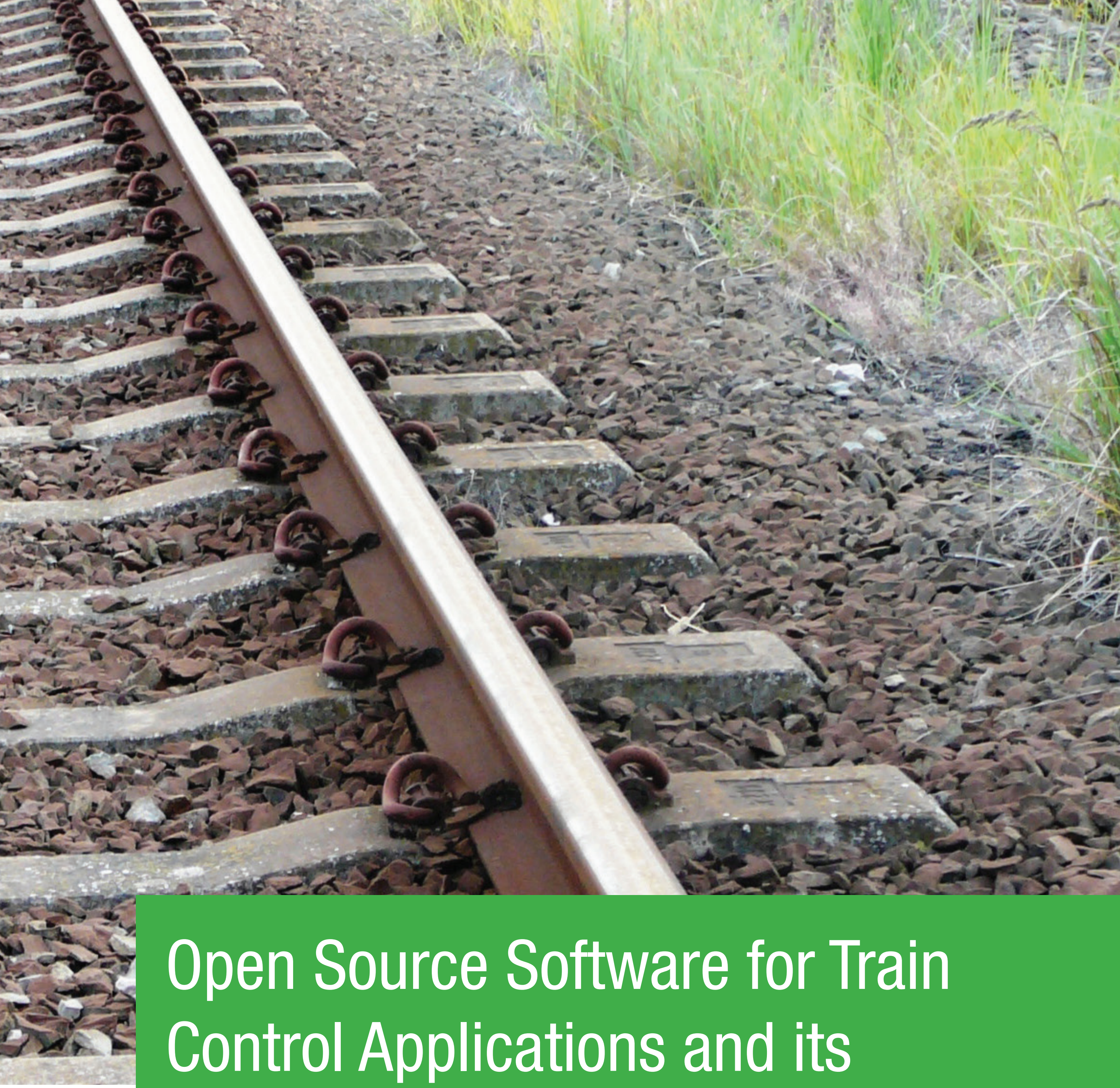


Open Source Software for Train Control Applications and its Architectural Implications

Johannes Feuser

Open Source Software for Train Control Applications and its Architectural Implications

Dissertation

zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften
– Dr.-Ing. –

Vorgelegt im Fachbereich 3 (Mathematik/Informatik)
der Universität Bremen

von

Dipl.-Ing. Johannes Feuser

im Dezember 2012

Datum des Promotionskolloquiums: 21.02.2013

Gutachter: Prof. Dr. Jan Peleska (Universität Bremen)
Prof. Dr. Martin Gogolla (Universität Bremen)

To my father Matthias Feuser

Acknowledgments

I would like to thank the following persons and institutions for their support during the preparation of this work:

> The Siemens AG for supporting me during this work through a research grant of the Graduate School on Embedded Systems GESy[1] at the University of Bremen.
>
> My supervisor Prof. Jan Peleska for his scientific guidance and support.
>
> My friends and former colleagues Erwin Wendland, Lothar Renner, and Dr. Oleg Ivlev, who supported me to form my interest for scientific research during my student years.
>
> My colleague Blagoy Genov for the inspiring, scientific discussions.
>
> My colleagues Dr. Uwe Schulze and Dr. Cecile Braunstein for their support during the finalisation of this document.
>
> My brother Kristof Feuser for creating the cover of this book.
>
> My wife Diana Valbuena for her permanent support and help.

Johannes Feuser
Bremen, 21st May 2014

[1] http://www.informatik.uni-bremen.de/gesy

Zusammenfassung

In dieser Arbeit werden die Forschungsergebnisse der Entwicklung sicherheitskritischer Software mittels den Prinzipien von Open-Source[2]-Software beschrieben. Es wurden verschiedene modellbasierte Entwürfe und Architekturen für das Gebiet des Schienenverkehrs, inklusive wiederverwendbare Formalismen zur Verifikation & Validation, untersucht. Außerdem wurden die Eindämmung von möglichen Sicherheitsbedrohungen[3] durch plattform- oder herstellerspeziefische Anpassung der modellierten und offenen Kern-Software analysiert, und als Lösung der Einsatz von Hardware-Virtualisierung im Gegensatz zu traditionellen Speicherverwaltung entwickelt. Der Hauptteil dieser Arbeit besteht aus der Entwicklung einer Domänen-spezifischer Sprache (DSL), um Teile des europäischen Zug-Kontrollsystems (ETCS) zu modellieren, welche aus dem Spezifikationsdokument speziell abgeleitete Daten-, Kontrollflussformalismen und Sprachelemente verwendet. Das neue GOPPRR Meta-Metamodell wurde als Erweiterung des bereits existierenden GOPRR Meta-Metamodell entwickelt, um den Anforderungen an die Syntax-Definition für die Modellierung für sicherheitskritische Systeme zu genügen. GOPPRR bietet Methoden für die Definitionen von Randbedingungen[4] mittels der Objekt-Randbedingung-Sprache (OCL), sodass eine statische Semantik definiert werden kann, um die Korrektheit von Modellen sicherzustellen. Teile der ETCS Spezifikation für die Kontrolleinheit in Zügen wurden mittels eines neuen Metamodells modelliert. Ein Domänen-Rahmenwerk[5] und ein entsprechender Code-Generator mit integrierter Unterstützung für Verifikation & Validation wurden entworfen und entwickelt, um die Transformation des Modells in eine ausführbare Anwendung zu ermöglichen. Um zu demonstrieren, dass das Modell der Spezifikation korrekt ist, wurde die so generierte Anwendung in einer Simulationsumgebung ausgeführt und entsprechende Simulationsprotokolle erstellt. Die Übereinstimmung dieser Protokolle mit dem erwartetem Verhalten aus dem Spezifikationsdokument bestätigten die verwendeten Methoden und Strategien als mögliches Konzept.

[2] dt. offene Quellen
[3] eng. security threats
[4] eng. constraints
[5] eng. domain framework

Abstract

This document describes the research results that were obtained from the development of safety-critical software under the principles of open source. Different model-based designs and architectures within the railway control system application domain, including re-usable formalisms for verification & validation, were investigated. The reduction of possible security threats caused by platform or supplier specific adaptations of modelled open-core software was analysed, and a possible solution by the usage of hardware virtualisation, instead of traditional memory management, was elaborated. At core of this work, the development of a graphical domain-specific language for modelling parts of the European Train Control System (ETCS) is presented, which is based on specialised data, control flow formalisms, and language elements derived from the specification document. For a more precise and therefore more appropriate syntax definition for safety-critical systems, the already existing GOPRR meta meta model was extended to the newly developed GOPPRR meta meta model. GOPPRR includes methods for defining constraints by the object constraint language, which supports the definition of static semantics to ensure correct model instances. Parts of the ETCS specification related to the train on-board unit were modelled in a new meta model. To transform the developed model of the ETCS specification into an executable application, a domain framework, according to the new meta model and the corresponding code generator, were designed and implemented, which have implicitly an integrated support for the verification & validation process. To proof the correctness of the modelled specification, the resulting application was executed in a simulative environment to obtain simulation traces. The correspondence of traces to the expected data from the specification document supported the used methods and strategies in this dissertation as proof of concept.

Contents

List of Figures

1 Introduction

In this chapter, the objectives of this dissertation are introduced. Furthermore, its main contributions are emphasised and explained in detail. Afterwards, the delimitation from already existing, related work is introduced. This chapter terminates with a summary about the structure of this document.

1.1. Objectives

This dissertation investigates the potential of model-based design and verification and validation of safety-critical control systems in presence of re-usable open source software. Railway control systems are used here as application domain, specifically the European Train Control System (ETCS) [23]. This topic is highly relevant from an industrial and also from a research perspective because manufacturers of complex safety-critical control systems, especially in the avionic and in the railway domains, currently perform a change of strategy regarding the development of software core components.

In the past, this was completely under responsibility of the supplier of the embedded controller. Since system integrators, like German Railways, always use several suppliers for each component in order to avoid single-source situations, this led to similar software components being redundantly developed by different suppliers and resulted in costs that were no longer acceptable. As a consequence, today's strategy is to manage a "pool" of core algorithms for crucial control algorithms to be re-used by every supplier on its proprietary hardware platform. There is already a general consent that this pool should not just consist of software code but instead of system (component) models, so that platform-specific code can be generated following the model-based development paradigm by extending, specializing, instantiating, and transforming these models. This new strategy leads to a number of interesting challenges for research activities in the fields of system modelling, operating systems, and embedded systems verification and are analysed in the context of this dissertation project.

The overall research process of this dissertation is graphically summarised in Figure 1.1, which shows the interconnection of the main research objectives. It should be noted that the order in the research process does not correspond directly to the order of topics or rather chapters in this document, which additionally will be explained in Section 1.4.

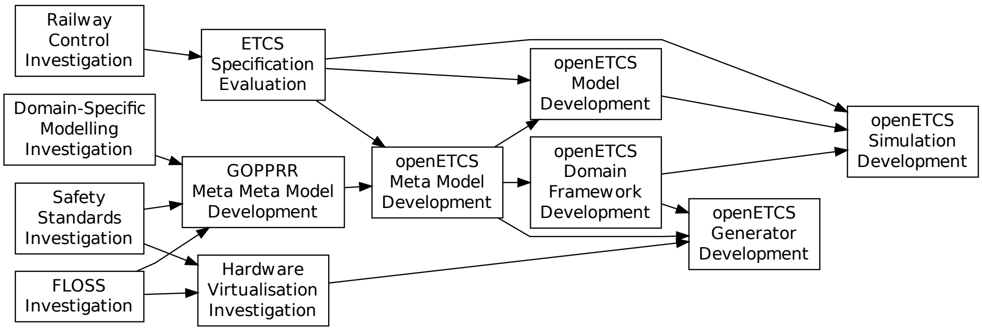

Railway Control Investigation
Domain-Specific Modelling Investigation
Safety Standards Investigation
FLOSS Investigation
ETCS Specification Evaluation
GOPPRR Meta Meta Model Development
Hardware Virtualisation Investigation
openETCS Meta Model Development
openETCS Model Development
openETCS Domain Framework Development
openETCS Generator Development
openETCS Simulation Development

Figure 1.1.: Research process of the main objectives

1.2. Main Contributions

The following new, scientific contributions were gained by this dissertation:

→ the introduction of open source software developed under model-driven aspects with the new denomination *open model software* [26]

→ the usage of *hardware virtualisation* as *security measure* in the railway domain for open source software [26]

→ the *GOPPRR*[1] *meta meta model* as an *extension* of the existing GOPRR meta meta model that is applicable for the *modelling of safety-critical systems* [28]

→ a graphical *domain-specific language* for a subset of the *ETCS specification* as case study including a *formal specification language as meta model*, a *domain framework*, a *code generator*, and a *model of parts of the ETCS specification* [71, 27, 28]

→ the development of a completely *model-driven tool chain* for a case study of the *European Train Control System (ETCS)* for *dependable software* [27]

Each point is discussed in detail in the following paragraphs.

Initially, it was necessary to examine, which prerequisites have to be met in order to develop safety-critical control systems as open source software. Since the usage of Domain-Specific Modelling (DSM) [46] and the extension of such models by supplier-specific implementations implies certain security related problems, those were analysed and discussed. The usage of hardware virtualisation [92] has been proposed as solution and is described in [26].

The selection of an appropriate meta meta model is of certain interest since it builds the basis for the syntax definition of the Domain-Specific Language (DSL) [46]. Although the Graphs, Objects, Properties, Roles, and Relationships (GOPRR) [45, app. A] meta meta model fulfils most of the needs, the Graphs, Objects, Ports, Properties, Roles, and Relationships (GOPPRR)

[1] It must be noted that this acronym was also claimed by MetaCASE – developer of the MetaEdit+ [56] application – during the development of this work to also reflect the presence of ports in their meta meta model. However, the GOPPRR meta meta model extension developed in this thesis must not be understood as a rival product of the GOPPRR meta meta model used in MetaEdit+ but as an extension for it that is required in the context of safety-critical systems.

meta meta model extension was developed with the aim to be able to define a more precise concrete syntax and certain model constraints as static semantics in a general way. Parts of these results will be published in a special issue of Science of Computer Programming [28].

A DSL or rather a meta model based on GOPPRR was developed for ETCS [38], which is an excellent choice for a case study, because it is a publicly available (and usable) train control standard. There also exists an initiative of the German Railways (Deutsche Bahn) to develop an open source solution for ETCS. This initiative led to an European project supported by the Information Technology for European Advancement (ITEA2) board [43] and funded in part by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung). The University of Bremen is a participant within the openETCS consortium.

During the ETCS meta model development, several new formalisms, like data and control flow, were integrated, tested, and adapted to certain specialities of the ETCS specification. Also, new formalisms mainly derived from the specification were developed to support later the transfer of the mostly textual specification document to a model of the meta model. Since the meta model had to be developed in several iterative steps to test certain formalisms or rather syntaxes, an initial version of the DSL was published in a book chapter [71] in [29]. A new version of the DSL will be also included in [28] and the latest results can be found in this document.

According to the instances of a DSL or a Model-Driven Architecture (MDA) [44], a domain framework was developed, which already integrates the results from the initial security analysis in form of hardware virtualisation and also hardware abstraction. Additionally, a code generator was realised as link between model and domain framework, which also provides methods for verification and validation by dynamic testing and model constraint checking. Parts of these methods and strategies are independent from the meta model for ETCS and are therefore completely re-usable for any DSL using the GOPPRR meta meta model. To test the ETCS model or rather the generated code, a virtual simulation environment was set up, which can be used to trace the specification model behaviour. The developed model-driven tool chain of the case study will be published in [27].

The requirements for verification and validation were investigated for applicable safety standards in the railway domain. Thus, verification mechanisms were integrated into the tool chain for testing all static code and the transformation processes for the code generation. A static semantics was defined by a set of model constraints to ensure the validity of concrete models.

1.3. Related Work

Domain-Specific Modelling in the Railway Domain The general applicability of DSM in the railway domain was successfully investigated in [57] and [75]. Especially in [57], the potential of domain-specific modelling for train control applications was investigated, but, in contrast to this thesis, with focus on the control centre perspective. More important, this work adds the aspect of open source software in the railway domain.

MontiCore MontiCore [50, 51] is a framework that is specialised for the development of text-based domain-specific languages [35]. A typical task during the development of these is the definition of the concrete and abstract syntax. In textual languages, the concrete syntax is also called grammar. While the definition of the grammar is mostly related to the development of the available textual constructs and elements, the abstract syntax is needed for parsing concrete text instances that conform to the grammar.

Therefore, both syntax types must be held synchronised or rather must be modified together. Normally, the concrete syntax / grammar is in the main focus during the development of the textual language. Thus, modifications in the grammar render necessary adaptations of the abstract syntax. To avoid this often recurring task, MontiCore provides an own grammar format for the ANTLR [69] parser generator, from which an abstract syntax (tree) can be generated automatically. Accordingly, the manual update and development of the abstract syntax can be omitted.

Although MontiCore seems to be an excellent starting point for the development of textual languages, the case study in Part III developed in this work uses a graphical modelling formalism, which is motivated in Chapter 3.

Open Proofs The development of safety-critical software as re-usable open source is one of the basic ideas that supports this dissertation. A similar concept called Open Proofs [65] also addresses this idea because the term "open proofs" refers to the public proofs of correctness of a certain software. Open Proofs requires

- the entire implementation,
- automatically-verifiable proof(s) of at least one key property, and
- required tools (for use and modification)

to be free/libre open source software (FLOSS) [95]. Open Proofs demands that in such a publicly developed tool-chain faults can be found more easily and eliminated due to a big community using them. Nevertheless, currently there exist only few examples of software that implement the Open Proofs principle.

In contrast to the contributions of this dissertation, Open Proofs is only a concept that can be applied to the software development for safety-critical systems.

openETCS The openETCS [38, 37] initiative of German Railways (Deutsche Bahn) is based on the Open Proofs idea [40, 39]. Reviewing evidence, where security threats have been purposefully integrated into closed-source commercial software products, the initiators argued that open source software could be useful – perhaps even mandatory in the future – to ensure safety and security of railway control systems. Even though the standards applicable for safety-critical systems software development in the railway domain [11, 10] require independent-third-party verification and validation, the complexity of the source code on the one hand and the limited budget available for V&V on the other hand can only mitigate the threat of safety and security vulnerabilities. A guarantee to uncover all compromising code fragments inadvertently or purposefully injected into the code cannot be given. As a consequence, in addition to the V&V efforts required by the standards the broad peer-review enabled by publicly available software could really increase software dependability[2].

As of today, the openETCS approach has stirred considerable interest, in particular among the research communities, which are now invited to act – at least in the V&V branch of the system development process – as equal partners to commercial railway manufacturers. This resulted in a European ITEA2 [43] project initiative, which is currently in progress, but, due to the short project life time, no remarkable results are available.

In contrast to this work, especially to the developed case study, the openETCS project heavily focusses on the tool development to fully realise the Open Proofs concept. The case study was primarily developed to demonstrate that the main contributions of this dissertation can be seen as proof of concept for the applicability of open source software in safety-critical systems in the railway domain. A realisation of a tool chain that completely corresponds to the Open Proofs concept was not a main goal.

AUTOSAR The idea of an open architecture is of course not limited to the railway domain. The Automotive Open System Architecture (AUTOSAR) [5] is an industrial approach that facilitates the interchange of software for control modules used in automotive systems. AUTOSAR does not only include a specification for the software architecture but also for the used development tools. It defines the following goals [5]:

- implementation and standardization of basic system functions as an OEM wide “Standard Core” solution
- scalability to different vehicle and platform variants
- transferability of functions throughout network
- integration of functional modules from multiple suppliers
- consideration of availability and safety requirements
- redundancy activation
- maintainability throughout the whole “Product Life Cycle”
- increase use of “Commercial off the shelf hardware”
- software updates and upgrades over vehicle lifetime

In contrast to the main goal of this dissertation project, AUTOSAR only defines an open architecture, but does not require the developed software or development tools to be open source or even to use a free/libre open source development strategy [95].

TOPCASED The Toolkit in Open Source for Critical Applications & Systems Development (TOPCASED) [81] is a different approach for a standardised development. It is an extension of the Eclipse [20] Integrated Development Environment (IDE), which provides methods and tools for the development of safety-critical software or rather systems for the avionics domain. In contrast to AUTOSAR, TOPCASED does not define a certain system and/or software architecture but the development process by formalisms and tools. Although TOPCASED is, like the Eclipse, IDE published under a FLOSS software license, the developed software is typically not.

[2] Following [53], dependability, in particular, safety, and security are emergent properties that can only be attributed to complete systems and not to software alone.

ERTMS Formal Specs The ERTMS Formal Specs [25] is an application for modelling parts of the ETCS specification [83]. The motivation is similar as for parts of this dissertation project since the transfer of the textual specification to a formal model provides better methods during the system development[3] and supports direct testing on the model for V&V. The ERTMS Formal Specs is developed by the company ERTMS Solutions, which addresses their product to costumers in the area of ETCS hardware / system suppliers.

Since only very few publications exist about the DSL, which only present small parts, a comparison with the in this dissertation developed DSL or rather meta model is hardly possible. It can be only determined that the ERTMS Formal Specs formalisms are strongly aligned to the textual formalisms used in the ETCS specification while parts have a graphical representation. The focus lays mainly on the execution of the model in a simulation environment instead of generating code for an executable binary. At the end of October 2012 an open source version of the ERTMS Formal Specs application was released, but the tool chain and DSL development in this dissertation was already finished at this point. Thus, possible contributions by the ERTMS Formal Specs application could not been taken into account for the case study development.

In contrast to the DSL developed for this dissertation the ERTMS Formal Specs is distributed as one single application and accordingly focuses and the tool development, similar to the openETCS project. Thus, no extendible tool chain is provided, no public meta meta model is employed, and neither a full definition of the concrete and abstract syntax and the static semantics of the meta model is available. Another difference is that the contributed case study is a pure graphical DSL because this approach in general provides the maximal possible abstraction.

In general, the ERTMS Formal Specs is a specialised, commercial software product for ETCS component suppliers, which source code is now published under an open source license. On the other hand, the case study in this work was used to investigate the potential of developing train control applications as open source software and not only distributing them under an open source software license. This takes the complete development process into account and is not especially focussed on the tool development. Accordingly, ETCS was only used as an example for a train control application.

1.4. Structure of this Document

This document is divided in four major parts. Part I provides background information needed for understanding the following parts. It introduces concepts for safe railway operation by means of the European Train Control System. Also, a brief introduction to Domain-Specific Modelling is given. This part concludes with the choice of a meta meta model and modelling application for this work. The last chapter in this part deals with the new developed extension for the selected meta model, which is needed for the integration of safety-critical software.

Part II explains mechanisms of verification and validation for safety-critical systems by using examples of applicable standards and how they might be used for the development of safety-critical open source software. The next chapter in this part elaborates security problems arose by the development of open source software and provides a solution by hardware virtualisation.

[3] by modelling

The new term open model software is defined as software with a model-driven architecture developed as open source software.

A case study for the European Train Control System as Domain-Specific Language for open model software is described in Part III. Following the ideas presented in [38] and [37], this open model software has been labelled openETCS. It should be noted that the name openETCS® was registered at the end of this dissertation project as trademark by German Railways. Since this work is non-commercial research, the term is used in this document as name for the case study and does not refer to a product of German Railways.

The four typical instances meta model, model, domain framework, and code generators are explained in separated chapters. The last chapter of this part discusses how the gained instances are validated in a simulation process. Especially for understanding Chapter 8 about the design and implementation of the domain framework and the generator application, basic knowledge (class, sequence/interaction, and deployment diagrams) of the UML superstructure [64] is required.

The case study is followed by Chapter 12 that concludes about all preceding chapters and discusses about possible further work.

Part IV merges all references like meta model syntaxes or source code for the case study in Part III as appendix.

Requirements, problems, and advantages that are of global interest in this document and not limited to a single chapter are enumerated in a certain style. Requirements are prefixed by a “Req.” followed by a unique, iterated number, advantages have a “Adv.”, and problems a “Prob.” prefix. Therefore, each Req.n, Adv.m and Prob.l exist only once in this document and can be uniquely referred to.

Part I.

Background

2

Concepts for Safe Railway Operation and Control

In general, a railway system can be divided in three components: [67, p. 94]

infrastructure trackwork, signalling equipment, stations, and elements for the electrical power supply[1]

rolling stock cars and locomotives

system of operating rules provides a set of operating rules and procedures to ensure safe[2] railway operation

Infrastructure and rolling stock can be interpreted as the hardware of a railway system while the operating rules and procedures represent the software. Train control systems are typically realised as software. They use the hardware of a railway system to ensure safe railway operation. There exist various kinds of strategies for train control systems. A comprehensive description of typically strategies can be found in [67].

The work presented in this dissertation focuses on Automatic Train Control (ATP) systems, which cause an automatic train stop in the case that any active limits are overpassed. Those limits can refer to the speed of the train or to a certain distance on the track. For ATP systems, data transmission from track-to-train is always needed to inform the train about active limits. Also, the precise determination of the train location on the track is needed for distance limits. More sophisticated ATP systems also use data transmission from train-to-track to inform a control centre about the train status. There exits two types of ATP systems:

- intermittent
- continuous

[1] not always available

[2] safe relates here to the avoidance of any possible incidences that could harm any human or the environment.

Intermittent ATP: In intermittent ATP systems, the track-to-train communication is only established on certain points. Possible safety-functions of an intermittent ATP [67, p. 95] are:

automatic warning warning on approach to a stop-point or on approach to other active limits

braking curve supervision guarantees that a stop-point is not overpassed at all or only at a certain distance

train stop ensures that the train stops after an overpassed stop signal

An example for an intermittent ATP is the German punktförmige Zugbeeinflussung (PZB) [68, pp. 71-77], which is also used in several other countries. The disadvantage of such system is that new limits can be only submitted to the train at certain points of the track. For example, a train-stop can not be triggered between two submission points. The benefits are the low investment and maintenance costs on the infrastructure side.

Continuous ATP: In continuous ATP systems, the train-to-track and track-to-train communication is permanently established. This may invoke cable-loop-devices on the track or radio. Discrete devices may be additionally used for continuous systems. Of course, a continuous ATP system can provide the same safety-functions as an intermittent one. Additionally, speed profiles [67, p. 99ff] can be generated to automatically guide a train in respect to

current speed limit active limit that may not be exceeded

target speed may not be exceeded at the target distance

target distance distance to new speed limit

An example of a continuous ATP is the German linienförmige Zugbeeinflussung (LZB) [68, pp. 77-84], which is the successor of PZB. Compared with intermittent ATP systems, continuous systems need more investment costs for the infrastructure but can provide more efficient safety-functions.

2.1. Requirements for a Train Control System as Open Source Software

The implementation of a train control system as Open Source Software (OSS) [94] or even as Free/Libre Open Source Software (FLOSS) implies that the software is published[3] and is accessible for a wide mass of software developers. Train control systems are normally defined by a set of documents called specification. A result of implementing a train control system by following its specification is that the relevant[4] parts of it can be found in the source code as a set of statements in a certain programming language. In other words, the specification is

[3] typically over the Internet

[4] also called: normative

indirectly published via the source code. This is a problem if the specification is not licensed in the manner of OSS or FLOSS, which means generally public accessible and usable. Therefore, it is only meaningful to develop OSS or FLOSS for train control systems which specification is adequately licensed. This circumstance directly influenced the choice of train control system used for the case study in Part III. Further issues related to OSS and FLOSS software for safety-critical systems are discussed in Chapter 6.

2.2. European Train Control System

The European Train Control System (ETCS) [67, pp. 102-105] is a component of the standardised European train traffic control system, which should replace the several different used train safety systems in European countries. The initiative for creating a general train control system for the European Union was already launched in 1990. Meanwhile, there exist several revisions of the specification of ETCS, which are now published and maintained by the European Railways Agency (ERA) [21]. The actual version of the ETCS specification can be found in [23]. The specification itself is divided in so-called Subsets, which combine a set of documents for a certain specification issue. The following description of ETCS mainly relates to the Subset-026 [83], which is also called System Requirement Specification (SRS). It must be emphasised that this work refers to the specification version 2.3.0, which was mandatory at the beginning of this work in 2010.

The SRS 2.3.0 consists of seven documents:

Introduction	general introduction to ETCS and the documents in the SRS [86]
System Description	description of an ETCS [84]
Principles	principles for ETCS [89]
Modes and Transition	description of possible modes and transition in ETCS [88]
Procedures	operation procedures [90]
ERTMS/ETCS Language	structures for data transmission [85]
Messages	data structures for radio communication in ETCS [87]

This SRS document often refers to another specification element outside the Subset-026, the Functional Requirement Specification (FRS) [22]. The FRS defines the functionality that is available in ETCS. It is more abstract because it only describes required functions, but not how those are realised. In the development cycle of the ETCS specification, the FRS is used as input for the SRS, in which the functional requirements are transformed to requirements of the whole system. Therefore, the FRS is never used directly for system development.

2.2.1. General Purpose

The ETCS introduction defines some general goals and advantages [86, p. 4]:

- cross border interoperability
- improvement of the safety of national and international train traffic
- improvement of international passengers and freight train traffic management
- shorter headway on heavily trafficked lines by driving on moving block, enabling exploitation of maximum track capacity
- the possibility of step-by-step introduction of the new technology
- enabling Pan-European competition between the manufacturers of ETCS components, strengthening the position of the European railway industry on the world market
- enabling preconditions for future harmonisation in other areas of rail traffic management

Not all of these advantages are technical, but they are also political and economical. While this work only focuses on technical aspects, also those advantages must be seen more critical. The improvement in certain countries heavily depends on the existing ATP systems, which have to be discussed in detail to decide about gained benefits by ETCS.

ETCS monitors, as most ATP systems, certain parameters:

- local maximum speed
- maximum speed of the train
- correctness of train's tracks
- train's movement direction
- suitability of train for current track
- compliance with special operating instructions

For this task several hardware components in the infrastructure and rolling stock are used [84, pp. 6-8]:

Eurobalise Local limited data transfer facilities in the rails. It is differentiated between Eurobalises that transfer fixed data and those that transfer dynamic data. The functionality is similar to a transponder because the data is only transferred when a train passes an Eurobalise.

Euroloop Cable-based semi-continuous data transfer facility, which is located in track bed.

Radio-Infill Radio based semi-continuous data transfer facility, which uses GSM-R.

Euroradio Standard for data transfer encryption, which is used for GSM-R.

STM The Specific Transmission Module is a component that can connect the ETCS on-board device with national systems. Each National system needs a certain STM.

ETCS vehicle on-board devices Devices on the train are mainly an European Vital Computer (EVC), a Driver Machine Interface (DMI), a distance measurement device[5], a GSM-R transmitting device / Euroradio, a reader for Eurobalises, and an access device to the brake systems. STMs are optional.

Most of those already exist and do not have to be developed especially for ETCS. Therefore, the development primary focuses on the software-side of railway systems. Figure 2.1 shows the (simplified) architecture [84, p. 8] of ETCS for track and train. The focus is on the communication and the data flow between the components of track and train. The integration of a control centre is omitted for simplification.

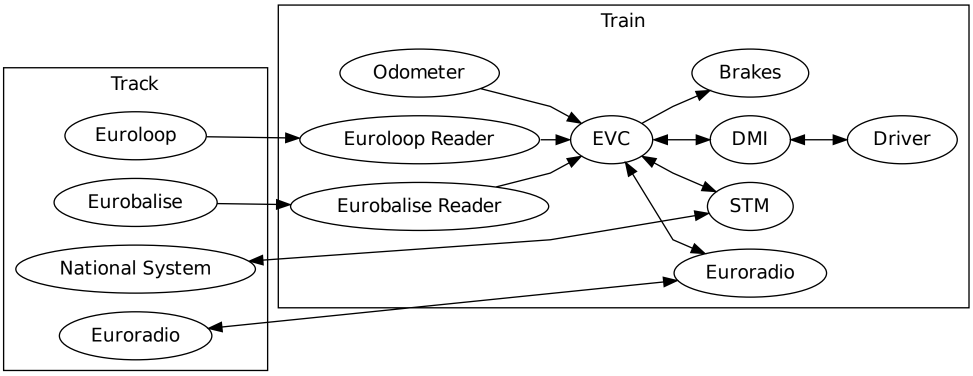


Figure 2.1.: ETCS architecture of track and train (simplified)

The following sections introduce some central parts for the train operation under ETCS.

2.2.2. Application Levels

The ETCS Application Levels were introduced to distinguish between different safety requirements. ETCS Level 0 is the lowest and 3 the highest. The ETCS Application Level describes, which components from Subsection 2.2.1 are available / used in the rail track and in the train. Trains are downwards compatible but tracks are not [84].

2.2.2.1. Level 0

In this Level, no ETCS track-side components are available. The train conductor only uses the conventional signal and signs [67]. The ETCS on-board equipment only supervises the static maximum speed of the train and the track [84, pp. 11-12].

[5] also called: odometer

2.2.2.2. Level 1

In ETCS Application Level 1, Eurobalises or other distance discrete transmission devices (see Subsection 2.2.1) are used for transmitting the local maximum speed for the track, the track gradient, and the next stop point. The usage of Euroloop devices is optional. The provided functionality in this Level is the control of the local maximum speed and the movement authority. Information about track clearance is acquired by conventional methods / devices, like axis counters [67]. The train is unknown For the track-side equipment [84, pp. 14-17].

2.2.2.3. Level 2

In Level 2, all information is mainly transmitted via GSM-R / Euroradio. Eurobalises are only used for transmitting the current position to the train on-board equipment. Level 2 provides an overlay functionality for an underlying signalling system. Train detection and train integrity are monitored by the track-side non-ETCS equipment. Conventional signals on the track are optional. The provided functionality is supervision of speed, distance, and movement authority. The track-side radio centre on each train can be identified individually [84, pp. 18-20].

2.2.2.4. Level 3

In ETCS Level 3, compared to Level 2 , the train submits additionally its position (acquired by Eurobalises) to the control centre by GSM-R or Euroradio. Furthermore, the train integrity information is sent. This means that the train integrity and position are monitored by train and track-side equipment in cooperation. Conventional signals are not used in Level 3. As in Level 2, the provided functionality is supervision of speed, distance, and movement authority. The track-side radio centre on each train can be identified individually [84, pp. 20-22].

2.2.2.5. Specific Transmission Module

In the Specific Transmission Module (STM) Level, all track to train information are handled by national systems. Also, all on-board functions are provided by national systems in cooperation with ETCS. Additionally, only Eurobalises are used to detect possible or required transitions to other ETCS Application Levels. Train detection and integrity supervision are performed by external equipment. The provided ETCS functionality depends on the implementation of the STM and the national system. This level is used for compatibility reasons for already existing national systems [84, pp. 13-14].

2.2.3. Operational States

In contrast to the different Levels for tracks and trains in Subsection 2.2.2, the ETCS Modes refer to the state of the EVC (Subsection 2.2.1) on board the train. Those modes are roughly introduced in the following. It must be noted that not every ETCS Mode is available in each ETCS Application Level [88]. Hence, each mode description is followed by a small table, in which all possible Levels are marked grey.

2.2.3.1. Full Supervision

In the Full Supervision (FS) [88, pp. 17-18] Mode, the speed and position of the train are supervised by ETCS. This Mode should automatically be used if all necessary train and track data are available.

0	**1** ▨	**2** ▨	**3** ▨	**STM**

2.2.3.2. On Sight

In the On Sight (OS) [88, pp. 22-23] Mode, the train is monitored by ETCS, but the train conductor drives on sight, for example, in an occupied track. In this Mode, the train speed is supervised against a dynamic speed profile submitted from the track-side equipment.

0	**1** ▨	**2** ▨	**3** ▨	**STM**

2.2.3.3. Staff Responsible

In the Staff Responsible (SR) [88, pp. 19-23] Mode, the train conductor is alone in charge of the train while driving on an track equipped for ETCS. This mode is normally used after starting the on-board equipment from Sleeping (Subsection 2.2.3.6) to pass a stop-signal at danger or after a failure of the track-side ETCS equipment. In this Mode, the train's static top speed and its position are supervised by Eurobalises.

0	**1** ▨	**2** ▨	**3** ▨	**STM**

2.2.3.4. Shunting

The Shunting (SH) [88, pp. 15-16] Mode is used for shunting[6] operations. In this Mode, the ETCS on-board equipment supervises the shunting top speed, which is normally a national value, and the train's position by Eurobalises. The Shunting mode does not require any train data because the train is only partly supervised.

0 ▨	**1** ▨	**2** ▨	**3** ▨	**STM**

2.2.3.5. Unfitted

In the Unfitted (UN) [88, p. 18] Mode, only the train static top speed is supervised by ETCS. This Mode is used for tracks that are not equipped for ETCS or only with a national track-side system and no corresponding STM is available.

0 ▨	**1**	**2**	**3**	**STM**

2.2.3.6. Sleeping

The Sleeping (SL) [88, pp. 12-13] Mode should be used for engines, which are used as slaves and are remote controlled. In this Mode, no movement supervision is done, only the acquisition of the train position. If the connection to the remote engine is lost, it should be switched to Stand By (Subsection 2.2.3.7) after the train has stopped.

0 ▨	**1** ▨	**2** ▨	**3** ▨	**STM** ▨

[6] American English: switching

2.2.3.7. Stand By

The Stand By (SB) [88, p. 14] Mode is the default mode after starting the ETCS on-board equipment. In this mode, a self-test and a test of all external devices is executed. ETCS supervises that the train does not move at all.

0	**1**	**2**	**3**	**STM**

2.2.3.8. Trip

In the Trip (TR) [88, p. 23] Mode, the emergency brake is active until the train completely stops and the conductor acknowledges the Trip. This changes the ETCS operational state to Post Trip (Subsection 2.2.3.9).

0	**1**	**2**	**3**	**STM**

2.2.3.9. Post Trip

The Post Trip (PT) [88, pp. 24-25] Mode is entered, when the train conductor acknowledges a Trip after an emergency stop, which then releases the breaks. In Post Trip, it is possible that the train drives a certain distance (defined by a national value) backwards. Only this distance is supervised in this mode by ETCS.

0	**1**	**2**	**3**	**STM**

2.2.3.10. System Failure

In case of an error in the ETCS on-board equipment that leads to an failure of the system the on-board equipment should switch to System Failure (SF) [88, p. 11]. In this Mode, the emergency braking should be permanently active.

0	**1**	**2**	**3**	**STM**

2.2.3.11. Isolation

The Isolation (IS) [88, p. 9] Mode should be used if the EVC is isolated from the other on-board equipment (including the driver) and is also physically isolated from the brake systems. In this Mode, the ETCS on-board equipment does not have any responsibility.

0	**1**	**2**	**3**	**STM**

2.2.3.12. No Power

The No Power (NP) [88, p. 10] Mode should be used if the ETCS on-board equipment is switched off.

0	**1**	**2**	**3**	**STM**

2.2.3.13. Non Leading

The Non-Leading (NL) [88, p. 26] Mode is used for slave engines, which are not electrically coupled and are therefore not remote controlled by the master, as it is the case for the Sleeping mode (Subsection 2.2.3.6). This situation is also called Tandem [22]. In this Mode, only the trains position is acquired.

0	**1**	**2**	**3**	**STM**

2.2.3.14. STM European

The STM European (SE) [88, pp. 27-28] Mode should enable the usage of supervision functionality of national systems by ETCS. Also, the access of the STM to on-board equipment via ETCS should be possible. ETCS supervises the train movement with a profile given by the STM.

0	**1**	**2**	**3**	**STM**

2.2.3.15. STM National

In the STM National (SN) [88, p. 29] Mode, the STM has access to the train on-board equipment via ETCS. The STM is responsible for all supervisions including the interaction with the driver.

0	**1**	**2**	**3**	**STM**

2.2.3.16. Reversing

In the Reversing (RV) [88, pp. 30-31] Mode, a train can drive backwards. The speed and position of the train is supervised by ETCS.

0	**1**	**2**	**3**	**STM**

2.2.4. Mode Transitions

There are various possible transitions between Modes under different conditions [88, pp. 36-40], which are, due to their extensiveness, not explicitly listed here. [88, pp. 37] provides a good overview in form of a transition table.

2.3. openETCS

openETCS [38, 37] describes the idea of the approach to implement the ETCS functionality on the basis of open source software (OSS) [94].

Due to the advantages, openETCS should not only be implemented by OSS but by Free/Libre Open-source Software (FLOSS). This means that the openETCS software should not only be distributed with open source code but also under licenses that allow users to use, study, modify, and (re-)distribute the source code. The main advantages compared to conventional closed source software methods are listed below:

reduction of licenses costs No external / commercial license fees are necessary because other FLOSS can be used for openETCS.

reduction of errors First experiences with ETCS software already showed that even with common testing methods complex software never can be assumed to be error-free [38, 37]. With FLOSS, it is possible to have a big community (of experts) reviewing and correcting the source code. Also, this includes the elimination of "Backdoors".

support of European Union The European Union does not only recommend the usage of FLOSS but also to make use of the European Union Public License (EUPL) [24], which is a license for FLOSS. Because openETCS is a project within the European Union, it seems obvious to use FLOSS under the EUPL for its realisation.

ETCS refers to a whole control system with hardware and software elements while openETCS only describes (parts) of the software of this system. The openETCS software is (mainly) located on trains on board unit, the EVC. This is connected with several hardware components on the train (see Subsection 2.2.1 and Figure 2.1), which are used by the openETCS software. To avoid dependencies from certain and proprietary hardware components in the software, a hardware abstraction layer must exist between openETCS and the hardware device drivers on the EVC. Therefore, openETCS is more generally usable. Figure 2.2 shows how openETCS integrates in the EVC with a hardware abstraction interacting with all other train on-board equipment from Figure 2.1.

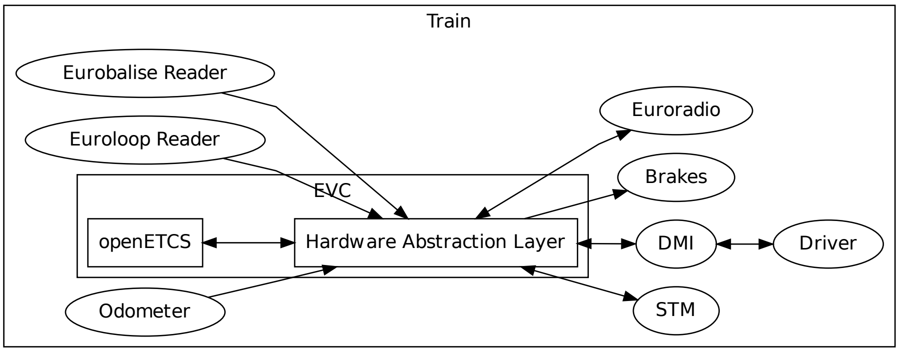


Figure 2.2.: openETCS system integration

2.4. Conclusion

ETCS or rather openETCS is used as case study in this work because, first of all, it fulfils the requirements defined in Section 2.1. Second, it has a public available specification, which also could be published as formal language or model. Additionally, it provides the focus on the software-side of the ATP system, which is of main interest in this work. The openETCS initiative demonstrates that the scope of this work especially with ETCS is of high research interest.

3
Domain-Specific Modelling

Domain-Specific Modelling (DSM) [46] is a modern software engineering method that uses models, instead of pure source code, to develop software, as applied in several currently used development strategies. The key difference is that it does not use a general modelling language to describe each certain problem but uses for each certain problem domain a certain Domain-Specific Language (DSL) [46]. This dramatically increases the level of abstraction for software development while the complexity is reduced [46].

In contrast to currently available CASE-Tools for UML [64], a DSL can provide the generation of all source code. With UML, this is not possible since UML is a very general language and does not provide much abstraction from source code. Although UML provides a specialisation mechanism with profiles [64], it can be shown that those cannot provide the same level of abstraction and effectiveness for code generation as when using a DSL [57]. In general, a DSM architecture consists of four elements as shown in Figure 3.1.

