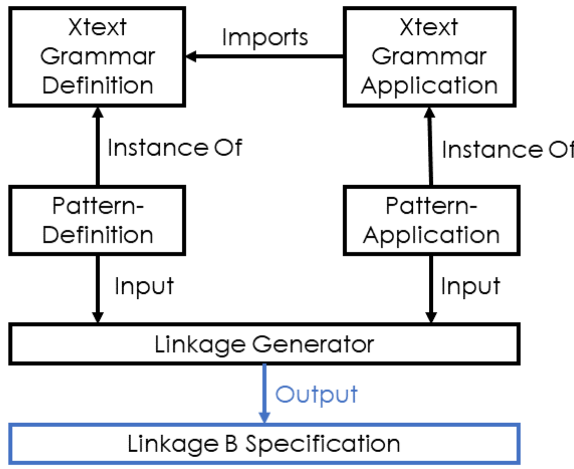


Figure 7.1: Linkage Generation Methodology

7.1.1 Pattern-Definition

Figure 7.2 gives the Xtext grammar of Pattern-Definition DSL, and Figure 7.4 shows an instance of this grammar. In this DSL, a template (defined with a rule Template) is a list of patterns (rule Pattern). The idea is that the user can define an exhaustive catalog of templates and patterns since there are numerous possible mappings depending on the application case. A pattern contains parameters, aliases, and mappings. Parameters are formal parameters that must be valuated in the application step, and aliases refer to local expressions that can be used in a mapping. Often parameters refer to a concept of a meta-model (*e.g.* EClass) and the mapping indicates how this concept is mapped to concepts of a B machine. In Figure 7.4 for example, pattern EClassToBMachine has only one parameter that is an EClass, and produces a variable, an invariant, and an initialisation. Rule mapping has two parts: clause and alternative. It defines the result of the pattern and prescribes to which B clause this result must be added. An alternative is a mixture of aliases and free text. For technical reasons, we need to alias the parameters of the pattern if they are used in the mappings.

```
Template:
    'template' name=ID
    (patterns+=Pattern)*;
Pattern:
    'pattern' name=ID '('
        (parameters+=Parameter)+
    ')'
    alias+=Alias*
    '{'
        (mapping+=Mapping)+
    '}'
;
Alias:
    'alias' name=ID ':' aliasAlt+=AliasAlternative+;
AliasAlternative :
    parameter=[Parameter] | text=STRING;
Mapping:
    clause=Clause ':' (alternative+=Alternative)+;
Clause:
    name=ID;

Alternative:
    alias=[Alias]  | text=STRING;
Parameter:
    name=ID;
```

Figure 7.2: Xtext grammar of Pattern-Definition

Figure 7.3 provides the EMF meta-model of Pattern-Definition DSL defined from the given Xtext grammar shown in Figure 7.2. The meta-model is composed of eight classes, each of which is issued from a grammar rule. Template is the root class and the associations represent the relationships between the objects.

Figure 7.4 defines a template (named Lift) with two patterns: EClassToBMachine

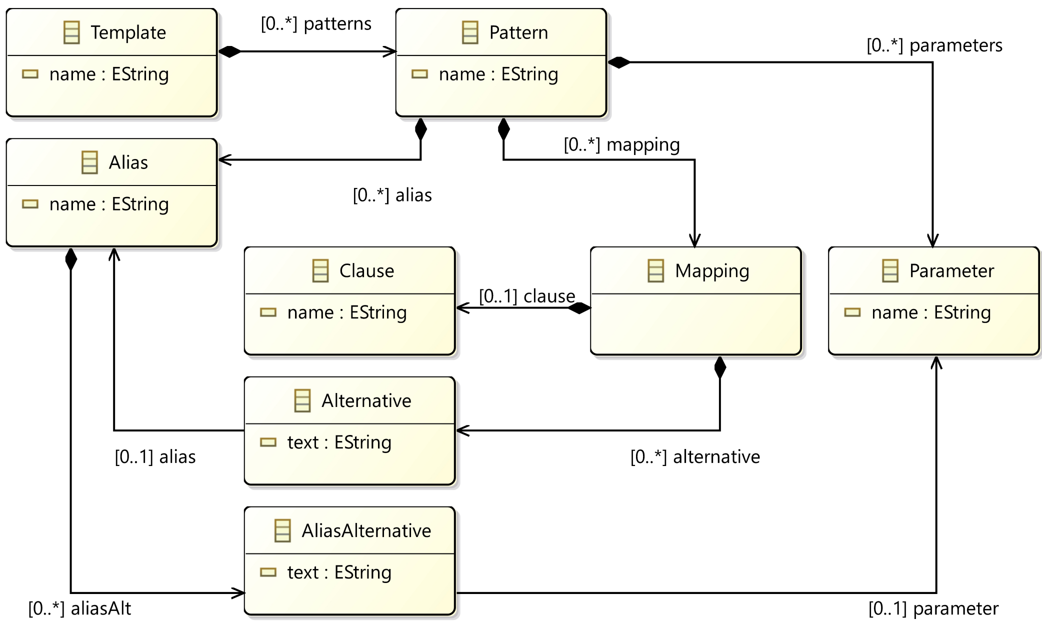


Figure 7.3: Pattern-Definition meta-model

and EClassToBSetExtended. Pattern EClassToBMachine is composed of three mappings with clauses: VARIABLES, INVARIANT and INITIALISATION respectively. In clause VARIABLES, this pattern adds a variable name, represented with alias varName whose value is the concatenation of string "linked_" with the value of parameter anEClass. The same principle is applied to the other clauses, (INVARIANT : varName ":" anEClass) and (INITIALISATION : varName "::" anEClass).

Pattern EClassToBSetExtended is defined with three parameters: anEClass, aBSet, and userDefined. Alias userDefined refers to the parameter userDefined while alias relation is defined as a bijection from parameter anEClass to parameter aBSet. Pattern EClassToBSetExtended is also composed of three mappings, that produce B concepts in clauses VARIABLES, INVARIANT and INITIALISATION.

7.1.2 Pattern-Application

The Xtext grammar of Pattern-Application DSL is given in Figure 7.5 while Figure 7.7 presents an instance of this grammar.

The objective of this DSL is to apply the patterns of a template defined using the Pattern-Definition DSL. Pattern-Application DSL is an application (Rule Application) with a technique (enum Technique), a header (Rule Header), multiple applyPatterns (Rule ApplyPattern), and bclauses (Rule BClauses). A technique can be one of our strategies, either RefinementInclusion or InclusionInclusion. A header contains headerfirst (enum HeaderFirst), which is the type of B specification that can be either a machine or refinement. The header also contains the name of the B specification and a text part for the other part of the header as an OSTRING. We introduced OSTRING as a terminal rule to define the B syntax within B{ syntax }B (line 28 and 29 in Figure 7.5). The applyPattern applies a pattern of a template and takes an exact number of parameters as defined using the Pattern-Definition DSL. The parts of the linkage specification, especially clauses that can not be produced by applying patterns, are defined manually in Bclauses, for instance, definitions, initialization, operations, etc.

The EMF meta-model of Pattern-Application is shown in Figure 7.6. It is defined

```
template Lift
pattern EClassToBMachine(anEClass)
alias anEClass : anEClass
alias varName : "linked_"anEClass
{
    VARIABLES : varName
    INVARIANT : varName ":" anEClass
    INITIALISATION : varName "::" anEClass
}
pattern EClassToBSetExtended(anEClass aBSet userDefined)
alias userDefined : userDefined
alias varName : "linked_"anEClass
alias relation : anEClass ">->>" aBSet
{
    VARIABLES : varName
    INVARIANT : varName ":" relation
    INITIALISATION:
    "ANY mapping WHERE
        mapping :" relation "
    &
    "
        userDefined
    "THEN
    "
        varName ":= mapping
    END"
}
```

Figure 7.4: User defined patterns

from the Xtext grammar shown in Figure 7.5. The meta-model consists of four classes, and the axiom Application is the root class. It is composed of three classes: BClauses, ApplyPattern, and Header. HeaderFirst and Technique are the two enumerations in the meta-model. Note that terminal rules are not shown in meta-model, such as OSTRING in this case.

Figure 7.7 defines an application with the technique RefinementInclusion. The type of B specification is defined as a refinement with the name Lift_Linkage, and the text part shows that it refines the Lift_Existing and includes the Lift_DSL. The application applies two patterns from the Lift template. The first one is the EClassToBMachine which takes an Eclass Lift as a parameter. The second one is pattern EClassToBSetExtended, which takes three parameters: anEClass, aBSet, and userDefined. We used the Floor as EClass, groundf..topf as the B set, and B syntax as user-defined. The application also contains a BClause which is an initialisation where cur_floor, door_open, call_buttons, and direction_up are initialized.

7.2 Experimentation with the Tool

We applied our approach to several case studies, including the realistic ERTMS/ETCS. We generated the linkage B specifications of the examples as mentioned earlier in chapters 5 and 6, first by defining patterns in Pattern-Definition DSL and then applying those patterns using Pattern-Application DSL. Table 7.1 shows the number of defined and applied patterns used during the application of our tool.

```
8  Application :
9      'technique' technique=Technique
10     (header=Header)
11     ((applyPatterns+=ApplyPattern) | (bclauses+=BClauses))*;
12 BClauses:
13     'BClause' name=ID
14         bclause=OSTRING;
15 Header:
16     'header' '{'
17         headerfirst=HeaderFirst  name=ID
18         text=OSTRING '}';
19 enum HeaderFirst:
20     mach='MACHINE' | ref='REFINEMENT';
21 enum Technique:
22     refinc= 'RefinementInclusion'
23     | incinc= 'InclusionInclusion';
24 ApplyPattern:
25     'applyPattern'
26         pattern=[Pattern|QualifiedName]
27             '(' value+=(ID | OSTRING)+ ')';
28 terminal OSTRING :
29             'B{' -> '}B';
30 QualifiedName:
31     ID ('.' ID)*;
```

Figure 7.5: Xtext grammar of Pattern-Application

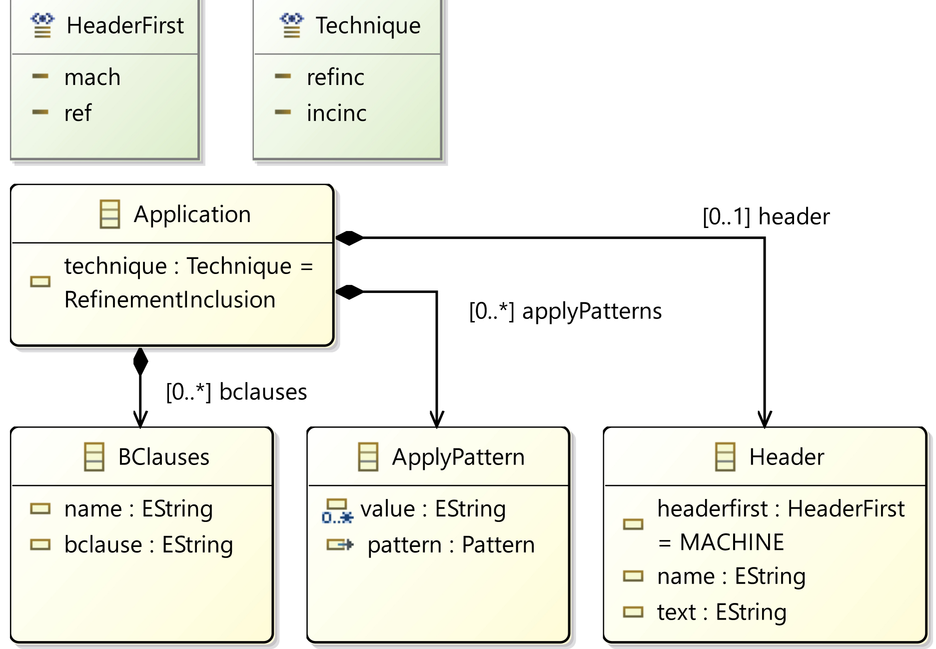


Figure 7.6: Pattern-Application meta-model

7.2.1 Lift

As shown in Table 7.1, we defined 5 patterns during the application of our tool to the example of Lift. To generate the Lift linkage B specification, we applied 6 patterns and introduced 2 BClauses. The first two defined patterns of Lift example are already illustrated in Figures 7.4 and Figure 7.7. Following are the other three defined patterns

```
technique RefinementInclusion
header{
    REFINEMENT Lift_Linkage
    B{
        REFINES Lift_Existing
        INCLUDES Lift_DSL
    }B
}
applyPattern Lift.EClassToBMachine(Lift)

applyPattern Lift.EClassToBSetExtended
    (
    Floor
    B{groundf..topf}B
    B{!(f1,f2).(f1 : Floor & f2 : Floor & (f1|->f2)
        : up_down => mapping(f2) = mapping(f1) + 1
    )}B
)

BClause INITIALISATION
B{   cur_floor := (groundf) ;
    door_open := FALSE ;
    call_buttons := ({}) ;
    direction_up := TRUE
}B
```

Figure 7.7: Pattern-Application textual editor

Table 7.1: Comparison of Defined and Applied Patterns

Example	Defined Patterns	Applied Patterns	B Clauses
Lift	5	6	2
Scheduler	2	4	3
ERTMS M0	8	10	2
ERTMS M1	8	12	2
ERTMS M2	8	14	2
ERTMS M3	8	26	2

and their application to the Lift example.

7.2.1.1 SingleValuedERefToSetElement.

As shown in Figure 7.8, two parameters are defined for this pattern. The first one is EReferenceExp which is an expression for an EReference, and the other one is an element of a set. Two aliases are defined for local expressions. The pattern maps EReference with the set element and produces an invariant.