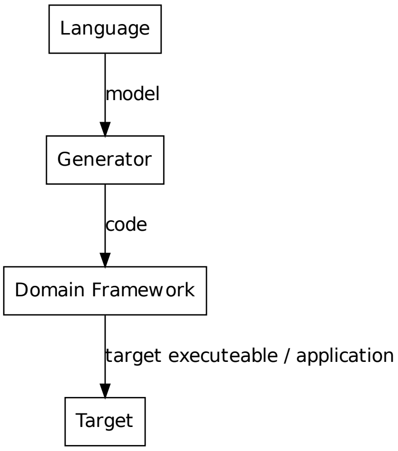


Figure 3.1.: Elements of a DSM architecture

Language or rather the DSL is a modelling formalism specialised for a certain problem domain, which provides a high level of abstraction in relation to a programming language / target. The DSL is developed by a domain expert.

Generator specifies how the parts of a concrete model are transformed to source code fitting to the target. The generator is normally developed by a programming expert to provide a high quality of the generated code.

Domain Framework provides an interface between the generated code and the target platform / hardware. Typically, it consists of all static source code that does not have to be generated from the model. Often, a domain framework provides encapsulation of platform or hardware depended operation to enable platform / hardware independence. The domain framework is also developed by a programming expert, but itself does not necessarily need to be developed under model-driven aspects.

Target is the generated code typically in a certain programming language. Normally, the target code is not executed or rather compiled alone but together with the domain framework and additional (static) source code.

A DSM architecture does not have to necessarily consist of all four elements. For example, for an architecture that generates (Java code and then) Java bytecode, no domain framework would be necessary since Java bytecode is (mostly) platform independent. The corresponding DSM architecture elements are shown in Figure 3.2.

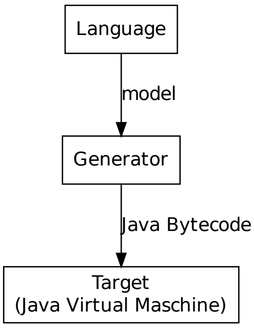


Figure 3.2.: Example of a DSM architecture for Java virtual machine as target

Besides the definition and development of a DSM architecture, its usage is also an integral part of the development process. While DSM architecture elements are developed by domain and programming experts, the DSM architecture and especially the DSL is used by application developers who neither have to be programming nor domain experts. The correspondents between DSM definition and their usage is graphically shown in Figure 3.3 [46, p. 67].

The following section introduces some prominent, public meta meta models. Afterwards, these are compared in combination with the corresponding modelling applications to select an appropriate meta meta model for the case study in Part III. Readers familiar with the meta

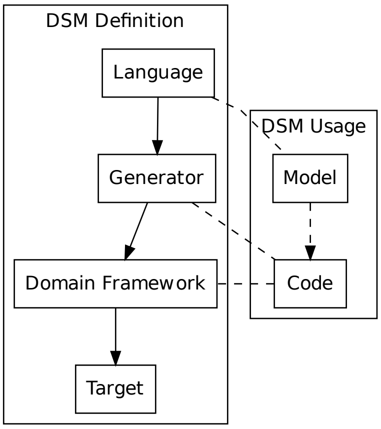


Figure 3.3.: DSM definition and usage

meta models in the first section of this chapter, may skip the section or the corresponding subsections.

3.1. Meta Meta Models

Normally, a DSL is defined by a meta model [76], which specifies the elements the DSL consist of and how they can be connected with each other. Analogue to the modelling of software in a DSL, a meta model itself is modelled in a meta meta model [76]. Therefore, a meta model can be interpreted as an instance of a meta meta model. Furthermore, a DSM architecture consists of two more instances that are the model and the application. Figure 3.4 shows all four instances and their relations.

To illustrate the meaning of the instances, Figure 3.5 shows examples for instances of a programming language and of a graphical user interface (GUI) as DSM architecture. Typically, an existing meta meta model is chosen and not developed. For an object-oriented programming language, a class concept could be defined as meta class. In C++, this would correspond to the `class` [79] statement. The declaration of a concrete class would be the next instance and an object of this class the final one. The same could be done for developing a graphical user interface (GUI): Available GUI elements are defined as so-called widgets, which could be combined to a dialogue, from which the corresponding application is generated.

As the preceding examples and figures show, the meta meta model is the starting point for every DSM architecture. Because of the big amount of available meta meta models, only the most significant are exemplary mentioned and explained in the following subsections.

3.1.1. Meta Object Facility

The Meta Object Facility (MOF) [76, pp. 64-71] was specified by the Object Management Group (OMG) by using UML. Since UML is an already well established formalism, no more

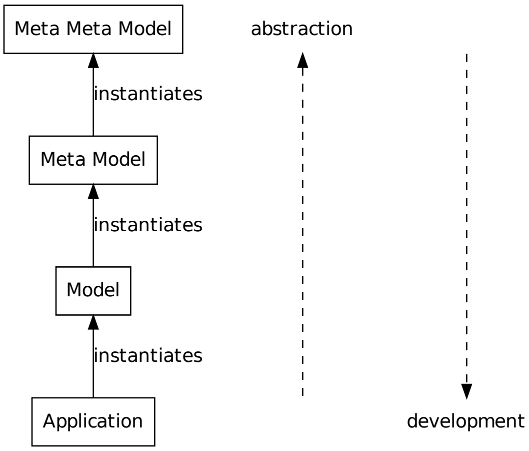


Figure 3.4.: DSM architecture instances

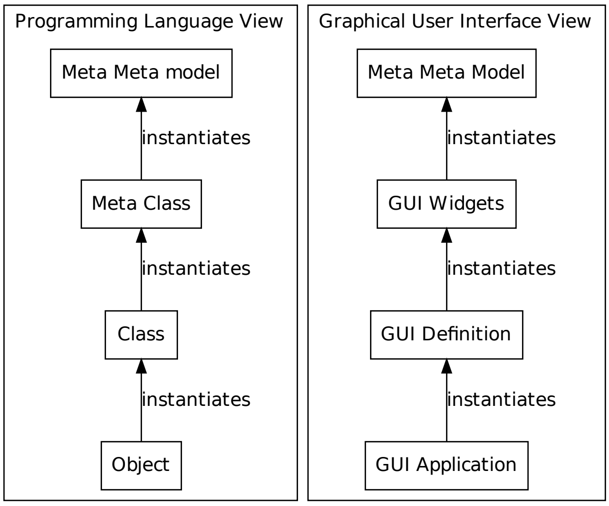


Figure 3.5.: DSM architecture instances examples

additional syntax for the meta meta model has to be investigated for understanding it. In other words, UML is the meta model and MOF the meta meta model. This grants a high duality between software design and meta model.

MOF can be generally divided in EMOF (Essential MOF) [60], which specifies the essential elements of MOF, and CMOF (Complete MOF) [60], which holds all available MOF elements. Due to the complexity of CMOF, only EMOF is explained here, beginning with all general EMOF types in Figure 3.6.

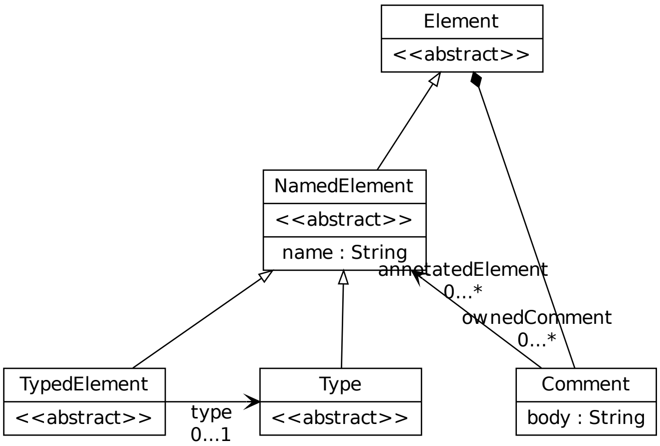


Figure 3.6.: EMOF types

Element is the most general and abstract base class for any element in MOF. It holds a set of comments (`ownedComment`).

NamedElement is the abstract base class for all elements with an additional, internal name. Hence, it has the attribute `name`.

Type is the abstract super class for all classes and data types.

TypedElement is the abstract base class for all typified elements.

Comment is used for commenting any element. Thus, it provides the attribute `body`, in which the literal comment is stored. It has an aggregation to its possessing element (`element`).

All classes marked with `<<abstract>>` are abstract classes [79], which generally means they cannot be instantiated directly, only by generalisation / inheritance [62, 79].

Furthermore, EMOF provides the possibility of modelling data types. Figure 3.7 introduces the relations between the built-in data types. New elements are:

DataType is the general class for all data types.

PrimitiveType is a class for all primitive data types.

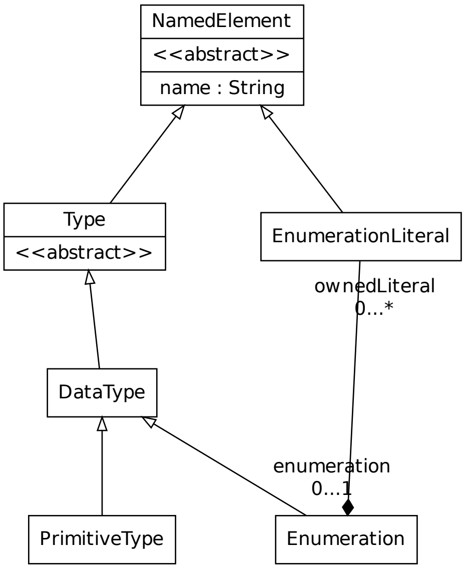


Figure 3.7.: EMOF data types

EnumerationLiteral is a class for all enumeration literals. It holds an aggregation to its possessing enumeration (`enumeration`).

Enumeration is a class for enumeration types while the literals are described by a set of enumeration literals (`ownedLiteral`).

To describe complex DSLs with EMOF, it is also necessary to model classes. This is shown in Figure 3.8. The new elements are described below:

MultiplicityElement is the abstract base type for all elements that can be multiplicities.

Property is a class for all properties / attributes of a class. It has a self association for a possible opposite type (`opposite`) and an aggregation [64] to its possessing class (`class`).

Operation describes a certain operation / method of a class. It holds an aggregation to its possessing class (`class`) and to a possible set of parameters (`ownedParameter`). It also has an association with types (`raisedException`), which specifies the possible kinds of thrown / raised exceptions.

Parameter is a class for all parameters of an operation. It has an aggregation to its possessing operation (`operation`).

Class describes a general class type. It holds a set of attributes (`ownedAttribute`) and a set of operations (`ownedOperation`).

Since the UML infrastructure should be aligned to the MOF specification [64, p. 8], Figure 3.9 shows the related DSM instances introduced in Section 3.1 for the UML specification.

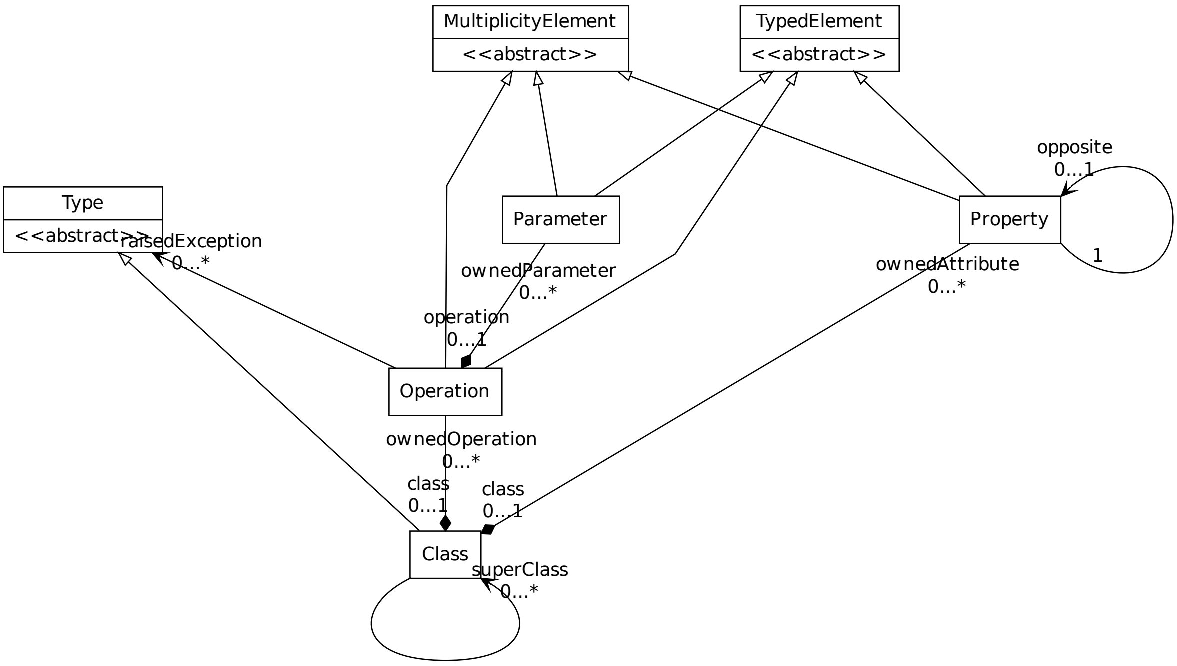


Figure 3.8.: EMOF classes

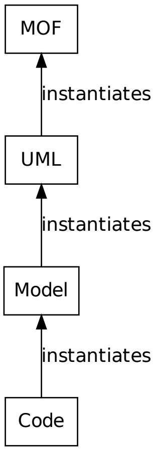


Figure 3.9.: MOF DSM instances

3.1.2. Ecore

Ecore [76, pp. 71-73] is very similar to MOF and uses the Eclipse Modelling Framework (EMF) [76, pp. 71-73]. EMF is a software framework based on Java, which provides functionality for model-driven software development. Furthermore, it provides mechanisms for creating new DSLs and for generating code. Its main disadvantage is are the code generation capabilities, which are limited to Java source code using the EMF.

Ecore is not explained in this document in detail because its similarity to MOF, which was already introduced in Subsection 3.1.1.

3.1.3. Extensible Markup Language with Schema Definition

The Extensible Markup Language [96] (XML) with XML Schema Definition (XSD) [42, 70] is very easy and simple to use for DSM because XSD can be directly used as meta meta model while the definition in XML describes corresponding meta models. A XML file complying to a certain Schema Definition is then a model of the meta model [76, p. 74].

An example for an XML file or rather model could be the simple description of a software class, as shown in Figure 3.10. This example can be taken to explain the most important

```
 1 <class name="CSomeClass">
 2   <implementation language="C++" />
 3   <method name="SomeMethod" visibility="public">
 4     <return type="void">
 5     <parameter name="iValue" type="int" direction="in" />
 6     <definition>
 7     // do something meaningful
 8     iValue++;
 9     </definition>
10   </method>
11 </class>
```

Figure 3.10.: XML class description example

elements of XML:

Markup All elements starting with < and ending with > are markups. In the example, all lines are markups except 7 and 8.

Content Every element that is not markup is content. In the example, the only content are the lines 7 and 8.

Tag A tag is a markup construct. There are three different tag types:

- start-tag (e.g. `<class name="CSomeClass">` in line 1)
- end-tag (e.g. `</class>` in line 11)
- empty-element-tag (e.g. `<implementation language="C++" />` in line 2)

Element An element is a logical component either encapsulated by a corresponding start-tag and end-tag or an empty-element-tag. Line 1 to 11 or 3 to 10 are elements in the example but also line 2.

Attribute An attribute is also a markup construct that consists of a name-value pair in a start-tag or an empty-element-tag. For example, `language="C++"` in line 2 or `name="SomeMethod"` and `visibility="public"` in line 3 are attributes.

The simple example in Figure 3.10 demonstrates how XML could be used to model / describe software classes or other kinds of models. As already mentioned, XML can be used as meta model and XSD as its corresponding meta meta model. XSD can be simplified and interpreted as a grammar definition for XML while XSD uses the same syntax as XML. Also, MOF and UML (Subsection 3.1.1) share this advantage.

Figure 3.11 shows an example of an XML Schema Definition, which defines a software class description formalism corresponding to the preceding XML example in Figure 3.10. Generally,

```
<xs:element name="class">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" />
        <xs:sequence>
          <xs:element name="implementation">
            <xs:complexType>
              <xs:attribute name="language" type="xs:string" />
            </xs:complexType>
          </xs:element>
          <xs:element name="method">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="return">
                  <xs:complexType>
                    <xs:attribute name="type" type="xs:string" />
                  </xs:complexType>
                </xs:element>
                <xs:element name="parameter">
                  <xs:complexType>
                    <xs:attribute name="type" type="xs:string" />
                    <xs:attribute name="direction" type="xs:string" />
                  </xs:complexType>
                </xs:element>
                <xs:element name="definition" type="xs:string" />
              </xs:sequence>
            </xs:complexType>
          </xs::element>
      </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 3.11.: XSD class definition example

all XSD elements are in the name space xs and must be prefixed by `xs:`. The first statement in line 1 starts with the definition of an XML element named class, which corresponds to line 1 of the XML example in Figure 3.10. Line 2 defines a class as a complex type, which means it may consist of more elements and may have attributes. Line 3 declares that the element class has an attribute name, which is a string (see line 1 in Figure 3.10). Lines 3 to 27 are a sequence of elements in the declared class, which also can be complex and holds more elements or attributes and so on. Of course, the example only demonstrates a small set of the possibilities in XSD.

Figure 3.12 shows how the DSM architecture instances are generally related for XML and XSD.

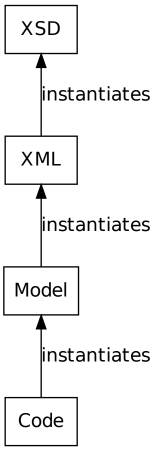


Figure 3.12.: XML/XSD DSM instances

3.1.4. Backus-Naur Form

The Backus-Naur Form (BNF) [48] is a meta syntax to describe context-free textual grammars. BNFs are widely spread for the definition of programming language grammars, but it can be also used to define a DSLs.

Generally, a BNF consists of a set of rules with the following syntax:

```
<symbol> ::= __expression__
```

A `<symbol>` is called a non-terminal and `__expression__` consists of one or more sequences of symbols. Several sequences can be separated by a `|` indicating that each sequence can be chosen for definition. Symbols that never appear on the left side of a rule are called terminals.

A BNF must be interpreted in a recursive way: Each non-terminal in a rule on the right side of `::=` must be replaced by its definition while choosing one of by `|` separated sequences, until the right side of a rule only consists of terminals. Figure 3.13 shows the BNF syntax as BNF. `<rule-name>` and `<text>` are here the only terminals and should be replaced by the rules literal name or literal text.

```
<syntax>          ::= <rule> | <rule> <syntax>
<rule>            ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace>
    "::=" <opt-whitespace> <expression> <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""
<expression>      ::= <list> | <list> "|" <expression>
<line-end>        ::= <opt-whitespace> "\n" | <line-end> <line-end>
<list>            ::= <term> | <term> <opt-whitespace> <list>
<term>            ::= <literal> | "<" <rule-name> ">"
<literal>         ::= '"' <text> '"' | "'" <text> "'"
```

Figure 3.13.: BNF syntax defined as BNF

To define a new DSL, a corresponding syntax has to be defined by a BNF and certain models in the new DSL have to comply to the grammar. Figure 3.14 shows how the BNF instances integrate in a DSM architecture.

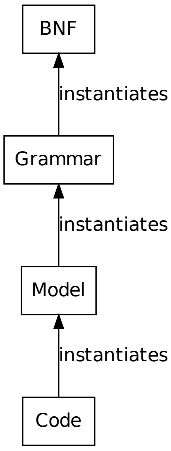


Figure 3.14.: BNF DSM instances

The Extended Backus-Naur Form (EBNF) is a small extension for the BNF that mainly adds possibilities to define optional or repetitive elements. Those can only be defined in a simple BNF by recursion, which causes more complex BNF statements.

3.1.5. Graph, Object, Property, Role, and Relationship

Graph, Object, Property, Role, and Relationship (GOPRR) [45, app. A] consists mainly of the elements its name mentions. It was designed for the easy and fast development of graphical meta models and models.

Generally, all elements in GOPRR are divided in two groups: Properties and non-properties while all non-properties can have properties. Figure 3.15 shows the relation of all elements. The possible types of properties do not have to be limited in GOPRR. Nevertheless, in practical

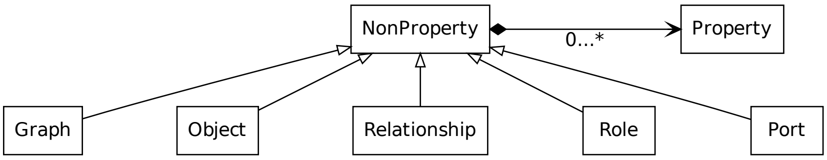


Figure 3.15.: roperties and non-properties in GOPRR

implementations, the set is often limited to strings, numbers, Boolean values, arrays, and non-properties.

Since GOPRR was designed to develop graphical meta models, a graph instance typically contains a set of the other non-property elements, as shown in Figure 3.16.

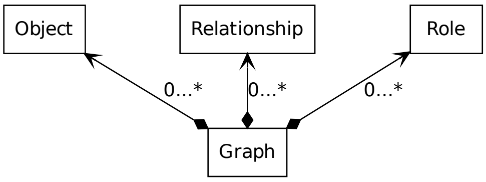


Figure 3.16.: GOPRR graph elements

The complete abstract meta meta model syntax [46, pp. 68-69] is far more complex and is introduced in Figure 3.17 and Figure 3.18 as an overview. The diagrams were assembled from parts of [45], [46], and [56]. The included elements are explained in detail in the following:

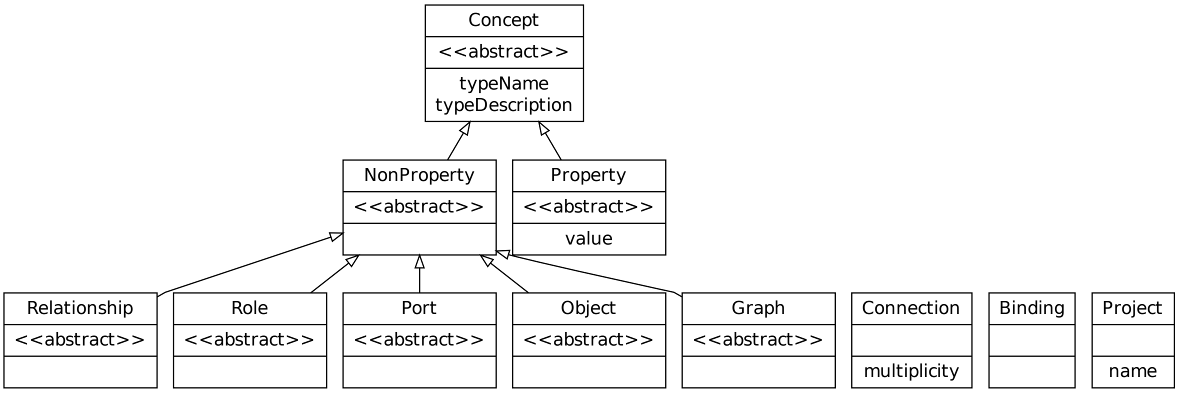


Figure 3.17.: GOPRR meta meta model abstract syntax for type generalisations

Concept is the ancestor or – in object-oriented words – base class [79] for all GOPRR types / classes. It holds two attributes: `typeName`, which is a string storing the name of the instance and `typeDescription`, which stores a description as string / text.

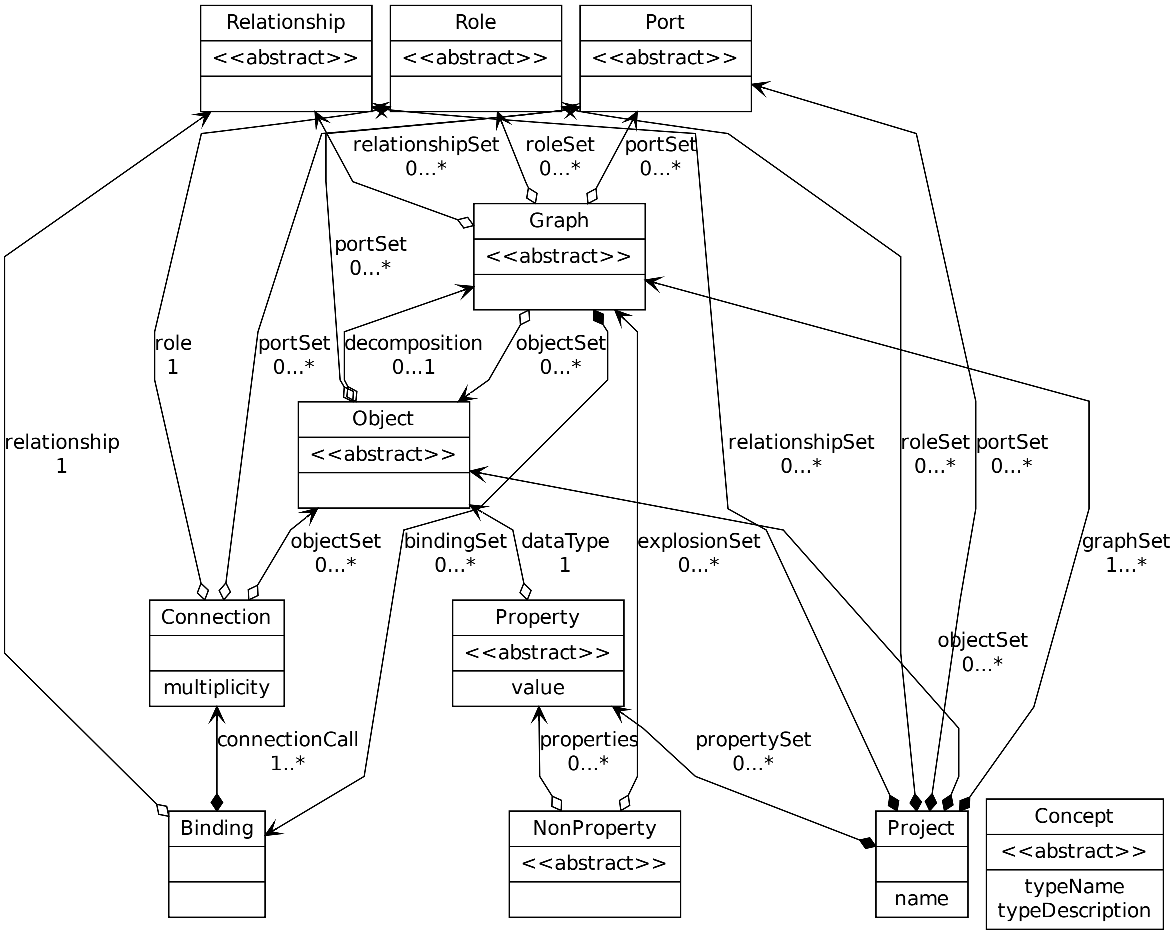


Figure 3.18.: GOPRR meta meta model abstract syntax for type associations

Property is the parent class for all property types. It holds the attribute `value`, which specifies the value of the property and the value's type by the `dataType` association.

NonProperty is the parent class for all non-properties, which was already explained in Figure 3.15. The association `explosionSet` provides the possibility to describe any non-property by one or more or sub graph.

Object is the parent class for all custom object types. It has an optional set of ports (`portSet`).

Relationship is the parent class for all relationship types. A relation specifies the kind of connection between two or more objects.

Role is the parent class for all types describing a role. A role describes the part of an object in a certain relationship.

Port is an optional element of GOPRR. Normally, objects can be directly connected via roles and relationships, but with ports it can be additionally specified, which parts of an object a certain role can connect to.

Connection describes a connection between a set of objects (`objectSet`) with a certain role (`role`) while the attribute `multiplicity` describes the cardinality / multiplicity of the connection. This class should be instantiated directly without further inheritance. Instead of a direct connection between objects, an optional port (`portSet`) (of an object) can be used.

Binding specifies the binding between a relationship (`relationship`) and one or more connections (`connectionCall`). This class should be instantiated directly without further inheritance.

Graph combines all necessary elements for a graphical representation. Therefore, it holds relationships (`relationshipSet`), roles (`roleSet`), objects (`objectSet`), and bindings (`bindingSet`) to connect the preceding ones. The set of ports (`portSet`) is optional.

Project combines one or more graphs (`graphSet`) to a project. This class should instantiated directly without further inheritance.

Again, all classes marked with `<<abstract>>` are abstract classes [79], which generally means they cannot be instantiated directly, only by generalisation / inheritance [62, 79].

In contrast to the abstract meta meta model syntax, Figure 3.19 shows the concrete syntax [46, p. 70] of bindings in GOPRR without ports. Additionally, Figure 3.20 shows the concrete syntax for the usage of optional ports. In Figure 3.19, the objects are connected directly with

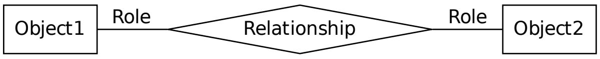


Figure 3.19.: GOPRR concrete binding syntax without ports

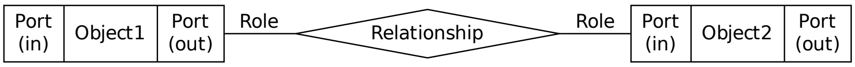


Figure 3.20.: GOPRR concrete binding syntax with ports

certain roles and a certain relationship. In Figure 3.20, roles and relationship are the same, but the objects are connected only indirectly by their ports (out and in). Therefore, it can be modelled that only certain ports can be connected under certain roles and relationship. For example, a connection out→out or in→in could be avoided in Figure 3.20. The concrete syntax of the GOPRR meta meta model consists of further elements for describing sub-graph connections by decompositions and explosions, but which are not explicitly introduced here.

Figure 3.21 shows the instantiation of GOPRR in DSM architectures.

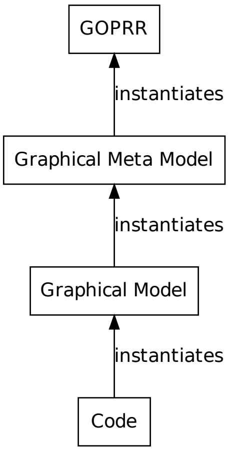


Figure 3.21.: GOPRR DSM instances

3.2. Meta Meta Model Comparison

Generally, DSLs can be divided in two categories:

- textual
- graphical

A textual model is defined by text statements while a graphical model is defined by graphics. Of course, combinations of both are also possible. The difference between those two types cannot be explained completely in general because this is also a matter of the concrete problem domain. A general discussion can be found in [46, p. 50ff].

A goal of this work – modelling an ATP system – includes the development of a meta model for the corresponding specification, the ETCS SRS. These documents are mostly written in natural language, which means text. In minor cases other textual formalisms like tables are used but not generally in a normalised or formalised manner.

To provide significant abstraction from the ETCS specification documents, a pure textual DSL would fail because pure text is always represented linearly. As in the specification documents themselves, this may lead to a lot of references or even cross-references. Combined with the complexity of the specification, this renders a textual representation confusing.

Instead, a graphical DSL provides a lot more representation techniques that are not limited to linear representation methods. For example, references can be represented by sub-graphs. Although the most of the above introduced meta meta models can be used for building models, MOF / Ecore and GOPRR are predestined because they already provide graphical formalisms for meta model definition. Especially, GOPRR was designed for graphical DSLs, which gives it a big advantage compared with the others.

3.3. Domain-Specific Modelling Development Applications

Although a modelling application for a certain DSL can always be implemented explicitly, there are already several applications and tools available for designing a DSL / meta model, modelling, and code generation. In the following, some of the most common and most appropriate for the openETCS case study are introduced.

3.3.1. Xtext

Xtext is a plug-in for Eclipse [9] that provides functionality for creating textual DSLs and generating code from those. It uses ANTLR, which is a parser for BNFs. Therefore, the grammar for a DSL and the code generator can be freely specified, which makes Xtext completely independent from any concrete implementation. It additionally offers the export to Ecore meta models. Its main disadvantage is the lack of any possibility for creating graphical models.

3.3.2. Graphical Modelling Framework

The Graphical Modelling Framework (GMF) is also a plug-in for the Eclipse IDE that combines EMF (see Subsection 3.1.2) and GEF (Graphical Editing Framework) to enable the creation of graphical DSLs. It uses Ecore as meta meta model and its code generation is limited[1] to Java.

3.3.3. Rascal

Like GMF, Rascal [17] is also a plug-in for Eclipse with special focus on graphical meta modelling and modelling. It also has the advantage that it is shipped under an OSS license. Although this approach seems to be quite promising, it is currently still under development and is only provided as an alpha release.

[1] because of EMF

3.3.4. MetaEdit+

MetaEdit+ [56] is a commercial application for creating DSLs. It uses the GOPRR meta meta model and is very specialised for the development of graphical meta models and models. The code is generated from graphs using the MetaEdit+ Reporting Language (MERL) [56]. MERL does not depend on any certain programming framework, like Ecore, and therefore code for any kind of programming language can be generated. Moreover, it provides easier generator development because it only uses the concrete syntax, as in Figure 3.19 and 3.20, for certain graph types. The abstract syntax from Figure 3.17 and Figure 3.18 is not visible to MERL. Also, generators can be restricted to certain graph types and cannot be used on the project-level. In contrast to all other DSM applications mentioned before, MetaEdit+ is the only one that provides a development standard on an industrial level. Since this work focuses on safety-critical systems, this fact cannot simply be ignored.

The main disadvantage of MetaEdit+, related to this work, is that no FLOSS version is available.

3.4. Conclusion

Although a full FLOSS tool chain – including the DSM development applications – fits the aim of this work better, MetaEdit+ as commercial software offers several advantages that cannot be simply ignored. Compared to the other applications and their meta meta models, MetaEdit+ is highly specialised for the development of graphical DSLs. Additionally, it provides a script language for generator development that does not depend on certain programming languages. Furthermore, the usage of the introduced FLOSS applications would always cause additional development effort for the tooling because none provides graphical modelling and general generation in one. Due to the fact that a complete tool development should not be in the main scope of this work, MetaEdit+ and the GOPRR meta meta model are employed for this work. To be able to publish all parts work the corresponding case study in Part III, any meta model and model can be exported to an appropriate XML format like XMI [61]. Such files are then free for publication under any license and are not limited to a certain application.

4 The GOPPRR Meta Meta Model – An Extension of GOPRR

On account of the idea of using a DSL to directly provide additional artefacts for the certification procedure for dependable software, the definition of the abstract syntax [46, pp. 68-69] – including the definition formalism – is crucial. In relation to this work, this primary means the selection of the meta meta model and the developed meta model. Although the selected GOPRR meta meta model provides a good documentation (Chapter 3), the formalisms for defining the syntax of meta models have to be evaluated.

The definition of the concrete syntax [46, p. 70] and the static semantics [46, pp. 69-70] of the underlying meta model is a very important task in the development of safety-critical software under the model-driven architecture (MDA) [44] paradigm. The concrete syntax defines how models can be instantiated from a meta model. Falsely or too "loose" defined concrete syntaxes may lead to models instantiated from those that are in conflict with the required safety properties for the modelled system or software. Those errors in the model could have an impact on all lower instances of a DSL down to the executable binary code.

GOPRR is a graphical meta meta model, which makes the decision for a graphical syntax description formalism obvious. The main advantage is that the syntax description of a meta model can be also defined as meta model of GOPRR. This results in the situation that a concrete syntax description is also a model of the same meta meta model. The complete and detailed description of the original GOPRR syntax description formalisms can be found in [45, app. A] and [56].

Unfortunately, the GOPRR syntax description meta model does not include ports (Subsection 3.1.5), but which are also elements of the meta meta model. Concrete explanations for this could not be found in the corresponding documentation, but a possible reason might be that ports were added later to the meta meta model. This might also be reflected in the acronym GOPRR (Graph, Objects, Properties, Roles and Relationships), which does not include a letter for ports. However, ports provide in some cases great advantages for meta models and are also heavily used in the case study presented in Part III for an interfacing concept. Without ports, this could only have been realised with a much more complex concrete syntax or rather meta model. Neither, the proper definition of the static semantics is included in the original GOPRR syntax description formalism.

A possible solution to these problems could be the extension of the already existing meta model for the syntax definition. The fact that it does not provide possibilities to define syntax for the graphical containment of objects opposed this option. Graphical containment means that the graphical representation of an object o_1 consisting of a set of points P_1 graphically contains another object o_2 if $P_2 \subseteq P_1$. This modelling mechanism can also provide good abstraction possibilities that should not be ignored. The additional integration of ports and graphical containment in the original GOPRR syntax description meta model does not seem very promising because a direct mapping of these meta meta model elements to model elements is not possible, as it is done for the other GOPRR elements.

Thus, to use all GOPRR features and elements for dependable software, a new syntax description formalism has to be developed that ideally can be defined as meta model in GOPRR itself. The formalisms for defining the concrete syntax and static semantics are introduced in the following sections. Furthermore, an abstract syntax model as C++ structure and its intermediate representation as XML are presented because the abstract syntax is needed for the generation process and the definition of the static semantics. Additionally, a possible transformation from the new GOPPRR meta meta model to MOF can be found in Appendix A.

4.1. Concrete Syntax Description Formalism

The name for the new meta meta model was chosen as Graph, Objects, Properties, Ports, Roles, and Relationships (GOPPRR) to emphasise the extension about ports. If the term GOPPRR is used as meta meta model, it refers to the GOPRR meta meta model with the GOPPRR extensions for the syntax definition, the new abstract syntax model, and the new definition formalism for static semantics.

In general, the GOPPRR concrete syntax description formalism provides three different graph types:

1. graph bindings syntax graph
2. sub-graph syntax graph
3. type property graph

Graph Bindings The syntax graph for graph bindings specifies how object types in a certain graph type can be connected. Figure 4.1 shows the elements of such a graph. The root node of a graph binding represents the graph type under consideration. Relationship and role types are child nodes of the graph type. Object types can be children of role or of port types, which are on the intermediate level between role and object types while the last possibility is an extension compared to the original GOPRR syntax description formalism. Another important addition is the definition of graphical inclusions. This means in other words that the graphical placement of an object within the graphical representation of another one can be used for the syntax definition. This graphical inclusion for a certain object type is a child node of the graph type node, and the child nodes are the possible included object types.

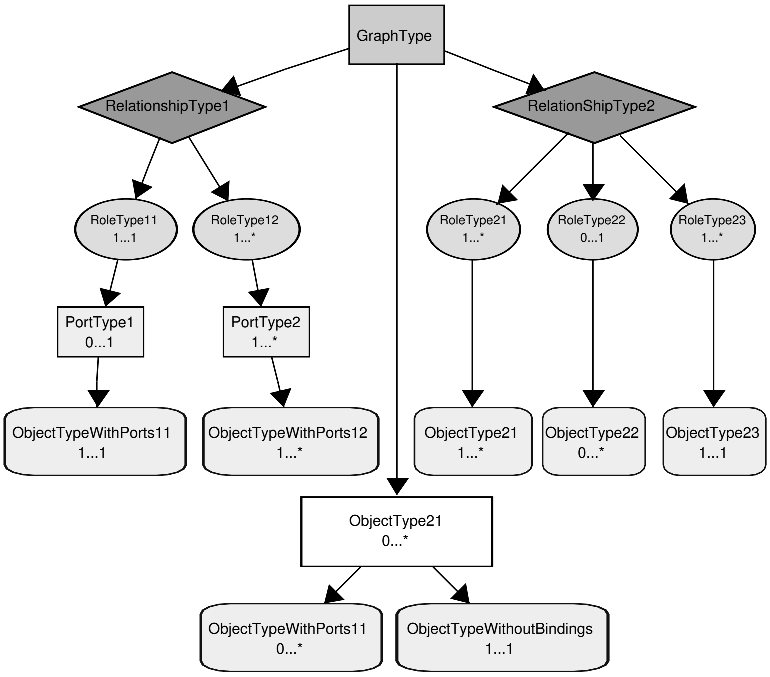


Figure 4.1.: Example of a meta model graph bindings definition

Role, port, and object type nodes have a cardinality written below the name label, which have different meanings for different nodes. The cardinality of roles define how often they can or have to be used in a certain relationship instance. For object and port types, the cardinality refers to their possible connections under the corresponding role type. The cardinality of graphical inclusion nodes refers to their occurrence in the corresponding graph type.

The multiple, graphical occurrence of a certain relationship type node under the same graph type node means a disjunction of the below defined role, port, and object types. In other words, those relationship type definitions can be used alternatively. Also, other elements than relationships in a certain graph (roles, ports, and objects) can be used multiple times since their cardinality may differ for different parent nodes.

Sub-Graphs The syntax graph for sub-graphs and occurrences defines the interconnection between different graph types and the occurrence of object types in a certain graph type. An example is given in Figure 4.2. As in the definition for bindings, the root node is always a graph type. Child nodes are object, relationship, and role types. As stated above, only object types can have decompositions, but all three types may have explosions. To define more complex, potentially recursive, graph interconnections, sub-graph nodes can have again object, relationship, and role type child nodes.

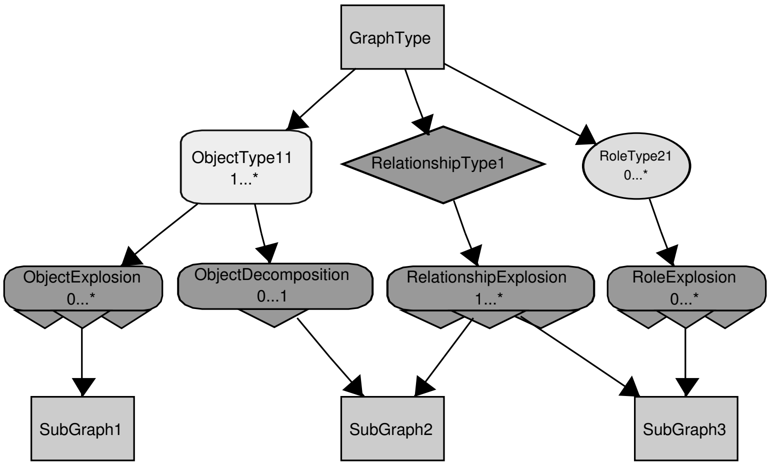


Figure 4.2.: Example of a meta model sub-graph and occurrence definition

Type Properties The GOPPRR type property graph defines the properties of all GOPPRR types, aggregations of non-properties, and inheritance, as shown in Figure 4.3. Graph, object,

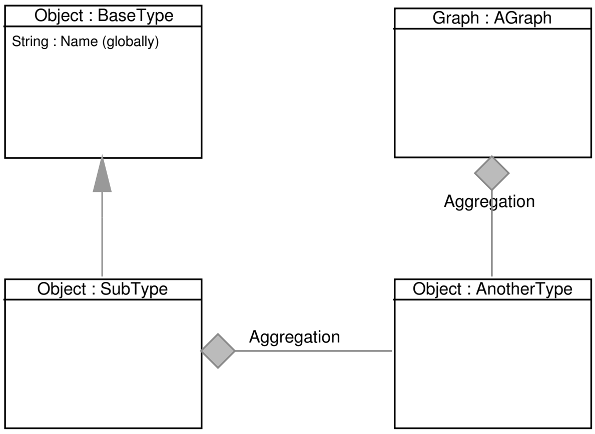


Figure 4.3.: Example of meta model properties definition

port, roles, and relationship types are indicated by rectangular boxes while properties are labelled in the format “[type] : [type name]”. Those types can have properties, which are drawn within the corresponding box of the surrounding type in the format “[property type] : [property name] ([uniqueness])”. Also, they can have aggregations [64] to other elements that are non-property types. Furthermore, all types, including property types, can inherit from others belonging to the same meta type. For example, a graph type can only inherit from another graph type but not from an object type.

GOPPRR Concrete Syntax Meta Model Similar to the original GOPRR concrete syntax description formalism [56], also a GOPRR meta model was created for the new GOPPRR concrete syntax formalism. Hence, a concrete syntax in GOPPRR can also be modelled in MetaEdit+ as instance of this meta model. The concrete syntax of this GOPPRR concrete syntax meta model is not explicitly documented here, but it is located in Section B.6 together with a model of the concrete syntax for the case study in Part III.

4.2. GOPPRR C++ Abstract Syntax Model

Additionally to the meta model concrete syntax definition formalism, the generator capabilities are of high relevance for the creation of dependable software. Typically, generators for models in a certain meta meta model are tool bounded and are not independently implemented. Due to the decision to use MetaEdit+ as meta modelling application (reasoned in Chapter 3), the relevant generator or rather generator language is MERL. Its main advantage is the provided abstraction for navigating through the bindings between objects in graphs. In contrast to the abstract GOPRR meta meta model structure in Figure 3.17 and Figure 3.18, the navigation is simplified in MERL to bindings in Figure 3.19 without ports and Figure 3.20 with ports. The direct drawback is that a generator is only related to the scope of a graph type but not to a project. Also, MERL generators for other GOPRR elements can be implemented but can only be used for graphical representation effects and not for output generation, like source code. Therefore, all elements in a graph are treated only as graph-global, although they are project global, and might be (re)used in several different graphs.

To obtain the information about the partition of an object, the implementation in MERL mostly ends in a complex generator. Additionally, MERL suffers in general from a weak syntax. For example, it lacks of the definition of functions and the specification of the scope of variables. This means simple generators can be easily implemented in MERL while complex and project related generators typically reach an unacceptable level of complexity in their implementation. To handle this issue, the concrete generation of source code and other components must be done outside of MetaEdit+ and MERL because the drawbacks cannot be eliminated by custom extensions and modifications. Furthermore, extensions or modifications of the generation module are not possible since MetaEdit+ is proprietary, closed source software.

For this reason, an external generator application that uses an abstract syntax model should be implemented. The term model refers here to a software class model specified in UML. Nevertheless, from the DSM view the, GOPPRR C++ abstract syntax model is a meta meta model while concrete instantiations corresponds to a model instances within that meta meta model. This abstract model must be provided in a project-oriented manner while the implementation in C++ eliminates the syntactical weaknesses of MERL. The transformation from the internal, concrete syntax model in MetaEdit+ to the C++ GOPPRR model should be done via XML[1], which is explained in Section 4.3 in detail.

In the following, the C++ abstract syntax model is explained in detail. For the initial development, the already existing GOPRR meta meta model abstract structure in Figure 3.17 and Figure 3.18 was taken and then transferred to an UML class structure to be extended as

[1] as intermediate model

usable for C++ classes and objects. The resulting structure of the GOPPRR C++ abstract syntax model is shown in Figure 4.4. Compared with the original abstract syntax, the following main extensions and modifications were done:

- An additional class CCall[2] was added, which is used by CConnection as aggregation [64] to combine an object (CObject), a role (CRole), and an optional port (CPort). Without this addition, it could not be distinguished, which objects and ports belong to the same call. Caused by the circumstance that ports are optional, the size of direct associations [64] of ports in CConnection could be different and the assignment of ports to a concrete pair of object and role would be non-unique. CCall should be interpreted as the definition of one concrete end of a connection (CConnection) or rather binding (CBinding).

- The class CGraphicalContainer was added to provide the information of graphical containment of objects and relationships. Instances are related to exact one graph object because the graphical containment feature is only valid in a certain graph but not for a whole project.

- For all associations that ends at classes representing GOPPRR base elements and have in their multiplicity [64] a not limited upper bound (stated by a *) a STL map [79] is used as container. The object identifier[3] (OID) is used as key [79] value in the map. The map ensures that no duplicates exist in association and elements can be easily accessed by their OID, but it also can be iterated, like a vector / stack or a list.

While the C++ GOPPRR abstract syntax model provides access to all model information, it makes the processing of bindings in a certain graph quite more complex. Compared to the concrete syntax model used for MERL, bindings now cannot easily navigated by directly stating the type name, like

```
.object1~role1>relationship~role2.object2
```

or

```
.object1#port1~role1>relationship~role2#port2.object2
```

To reduce this drawback, the graph representation class CGraph provides several methods that abstracts the navigation via bindings. In the example below, the method returns all roles of a certain type connected to a certain object. The returned map of roles can be then processed or used for further navigation to relationships or roles.

```
/*!
 *  \brief abstraction method
 *
 *  Gets certain roles connected to a certain object in a certain graph.
 *
 *  \param[in] pObject:      pointer to the object
 *  \param[in] Type:         role type
 *  \param[in] bSubString:   if true, Type only must be a substring of role type
```

[2] The name was chosen according to the connectionCall composition in the original structure.
[3] OIDs are not part of the original GOPRR meta meta model but are created in MetaEdit+ for each instance of a GOPRR element and are unique within a project.

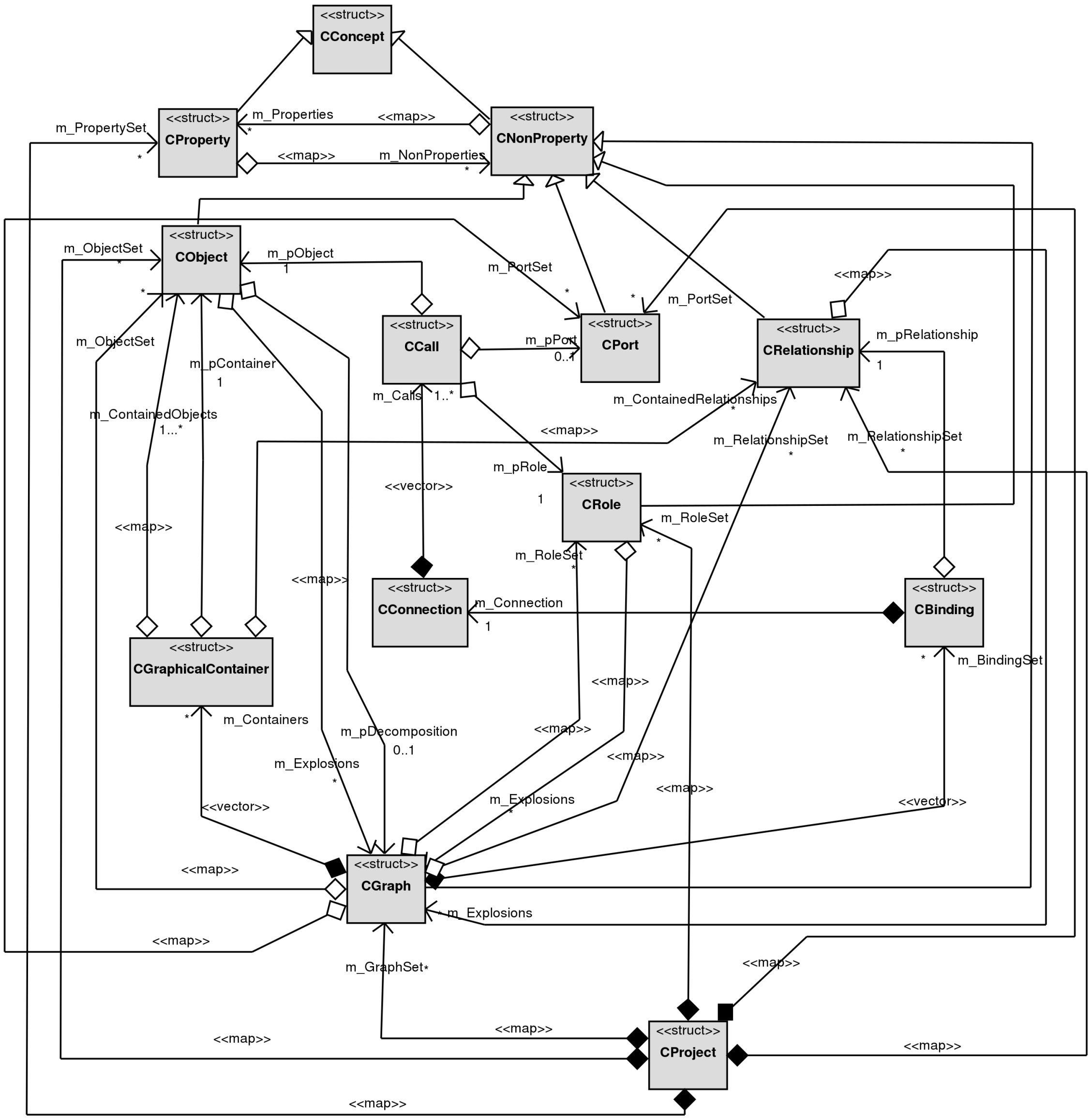


Figure 4.4.: UML class diagram for the GOPPRR C++ abstract syntax model

```
 *   \param[in] bUseException: optional Boolean flag for throwing an exception
 *                             in case of an empty map
 *   \return map of pointers to roles
 */
::std::map< ::std::string, GOPPRR::CRole* >
    Roles(GOPPRR::CObject * const   pObject,
          const ::std::string&      Type,
          const bool&               bSubString     = false,
          const bool&               bUseException  = true
          ) const throw(::GOPPRR::Error::CEmpty);
```

All abstraction methods can be found in Section E.1. Details about the deployment and the implementation are discussed in Chapter 9 along with the openETCS case study code generator application.

Limitations It is necessary to remark that a concrete GOPPRR C++ abstract syntax model only holds information about the related model but not about the related meta model. This means, for example, that for two instances of CObject no information is available if they are sub- and parent-type, or not. Nevertheless, this is not a major disadvantage because the meta modelling and the modelling itself should only be related to the modeller, which means here MetaEdit+.

4.3. GOPPRR XML Schema Definition

To transform an internal MetaEdit+ GOPPRR model[4], an interface between MetaEdit+ and the custom generator or rather C++ abstract syntax model has to be established. This is done by XML as an intermediate model representation. As already introduced in Subsection 3.1.3, the XML Schema Definition [42, 70] provides a dictionary definition mechanism for XML. Accordingly, a concrete instance of a XSD can be interpreted as meta model while an XML valid to the Schema corresponds to a model instance (Figure 3.10).

The fact that XSD provides an object-oriented syntax enables the definition of the XML Schema in the exact same manner as the GOPPRR C++ abstract syntax model in Figure 4.4. Furthermore, the usage of so-called key and keyref [42] elements provides the definition of an even more detailed meta meta model. The C++ abstract syntax model only defines that all GOPPRR elements are instances / objects in a project object, but in all other associations among the model those are only connected by references / pointers [79]. An important constraint is that all GOPPRR elements referenced at any point of the model must occur once as instance in the related project object. The definition of this constraint in the XSD for GOPPRR renders the integration into the C++ model unnecessary because it is always instantiated by a XML model fulfilling it. The complete GOPPRR XSD is located in Section E.2 in the appendix.

4.4. MERL GOPPRR Generator

The generator for transforming GOPRR models in MetaEdit+ to external XML GOPPRR files is briefly introduced here. Corresponding to the requirement that the GOPPRR C++ abstract

[4] GOPPRR model means a concrete instance of a meta model in GOPRR.

syntax model should be independent from a certain meta model, which is also reflected in the corresponding XSD in Section 4.3, the generator in MetaEdit+ should be held general. Most of this could be easily covered by implementing general generators for each GOPRR or rather GOPPRR element, which are called for each graph. Owing to the limitation of MERL / MetaEdit to graph-oriented generators, these general generators have to be initially called from a graph and then executed recursively. Therefore, the start-point for the generation must be some kind of root-graph[5]. This also means that the meta model graph types must be all related in sub-graph structure. Otherwise, generators specialised for the concrete meta model must be developed. The complete MERL source code of the GOPPRR XML generator can be found in Section F.1

Drawbacks Although all meta models with a coherent sub-graph structure can be generated by the general generators, for each root-graph type[6] a specialised but simple generator must be implemented that calls the others. A more grave problem is a limitation in MERL / MetaEdit+ that occurs if non-property elements are used as property. For example, if an object type is used as property in another object type. It is possible in MERL to access such non-properties as properties, including their properties, but only by using the concrete type name of the non-property. Obviously, in a general generator, concrete type names are not available. The only solution as a work-around is to provide for every meta model a specialised generator that generates the corresponding XML elements for all non-properties used as properties.

It must be remarked that this drawback is only related to the modelling application but not to the meta meta model.

4.5. The Object Constraint Language for Static Semantics

The Object Constraint Language (OCL) [63] was developed to refine software models described by the Unified Modelling Language (UML) [64] in more detail. It provides a very general and big syntax to cover most object-oriented model situations. Nevertheless, not all available syntax expressions are required by the usage of a GOPPRR model because the related class-model is invariant and well-defined by the GOPPRR C++ abstract syntax model (see Section 4.2). Additionally, OCL is also well-defined and -documented, e.g., in [63]. Hence, a comprehensive introduction of OCL will be omitted and only parts that are relevant for the GOPPRR meta meta model will be presented.

OCL expressions are always related to a certain UML graph instance, which can be of different types. One possibility is a class diagram [64] as it is used in Figure 4.4 for the definition of the GOPPRR C++ abstract syntax model. Due to the fact that the C++ abstract syntax model primary consists of relations – methods / operations are only used for abstraction – and these relations only hold all relevant information about the related GOPPRR model, the main OCL construct needed is the invariant. An invariant must always be fulfilled. If they are not graphically bounded to an element in an UML diagram, all OCL expressions must be defined in a certain context. This is typically the class name. For example, to define one or

[5] related to sub-graphs by decompositions and explosions

[6] typically only one per project

more invariants for a GOPPRR project, the context would be `CProject`. To select this context, the following OCL statement is used:

```
context CProject
```

An invariant construct within a certain context is prefixed by an `inv:` statement followed by the invariant expression itself. This expression follows in part the C++ syntax for accessing methods and attributes of objects [79]. For clarification, the following OCL statement defines that all GOPPRR project objects must have a non empty ID:

```
context CProject
inv: m_ID <> ''
```

The following examples are used to clarify the usage of OCL for constraint definition, which are taken from the case study presented in Part III.

4.5.1. OCL Example: Global, Numerical Criteria

The global occurrence of the graph type "EVCStateMachine" must be exact one:

```
context CProject
inv: m_GraphSet->select(m_Type = 'EVCStateMachine')->size() = 1
```

4.5.2. OCL Example: Numerical Criteria within a Graph

All objects of the type "EVCState" within any graph of type "EVCStateMachine" must have at least one explosion:

```
context CProject
inv: m_GraphSet->select(m_Type = 'EVCStateMachine')->forAll(m_ObjectSet->select(m_Type
    = 'EVCState')->forAll(m_Explosions->size() >= 1) )
```

Combined with the prior constraint (Subsection 4.5.1), it can be reduced to:

```
context CProject
inv: m_GraphSet->select(m_Type = 'EVCStateMachine')->size() = 1
inv: m_GraphSet->any(m_Type = 'EVCStateMachine').m_ObjectSet->select(m_Type =
    'EVCState')->forAll(m_Explosions->size() >= 1)
```

An iteration (by the `forAll()` statement) over a set of multiple "EVCStateMachine" graphs is not required because the first invariant already requires exact one instance of it. For simplification, this reduction is done for all following constraint examples.

4.5.3. OCL Example: Boolean Criteria within a Graph by further Type Specification

All "EVCGuard" properties of all "EVCTransition" relationships within all "EVCStateMachine" graphs must also exist in at least one explosion of the corresponding objects connected by the "CurrentEVCState" role of the before selected relationship:

```
context CProject
inv: m_GraphSet->select(m_Type = 'EVCStateMachine')->size() = 1
inv: m_GraphSet->forAll(
  graph |
  graph.m_RelationshipSet->select(m_Type = 'ModeTransition')->forAll(
    relationship |
    graph.m_BindingSet->exists(
      m_Connection.m_pRole.m_Type = 'CurrentState'
      and
      m_Connection.m_Calls->exists(
        m_pObject.m_Explosions->exists(
          explosion |
          explosion.m_ObjectSet->exists(
            relationship.m_Properties->any(m_Type =
                'EVCGuard').m_NonProperties->any(m_Type = 'ModeGuard')
          )
        )
      )
    )
  )
)
```

4.5.4. OCL Example: Port Connection Specification

For all graphs and for all ports, the amount of connected “DataInput” roles must be exact one:

```
context CProject
inv: m_GraphSet->select(m_Type = 'EVCStateMachine')->size() = 1
inv: m_GraphSet->forAll(
  graph |
  graph.m_PortSet->forAll(
    port |
    graph.m_BindingSet->select(m_Connection.m_Calls->exists(m_pPort = port and
        m_pRole.m_Type = 'DataInput'))->size() = 1
  )
)
```

4.5.5. OCL Example: Graphical Containment

In all “CommunicationIn” graphs, all “Packet” objects that are graphically contained by any “Telegram” object must exist also in the decomposition of the related “Telegram” object:

```
context CProject
inv: m_GraphSet->select(m_Type = 'EVCStateMachine')->size() = 1
inv: m_GraphSet->select(m_Type = 'CommunicationIn')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'Telegram')->forAll(
    telegram |
    graph.m_Containers->any(m_pContainer = telegram).m_ContainedObjects->forAll(
      object |
      telegram.m_pDecomposition.m_ObjectSet->includes(object)
    )
  )
)
```

4.5.6. Limitations

Although the direct usage of the GOPPRR C++ abstract syntax model provides better possibilities to define static semantics [46, pp. 69-70], as it is available within the MetaEdit+ application and its concrete syntax model, specific limitations cannot be resolved because the XML file used as intermediate model is generated by MERL. Therefore, information that is not accessible or visible by MERL cannot be exported to the XML file and accordingly cannot be included in the GOPPRR C++ abstract syntax model. This results in the following limitations:

- The most remarkable limitation is the absence of sub- or supertypes. This means if “Parent” is an object super type, “Child” is a subtype in the meta model, a concrete model would have an instance A of “Parent”, and an instance B of “Child” the constraint