Figure 7.9 shows the application of the pattern SingleValuedERefToSetElement where it takes two parameters as defined. EReferenceExp is the expression of liftPosition which is a single-valued reference from EClass linked_lift to EClass linked_floors. The cur_floor is the current position of the floor, and it is an element from the set groundf..topf.

```
pattern SingleValuedERefToSetElement
(EReferenceExp aSetElement)
alias EReferenceExp : EReferenceExp
alias aSetElement : aSetElement
{
INVARIANT : EReferenceExp "=" aSetElement
}
```

Figure 7.8: SingleValuedERefToSetElement Pattern

```
applyPattern Lift.SingleValuedERefToSetElement(
B{linked_floors(liftPosition(linked_lift))}B
cur_floor
)
```

Figure 7.9: SingleValuedERefToSetElement Application

7.2.1.2 MultipleValuedERefToSet.

Figure 7.10 illustrates the pattern MultipleValuedERefToSet. Two parameters are defined for this pattern: An EReferenceExp and aSet. The pattern also contains two aliases for local expressions. The pattern produces an invariant using both parameters.

```
pattern MultipleValuedERefToSet
(EReferenceExp aSet)
alias EReferenceExp : EReferenceExp
alias aSet : aSet
{
INVARIANT : EReferenceExp "=" aSet
}
```

Figure 7.10: MultipleValuedERefToSet Pattern

The application of the pattern MultipleValuedERefToSet is shown in Figure 7.11 where the first parameter is the EReference expression for selectedFloors. It is a multi-valued reference from linked_lift to linked_floors. The second parameter is the set call_buttons.

```
applyPattern Lift.MultileValuedERefToSet(
B{linked_floors[selectedFloors[{linked_lift}]]}B
call_buttons
)
```

Figure 7.11: MultipleValuedERefToSet Application

7.2.1.3 EnumTypeToBoolean.

Pattern EnumTypeToBoolean is shown in Figure 7.12 where it is defined to accept four parameters. This pattern maps an enumeration type attribute (with two values) to a Boolean. The first parameter of this pattern is the attribute expression, the second one is the Boolean expression. The other two parameters: aVal1 and aVal2 are the values of enumeration type attribute. The pattern produces two invariants, each with a different

```
pattern EnumTypeToBoolean
(attributeExp bExp aVal1 aVal2)
alias attributeExp : attributeExp
alias bExp : bExp
alias aVal1 : aVal1
alias aVal2 : aVal2
{
INVARIANT :
"("attributeExp"="aVal1"<=>"bExp"=""FALSE"")"
"&"
"("attributeExp"="aVal2"<=>"bExp"=""TRUE"")"
}
```

Figure 7.12: EnumTypeToBoolean Pattern

value of enumeration type attribute. One value is mapped with the FALSE, and the other one is mapped with the TRUE value of Boolean.

Figure 7.13 presents the application of EnumTypeToBoolean pattern with two applyPatterns. This pattern accepts four parameters. In the first applyPattern, the first parameter is the expression for the enumeration type attribute Door of EClass linked_lift and it is mapped with the Boolean door_open which can be expressed in the second parameter as a B expression. The third parameter aVal1 is the value Closed of attribute Door and similarly, the fourth parameter aVal2 is the value Open of attribute Door.

In the second applyPattern, the enumeration type attribute Direction of EClass linked_lift is mapped to the Boolean door_open. Parameters aVal1 and aVal2 are the enumeration values: Down and Up, respectively.

```
applyPattern Lift.EnumTypeToBoolean(
    B{Door(linked_lift)}B
    door_open
    Closed
    Open
)
applyPattern Lift.EnumTypeToBoolean(
    B{Direction(linked_lift)}B
    direction_up
    Down
    Up
)
```

Figure 7.13: EnumTypeToBoolean Application

The result of applying the pattern EnumTypeToBoolean can be seen in Section 5 Listing 5.

We also defined two BClauses in the application for the complete generation of lift linkage B specification. First one is for the initialisation and the other one is for the operations.

7.2.2 Scheduler

To produce the scheduler linkage B specification while using our tool, we defined patterns in a template called scheduler that is shown in Figure 7.14. The template consists of two

patterns: EClassToExtendedConstant and ReferenceToVariable. Pattern EClassToExtendedConstant is defined with three parameters (anEClass, aConstant and userDefined) and maps an EClass to a constant. The pattern contains four aliases, and it will produce a constant and a property. In clause CONSTANT, this pattern adds a constant name, represented with alias consName whose value is the concatenation of string "Linked_" with the value of parameter anEClass. Alias relation is defined as a bijection from parameter anEClass to parameter aConstant. Pattern ReferenceToVariable maps a reference to a variable and it is defined with two parameters: bExp (B expression) and aVar (Variable). It only produces an invariant. Next in this sub-section, we discuss the application of both patterns.

```
template scheduler
pattern EClassToExtendedConstant
(anEClass aConstant userDefined)
alias anEClass:anEClass
alias aConstant: aConstant
alias consName: "Linked_"anEClass
alias userDefined : userDefined
alias relation : anEClass ">->>" aConstant
{
    CONSTANTS:
        consName
    PROPERTIES:
        consName ":" relation
        " & " userDefined
}
pattern ReferenceToVariable(bExp aVar)
alias bExp: bExp
alias aVar: aVar
{
    INVARIANT:
        aVar "=" bExp
}
```

Figure 7.14: Scheduler Pattern Template

7.2.2.1 EClassToExtendedConstant.

The application of pattern EClassToExtendedConstant is demonstrated in Figure 7.15 where it takes the class PROCESSCLASS as the first parameter anEClass. The second parameter is a constant PID. The third parameter is userdefined which is defined as the cardinality of PROCESSCLASS equal to the cardinality of PID (card(PROCESSCLASS)=card(PID)).

```
applyPattern scheduler.EClassToExtendedConstant(
    PROCESSCLASS
    PID
    B{ card(PROCESSCLASS) = card(PID) }B
)
```

Figure 7.15: EClassToExtendedConstant Application

The result of applying the pattern EClassToExtendedConstant can be seen in Listing 13.

Listing 13: EClass to extended Constant

$$
\begin{array}{l}
\text{CONSTANTS} \\
\ \mathit{Linked_PROCESSCLASS} \\
\text{PROPERTIES} \\
\ \mathit{Linked_PROCESSCLASS} \in \mathit{PROCESSCLASS} \rightarrowtail\!\!\!\!\rightarrow \mathit{PID} \\
\ \wedge\ \mathbf{card}(\mathit{PROCESSCLASS}) = \mathbf{card}(\mathit{PID})
\end{array}
$$

7.2.2.2 ReferenceToVariable.

Pattern ReferenceToVariable is applied three times in the application as shown in Figure 7.16. The first parameter contains the reference as a B expression. In these applyPatterns, the variables waiting, ready, and active from existing scheduler B specification are used in the second parameter. They are mapped with their corresponding equivalent concepts (Waiting, Ready, and Active) from the DSL machine.

```
applyPattern scheduler.ReferenceToVariable(
    B{linked_process[dom(Status|>{Waiting}) ]}B
    waiting
)
applyPattern scheduler.ReferenceToVariable(
    B{linked_process[dom(Status|>{Ready}) ]}B
    ready
)
applyPattern scheduler.ReferenceToVariable(
    B{linked_process[dom(Status|>{Active}) ]}B
    active
)
```

Figure 7.16: ReferenceToVariable Application

In this example of the scheduler, we used the Refinement/Inclusion technique. To generate the complete scheduler B specification, we used the BClause tag three times in the application: for definition, initialization, and operations.

7.2.3 ERTMS/ETCS

To use our tool on the realistic case study of ERTMS/ETCS, we defined eight patterns in the Pattern-Definition DSL. The patterns are defined based on the ERTMS/ETCS B specification provided by Amel Mammar [87]. In our previous examples, we already illustrated the application of four out of the eight patterns. The already illustrated patterns are EClassToExtendedConstant, EclassToBMachine, EnumTypeToBoolean and ReferenceToVariable. We now illustrate the following four new patterns:

7.2.3.1 EClassToConstant.

Pattern EClassToConstant is shown in Figure 7.17. It is defined to map an EClass to a constant. The difference between the pattern EClassToConstant and the pattern EClassToExtendedConstant is that it takes two parameters: anEClass and aConstant. Whereas EClassToExtendedConstant takes three parameters, including the userDefined which was used to produce an additional invariant. Apart from the parameter userDefined, pattern EClassToConstant is the same as EClassToExtendedConstant.

```
pattern EClassToConstant
(anEClass aConstant)
alias anEClass:anEClass
alias aConstant: aConstant
alias consName: "Linked_"anEClass
alias relation : anEClass ">->>" aConstant
{
    CONSTANTS:
        consName
    PROPERTIES:
        consName ":" relation
}
```

Figure 7.17: EClassToConstant Pattern

Figure 7.18 gives the application of EClassToConstant pattern. The first parameter is EClass Train from DSL, and the second parameter is the constant Trains from ETCS specification.

```
applyPattern amel.EClassToConstant(
    Train
    Trains)
```

Figure 7.18: EClassToConstant Application

7.2.3.2 EnumToSet.

To map an enumeration to a set, we defined a pattern EnumToSet with two parameters which can be seen in Figure 7.19. It is defined with two parameters: anEnum (Enumerator) and bExp (B expression). It will produce a constant and a property.

```
pattern EnumToSet(anEnum bExp)
alias consName: "Linked_"anEnum
alias bExp: bExp
{
    CONSTANTS:
        consName
    PROPERTIES:
        consName "={" bExp "}"
}
```

Figure 7.19: EnumToSet Pattern

The application of Pattern EnumToSet is shown in Figure 7.20. The first parameter is enumeration Status from the DSL and the second parameter is a B expression. In B expression, the elements of set StateTTD from ETCS machine are linked to the corresponding values of enumeration Status.

```
applyPattern amel.EnumToSet
(
Status
B{freeT|->Free,occupiedT|->Occupied}B
)
```

Figure 7.20: EnumToSet Application

7.2.3.3 BoolAttributeToBoolVariable.

Figure 7.21 gives the pattern to map a Boolean attribute to a Boolean variable. The pattern is defined with three parameters. The first parameter is called consVarName; it can be either a constant or a variable. The second parameter is an attribute from DSL, and the third one is a variable from the existing B machine. The pattern produces the variable of the existing machine in the linkage specification. It also produces an invariant.

```
pattern BoolAttributeToBoolVariable
(consVarName anAttribute aVar)
alias consVarName : consVarName
alias anAttribute: anAttribute
alias aVar: aVar
{
VARIABLES: aVar
INVARIANT: anAttribute"=("consVarName","aVar")"
}
```

Figure 7.21: BoolAttributeToBoolVariable Pattern

Pattern BoolAttributeToBoolVariable is applied to map the attribute Connected (second parameter) from DSL, and the variable isConnected (third parameter) from the ETCS machine, which can be seen in Figure 7.22. The first parameter consVarName is constant Linked_Train.

```
applyPattern amel.BoolAttributeToBoolVariable
(
    Linked_Train
    Connected
    isConnected
)
```

Figure 7.22: BoolAttributeToBoolVariable Application

7.2.3.4 EnumTypeAttributeToSetValued- Variable.

The pattern to map an enumeration type attribute with a variable containing set values is given in Figure 7.23. The pattern is defined with four parameters. The first parameter is the enumeration type attribute, and the second parameter is consVarName1, which is a constant or variable related to the attribute. The third parameter is the variable from the existing machine, and the last parameter is the consVarName2, which is related to the variable. The pattern also produces the variable of the existing machine with an invariant in the linkage specification.

The application of the pattern EnumTypeAttributeToSetValuedVariable can be seen in Figure 7.24. It maps the attribute TrackStatus from DSL to the variable stateTTD of

```
pattern EnumTypeAttributeToSetValuedVariable
(anAttribute consVarName1 aVar consVarName2)
alias consVarName1 : consVarName1
alias consVarName2 : consVarName2
alias anAttribute: anAttribute
alias aVar: aVar
{
VARIABLES: aVar
INVARIANT:
anAttribute"=("consVarName1","
             aVar","consVarName2")"
}
```

Figure 7.23: EnumTypeAttributeToSetValuedVariable Pattern

the existing ETCS machine. The second parameter consVarName1 is Linked_Trackside, and the fourth parameter consVarName2 is Linked_Status.

```
applyPattern
amel.EnumTypeAttributeToSetValuedVariable
(
    TrackStatus
    Linked_Trackside
    stateTTD
    Linked_Status
)
```

Figure 7.24: EnumTypeAttributeToSetValuedVariable Application

As mentioned earlier in chapter 6, the existing ERTMS/ETCS B specification comprises one machine (M0) and three refinements (M1, M2, M3). Using the above patterns from Pattern-Definition DSL, we applied the patterns in 4 separate application files of Pattern-Application DSL, which produced four different linkage machines. The number of applyPatterns in each file varies as each machine in ERTMS/ETCS specification varies based on the number of constants, properties, variables, and invariants. The applyPatterns in M0, M1, M2 and M3 are 10, 12, 14, and 26, respectively. In each application file, two BClauses are defined: one for initialisation and one for operations.

7.3 Discussion

In software engineering, patterns are used as reusable solutions for the development of systems and applications. To the same idea, we used reusable patterns in order to create the linkage B machines which link existing B specifications with B specifications of the DSL meta-model. Our DSL-based tool, first allows to define the patterns then allows to select the ones to apply. Table 7.1 shows the defined and applied patterns in multiple examples using our tool. In the Lift example, we defined 5 patterns and the number of applied patterns is 6, which shows that one of the patterns is used two times. In the scheduler example, we defined two patterns and applied 4 patterns, which again shows the reuse of patterns. The interesting part of pattern reuse is illustrated during the ERTMS/ETCS case study, where we only defined 8 patterns once, and those 8 patterns are reused to generate 4 different linkage machines. In Table 7.1, we can see that the applied patterns are 10, 12, and 14 for examples of ERTMS M0, M1, and M2 respectively.