  ```
  inv: objects->select(m_Type = 'Parent')->size() = 2
  ```

  would not be fulfilled because `m_Type = 'Parent'` only matches A.
- As already stated in Section 4.2, OCL cannot be used to check constraints for a meta model without a concrete model instance.

4.5.7. Conclusion

The examples for certain constraints show that general constraints are very simple to be defined by the OCL combined with the GOPPRR C++ abstract syntax model. The statement complexity increases if constraints need the navigation through the GOPPRR bindings in graphs. The reason is that the transformation from the MetaEdit+ internal GOPRR model to the GOPPRR C++ abstract syntax model does not transform the navigation on GOPRR / GOPPRR bindings into associations between classes. This was reasoned in Section 4.2. Also, OCL only provides navigation through classes by their associations. Thus, OCL statements have to use the classes for bindings, connection, and calls for navigation, which causes longer and complexer statements.

In spite of this disadvantage, the following advantages of using OCL for GOPPRR prevail:

- The syntax is already well-defined and -documented as specified in [63].
- It is already widely used and known in the field of model-driven software development with UML.
- It is based on MOF, the UML meta meta model.

This reasons the adoption of OCL as constraint language in this work.

4.6. Tool Chain

To integrate all elements introduced in this chapter in the development cycle of open source software for dependable train control systems, a tool chain must be established. In a model-driven architecture (MDA), those elements are generators and artefacts [46]. Generators can only be connected with artefacts and vice versa. The resulting graph is shown in Figure 4.5

where artefacts are represented by small boxes, generators by small ellipses, and applications by big boxes surrounding a set of artefacts and generators. All elements are explained in detail below and are grouped by applications. To better distinguish between element properties, the following abbreviations are used:

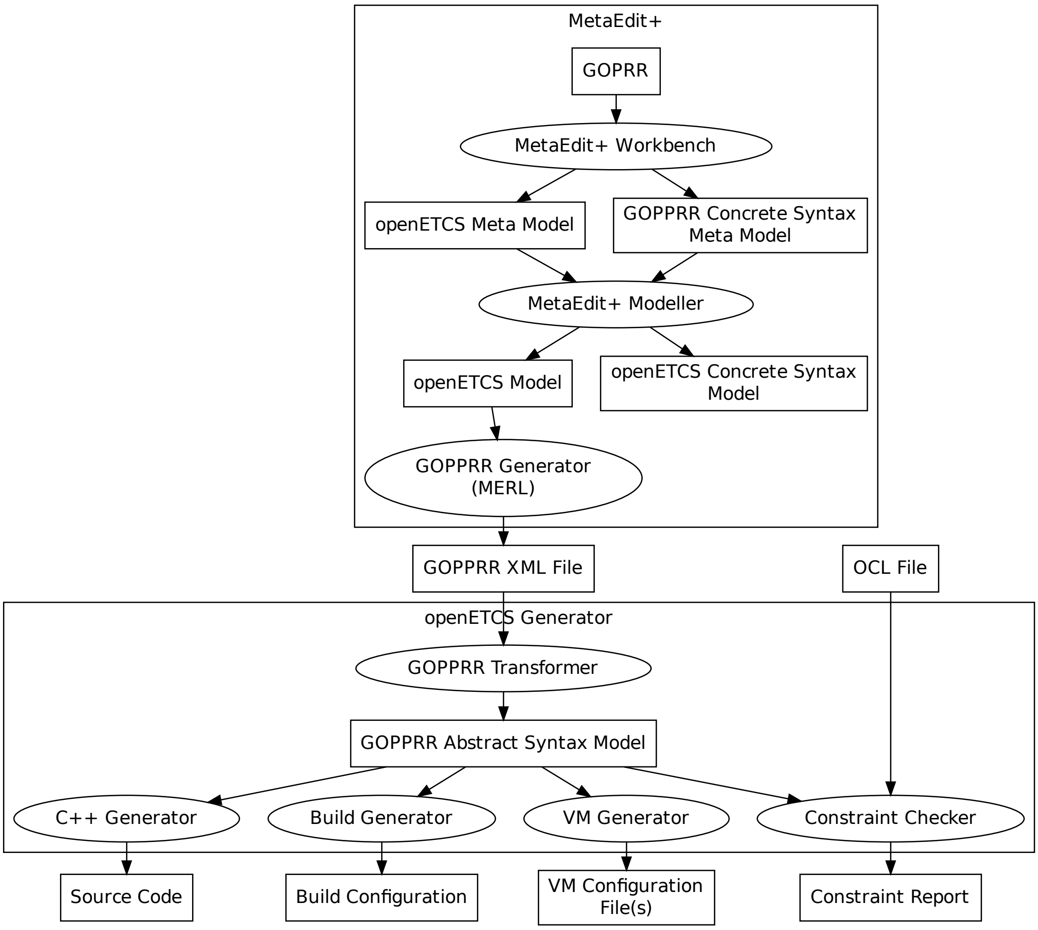


Figure 4.5.: Tool chain for dependable open source software for openETCS

G	generator
A	artefact (generation not specified)
AF	fully generated artefact
AM	manually[7] generated artefact

MetaEdit+ All elements within this box are located in and managed by the MetaEdit+ application. Artefacts, output by MetaEdit+, are outside the box.

[7] in the meaning of human interaction

GOPRR (A) The GOPRR meta meta model.

MetaEdit+ Workbench (G) Part of MetaEdit+ for meta modelling / development of meta models.

openETCS Meta Model (AM) The formal openETCS specification language.

GOPPRR Abstract Syntax Meta Model (AM) Abstract syntax definition language for GOPPRR meta models.

MetaEdit+ Modeller (G) Part of MetaEdit+ for modelling / development of models of certain meta models.

openETCS Model (AM) Formal and graphical model of the ETCS specification related to the EVC and openETCS.

openETCS Concrete Syntax Model (AM) Graphical model of the openETCS concrete syntax model.

GOPPRR Generator (G) Transforms any model in GOPRR to a GOPPRR XML intermediate model file.

openETCS Generator Although this generator application is implemented in C++ and uses the general GOPPRR XML file format as interface to MetaEdit+, the majority of its generators are especially designed for the usage with openETCS models. This issue will be specified in detail in Part III.

GOPPRR Transformer (G) Transforms a GOPPRR XML file model into a GOPPRR C++ abstract syntax model.

GOPPRR Abstract Syntax Model (AF) Concrete instance of the C++ abstract syntax class model representing the openETCS model.

C++ Generator (G) Generates the source code for the openETCS model.

Build Generator (G) Generates the build configuration for the openETCS source code.

VM Generator (G) Generates the configuration files for virtual machines for openETCS (see Chapter 6).

Constraint Checker (G) Generates a report about fulfilled and unfulfilled constraints for a GOPPRR by OCL statements.

External Artefacts The following artefacts are not located in any application and are used as interface, input, or final output of the tool chain.

OCL File (AM) List of OCL statements defining constraints for models of the openETCS meta model as static semantics.

Source Code (AF) Generated C++ source code for the openETCS model.

Build Configuration (AF) Generated build configuration to compile and link the generated openETCS source code.

VM Configuration File(s) (AF) Generated configuration files for hypervisors running virtual machines corresponding to the openETCS model.

Constraint Report (AF) List of unfulfilled constraints within the openETCS model.

The tool chain cannot end with the final generated artefacts as output because none of them is executable. This chapter only provides background information needed for understanding the following parts of this work and therefore the complete tool chain for the case study is presented in Part III.

Drawback One idea of modelling the syntax of a meta model as model under the same meta meta model was to be able to directly generate the meta model from this syntax model. Unfortunately, it emerged that the XML file format used by MetaEdit+ for import and export of meta models [56] does not support port types. Ports are only exported (and imported) type-less as graphical connectors in object symbols / graphical representations. Therefore, after (re-)importing a meta model via XML, all port types are stripped from the meta model. This lack of port type support is not documented within the MetaEdit+ manual [56] and was only found in support forums.

An alternative would be to use the internal MetaEdit+ patch file format [56], but it is proprietary, not documented, and binary. To use it, the format would have to be re-engineered – without any guarantee of success – and a binary generator would have to be developed. Nevertheless, because the extension of existing modelling application is out of the scope of this work, the openETCS concrete syntax model is manually kept synchronous with the openETCS meta model and primary used for documentation purposes. This drawback is only related to the modelling application but not to the meta meta model.

4.7. Conclusion

The GOPPRR extension of the GOPRR meta meta model provides good techniques and formalisms integrated in a tool chain to fulfil the objectives of this work. The GOPPRR C++ abstract syntax model provides now all formalisms to fully describe the static semantics for a meta model of GOPRR, which is a must for modelling of safety-critical systems. Additionally, it provides better abstraction because it separates definition of element properties and sub-types from graph-binding and sub-graphing. That the abstract syntax model cannot be used to automatically generate a meta model concrete syntax is an obvious disadvantage, but it is acceptable within the context of this work.

The interconnection of the GOPPRR generator, a GOPPRR XML file, the GOPPRR transformer, and finally the GOPPRR C++ abstract syntax model enables a separation of (meta) modelling application and generator to develop more sophisticated generators that are not limited to the syntax of a certain generator language. As a by-product, this interconnection is meta model independent and could be used for any further project. There are some minor limitations to this independence, but they can be compensated with little effort for each meta model.

With OCL, an already well established formalism for defining object-oriented constraints for the static semantics is integrated. In certain cases, its syntax might not provide the same abstraction as a custom constraint language, but this small difference does not justify the complete new development of tools like a parser. Also, the constraint checker is meta model independent, which is not on account of OCL but of the independent GOPPRR C++ abstract syntax model. It could be argued that the introduction of the GOPPRR C++ abstract syntax model combined with the OCL would render the graphical description formalisms for graph bindings, sub-graphs, and type properties obsolete. Although, this is correct – all allowed bindings, decompositions, explosions, and properties can be defined as constraints for each graph type – the resulting OCL statements would be much more complex than the presented examples. Hence, those are very bad intuitively understandable. Furthermore, the three graphical, concrete syntax formalisms could be interpreted as abstraction from the very general but more complex description formalism provided by OCL.

All final output generators are directly related to the meta model and will be discussed in Part III. The idea of the transformation from the GOPPRR to the MOF meta meta model in Appendix A offers the possibility to use another meta meta model and modelling tool and to reuse existing meta models, models, and, of course, the external generator elements.

Part II.

Dependability

5 Verification and Validation

Since software engineering methods exist, verification and validation (often abbreviated V&V) are integral parts in software development. They are used in traditional software development strategies such as waterfall or V-model [78, p. 314] but also in modern methods.

Verification and validation mean in general to check if a certain software is correct, but they also relate to two different aspects:

Verification means to check if a software is implemented correctly. It can be interpreted as low-level checking and is mostly related to the implementation of the software.

Validation means to check if a software fulfils the requirements. It can be interpreted as high-level checking and is mostly related to the design and purpose of the software.

Although verification and validation are often applied at the end of a software development cycle, according to the aims of this work previously stated in Chapter 1, verification and validation should be also taken into account during the design and implementation phases to implicitly reduce errors in these development phases. This approach is also required by most of the relevant standards related to the case study in this work, which will be described in the following sections. This chapter introduces the concepts of verification and validation in the scope of this work while the concrete realisation and results are described in Part III.

This chapter starts with the introduction of standards that are applicable in the railway domain and strategies and techniques they define to develop certified software for safety-critical systems. Additionally, some related[1] standards are shortly explained to compare the proposed methods. Afterwards, special issues related to the development of safety-critical software as OSS / FLOSS are illustrated and a specialised software life cycle is presented. This is also used to extend the general tool chain from Figure 4.5 about the derived integration of verification and validation.

[1] not directly applicable

5.1. Applicable and Related Standards

For the development of safety-critical systems exist several (industrial) standards. Figure 5.1 introduces those standards relevant for this work, which will be roughly discussed in the following:

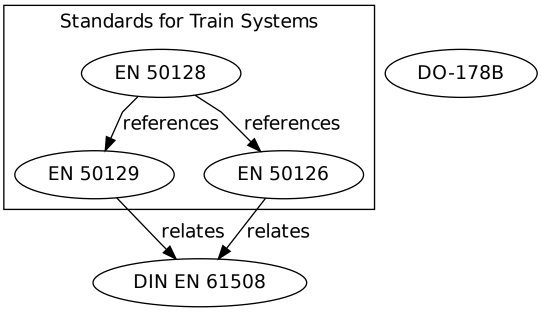


Figure 5.1.: Standards for the development of safety-critical systems

DIN EN 61508 Functional safety of electrical / electronic/ programmable electronic safety-related systems: The DIN EN 61508 is a general standard, which is applicable to all safety relevant electrical, electronic, and programmable electronic systems. It can be interpreted as a superior standard [19].

EN 50128 Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems: The EN 50128 is a specialised standard for software for train control systems. It refers to the EN 50129 and EN 50126 [11].

EN 50129 Railway applications – Communication, signalling and processing systems – Safety related electronic systems for signalling: The EN 50129 is a specialised standard for communication, signalling, and processing in train systems. It is related to the DIN EN 61508 and bases partly on it [13].

EN 50126 Railway applications – The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS): The EN 50126 is a standard for safety and reliability aspects in the field of train operators. It is also related to the DIN EN 61508 because its analysis procedures are based on it [10].

DO-178B Software Considerations in Airborne Systems and Equipment Certification: The DO-178B is a standard developed by the RTCA for software for airborne systems in general [74].

Although this is obviously not an applicable standard for the railway domain, it is used in this document to additionally introduce techniques and methodologies from other domains.

All standards are described in detail in the following subsections.

5.1.1. DIN EN 61508

The DIN EN 61508 is a general and superior standard, which is released by the German Institute for Standardisation[2], but it is also a European standard. Its focus is not only on the development of software but on the hardware or rather system development. In this work, only the development of software is included, but to enable a verification for a complete hardware and software system Part 3 [19, ch. 3] of the DIN EN 61508 should be taken into account. The core concept of the DIN EN 61508 is the so-called safety life cycle, which defines the development process for the whole system. Due to its complexity and copyright restrictions, the safety life cycle is not explained in detail in this document but can be found in [19, ch. 3].

The main principle of the DIN EN 61508 is that every possible hazardous failure for humans or the environment must be avoided by safety-functions of the related system. Each safety-function has a safety-integrity-level (SIL). The SIL describes the performance of a safety-function. This means how reliable is the execution of a safety-function related to a certain possible, hazardous failure. There are four safety-integrity-levels ($n \in SIL = \{1, 2, 3, 4\}$) while 1 is the lowest and 4 the highest performance. If a system only needs safety-functions with a SIL lower than 1, the DIN EN 61508 is not applicable.

Since this work is only related to software development and not to a (complete) programmable electronic system, this standard is not applicable and is only used as reference and for clarification of the general certification process.

5.1.2. EN 50128, EN 50129, and EN 50126

Figure 5.1 shows that EN 50128, 50129, and 50126 are a group of related standards for train systems. Therefore, they are explained in detail in one section as group and not separated. While EN 50128 is the main standard for this work, 50129 and 50126 are introduced because they are referenced by the main standard. Analogue to the the safety-integrity-level of the DIN EN 61508, a software safety-integrity-level (SWSIL) is used. The difference is that for the SWSIL an additional level ($m \in SWSIL = SIL \cup \{0\}$) is used, which was explicitly forbidden for the SIL. SWSIL $m = 0$ is used for software functions that are non-safety related. Similar to he DIN EN 61508, the EN 50128 defines a life cycle, which is especially related to the software development. Hence, it is called software life cycle, which can be found in [11].

Furthermore, the EN 50128 defines techniques and measures for the different steps in the software life cycle. Depending on the SWSIL they are sorted by categories:

- mandatory
- highly recommended
- recommended
- no recommendation
- not recommended

[2] German: Deutsches Institut für Normung

The category of a certain technique or measure depends on the SWSIL. Due to their extensiveness, the techniques and measures are not presented here in detail but can be found in [12].

5.1.3. DO-178B

In contrast to the preceding ones, the DO-178B is neither a German nor a European standard. It is released by the RTCA located in the United States. It is applicable for software in airborne systems and therefore not applicable for this work. Anyway, it is additionally introduced because it provides different techniques and measures from the EN 50128. [74]

Due to the fact that the DO-178B is not related to any other of the relevant standards, no direct corresponds can be found, but nevertheless there exist similarities. Like all preceding standards, it defines a certain life cycle, which includes a software life cycle [74]. Most of the life cycle elements can be set in a relation to elements of the software life cycle of the EN 50128 in Subsection 5.1.2. A detailed description of the software development life cycle can be found in [74].

The DO-178B also defines levels for software, but, in contrast to the preceding standards, these levels $o \in \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$ describe the maximal hazard a possible failure could cause in the software.

Level A describes software which failure would prevent the continuation of a safe flight or a landing.

Level B describes software which failure would cause a large reduction of safety margins or functional capabilities.

Level C describes software which failure would cause a reduction of safety margins or functional capabilities.

Level D describes software which failure would cause a not significant reduction of safety margins or functional capabilities.

Level E describes software which failure would not affect the operational capability.

Similar to the EN 50128, the DO-178B also provides techniques and measures that depend on the software level. In contrast to the EN 50128, these techniques and measures are not sort in different categories. It is only distinguished between techniques and measures that must be satisfied for a certain level and are not mandatory, but are possible to use. For mandatory elements, it is additionally distinguished between techniques and measures that only have to be satisfied and those that have to satisfy with independence. Independence means here that the result of a certain technique or measurement must be at least archived by two different software development teams.

5.2. Verification and Validation as Open Source Software

As already defined in Chapter 1, one of the aims of this dissertation is the creation of a suitable open source solution for model-based design, verification, and validation of safety-critical control systems. Consequently, this means that the used tools, mechanisms, and software for verification and validation must be distributed also as open source software. This is also defined by the Open Proofs concept [65].

Section 2.3 mentioned and explained the case of openETCS, for which the development as open source software can mean more safety compared to the development as proprietary software [38, 37]. For this purpose, there exist already approaches and solutions such as TOPCASED [81], an Eclipse plug-in and toolkit for the software development of safety-critical systems, which contains tools for modelling (in UML), model-checking, simulation, documentation, and further more. Unfortunately, TOPCASED currently does not provided directly any meta modelling [81] and therefore is not usable for this work.

Because no complete open source software suite or toolkit for certified meta modelling is available, all used tools and software should be open source software, if possible. If proprietary software or tools are used, they must have at least to support the export of the corresponding content to an open file or data format. For example, the meta modeller MetaEdit+ (Subsection 3.3.4) is not distributed as open source, but it can export any model as XMI, which is an open standard [61]. This means any meta model and model created with MetaEdit+ cannot be accessed directly without a licensed version, but each meta model and model exported to XMI can be viewed, inspected, and modified by an open source community. Also, the in this work developed GOPPRR XSD is publicly available and can be used for the distribution as OSS / FLOSS.

5.3. Software Life Cycle with Domain-Specific Modelling

The software life cycle for DSM describes the life cycle of the development of safety-critical open source software with DSM in the railway domain. It takes all relevant information from preceding chapters and sections into account that are necessary to finally fulfil all aims for the software. These are mainly, on the one side, the necessities for open source software (Section 5.2) and, on the other side, all specified requirements for the applicable standard EN 50128 (Subsection 5.1.2). Figure 5.2 sketches the DSM software development concept as software life cycle. It shows the connection of the different phases of the development over time. Some phases must be executed in parallel because they have a high cohesion and cannot be executed independently. It is obvious that most phases are executed iterative and maybe lead back to their predecessors as it is common in software development strategies. The corresponding edges are neglected in Figure 5.2 for graphical clearance. This life cycle can be used to refine the software life cycle of the EN 50128.

The Verification & Validation Suite is additionally introduced, which combines both aspects and defines for each a superior technique:

Dynamic Testing is a technique used for verification, which focuses on the functionality [78, pp. 316ff] of a system or component. The corresponding test implementations are often derived from the Software Architecture Specifications [11].

Model Constraint Checking is a technique used for validation, which checks model instances for certain constraints and static semantics. Typically, these are derived from the Software

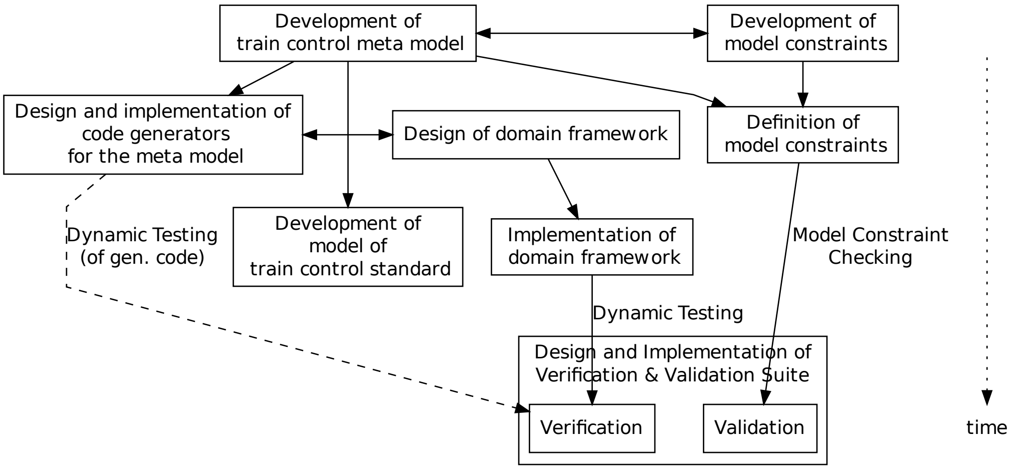

Development of train control meta model
Development of model constraints
Design and implementation of code generators for the meta model
Design of domain framework
Definition of model constraints
Dynamic Testing (of gen. code)
Development of model of train control standard
Implementation of domain framework
Model Constraint Checking
Dynamic Testing
Design and Implementation of Verification & Validation Suite
Verification
Validation
time

Figure 5.2.: Software development life cycle for DSM in the railway domain

Requirements Specifications [11].

To directly integrate verification and validation in the development process, the original tool chain for the GOPPRR meta meta model extension (see Section 4.6) has to be extended, which is shown in Figure 5.3. The generator for functional testing [78, p. 316] refers to a concrete dynamic testing category that should be applied.

It must be emphasised that this extended tool chain still is very general since it does not yet take the domain framework into account, which always heavily depends on the concrete meta model. Furthermore, the code generation also depends on the domain framework because it builds the link between models and the framework. Thus, a complete or rather more concrete tool chain is given in Part III for a certain example in form of a case study.

5.4. Conclusion

This chapter introduced standards for safety-critical systems and discussed the issues for software development of such systems. Besides standards directly related to the domain of train control systems, also a general standard in form of the DIN EN 61508 was exemplary introduced. The DO-178B for airborne systems was used to additionally introduce principles and techniques for a complete different problem domain.

Since all standards are life cycle oriented, a specialised software life cycle was developed, which directly integrates the development of a DSL for a certain train control standards in general. Also, considerations for the development as FLOSS were taken into account.

The general tool chain for the GOPPRR meta model was extended about the integration of verification and validation according to the considerations related to the standards. Nevertheless, the integration of the software life cycle for DSM into the life cycle of the EN 50128 cannot be realised in the case study because it would exceed the limits of a dissertation.

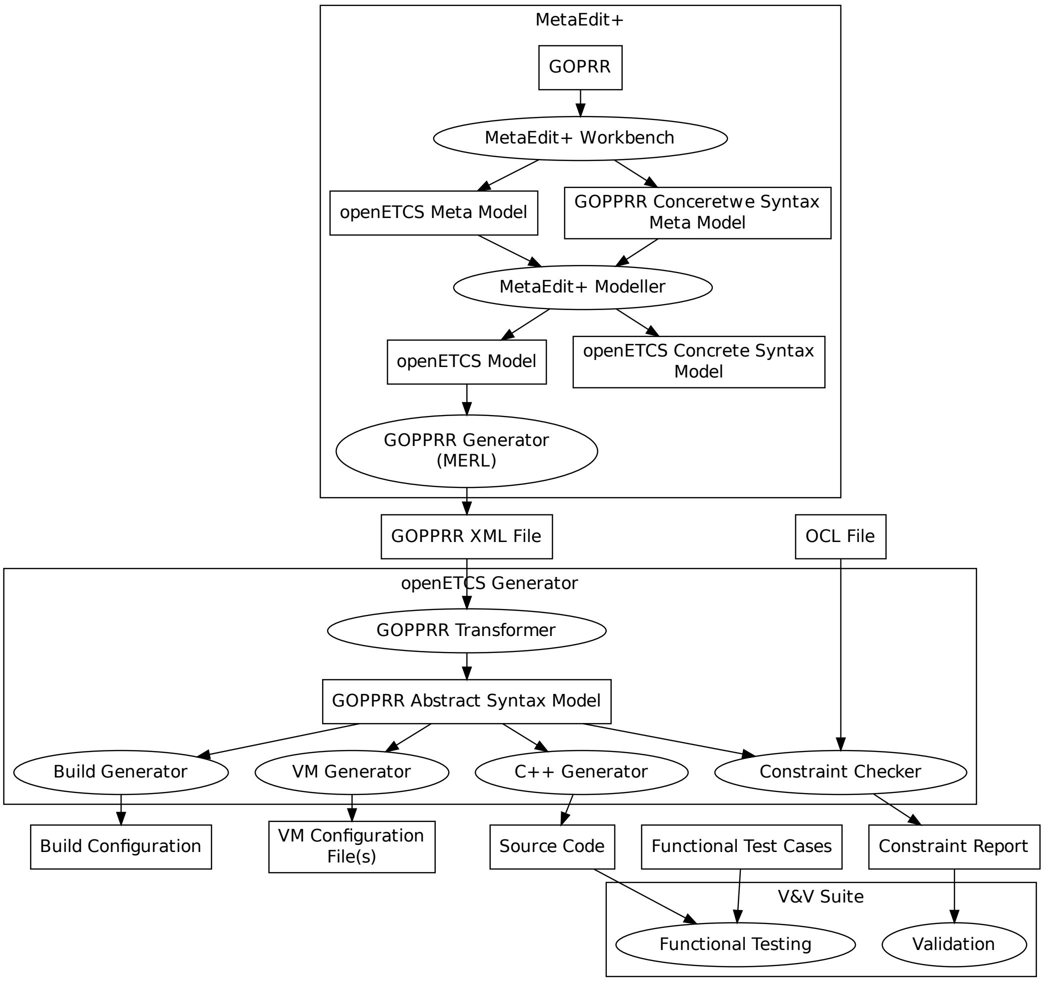


Figure 5.3.: Extended GOPPRR software development tool chain with V&V

6 Security in Open Source Software

As described in Chapter 3, an important part of this work is the modelling of software. Specifically, the development of a DSL for a train control system using ETCS as case study is one of the superior aims of this work. Several advantages for the usage of OSS were already introduced in Chapter 1 and Chapter 5 while the latter one explained why the development as and with FLOSS is relevant for this work and how it should be integrated.

Since the terms OSS and FLOSS mainly refer to source code, the term open model or open architecture is more accurate in relation to this work. Generally, the denomination for open DSM software can be related to the instance of a DSL, which it relates to. The resulting denominations and their correspondence of Figure 3.4 from Section 3.1 are presented in Figure 6.1.

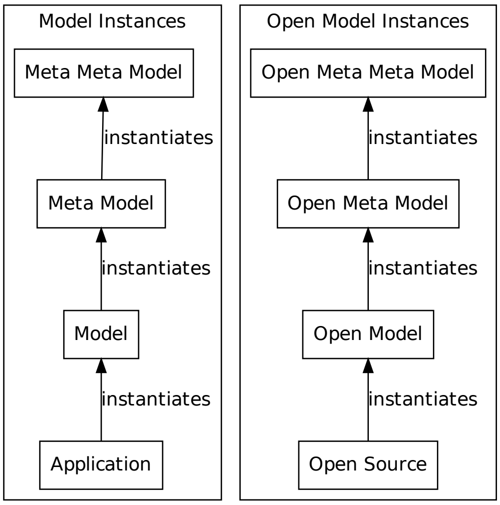


Figure 6.1.: Denomination for open DSM architecture instances

A meta meta model distributed under the principles of OSS / FLOSS is called open meta

meta model, a meta model is called open meta model, a model open model, and an application (or software), of course, open source.

This means that most parts of the software for this work should be developed (and published) as open meta model and open model. Only the domain framework, generators, and parts of the verification and validation suite apply directly to the term open source.

This approach leads to a security related problem because parts of the architecture can or should be (re)implemented or extended by users / suppliers. As explained in Chapter 5, the developed software should be verified, validated, and certified, which cannot be done for (later) additional implemented software from third-parties in advance. This means that verification and validation cannot be assured for all parts of the openETCS software because the third-party implementations from other users / suppliers can only be verified and validated by their developers or later in an additional process. Therefore, it have to be assumed that third-party software that is not verified and validated does not comply with the EN 50128 (Subsection 5.1.2) and may even hold faulty or malicious software, e.g., a Backdoor. Malicious software may even compromise software parts that were verified and validated before.

On the other hand, if additional supplier implementations are validated and certified, this has to be done for the additional software, the open source, and the open model software together. The result is an extensive process because all possible impacts of the additional software to the already certified open source and open model software have to be analysed.

Figure 6.2 shows a very general example of how a third-party or supplier implementation could compromise other parts of an open model software. This example has one model implementation, which is certified, because it was directly generated from the certified open model. Additionally, it holds two supplier implementations, which both instantiates certain model parts of the open model software. These supplier implementations cannot guarantee to be certified whereas one of them holds even malicious code. The communication between different implementation parts is often required. A typical method is the usage of shared memory. This means two (or more) implementations read and write to same areas in the memory to communicate. Unfortunately, this mechanism can also be misused to compromise other implementation parts. Therefore, the malicious supplier implementation could access the memory of any other implementation and change their data used for execution or even the execution itself.

Measures and techniques should be found to minimise the possible influence of a faulty or malicious supplier implementation. Already existing strategies and a novel approach with the usage hardware virtualisation are introduced and compared in the following sections and are also described in [26].

In the following section, traditional methods for the management of memory are shortly explained followed by the description of concepts for the usage of hardware virtualisation as a new strategy. Afterwards, certain problems related to hardware virtualisation are discussed. Possible solutions are elaborated and presented, which will be used for the case study in Part III.

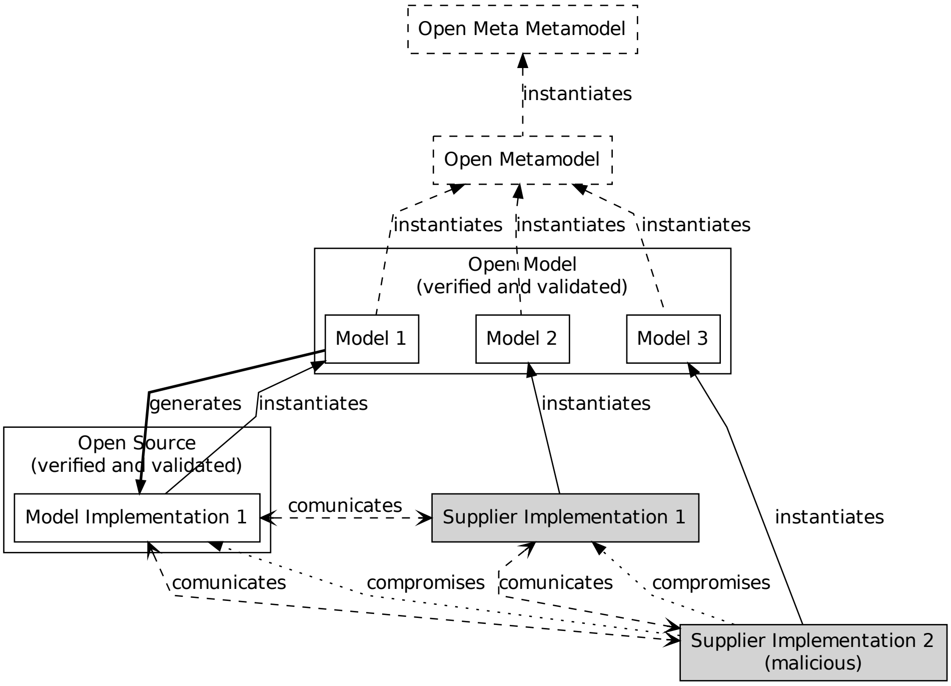


Figure 6.2.: Possible spoiling with open models

6.1. Memory Management

The security problem with third-party implementations in an open model concept cannot be completely avoided, but its influence to other components should be minimised. Because this problem is not very uncommon, there exist different strategies for keeping the security of the rest of the system uninfluenced from a faulty and malicious component.

There are different ways on how a component can influence another, but the access of the used memory / data is the most problematic one. This means if a component can read and write to the memory of another component it can change the data used for execution or even the execution itself. Therefore, the following different memory management strategies are described that reduce or avoid this problem in multiprocessing systems.

6.1.1. Partitioning

Partitioning means that the main memory of multiprocessing operating systems is divided in sequential partitions. Programs or processes can only be loaded and executed in such a partition and can only access the memory of their partition. Partitioning can generally be divided in:

Static partitioning: The memory M is divided in $n =$ const fixed sized partitions p_i, $\forall i :$ $\text{length}(p_i) = \text{const}$, $M = \{p_0, p_1, \ldots, p_n\}$ while $\text{length}(M) \geq \sum_{i=0}^{n} \text{length}(p_i)$ [77, pp. 353-453]. Static partitioning is very easy to implement, but it provides a very inefficient usage of the main memory. In a fixed environment, like in embedded systems, it can be used as a very simple but efficient memory management strategy due to the systems constrains.

For example, the ARINC specification 653P1-2 [3], which is applicable for avionics software standard interfaces, requires the usage of static partitioning for operating systems for the corresponding embedded systems.

Dynamic partitioning: The memory M is divided in $m = \{1, 2, 3, \dots\}$ partitions p_j with dynamic sizes: $\forall j$: length$(p_j) > 0$, $M = \{p_0, p_1, \dots, p_{m-1}\}$ while length$(M) \geq \sum_{j=0}^{m-1}$ length(p_j) [77, pp. 353-453]. Compared to static partitioning, dynamic partitioning provides a more efficient usage of the main memory but needs in general more computation time for its management.

6.1.2. Paging

For paging with virtual memory, the virtual memory V consists of the main memory M and the swap space S: $V = M \cup S$. This virtual memory V is divided in pages p_k, which all have the same size: $\forall k$: length$(p_k) = s$. Programs or processes are loaded in different but not necessarily sequential pages depending on their required memory. Pages of a program can be loaded on demand and do not need to be loaded completely at program start. A program can access its memory in a sequential manner because the physical memory addresses are mapped. Therefore, a program can only access pages that belong to it [77, pp. 353-453]. Paging with virtual memory provides an effective way of memory management for multiprocessing operating systems, but it is quite complex to implement.

6.1.3. Segmentation

For segmentation with virtual memory, also the virtual memory V consists of the main memory M and the swap space S: $V = M \cup S$. A program $e \in \{0, 1, 2, \dots, n\}$ is divided in certain number of segments $s_{e,l}$ ($l \in \{1, 2, 3, \dots m\}$), which have a dynamic size. This size can be influenced by the developer of a program or the used compiler for high-level languages: $\forall e : \forall l :$ length$(s_{e,l}) > 0$ while $V \geq \sum_{e=0}^{n} \sum_{l=1}^{m}$ length$(s_{e,l})$.

As in paging, the segments are not necessarily sequential and not all segments of a program must be loaded at its start. Also, the physical memory is mapped to provide each program a sequential access to its allocated memory / segments [77, pp. 353-453]. The main difference, compared with the advantages of paging, is that a developer can influence the size of the segments. This can also be used to define areas of shared data segments, where several programs have access to. Its implementation is even more complex as for paging.

6.1.4. Segmentation combined with Paging

To gain the advantages from paging and segmentation together, another more sophisticated strategy is to combine both strategies. This means that segmentation is used for programs while the segments $s_{e,l}$ of each program $e \in \{0, 1, 2, \dots, n\}$ are again divided in pages $p_{e,l,k}$ of a fixed size s: $\forall e : \forall l : s_{e,l} = \{p_{e,l,0}, p_{e,l,1}, \dots\}$, $\text{length}(s_e, l) > 0$, $V \geq \sum_{e=0}^{n} \sum_{l=1}^{m} \text{length}(s_{e,l})$, $\forall k : \text{length}(p_{e,l,k}) = s$ [77, pp. 353-453].

This strategy is used in most of the current multi-user-multiprocessing operating systems, like GNU/Linux.

6.2. Hardware Virtualisation

Section 6.1 described traditional strategies for avoiding the influence of faulty or malicious components on the rest of the system. This strategy must be integrated in the hardware[1] and the software and/or the operating system. That means for the usage of these strategies, it is important how the hardware platform and the operating system are chosen.

This is often problematic because open source software is typically not limited to a certain hardware or software platform. For a concrete open model software for industrial usage, this is almost mandatory because during its development not all potentially used hardware and software platforms can be known in advance. This leads to the need for another mechanism for memory protection that better fulfils the requirements of open model software in industrial applications.

The solution proposed in this work is the usage of hardware virtualisation [92]. Ideally, each supplier implementation or program should be executed in a separated virtual machine. The term virtual machine (VM) refers to the hardware virtualisation of any operating system representing a computer. Since a VM is completely separated from its host's operating system, the virtualised operating system does not have any direct access or knowledge about it. Programs executed in a VM can only communicate via a (virtual) network or a shared file system with programs on the host system or on other VMs.

The application of the hardware virtualisation concept to the initial problem of open models is shown in Figure 6.3. It holds the generated and certified model implementation and the two supplier implementations from Figure 6.2, but, in contrast, all supplier implementations are now locked in own virtual machines. This assures that the malicious implementation cannot compromise any other part of the software while a communication still is possible.

This hardware virtualisation concept fits the typical use cases of open model software in industrial applications because:

- Supplier implementations in a VM can never access memory of the host system or any other VM independent from the used operating system or memory management strategies and can only communicate with other components over defined and known channels.

- There exist several OSS / FLOSS implementations for hardware virtualisation, e.g., QEMU (KVM) [73], VirtualBox [66], Xen [97], and User Mode Linux [91].

Furthermore, the usage of hardware virtualisation provides additional advantages compared to traditional memory management strategies:

Adv.1: Hardware virtualisation protects the host operating system from any kind of negative and direct influence by failures, errors, or malicious behaviours of components executed

[1] typically CPU and MMU

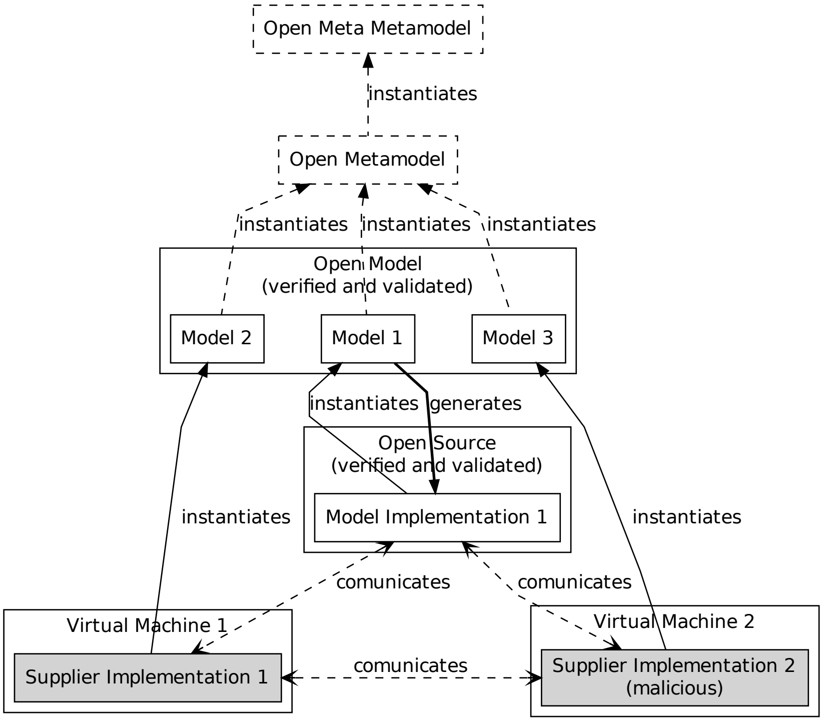


Figure 6.3.: Hardware virtualisation for open models

in a virtual machine. This means, in the worst case other, components only have to cope with the unavailability of other components in the corresponding virtual machine or false data delivered or sent by them.

Adv.2: Each hardware device and its software interfaces are protected from direct access from software components executed in a virtual machine. Several implementations of hardware virtualisation hypervisors already provide for certain bus-types, like USB, the possibility to configure the accessible devices in a certain virtual machine.

Adv.3: Virtualisation reduces dramatically the costs of partitioning analysis because software components executed in a virtual machine can only have an impact to other components over defined and known channels or interfaces.

Adv.4: Platform dependent source code can be executed in any (proprietary) operating system and does not influence the choice of the host operating system.

Adv.5: Costs for hardware could be reduced because one hardware platform can host several operating systems.

Adv.6: Virtual machines can be easily maintained or (re)installed because their hard disk images can simply be copied.

There exist also some disadvantages and unsolved problems:

Prob.1: Hardware virtualisation itself does not implicitly provide any mechanism to protect the bandwidth of real hardware resources, like the CPU(s) or the network interface(s) on the host system. A hypervisor process can be assigned to a certain CPU (in a multi-core system), which would at least protect the bandwidth of other cores. Otherwise, if a (faulty or malicious) implementation consumes too much bandwidth, it may influence any other executed implementation and limit its functionality.

Prob.2: Current open source hypervisors implementations are not too complex to be validated and certified [15], but which is currently for typical multi-user operating systems impossible because of their complexity. This means although the hypervisor can be assumed to be trustworthy that the operating system cannot. Therefore, the operating system itself may cause security problems.

Prob.3: Overhead for computation, main memory, and storage space is generated because each virtual machine needs physical memory, each virtualised operating system needs additional computation time, and each virtual operating system needs physical storage space.

Prob.4: Access to shared memory outside a VM is not possible even if it is desired. Communication with processes outside a VM can only be done by mechanisms also used for communication between different computers / hardware platforms that is typically a network or memory mapped files on a shared file system.

Prob.5: Direct access to hardware devices or their interfaces is only possible for certain bus-types.

Possible solutions for the problems Prob.1 till Prob.5 are introduced in the following subsections.

6.2.1. Bandwidth Protection

As the bandwidth protection of the host system's hardware resource is of high relevance, possible solutions for Prob.1 are introduced in this subsection. Unlimited bandwidth consumption is a possibility to influence or compromise the execution of other implementations over the boundaries of a VM.

Unfortunately, a general solution for any kind of hardware resource cannot currently be provided. Nevertheless, a possible solution is the usage of partitioning, which will be described exemplary for CPUs and network interfaces.

6.2.1.1. Process Scheduling

For a host system, a VM is mainly like a normal process, which is executed, but knowledge about certain processes in a VM is typically not available in the host. Therefore, partitioning for the process execution must be done on the host system for the hypervisor process.

As for static memory partitioning (Subsection 6.1.1), the ARINC 653P1-2 requires for embedded avionics systems the usage of static temporal partitioning [3]. This means a global sample or cycle time T_c is defined, which is divided in n temporal slots s_i with $i \in \{0, 1, 2, \dots, n-1\}$, $t_i = \text{length}(s_i)$, and $T_c = \sum_{i=0}^{n-1} t_i$. Each temporal slot s_i is exactly executed once in one cycle T_c for the duration of its execution time t_i while a certain process is assigned to each temporal slot s_i. Static temporal partitioning is very robust and and easy to implement but inflexible and mainly usable in embedded systems.

Process scheduling for multi-process operating systems with a not-fixed number of processes is far more complex because statical temporal partitioning cannot be used. One important requirement for the scheduler of the host system is that it behaves – at least partly – deterministically. Thus, it must be assured that no process starves [77, pp. 457-504]. It exist several scheduler strategies that are deterministic and avoid starving, but a fair-share scheduler [77, pp. 457-504] probably fits the requirements the best. It does not only avoid starving of processes completely and is deterministic but also provides the possibility to group processes while the execution of these groups is done in a "fair way". Accordingly, if supplier and open model implementations are in separated groups, a malicious or faulty supplier implementation could only influence other supplier implementation in their scheduling.

6.2.1.2. Network Traffic Scheduling

It is not avoidable that processes of the software have to communicate with processes on other computers or systems and also processes jailed in VMs need access to real network interfaces. Although this access is never direct, a malicious or faulty implementation could consume too much network bandwidth by, for example, sending plenty of user datagram protocol (UDP) packets. This could limit the ability of any other process on the local system – independent from if it is executed in a VM or not – to transfer data. This problem is mainly related to network traffic or bandwidth outgoing from the local system because network traffic incoming to the local system is generated by other computers on the network, which can be hardly influenced by the retrieving system.

Again, partitioning is a possible solution for this bandwidth problem. Similar to the scheduling of processes, this is a temporal scheduling, not of CPU time but of network bandwidth usage. Temporal slots are defined, which certain services / connections are assigned to. An example for static temporal partitioning is the Time Triggered Protocol (TTP) [49], which defines fixed temporal slots for each node on the bus.

The scheduling of network traffic is provided by hardware or by software. An industrial solution for hardware scheduling of network traffic is, for example, the Avionics Full-Duplex Ethernet (AFDX) [2], which is a real-time extension for Ethernet in avionic systems. Its main disadvantage is that additional hardware is needed.

Of course, there exist also open source solutions, like the traffic control (tc) tool for GNU/Linux, which is a part of the iproute suite [52]. With tc, it is possible to assign to each network interface a so-called queuing discipline. The default discipline is a simple first-in-first-out (FIFO) discipline, which does not protect the bandwidth of network interfaces. A possible solution for bandwith protection is the usage of the Stochastic Fairness Queuing (SFQ) discipline [52], which is a network traffic scheduler. Like the fair-share scheduler (Subsection 6.2.1.1), the SFQ schedules all network connections in a fair way that no connection can starve. Unfortunately, the term “stochastic” in its name is misleading because the scheduler behaviour is deterministic. It divides the network traffic on a certain interface into certain number n of FIFO queues. Network traffic is assigned to this n FIFO queues by a hash function, which is chosen in a stochastic way. The n FIFO queues are dequeued by a Round Robin [77, pp. 457-504] algorithm while here the quantum q [77, pp. 457-504] is not in time but in data size. This means that the SFQ does not provide temporal partitioning directly, but because time t, bandwidth b ,and data size s are related by $t = \frac{s}{b}$ with typically $b = \text{const}$ it can be called temporal partitioning anyway.

6.2.2. Minimal Host Operating System

A possible solution for Prob.2 could be the usage of an additional on-top security layer for the host operating system, such as SELinux for Linux. Its source code would be simple enough to be validated and certified and it supervises all security functions of the host operating system. The disadvantage of this solution is that an additional execution layer is added to the host operating system, which increase the complexity.

Therefore, it the usage of some kind of minimal operating system is proposed here. This operating system should mainly consist of the hypervisor implementation(s), a scheduler, which only switches between hypervisor processes and device handle routines, and device drivers and interfaces. An example for such a minimal host operating system for hardware virtualisation is LynxSecure [55]. Due to its reduced complexity, it can be validated and certified and then the host operating system and hypervisors can be assumed to be secure.

6.2.3. Hardware Assisted Virtualisation

This section describes a possible solution for Prob.3 with the usage of hardware assisted virtualisation. Most modern x86 compatible CPUs provide a support for virtualisation. Hence, they offer two modes: Host and guest. The host mode is the “normal” mode of a CPU while the guest mode is used for VMs. In the guest mode, the virtualised operating system still sees the four privilege levels of x86 CPUs in protected mode. Additionally, in guest mode, a virtualised operating system can be executed in the highest privilege level. This does not mean a security risk because in guest mode all privileged executions are trapped and the control returned to the hypervisor, which handles then the execution in the host mode. The advantage of the two modes is that the hypervisor does not need to provide a software implemented virtual CPU. This can drastically reduce the computational overhead [1].

The usage of host and guest mode still leaves an overhead problem with the memory management inside a VM: The memory management unit (MMU) of a CPU, which is used for paging and segmentation [77, pp. 353-453], has to be provided by the hypervisor as software implementation. For this problem also exists a hardware assisted solution because a virtual MMU can also be provided by x86 CPUs by a feature known as Rapid Virtualisation Indexing (RVI) or Extended Page Table (EPT) [1]. The combination of both CPU features in a hypervisor may provide the best performance or rather overhead reduction that is currently available.

6.2.4. Process Communication

The limitation of the communication to network mechanisms (Prob.4) could be directly integrated in the software design to reduce the drawbacks. For example, CORBA [41] can be used for user or supplier implementations. Simply and generally expressed, the typical CORBA development case is that an interface [41] is defined, which has to be implemented in an object-oriented programming language, e.g. C++ or Java. The implementation is called Servant [41] and is a software class. Additionally, CORBA provides a so-called Proxy [41], which is generated from the interface and can be used to access the Servant. Figure 6.4 shows a simplified example of a CORBA development case.

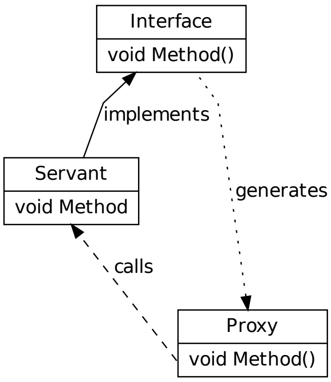


Figure 6.4.: Simple CORBA usage example

This Servant-Proxy mechanism could be directly included in the software model by, for example, defining each part of the open model (from Figure 6.3) that could or should be (re)implemented as supplier parts as CORBA interface. Consequently, each supplier implementation would be done by implementing a CORBA Servant for a certain interface. Besides the fact that the usage of CORBA can solve communication problems in a distributed software architecture with VMs, it also provides additional advantages:

- CORBA is (quite) platform-independent because there exist several implementations for various object-oriented programming and scripting languages under different operating systems.
- CORBA is an industrial standard for middle-ware, which supports the processes of verification and validation.
- CORBA supports in general the usage of distributed systems.

The presented example for open models with hardware virtualisation and CORBA was very simplified because all software models normally do not consist of only one interface, but it is for example possible to build a complex model by using several interfaces.

An alternative for the integration of CORBA could be the D-Bus [31] system, which is quite popular in most Linux [47] distributions. Like CORBA, D-Bus implementations are available for several different platforms and supports platform crossing communication. It also uses an object model and differs only slightly in the terminology since the concrete interface implementations are called adaptors. The main difference, compared to CORBA, is that the bus, the interconnection between objects and proxies, is the central component. In CORBA, there exist different possibilities for the proxy to servant connection since always a so-called Interoperable Object Reference (IOR) [41] of the servant is needed. Only one possible way is to exchange the IOR via a CORBA name server [41], which is similar to the bus mechanism of D-Bus. Interfaces are defined for D-Bus as XML, from which a class for the adaptor implementation and for the proxy is generated [31].

D-Bus mainly provides the same advantages as CORBA besides that it is not that frequently used in industrial applications but more on Linux (desktop) systems. Especially, when executed on a Linux system, D-Bus adds the advantage that system wide bus [31] is mostly available and session related buses can easily be created.

6.2.5. Hardware Device Access

As for the inter-processes communication (Subsection 6.2.4), also the access to certain hardware components could be solved by the employment of a middle-ware, like CORBA or D-Bus. Accordingly, a general interface for each hardware component type would have to be defined, which must be implemented by any hardware supplier. Unfortunately, this platform-specification adoptions would have be executed directly on the target platform to guarantee access to the related hardware component, which is in conflict with the idea of using hardware virtualisation.

Hence, the usage of a hypervisor that provides configurable access to certain hardware components renders the concept of hardware device access through a middle-ware obsolete and accordingly is preferable.

6.2.6. Hardware Virtualisation on Certified Supplier Hardware

A typical usage case for an open model software is that a supplier uses the open model and open source elements, adds certain own supplier implementations, and compiles it for a certain (own) hardware platform. Often, this hardware is already certified itself and holds components that provide or support management mechanisms, like temporal and spacial partitioning (see Subsection 6.1.1 and Subsection 6.2.1.1).

One reason for additional supplier implementations was that suppliers or vendors often do not use the enterprise model of open source (or open model) and prefer to keep their software closed source and proprietary. Another reason is that not every existing application case can be covered in advance during the development of the open model software.

The primary motivation for the usage of hardware virtualisation was the protection of verified, validated, and certified software from possibly malicious and faulty ones. Certified supplier hardware normally covers this task, but hardware virtualisation running on certified hardware still provides the advantages Adv.1 to Adv.6 anyway and therefore still is worth to be used.

6.2.7. Hypervisor Requirements

Taking the before mentioned advantages Adv.1 to Adv.6 and problems Prob.1 to Prob.5 into account, the following requirements must be fulfilled for the usage of hardware virtualisation for security in open source and open model software:

Req.1: The minimal host operating system (Subsection 6.2.2) must be open source to enable its validation.

Req.2: The hypervisor must be open source to enable its validation.

Req.3: Temporal partitioning or a deterministic scheduler algorithm (Subsection 6.2.1.1) must be used for process scheduling in the minimal host operating system.

Req.4: Temporal partitioning or a deterministic scheduler algorithm (Subsection 6.2.1.1) must be used for network connections / interfaces in the minimal host operating system.

6.3. Conclusion

In this chapter, the issues related to security raised if software realised by DSM is distributed and developed under the principles of OSS / FLOSS were analysed and discussed. Besides the traditional strategies for memory management, hardware virtualisation was presented as a new and more general platform independent approach. A list of advantages and requirements for the hypervisor integration was elaborated. Additionally, two concrete possibilities for the realisation of required interprocess communication were presented.

Part III.

openETCS Case Study

7 openETCS Meta Model

The meta model is the first instance to be developed in every DSL development cycle. In this work, the selected meta meta model is GOPPRR, as already explained in Chapter 4. Figure 7.1 repeats the concrete DSL open model instances from Figure 6.1 and adds the concrete instances for the openETCS case study.

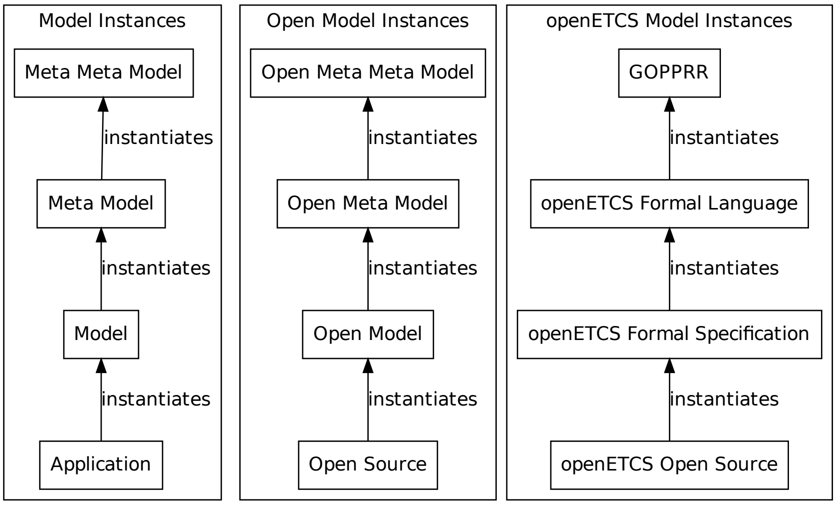


Figure 7.1.: DSM instances for the openETCS case study

The openETCS[1] meta model corresponds to the formal specification language. One of the aims of this work in relation to the formal specification language is to provide more abstraction than the specification documents of the ETCS SRS. The openETCS meta model should be comprehensible for a domain expert[2] and additionally should reflect the structure of the SRS, as

[1] The name openETCS was chosen for the case study. Although this work follows similar goals as the German Railways initiative, it is not an immediate part of it.

[2] ATP or ETCS expert

far as possible. Furthermore, the meta model must be formal [78, Ch. 11]. Hence, the complete concrete syntax and the static semantics must be completely defined. A formal representation of the dynamic semantics can support this feature of the meta model.

This chapter starts with the selection of the used parts or rather the subset[3] of the ETCS SRS for the case study because this must already be taken into account during the development of the openETCS meta model. The concrete syntax [46, p. 70] of the meta model is explained by the GOPPRR formalisms for sub-graphs, graph bindings, and type properties from Section 4.1, which mainly corresponds to the concrete syntax used in MetaEdit+. Furthermore, the static semantics [46, pp. 69-70] are defined by using the GOPPRR (C++) abstract syntax with OCL, which is a set of constraints for any instance of the openETCS meta model. Afterwards, the dynamic semantics [46, pp. 69-70], which means the behavioural interpretation of the meta model, is discussed followed by a mathematical model for the dynamic semantics.

7.1. Selection of Specification Subset

Due to the complexity of the ETCS specification, even of the Subset-026 of the SRS, the modelling of the complete SRS would no be realisable in the scope of this work. Therefore, a certain and small enough subset of the SRS was selected for the case study:

- EVC implementation (mainly Subset-026)
  - ETCS Application Levels: 0, 1
  - ETCS Modes:
    - No Power (NP)
    - Stand By (SB)
    - System Failure (SF)
    - Isolation (IS)
    - Trip (TR)
    - Post Trip (PT)
    - Unfitted (UN)
    - Staff Responsible (SR)
    - Full Supervision (FS)

This reduced subset of the SRS should directly influence the complexity of the later introduced model instance. On the other hand, the limitation to the Application Levels 0 and 1 means that only Eurobalises are used for track-to-train communication. As device types are also included in the meta model, which is explained in the following section, this already reduces the complexity for the meta modelling process.

The aim of this specification subset and the corresponding DSM instances is to provide a minimal executable case study for the EVC that proves the feasibility of the open model concept. Necessary extensions to all DSM instances needed for covering all SRS parts should not be a reduction of this prove. Hence, those extensions were also taken into account during the DSL development but are not (yet) implemented.

[3] Subset does not refer here to the term used in the ETCS SRS for combining several chapters, for example, Subset-026.

The developed meta model is based on the ETCS SRS version 2.3.0, which was already referred to in the general ETCS introduction in Section 2.2. Accordingly, all following instances of the openETCS DSL in this case study are also based on this SRS version.

7.2. Concrete Syntax for Graph Types and Sub-Graphs

To begin the meta model description with an overview, Figure 7.2 introduces all graph types and their connection by explosions and decompositions. For clarification, all graph types are denoted by a "g" as prefix and objects by an "o". In the real meta model, those prefixes do not exist. The general meaning of graph types is explained in this section while the certain binding syntax and the used object types for each graph type are explained in the next section.

gEVCStateMachine graph type The ETCS standard specifies EVC behaviour according to operational Modes. In each Mode, a well-defined collection of functions is active. Transitions between Modes are specified in the standard as the so-called transition tables while related descriptions of guard conditions are presented in structured natural language [88, pp. 37-40].

gMainFunctionBlock, gSubFunctionBlock, and gEmbeddedStateMachine graph types The specification of ETCS Application Levels applicable in a certain Mode is realised in the openETCS meta model by explosions from Modes (object type oMode) to graphs of type gMainFunctionBlock: One for each level applicable in the respective Mode. A gMainFunctionBlock graph defines the EVC operations to be executed in a given mode and ETCS Application Level [88, 90, 89]. To determine the ETCS Application Level of a function block, the gMainFunctionBlock graph type holds an oApplicationLevelType object in its property list. This object carries the level identification of each gMainFunctionBlock instance associated with an ETCS mode via explosion.

Function blocks are used to model data flow between objects that can be sources, like sensors or actuators, but also to model variables describing internal states. Experience with system modelling has shown that it is often necessary to complement data flow specifications by control flow descriptions in order to model the complete system behaviour [8]. Therefore, the oEmbeddedStateMachine object type in the gMainFunctionBlock graph type has a decomposition into state machines (gEmbeddedStateMachine graph type) embedded within the data flow. Conversely, state machine control states may be decomposed again into gSubFunctionBlock, which model the active behaviour while residing in the control state. It should be noted that this recursive relationship allows to specify hierarchic control states as used in the so-called "OR-states" of statecharts [36].

Another special object is the oSubFunction object within the gMainFunctionBlock graph type. Its purpose is to structure certain functionality into sub-graphs for better graphical clarity and re-usability. For this reason, it has a decomposition to the gSubFunctionBlock graph type.

The gSubFunctionBlock type differs from gMainFunctionBlock mainly in the lack of the assignment to a certain Application Level by a property. This enables the re-usability of certain (sub) functionality in several Modes and Application Levels. Therefore, object types related to

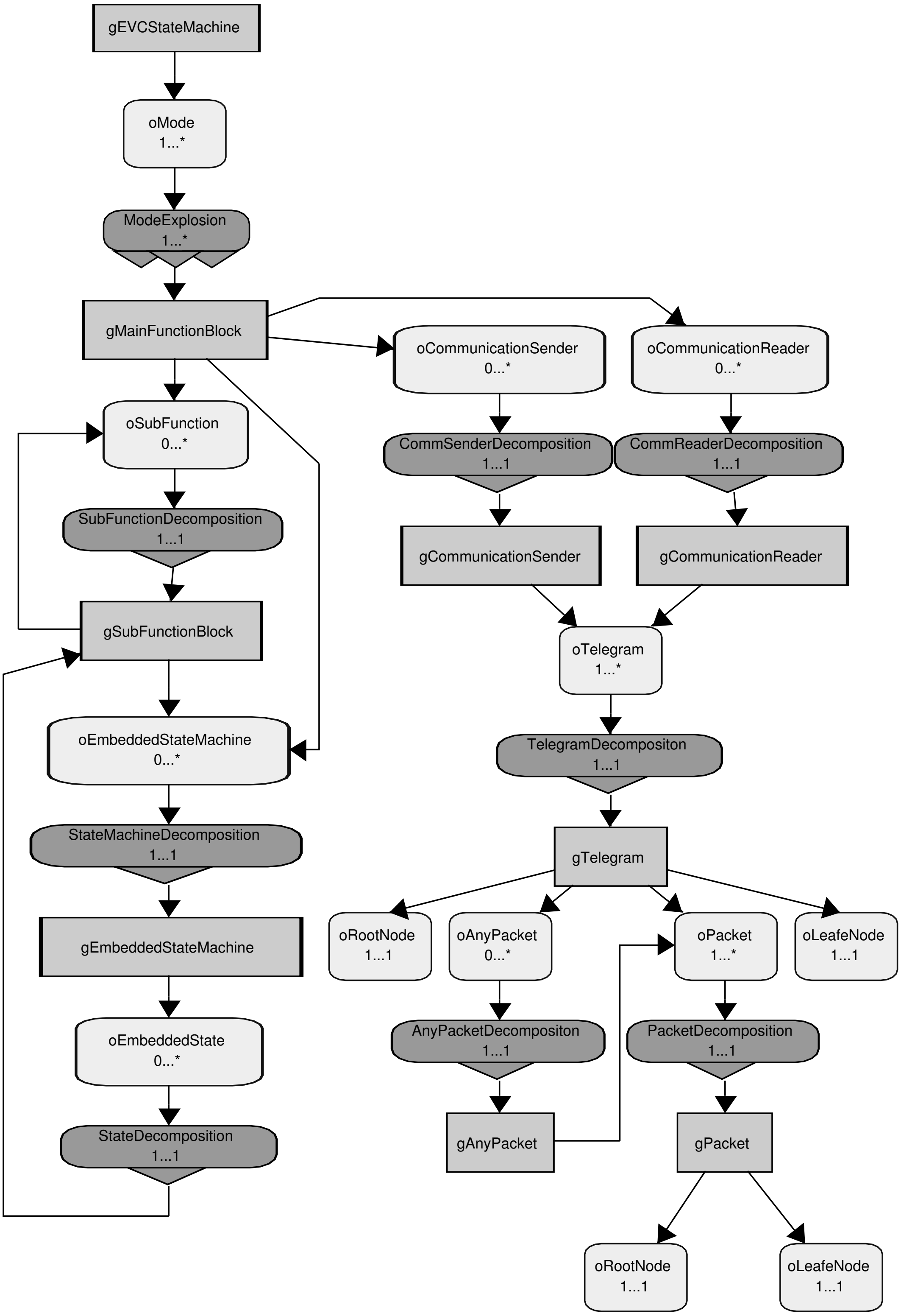


Figure 7.2.: openETCS meta model graphs, sub-graphs, and object occurrences

hardware elements, like sensors and actuators, are only directly usable in a gMainFunctionBlock graph. Access from gSubFunctionBlock is only possible via model variables. Furthermore, gMainFunctionBlock has the property FailureGuard, which holds a oModeGuard (to be explained in Subsection 7.3.2) object for defining the transition to be used in case of an error in the EVC. If this property is not set, no failure state is defined for the corresponding ETCS Mode and Application Level.

Apart from describing Mode- and Level-dependent functionality, the gMainFunctionBlock and gSubFunctionBlock graph types are also used to specify behaviour at the interface between train engine drivers and the EVC.

CommunicationSender and CommunicationReader graph types These graph types are used to specify information exchange between train and track side components [87]. The data objects used in train-to-track (oCommunicationSender) and track-to-train (oCommunicationSender) communication are of type oTelegram.

gTelegram and gPacket graph types The ETCS SRS defines telegrams as the most complex data structure for train-to-track communication via balises. Telegrams are composed of packets [85]. According to the SRS, an oPacket is a data element in a telegram [85]. It is modelled by a sequence of ETCS (language) variables, which are the atomic data items of ETCS communication between train and track-side. The oAnyPacket object type is not directly derived from the SRS. It is used to provide patterns, selections, or sub-structures of several packets inside telegrams.

gAnyGraph This is a very simple graph type, which only holds packet object instances to define a pattern for oAnyPacket objects.

7.3. Concrete Syntax for Graph Bindings

Section 7.2 explained the general purpose of all graph types of the openETCS meta model and their interconnection. The concrete syntax of object bindings in each graph type is defined and explained in the following subsections.

7.3.1. gEVCStateMachine Graph Type

The gEVCStateMachine graph type is applied to specify EVC Mode transitions, which are labelled by guard conditions, as specified in Figure 7.3. Each mode transition has two attributes:

1. A ModeTransitionGuard object, which represents a Boolean trigger condition for a given transition between two EVC Modes. Those conditions can be found in the modes and transitions tables of the ETCS specification [88, pp. 38-40].
2. A ModeTransitionPriority, which is a non-negative integral number defining a priority for situations where several guards of transitions emanating from the current mode evaluate to true.

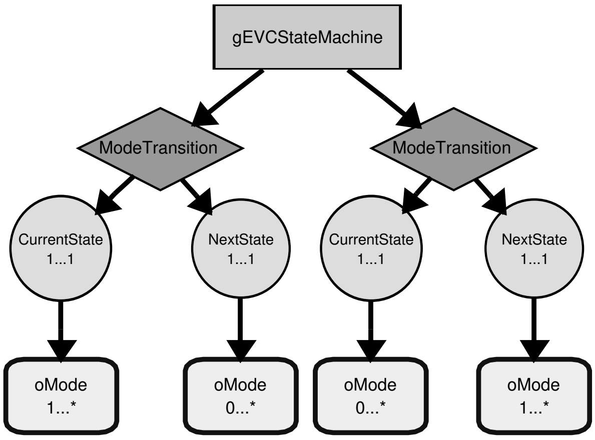


Figure 7.3.: gEVCStateMachine bindings

It must be observed that the gEVCStateMachine graph itself does not provide any syntax to define temporal behaviour, but the evaluation of its guard objects is manipulated in gMainFunctionBlock graphs where causal and time-dependent behaviour can be described, as shown below.

A certain question arises if the gained abstraction for modes and their transitions in the openETCS meta model is directly evaluated here. A representation of 8 modes and 30 guard conditions for transitions would result in a huge graph, which would be difficult to read and therefore would not provide good abstraction. On the other hand, the transition table in [88, p. 40] already is somehow in an optimized form. Fortunately, MetaEdit+ not only can instantiate graph types directly with objects and bindings but also in matrix representation [56].

In contrary to the transition table, the modes are listed on the row and column index of the matrix while the transitions or rather the ModeTransition relationships are located in the cells. Any transition between a state i to state j can be found then in the cell (i, j). An example taken from [88, p. 40] is shown in Figure 7.4. The EVC can be switched from the mode NP to SB under the condition / guard "4" and from SB to NP under condition "29". Both transitions have the priority 2. The result of converting a gEVCStateMachine to a matrix instance is presented in Figure 7.5. A more concrete example for a gEVCStateMachine matrix is presented in Chapter 10.

NP	<29 -p2-	⋯
4> -p2-	SB	⋯
⋮	⋮	⋱

Figure 7.4.: Simple example from SRS transition table

	NP	SB	⋯
NP		c4-p2	⋯
SB	c29-p2		⋯
⋮	⋮	⋮	⋱

Figure 7.5.: Simple example for a gEVCStateMachine matrix

7.3.2. gMainFunctionBlock and gSubFunctionBlock Graph Types

Main function blocks are used to model transformations of the data flow performed by the EVC in a specific Mode and ETCS Application Level (Figure 7.6). Since these transformations may be quite extensive, gMainFunctionBlock graphs may contain oSubFunction objects allowing top-down decomposition into gSubFunctionBlock graphs. The behavioural interpretation is the same as if all details occurring in the lower-level function block had been presented already in the original function block without referencing an oSubFunction object.

Objects in the gMainFunctionBlock and gSubFunctionBlock graph type can be related by the binary DataFlow relationships connecting objects by means of a DataOutput role (sending object) and a DataInput role (receiving object). The possibility to connect general objects by means of data flows and ports enables the specification of feed-forward or feed-back structures [30], which are typically used in loop controllers. The GOPPRR port concept is used to avoid modelling errors and facilitates the checking of static model semantics with respect to interface consistency.

Compared to other graph types, both function block types use the majority of object types in the meta model because they mainly describe the complete functionality in a certain ETCS Application Level. Those object types can be divided in four categories:

sources Object types that only provide outputs. Those are mostly related to sensor devices, like odometers. All types in this category use oFunctionBlockOut as super type.

sinks Object types that only have inputs. Those are mostly related to actuator devices, like service brakes. All types in this category use oFunctionBlockIn as super type.

transformations Object types with inputs and outputs. Those apply a certain transformation on the inputs and give the results on the output. For example, the computation of a braking-curve or the simple calculation of a sum. All types in this category use oFunctionBlockIn as super type.

storages Object types for storing and restoring data values. No super type is used for storages.

specials Object types that do not have any inputs and outputs and are not a direct part of a data flow. No super type is used for special object types.

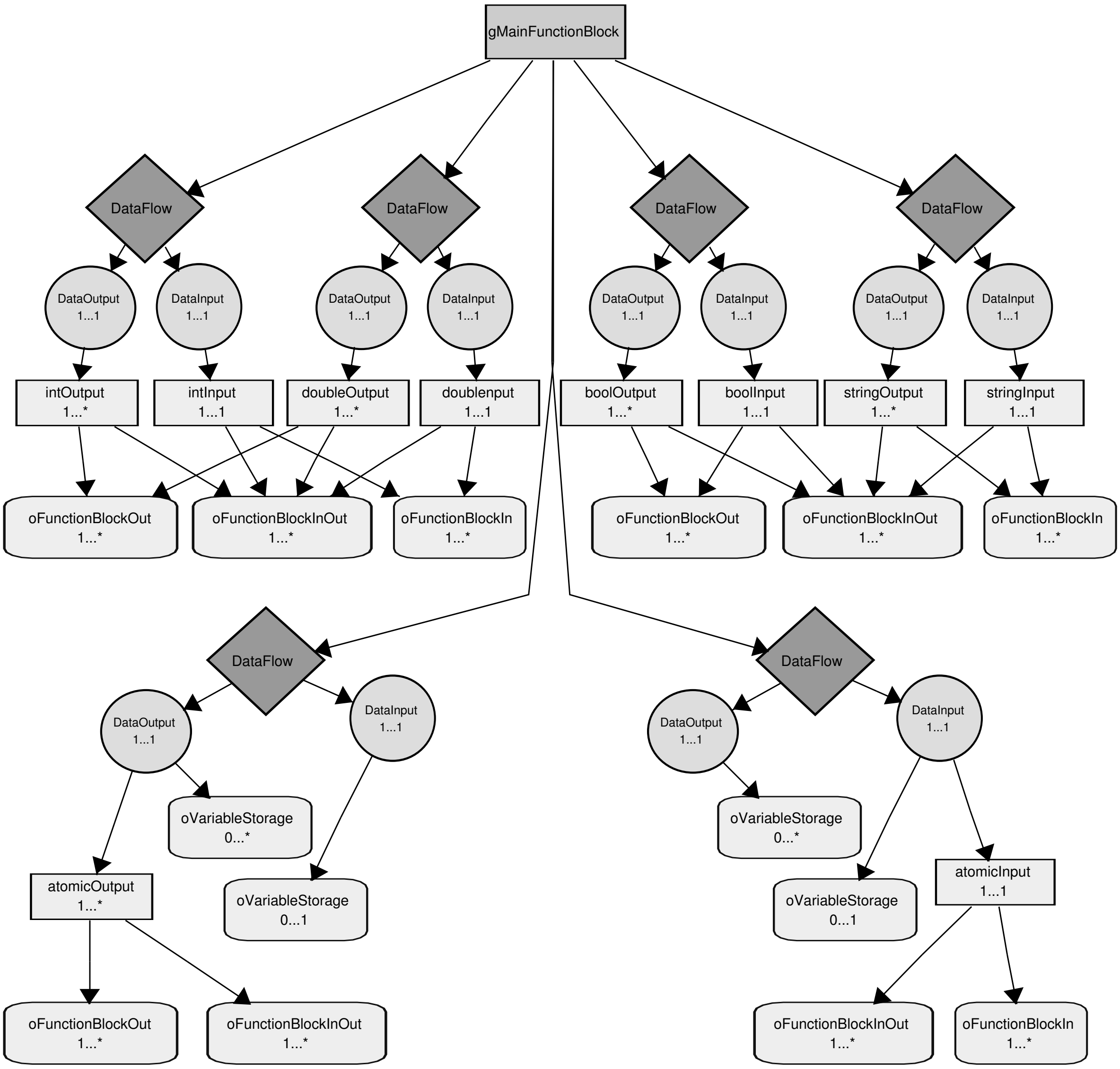


Figure 7.6.: gMainFunctionBlock binding syntax

The data flow between those object type categories is in general not type-less, which ensures data connections only between input and output of the same type. Data flow types are determined by the port types of input and output, which are:

intOutput for integer outputs

intInput for integer inputs

doubleOutput for double precision floating point outputs

doubleInput for double precision floating point inputs

boolOutput for Boolean outputs

boolInput for Boolean inputs

stringOuput for string outputs

stringInput for string inputs

All output port types use atomicOutput as super type and all input types use atomicInput.

Figure 7.6 presents the bindings for the gMainFunctionBlock graph type using super types for object and port types and Table 7.1 introduces the concrete object types. A documentation of all types in Figure 7.6 would be an extensive diagram with low level of clarity.

Before continuing with the detailed explanation of object types, it must be noted that in some minor cases the double data type has to be used as an array type. This means that the data flow does not only transport single double values but vectors of a certain size. The array type is only available for double data flows and rarely used.

All object types introduced in the table are shortly explained below and are grouped by the category.

Source

oOdometer is an interface to an odometer device as sensor hardware component. It provides the current position and the current speed as double values.

oCommunicationReader defines an abstract interface for certain telegrams or messages for track-to-train communication [85, 87]. Its Boolean output switches to true if a corresponding telegram / message was received.

oDMIInput is a certain element on the Driver Machine Interface (DMI) [84] for receiving data from the driver. For each possible data type, a port is available, but only one port can be used simultaneously. The used output port type determines the type of data, which can be entered by the driver. Additionally it has a Boolean input port to set if the element

[4] Boolean
[5] integer
[6] double
[7] string
[8] array

		Output				Input			
		Port Types				Port Types			
Category	Object Type	b[4]	i[5]	d[6]	s[7]	b[4]	i[5]	d[6]	s[7]
source	oOdometer			1					
	oCommunicationReader	1							
	oDMIInput	2	1	1	1	1			
	oEnteredTrigger	1							
sink	oServiceBrake							1	
	oEmegergencyBrake					1			
	oCommunicationSender					1			
	oDMIOutput					2	1	1	1
	oApplicationLevelType					1			
	oModeGuard					1			
	oStateGuard					1			
transformation	oAND	1				2			
	oOR	1				2			
	oXOR	1				2			
	oNOT	1				1			
	oSum			1				3	
	oSubstraction			1				2	
	oDivision			1				2	
	oMultiplication			1				2	
	oDoubleEqual	1						2	
	oIntEqual	1					2		
	oStringEqual	1							2
	oDoubleGreater	1						2	
	oDoubleGreaterOrEqual	1						2	
	oIntGreater	1					2		
	oDoubleArrayAcessor			1			1	1[8]	
	oBoolGate	1				2			
	oDoubleGate	1		1				1	
	oStringGate				1	1			1
	oBoolSwitch	1				3			
	oDoubleSwitch			1		1		2	
	oStringSwitch				1	1			3
	oEmbeddedStateMachine				1	1			
	oBrakingToTargetSpeed	4		1				5 + 2[8]	
	oCeilingSpeedControl	4						4 + 2[8]	
special	oVariableStorage								
	oSubFunction								
	oNote								

Table 7.1.: Concrete object types in function block graph types

is currently visible or not. A further Boolean output port switches to true if the driver entered data in the previous computation cycle.

oEnteredTrigger has a true output only in the first computation cycle after switching to a new ETCS Mode and/or an new Application Level.

Sink

oServiceBrake is an interface to a service brake hardware component. The double input can be set from 0 to 100%, which corresponds to the pressure level of the brake system.

oEmergencyBrake is an interface to an emergency brake hardware component. If its Boolean input is set to true, the emergency brake system is activated.

oCommunicationSender defines an abstract interface for certain telegrams or messages for train-to-track communication [85, 87]. If its Boolean input is set to true, the corresponding telegram / message structure is sent.

oDMIOutput is a certain element on the Driver Machine Interface (DMI) [84] for displaying data to the driver. For each possible data type, a port is available, but only one port can be used simultaneously. Additionally, it has an Boolean input port to set if the element is currently visible or not.

oApplicationLevelType is used to switch between ETCS Application Levels. If its Boolean input is set to true, it switched to the corresponding Application Level defined by the property ApplicationLevelName in the same ETCS Mode.

oModeGuard is, similar to oApplicationLevelType, used to switch between different ETCS Modes. In contrast to oApplicationLevelType, the next Mode is determined by the parent gEVCStateMachine graph. Each ModeTransition relationship holds an oModeGuard object property that must correspond to an instance[9] in an explosion of the related current state oMode object. This means if the Boolean input of an oModeGuard object is set to true the corresponding ModeTransition relation is passed to the new state. The new mode is then executed in the same Application Level.

oStateGuard is the same as the oModeGuard type but is used for control flows. This means in decompositions of oEmbeddedState objects used in gEmbeddedStateMachine graph types, which will be explained in more detail in the section describing the binding syntax of the gEmbeddedStateMachine graph.

Transformation In contrast to sources and sinks, most transitional object types provide basic mathematical operations. Their functionality can be simply derived from

[9] by reference

their name:

- oAND, oOR, oXOR, and oNOT provide Boolean operations.
- oSum, oSubstraction, oDivision, and oMultiplication provide arithmetical calculations.
- oDoubleEqual, oIntEqual, oStringEqual, oDoubleGreater, oDoubleGreaterOrEqual, and oIntGreater compare data flows of the corresponding types.

Nevertheless, there exist also some object types which functionality must be explained more accurate:

oDoubleArrayAcessor accesses a certain element of a double array input by using an integer input as index. The element is used as double output.

oBoolGate controls a Boolean data flow by using another Boolean input. If this control input is true, then the Boolean input is directly transferred to the output, else nothing is output. This is the same functionality of inhibit gates elements in Fault Tree Analysis [78, pp. 43-50].

oDoubleGate provides the same functionality as oBoolGate but for double data flows.

oStringGate provides the same functionality as oBoolGate but for string data flows.

oBoolSwitch is similar to oBoolGate, but instead of only copying or not output at all it switches between two Boolean inputs. If the Boolean control input is true, the first Boolean input is transferred to the output, else the second input is transferred.

oDoubleSwitch has the same functionality as oBoolSwitch but for double data flows.

oStringSwitch has the same functionality as oBoolSwitch but for string data flows.

oEmbeddedStateMachine is used to define control flows by a decomposition to a gEmbeddedStateMachine graph. It has one Boolean input to start the underlying control flow. The string output delivers the literal name of the current active state if the control was started and did not finish yet.

oBrakingToTargetSpeed defines the calculation of a braking-curve for a certain target speed as double input. It uses the gradient distance and gradient values of the track as double array input. The current speed and distance, the distance to the new speed limit, the new speed limit, and the adhesion factor for the track are taken as double inputs. It provides a Boolean output for the emergency brakes and a double output for the service brakes. Additionally, it has Boolean outputs to inform about exceeded speed limit and applied service and emergency brakes.

oCeilingSpeedControl provides a supervision for a certain ceiling speed of the train. It has the same inputs and outputs as oBrakingToTargetSpeed but does not use the distance to the speed limit input because it provides no braking-curve calculation.

Special

oVariableStorage stores data flow values of any kind. It can be used to copy values from a gMainFunctionBlock to a gSubFunctionBlock, and vice versa, or also between different gSubFunctionBlock instances. This means also between different ETCS Modes and Application Levels. The only functionality it provides is the conversion or rather casting [79] between the different data flow types. For example, it can store a Boolean output and can then be used as integer input. Therefore, it does not have any input or output ports and the data flow type is always determined by the input or output port type of connected objects. A data flow between two oVariableStorage objects is possible but is type-less and is declared as bit data flow.

oSubFunction only defines a sub data flow to be used, which is defined as decomposition to gSubFunctionBlock graph. It does not have any input or output and does not participate in any data flow directly. It could be interpreted as an include-statement [79] in high-level programming languages.

oNote only is used for documentation purposes in a model to define comments. Any object instances of oNote do not influence the semantics of the model and thus neither the finally generated source code.

Difference between gMainFunctionBlock and gSubFunctionBlock Until now both data flow graph types gMainFunctionBlock and gSubFunctionBlock were explained equally, but gMainFunctionBlock is only used as explosion of oMode objects in the gEVCStateMachine graph type, as already defined in Section 7.2. Additionally, it holds a property for defining the ETCS Application Level, which it is used in. gSubFunctionBlock graphs are not limited to a certain Application Level and can be reused for any Mode or Level.

Therefore, gSubFunctionBlock graph may not use any instance of oModeGuard object types because those are always directly related to the parent oMode objects. Neither, the direct access to hardware interface should be possible to ensure graphical clarity about used hardware on the first data flow level for each ETCS Mode and Application Level. In general, this means that mainly all sub-types of oFunctionBlockIn and oFunctionBlockOut are not available in gSubFunctionBlock graphs. The resulting bindings syntax is shown in Figure 7.7. The oStateGuard is the only object type, which is used in gSubFunctionBlock but not in gMainFunctionBlock because it is always related to oEmbeddedState object, which gMainFunctionBlock never can be a decomposition of. In its syntax definition in Figure 7.6, it is available for gMainFunctionBlock because it is a sub-type of oFunctionBlockIn. To avoid its usage there, a constraint is used, which will be defined later by the static semantics.

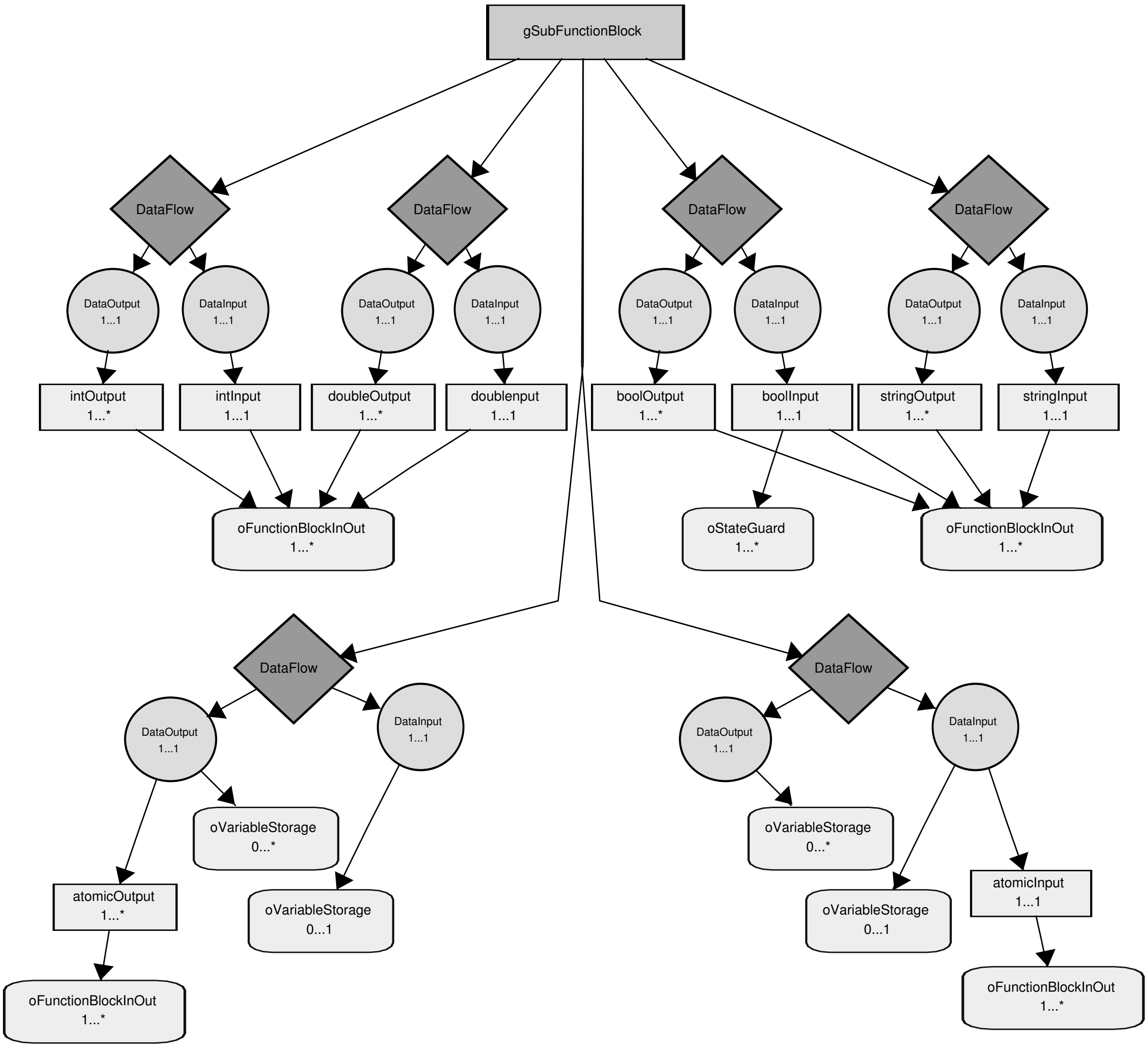


Figure 7.7.: gSubFunctionBlock binding syntax

7.3.3. gEmbeddedStateMachine Graph Type

The gEmbeddedStateMachine graph type syntax is similar to the gEVCStateMachine, but is also used, as the gMainFunctionBlock and gSubFunctionBlock graph types, to define the functionality or rather the behaviour in a certain EVC Mode. In contrast to those graph types that are used to model data flows, this graph type is used to specify control flows. Experience with system modelling shows that the specification of data flows alone is insufficient in most cases because a separate means for modelling control is needed [8]. For example, to model communication protocols. The syntax definition of the gEmbeddedStateMachine graph type is shown in Figure 7.8, which only differs from Figure 7.3 in the object type used for states. Of course, it is also meaningful to present instances of gEmbeddedStateMachine as matrix as for gEVCStateMachine, but for a smaller amount of states the “normal” graphical representation might provide the same or even better abstraction. Therefore, the representation kind for this graph type can be chosen accordingly to the concrete state machine to be modelled.

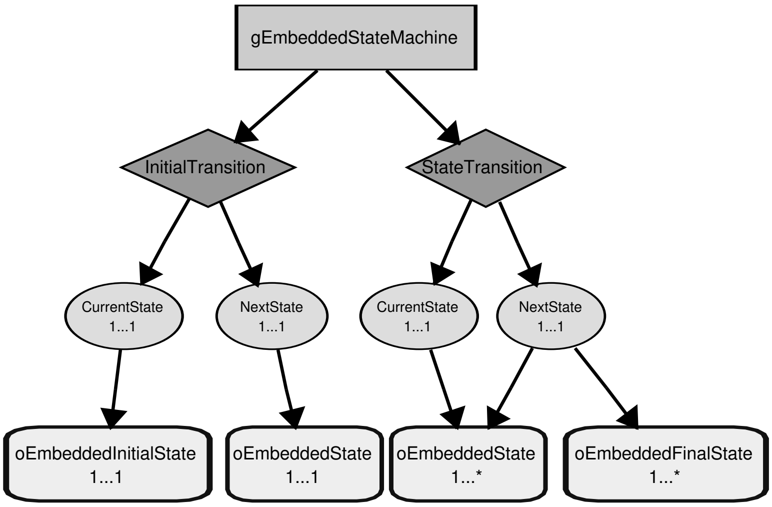


Figure 7.8.: gEmbeddedStateMachine binding syntax

7.3.4. gCommunicationSender Graph Type

The purpose of gCommunicationSender graphs is to compose telegrams with data from gFunctionBlock graphs and to transmit them via a communication device. Currently, for this case study, only Eurobalises [84] are used. The object types they operate on are oTelegram, oPacket, oVariableInstace, oVariableStorage, and the different devices types for transmission. To model or specify operations on an oTelegram, an oVariableStorage object can be connected by a DataFlow relationship with an oVariableInstance object. Composition of gPacket objects by oVariableInstance objects is modelled by graphical containment. The same is done for the composition of oTelegram objects. Composed oTelegram objects can also be connected by a DataFlow relationship with a certain sending device to specify the transmission of a telegram.

The syntax is presented in Figure 7.9. The gCommunicationSender graph type also uses a data flow syntax, but this is reduced to the flow from a composed telegram to a sending device – a sub-type instance of oSenderDevice – and from oVariableStorage objects to oVariableInstance objects. The oVariableInstance object type represents the atomic data element used in balise communication [85]. The concrete structuring of the large elements will be explained in sections about the gTelegram and gPacket graph types.

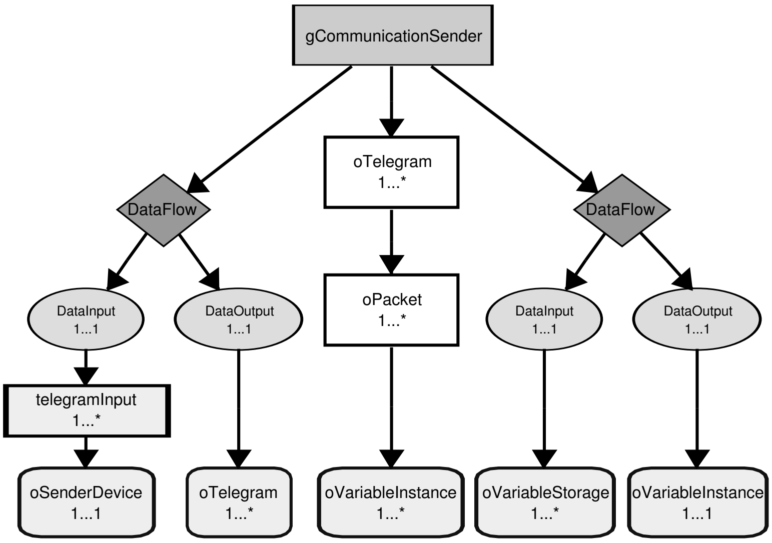


Figure 7.9.: gCommunicationSender binding syntax

In this graph type, only those modelled structures must be used. To compose a certain oTelegram object, it has to be modelled that it graphically includes one or more oPacket objects that are also part of the gTelegram graph decomposition of it. The same must be done for each oVariableInstace object in the gPacket graph decompositions of the oPacket objects. The data flow from an oVariableStorage to an oVariableInstace object can be interpreted as filling of the telegram elements with values from the active data flows. This data flow is also of the pseudo type bit because telegrams are transferred only digital.

7.3.5. gCommunicationReader Graph Type

The gCommunicationReader graph type is used for the opposite purpose of the gCommunicationSender to get values from received telegrams by track-to-train communication. Therefore, only the direction of the data flows is changed: From the receiving device – a sub type of oReaderDevice – to a oTelegram object and from the included oVariableInstance objects to oVariableInstace objects. The binding syntax is shown in Figure 7.10.

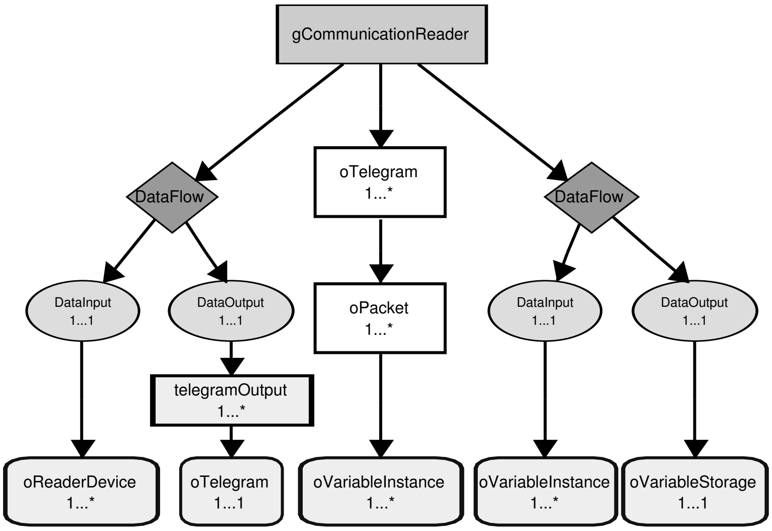


Figure 7.10.: gCommunicationReader binding syntax

7.3.6. gTelegram Graph Type

To specify an ETCS telegram [85], a certain order of ETCS packets has to be modelled. To provide the possibility to define a telegram not only by exact one order but rather by a pattern, the additional object type oAnyPacket is used. A gTelegram graph type consists in general of five object types: oVariableInstace, oPacket, oAnyPacket, oRootNode, and oLeafNode. oRootNode and oLeafNode are only used to define the first and last packet in the order of all packets. All object types can be connected by the directed relationships VariableOrder and PacketOrder. The corresponding syntax can be found in Figure 7.11. oVariableInstance objects

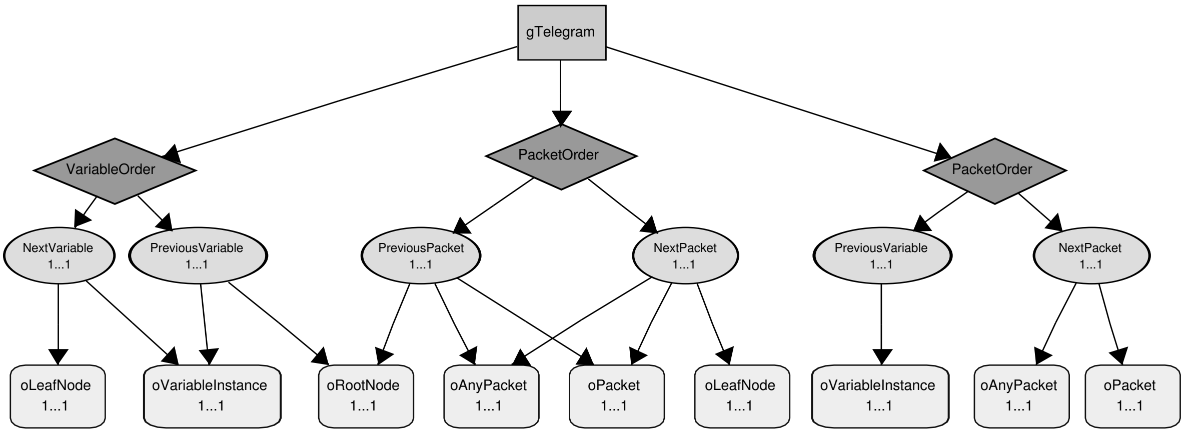


Figure 7.11.: gTelegram binding syntax

can be only used at the beginning of a telegram to define a telegram header [85]. Therefore, they only can be modelled in the order between the oRootNode object and the first oPacket or oAnyPacket instance. The VariableOrder relationship is used between the connection between two oVariableInstance objects. PacketOrder is valid between oAnyPacket and/or oPacket types, and between the last oVariableInstace and the two packet types.

The oAnyPacket object type is different from the others because it cannot be directly found in SRS [85]. It can be used to model that at a certain point in the variable-packet order several oPacket objects are possible. The concrete set of oPacket objects is specified in the decomposition of the oAnyPacket instance. In that way, ETCS telegrams only differing from each other in a certain packet can be easier and faster modelled.

7.3.7. gPacket Graph Type

Similar to the gTelegram graph type, the gPacket graph type is used to specify the order of oVariableInstance objects in a certain ETCS packet [85]. Three object types are provided:

oRootNode defines the start point of a packet order.

oVariableInstance defines a certain ETCS variable of a certain type [85]. Therefore, the oVariableInstance object type holds an oVariableType object as property. The oVariableType type is not graphically used (only as property) and specifies the type of an oVariableInstance object, which is similar to a template-mechanism [79] in object-oriented programming languages. The oVariableType type holds several properties for defining the attributes of an oVariableInstance object:

1. size in bits
2. the variable's resolution [85]
3. the variable's physical unit

oLeafNode determines the last gPacket object in an order.

The advantage of defining a property of oVariableType object type is that the concrete oVariableType objects can be reused for several oVariableInstance objects.

Apart from enabling the specification of variable orders according to the ETCS specifications [85], also a possibility to model the scaling of certain variables is needed. Scaling means that a value of a certain variable can scale (by multiplication) the value of another. This is modelled by the additional Scaling relationship while the VariableOrder is used for specifying the oVariableInstance object order. The full syntax is illustrated in Figure 7.12.

Moreover, the ETCS language specification [85] defines so-called iterating variables. This concept is similar to arrays in C/C++ [79] or other high-level programming languages. A certain variable holds the size of an array and the following variables are components of an array of the specified size. An iterating variable is not modelled by a relationship but by graphical containment, as introduced in Chapter 4. Therefore, oVariableInstance objects that are iterated by another oVariableInstance object are drawn within the iterating instance. In this way, it is also possible to specify nested iterations.

A specialisation of iterating variables are so-called conditional iterator variables. In contrast to the normal iteration concept, conditional iterators do not iterate the graphically contained

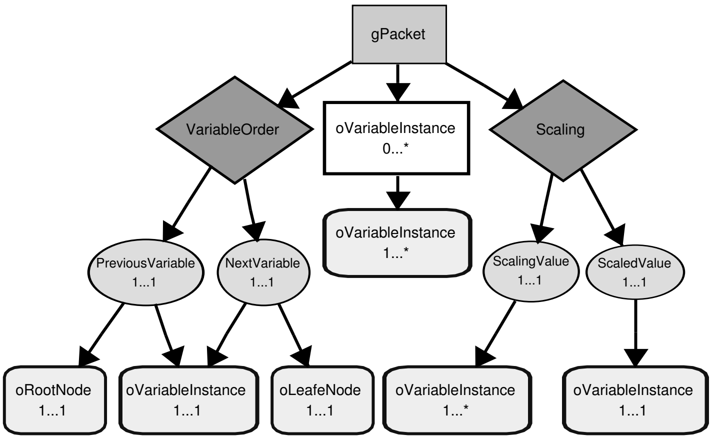


Figure 7.12.: gPacket binding syntax

objects corresponding to their numerical value. A (numerical) conditional value can be defined, for which all iterated variables (by graphically containment) are exactly iterated once. Otherwise, they will not exits. In this manner, it is possible to define the existence of certain oVariableInstace objects only for certain (numerical) conditions.

7.3.8. gAnyPacket Graph Type

The gAnyPacket type is only used to specify a set of oPacket objects for oAnyPacket objects. Hence, it has no binding syntax at all.

7.4. Concrete Syntax for Type Properties

As stated in Chapter 4, for a complete meta model syntax definition, besides sub-graphs and graph-binding, the full definition for all types[10] is needed. For the most important properties, this was already done in Section 7.2 and Section 7.3. As the definition of all types is not necessary for understanding the case study and for further reading of this document, this can be found in Appendix B.

7.5. Static Semantics for Models

Section 7.3 defined all bindings as concrete syntax for each graph type by means of the GOPPRR meta meta model. However, it is additionally necessary to define constraints for some graph types as static semantics to reduce or refine the allowed syntax for models in special situations. The use of constrains avoids models that do not have a semantic meaning with respect to the SRS and is also an issue related to the safety properties of the generated software. According to Section 4.5, constraints are defined by OCL.

[10] graphs, objects, ports, roles, and relationships

The static semantics for openETCS meta model instances will be introduced by the graph type they are mainly related to. It must be remarked that the majority of the following constraints are only meaningful combined with others. Those dependencies between constraints are not specially highlighted and therefore the following constraints should be always interpreted in combination.

7.5.1. gEVCStateMachine

This graph type is used to define all available ETCS Modes of the EVC and the transitions between those modes. An important constraint is that only one instance of this graph type must exist in a model.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEVCStateMachine')->size() = 1
```

Listing 7.1: gEVCStateMachine constraint 1

It is also crucial that each mode defines the functionality it provides when it is active. This means that in a model all oMode objects have at least one explosion to a gMainFunctionBlock graph.

```
context CProject
inv: m_GraphSet->any(m_Type = 'gEVCStateMachine').m_ObjectSet->select(m_Type =
    'oEVCState')->forAll(m_Explosions->size() >= 1)
```

Listing 7.2: gEVCStateMachine constraint 2

It must be guaranteed that all oModeGuard objects used as properties in ModeTransition relationships exist in at least one explosion of the oMode object connected by the CurrentState role to ensure the correctness of the Mode. Otherwise, ModeTransition relationships could be modelled but are never used.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEVCStateMachine')->forAll(
  graph |
  graph.m_RelationshipSet->select(m_Type = 'ModeTransition')->forAll(
    relationship |
    graph.m_BindingSet->exists(
      m_Connection.m_pRole.m_Type = 'CurrentState'
      and
      m_Connection.m_Calls->exists(
        m_pObject.m_Explosions->exists(
          explosion |
          explosion.m_ObjectSet->includes(
          relationship.m_Properties->any(m_Type =
              'EVCGuard').m_NonProperties->any(m_Type = 'oModeGuard')
          )
        )
      )
    )
  )
)
```

Listing 7.3: gEVCStateMachine constraint 2

A proper EVC implementation always needs exact one initial state. In a gEVCStateMachine graph this is modelled by the Boolean property IsInitial of the oMode object type. Thus, in a certain graph exact one oMode object must have this property set to true while all others are false.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEVCStateMachine')->forAll(
  graph |
  graph.m_ObjectSet->select(
    object |
    object.m_Type = 'oMode' and object.m_Properties->any(m_Type = 'IsInitial').m_Value
        = 'T'
  )->size() = 1
)
```

Listing 7.4: gEVCStateMachine constraint 3

Furthermore, it must be ensured that all possible transitions from a certain oMode object have different priorities. Otherwise, it is not possible to uniquely decide in a concurrent situation, which transition is going to be used.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEVCStateMachine')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'oMode')->forAll(
    mode |
    graph.m_BindingSet->select(m_Connection.m_Calls->exists(m_pObject =
        mode)).m_pRelationship.m_Properties->select(m_Type =
        'Priority')->isUnique(m_Value)
  )
)
```

Listing 7.5: gEVCStateMachine constraint 4

7.5.2. gMainFunctionBlock and gSubFunctionBlock

As both graph types define data flows, the first constraint is that each input port – independent from its data type – should be connected only with one data flow. Multiple inputs are not semantically meaningful.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock' or m_Type =
    'gSubFunctionBlock')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'oSubFunction').m_pDecomposition->closure(
    subgraph |
    subgraph.m_ObjectSet->select(m_Type = 'oSubFunction').m_pDecomposition
  )->including(graph)->forAll(
    graph |
    graph.m_PortSet->forAll(
      port |
      graph.m_BindingSet->select(m_Connection.m_Calls->exists(
        m_pPort = port and m_pRole.m_Type = 'DataInput'
      ))->size() = 1
    )
  )
)
```

Listing 7.6: gFunctionBlock constraint 1

The OCL closure [63, pp. 30-31] statement (lines 4 to 7) is not only used to check this constraint for each graph instance separately but also to include gSubFunctionBlock graphs referenced by oSubFunction object decompositions. In other words, if an input port is already connected in a gMainFunctionBlock graph and also in a gSubFunctionBlock graph included by a oSubFunction object, this constraints fails. Due to the easy usage of this complex operation, the closure statement is also used for further constraints.

Because oVariableStorage objects do not have any input ports, an additional constraint for their input count must be defined.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock' or m_Type = 'gSubFunctionBlock')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'oSubFunction').m_pDecomposition->closure(
    subgraph |
    subgraph.m_ObjectSet->select(m_Type = 'oSubFunction').m_pDecomposition
  )->including(graph)->forAll(
    graph |
    graph.m_ObjectSet->select(m_Type = 'oVariableStorage')->forAll(
      object |
      graph.m_BindingSet->select(m_Connection.m_Calls->exists(
        m_pObject = object and m_pRole.m_Type = 'DataInput'
      ))->size() = 1
    )
  )
)
```

Listing 7.7: gFunctionBlock constraint 2

Regarding the inclusion of gSubFunctionBlock graphs, it is important that a model instance does not hold a recursive inclusion of gSubFunctionBlock graphs by oSubFunction objects. Recursive inclusion means that a parent graph which includes a child graph by an oSubFunction object is also included in the child graph representation – or any other further child graph – by another oSubFunction object.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock' or m_Type = 'gSubFunctionBlock')->forAll(
  graph |
  let subobjects : Collection(GOPPRR::CObject) =
    graph.m_ObjectSet->select(m_Type = 'oSubFunction').m_pDecomposition->closure(
      subgraph |
      subgraph.m_ObjectSet->select(m_Type = 'oSubFunction').m_pDecomposition
    )->including(graph).m_ObjectSet->flatten()->select(m_Type = 'oSubFunction')
  in
    subobjects->size() = subobjects->asSet()->size()
)
```

Listing 7.8: gFunctionBlock constraint 3

The constraint that each instance of oSubFunction must have a decomposition is not extremely necessary, but it is defined to ensure model correctness.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock' or m_Type =
    'gSubFunctionBlock').m_ObjectSet->select(m_Type = 'oSubFunction')->forAll(
  subobject |
  subobject.m_pDecomposition->size() = 1
)
```

Listing 7.9: gFunctionBlock constraint 4

An instance without decomposition would not oppose the static semantics but might falsify the information provided by the graph for human beings.

The same is required for the oEmbeddedStateMachine object type.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock' or m_Type =
    'gSubFunctionBlock').m_ObjectSet->select(m_Type = 'oEmbeddedStateMachine')->forAll(
  embedded |
  embedded.m_pDecomposition->size() = 1
)
```

Listing 7.10: gFunctionBlock constraint 5

As required for the gEVCStateMachine graph type, all oModeGuard objects used as properties in ModeTransition relationships must exist at least in one explosion of the related oMode object. It is also necessary that all oModeGuard objects in a gMainFunctionBlock graph exist at least once as property of a ModeTransition relation connected to the parent oMode object. Otherwise, the activation of such oModeGuard object would be without any effect.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEVCStateMachine')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'oMode')->forAll(
    mode |
    mode.m_Explosions->forAll(
      function |
      function.m_ObjectSet->select(m_Type = 'oModeGuard')->forAll(
        guard |
        graph.m_BindingSet->select(
          m_Connection.m_Calls->exists(m_pObject = mode and m_pRole.m_Type =
              'CurrentState')
        )->exists(
          m_pRelationship.m_Properties->any(m_Type =
              'EVCGuard').m_NonProperties->any(m_Type = 'EVCGuard') = guard
        )
      )
    )
  )
)
```

Listing 7.11: gFunctionBlock constraint 6

A very similar constraint is required for oStateGuard objects that are used in gSubFunctionBlock and gEmbeddedStateMachine graphs.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEmbeddedStateMachine')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'oEmbeddedState')->forAll(
    state |
    state.m_pDecomposition.m_ObjectSet->select(m_Type = 'oStateGuard')->forAll(
      guard |
      graph.m_BindingSet->select(
        m_Connection.m_Calls->exists(m_pObject = mode and m_pRole.m_Type =
            'CurrentState')
      )->exists(
        m_pRelationship.m_Properties->any(m_Type =
            'StateGuard').m_NonProperties->any(m_Type = 'StateGuard') = guard
      )
    )
  )
)
```

Listing 7.12: gFunctionBlock constraint 7

Since the concrete syntax only uses the atomic types[11] for input and output ports, it must be ensured that in each data flow both are of the same type.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock').m_BindingSet->forAll(
  binding |
  binding.m_Connection.m_Calls.m_pPort.m_Type->asSet()->size() = 1
)
```

Listing 7.13: gFunctionBlock constraint 8

A final constraint for gMainFunctionBlock was already derived in Subsection 7.3.2 that requires that no oStateGuard objects may exist in a gMainFunctionBlock graph.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock').m_ObjectSet->select(m_Type =
    'oStateGuard')->size() = 0
```

Listing 7.14: gFunctionBlock constraint 9

7.5.3. gEmbeddedStateMachine Graph Type

Similar to the gEVCStateMachine graph type, it is required that the oStateGuard objects used as guards in StateTransition relationships exist in the decomposition of the oEmbeddedState object on the connected CurrentState role.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEmbeddedStateMachine')->forAll(
  graph |
  graph.m_RelationshipSet->select(m_Type = 'StateTransition')->forAll(
    relationship |
    graph.m_BindingSet->exists(
      m_Connection.m_pRole.m_Type = 'CurrentState'
      and
      m_Connection.m_Calls->exists(
        m_pObject.m_pDecomposition.m_ObjectSet->includes(
        relationship.m_Properties->any(m_Type =
            'StateGuard').m_NonProperties->any(m_Type = 'oStateGuard')
        )
      )
```

[11] atomicInput and atomicOutput

```
    )
  )
)
```

Listing 7.15: gEmbeddedStateMachine constraint 1

This constraint implicitly defines that all oEmbeddedState objects have a decomposition.

A state machine always needs to have exact one initial state to be able to start.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEmbeddedStateMachine')->forAll(
  graph |
  graph.m_ObjectSet.select(m_Type = 'oEmbeddedInitialState')->size() = 1
)
```

Listing 7.16: gEmbeddedStateMachine constraint 2

Additionally, each oEmbeddedInitialState object must be exact in one current state role or rather an InitialTransition relationship.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEmbeddedStateMachine')->forAll(
  graph |
  graph.m_BindingSet->select(m_pRelationship.m_Type = 'InitialTransition')->size() = 1
)
```

Listing 7.17: gEmbeddedStateMachine constraint 3

It is necessary that all oEmbeddedState objects are in at least one CurrentState role to avoid dead locks / states.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEmbeddedStateMachine')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'oEmbeddedState')->forAll(
    object |
    graph.m_BindingSet->select(m_pRelationship.m_Type =
        'Transition').m_Connection.m_Calls->select(m_pRole.m_Type = 'CurrentState' and
        m_pObject = object)->size() >= 1
  )
)
```

Listing 7.18: gEmbeddedStateMachine constraint 4

Similar to the constraint for non-concurrent priorities for transition of ETCS Modes in Subsection 7.5.1, also those situations must be avoided for the priority of transitions of embedded state machines.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gEmbeddedStateMachine')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'oEmbeddedState')->forAll(
    state |
    graph.m_BindingSet->select(m_Connection.m_Calls->exists(m_pObject =
        state)).m_pRelationship->select(m_Type =
        'Transition').m_Properties->select(m_Type = 'Priority')->isUnique(m_Value)
  )
)
```

Listing 7.19: gEmbeddedStateMachine constraint 5

7.5.4. gCommunicationReader and gCommunicationSender Graph Type

The communication graph type provides a reduced set of possibilities for modelling data flows between oTelegram objects and oSenderDevice or oReaderDevice objects. As for the graph types for data flows, it also has to be ensured that each input port is only used once.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gCommunicationReader' or m_Type =
     'gCommunicationSender')->forAll(
  graph |
  graph.m_PortSet->forAll(
    port |
    graph.m_BindingSet->select(m_Connection.m_Calls->exists(
      m_pPort = port and m_pRole.m_Type = 'DataInput'
    ))->size() = 1
  )
)
```

Listing 7.20: Communication graph constraint 1

The following two constraints are defined to guarantee that all modelled oPacket objects are graphically included by exact one oTelegram object and are also part of the decomposition graph (of type gTelegram) of this oTelegram object.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gCommunicationReader' or m_Type =
     'gCommunicationSender')->forAll(
  graph |
  graph.m_Containers->select(m_pContainer.m_Type =
       'oTelegram').m_ContainedObjects->size() =
      graph.m_Containers->select(m_pContainer.m_Type =
       'oTelegram').m_ContainedObjects->asSet->size()
)
```

Listing 7.21: Communication graph constraint 1

```
context CProject
inv: m_GraphSet->select(m_Type = 'gCommunicationReader' or m_Type =
     'gCommunicationSender')->forAll(
  graph |
  graph.m_Containers->select(m_pContainer.m_Type = 'oTelegram')->forAll(
    container |
    container.m_ContainedObjects->select(m_Type = 'oPacket')->forAll(
      packet |
      container.m_pContainer.m_pDecomposition.m_ObjectSet->includes(packet)
    )
  )
)
```

Listing 7.22: Communication graph constraint 2

The same constraints are also needed for oVariableInstance objects that are graphically contained by oPacket objects.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gCommunicationReader' or m_Type =
     'gCommunicationSender')->forAll(
  graph |
  graph.m_Containers->select(m_pContainer.m_Type =
      'oPacket')->m_ContainedObjects->size() =
      graph.m_Containers->select(m_pContainer.m_Type =
      'oPacket').m_ContainedObjects->asSet->size()
)
```

Listing 7.23: Communication graph constraint 3

```
context CProject
inv: m_GraphSet->select(m_Type = 'gCommunicationReader' or m_Type =
    'gCommunicationSender')->forAll(
  graph |
  graph.m_Containers->select(m_pContainer.m_Type = 'oPacket')->forAll(
    container |
    container.m_ContainedObjects->select(m_Type = 'oVariableInstace')->forAll(
      variable |
      container.m_pContainer.m_pDecomposition.m_ObjectSet->includes(variable)
    )
  )
)
```

Listing 7.24: Communication graph constraint 4

7.5.5. gTelegram Graph Type

The gTelegram graph type is used to define an ordered set of oVariableInstance and oPacket objects that build together an ETCS telegram. Therefore, it is important that this ordered set has a defined beginning and ending, which is modelled by oRootNode and oLeafNode objects. The derived constraint is that in each gTelegram exists exact one instance of each type.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gTelegram')->forAll(
  graph |
  graph.m_ObjectSet->select(m_Type = 'oRootNode')->size() = 1
  and
  graph.m_ObjectSet->select(m_Type = 'oLeafNode')->size() = 1
)
```

Listing 7.25: gTelegram constraint 1

Furthermore, no “loose” objects are allowed, which means that all objects besides the oLeafNode and oRootNode instances, must have exact one NextVariable or NextPacket role and exact one PreviousVariable or PreviousPacket role.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gTelegram')->forAll(
  graph |
  graph.m_ObjectSet->reject(m_Type = 'oRootNode' or m_Type = 'oLeafNode')->forAll(
    object |
    graph.m_BindingSet.m_Connection.m_Calls->select(
      (m_pRole.m_Type = 'NextVariable' or m_pRole.m_Type = 'NextPacket') and m_pObject
          = object
    )->size() = 1
    and
    graph.m_BindingSet.m_Connection.m_Calls->select(
      (m_pRole.m_Type = 'PreviousVariable' or m_pRole.m_Type = 'PreviousPacket') and
          m_pObject = object
      )->size() = 1
   )
)
```

Listing 7.26: gTelegram constraint 2

This constraint implicitly contains the requirement that the oRootNode and oLeafNode objects are always connected because only by their usage it is possible to not model "loose" oVariableInstace or oPacket objects.