For ERTMS M3, applied patterns are 26, which shows the high reuse of patterns in the example.

7.4 Conclusion

This chapter discussed the DSL-based tool to generate the linkage B specification. It allows the FM expert to define reusable patterns and apply them to different specifications. The tool comprises two DSLs: Pattern-Definition and Pattern-Application, and the generator component. The tool successfully generated the linkage B specifications for the examples Lift, Scheduler, and ERTMS/ETCS mentioned in earlier chapters.

Linkage B specification is part of the dynamic semantics shown in Figure 1.3 discussed during Chapter 1. We showed that it could be generated in a semi-automatic way, and it also makes it possible to use proved B specifications as operational (dynamic) semantics of a DSL. In the next chapter, we move toward the main objective of our thesis, the application of our xDSL-based framework. In the framework, the static semantics (structure) of the DSL is based on railway standard notations, while the execution (behavior) of the DSL is made possible with ERTMS/ETCS proved B specifications.

7.5 Résumé en français

Dans ce chapitre, nous introduisons un outil basé sur des DSLs pour automatiser l'extraction de la spécification de liaison. Nous introduisons une approche basée sur des modèles de génération que l'utilisateur définit et qui peuvent être appliqués (et réutilisés) pour diverses spécifications et modèles. Premièrement, nous proposons un DSL textuel, appelé Pattern-Definition, qui permet de définir des modèles de correspondances. Deuxièmement, pour appliquer les modèles définis avec Pattern-Definition, un autre DSL est proposé. Nous appelons ce dernier Pattern-Application. Les deux sont des DSLs textuels conformes à leurs grammaires XText. Grâce à ces DSLs, l'utilisateur propose dans un premier temps un catalogue de motifs génériques réutilisables puis il peut sélectionner ceux à appliquer. Ayant la définition des modèles et de leur application, le composant Linkage Generator produit automatiquement la machine de liaison. Ce composant est construit à l'aide d'Acceleo [1], un outil de transformation de modèles en texte basé sur Eclipse.

L'outil a généré avec succès les spécifications B de liaison pour les exemples Lift, Scheduler et ERTMS/ETCS mentionnés dans les chapitres précédents. Dans l'exemple Lift, nous avons défini 5 motifs et le nombre de motifs appliqués est de 6, ce qui montre qu'un des motifs est utilisé deux fois. Dans l'exemple du Scheduler, nous avons défini deux motifs et appliqué 4 motifs, ce qui montre encore une fois la réutilisation des motifs . La partie intéressante de la réutilisation des modèles est illustrée lors de l'étude de cas ERTMS/ETCS, où nous n'avons défini qu'une seule fois 8 motifs , et ces 8 motifs sont réutilisés pour générer 4 machines de liaison différentes. Les nombres de motifs appliqués sont 10, 12 et 14 pour les exemples d'ERTMS M0, M1 et M2 respectivement. Pour l'ERTMS M3, les motifs appliqués sont au nombre de 26, ce qui montre la forte réutilisation des motifs dans l'exemple.

Chapter 8

Application

In the previous chapter, we demonstrated our DSL-based tool to generate linkage machines in a semi-automated way. The linkage machines are part of our DSL-based framework, which helps to validate proved B specifications by visual animation using DSLs, and these machines can also be helpful to use existing B specifications as dynamic semantics of a DSL. This chapter discusses and applies our approach to railway standard notations.

8.1 Introduction

In the railway field, standards are considered to provide cross-border interoperability and harmonize the interfaces. In Chapter 6, we showed how a DSL is used to validate existing proved ERTMS/ETCS B specifications. The DSL is built systematically based on the existing B specifications. In this chapter, we define the DSL based on railway standard notations. In chapter 3, we discussed standards like ERTMS/ETCS, RSM, and EULYNX, and we also mentioned that we use a subset of EULYNX as our railway meta-model. To recall EULYNX, it provides almost all the concepts to be used in railway domain and it is accepted by various European railway stakeholders. In this thesis, we developed interest to use EULYNX as it is aligned with RSM and provides ERTMS/ETCS concepts which makes our xDSL-based framework conformant to standards.

8.2 Methodology

Figure 8.1 presents the methodology of our application for merging EULYNX with ERTMS/ETCS B specifications such that the B Specification is visualized in order to be validated. The EULYNX subset is the meta-model, which is extracted from the EULYNX DP model. This subset is composed of RSM topology and ETCS concepts of EULYNX like VSS, TVP, Route (list of sections), etc as mentioned and discussed earlier in chapter 3. The railway expert can design railway models as instances of these concepts. Note that EULYNX does not contain some concepts of ETCS like train; for this purpose, we introduced another meta-model called ETCS Data, where we include concepts like train. ETCS Data meta-model refers to the EULYNX subset, and translating ETCS Data using Meeduse will translate both ETCS Data and EULYNX subset into DSL B specification. Since the existing ERTMS/ETCS B specification is composed of four components, we also generate four linkage machines. Each linkage machine maps the concepts of its refined existing B specification with the corresponding concept in DSL B specification. Executing a linkage machine in Meeduse will allow the user to compute the list of possible operations (from existing B specifications) that can be animated from that given state of DSL.

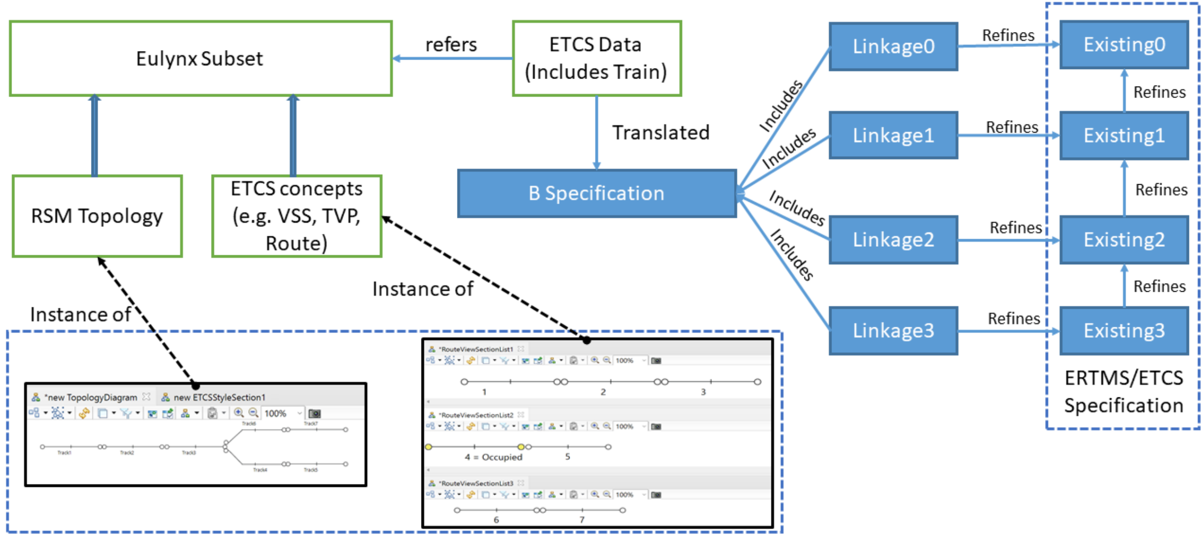


Figure 8.1: Merging EULYNX with existing ERTMS/ETCS B specifications

8.3 Alignment of Meta-models

The ETCS Data meta-model is shown in Figure 8.2, which contains two classes: the root class **ETCSModel** and class **Train**. The alignment with EULYNX is done using references. First, an association called **EulynxModel** is established from the root class to the **DocumentRoot** class of the EULYNX. Then, in order to establish the relationship of the train with EULYNX, the references from class Train to a EULYNX class were required. Keeping in mind the relations used in each level of existing ERTMS/ETCS specification, we established multiple associations (from class **Train**) to class **VirtualSubSection** of EULYNX. The associations are **occupyFront**, **occupyRear**, **locationFront**, **locationRear**, **frontMA**, and **rearMA**, which demonstrate concepts like train occupation (actual position of the train), train location (position of the train to the system), and train's movement authority.

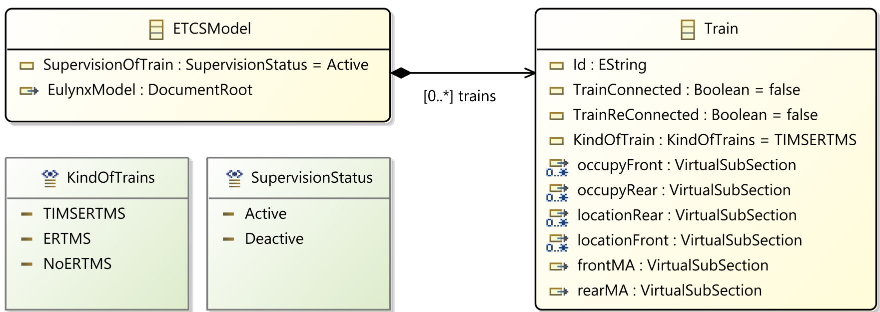


Figure 8.2: ETCS Data Meta-model

We also included some attributes and enumerations in the ETCS Data meta-model based on the existing ERTMS/ETCS B model. The enumerations are **KindOfTrains** (values: TIMSERTMS, ERTMS, and NoERTMS) and **SupervisionStatus** (values: Active and Deactive). In class ETCSModel, the attribute **SupervisionOfTrain** of type enumeration **SupervisionStatus** is introduced to show concept **supervisor** from the existing B specification. In class **Train**, first attributes **TrainConnected** and **TrainReConnected** of type Boolean are introduced to show concepts **isConnected** and **reConnected** respectively and then attribute **KindofTrain** of type enumeration **KindOfTrains** is introduced to show concept **TrainKind**.

8.4 Translation of Meta-Model into B

Once the alignment between the ETCS data meta-model and EULYNX is done, Meeduse tool is used on the ETCS data meta-model for the translation into a B specification (eTCSData.mch). Due to the associations from ETCS data to EULYNX, the classes and associations of the EULYNX subset will also be translated. Figure 8.3 shows the translated B specification (without constants and variables) of selected concepts.

8.5 Linkage Invariants and Properties

After the translation of the meta-model into B, we apply our RefinementInclusion strategy in this application and generate the linkage B specifications. The abstract machine and refinements of existing B specification are refined with a different linkage machine at each level where the B machine issued from the meta-model is included. During this application, we generated different linkage properties and invariants in the linkage machines, which map concepts of EULYNX with the existing proved ERTMS/ETCS B specification. The linkage invariants and properties were previously discussed in chapters 5 and 6 and their semi-automatic generation in chapter 7. In this section, we discuss some of the important mappings (linkages) which illustrate how our approach is applied to EULYNX.

8.5.1 Mapping class SectionList to route (minTTD to maxTTD)

The existing B specification is built on a single route model, which starts from constant **minTTD** to constant **maxTTD**. EULYNX allows the design of railway topologies with multiple routes. During this application, we mapped only one route at a time with the existing B specification. In EULYNX, class **SectionList** is part of the route body, which contains the list of sections (**TvpSection**) via composition association **HasSection**. This mapping generates the following constant and properties.

CONSTANTS:
$\quad Linked_SectionList$
PROPERTIES:
$\quad Linked_SectionList \in SectionList \wedge$
$\quad Linked_SectionList \in \{ROUTEBODYPROPERTY1\} \wedge$
$\quad minTTD = 1 \wedge$
$\quad maxTTD = \mathbf{card}(\mathbf{ran}(\{Linked_SectionList\} \lhd HasSection))$

The constant **Linked_SectionList** is defined as an element of **SectionList** and *ROUTEBODYPROPERTY1* is assigned to it, which is an instance of **SectionList**. The value of **minTTD** is set to 1 and the value of **maxTTD** is set to the cardinality of the range (ran) of sections in the **Linked_SectionList**.

8.5.2 Mapping class TvpSection to Ttds

To map the **TvpSection** with **Ttds**, first, we introduced two definitions. The **TTD** which is defined as the **TvpSections** of the mapped route (**Linked_SectionList**). The **TTDConnection** defines the relation between instances of class **TvpSection** using the associations **elementA** and **elementB** in the RSM entities of EULYNX. This relation is achieved using multiple set operations like the inverse of relation (-1) and relation composition (;). Once the definitions are introduced, we generate a constant **Linked_TvpSection** and the corresponding properties.