Finally, for model correctness all oPacket objects should have a decomposition.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gTelegram')->forAll(
   graph |
   graph.m_ObjectSet->select(m_Type = 'oPacket')->forAll(
      object |
      object.m_pDecomposition->size() = 1
   )
)
```

Listing 7.27: gTelegram constraint 2

7.5.6. gPacket Graph Type

Like the gTelegram graph type, the gPacket graph type is used to model an ordered set of object types. The difference is that only oVariableInstace objects can be used. Therefore, the constraint for the existence of an oRootNode and an oLeafNode object is mainly the same.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gPacket')->forAll(
   graph |
   graph.m_ObjectSet->select(m_Type = 'oRootNode')->size() = 1
   and
   graph.m_ObjectSet->select(m_Type = 'oLeafNode')->size() = 1
)
```

Listing 7.28: gPacket constraint 1

Also, the concrete constraint for no "loose" oVariableInstace is the same, only the packet related roles are not used.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gPacket')->forAll(
   graph |
   graph.m_ObjectSet->reject(m_Type = 'oRootNode' or m_Type = 'oLeafNode')->forAll(
      object |
      graph.m_BindingSet.m_Connection.m_Calls->select(
         (m_pRole.m_Type = 'NextVariable') and m_pObject = object
      )->size() = 1
      and
      graph.m_BindingSet.m_Connection.m_Calls->select(
         (m_pRole.m_Type = 'PreviousVariable') and m_pObject = object
      )->size() = 1
   )
)
```

Listing 7.29: gTelegram constraint 2

7.5.7. Undefinable Static Semantics

A constraint type for the data flow graph types that is probably meaningful was not yet specified. It concerns to not used input and output ports of any data type in gMainFunctionBlock and gSubFunctionBlock graphs. It seems meaningful to require that any available port of objects in those graph types should be used or rather connected. The behaviour is that in cases, in which inputs are not set, the corresponding output(s) cannot be directly defined. It can be an initial value but also the last value set in ETCS Mode or Application Level, in which the output was used last.

The reason why a general OCL constraint cannot be defined for this purpose is that in MERL only ports that are in a connection are accessible. Unconnected ports are simply not visible to any generator. Therefore, unconnected ports cannot be found in the GOPPRR C++ abstract syntax model, which is used for the constraint checking.

The only currently possible work-around for this situation is to define an own constraint for each concrete object type with ports while using the knowledge about its port number in the constraint.

For example, for the oAND type, which has two Boolean inputs and one Boolean output, the following statement is defined to require that all its input ports must be used,

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock' or m_Type =
    'gSubFunctionBlock')->forAll(
  function |
  function.m_ObjectSet->select(m_Type = 'oAND')->forAll(
    object |
    function.m_BindingSet.m_Connection.m_Calls->select(m_pObject = object and m_pRole
        = 'DataInput').m_pPort->size() = 2
  )
)
```

Listing 7.30: oAND input port constraint

The requirement that all output ports are connected is not that crucial because those unconnected ports cannot cause an undefined calculation result, but it might be useful to avoid unused objects in graphs. The corresponding constraint for the oAND type is defined as follows.

```
context CProject
inv: m_GraphSet->select(m_Type = 'gMainFunctionBlock' or m_Type =
    'gSubFunctionBlock')->forAll(
  function |
  function.m_ObjectSet->select(m_Type = 'oAND')->forAll(
    object |
    function.m_BindingSet.m_Connection.m_Calls->select(m_pObject = object and m_pRole
        = 'DataOutput').m_pPort->asSet()->size() = 1
  )
)
```

Listing 7.31: oAND output port constraint

The main difference to the input port constraint is that output ports can be connected more than once. Therefore, a multiple times connected port may only be counted once. This is done by the OCL asSet() [63, p. 158] statement. Because those constraints only differ in the object type name and the number of input and output ports, those are not presented here for other object types.

7.5.8. Static Semantics directly used in the Modelling Process

According to the tool chain introduced in Section 4.6, constraints are defined outside the MetaEdit+ application and checked by an external generator. Therefore, the generated report is pure textual and it has to be (manually) searched in MetaEdit+ for incorrect model elements. This drawback cannot easily be avoided because the used GOPPRR XML format used as interface is general and accordingly independent from the concrete meta model. However, it is possible to check simple constraints directly during the manual modelling process.

The best way to do this is to use generators that do not produce any direct output to files but graphical feedback. In MetaEdit+ [56], this is done by MERL scripts that directly output information to the graphical representation of a GOPPRR element that is not a graph, a port, or a property[12]. Those scripts are always executed in the scope of the current graph, which the representation is displayed in. This is the obvious limitation for this method because, as discussed in Chapter 4, constraints related to the project scope are in this way very difficult to implement. Constraints that are related to graphs cannot be implemented because MERL generators for graphs cannot be output on the graph representation itself. Properties do not have any direct graphical representation at all, and ports are only graphical connection points.

Thus, only a subset of the above introduced constraints are implemented directly in the graphical elements of the openETCS meta model. However, even this subset providing direct graphical feedback in the case of model or rather static semantics violations can help an ETCS expert during the specification process.

7.6. Dynamic Semantics for the Cyclic Execution

The presented openETCS meta model or formal specification language can mainly divided in three major parts:

1. a superior state machine for ETCS Modes
2. data and control flows for behavioural and functional specification
3. specification of data structures for communication

The dynamic semantics of the superior state machine defined in a gEVCStateMachine graph is described in general as for state machines [80]. As already discussed, no temporal behaviour can be derived directly from a gEVCStateMachine graph because oModeGuard objects are used as conditions for transitions. Since the oModeGuard objects are used in the data flows, only the specification language part for data and control flow defines temporal behaviour of the whole model. Although data flows can be modelled in general, it is obvious that when the model is transformed to an executable binary[13] the data flow has to be implemented in discrete manner. From control theory, it is known that for the establishment of stable, discrete systems those must have a fixed sample time T_s [30]. This means that the data flows or rather their calculations k are executed at certain time points t, which all have the same temporal distance:

$$t = kT_s; \quad k = 0, 1, 2, \ldots \tag{7.1}$$

[12] objects, roles, and relationships

[13] for a certain computer platform

It has to be acknowledged that not for all objects with outputs those cannot be calculated in parallel for the dynamic semantics of a gMainFunctionBlock and gSubFunctionBlock graph. Furthermore, it is meaningful if first the outputs are calculated, which then are used as inputs for other objects. In a simple chain or rather an open loop [30], the first element can easily be identified at the beginning of the open loop. Figure 7.13 shows a simple example for an open loop. In this example, in every execution step k the outputs are calculated in the following order:

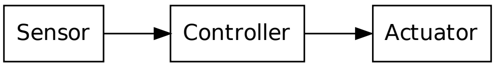


Figure 7.13.: Simple example of an open loop

1. Sensor
2. Controller
3. Actuator

Also, the definition for closed loops is possible if they still have a “start” element, which normally only have outputs as in Figure 7.14.

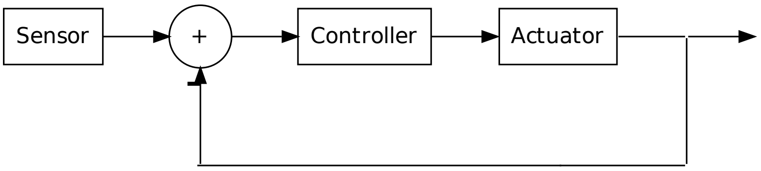


Figure 7.14.: Simple example of a closed loop

Similar to the open loop example the execution order is accordingly:

1. Sensor
2. Sum (“+”)
3. Controller
4. Actuator

When the Sum is calculated, the new output of the Controller is not yet available. Hence, for such feedback flows the last output value is taken. In other words, the negative input of the Sum is in calculation k the output of the Actuator in calculation[14] $k-1$. This also demonstrates that it is necessary that all data flow elements must have initial output values if they have outputs to avoid undefined calculation results for $k=0$.

In rare cases, it also may occur that a closed loop does not have a beginning or ending element. For example, Figure 7.14 without the Sensor. Then the starting element would be once chosen arbitrarily.

[14] Although in the openETCS meta model actuators do not have outputs, here it is used for simplification.

Thus, the oEmbeddedStateMachine objects, which are part of the data flow graph types gMainFunctionBlock and gSubFunctionBlock, are also executed with the sample time T_s by their parent graph instance. Very similar to the gEVCStateMachine graph type, the gEmbeddedStateMachine graph type does not provide any temporal behaviour by itself. This is provided by the decomposition of each oEmbeddedState object to a gSubFunctionBlock graph. Therefore, the currently active oEmbeddedState object is executed if its parent oEmbeddedStateMachine object is executed within a data flow graph.

The dynamic semantics of oSubFunction objects can be interpreted in two ways: First, when they are executed by the data flow graph[15] that they are part of, they only delegate this execution to the data flows in their decomposition. Second, the content of their decomposition is virtually copied to the parent graph contenting the oSubFunction object, and that all objects are executed directly by the parent graph. Actually, the latter interpretation indeed represents the implementation of the domain framework and generator, in which after the graphical "unrolling" all objects are located in a gMainFunctionBlock, which is executed with a certain sample time. Nevertheless, the first dynamic semantics interpretation is also valid, and both correspond to the provided functionality.

The transformation from an openETCS model with general data flows to a concrete sample system is done by the generator and the domain framework. The generator defines the calculation order in each execution step k while the domain framework ensures that those are executed in equidistant time points t with a fixed sample time T_s. The concrete value for T_s is not defined in the model but in the domain framework. Related to the definition of control loops, this can arise severe problems related to stability because the sample time directly influences the parameters of a controller [30]. Simplified expressed, if a control loop with the same parameters is stable for T_s, it can be unstable for T'_s (with $T_s \neq T'_s$). Therefore, it must be emphasised that the openETCS meta model is not specialised for building control loops in the meaning of control theory but for defining control functions for ETCS. In addition, concrete control loop issues are handled directly in the domain framework design and implementation.

7.7. Mathematical Model of the Dynamic Semantics

The preceding sections Section 7.2, Section 7.3, and Section 7.5 introduced the concrete syntax and static semantics but did not explain how the correctness of model instances can be ensured in an analytical way. One possibility is to transform the dynamic semantics to a mathematical model, which certain properties can be analysed and proven for. Because some graph types use very similar syntax and semantics, those can be grouped for defining corresponding mathematical models in equivalence groups:

data flows gMainFunctionBlock, gSubFunctionBlock, gCommunicationReader, gCommunicationSender

state machines gEVCStateMachine, gEmbeddedStateMachine, gMainFunctionBlock / oApplicationLevelType

[15] gMainFunctionBlock or gSubFunctionBlock graph

data structures gTelegram, gPacket, gAnyPacket

While state machines together with the data flows describe the temporal behaviour, data structures are time invariant and therefore easier to define. A corresponding mathematical model for each group is defined and described in the following subsections.

7.7.1. Data Flows

Generally, a data flow consist of a set of transfer functions F determined by the objects, the available inputs X and outputs Y, the internal states S and initial states S_0 of those, and the connection matrix $\underline{C}$ for the data flow relationships. This tuple defines the data flow D:

$$D = (F, X, Y, S, S_0, \underline{C}) \tag{7.2}$$

The set of transfer functions consists of n_f functions

$$F \equiv \{f_i;\ i = 1, 2, 3, \ldots, n_f\} \tag{7.3}$$

while each f_i is generally defined by a function

$$f_i = \underline{f}_i(\underline{x}_i, \underline{s}_i) \tag{7.4}$$

where $\underline{x}_i$ is a vector with all inputs and $\underline{s}_i$ a vector with all internal states. The result of $\underline{f}_i$ is its new internal state vector $\underline{s}'_i$ and output vector $\underline{y}_i$:

$$\begin{bmatrix} \underline{s}'_i \\ \underline{y}_i \end{bmatrix} = \underline{f}_i(\underline{x}_i, \underline{s}_i) \tag{7.5}$$

For clarification, the vectors can be written as

$$\underline{x}_i = \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,m_i} \end{bmatrix}; \quad \underline{s}_i = \begin{bmatrix} s_{i,1} \\ s_{i,2} \\ \vdots \\ s_{i,o_i} \end{bmatrix}; \quad \underline{y}_i = \begin{bmatrix} y_{i,1} \\ y_{i,2} \\ \vdots \\ y_{i,p_i} \end{bmatrix} \Rightarrow \begin{bmatrix} \underline{s}'_i \\ \underline{y}_i \end{bmatrix} = \begin{bmatrix} s'_{i,1} \\ s'_{i,2} \\ \vdots \\ s'_{i,o_i} \\ y_{i,1} \\ y_{i,2} \\ \vdots \\ y_{i,p_i} \end{bmatrix} \tag{7.6}$$

All time variant values are digitised by the index k for the current execution point to additionally consider that the data flow is discrete:

$$\begin{bmatrix} \underline{s}_{i,k} \\ \underline{y}_{i,k} \end{bmatrix} = \underline{f}_i(\underline{x}_{i,k}, \underline{s}_{i,k-1}); \quad k = 0, 1, 2, 3, \ldots \tag{7.7}$$

In the case $k = 0$ for the initial calculation, the constant initial state vector $\underline{s}_{i,0}$ must be used because the usage of negative calculation time points $t \mid_{k=-1} = -T_s < 0$ would render the system non-causal. Thus, the calculation time point must be translated about 1:

$$\begin{bmatrix} \underline{s}_{i,k+1} \\ \underline{y}_{i,k+1} \end{bmatrix} = \underline{f}_i(\underline{x}_{i,k+1}, \underline{s}_{i,k}); \quad k = 0, 1, 2, 3, \ldots \tag{7.8}$$

Furthermore, not all data flow objects or rather transfer functions have inputs and outputs. If they only have inputs ($p_i = 0$), their transfer function equation is reduced to

$$\underline{s}_{i,k+1} = \underline{f}_i(\underline{x}_{i,k+1}, \underline{s}_{i,k}); \quad k = 0, 1, 2, 3, \ldots \tag{7.9}$$

Analogously, transfer functions without any input ($m_i = 0$) can be reduced to

$$\begin{bmatrix} \underline{s}_{i,k+1} \\ \underline{y}_{i,k+1} \end{bmatrix} = \underline{f}_i(\underline{s}_{i,k}); \quad k = 0, 1, 2, 3, \ldots \tag{7.10}$$

While

$$X \equiv \left\{\underline{x}_j;\ j = 1, 2, 3, \ldots, n_x\right\} \tag{7.11}$$

$$Y \equiv \left\{\underline{y}_l;\ l = 1, 2, 3, \ldots, n_y\right\} \tag{7.12}$$

$$S \equiv \left\{\underline{s}_p;\ p = 1, 2, 3, \ldots, n_s\right\} \tag{7.13}$$

$$S_0 \equiv \left\{\underline{s}_{o_q};\ q = 1, 2, 3, \ldots, n_{s_0}\right\} \tag{7.14}$$

are defined for a whole data flow model, the connection matrix $\underline{C}$ defines the connections between all outputs and inputs by the connection equation:

$$\underline{x}_k = \underline{C}\underline{y}_k \tag{7.15}$$

The global vectors $\underline{x}$ and $\underline{y}$ with all inputs and outputs of a data flow model are defined as follows:

$$\underline{x} = \begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \\ \vdots \\ \underline{x}_{n_x} \end{bmatrix} = \begin{bmatrix} x_{1,1} \\ \vdots \\ x_{1,m_1} \\ x_{2,1} \\ \vdots \\ x_{2,m_2} \\ \vdots \\ x_{n_x,1} \\ \vdots \\ x_{n_x,m_{n_x}} \end{bmatrix} \tag{7.16}$$

$$\underline{y} = \begin{bmatrix} \underline{y}_1 \\ \underline{y}_2 \\ \vdots \\ \underline{y}_{n_y} \end{bmatrix} = \begin{bmatrix} y_{1,1} \\ \vdots \\ y_{1,p_1} \\ y_{2,1} \\ \vdots \\ y_{2,p_2} \\ \vdots \\ y_{n_y,1} \\ \vdots \\ y_{n_y,p_{n_y}} \end{bmatrix} \tag{7.17}$$

Since there may not be any unused output in a data flow the number of outputs must always be smaller or equal the number inputs. Correspondingly, the following mathematical constraint must be fulfilled:

$$\underbrace{\sum_{u=1}^{n_y} p_u}_{p} \leq \underbrace{\sum_{v=1}^{n_x} m_v}_{m} \tag{7.18}$$

Additionally, the following two constraints have to be fulfilled for $\underline{C}$ because its elements can only be 0 or 1 and only one input can be connected with each output:

$$c_{i,j} \in \{0;1\} \tag{7.19}$$

$$\forall i : \sum_{j=1}^{p} c_{i,j} \overset{!}{=} 1 \tag{7.20}$$

Example For clarification, the simple example for a data flow in Figure 7.15 is taken. It only consists of a oSum object and three oVariableStorage objects. The object “0” is only used to ensure that all inputs of the oSum object are used, but it does not influence the calculation.

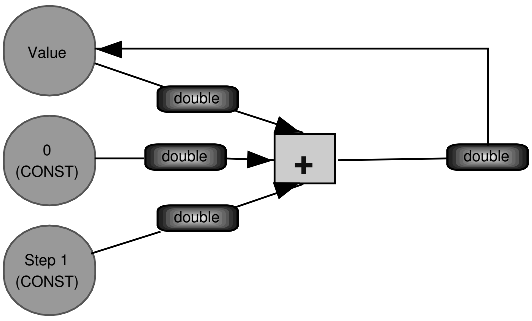


Figure 7.15.: Simple data flow example

The object “Step 1” is constant and therefore always holds the value 1. “Value” is an input to the oSum object, but is also connected to its output. Therefore this example increases the “Value” in each calculation about 1. Furthermore, “Value” has an initial value of 0.

The transfer function for the oSum object is defined as

$$y_{1,1,k+1} = f_1 = f_{\text{sum}}(\underline{x}_{1,k+1}) = x_{1,1,k+1} + x_{1,2,k+1} + x_{1,3,k+1}; \quad \underline{x}_{1,k+1} = \begin{bmatrix} x_{1,1,k+1} \\ x_{1,2,k+1} \\ x_{1,3,k+1} \end{bmatrix} \tag{7.21}$$

It can be observed that the oSum transfer function does not depend on internal states and therefore only has the input vector $\underline{x}_1$ as parameter. The oVariableStorage objects have the following transfer functions:

$$\begin{bmatrix} s_{2,1,k+1} \\ y_{2,1,k+1} \end{bmatrix} = \underline{f}_2 = \underline{f}_{\text{Value}}(\underline{x}_{2,k+1}, \underline{s}_{2,k}) = \begin{bmatrix} x_{2,1,k+1} \\ s_{2,1,k} \end{bmatrix}; \quad \underline{s}_{0_2} = s_{2,1,0} = 0 \tag{7.22}$$

$$y_{3,1,k+1} = f_3 = f_{\text{Null}}(\underline{s}_{3,k}) = s_{3,1,k}; \quad \underline{s}_{0_3} = s_{3,1,0} = 0 = \text{const} \tag{7.23}$$

$$y_{4,1,k+1} = f_4 = f_{\text{Step1}}(\underline{s}_{4,k}) = s_{4,1,k}; \quad \underline{s}_{0_4} = s_{4,1,0} = 1 = \text{const} \tag{7.24}$$

The global input $\underline{x}$ and output $\underline{y}$ vector are defined as

$$\underline{x}_k = \begin{bmatrix} x_{1,1,k} \\ x_{1,2,k} \\ x_{1,3,k} \\ x_{2,1,k} \end{bmatrix} \tag{7.25}$$

$$\underline{y}_k = \begin{bmatrix} y_{1,1,k} \\ y_{2,1,k} \\ y_{3,1,k} \\ y_{4,1,k} \end{bmatrix} \tag{7.26}$$

The corresponding connection matrix $\underline{C}$ for Figure 7.15 must be defined as

$$\underline{C} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{7.27}$$

Finally, the complete connection equation is determined by

$$\begin{aligned} \begin{bmatrix} x_{1,1,k} \\ x_{1,2,k} \\ x_{1,3,k} \\ x_{2,1,k} \end{bmatrix} &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} y_{1,1,k} \\ y_{2,1,k} \\ y_{3,1,k} \\ y_{4,1,k} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,1,k} + x_{1,2,k} + x_{1,3,k} \\ s_{2,1,k-1} \\ s_{3,1,k-1} \\ s_{4,1,k-1} \end{bmatrix} \end{aligned} \tag{7.28}$$

It must be remarked that the equations for each element of $\underline{x}_k$ must be calculated in the order of the elements from top to bottom. This was reasoned in Section 7.6. Thus, the element order for $\underline{x}_k$ cannot be chosen arbitrarily.

No element $x_{i,j,k}$ may occur at the same time on the left and the right side of the equation to have meaningful equations for each element of $\underline{x}_k$. Corresponding to the openETCS model, this would be the case if an output of an object is directly connected with an own input. This could be, for example, Figure 7.15 without the "Value" object. From the view of the mathematical model, such connections are not possible and lead to undefined connection equations because for the calculation of an output is an input used that is not yet available. In the domain frame work, for each existing input is a certain default / initial value defined to allow such data flows anyway. However, the mathematical model suggests that it is in any case better to use intermediate oVariableStorage objects as "Value" and to model the initial value implicitly. A virtual storage object has always to be inserted to define connections between input and output in the mathematical model.

7.7.2. State Machines

The state machine types used in the openETCS meta model do not provide any direct outputs and accordingly can be called transition systems. Generally, such a transition system T is a tuple of a set of states S, an initial state s_0, inputs alphabet Σ, a transition function δ, and a set of final states F:

$$T = (S, s_0, \Sigma, \delta, F) \tag{7.29}$$

with

$$s_0 \in S \equiv \{s_h;\ h = 1, 2, 3, \ldots, n_s\} \tag{7.30}$$

$$F \subset \{\emptyset\} \cup S \tag{7.31}$$

$$\delta : S \times \Sigma \rightarrow S \tag{7.32}$$

This mathematical model is quite similar to a Moore machine [58]. According to the openETCS meta model, the inputs are defined by Boolean data flows to guard objects (oModeGuard, oStateGuard, oApplicationLevelType). Thus, this input can be defined as

$$\Sigma \equiv \{i_j;\ j = 1, 2, 3, \ldots, n_i\}\,;\ i_j \in \{0; 1\} \tag{7.33}$$

The transition function δ uses the state transition vector $\underline{v}(s)$, which defines for each certain / active state s the possible transitions to all others depending on the current inputs:

$$s' = \delta(s) = \delta\left(\underline{v}(s)^{\mathrm{T}}\underline{s}\right) \tag{7.34}$$

s' is the new active state and $\underline{s}$ the state vector with all available states:

$$\underline{s} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_{n_s} \end{bmatrix}\,;\ \forall o : s_o \in S \tag{7.35}$$

The Sum of all available inputs multiplied by their numerical priority, which is in the meta model a property of the related Transition relationship, are the elements of the state transition vector $\underline{v}(s)$:

$$\underline{v}(s) = \begin{bmatrix} c_1(s) \\ c_2(s) \\ \ldots \\ c_{n_s}(s) \end{bmatrix} \tag{7.36}$$

If several transition to the same new state s' exist but under different input conditions, the corresponding element c_i of the state transition vector is the sum of the products of inputs and their priority. In general, this means for each element

$$\forall l : c_l = \sum_{j=0}^{n_i} \left(i_j(p_j + 1)\right)\,; \quad p_j \in \mathbb{N} \cup \{0\} \tag{7.37}$$

p_j is the priority of the input i_j. $i_j = 1$ means that the input i_j is a condition for the corresponding new state. Otherwise, $i_j = 0$ is set. In the openETCS model, p_j is taken from the Transition relationship. The result of the $\underline{v}(s)^{\mathrm{T}}\underline{s}$ operation is a sum of possible different states s_j , which have a numerical factor each. The δ function selects from this sum the state s_j with the highest factor and returns it. Therefore, it is important, that the priority is different for all possible transitions from a certain state. Otherwise, two states with the same factor may occur and the result of the δ function is undefined. Since not at every execution point an input for a transition to a new state is set, the element in transition vector $\underline{v}(s)$ for a self transition has always the value 1.

$$s' = s = s_j \Rightarrow c_j \overset{!}{=} 1 \tag{7.38}$$

Therefore, 1 is added to the priority p_j in (7.36) in order that transitions that are not a self transition are always preferred by the δ function. Although the state machines in the openETCS meta model do not provide directly any temporal behaviour, those are driven by the active gMainFunctionBlock, which builds a sample system. According to the notation for sample systems, (7.34) can be rewritten as

$$s_{k+1} = \delta(s) = \delta\left(\underline{v}(s_k)^{\mathrm{T}}\underline{s}(s_k)\right) \tag{7.39}$$

with

$$s_k|_{k=0} = s_0 \tag{7.40}$$

The vector $\underline{s}(s_k)$ with all available states only depends on the current active state s_k but not on the sample point k and therefore does not have the corresponding index.

Example Again, a simple example of an oEmbeddedStateMachine object or rather its decomposition, the gEmbeddedStateMachine graph in Figure 7.16, is discussed to provide a better understandability. The example is a simplified life cycle of a process with four states:

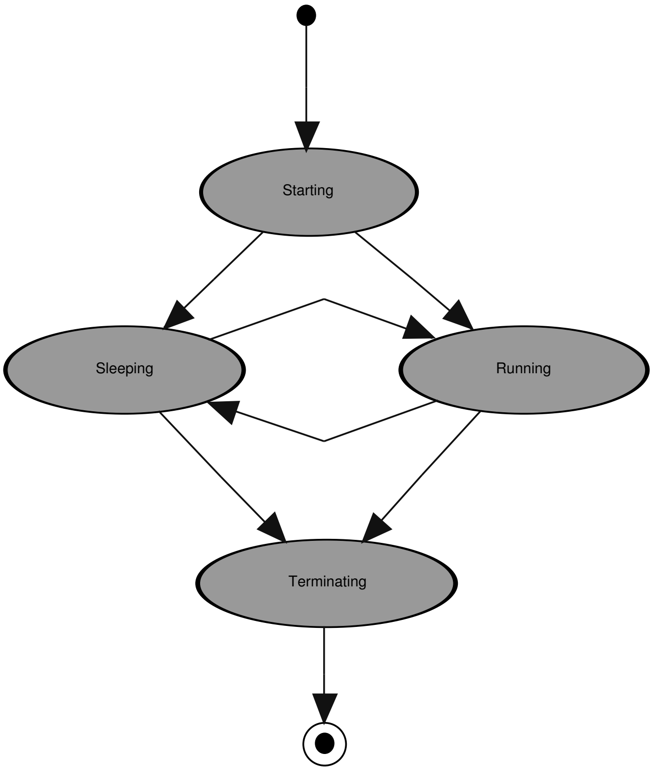


Figure 7.16.: Simple state machine example

$$S \equiv \{s_{\text{Starting}}; s_{\text{Sleeping}}; s_{\text{Running}}; s_{\text{Terminating}}\} \tag{7.41}$$

"Starting" is the initial state

$$s_0 = s_{\text{Starting}} \tag{7.42}$$

and the only final state is marked with doubled circled dot

$$F \equiv \{s_{\text{END}}\} \tag{7.43}$$

There are seven transitions, which also corresponds to the number of inputs:

$$n_i = 7 \tag{7.44}$$

The transition between the initial state, drawn by the black spot, is not counted because it does not have any condition / input and is always passed in the beginning. The conditions and inputs are numbered as follows:

Transition	Condition	Input, Priority
Starting $\longrightarrow$ Sleeping	c_1	i_1, $p_1 = 0$
Starting $\longrightarrow$ Running	c_2	i_2, $p_2 = 1$
Sleeping $\longrightarrow$ Running	c_3	i_3, $p_3 = 0$
Running $\longrightarrow$ Sleeping	c_4	i_4, $p_4 = 0$
Sleeping $\longrightarrow$ Terminating	c_5	i_5, $p_5 = 1$
Running $\longrightarrow$ Terminating	c_6	i_6, $p_6 = 1$
Terminating $\longrightarrow$ END	c_7	i_7, $p_7 = 0$

Accordingly, the transition vector $\underline{v}(s)$ can be build for each state s:

$$\begin{aligned}
\underline{s} &= \begin{bmatrix} s_{\text{Starting}} \\ s_{\text{Sleeping}} \\ s_{\text{Running}} \\ s_{\text{Terminating}} \\ s_{\text{END}} \end{bmatrix} && (7.45)\\
\underline{v}(s_{\text{Starting}})^{\text{T}} &= [1; i_{1,k}; 2i_{2,k}; 0; 0] && (7.46)\\
\underline{v}(s_{\text{Sleeping}})^{\text{T}} &= [0; 1; i_{3;,k}2i_{5,k}; 0] && (7.47)\\
\underline{v}(s_{\text{Running}})^{\text{T}} &= [0; i_{4,k}; 1; 2i_{6,k}; 0] && (7.48)\\
\underline{v}(s_{\text{Terminating}})^{\text{T}} &= [0; 0; 0; 1; i_{7,k}] && (7.49)
\end{aligned}$$

A transition vector for the final state s_{END} is not required. Since s_{Starting} is the initial state s_0, the corresponding transition function can be defined as

$$\begin{aligned}
s_k = s_{\text{Starting}} : s_{k+1} &= \delta\left([1; 2i_{1,k}; 3i_{2,k}; 0; 0] \begin{bmatrix} s_{\text{Starting}} \\ s_{\text{Sleeping}} \\ s_{\text{Running}} \\ s_{\text{Terminating}} \\ s_{\text{END}} \end{bmatrix}\right) \\
&= \delta\left(s_{\text{Starting}} + 2i_{1,k}s_{\text{Sleeping}} + 3i_{2,k}s_{\text{Running}}\right) && (7.50)
\end{aligned}$$

This is done analogously for all other non-final states:

$$s_k = s_{\text{Sleeping}} : s_{k+1} = \delta \left(s_{\text{Sleeping}} + 2i_{3,k} s_{\text{Running}} + 3i_{5,k} s_{\text{Terminating}} \right) \tag{7.51}$$

$$s_k = s_{\text{Running}} : s_{k+1} = \delta \left(2i_{4,k} s_{\text{Sleeping}} + s_{\text{Running}} + 3i_{6,k} s_{\text{Terminating}} \right) \tag{7.52}$$

$$s_k = s_{\text{Terminating}} : s_{k+1} = \delta \left(s_{\text{Terminating}} + 2i_{7,k} s_{\text{END}} \right) \tag{7.53}$$

Because the inputs $i_{j,k}$ are set in data flows, for which its mathematical model was already discussed in Subsection 7.7.1, this small example ends with the definition of the transition functions.

7.7.3. Data Structures

In contrast to the two preceding groups, the data structures do not define any behaviour but only the structure of data for communication. On the highest level, this is a telegram for Balise communication. Therefore, a set of telegrams T_i define the language L:

$$L \equiv \{T_i;\ i = 1, 2, 3, \ldots, n_T\} \tag{7.54}$$

T_i corresponds to instances of oTelegram or rather gTelegram in an openETCS model. Furthermore, each T_i is again a set of oVariableInstance V_j, oPacket P_l, and oAnyPacket A_m objects:

$$\forall i : T_i \equiv \{V_{i,j}; P_{i,l}; A_{i,m};\ j = 1, 2, \ldots, n_{V_i};\ l = 1, 2, \ldots, n_{P_i};\ m = 1, 2, \ldots, n_{A_i}\} \tag{7.55}$$

Again, all those are sets. $P_{i,l}$ is a set of oVariableInstace objects:

$$\forall i, l : P_{i,l} = \left\{V_{i,l,o};\ o = 1, 2, \ldots, n_{V_{i,o}}\right\} \tag{7.56}$$

$A_{i,m}$ is a set of oPacket objects:

$$\forall i, m : A_{i,m} \equiv \left\{P_{i,m,q};\ q = 1, 2, \ldots, n_{P_{i,q}}\right\} \tag{7.57}$$

Each $V_{i,j}$ and $V_{i,l,o}$ represent an atomic element of the language, which combines certain properties[16] p_s:

$$\forall i, j : V_{i,j} \equiv \{p_{i,l,s}; I_{i,j}; M_{i,j};\ s = 1, 2, \ldots, n_p\} \tag{7.58}$$

$$\forall i, l, o : V_{i,l,o} \equiv \{p_{i,l,o,s}; I_{i,l,o}; M_{i,l,o};\ s = 1, 2, \ldots, n_p\} \tag{7.59}$$

$I_{i,j}$ and $I_{i,l,o}$ are sets of oVariableInstace objects that are iterated:

$$\forall i, j : I_{i,j} \subset \{\emptyset\} \cup V_{i,j} \tag{7.60}$$

$$\forall i, l, o : I_{i,l,o} \subset \{\emptyset\} \cup V_{i,l,o} \tag{7.61}$$

Analogously, $M_{i,j}$ and $M_{i,l,o}$ are sets of oVariableInstace objects that are scaled:

$$\forall i, j : M_{i,j} \subset \{\emptyset\} \cup V_{i,j} \tag{7.62}$$

$$\forall i, l, o : M_{i,l,o} \subset \{\emptyset\} \cup V_{i,l,o} \tag{7.63}$$

Which concrete properties p_s exist, is not relevant for the mathematical structure and their additional introduction is therefore omitted. The meta model properties of the oVariableInstace object type can be found in Section B.2.

[16] corresponding to the properties of the language objects types in the meta model in Section B.2

7.8. Conclusion

The openETCS meta model should be considered as the most crucial instance of a DSL (Figure 7.1) related to the complete development process. Design errors made here can cause faults and resulting errors in all lower instances (model and application) that then may lead to system failures [78, p. 12]. In the worst case, such errors are neither easy detectable within a model instance nor in the generated code nor the domain framework.

Hence, it is very important to properly define the concrete syntax of the meta model to oppose such faults. For openETCS, this was done by the GOPPRR meta model for the sub-graph hierarchy, for the bindings of all graph types, and for property definition of all meta model types. The static semantics is defined for all graph types by OCL statements to “tighten” the syntax and to forbid undesired model constellations.

Furthermore, the description of the dynamic semantics of the meta model makes easier to understand how things from the ETCS SRS should be modelled in a concrete model and how they are executed in the generated binary.

Even, if syntax and static semantics are defined in a correct manner, the dynamic semantics must be analysable in a formal way to ensure that corresponding model instances do not carry any immanent defects or faults. Thus, mathematical representations of the dynamic semantics for the three major groups of the meta model graphs were introduced.

Only errors in the modelling process of transforming the textual SRS in an openETCS model cannot be avoided by the meta model because this is typically a manual process.

8 openETCS Domain Framework

This chapter starts with a definition of a set of requirements for the openETCS domain framework, which are used as starting point for its development process. The design strategy and the general use cases are discussed to identify in a first step the needed basic classes. The description of the structural software design refines the initial class structure and provides a detailed description of all domain framework types. Afterwards, the corresponding behaviour or rather the behavioural design is introduced followed by the description of the deployment. The concrete implementation of the openETCS domain framework is illustrated by examples of the source code. Finally, the strategy for the verification of the domain framework is presented. UML is used as formalism for all design descriptions.

8.1. Requirements

According to Chapter 3, a domain framework in a DSL is the interface between the generated code and the target platform. Since this is a very general description, some additional requirements were derived before the development of the openETCS domain framework:

Req.5: highest possible simplification of code generators
Req.6: maximal transparency between meta model, model, and domain framework
Req.7: as platform independent as possible
Req.8: direct support for interprocess communication(IPC)

Req.5 There are mainly two reasons for keeping the code generation as simple as possible: First, for complex meta models with a lot of different types, the implementation of code generators can easily reach a level of complexness that is not any more good manageable. Second and more important, because the domain framework is implemented once and normally is only modified due to changes in the meta model, tests provided for the domain framework can cover accordingly more static code and can be executed separately from the generated code. To test the generated code, additionally the generator or rather the generation process has to be taken into account, which increases the effort in general.

Req.6 Transparency between meta model and the domain framework means that elements in the meta model directly correspond to elements in the domain framework. The meta model primary consists of object types while objects types should correspond to classes in object-oriented programming languages, like C++. Therefore, the domain framework should contain for each object type of the meta model a corresponding class so far this is possible and meaningful for each type.

Req.7 Although this openETCS case study is not supposed to run on special target platforms, it should be executable on the prominent, general-purpose operating systems to be available to a broad community of developers. This can be accomplished by the usage of certain platform independent FLOSS libraries that encapsulates platform dependent functions, like threading or graphical user interface (GUI) creation, and provide a general interface to them.

Req.8 The necessity of the separation of processes by hardware virtualisation due to security issues was discussed in Chapter 6. Hence, processes spread over different (virtual) machines cannot simply communicate via shared memory or similar techniques. The required IPC should be implemented in the domain framework. Ideally, according to Req.7, in a platform independent way.

8.2. Design Strategy

To initially identify the major classes required in the openETCS domain framework, the UML use case diagram [64] in Figure 8.1 is used. It shows the very basic, directed use cases derived from the meta model and the general dynamic semantics in Section 7.6. The Driver actor is the only external one [64] while all others directly correspond to software classes, which are prefixed with a capital "C". All classes and the related use cases are discussed in the following.

CEVCStateMachine Provides the interface to the EVC implementation. Furthermore, it holds the functionality of the superior EVC state machine, which corresponds to the gEVCStateMachine graph in the openETCS meta model.

CDMISubject Is the driver machine interface (DMI) class. It is provided to the EVC through the CEVCStateMachine class.

CEVState Represents a ETCS Mode of the EVC and is part of CEVCStateMachine. Corresponds to the oMode object type in the meta model.

CEVCCondition Is used as transition guard between two CEVCState objects. It corresponds to oModeGuard in the meta model.

CDataFlow Builds the data flow for each CEVCState object in a certain ETCS Application Level. It is similar to the gMainFunctionBlock graph type.

CFunctionBlock Is the base class for all object types used in data flows.

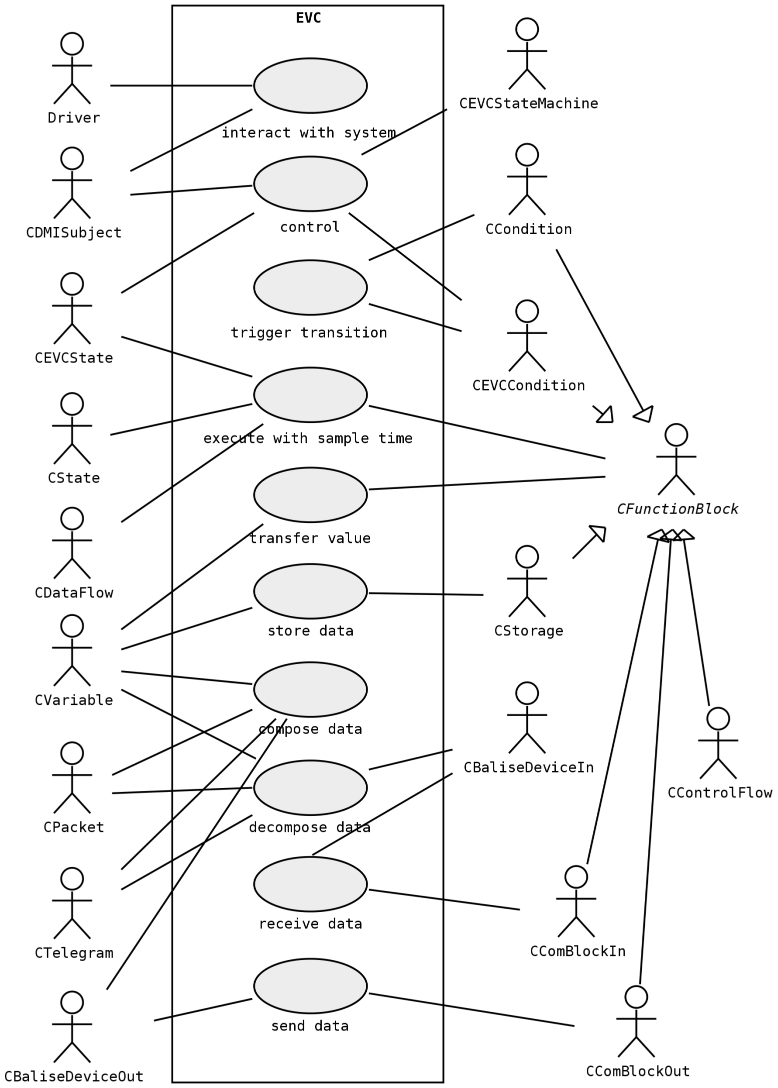


Figure 8.1.: UML use case diagram for the openETCS domain framework

CStorage Can store data flow values in different data (flow) types. It corresponds to the oVariableStorage type.

CControlFlow Provides control flows embedded in data flows according to the oEmbeddedStateMachine and gEmbeddedStateMachine types.

CState Similar to CEVCState, represents states of a control flow. It corresponds to the oEmbeddedStateMachine type in the meta model.

CCondition Like CEVCCondition, CCondition is used as a guard for transitions between two CState objects. The correspondence in the openETCS meta model is the oStateGuard object type.

CComBlockOut Represents a data flow element for defining train-to-track communication. It is owned by a CComBlockOut object and corresponds to the oCommunicationSender type.

CComBlockIn Is similar to CComBlockOut but used for track-to-train communication and corresponds to the oCommunicationReader type in the meta model.