MACHINE $\mathit{eTCSData}$
SETS
$\mathit{KindOfTrains} = \{\mathit{TIMSERTMS}, \mathit{ERTMS}, \mathit{NoERTMS}\};$
$\mathit{SupervisionStatus} = \{\mathit{Active}, \mathit{Deactive}\};$
$\mathit{SectionVacancyTypes} = \{\mathit{E_occupied}, \mathit{E_vacant}, \mathit{E_ambigious}, \mathit{E_unknown}\};$
$\mathit{ETCSMODEL};\ \mathit{TRAIN};\ \mathit{POSITIONINGNETELEMENT};$
$\mathit{POSITIONEDRELATION};\ \mathit{ASSOCIATEDNETELEMENT};$
$\mathit{AREALOCATION};\ \mathit{TRACKASSET};\ \mathit{ROUTEBODYPROPERTY};$
$\mathit{DOCUMENTROOT};\ \mathit{DATAPREPENTITIES};$
$\mathit{ASSETANDSTATE};\ \mathit{RSMENTITIES};\ \mathit{TOPOLOGY}$
VARIABLES [. . .]
CONSTANTS [. . .]
PROPERTIES
$\mathit{ETCSModel} \in \mathrm{Pow}\ (\mathit{ETCSMODEL}) \wedge \mathit{Train} \in \mathrm{Pow}\ (\mathit{TRAIN}) \wedge$
$\mathit{PositioningNetElement} \in \mathrm{Pow}\ (\mathit{POSITIONINGNETELEMENT}) \wedge$
$\mathit{PositionedRelation} \in \mathrm{Pow}\ (\mathit{POSITIONEDRELATION}) \wedge$
$\mathit{AssociatedNetElement} \in \mathrm{Pow}\ (\mathit{ASSOCIATEDNETELEMENT}) \wedge$
$\mathit{AreaLocation} \in \mathrm{Pow}\ (\mathit{AREALOCATION}) \wedge \mathit{RsmEntities} \in \mathrm{Pow}\ (\mathit{RSMENTITIES}) \wedge$
$\mathit{DocumentRoot} \in \mathrm{Pow}\ (\mathit{DOCUMENTROOT}) \wedge$
$\mathit{DataPrepEntities} \in \mathrm{Pow}\ (\mathit{DATAPREPENTITIES}) \wedge$
$\mathit{LinearElement} \subseteq \mathit{PositioningNetElement} \wedge \mathit{SectionList} \subseteq \mathit{ROUTEBODYPROPERTY} \wedge$
$\mathit{SectionAndVacancy} \subseteq \mathit{ASSETANDSTATE} \wedge \mathit{TvpSection} \subseteq \mathit{TRACKASSET} \wedge$
$\mathit{TdsSection} \subseteq \mathit{TRACKASSET} \wedge \mathit{TdsCompnent} \subseteq \mathit{TRACKASSET} \wedge$
$\mathit{VirtualSubSection} \subseteq \mathit{TdsSection} \wedge \mathit{VssLimit} \subseteq \mathit{TdsCompnent} \wedge$
$\mathit{KindOfTrain} \in \mathit{Train} \nrightarrow \mathit{KindOfTrains} \wedge$
$\mathit{ElementARelation} \in \mathit{PositionedRelation} \rightarrow \mathit{PositioningNetElement} \wedge$
$\mathit{ElementBRelation} \in \mathit{PositionedRelation} \rightarrow \mathit{PositioningNetElement} \wedge$
$\mathit{ToNetElement} \in \mathit{AssociatedNetElement} \rightarrow \mathit{PositioningNetElement} \wedge$
$\mathit{AreaAssociatedNetElements} \in \mathit{AssociatedNetElement} \nrightarrow \mathit{AreaLocation} \wedge$
$\mathit{TvpSectionLocation} \in \mathit{TvpSection} \rightarrow \mathit{AreaLocation} \wedge$
$\mathit{ApplyTdsToTvp} \in \mathit{TdsSection} \rightarrow \mathit{TvpSection} \wedge$
$\mathit{LimitsTdsSection} \in \mathit{TdsCompnent} \leftrightarrow \mathit{TdsSection} \wedge$
$\mathit{HasSection} \in \mathit{SectionList} \leftrightarrow \mathit{TvpSection} \wedge$
$\mathit{VacancyToTds} \in \mathit{SectionAndVacancy} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathit{TdsSection}$
INVARIANT
$\mathit{TrackAsset} \in \mathrm{Pow}\ (\mathit{TRACKASSET}) \wedge$
$\mathit{RouteBodyProperty} \in \mathrm{Pow}\ (\mathit{ROUTEBODYPROPERTY}) \wedge$
$\mathit{AssetAndState} \in \mathrm{Pow}\ (\mathit{ASSETANDSTATE}) \wedge \mathit{Topology} \in \mathrm{Pow}\ (\mathit{TOPOLOGY}) \wedge$
$\mathit{VirtualTrackAsset} \subseteq \mathit{TrackAsset} \wedge \mathit{PhysicalTrackAsset} \subseteq \mathit{TrackAsset} \wedge$
$\mathit{TrainFront} \in \leftrightarrow \mathit{VirtualSubSection} \wedge \in \mathit{Train} \leftrightarrow \mathit{VirtualSubSection} \wedge$
$\mathit{TrainRearLocation} \in \mathit{Train} \leftrightarrow \mathit{VirtualSubSection} \wedge$
$\mathit{TrainFrontLocation} \in \mathit{Train} \leftrightarrow \mathit{VirtualSubSection} \wedge$
$\mathit{frontMA} \in \mathit{Train} \nrightarrow \mathit{VirtualSubSection} \wedge \mathit{rearMA} \in \mathit{Train} \nrightarrow \mathit{VirtualSubSection} \wedge$
$\mathit{ListOfTrains} \in \mathit{Train} \nrightarrow \mathit{ETCSModel} \wedge \mathit{EulynxModels} \in \mathit{ETCSModel} \nrightarrow \mathit{DocumentRoot} \wedge$
$\mathit{SupervisionOfTrain} \in \mathit{ETCSModel} \nrightarrow \mathit{SupervisionStatus} \wedge$
$\mathit{TrainConnected} \in \mathit{Train} \nrightarrow \mathbf{BOOL} \wedge \mathit{TrainReConnected} \in \mathit{Train} \nrightarrow \mathbf{BOOL} \wedge$
$\mathit{inVacancyState} \in \mathit{SectionAndVacancy} \nrightarrow \mathit{SectionVacancyTypes} \wedge$
$\mathit{TdsSection} \cap \mathit{TvpSection} = \emptyset \wedge \mathbf{dom}(\mathit{HasSection}) = \mathit{SectionList} \wedge$
$\mathit{SectionList} \subseteq \mathit{RouteBodyProperty} \wedge \mathit{SectionAndVacancy} \subseteq \mathit{AssetAndState} \wedge$
$\mathit{TvpSection} \subseteq \mathit{VirtualTrackAsset} \wedge \mathit{TdsSection} \subseteq \mathit{VirtualTrackAsset} \wedge$

Figure 8.3: Structural part of machine eTCSData.mch (without constants and variables)

DEFINITIONS:
$TTD == HasSection[\{Linked_SectionList\}];$
$TTDConnection == (\ (TvpSectionLocation;AreaAssociatedNetElements^{-1};$
$\quad ToNetElement);(ElementARelation^{-1};ElementBRelation);$
$\quad (TvpSectionLocation;AreaAssociatedNetElements^{-1};ToNetElement)^{-1}\ );$
CONSTANTS:
$Linked_TvpSection$
PROPERTIES:
$Linked_TvpSection \in TTD \rightarrowtail\!\!\!\!\twoheadrightarrow Ttds \land$
$\forall\ (tk1,tk2).(tk1 \in TTD \land tk2 \in TTD \land (tk1 \mapsto tk2) \in TTDConnection \Rightarrow$
$\quad Linked_TvpSection(tk2) = Linked_TvpSection(tk1) + 1)$

In the properties, **Linked_TvpSection** is defined as a relation: $TTD \rightarrowtail\!\!\!\!\twoheadrightarrow Ttds$ (TTD is bijection to Ttds). The second property states that for all elements (pairs) tk1 and tk2 in the set TTD. Further it states that, if tk1 and tk2 satisfy the condition specified by the TTDConnection relation then the value of **Linked_TvpSection** for tk2 is equal to the value of **Linked_TvpSection** for tk1 incremented by 1.

8.5.3 Mapping enumeration SectionVacancyTypes to set StateTTD

SectionVacancyTypes is an enumeration in EULYNX and **StateTTD** is an enumeration in M0 machine of the existing B specification with values *freeT* and *occupiedT* . Mapping both of them generates the following constant and property.

CONSTANTS:
$Linked_SectionVacancyTypes$
PROPERTIES:
$Linked_SectionVacancyTypes = \{freeT \mapsto E_vacant, occupiedT \mapsto E_occupied\}$

SectionVacancyTypes has four values in EULYNX subset, but here we just mapped *E_vacant* and *E_occupied*with corresponding values *freeT* and *occupiedT* .

8.5.4 Mapping TvpSection occupancy to Ttds occupancy (variable stateTTD)

In EULYNX, the occupancy of TvpSection is defined using a variable **inVacancyState** of type enum **SectionVacancyTypes** in class **SectionAndVacancy**. Then this class refers to the class **TdsSection**, which further refers to the class **TvpSection**. Mapping this relation to the variable **stateTTD** produces the following invariant, which also involves the inverse of relation (-1) and relation composition (;).

INVARIANTS:
$(inVacancyState^{-1}\ ;\ VacancyToTds\ ;\ ApplyTdsToTvp\ ;\ Linked_TvpSection)$
$\quad = (Linked_SectionVacancyTypes^{-1}\ ;\ stateTTD^{-1}\ ;Linked_TvpSection^{-1}\ ;$
$\quad Linked_TvpSection)$

8.5.5 Mapping class VirtualSubSection to Vss

We also introduced definitions for this mapping. The **VSS** is defined as Virtual subsections (class **VirtualSubSection**) of the Tvp Sections (class **TvpSection**) of the linked route (**Linked_SectionList**). In EULYNX, the relation between two instances of **VirtualSubSection** is defined via class **VssLimit** and association **LimitsTdsSection**. The definition **VSSConnections** gives us the set of all the related virtual sub-sections for the mapped route. The next definition is a function **next_previous** that takes two elements, vs1 and vs2, and determines whether they have a next or previous relationship based on the difference of their **Linked_VirtualSubSection** values (it is a linkage constant defined as a relation: $VSS \rightarrowtail\!\!\!\twoheadrightarrow Vss$). If the **Linked_VirtualSubSection** value of vs1 is equal to the **Linked_VirtualSubSection** value of vs2 plus one, or if it is equal to the **Linked_VirtualSubSection** value of vs2 minus one, then the function returns true, indicating that vs1 and vs2 are next to each other or previous to each other in terms of their **Linked_VirtualSubSection** values. Otherwise, the function returns false. Once the definitions are done, we produce the constant **Linked_VirtualSubSection** and corresponding properties.

DEFINITIONS:
$VSS == \mathbf{ran}((\{Linked_SectionList\} \lhd HasSection);ApplyTdsToTvp^{-1})$;
$VSSConnections == \mathbf{ran}(\mathbf{fnc}(LimitsTdsSection \rhd VSS))$;
$next_previous(vs1,vs2) == ((Linked_VirtualSubSection(vs1) =$
$\quad Linked_VirtualSubSection(vs2)+1) \lor (Linked_VirtualSubSection(vs1) =$
$\quad Linked_VirtualSubSection(vs2)-1))$;
CONSTANTS:
$Linked_VirtualSubSection$
PROPERTIES:
$Linked_VirtualSubSection \in VSS \rightarrowtail\!\!\!\twoheadrightarrow Vss$
$\land \forall (vs1,vs2).(vs1 \in VSS \land vs2 \in VSS \land vs1 \neq vs2 \land \{vs1, vs2\} \in VSSConnections$
$\quad \Rightarrow next_previous(vs1,vs2))$
$\land \forall (vs1, vs2).(vs1 \in VSS \land vs2 \in VSS \land Linked_TvpSection(ApplyTdsToTvp(vs1))$
$\quad > Linked_TvpSection(ApplyTdsToTvp(vs2))$
$\quad \Rightarrow Linked_VirtualSubSection(vs1) > Linked_VirtualSubSection(vs2))$

There are three properties in this mapping. The first one is that of **Linked_VirtualSubSection**. The second property asserts that for all elements (pairs) vs1 and vs2 in the set **VSS**, if vs1 and vs2 are distinct elements and the set vs1, vs2 is a valid element in the set **VSSConnections**, then the function **next_previous(vs1, vs2)** returns true. The third property makes sure that virtual subsections in **Linked_VirtualSubSection** are mapped in the same order as their associated Tvp sections in the **Linked_TvpSection**. It asserts that for all elements (pairs) vs1 and vs2 in the set VSS, if the value associated with vs1, obtained by applying the **ApplyTdsToTvp** in **Linked_TvpSection**, is greater than the value associated with vs2 obtained in the same way, then the value associated with vs1, in **Linked_VirtualSubSection**, is also greater than the value associated with vs2 obtained similarly.

8.5.6 Other examples

We do not cover all the linkage invariants and properties in this chapter, but some other examples, which will be used in the linkage operations illustrated in the next section. These include mapping class **Train** to set **Trains** (of the existing specification), class

ETCSModel to a B machine, and an attribute **SupervisionOfTrain** (of enum type) to the boolean supervisor. These mappings are also illustrated in chapter 7. Following are the produced constants, properties and invariants from these mappings.

CONSTANTS:
$Linked_Train$,
$Linked_ETCSModel$
PROPERTIES:
$Linked_Train \in Train \rightarrowtail\!\!\!\!\twoheadrightarrow Trains \wedge$
$Linked_ETCSModel \in ETCSModel$
INVARIANTS:
$(supervisor = \textbf{TRUE} \Rightarrow SupervisionOfTrain(Linked_ETCSModel) = Active) \wedge$
$(supervisor = \textbf{FALSE} \Rightarrow SupervisionOfTrain(Linked_ETCSModel) = Deactive)$

8.6 Linkage Operations

As mentioned in chapter 5, we refine the operations in our RefinementInclusion strategy. We copy the body of the operation from the existing B specification and introduce a setter operation from the included machine to preserve the linkage variables. Figure 8.4 shows such a refined operation called **trainEntring** from the linkage machines. The operation is intended to handle the process of a train entering a particular section. It first selects an appropriate train (*tr*) that satisfies certain conditions (not yet assigned to a front occupation, **FALSE** supervisor). Then, it sets the front and rear occupations of the train to the *minTTD* (the first section in the route) and assigns the supervisor. Finally, it activates the supervision of the train. In order to preserve the linkage variables that we produced, such as for the train's front and rear occupation and also the activation of the supervisor, we introduced the setters **SetTvpSectionFront**, **SetTvpSectionRear**, and **SetSupervisionOfTrain** from the included machine.