CBaliseDeviceOut Is a sub class as interface to any concrete hardware device type for sending telegrams to balises. It is the oBaliseSender type in the meta model.

CBaliseDeviceIn Is the same as the CBaliseDeviceOut class but for receiving telegrams from balises. It corresponds to oBaliseReader.

CTelegram Builds the data superstructure for balise communication. It is accessed by CBaliseDeviceIn and CBaliseDeviceOut objects. Its correspondence in the meta model is oTelegram.

CPacket Its instances are parts of CTelegram objects. It is the oPacket type in the meta model.

CVariable Is the atomic data element in balise communication. Instances are part of CTelegram and CPacket objects. It can be set or read by a CStorage object and corresponds to oVariableInstance in the openETCS meta model.

While the use cases were utilised to identify the basic classes in the openETCS domain framework, the following sections explain its software design in detail. Mainly, UML diagrams [64] are used as description formalism.

8.3. Structural Design

The main goal of the structural design is to define for the classes, found by the first use case analysis in Section 8.2, certain object oriented design patterns [32]. Additional classes that are

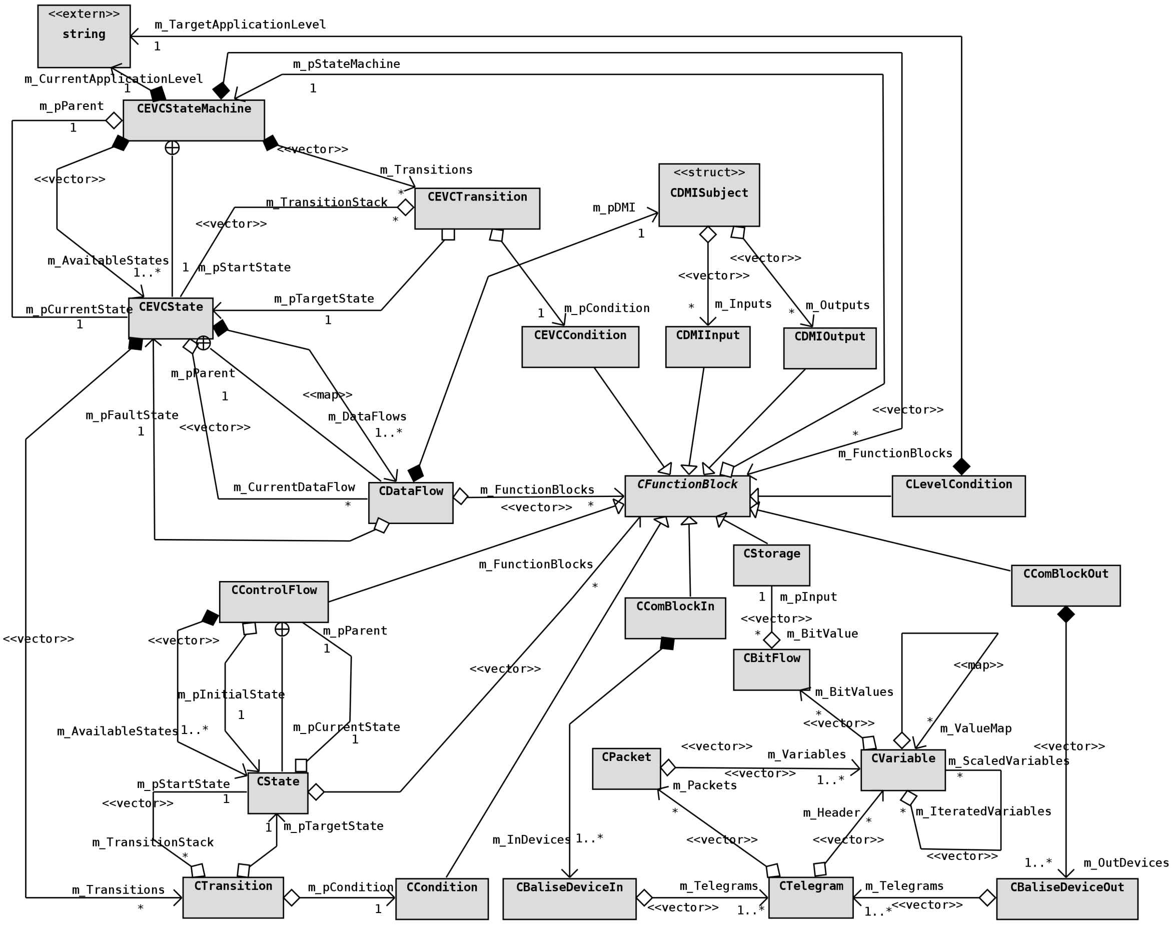


Figure 8.2.: Class diagram as overview of the openETCS domain framework

also needed should be identified during this process, too. Figure 8.2 is a UML class diagram with an overview of the structural design of the openETCS domain framework. Although the diagram already holds several classes, the openETCS domain framework consists of more. Therefore, Figure 8.2 provides only an initial overview about the central class structures. Details of further classes are introduced later in this section by further UML class diagrams. The following overview is discussed in groups according to the openETCS meta model structure.

EVC State Machine The EVC state machine is the basis in all openETCS implementations[1]. Thus, exact one object of CEVCStateMachine is used for building an EVC. Since the EVC is a state machine or rather an ETCS Mode machine, a general state design pattern [32, pp. 305-313] was specialised for openETCS: CEVCState is the only class representing ETCS Modes, accordingly no base class for states is provided. Furthermore, CEVCState is nested [79] in CEVCStateMachine to provide its instances direct access to its parent object via the `m_pParent` aggregation [64]. The current active state is indicated by the `m_pCurrentState` aggregation end in CEVCStateMachine. Additionally, it holds all available states in the `m_AvailbleStates` composition [64]. The current ETCS Application Level is stored in a string as a `m_CurrentApplicationLevel` composition.

The CEVCTransition class can be interpreted as the Transition relationship in the meta model. Each of its object is associated with a start state `m_pStartState`, a target state `m_pTargetState`, and a condition `m_pCondition`, which it is activated in. The `m_TransitionStack` aggregation stores enabled transition as a stack to support situations, which more than one transition is enabled in. In such cases, the transition to be executed should be selected by the highest priority.

The CEVCCondition class is part of data flows and therefore inherits from CFunctionBlock and only takes a Boolean input as condition for the corresponding CEVCTransition object. CLevelCondition objects are used similarly to switch to a new Application Level depending on their `m_TargetApplicationLevel` composition.

Data Flows For each available ETCS Application Level in CEVCState, at least one CDataFlow object must be available. Those are combined in the `m_DataFlows` composition of CEVCState while the currently executed data flows are indicated by the `m_CurrentDataFlow` aggregation. If more than one CDataFlow object is used in a certain CEVCState and a certain Application Level `m_CurrentApplicationLevel`, this means that those data flows are independent and can be executed in parallel to gain performance on multi-core systems. Anyway, how independent data flows are identified is an issue of the code generator and will be discussed in Chapter 9. Each CDataFlow object holds an aggregation `m_pParent` to its parent CEVCState object. Again, to provide direct access to members in CEVCState and CEVCStateMachine, CDataFlow is nested in the state class.

A CDataFlow object holds a set of CFunctionBlock objects in the `m_FunctionBlocks` aggregation, which are executed in the order as they are stored in the aggregation. The cyclic execution described in Section 7.6 is actually implemented in the CEVCState class, which starts the execution of `m_CurrentDataFlow` with a fixed sample time.

[1] in the meaning of the generated code

CFunctionBlock objects have an aggregation `m_pParent` to the CEVCStateMachine instance while all CFunctionBlock objects used in the EVC state machine are owned by the CEVCStateMachine instance via the `m_FunctionBlocks` composition. This is necessary to make CFunctionBlock objects CEVCStateMachine global and not only related to a certain CDataFlow object, as it is also the case in the openETCS meta mode or rather model.

The CDataFlow class has an aggregation `m_pFaultState`, which indicates the state that is switched to, if an error in the data flow execution occurs. This corresponds to the FailureGuard property of the gMainFunctionBlock graph type in the meta model (see Appendix B).

Control Flows In contrast to CDataFlow, the CControlFlow class represents the embedded control flows. As a consequence, it inherits from CFunctionBlock. Due to the circumstances that a control flow is defined by a state machine formalism, the design pattern used for the EVC state machine is integrated again. The `m_AvailbleStates` composition holds all available states of a control flow while `m_pCurrentState` determines the currently active one. The initial CState object is defined by the `m_pInitialState` aggregation. According to the design pattern, CState is a nested class of CControlFlow. CTransition represents possible transitions between two CState objects defined by the `m_pStartState` and `m_pTargetState` aggregations. CCondition objects are used like CEVCCondition objects in data flows to activate CTransition objects which have them in their `m_pCondition` aggregation.

Communications CComBlockIn and CComBlockOut are both part of data flows and are accordingly CFunctionBlock types. Corresponding to the decomposition to the gCommunicationSender and gCommunicationReader graph types in the meta model, both hold a composition to CBaliseDeviceIn and CBaliseDeviceOut objects. To each balise device object at least one CTelegram object by the `m_Telegram` aggregation is assigned, which is used for receiving or sending.

Language Since the ETCS language elements in the openETCS meta model are only used to model data structures and no behaviour, the meta model elements can be transferred more or less directly to the structural design. CTelegram objects consist of CVariable (`m_Header` aggregation) and CPacket (`m_Packets` aggregation) instances while their order is determined by the order in the aggregations[2]. CPacket objects hold one ore more CVariable objects by the `m_Variables` aggregation, which corresponds to the meta model. The ETCS variable iteration and scaling is modelled by the self aggregations in CVariable `m_IteratedVariables` and `m_ScaledVariables`. The mapping of certain values, which is modelled by a property in the meta model, is defined by the `m_ValueMap` aggregation.

Driver Machine Interface Within the openETCS domain framework, the DMI mainly consists of three classes: First, CDMIInput for modelling inputs from the driver, which corresponds to oDMIInput in the meta model. Second, CDMIOutput for modelling the output of information to the driver, which corresponds to oDMIOutput. Third, CDMISubject combines all exiting input and output objects for a certain CDataFlow instance, which is defined by the `m_Inputs`

[2] as stacks / sequences

and `m_Outputs` aggregation while it is owned by a CDataFlow instance through the `m_pDMI` composition. CDMIInput and CDMIOutput are both, according to the meta model, data flow elements and therefore inherit from CFunctionBlock.

The current active subject depending on the currently executed data flow can be accessed by the parent CEVCStateMachine instance via the executed CEVCState instance and its `m_pCurrentState` composition.

To not directly bind the concrete implementation of the graphical user interface (GUI) to the openETCS domain framework, a modified observer design pattern [32, pp. 293-303] is used. The CDMIWidget class (not shown in Figure 8.2) is a concrete implementation of an observer, which means the GUI representation of the DMI. An abstract subject [32, pp. 293-303] is not needed here because exact one data basis, the DMI, exists.

8.3.1. Data Flow Details

Due to the fact that the data flow is the central part of the openETCS meta model, also the domain framework reflects this. All object types in the meta model (Subsection 7.3.2) with inputs or/and outputs have a corresponding class in the domain framework. Figure 8.3 shows all special data flow classes. Since all class names only differ from the meta model object type names in the prefix ("o" → "C"), the description of their functionality will be not repeated here but can be found in Subsection 7.3.2.

The openETCS meta model defines in graph types related to data flows the connection between outputs and inputs of objects by ports in an interfacing concept. Therefore, the transfer of this port or rather interfacing concept to the domain framework is an important part. A possible solution is provided by the chain of responsibility design pattern [32, pp. 223-232], which is used to connect objects[3] in a chain that passes a request after handling it to the next instance in the chain. This pattern would have to be adapted to support different requests in form of different data flow types and multiple inputs and outputs for different function block classes. Also, the support of closed loops would have to be added. On the other hand, the instantiation and parametrisation of the domain framework classes is done by source code created from a code generator. This code generator has access to the model instance and therefore has certain knowledge about the data flow structure. Thus, it can generate the instantiations in an appropriate way that renders a complex design pattern for this issue unnecessary. As a consequence, inputs can be easily defined by class properties of the corresponding data type, which can directly store the current value. Outputs are class properties as references or rather pointers [79] to the data types and can be connected to inputs by storing their reference / address. Furthermore, outputs can be connected with more than one input and therefore the class properties must be sequences [63] of references / pointers.

Unfortunately, a certain problem arises with this design: Since all data flow class objects are globally available for an EVC implementation, as they are in a model instance, their inputs and outputs can be used in several data flows or rather gMainFunctionBlock and gSubFunctionBlock graphs. The concrete problem is that a certain CFunctionBlock object, after its execution, sets all inputs of other objects referenced in its output properties. This might include inputs of

[3] called handler in the pattern

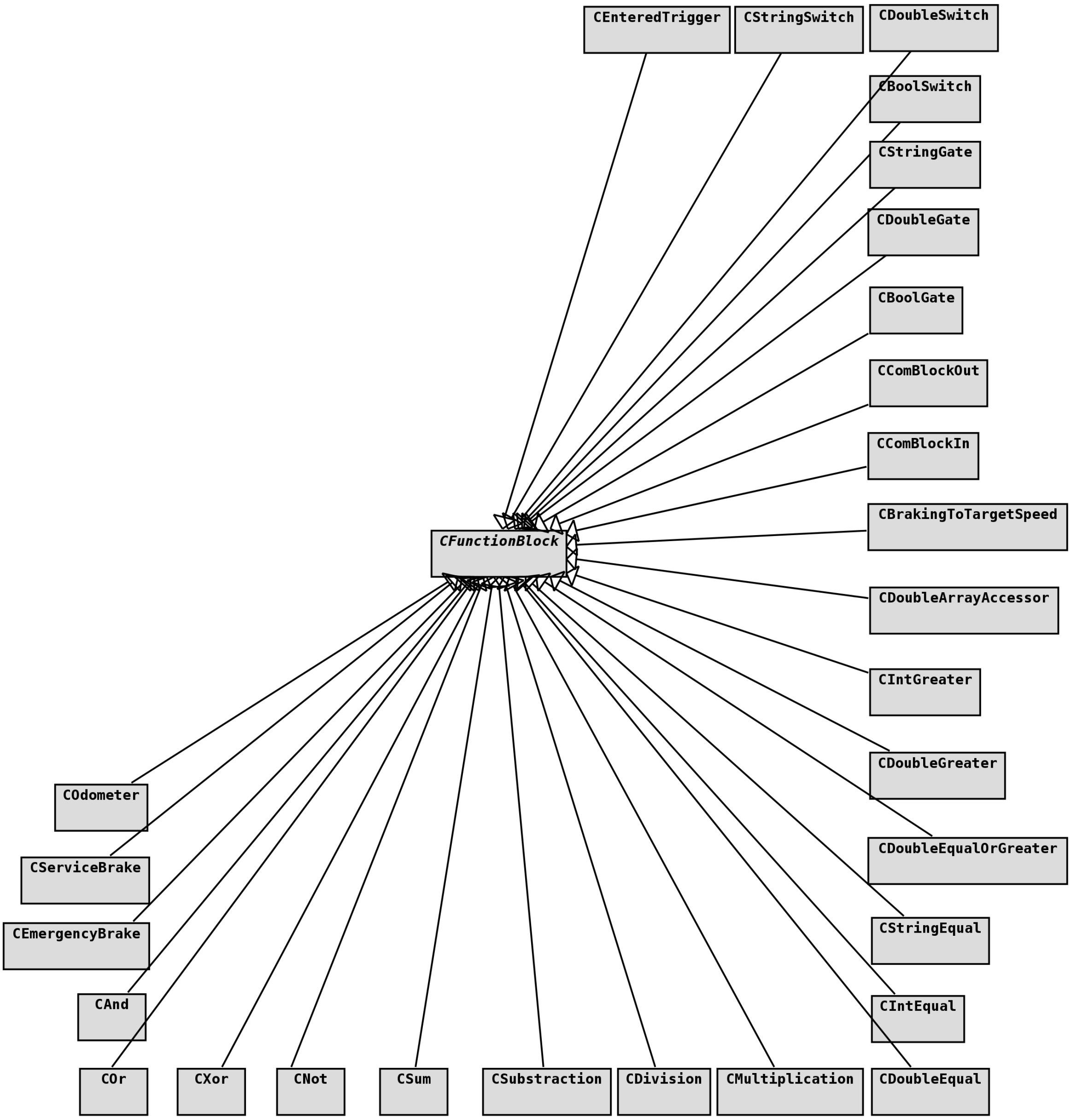


Figure 8.3.: Class diagram of the function block classes

objects that are not used in the current data flow. This modification of a currently unused object can lead to undefined and/or undesired behaviour not corresponding to the concrete model.

The concrete syntax of data flows between function block objects in the openETCS meta model in Subsection 7.3.2 defines DataFlow relationships as the related interconnection. Although function block objects can be reused in any graph of a model[4], this DataFlow relationship is unique. Therefore, an obvious and direct solution is to additionally design a class representing a data flow or rather an interconnection between two function block objects. This class should be instantiated[5] for each relationship in the openETCS model. The classes for the data flow interconnection are shown in Figure 8.4.

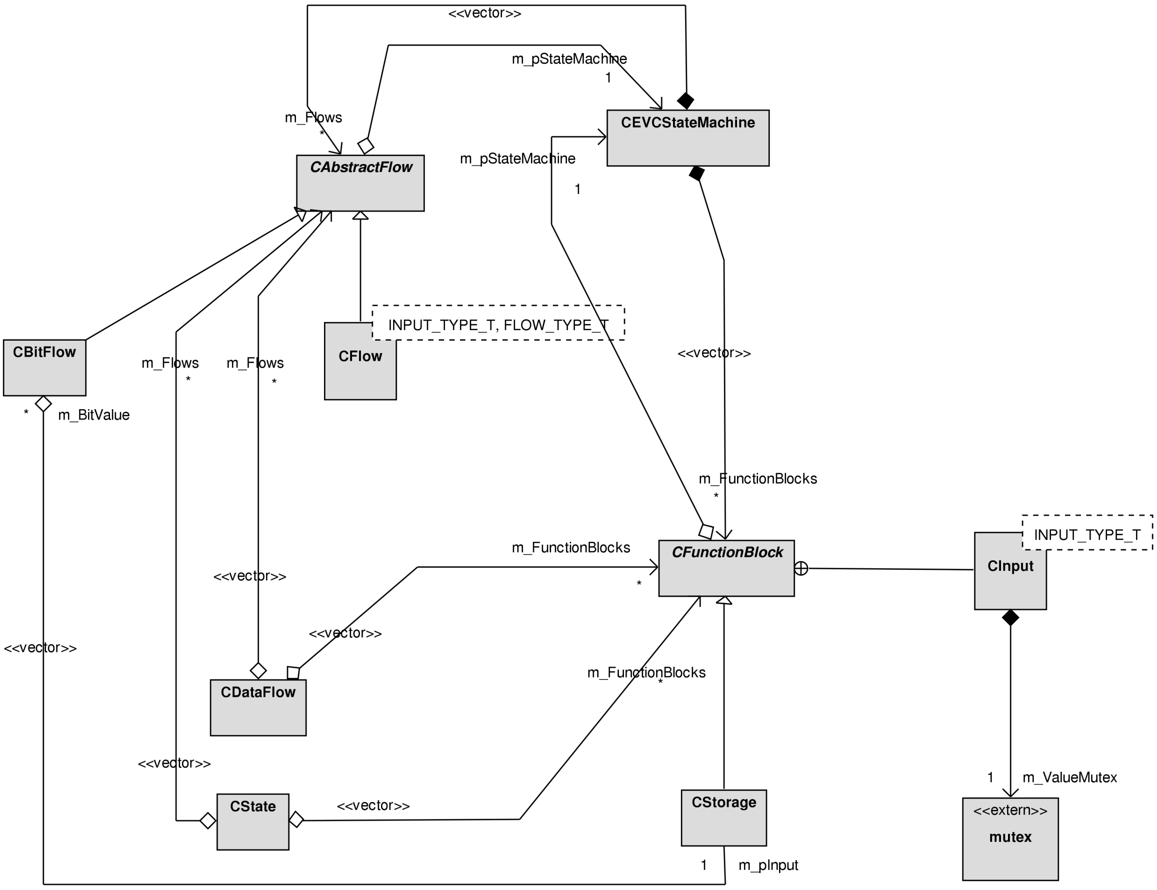


Figure 8.4.: Class diagram of the function block classes for the data flow

As data flows can be of different types (double, integer, string, etc.), CFlow, the additional class for interconnections, is a template class [79]. Its first template parameter `INPUT_TYPE_T`

[4] instance of the openETCS meta model
[5] generated by openETCS generator

defines the corresponding implementation or rather C++ type [79], like bool, double, int, etc. The second template parameter FLOW_TYPE_T defines the type of the related CInput object. This parameter has the default value CInput< INPUT_TYPE_T > and normally does not have to be set explicitly. It is used to define the attribute m_pInput of CFlow that is the connected CInput instance. m_pInput is not modelled as composition in Figure 8.4 because the usage of template parameters in UML class diagrams is limited.

The CInput template class, which is nested in the CFunctionBlock base class, is used to encapsulate each data flow input as property. Its template parameter INPUT_TYPE_T defines, as for the CFlow class, the concrete implementation data type. This means each input of a certain function block type or rather class is defined as attribute of the type CInput with the corresponding template parameter for the concrete data flow type corresponding to the meta model. Compared with the direct usage of an implementation data type, this has the advantage that the modification state of each input can be made available or visible to its possessing function block object. Furthermore, the access of inputs can be made thread-safe [7] by using a mutable exclusion (mutex) [7] object via the m_ValueMutex composition. It must be noted that the mutex class is located within the std:: name space of the standard C++ library [7], which is not explicitly modelled in Figure 8.4.

Unfortunately, the bit data flow type between oVariableStorage objects in the openETCS meta model (see Subsection 7.3.2) cannot be directly mapped to a data type in the implementation. Hence, the CFlow template class cannot be used for those data flows and the additional CBitFlow class was added, which is not a template class and is especially designed for data flows between CStorage instances. Correspondingly, the aggregation between those two classes has the two ends m_pInput and m_BitValue. The first is the input storage object, from the view of CFlow while the latter holds all outputs to other CStorage objects. It should be remarked that m_pInput is modelled as aggregation compared to the m_pInput because CBitFlow is not a template class.

Since all interconnections or rather all flows are assigned to a certain parent CEVCStateMachine object via the m_Flows composition, CAbstractFlow is used as base class for the both concrete flow types. Finally, to solve the initial problem of the general usage or rather execution of all existing interconnection / data flows, each CAbstractFlow instance requests via the m_pStateMachine aggregation if it is part of the currently executed data flow, which means an element of the m_Flows aggregation. If not, any message from a connected function block object using operator=() is ignored. Otherwise, the value is marshalled to the related input. Since the flows between function block objects that are part of an embedded state machine respectively a CControlFlow instance should only be executed if the related CState object is currently active, also CState posses an aggregation m_Flows to CAbstractFlow.

8.3.2. Driver Machine Interface Details

All classes especially related to the DMI are presented in a separated class diagram in Figure 8.5. CDMIObserver is the abstract observer [32, pp. 293-303], from which each concrete observer implementation must be derived and methods called by the CEVCStateMachine class must be implemented by overriding [79]. The parent CEVCStateMachine instance holds pointers to all registered, concrete observers in the aggregation m_Observers via the abstract base type to be

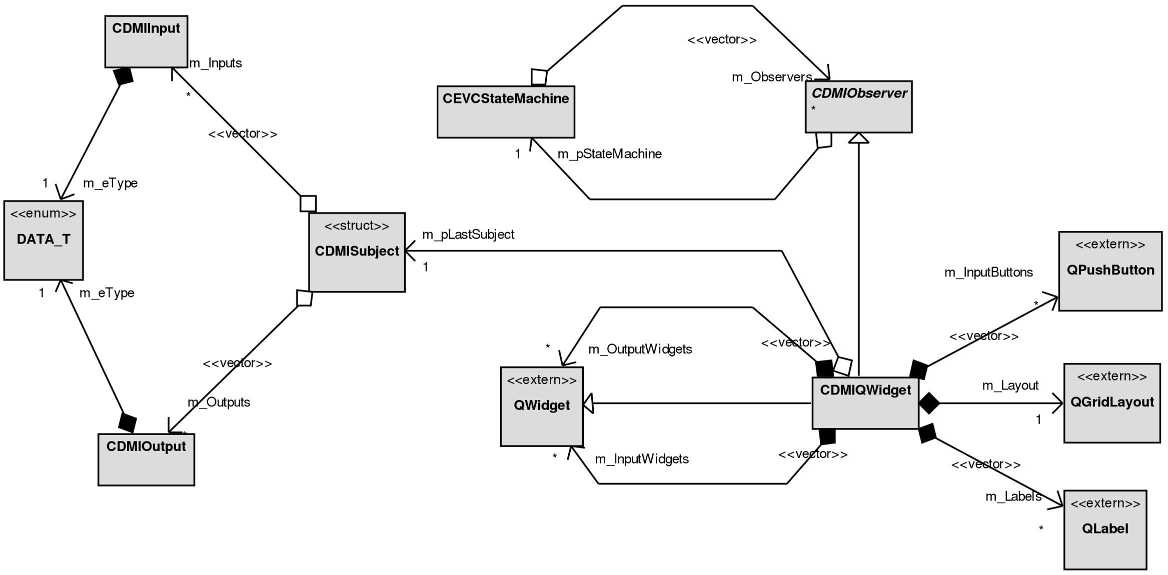


Figure 8.5.: Class diagram of the DMI classes

able to inform any observer about a changed subject. On the other hand, each observer only holds a pointer to one CEVCStateMachine object that it observes. Via this `m_pStateMachine` aggregation, an observer can access the currently active CDMISubject object to read data from the EVC to the driver or set data entered in the observer by the driver.

CDMIQWidget is a concrete observer implementation using the Qt 4 [59] framework. Qt 4 was chosen because it provides platform independent GUI development in the C++ programming language [79], which means in an object-oriented manner. This is required by Req.6 and Req.7. The concrete elements[6] for inputs and outputs are generated dynamically since those typically vary for different ETCS Modes or rather CEVCState instances. The input and output elements are stored in the `m_OutputWidgets` and `m_InputWidgets` compositions. For those, a pointer to the QWidget [59] class is used as data type. A QWidget is the very basic type of all classes in Qt 4 representing graphical elements. Which concrete, derived classes are used, depends on the CDMIInput and CDMIOuput objects in the current CDMISubject. Since this is of low relevance, those certain types will be not discussed here. The CDMIQWidget observer class itself is also a QWidget, which mainly has the advantage that it can be used as standalone dialogue but also can be integrated or combined with other Qt 4 elements if required. The inputs and outputs of the subject are aligned in a grid respectively in two columns: Outputs on the left side, inputs on the right side. The corresponding graphical arrangement is provided by the `m_Layout` composition to the QGridLayout [59] class. For the graphical definition of the meaning of each input or output element in CDMIQWidget, labels are used, which are located on the left side of each graphical element. Those correspond to the `m_Labels` composition to

[6] widgets in terms of Qt

the QLabel [59] class. Furthermore, each input device has a QPushButton [59] element to ensure that the driver can first enter or modify data in the widget and then can activate it by a single click on the button. Those buttons are stored in the `m_InputButtons` aggregation.

8.3.3. Language Details

Figure 8.6 sketches only the classes and relevant associations that are related to the ETCS language in a UML class diagram. CLanguage is the base class for all other already introduced language classes. Since a telegram or rather a CTelegram object can be used for several CBaliseDeviceIn and CBaliseDeviceOut instances, those are CEVCStateMachine global indicated by the `m_Language` composition. This association should hold all available instances of CTelegram, CPacket, and CVariable because CPacket and CVariable objects can also be reused for different CTelegram instances and CVariable objects for CPacket instances.

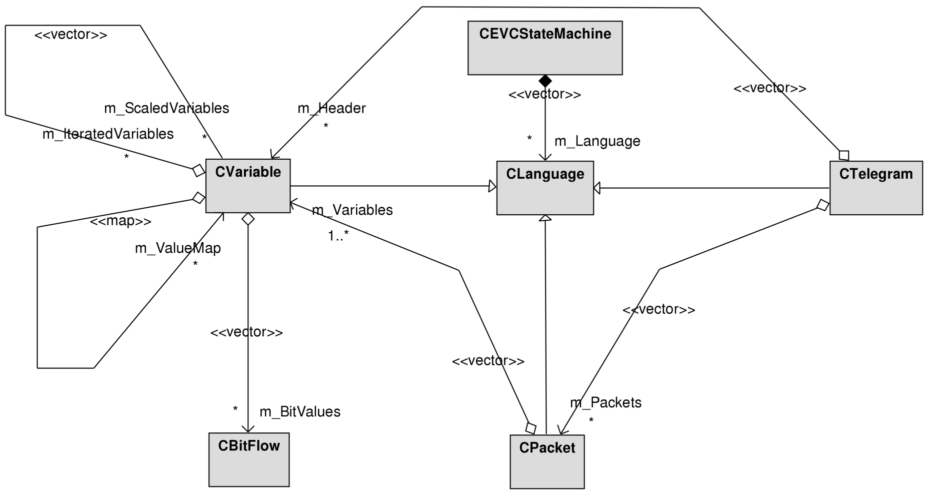


Figure 8.6.: Class diagram of the language classes

The bit data flows between oVariableStorage and oVariableStorage objects in oCommunicationReader and oCommunicationSender graphs in the openETCS model[7] are, like in function blocks, mapped to CBitFlow instances. Hence, CVariable has a aggregation `m_BitValues` to CBitFlow in Figure 8.6.

8.4. Behavioural Design

In contrast to the previous Section 8.3 about the structural design of the openETCS domain framework, this section describes its behavioural design. The main difference between the structural and the behavioural design is that the structural design is constant and cannot be

[7] instance of the openETCS meta model

changed by code generators if a statically typed programming language [79], as C++, is used. Nevertheless, the behaviour depends heavily on the instantiation of objects, which means here on the generated code and accordingly on the openETCS model. The consequence is that this section can only show examples of models and their corresponding generated code by UML diagrams for clarification.

Data Flow Example The data flow model in Figure 8.7 is used as example to describe the corresponding behaviour, which is explained as follows. This example is very similar to the one in Figure 7.15 that was used for the description of the mathematical model of the dynamic semantics in Section 7.7. It is only extended about three objects on the right side. Those and the new data flows between them add the following functionality: If the incremented oVariableStorage object “Value” (initial value 0) is greater than 1, the oStateGuard object, “Start→Running” is activated. Furthermore, Figure 8.7 should be a decomposition of the “Starting” state in Figure 7.16.

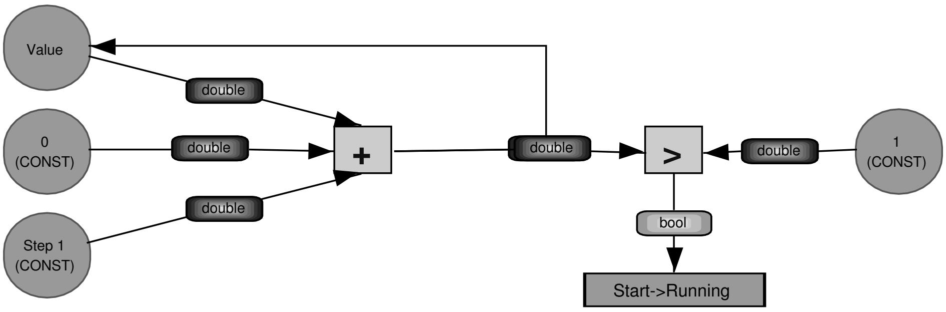


Figure 8.7.: Example of a gSubFunctionBlock graph with a simple data flow

Figure 8.8 introduces a UML interaction diagram [64] for two execution cycles ($k = 0, 1$) of the example mapped / transformed to the domain framework. According to the description in Subsection 8.3.1, all model objects are transformed to class instances of the openETCS domain framework. Those are referenced by the `m_FunctionBlocks` aggregation of the CDataFlow instance “DataFlow”. Due to the fact that an oStateGuard respectively CCondition object is used, “DataFlow” is part of the CControlFlow “Parent” (Figure 7.16). Figure 8.8 introduces the corresponding UML interaction diagram [64] for two execution cycles. Both execution steps k are started by the `Execute()` message from the “Parent” control flow object to its CDataFlow instance. To be precise, the control flow object must be a part of a superior CDataFlow object that itself is part of a CEVCState instance starting the execution chain. For simplification reasons, this is omitted in Figure 8.8. Neither, the CState object owning “DataFlow” is not relevant for understanding the data flow behaviour and is omitted, too.

After the first `Execute()` message “Value” holds the value 0, “Sum” has the output 1 and the “Greater” operator `false`. During the second `Execute()` message the output of “Sum” reaches 2, which causes that the “Greater” output switches to `true` and, as a consequence, triggers the “Start->Running” condition. The triggered condition locates the related CTransition object (not in the diagram) `pRelatedTransition` and places it on the the transition stack

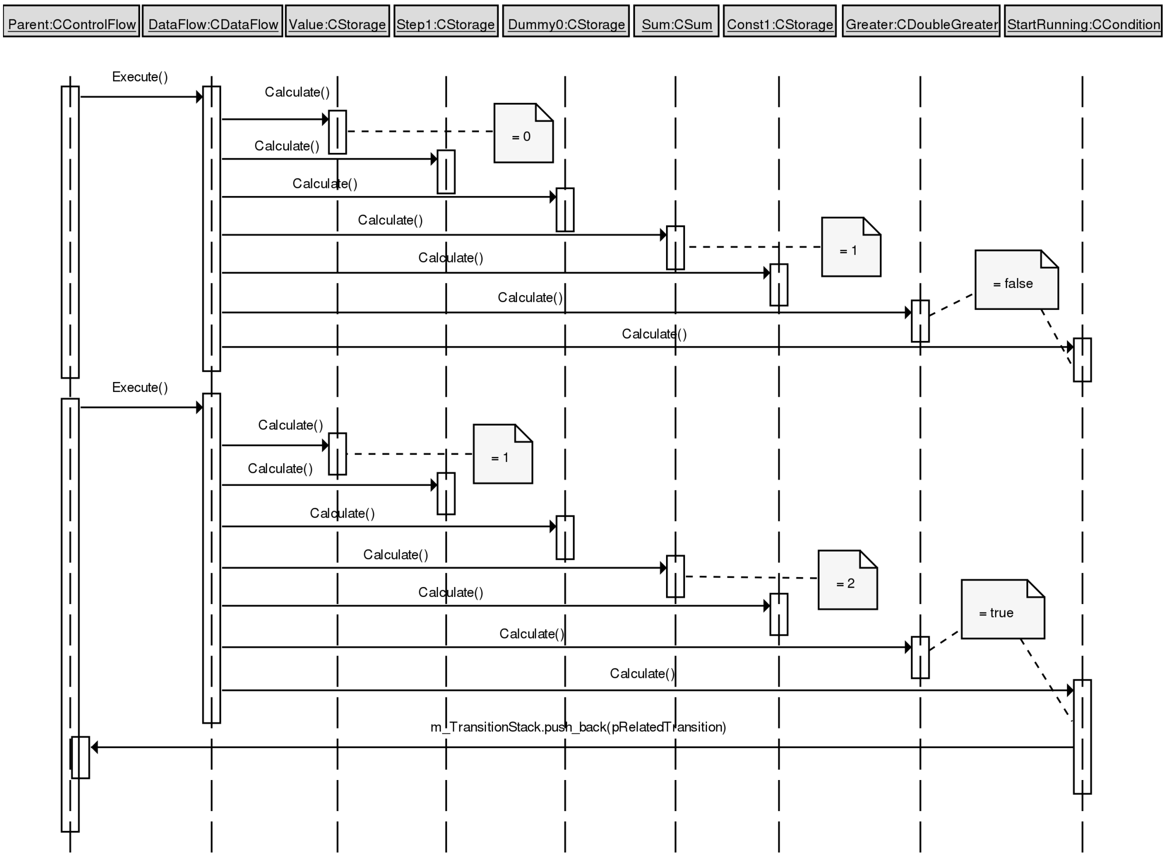


Figure 8.8.: UML interaction diagram for the example data flow in Figure 8.7

(m_TransitionStack) of the "Parent" control flow to initiate a state switch within the "Parent" control flow.

Control Flow Example As used for the data flow description above, the same combination of gEmbeddedStateMachine and gSubFunctionBlock graph is now employed to explain the corresponding behaviour from the perspective of the parent control flow. Like the data flow in Figure 8.7, all objects in the graph have to be transformed to objects of the openETCS domain framework. This does not only include oEmbeddedState→CState but also the transformation from Transition relationships to CTransition objects. Figure 8.9 shows the first execution cycles in a UML interaction diagram.

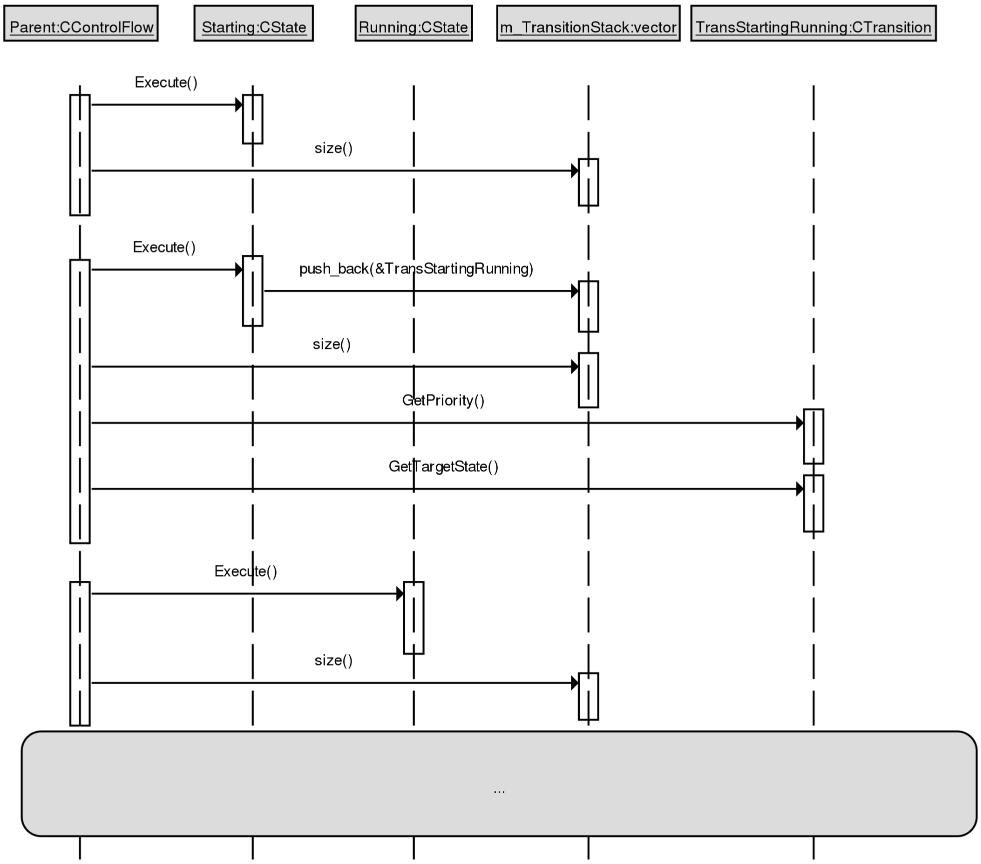
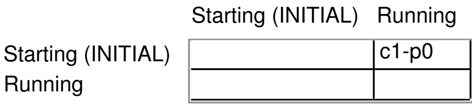


Figure 8.9.: UML interaction diagram for the example control flow in Figure 7.16

The "Parent" control flow object sends an `Execute()` message to the initial and current state "Starting". This messages corresponds to the `Execute` message in Figure 8.7. Therefore, the execution of "Starting" can be found in the previous interaction diagram. The first execution of the state and its data flow does not activate any CCondition object, which is checked by the "Parent" object by getting the size of the transition stack via the `size()` message. The second execution (see Figure 8.7) of “Starting” finally activates the “Start->Running” condition, which causes in Figure 8.9 that the reference of the related CTransition “TransSleepingRunning” is placed on the `m_TransitionStack` of “Parent” by the `push_back()` message. After the parent control flow noticed that the transition stack is not empty (result of `size`), it gets the priority by `GetPriority()` and the reference to the target state by `GetTargetState()`. The priority is used to determine, which new state is switched to in the case if the stack would hold more than one transition object. Finally, in the current execution step “Parent” sets internally its `m_pCurrentState` aggregation to the reference of “Running”. Therefore, in the next execution step “Parent” sends the `Execute()` to “Running” to execute its data flow and so on.

EVC State Machine Example Although the EVC state machine graph type gEVCStateMachine also defines a control flow, it differs a little bit from the gEmbeddedStateMachine graph type in the behaviour because it describes two nested state machines. The superior one is for the EVC Modes (CEVCState class) while the switching of ETCS Application Levels is nested within. Because the switching of the Application Level only means a context switch of the current CEVCState object `m_pCurrentState`, the levels are simply distinguished by a string stored in `m_CurrentApplicationLevel`. A certain class for Application Level switching, like CTransition, is accordingly not required because the condition objects of class CLevelCondition, which corresponds to oApplicationLevelType in the meta model, can directly write their level in the parent CEVCCondition object if they are activated by a Boolean data flow.

Thus, the preceding example for a control flow does not cover this speciality, and an additional example for an EVC is discussed. Figure 8.10 represents the transition matrix (of an gEVCStateMachine graph) of a strongly simplified EVC state machine. It only consists of the states “Starting”, which is the initial one, and “Running”. From “Starting” to “Running” exists exact one transition under the condition “c1” with the priority 0. It is obvious that this example does not correspond to any Modes or parts of the ETCS SRS and is only used to exemplary demonstrate the behavioural design of the EVC state machine. “Running” is available in Application Level 0 and 1, which means the oEVCState object must have two explosions: One for each Level. Figure 8.11 shows the gMainFunctionBlock graph for Application Level 0 and Figure 8.12 for Level 1. “Running” is only available in Application Level 1, but its explosion is not relevant for this example and accordingly omitted.



	Starting (INITIAL)	Running
Starting (INITIAL)		c1-p0
Running		

Figure 8.10.: Transition matrix for an simple EVC example

Both data flows of “Starting” use the already introduced and described counter construct. In Level 0, after one calculation step, the oApplicationLevelType object for Level 1 is activated by a true Boolean data flow, which means a context switch to Level 1 is then initiated. In Level 1, the counter has the limit 2. If it is reached, the condition “c1” is activated, with means a transition to the state “Running”. The UML interaction diagram with the primary objects

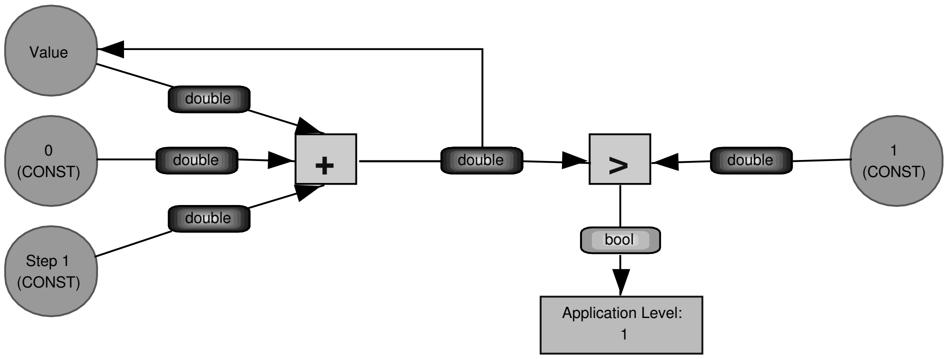


Figure 8.11.: gMainFunctionBlock graph of “Starting” state (Figure 8.10) in Application Level 0

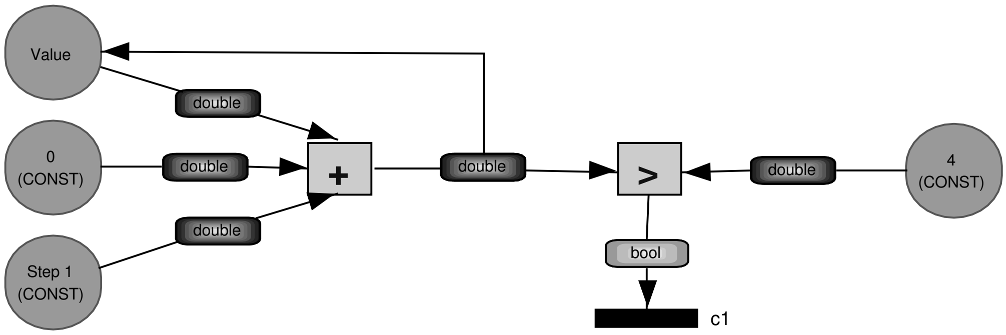


Figure 8.12.: gMainFunctionBlock graph of “Starting” state (Figure 8.10) in Application Level 1

for the first six calculation steps ($k = 0, 1, \ldots, 5$) is sketched in Figure 8.13.