OPERATIONS:
trainEntring =
  ANY *tr* **WHERE**
    $tr \in TRAINS$ - **dom**$(trainOccupationTTDFront)$
    $\wedge$ $supervisor =$ **FALSE**
    $\wedge$ $tr \in Trains$
  THEN
    trainOccupationTTDFront$(tr) := minTTD$;
    SetTvpSectionFront$(Linked_Train^{-1}(tr),(Linked_TvpSection^{-1}(minTTD)))$;
    trainOccupationTTDRear$(tr) := minTTD$;
    SetTvpSectionRear$(Linked_Train^{-1}(tr),(Linked_TvpSection^{-1}(minTTD)))$;
    $supervisor :=$ **TRUE**;
    SetSupervisionOfTrain$(Linked_ETCSModel, Active)$
  END

Figure 8.4: Operation trainEntering

The operations **SetTvpSectionFront** and **SetTvpSectionRear** take a train and a Tvp section as arguments and set the train's front and rear to the provided Tvp section. In this operation, we set them to the Tvp section mapped to the *minTTD*. The operation **SetSupervisionOfTrain** updates the "SupervisionOfTrain" relation within an ETCSModel. It takes an ETCSModel and a SupervisionStatus as input. Here, we provide

the ETCSModel, which is mapped to B machine. For the status, we provide *Active* as it is mapped to the boolean value **TRUE**. We applied this approach to all the operations during this application.

8.7 Visualization

Once the linkage machines are produced, we can animate the existing B specifications, but first, we need the graphical models and views for our DSL. For this application, we provide a graphical editor to design topology, a tabular view (previously illustrated in chapter 6), state view of TTDs and VSSs, and a TTD-VSS view (inspired by ETCS document). All the views of a model in Eclipse Sirius are synchronized.

8.7.1 Topology Design

The EULYNX subset contains RSM concepts which provides an abstraction for the railway topology. Using these concepts, we provide a graphical editor which allow users to design railway topologies using tracks where each track corresponds to an instance of class TvpSection. Figure 8.5 shows such a designed topology composed of seven track sections. As EULYNX also provides the definition of a route (class SectionList with composition *HasSection* to class TvpSection), we extract two routes from the given topology in Figure 8.5. Route1 is composed of Track1, Track2, Track3, Track4, and Track5 while Route2 is composed of Track1, Track2, Track3, Track6, and Track7. The tracks: Track1, Track2, and Track3 are the common (intersecting) tracks in both routes.

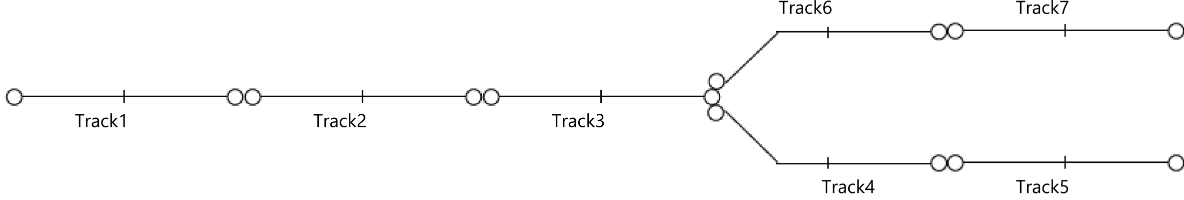


Figure 8.5: Topology Design

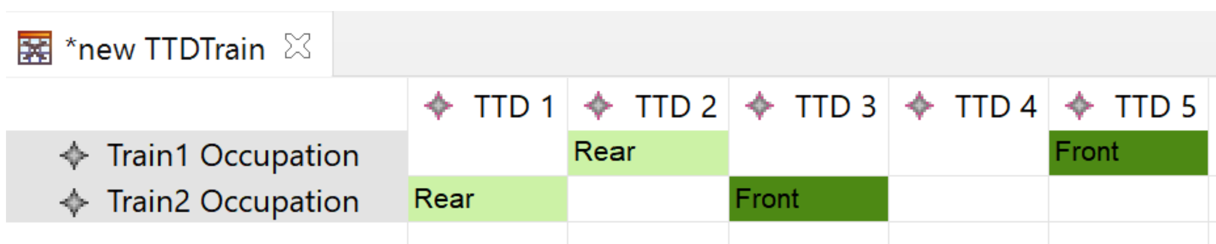
8.7.2 Tabular Route View

The tabular views of sections and trains (such as Figure 6.5)are also illustrated in chapter 6 during the validation of the same existing ERTMS/ETCS B specification where the DSL is built in an iteratively formal driven way. During this application, we keep the same tabular view for illustrating trains' front and rear occupation and location. In our model, we provided two TIMS trains: Train1 and Train2, where Train1 is a connected train. Note that the EULYNX concept TvpSection and Ttds (from the existing B specification) are the same and mapped in the linkage machines, so in our graphical views, we name this concept TTD.

8.7.2.1 Level 1: M0

Figure 8.6 illustrates the tabular view of Route1 with trains' occupation. This view is developed keeping in mind the concepts and executing operations of existing machine M0. In this Figure, the front and rear of Train1 are on sections TTD 5 and TTD 2, respectively. At the same time, the front and rear of Train2 are on sections TTD 3

and TTD 1, respectively. From this illustration, the user can identify which sections are occupied by which train. It can be observed that TTD 2 and TTD 3 intersect between the occupations of Train1 and Train2. This is something unusual, but at this level (M0), it is normal as the condition that a section is only occupied by one train is provided in later levels.



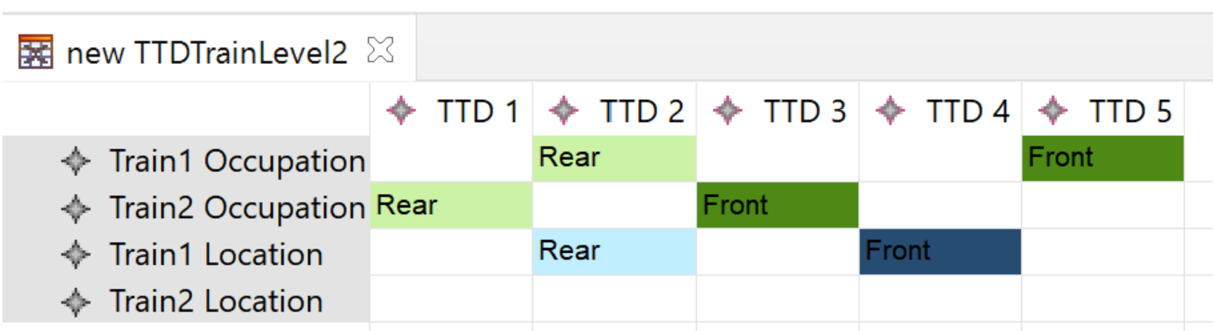
*new TTDTrain

	TTD 1	TTD 2	TTD 3	TTD 4	TTD 5
Train1 Occupation		Rear			Front
Train2 Occupation	Rear		Front		

Figure 8.6: Level 1

8.7.2.2 Level 2: M1

In level 2 of the existing B specification (refinement M1), the concept of train location is introduced, where a connected train communicates its location to the system. This concept is illustrated in Figure 8.7, where the front and rear locations of the connected train (Train1) are on TTD 4 and TTD 2, respectively. It can be observed that Train1's front occupation (TTD 5) and location (TTD 4) are not the same because the recent movement of the train is not yet communicated.



new TTDTrainLevel2

	TTD 1	TTD 2	TTD 3	TTD 4	TTD 5
Train1 Occupation		Rear			Front
Train2 Occupation	Rear		Front		
Train1 Location		Rear		Front	
Train2 Location					

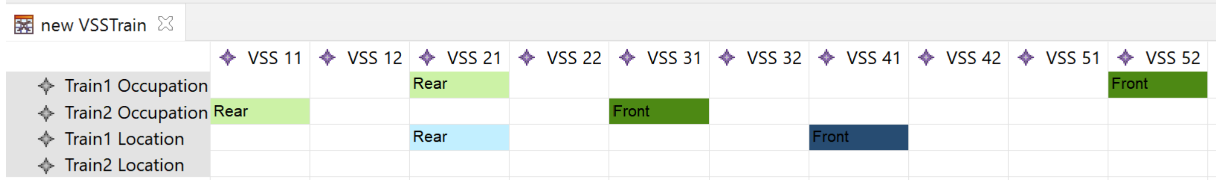
Figure 8.7: Level 2

8.7.2.3 Level 3: M2

In ERTMS/ETCS HL3, a TTD is further divided into smaller subsections, VSS. In level 3 (refinement M2) of the existing B specification, the concept of VSS replaces the TTD. In this regard, we provide a tabular view of Route1 with corresponding VSSs of TTD sections, shown in Figure 8.7. We executed the operations of M2 to see the movement of the train with this view, and we observed that the occupied sections of the train were still intersecting with each other as the condition that a section is only occupied by one train is still not yet provided.

8.7.2.4 Level 4: M3

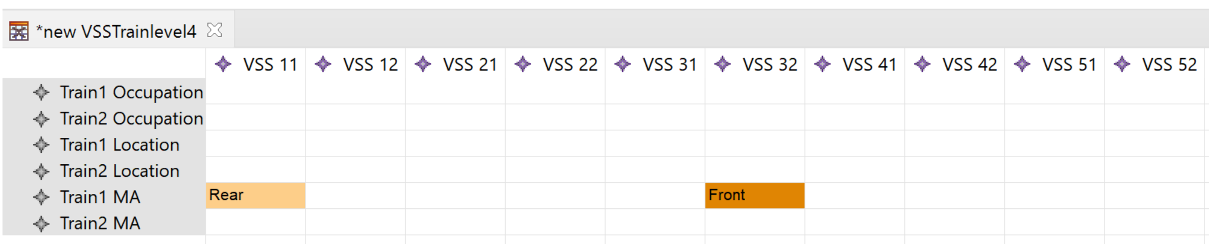
In level 4 (refinement M3), the concept of movement authority (MA) is introduced. To illustrate this concept, we provided a view, shown in Figure 8.9. In this Figure, train MA to corresponding VSSs are illustrated where the front and rear MA of Train1 are on VSS


new VSSTrain

	VSS 11	VSS 12	VSS 21	VSS 22	VSS 31	VSS 32	VSS 41	VSS 42	VSS 51	VSS 52
Train1 Occupation			Rear							Front
Train2 Occupation	Rear				Front					
Train1 Location			Rear				Front			
Train2 Location										

Figure 8.8: Level 3

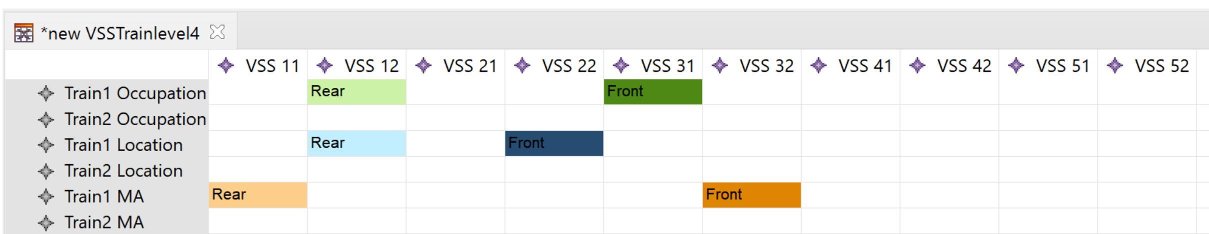
32 and VSS 11, respectively. This shows that Train1 has the authority to move from Vss 11 to VSS 32, and these VSSs cannot be assigned to the MA of another train.


*new VSSTrainlevel4

	VSS 11	VSS 12	VSS 21	VSS 22	VSS 31	VSS 32	VSS 41	VSS 42	VSS 51	VSS 52
Train1 Occupation										
Train2 Occupation										
Train1 Location										
Train2 Location										
Train1 MA	Rear					Front				
Train2 MA										

Figure 8.9: Level 4 (Assigning MA)

Once an MA is assigned to a train, the operations of M3 allow this train to move on the corresponding VSSs. This train movement is illustrated in Figure 8.10 Train1's occupation and location can be seen on the VSSs within the range of its MA.


*new VSSTrainlevel4

	VSS 11	VSS 12	VSS 21	VSS 22	VSS 31	VSS 32	VSS 41	VSS 42	VSS 51	VSS 52
Train1 Occupation		Rear			Front					
Train2 Occupation										
Train1 Location		Rear		Front						
Train2 Location										
Train1 MA	Rear					Front				
Train2 MA										

Figure 8.10: Level 4 (Train Movement)

8.7.3 State View

Our DSL also provides the state view of sections for each route. Figure 8.11 shows the legend of state view for the four states: Free, Occupied, Unknown, and Ambiguous. The section with green dots on the sides illustrates the state Free, yellow dots on the sides illustrate the state Occupied, red dots on the sides illustrate the state Unknown, and the ambiguous state is illustrated with orange dots on the side. The colors for the states are inspired by the ETCS document [32].

8.7.3.1 TTD states

Figure 8.12 shows the state views of Route1 and Route2 with respect to TTDs. This view only shows the states: Free and Occupied; and is helpful during the animation of M0 and M1, where the states of the TTDs can be observed while moving the train. The upper part of the Figure shows the Route1 where TTD 2 and TTD 5 are occupied. The bottom part shows the Route2 where only TTD 2 is occupied (TTD 1, TTD 2, and TTD 3 are common sections in both routes).

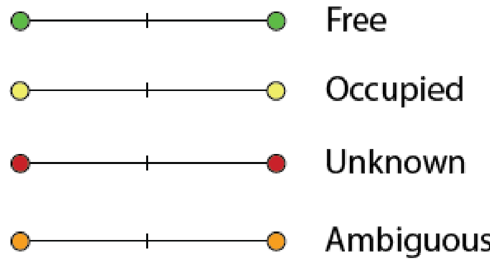


Figure 8.11: States Legend

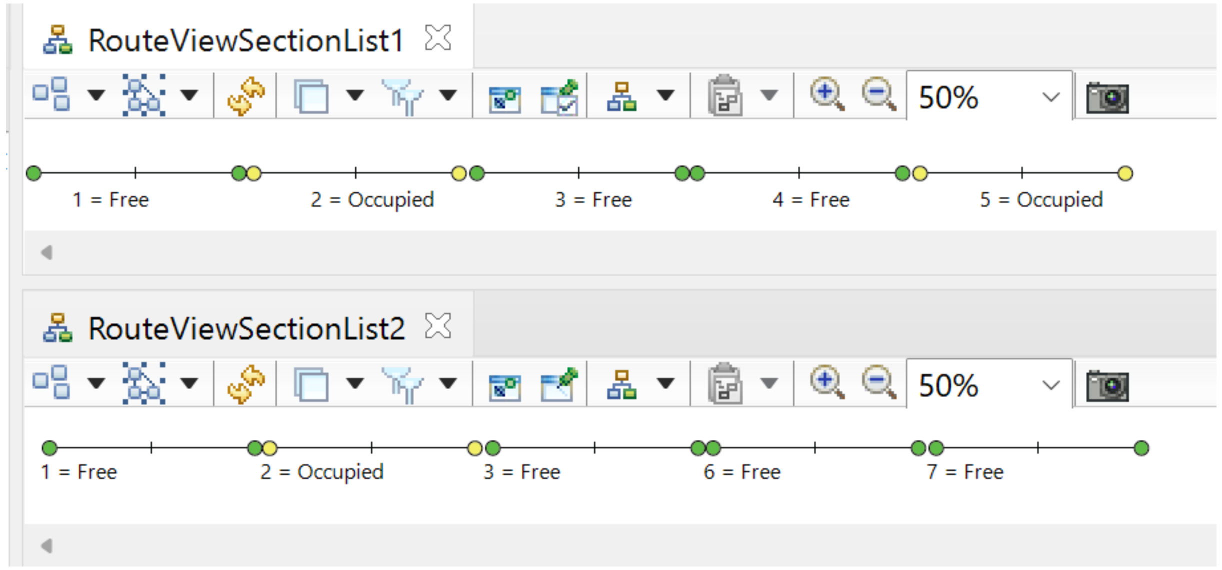


Figure 8.12: TTD States

8.7.3.2 VSS states

Figure 8.13 shows the state view of Route1 with respect to VSSs. This view shows all four states on a VSS, and it is useful during the animation of M2 and M3. The Figure shows a scenario where VSS 11, VSS 12, and VSS 21 are in the state Unknown. VSS 22 and VSS 31 are ambiguous, while all other VSSs are in the state Free.

Figure 8.13: VSS States of Route 1

8.7.4 ETCS Document's Style View

Our DSL provides another view of a route with TTDs and their corresponding VSSs, inspired by a graphical view provided in the ETCS document [32]. This view is illustrated in Figure 8.14, which shows the TTDs and VSSs of Route1. View clearly shows the ETCS rule; if the state of a single VSS is Ambiguous, Unknown, or Occupied, then its TTD is Occupied. In the Figure, there are the five TTDs of Route1 and each with its two VSSs. The state of TTD 1 is Occupied, as both of its VSSs are Unknown. The state of TTD 2 is also Occupied as one of its VSS is Unknown, and the other one is Ambiguous. One

VSS of TTD 3 is Ambiguous, and the other one is Free, but TTD is defined as Occupied due to that one Ambiguous VSS.

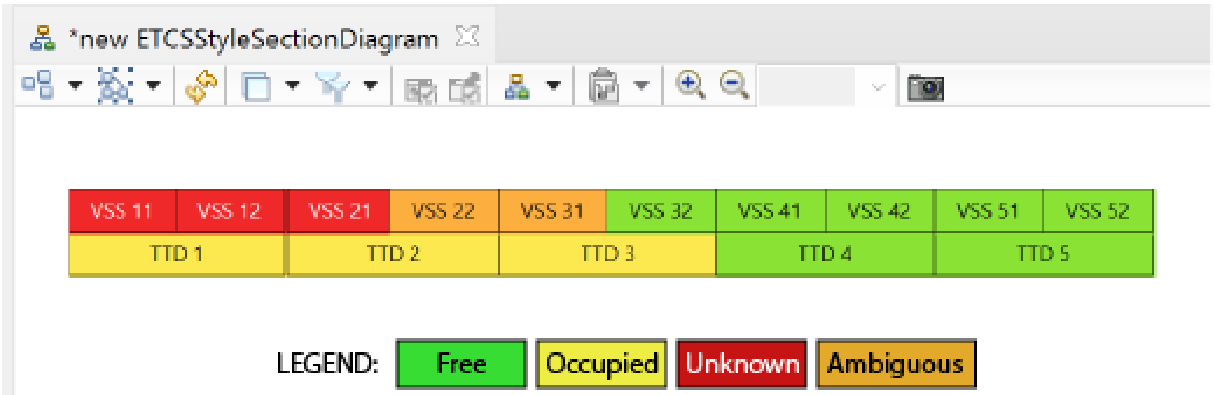


Figure 8.14: ETCS Document's Style View of Route 1

8.8 Conclusion

In this chapter, we provide a formalized DSL based on a EULYNX subset. We used our approach of linkage machines to validate an existing ERTMS/ETCS B specification provided by Amal Mammar [87]. We successfully animated the existing B specification using the DSL and visually observed its behavior. The EULYNX-based DSL provides different graphical views of a railway model, which includes the track-based railway topology, a tabular view of routes with trains and sections, a state view of TTDs and VSSs, and an ETCS-style view of the route. Apart from validation of the existing B specification, this application also contributes towards the formalization of EULYNX. In the literature, we did not find a lot of works related to the formalization of EULYNX, apart from [45] and [105]. The approach of [45] translates SysML models created within EULYNX to formal mCRL2 [18], a specification language for describing concurrent discrete event systems. They verify and test the EULYNX interface for point subsystems (also called 'turnouts' or 'switches') using model-checking and model-based testing. The paper [105] describes a formal verification approach for EULYNX models using the Event-B. They manually transform EULYNX SysML models into UML-B and embed the safety requirements (represented as invariants) with this UML-B. Then they use the Rodin modeling platform to check the models for any ambiguity or errors. They repeat this process until no errors are found. These approaches provide the formalization of EULYNX models but do not provide any DSL for designing EULYNX models. Our approach provides a DSL for the EULYNX subset and allows the user to design a graphical model of the system and illustrate this model using multiple views. To our knowledge, it is the only EULYNX-based DSL, and the behavior of the DSL is provided by ERTMS/ETCS B operations.

8.9 Résumé en français

Ce chapitre présente l'application de notre approche à EULYNX et ERTMS/ETCS. Nous fournissons un DSL formel basé sur un sous-ensemble de EULYNX. Nous avons utilisé notre approche de machines de liaison pour valider une spécification ERTMS/ETCS B existante fournie par Amel Mammar [87]. Nous avons animé avec succès la spécification B existante à l'aide du DSL et observé visuellement son comportement. Le DSL basé sur EULYNX fournit différentes vues graphiques d'un modèle ferroviaire, qui incluent la

topologie ferroviaire basée sur les voies, une vue tabulaire des itinéraires avec des trains et des sections, une vue d'état des TTD et des VSS et une vue de l'itinéraire adoptant le style de ETCS. Outre la validation de la spécification B existante, cette application contribue également à la formalisation d'EULYNX. Dans la littérature, nous n'avons pas trouvé beaucoup de travaux liés à la formalisation d'EULYNX, en dehors de [45] et [105]. L'approche de [45] traduit les modèles SysML créés dans EULYNX en mCRL2 [18], un langage de spécification pour décrire les systèmes à événements discrets concurrents. Ils vérifient et testent l'interface EULYNX pour les sous-systèmes d'aiguillages à l'aide de la vérification de modèles et de tests basés sur des modèles. L'article [105] décrit une approche de vérification formelle pour les modèles EULYNX utilisant Event-B. Ils transforment manuellement les modèles EULYNX SysML en UML-B et intègrent les exigences de sûreté (représentées comme des invariants) dans cet UML-B. Ensuite, ils utilisent la plateforme de modélisation Rodin pour vérifier les modèles en décelant toute ambiguïté ou erreur. Ils répètent ce processus jusqu'à ce qu'aucune erreur ne soit détectée. Ces approches fournissent la formalisation des modèles EULYNX mais ne fournissent aucun DSL pour la conception de modèles EULYNX. Notre approche fournit un DSL pour un sous-ensemble de EULYNX et permet à l'utilisateur de concevoir un modèle graphique du système et d'illustrer ce modèle à l'aide de plusieurs vues. À notre connaissance, il s'agit du seul DSL basé sur EULYNX, et le comportement du DSL est assuré par les opérations ERTMS/ETCS écrites en B et prouvées correctes.

Chapter 9

Conclusion and Perspectives

In the previous chapter, we demonstrated the application of our xDSL-based framework, which was presented in the first chapter. In this final chapter, we summarize the work done during this thesis. Then we present the perspectives of this work by discussing possible research directions that can extend this work.

9.1 Contribution

This thesis contributes to the areas of model-driven engineering (MDE) paradigm, formal methods (FM), and railway systems by providing a xDSL-based framework for merging railway standard notations, shown in Figure 9.1. The framework first allows railway experts to graphically design railway models based on EULYNX [11], which is aligned with RailSystemModel (RSM) [27]. Then the railway expert him/herself validates these models based on ERTMS/ETCS [32] operational rules. The framework is composed of two layers: the semantics layer and the execution layer.

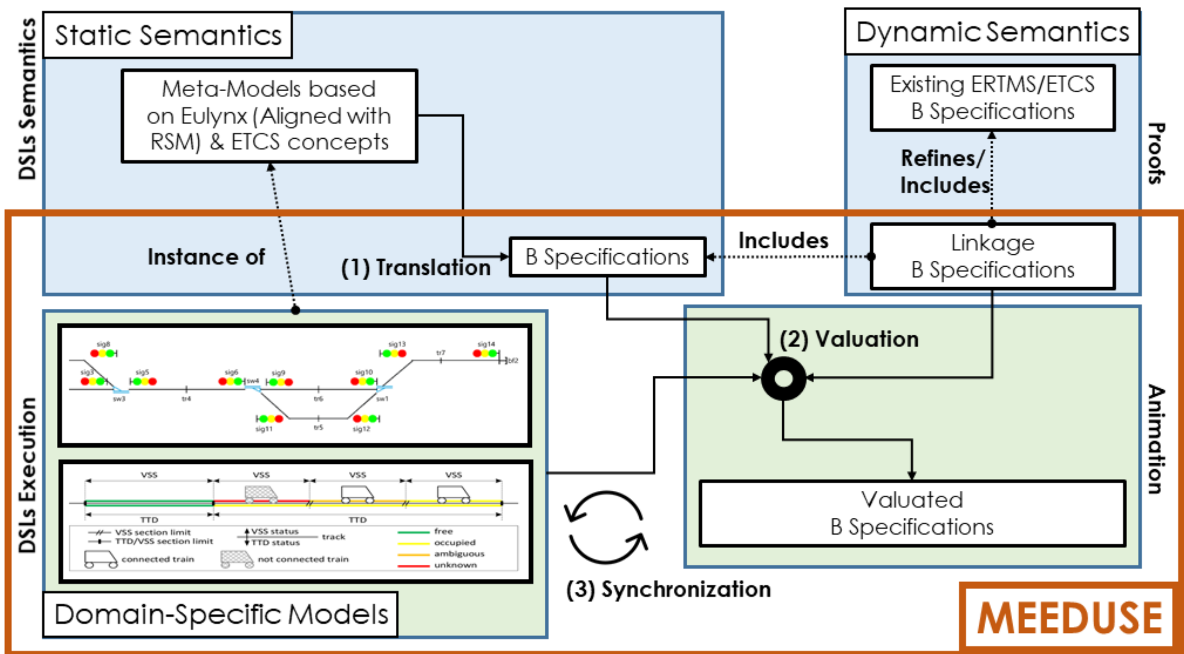


Figure 9.1: Our xDSL-based framework for merging railway standard notations

The semantics layer is divided into static semantics and dynamic semantics. The static semantics include meta-models based on EULYNX (a platform to standardize elements

and interfaces of railway signalling systems) and ETCS concepts (e.g., train). The meta-models are translated into B specifications using the Meeduse language workbench [60]. In our framework, we provide ERTMS/ETCS operational rules as the dynamic semantics of the DSL. For this purpose, existing proved ERTMS/ETCS B specifications are re-used. Dynamic semantics are also equipped with Linkage B specifications that link existing ERTMS/ETCS B specifications to B specifications translated from meta-models of EULYNX and ETCS concepts.

The execution layer is supported by Meeduse, where domain-specific models represent railway notations conforming to EULYNX. First, Meeduse injects these models into B specifications issued from meta-models. Then it asks ProB to compute the list of possible operations (from existing B specifications) that can be animated from that given state. Meeduse synchronizes the current state of the B specifications with the model resulting in graphical animation. Our framework allows railway experts to define their models and validate the behavior of these models. The concerns of the FM expert, railway expert, and MDE expert are separated in this framework. FM expert is responsible for verifying their specification and generating the linkage B specification. The definition of DSL in this framework is the task of an MDE expert. We derived four aspects from the application of our DSL-based framework during this thesis.

1. **Aspect 1. Visual animation of B specifications using DSLs.** In the literature, there are tools providing graphical animation and visualization of B specifications such as BRAMA [107], AnimB[1], B-Motion Studio [78], B-Motion Web [79], VisB [119], and animation function in [84]. However, typical drawbacks are associated with these tools: they use scripting or programming languages for visualization or for mapping. This added programming layer is error-prone. The diagnosis becomes more difficult because errors may come from the added programming layer used for visualization. Our first result of this thesis is to provide a solution to complement the tools mentioned above. Our solution is built on the Meeduse language workbench, where existing B models are mapped with B static semantics of DSL. Mapping is accomplished through a linkage B machine where the existing model is either refined or included in the linkage machine, and the static DSL machine is included. Our approach is applied to the existing B models of Lift, Scheduler, and realistic case study of ERTMS/ETCS. The linkage approach built around the linkage B machine is part of dynamic semantics in our DSL-based framework.

2. **Aspect 2. Pattern-based semi-automated generation of Linkage B specifications.** Our second result of this thesis is a follow-up of our first result, where linkage B machines are generated in a semi-automated way. We provided a pattern-based DSL tool where the user (FM expert) can define a catalog of generic reusable patterns (in Pattern-Definition DSL) and then apply those patterns (in Pattern-Application DSL) with parameters from the existing B and DSL models. In the last, the component Linkage Generator tool (built using Acceleo) produces the linkage B machine. The tool is experimented with models of Lift, Scheduler, and multi-component ERTMS/ETCS.

3. **Aspect 3. Iterative formal model-driven approach (B Model to DSL).** In our formal DSL-based framework, one of the objectives is to use the existing proved ERTMS/ETCS B specifications as dynamic semantics. We selected the ERTMS/ETCS hybrid level 3 specification of Amel Mammar [87] from the case study of ABZ'2018 [49]. The existing specification comprises four components (an

[1]AnimB: https://wiki.event-b.org/index.php/AnimB

abstract machine and three refinements). We propose an iterative formal model-driven approach where an ERTMS DSL is built and updated incrementally with each refinement of existing B specification. The operations of existing specifications are used to animate the graphical model based on DSL and allow to validate the system. During the validation, an error by the domain expert was identified where a track section appeared as "ambiguous" although a train was in the section, and the section should appear as "occupied". Our iterative formal model-driven approach resulted in, firstly, the creation of DSL from the existing B model and, secondly, proving the hypotheses *"Though FMs prove the consistency of a railway system, it does not guarantee it is correctly built"*.

4. **Aspect 4. Application to EULYNX.** Another objective of this thesis was to provide a DSL based on railway standard notations. We used the EULYNX and applied our approach of linkage machines to map it with the existing proved ERTMS/ETCS B specification such that the specification is visualized in order to be validated. The application resulted in a formalized EULYNX-based graphical DSL with multiple views where ERTMS/ETCS B operations are used to observe the model's behavior, such as train movement and change in the state of the sections.

9.2 Perspectives

To extend this thesis, some research works could be developed:

1. Perspective 1. In the contribution section, Aspect 3 discussed the creation of DSL from the existing B model. During this thesis, this step is done manually but systematically. This step can be automated using transformation tools but requires further investigation and study. Currently, Meeduse supports the generation of B specifications from DSL; a reverse engineering approach is required to generate DSLs from B specifications.

2. Perspective 2. The second perspective is somehow associated with the first perspective. The first perspective discusses the creation of DSL from the B model. In this thesis, we developed a DSL issued from B specification where DSL is updated incrementally with each refinement, but another interesting approach could be investigated. The new approach could be to issue a DSL from each component of the B model, and the DSL is refined by the other DSL issued from the refined component. This topic of DSL refinement is currently one of the hot topics in the field of DSLs and model-driven engineering.

3. Perspective 3. While applying our formal DSL-based framework, we used Meeduse to translate DSL semantics to B specification. Meeduse generates static semantics (B data, predicates, etc.) and dynamic semantics (setters, getters, constructors, and deconstructors as operations). In this thesis, we used operations of existing proved B specifications as dynamic semantics. The existing operations are updated with setters and getters of the DSL machine. Another approach could be to define railway standardized events or actions using activity diagrams and state chart diagrams and then to associate these diagrams with the concerned DSL. Next step could be to equip Meeduse with functionality to generate the dynamic semantics of the DSL from these associated activity and state chart diagrams.

9.3 Résumé en français

Ce chapitre résume le travail effectué au cours de cette thèse. Il présente ensuite les perspectives de ce travail en discutant des orientations de recherche possibles pouvant prolonger ce travail. Dans la première partie, il est indiqué que cette thèse combine l'ingénierie dirigée par les modèles (MDE), les méthodes formelles (FM) et les standards ferroviaires en fournissant un cadre basé sur des DSLs. Cela permet d'abord aux experts ferroviaires de concevoir graphiquement des modèles ferroviaires basés sur EULYNX [11], qui est aligné sur RailSystemModel (RSM) [27]. Ensuite, l'expert ferroviaire valide lui-même ces modèles en s'appuyant sur les règles opérationnelles de ERTMS/ETCS [32]. La contribution de ce travail de cette thèse se décline selon quatre points. Le premier est l'animation visuelle des spécifications B à l'aide de DSLs. Le second est la génération semi-automatique basée sur des modèles de spécifications B de liaison. Le troisième est l'approche itérative formelle basée sur un modèle dans laquelle un DSL ERTMS est construit et mis à jour progressivement à chaque raffinement de la spécification B existante. Le dernier est l'application à EULYNX.

Dans la deuxième partie, trois perspectives sont énoncées qui pourraient prolonger les travaux de cette thèse. La première est la création automatisée de DSLs à partir de modèles B. La deuxième perspective pointe vers le raffinement de DSLs. La troisième et dernière est liée à la génération de la sémantique dynamique du DSL à partir de diagrammes d'activités et d'états.

Bibliography

[1] Acceleo. https://www.eclipse.org/acceleo/. Accessed: 2023-02-05.

[2] AtelierB. https://www.atelierb.eu/en/. Accessed: 2023-03-14.

[3] B Method. https://www.systerel.fr/en/expertise/formal-methods/b-method/. Accessed: 2023-05-17.

[4] CENELEC - Railways and Hyperloop Systems. https://www.cencenelec.eu/areas-of-work/cenelec-sectors/transport-and-packaging-cenelec/railways-and-hyperloop-systems/. Accessed: 2023-07-06.

[5] Deliverable d 2.1 specification of formal development demonstrator. https://projects.shift2rail.org/download.aspx?id=560cdd44-83e7-4f5d-879e-d8dcdf2e2b1b. Accessed: 2021-07-24.

[6] Eclipse OCL. https://projects.eclipse.org/projects/modeling.mdt.ocl. Accessed: 2023-07-10.

[7] Eclipse Sirius. https://www.obeosoft.com/fr/produits/Eclipse-sirius. Accessed: 2023-06-07.

[8] ENTERPRISE ARCHITECT. https://sparxsystems.com/products/ea/. Accessed: 2023-06-30.

[9] ERTMS Hybrid Level 3. https://github.com/meeduse/Samples/blob/main/ETCSLevel3/ERTMS_Hybrid_Level_3.pdf. Accessed: 2020-06-26.

[10] Eugenia. https://www.eclipse.org/epsilon/doc/eugenia/. Accessed: 2023-06-07.

[11] eulynx.eu. https://eulynx.eu/. Accessed: 2023-01-13.

[12] EventB.org. http://www.event-b.org/. Accessed: 2023-05-16.

[13] Formal Methods Case Studies for DO-333. http://ntrs.nasa.gov/citations/20140004055. Accessed: 2023-02-28.

[14] Graphical Modeling Project (GMP). https://www.eclipse.org/modeling/gmp/. Accessed: 2023-06-07.

[15] IFC Rail Project. https://www.buildingsmartusa.org/wp-content/uploads/2020/06/RWR-IFC_Rail-Context-Approach.pdf. Accessed: 2023-07-03.

[16] Industrial Railway CAD software. https://www.railcomplete.com/.

[17] lotus-tool. https://gesad.github.io/projects/lotus-tool/. Accessed: 2023-08-12.

[18] mCRL2. https://mcrl2.org/web/index.html. Accessed: 2023-08-04.

[19] MDE Tools. https://openembedd.inria.fr/MDE/MDEtools/index.html. Accessed: 2023-06-06.

[20] Meeduse Git Repository. https://github.com/meeduse/Samples. Accessed: 2023-08-16.

[21] MetaEdit+. https://www.metacase.com/mep/. Accessed: 2023-06-30.

[22] Object Management Group (OMG). https://www.omg.org/index.htm. Accessed: 2023-06-08.

[23] OpenETCS Project. http://openetcs.org/. Accessed: 2021-07-19.

[24] Overview of the Domain-Specific Language Tools User Interface. https://learn.microsoft.com/en-us/visualstudio/modeling/overview-of-the-domain-specific-language-tools-user-interface?view=vs-2022. Accessed: 2023-06-30.

[25] ProB-Examples. https://github.com/hhu-stups/specifications/tree/master/prob-examples/B. Accessed: 2023-06-30.

[26] RailML.org. https://www.railml.org/en/home.html. Accessed: 2020-01-06.

[27] RailSystemModel. https://rsm.uic.org/. Accessed: 2023-05-23.

[28] Railway Infrastructure and Layout Aided Designer. https://www.rail-aid.com/.

[29] SafeCap Platform. http://safecap.sourceforge.net/index.shtml. Accessed: 2021-07-15.

[30] SPARK Pro. https://www.adacore.com/sparkpro. Accessed: 2023-08-12.

[31] The B-Toolkit. https://web.archive.org/web/20041012141220/http://www.b-core.com/ONLINEDOC/BToolkit.html. Accessed: 2023-05-16.

[32] The ERTMS/ETCS signalling system. http://www.railwaysignalling.eu/wp-content/uploads/2016/09/ERTMS_ETCS_signalling_system_revF.pdf. Accessed: 2022-03-16.

[33] UIC - International Union of Railways. https://uic.org/. Accessed: 2023-06-26.

[34] WHAT IS FORMAL METHODS? http://shemesh.larc.nasa.gov/fm/fm-what.html. Accessed: 2023-02-28.

[35] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, USA, 1996.

[36] V. S. Alagar and K. Periyasamy. *The Z Notation*, pages 461–538. Springer London, London, 2011.

[37] Jon Barwise. An Introduction to First-Order Logic. In Jon Barwise, editor, *HANDBOOK OF MATHEMATICAL LOGIC*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 5–46. Elsevier, 1977.

[38] Luteberget Bjørnar and Johansen Christian. Efficient verification of railway infrastructure designs against standard regulations. *Formal Methods in System Design*, 51:1–32, 2018.

[39] Dines Bjørner. Formal Software Techniques for Railway Systems. *IFAC Proceedings Volumes*, 33(9):101–108, 2000. 9th IFAC Symposium on Control in Transportation Systems 2000, Braunschweig, Germany, 13-15 June 2000.

[40] Dines Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In *CTS2003: 10th IFAC Symposium on Control in Transportation Systems, August, Seikei University*, United Kingdom, 2003. Elsevier.

[41] Dines Bjørner. *Domain Engineering - Technology Management, Research and Engineering*, volume 4 of *COE Research Monograph Series*. JAIST, 2009.

[42] Jean-Paul Bodeveix, Mamoun Filali, Julia Lawall, and Gilles Muller. Formal Methods Meet Domain Specific Languages. In *Proceedings of the 5th International Conference on Integrated Formal Methods*, page 187–206. Springer, 2005.

[43] Mark Bosschaart, Egidio Quaglietta, Bob Janssen, and Rob M.P. Goverde. Efficient formalization of railway interlocking data in RailML. *Information Systems*, 49:126–141, 2015.

[44] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient Debugging for Executable DSLs. *Journal of Systems and Software*, 137:261–288, 2018.

[45] Mark Bouwman, Djurre van der Wal, Bas Luttik, Mariëlle Stoelinga, and Arend Rensink. A Case in Point: Verification and Testing of a EULYNX Interface. *Form. Asp. Comput.*, 35(1), mar 2023.

[46] Jonathan P. Bowen and Michael G. Hinchey. *Applications of Formal Methods.* Prentice Hall PTR, USA, 1st edition, 1995.

[47] Stephen D. Brookes and A.W. Roscoe. *CSP: A Practical Process Algebra*, page 187–222. Association for Computing Machinery, New York, NY, USA, 1 edition, 2021.

[48] Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, and Gabor Karsai. Challenges and Directions in Formalizing the Semantics of Modeling Languages. *Computer Science and Information Systems*, 8(2):225–253, 2011.

[49] Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors. *6th International Conference, ABZ*, volume 10817 of *LNCS*. Springer, 2018.

[50] A. Chiappini, A. Cimatti, L. Macchi, O. Rebollo, M. Roveri, A. Susi, S. Tonetta, and B. Vittorini. Formalization and validation of a subset of the European Train Control System. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 109–118, 2010.

[51] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude – a high-performance logical framework. How to specify, program and verify systems in rewriting logic. With CD-ROM.*, volume 4350. Berlin: Springer, 2007.

[52] Marcelo Paternostro Ed Merks Dave Steinberg, Frank Budinsky. *EMF: Eclipse Modeling Frameworke*. Addison-Wesley, 2nd edition, 2009.

[53] Jack B. Dennis. *Petri Nets*, pages 1525–1530. Springer US, Boston, MA, 2011.

[54] S. Eker, J. Meseguer, and Ambarish Sridharanarayanan. The Maude LTL Model Checker. *Electron. Notes Theor. Comput. Sci.*, 71:162–187, 2002.

[55] J. Endresen, Erik Carlson, Thomas Moen, K. Alme, Øystein Haugen, Gøran K. Olsen, and A. Svendsen. Train Control Language – Teaching Computers Interlocking. *WIT Transactions on the Built Environment*, 103:651–660, 2008.

[56] Alessio Ferrari and Maurice H. Ter Beek. Formal Methods in Railways: A Systematic Mapping Study. *ACM Comput. Surv.*, 55(4), nov 2022.

[57] Brian T. Graham. *Formal Methods and Verification*, pages 1–7. Springer US, Boston, MA, 1992.

[58] Nicolas Halbwachs. A Synchronous Language at Work: The Story of Lustre. In Stefania Gnesi and Tiziana Margaria, editors, *Formal Methods for Industrial Critical Systems*, pages 15–31. John Wiley & Sons, Inc., Hoboken, NJ, USA, November 2012.

[59] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, USA, 2004.

[60] Akram Idani. Meeduse: A Tool to Build and Run Proved DSLs. In Brijesh Dongol and Elena Troubitsyna, editors, *Integrated Formal Methods*, pages 349–367, Cham, 2020. Springer International Publishing.

[61] Akram Idani and Yves Ledru. B for Modeling Secure Information Systems - The B4MSecure Platform. In *ICFEM 2015*, volume 9407 of *LNCS*, pages 312–318. Springer, 2015.

[62] Akram Idani, Yves Ledru, Abderrahim Ait Wakrime, Rahma Ben Ayed, and Philippe Bon. Towards a Tool-Based Domain Specific Approach for Railway Systems Modeling and Validation. In Simon Collart-Dutilleul, Thierry Lecomte, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 23–40, Cham, 2019. Springer International Publishing.

[63] Akram Idani, Yves Ledru, Abderrahim Ait Wakrime, Rahma Ben Ayed, and Simon Collart-Dutilleul. Incremental Development of a Safety Critical System Combining formal Methods and DSMLs. In Kim Guldstrand Larsen and Tim Willemse, editors, *Formal Methods for Industrial Critical Systems*, pages 93–109, Cham, 2019. Springer International Publishing.

[64] Akram Idani, Yves Ledru, and Germán Vega. Alliance of model-driven engineering with a proof-based formal approach. *Innov. Syst. Softw. Eng.*, 16(3):289–307, 2020.

[65] Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky. The SafeCap Platform for Modelling Railway Safety and Capacity. In *Computer Safety, Reliability, and Security*, volume 8153 of *LNCS*, pages 130–137. Springer, 2013.

[66] Alexei Iliasov and Alexander Romanovsky. SafeCap Domain Language for Reasoning about Safety and Capacity. In *2012 Workshop on Dependable Transportation Systems/Recent Advances in Software Dependability*, pages 1–10, 2012.

[67] Alexei Iliasov and Alexander Romanovsky. The SafeCap toolset for improving railway capacity while ensuring its safety. 2012.

[68] International Union of Railways (UIC). RailTopoModel - Railway infrastructure topological model, 2016. ISBN 978-2-7461-2513-1.

[69] Aníbal Iung, João Carbonell, Luciano Marchezan, Elder Macedo Rodrigues, Maicon Bernardino, Fabio Paulo Basso, and Bruno Medeiros. Systematic mapping study on Domain-Specific Language development tools. *Empirical Software Engineering*, 25(5):4205–4249, 2020.

[70] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[71] Karim Jahed, Mojtaba Bagherzadeh, and Juergen Dingel. On the benefits of file-level modularity for EMF models. *Software and Systems Modeling*, 20(1):267–286, Feb 2021.

[72] Phillip JAMES. *Designing Domain Specific Languages for Verification and Applications to the Railway Domain*. Theses, Swansea University, January 2014.

[73] Phillip James, Alexander Knapp, Till Mossakowski, and Markus Roggenbach. Designing Domain Specific Languages – A Craftsman's Approach for the Railway Domain Using Casl. In Narciso Martí-Oliet and Miguel Palomino, editors, *Recent Trends in Algebraic Development Techniques*, pages 178–194, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[74] Phillip James, Andrew Lawrence, Markus Roggenbach, and Monika Seisenberger. Towards Safety Analysis of ERTMS/ETCS Level 2 in Real-Time Maude. In Cyrille Artho and Peter Csaba Ölveczky, editors, *Formal Techniques for Safety-Critical Systems*, pages 103–120, Cham, 2016. Springer International Publishing.

[75] Phillip James and M. Roggenbach. Encapsulating Formal Methods within Domain Specific Languages: A Solution for Verifying Railway Scheme Plans. *Mathematics in Computer Science*, 8:11–38, 2014.

[76] Phillip James, Matthew Trumble, Helen Treharne, Markus Roggenbach, and Steve Schneider.

[77] Sebastian Krings and Philipp Körner. Prototyping Games Using Formal Methods. In Antonio Cerone and Markus Roggenbach, editors, *Formal Methods – Fun for Everybody*, pages 124–142, Cham, 2021. Springer International Publishing.

[78] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising Event-B Models with B-Motion Studio. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, pages 202–204, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[79] Lukas Ladenberger and Michael Leuschel. BMotionWeb: A Tool for Rapid Creation of Formal Prototypes. In Rocco De Nicola and Eva Kühn, editors, *Software Engineering and Formal Methods*, pages 403–417, Cham, 2016. Springer International Publishing.

[80] Thierry Lecomte. Applying a Formal Method in Industry: A 15-Year Trajectory. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, pages 26–34, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[81] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated Boundary Testing from Z and B. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *FME 2002:Formal Methods—Getting IT Right*, pages 21–40, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[82] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, pages 855–874, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[83] Michael Leuschel and Michael Butler. ProB: An Automated Analysis Toolset for the B Method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, feb 2008.

[84] Michael Leuschel, Mireille Samia, and Jens Bendisposto. Easy graphical animation and formula visualisation for teaching B. 2008.

[85] Shaoying Liu and Weikai Miao. A formal specification animation method for operation validation. *Journal of Systems and Software*, 178:110948, 2021.

[86] Bjørnar Luteberget, John J. Camilleri, Christian Johansen, and Gerardo Schneider. Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL. In Alessandro Cimatti and Marjan Sirjani, editors, *Software Engineering and Formal Methods*, pages 87–103, Cham, 2017. Springer International Publishing.

[87] Amel Mammar, Marc Frappier, Steve Jeffrey Tueno Fotso, and Régine Laleau. A formal refinement-based analysis of the hybrid ERTMS/ETCS level 3 standard. *International Journal on Software Tools for Technology Transfer*, 22(3):333–347, June 2020.

[88] Atif Mashkoor, Felix Kossak, and Alexander Egyed. Evaluating the suitability of state-based formal methods for industrial deployment. *Software: Practice and Experience*, 48(12):2350–2379, 2018.

[89] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[90] Bart Meyers, Hans Vangheluwe, Joachim Denil, and Rick Salay. A Framework for Temporal Verification Support in Domain-Specific Modelling. *IEEE Transactions on Software Engineering*, 46(4):362–404, 2020.

[91] Peter D. Mosses. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.

[92] Monika Müllerburg, Leszek Holenderski, Olivier Maffeïs, Agathe Merceron, and Matthew Morley. Systematic testing and formal verification to validate reactive programs. *Softw. Qual. J.*, 4(4):287–307, 1995.

[93] Mogens Nielsen, Klaus Havelund, Kim Ritter Wagner, and Chris George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1(1):85–114, Mar 1989.

[94] Liu Ning, Wang Keming, Hou Xili, Simon Xia, Wang, and Cheng Peng. Application Exploration of B Method in the Development of Safety-Critical Control Systems. In *Computers in Railways XVII: Railway Engineering Design and Operation*, page 285–292, Southampton, United Kingdom, 2020. WIT Press.

[95] Leonel Nóbrega, Nuno Jardim Nunes, and Helder Coelho. The Meta Sketch Editor. In Gaëlle Calvary, Costin Pribeanu, Giuseppe Santucci, and Jean Vanderdonckt, editors, *Computer-Aided Design of User Interfaces V*, pages 201–214, Dordrecht, 2007. Springer Netherlands.

[96] Object Managment Group. Unified Modeling Language (UML), v2.5.1, 2017.

[97] Peter Csaba Ölveczky and José Meseguer. The Real-Time Maude Tool. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 332–336, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[98] Jan Peleska. Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. In Alexander K. Petrenko and Holger Schlingloff, editors, Proceedings Eighth Workshop on *Model-Based Testing,* Rome, Italy, 17th March 2013, volume 111 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–28. Open Publishing Association, 2013.

[99] Iyad Rahwan, Manuel Cebrian, Nick Obradovich, Josh Bongard, Jean-François Bonnefon, Cynthia Breazeal, Jacob W. Crandall, Nicholas A. Christakis, Iain D. Couzin, Matthew O. Jackson, Nicholas R. Jennings, Ece Kamar, Isabel M. Kloumann, Hugo Larochelle, David Lazer, Richard McElreath, Alan Mislove, David C. Parkes, Alex 'Sandy' Pentland, Margaret E. Roberts, Azim Shariff, Joshua B. Tenenbaum, and Michael Wellman. Machine behaviour. *Nature*, 568(7753):477–486, 2019.

[100] A. E. Haxthausen RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice Hall Int., 1992.

[101] Silvio Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. 2005.

[102] Francesco Rispoli, Alessandro Neri, and Fabio Senesi. Innovative train control systems based on ERTMS and satellite-public TLC networks. *WIT Transactions on the Built Environment*, 135:51–61, 2014.

[103] José Rivera, Francisco Durán, and Antonio Vallecillo. Formal Specification and Analysis of Domain Specific Models Using Maude. *Simulation*, 85:778–792, 10 2009.

[104] Abhik Roychoudhury. *Embedded Systems and Software Validation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[105] Shubhangi Salunkhe, Randolf Berglehner, and Abdul Rasheeq. Formal Verification of EULYNX Models Using Event-B and RODIN. https://wiki.event-b.org/images/RodinWorkshop2021_Formal_Verification_of_EULYNX_Models_Using_Event-B_and_RODIN_slides.pdf, 2021. Accessed: 2023-08-04.

[106] Stephan Schulz. E – a brainiac theorem prover. *AI Communications*, 15:111–126, 2002.

[107] Thierry Servat. BRAMA: A New Graphic Animation Tool for B Models. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, pages 274–276, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[108] Pengfei SUN. *Model based system engineering for safety of railway critical systems*. Theses, Ecole Centrale de Lille, July 2015.

[109] Andreas Svendsen, Øystein Haugen, and Birger Møller-Pedersen. Synthesizing Software Models: Generating Train Station Models Automatically. In Iulian Ober and Ileana Ober, editors, *SDL 2011: Integrating System and Software Modeling*, pages 38–53, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[110] Andreas Svendsen, Gøran K. Olsen, Jan Endresen, Thomas Moen, Erik Carlson, Kjell-Joar Alme, and Øystein Haugen. The Future of Train Signaling. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, pages 128–142, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[111] U. Tikhonova, M.W. Manders, M.G.J. Brand, van den, S. Andova, and T. Verhoeff. Applying model transformation and Event-B for specifying an industrial DSL. In *Workshop on Model Driven Engineering, Verification and Validation*, CEUR Workshop Proceedings, pages 41–50. CEUR-WS.org, 2013.

[112] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, Inc., USA, 1988.

[113] VITE: Virtualisation of the Test Environment. Lab architecture State of the art analysis, 2017.

[114] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, K. Czarnecki, and B. von Stockfleth. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley Software Patterns Series. Wiley, 2013.

[115] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. A Domain-Specific Language for Generic Interlocking Models and Their Properties. In Alessandro Fantechi, Thierry Lecomte, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 99–115, Cham, 2017. Springer International Publishing.

[116] Linh Hong Vu, Anne Elisabeth Haxthausen, and Jan Peleska. A Domain-Specific Language for Railway Interlocking Systems. In Eckehard Schnieder and Geza Tarnai, editors, *Proceedings of the 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems, FORMS/FORMAT 2014*, pages 200–209. Technische Universität Braunschweig, 2014.

[117] Nathaniel Watson, Steve Reeves, and Paolo Masci. Integrating User Design and Formal Models within PVSio-Web. volume 284, 11 2018.

[118] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topić. Spass Version 2.0. In Andrei Voronkov, editor, *Automated Deduction—CADE-18*, pages 275–279, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[119] Michelle Werth and Michael Leuschel. VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics. In Alexander Raschke, Dominique Méry, and Frank Houdek, editors, *Rigorous State-Based Methods*, pages 260–265, Cham, 2020. Springer International Publishing.

[120] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Comput. Surv.*, 41(4), oct 2009.

[121] Yuchen XIE. *Formal Modeling and Verification of Train Control Systems.* Theses, Ecole Centrale Lille, February 2019.

[122] Asfand Yar, Akram Idani, Yves Ledru, and Simon Collart-Dutilleul. Visual Animation of B Specifications Using Executable DSLs. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '22, page 617–626, New York, NY, USA, 2022. Association for Computing Machinery.

[123] Faiez Zalila, Xavier Crégut, and Marc Pantel. A DSL to Feedback Formal Verification Results. In Michalis Famelis, Daniel Ratiu, and Gehan M. K. Selim, editors, *13th Model-Driven Engineering, Verification and Validation Workshop at MODELS conference 2016 (MoDeVVa 2016)*, volume 1713, pages 30–39, Saint Malo, France, October 2016. CEUR-WS : Workshop proceedings.

[124] Chen Zheng, Samir Assaf, and Benoît Eynard. Towards Modelling and Standardisation Techniques for Railway Infrastructure. In José Ríos, Alain Bernard, Abdelaziz Bouras, and Sebti Foufou, editors, *Product Lifecycle Management and the Industry of the Future*, pages 254–263, Cham, 2017. Springer International Publishing.

[125] Zhi Zhu, Yongling Lei, Qun Li, and Yifan Zhu. Formalizing Model Transformations Within MDE. In Houbing Song and Dingde Jiang, editors, *Simulation Tools and Techniques*, pages 25–42, Cham, 2019. Springer International Publishing.