Any execution is started by the external EVC actor via an asynchronous `Start()` message to the “StateMachine” object. At first, “StateMachine” executes the initial CEVCState “Running” (“Running” in Application Level 0) by a further `Start()` message. A running CEVCState object always starts for each independent data flow object in `m_CurrentDataFlow` for the current active Application Level an own thread [77] by a `DataFlowThread()` message to itself. In this example, only one CDataFlow object is defined. “Running” handles the execution of the data flow by a `Execute()` to the current CDataFlow object that is here “SDL0” at equidistant time points (see Section 7.6). For each `Execute()` message from the related EVC state object, the data flow object sends a `Calculate()` message to all CFunctionBlock objects in `m_FunctionBlocks`. At the first execution step ($k = 0$), only the value of the counter storage “Value” in Figure 8.11 is increased by one. At the second step ($k = 1$), the CLevelCondition object “L1” is activated by a Boolean true input. Therefore, it sends a `m_CurrentApplicationLevel = "1"` message to the parent CEVCStateMachine object by its `m_pParent` aggregation. Furthermore, it informs the parent CDataFlow object about an upcoming switch of the EVC state or/and the ETCS Application Level by the `m_bStateLevelSwitch = true` message. As a consequence, the execution of `DataFlowThread(0)` and `Start()` in “Running” is terminated and accordingly returned to the “EVC” object execution. This then directly starts the execution of the current active CEVCState object defined by `m_pCurrentState`, which is the same as before but now with the new Application Level 1 (`m_CurrentApplicationLevel`).

Again, for each CDataFlow object (here only “SDL1”) in “Running” for the current Application Level, a separated thread is started by the `DataFlowThread()` message that is for this example only one. This thread executes the related CDataFlow object “SDL1” by the `Execute()` message. For $k = 2$ and $k = 3$, only the value of “Value” is further increased. For $k = 4$, the output of the CSum object is 5, which causes the activation of the CEVCCondition “c1”. Thus, it places a reference of the corresponding CEVCTransition object “Tc1” of the transition stack of “StateMachine” by the `m_TransitionStack.push_back(&Tc1)` message. The `m_bStateLevelSwitch = true` is used to inform the “StateMachine” object about an upcoming transition to a another CEVCState object. Like in Application Level 0 before, then the execution of `DataFlowThread()` and `Start()` is terminated.

The “StateMachine” gets the target state of the “Tc1” object on the transition stack by the `GetTargetState()` message that is here the only CEVCTransition object on the stack. In the case of several objects, their priority property would be used to determine, which one is going to be executed. The target state `m_pTargetState` of “Tc1” is “Running”, which reference is then stored in `m_pCurrentState`.

Since “StateMachine” is still started by the EVC, it starts the new state “Running” by a `Start()` message. As for “Running”, a thread for the only data flow object “RDL1” in “Running” for Application Level 1 is started by `Start()`. This executes “RDL1” via the `Execute()` messages ($k = 6, 7, \ldots$). Further execution steps are not explained in this example.

Finally, “StateMachine” is stopped by the EVC through a `Stop()` message.

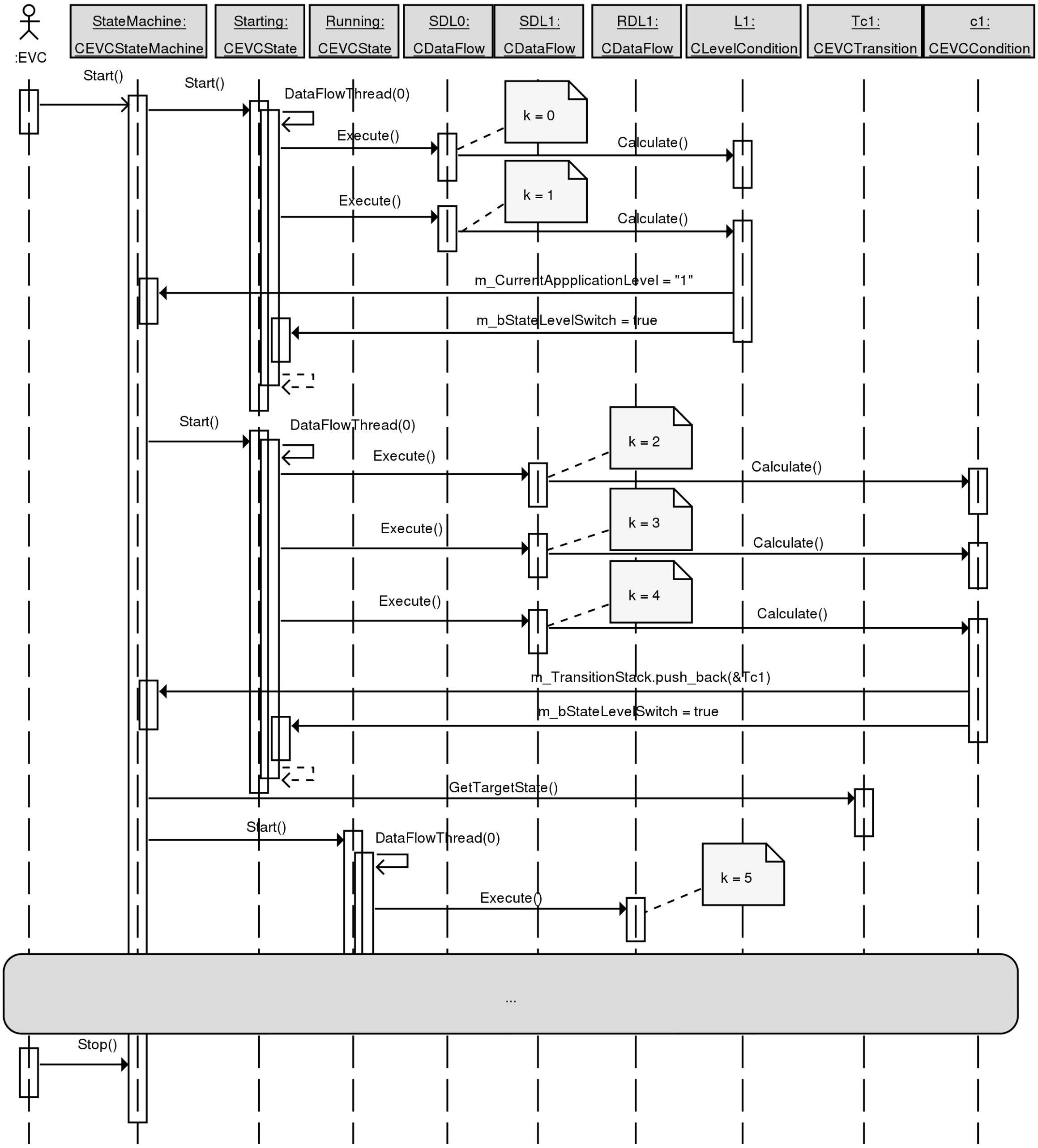


Figure 8.13.: UML interaction diagram for EVC state machine example in Figure 8.10

8.5. Deployment Design

The deployment design describes how the source code and the compiled binary code is distributed on different platforms for their execution. For openETCS and its domain framework, this is of special interest regarding the separation of platform-specific adaptations from the directly generated code by hardware virtualisation, which was discussed in Chapter 6.

For clarification, the openETCS architecture is divided in three categories:

Computational Independent Model (CIM) is the part of the DSL that does not hold any certain information about computation structure and processing of the system [44, p. 2-5]. This corresponds to the openETCS model.

Platform independent model (PIM) provides information about computational and processing structure of the system without any information depending on a certain target platform [44, p. 2-6]. This corresponds to the openETCS domain framework (see Req.7) and the code generated from the openETCS model / CIM.

Platform specific model (PSM) extends the PIM or is a transformation of it to be applicable on a certain platform [44, p. 2-6]. In the openETCS architecture, this corresponds to possible adaptations for concrete hardware devices, like sensors and actuators (Section 7.2).

While CIM and PIM are already defined for the openETCS case study, the PSM is defined in the deployment. Due to the fact that this work primary focusses on the CIM and PIM, platform-specific adaptations are only used for testing and simulation purposes. Nevertheless, the deployment design should provide a general approach for extending the PIM about PSM elements of any kind. Figure 8.14 introduces the deployment of the openETCS domain framework with the separation of PIM and PSM as UML deployment diagram [64]. Both are located on the target platform, the EVC, but are executed in separated execution environments. Both execution environments can communicate (over a network link) with the EVC but not directly with each other. Communication between them is only possible via the EVC, on which this communication could be limited by a firewall or other mechanisms.

All source artefacts located in the PIM execution environment manifest together the libopenETCSPIM component, which is a library. This library represents the platform independent implementation (parts) of the openETCS domain framework. On the other hand, the libopenETCSPSM is the manifestation of the platform specific implementation base within the PSM execution environment. Each source artefact is shortly explained in the following:

EVCStateMachine Holds all source code that is basically needed by the CEVCStateMachine class. This includes all required types for data flows.

ControlFlow All sources for control flows, including CControlFlow and CState.

FunctionBlocks Sources of all platform independent function block classes.

Language Includes all sources for the ETCS language classes.

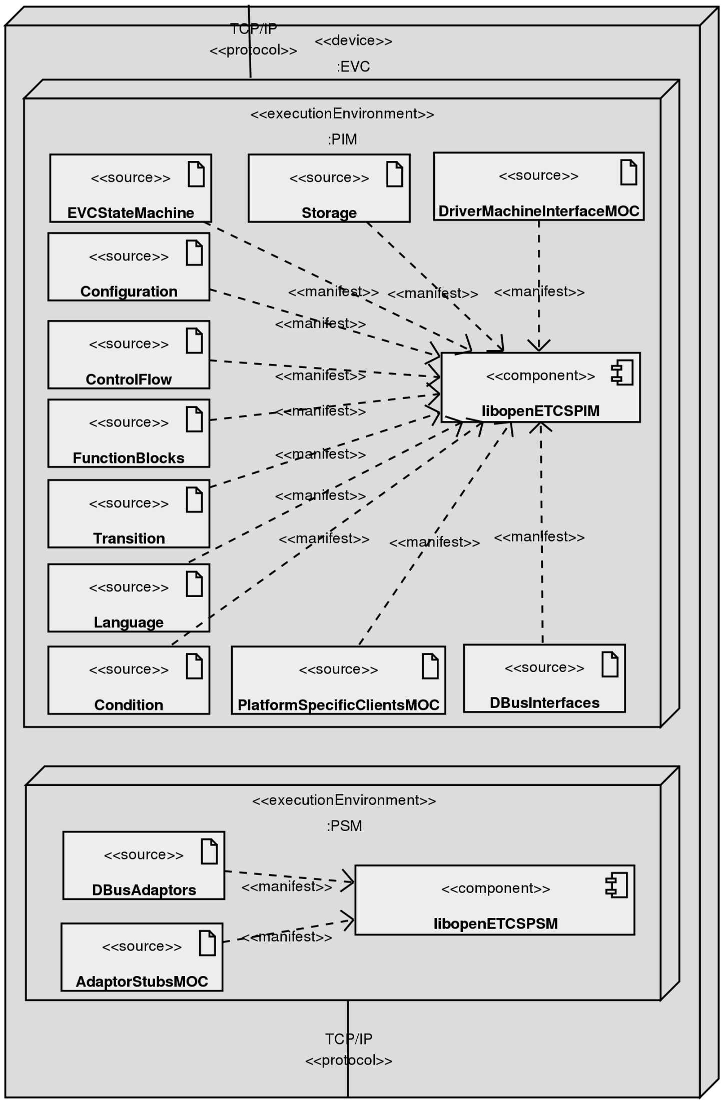


Figure 8.14.: UML deployment diagram of the openETCS domain framework

DriverMachineInterfaceMOC Holds the implementation of the concrete DMI Qt 4 observer CDMIQWidget.

PlatformSpecificClientsMOC Includes all source code for accessing the platform-specific extensions for COdomoter, CEmergencyBrake, CServiceBrake, CBaliseDeviceIn, and CBaliseDeviceOut. For testing purposes without platform-specific adaptations, also a stub[8] implementation is provided for each class.

Storage Source code of the CStorage class. It is provided in a separated artefact because it is used by several other artefacts like EVCStateMachine and Language.

Condition Source code of the CCondition class, which is also provided in a separated class due to multiple dependencies.

Transition Source code of the CCondition class, which is separated for the same reasons as for CCondition and CStorage.

Configuration Provides type definitions and global constants for the whole domain framework.

DBusInterfaces Holds the source code for all D-Bus interfaces [31] for platform specific classes.

DBusAdaptors Provides the source code for all D-Bus adaptors [31] for platform specific classes. The usage and integration of D-Bus [31] will be additionally discussed at a later point of this section.

AdaptorStubsMOC Source code of all stubs classes for D-Bus adaptors. For concrete platform-specific adaptations, those should be used or rather implemented by class inheritance [79].

All source artefacts with the postfix “MOC” in their name require the usage of Qt 4’s meta object compiler (moc) [59].

As each library component is located in another execution environment respectively another virtual machine, a middleware for IPC communication has to be used (see Req.8), which was partly discussed in Subsection 6.2.4. For this purpose D-Bus is employed, which was also introduced in Subsection 6.2.4. The D-Bus daemon [31] is executed directly on the EVC and therefore enables the communication between both execution environments but only through defined interfaces (see Chapter 6). Details about this interconnection are shown in Figure 8.15 as UML component diagram [64].

The interfaces required by the libopenETCSPIM component are IEmergencyBrake, IOdomoter, IServiceBrake, IBaliseDeviceIn, and IBaliseDeviceOut. Those define the interfaces that have to be implemented by suppliers of hardware components defined in the ETCS SRS.

[8] empty

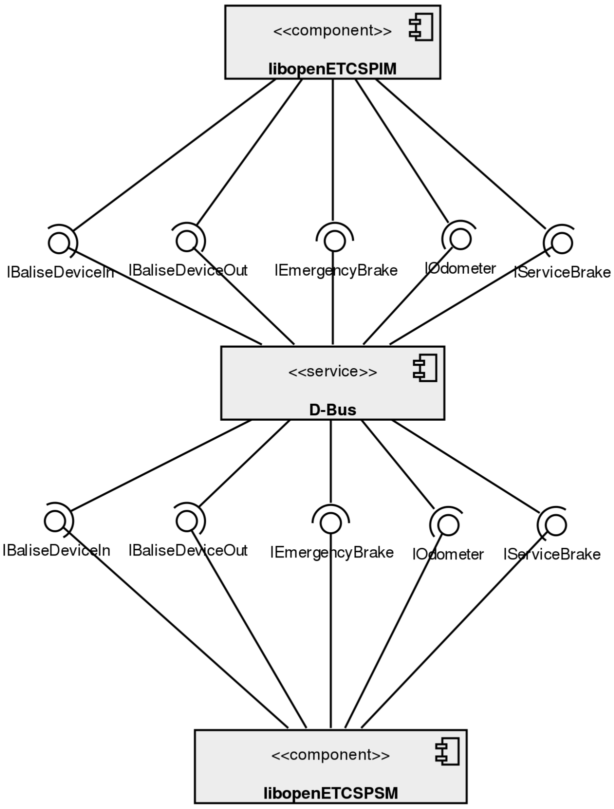


Figure 8.15.: UML component diagram for PIM and PSM of openETCS domain framework

Hence, those classes can only have a stub implementation in the PIM because those hardware components are always specific. The D-Bus component / daemon provides those interfaces to the libopenETCSPIM component while it requires them from the libopenETCSPSM component. In that way, messages from and to the specific classes in the PIM can be dispatched to the PSM without any direct connection in an object-oriented manner.

Of course, it is possible, as proposed in Section 6.2, to place each hardware interface implementation in an own execution environment or rather virtual machine and to connect each to the D-Bus component in the EVC. Nevertheless, the domain framework / PIM only defines the interfaces to the PSM but does not define how (and where) the concrete implementations are provided because such details are not relevant for the domain framework deployment design in general.

Finally, it must be emphasised that for an executable EVC state machine not only the source artefacts from the openETCS domain framework are required but also the generated code from the openETCS model, which instantiates the classes of the domain framework. However, that generated code is mutable because it depends on the openETCS model and therefore is not a direct part of the domain framework deployment design.

D-Bus Integration Since the preceding paragraphs in this section mainly discussed the general integration of the D-Bus as middleware, the following paragraphs will explain the details about the D-Bus usage. The defined interfaces IEmergencyBrake, IOdomoter, IServiceBrake, IBaliseDeviceIn, and IBaliseDeviceOut are not directly types or rather abstract classes [79] and therefore cannot be simply integrated into the class structure by inheritance. Therefore, the corresponding openETCS domain framework classes must make use of the interface in a different manner, which is shown in Figure 8.16. The function block classes that are all located in the PlatformSpecificClientsMOC artefact have all a composition `m_pProxy` to the corresponding interface[9] type. Since the general D-Bus implementation provides only a GLib [31] API, which is not object oriented, its direct usage in the openETCS domain framework would cause a break with the so far pure object-oriented design and implementation. Thus, as for the concrete observer implementation CDMIQWidget, Qt 4 is employed because its D-Bus module [59] provides an object-oriented API. Accordingly, the platform specific classes are not only CFunctionBlock types but additionally inherit from QObject to be able to use Qt 4's signal and slot mechanism [59]. The function block types use a `m_Mutex` composition to ensure thread-safe manipulations of their attributes.

On the one hand, D-Bus itself defines interfaces that can be used as proxies for the communication and, on the other hand, adaptors that must be (re-)implemented with the "real" implementation [31]. Interface and adaptor are generated from an XML file that defines the messages and signals of an adaptor (and its interface), which is sketched in Figure 8.17. The contents of the XML interface definition can be found in Appendix D. It must be noted that the generated interface classes are no interfaces in the meaning of abstract classes [79], which are types that cannot instantiated directly. Instead, they can be interpreted as proxy objects that can be used in place of the corresponding adaptors. The artefacts are generated in the case of the Qt 4 D-Bus API by the "qdbusxmll2cpp" application [59].

[9] prefixed by an "I"

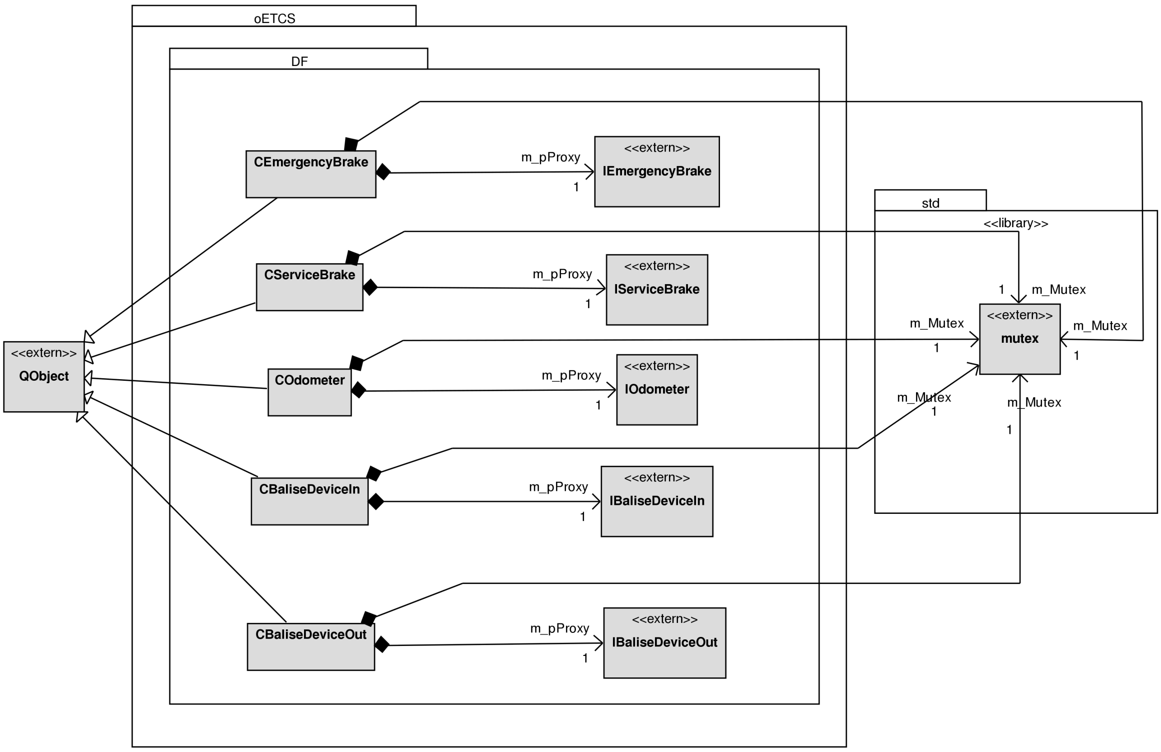


Figure 8.16.: UML class diagram of the integration of the platform specific interfaces

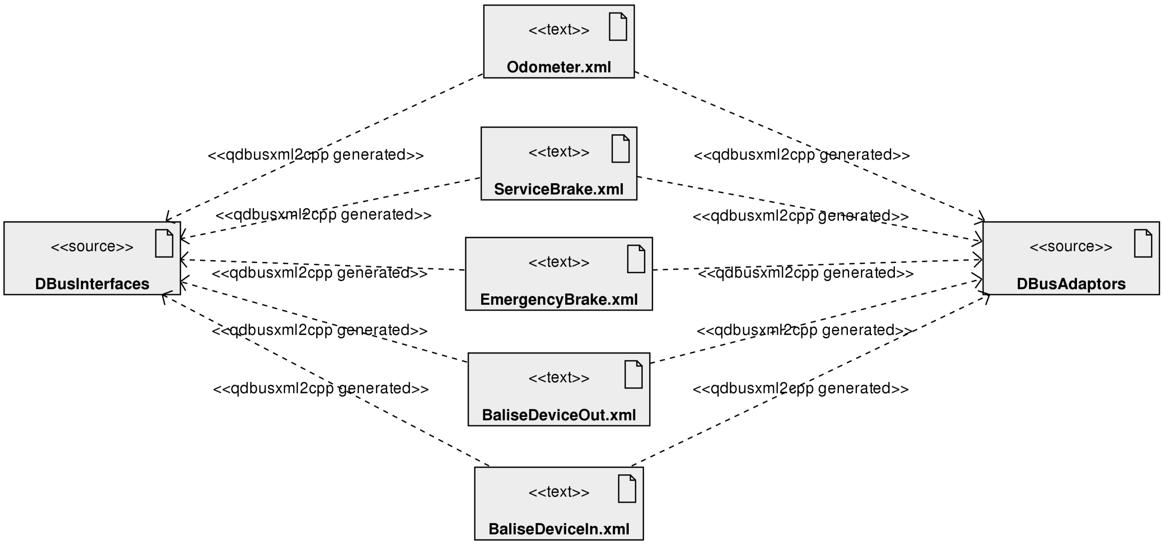

<<text>> Odometer.xml
<<text>> ServiceBrake.xml
<<text>> EmergencyBrake.xml
<<text>> BaliseDeviceOut.xml
<<text>> BaliseDeviceIn.xml
<<source>> DBusInterfaces
<<source>> DBusAdaptors
<<qdbusxml2cpp generated>>

Figure 8.17.: UML deployment diagram of the adaptor and interface artefacts

According to the discussion about security problems caused by platform-specific adaptations in Chapter 6, implicitly, only direct communication – in form of method calls – from the PIM to the PSM is allowed to further minimise the possible, malicious influence of platform-specific adaptations on the PIM. Hence, communication from the PSM to the PIM is only possible through the D-Bus signalling mechanism [31], which avoids that the adaptors in the PSM have any knowledge about connected proxies in the PIM.

In contrast to the interface classes, the adaptor classes cannot be used directly since the messages and signals defined in the XML file must be implemented. This implementation is an often recurring task because the implementation must be adapted for each (hardware) interface even for simulative implementations for testing. Therefore, a stub class is provided for each adaptor class, which methods only have to be extended by class inheritance and overriding [79]. The stub classes are shown in Figure 8.18. Similar to the function block classes, also the corresponding adaptor stubs, CBaliseDeviceIn, CBaliseDeviceOut, CEmergencyBrake, CServiceBrake, and COdomoter inherit from QObject to be able to use the Qt 4 signal and slot mechanism. The classes are located within the “PS”[10] package of the openETCS domain framework package, which means in C++ in the `::oETCS::DF::PS` name space. In contrast to other middleware, like CORBA, adaptor implementations for the Qt 4 D-Bus API are not done by inheritance from the adaptor classes but by an association of the generated adaptor to a QObject that provides the implementation of the adaptor methods. Thus, these associations are not sketched in Figure 8.18.

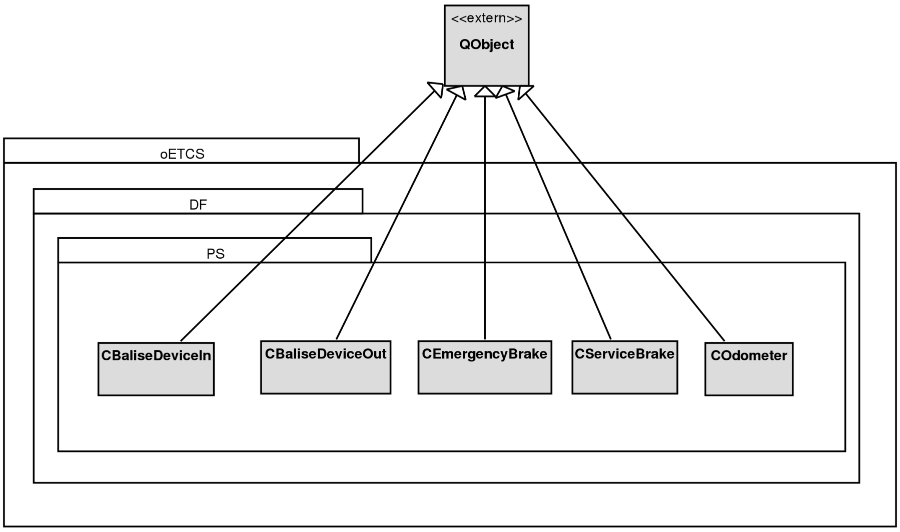


Figure 8.18.: UML class diagram of the adaptor stub classes

The UML deployment digram in Figure 8.14 only refers to two library components: One for the platform independent and the other for the platform specific implementation. Of course,

[10] abbreviation: platform specific

these two libraries alone cannot build executable binaries for the two execution environments. As already discussed in this chapter and according to Req.6 for the PIM, this is done by the instantiation of the openETCS domain framework classes by the code generator. For the execution of the PSM, any kind of adaptor implementation is required even if it only consists of stub classes. The deployment including these instantiations for PIM and PSM is shown in Figure 8.19.

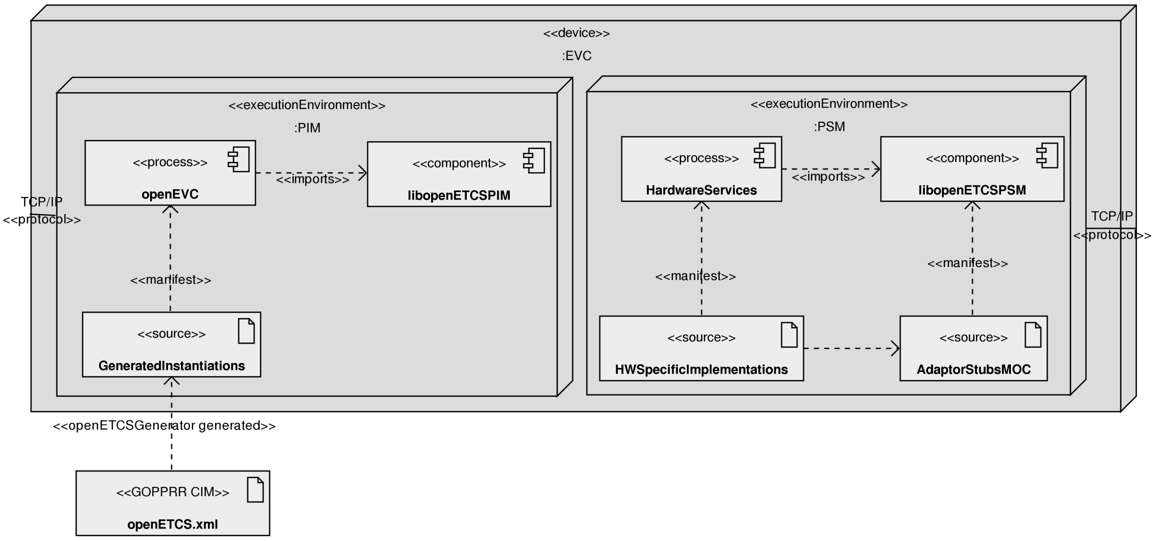


Figure 8.19.: UML deployment diagram of the instantiation of PIM and PSM

The openETCS.xml artefact holds the CIM respectively the openETCS model in the GOPPRR meta meta model (see Chapter 4). The openETCSGenerator (to be discussed in Chapter 9) application uses it to generate the C++ source artefact that holds the corresponding instantiations of domain framework classes and the `main()` function [79]. The openEVC process / binary is manifested by the GeneratedInstantiations artefact and imports or rather links against the libopenETCSPIM component with the platform independent types. The HWSpecificImplementations source artefact holds according to its name the concrete platform and/or hardware specific implementations. Currently, there exists only a simulative implementation, which will be explained and used in Chapter 11. This specific artefact uses or rather includes the AdaptorStubsMOC artefact, which holds the before presented D-Bus adaptor stub classes in Figure 8.18. Obviously, the HardwareServices process / binary must be manifested by the HWSpecificImplementations artefact and is linked against the libopenETCSPSM component.

8.6. Implementation

The previous sections in this chapter described the design of the openETCS domain framework based on UML diagrams. Since UML is a very general description formalism for object-oriented programming languages, those typically cannot provide certain information about the implementation in a concrete programming language. Mainly, two strategies exist for implementing a software design defined by UML:

1. manually implement the complete design
2. generate as much as possible code from the UML model and manually implement missing parts

As already illustrated in Chapter 3, general modelling formalisms respectively meta models normally do not provide enough abstraction to enable full code generation. However, strategy 2 is anyway preferable because the workload is in any case smaller compared to a complete manual implementation.

As for this work the tooling for the domain framework development is not of main interest, a description of this specific tool chain is omitted.

8.6.1. Programming Language and Target Platform

According to Section 8.5, the domain framework is the PIM of the openETCS architecture and should be (mostly) platform independent. Hence, the C++ programming language is chosen for the implementation because it provides object-oriented programming (corresponding to Req.6) and is suitable for technical applications. Since the goal of this case study is primary a proof of concept, the new C++ standard also called C++11 [7] is already used, but which is currently in draft status. Nevertheless, it provides certain advantages for the domain framework implementation. For example, the implementation must establish a sample system (described in Section 7.6) with a constant sample time. This is related to the scheduling of the executing platform, which normally refers to an operating system. Since interfaces to operating system functionality, e.g. system calls [77], generally differ much, access via C++ objects, methods, and functions [7, Ch. 30] that are available for any platform providing a C++ compiler is preferable. The gcc or rather the g++ [33] compiler is used for the software development of the openETCS case study because it is developed under a FLOSS license and supports since version 4.4.7 the draft standard C++11. Although the choice for the target platform should not be relevant for the PIM, the usage of Linux [47] provides the integration in a platform that is also FLOSS.

Obviously, not all elements of the domain framework implementation can be discussed in detail. Therefore, the following subsections discuss certain parts of the domain framework implementation that are of highest relevance and/or interest. The full software reference of the openETCS domain framework is located in Appendix D.

Before proceeding with concrete implementation examples, it is noted that all classes of the openETCS domain framework are located in the `::oETCS::DF` C++ name space [79].

8.6.2. Data Flow Implementation

Most of the functionality of CFunctionBlock elements is simple to implement, like mathematical operations, but also the implementation of braking curves is mainly a transfer of a mathematical

/ physical model to C++ code. More crucial for the correct operation of any data flow is the execution in equidistant time points, which will be discussed here.

Section 8.4 demonstrated that the cycles are generated in the CEVCState class by an own thread for each independent data flow. Those threads are implemented in the following method:

```
void CEVCStateMachine::CEVCState::DataFlowThread(const unsigned int& iIndex) throw();
```

Since for each independent CDataFlow object an additional thread of this method is started, `iIndex` determines for which CDataFlow object in `m_CurrentDataFlow` the thread is responsible. In the implementation, the `m_CurrentDataFlow` aggregation is declared as

```
::std::vector< CDataFlow* > m_CurrentDataFlow;
```

Creating threads according to C++11 is done by using the `::std::thread` class [7, Ch. 30]. Hence, all data flow threads are created and started by the lines of code in Listing D.1 in Appendix D.

These are located in the definition of the following method:

```
void CEVCStateMachine::CEVCState::Start()  throw(::oETCS::DF::Error::CException)
```

Since the `CEVCStateMachine::CEVCState::Start()` method should be blocking and should only return if all threads are terminated, the `::std::thread::join()` method is used, which blocks until the corresponding thread has been terminated in Listing D.2. While this is mainly an object-oriented and platform independent way of starting a group of worker-thread, the thread-method implementation is more complex and therefore of more interest.

The execution of the related CDataFlow object at equidistant time points must be implemented, which is done in corresponding CEVCState method:

```
void CEVCStateMachine::CEVCState::DataFlowThread(const unsigned int & iIndex)  throw()
```

Furthermore, all `m_Threads` must be synchronised in that way that the execution of each thread should start for every k at the same time point $t = kT_s$ (see (7.1)). This problem is related to real-time scheduling [77, pp. 457-504]. Although this is very difficult to realise in real technical systems, the time difference between the execution start points should be minimised. Figure 8.20 sketches a sample execution in the time domain for two unsynchronised threads. Besides that the second thread always starts the execution with a delay Δt_i, those delays also differ: $\Delta t_1 \neq \Delta t_2 \neq \Delta t_3 \neq \ldots$. As already discussed in Section 7.6, this can lead to an undefined behaviour of the data flows in the time domain. The ideal sample execution for two threads is shown in Figure 8.21

The execution is implemented as a loop that calculates for each execution cycle k the consumed time and puts the thread for the remaining time to the next execution time point $(k + 1)T_s$ to sleep. The corresponding C++ source code can be found in Listing D.3 in Appendix D.

At the very first execution of the while-loop, the time point of the execution start is measured once (Line 21) by the `::std::chrono::high_resolution_clock::now()` method, which is also a new asset of the C++11 specification. The time for putting the thread to sleep state is calculated in Line 105 while `SAMPLE_TIME` is the system constant sample time T_s, which is typically in the range of milliseconds.

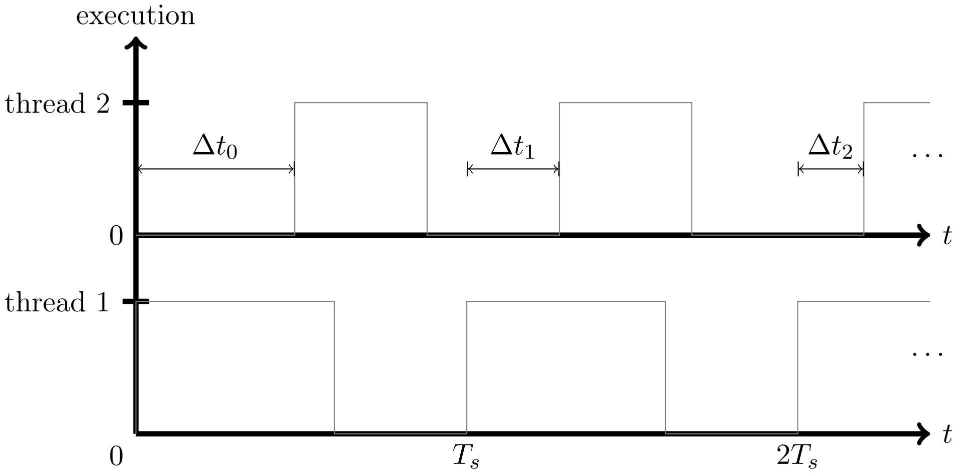


Figure 8.20.: Temporal execution of two unsynchronised data flow threads with a constant sample time T_s

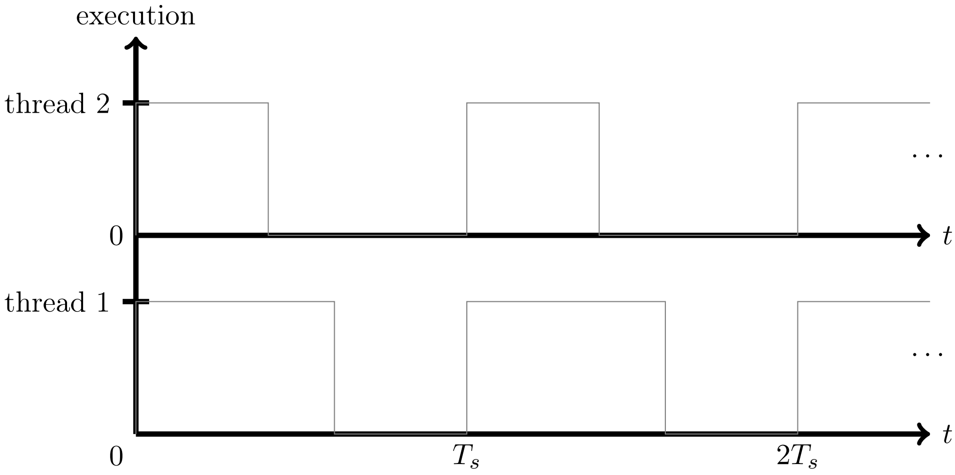


Figure 8.21.: Ideal temporal execution of two synchronised data flow threads with a constant sample time T_s

To synchronise all threads right from the beginning, the Lines 3 - 18 are implemented. `m_Barrier` is of type `::std::condition_variable` and also part of the new thread library [7, Ch. 30] of the C++11 draft standard. Its `wait(::std::mutex)` method locks a calling thread until `notify_all()` is executed from any other thread. It seems that this is the best that can be implemented to avoid delays as in Figure 8.20 with the usage of soft real-time [77, pp. 457-504] mechanisms. Since hard real-time [77, pp. 457-504] mechanisms and interfaces are mostly platform specific, those cannot be used in the domain framework because it represents the PIM. Hence, hard real-time mechanisms could only be integrated in the PSM.

Unfortunately, it is crucial that no thread starts its next execution before all others are finished. Otherwise, it cannot be assured that all used CEVCTransition, CTransition, and CLevelCondition objects are already calculated and a transition to a new CEVCState or CState object is not ignored. Thus, before calculating the sleep time and putting the thread to sleep in Line 111, all parallel executed threads must be synchronised. This is done by the same construct used for the initial synchronisation (Lines 3 - 18) but extended about the evaluation of activated transitions in Lines 46 - 101. In the case that the thread is not the last active thread, which is determined by `m_iLockedThreads == (m_Threads.size() - 1)`, it is simply blocked by the `m_Barrier` object. In contrast to the initial synchronisation, the last thread that enters the block not only notify all others via `m_Barrier` to proceed with the execution but also checks the stack of the `m_TransitionStack` with the activated CEVCTransition objects of the current cycle k. If more than one transition was activated, the one with the highest priority is chosen and set in Line 76. A consequence is that the slowest thread or rather the thread with the latest ending time point of its `m_CurrentDataFlow[iIndex]->Execute()` execution defines the blocking / sleeping time for all others by the synchronisation. Although this can lead to problems if a thread is severe delayed, it is necessary to guarantee that CEVCTransition are always evaluated properly at the end of each execution k.

At the very end of the while-loop in Line 122, the start time for each thread is calculated instead of measured as for the first execution in Line 21. This has the advantages that global drifts in execution time are avoided and only local drifts may occur. Within a soft real-time system, the absence of drifts cannot be guaranteed because the execution of all processes is managed by a global scheduler that works under the regime of the operating system kernel. Therefore, kernel processes may always block the execution of a user space process or thread even if it has a high priority [77, pp. 457-504]. The difference between global and local drifts are exemplary shown for one thread in Figure 8.22.

In the case of a global drift, the delay of the execution start Δt delays all further execution about the same Δt because the start time is measured, which includes the delay. Even further delays would be accumulated, so that the execution would end to be completely unsynchronised with the desired sample time points kT_s. Since the calculation of the new start time, instead of measuring it, does not take any occurred delay Δt into account, the next execution starts at the correct time point $2T_s$ for the local drift.

Of course, a general constraint to avoid drifts is that the minimal required execution time is not bigger than the sample time: $t_e \leq T_s$. To this property is also referred as scheduleability [77, pp. 457-504] of a real-time system.

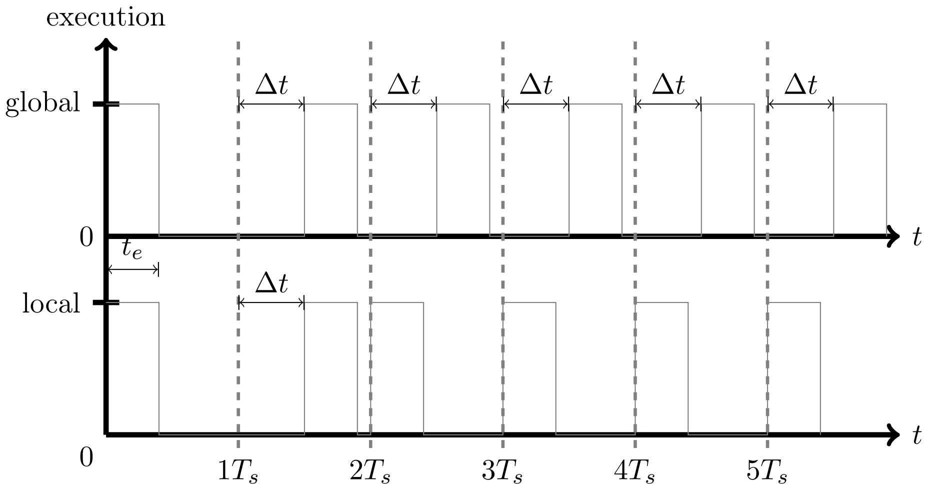


Figure 8.22.: Example of global and local execution drift

8.6.3. Control Flow Implementation

As the execution of CControlFlow objects is driven by the CDataFlow objects and CControlFlow objects are always a part of CDataFlow objects, their implementation is not relevant for a detailed discussion here. While the CDataFlow instances handle the transition to new CEVCState objects by their `m_TransitionStack`, CControlFlow must take care of transition between its CState objects by itself. In contrast to the transition between CEVCState instances, the evaluation of transition between CState objects is done in the currently executed CState object and not in the parent CControlFlow instance. This simplification can be done because a control flow cannot be executed in parallel as independent data flow.

Thus, a CState object executes all CFunctionBlock objects `m_FunctionBlocks` and afterwards evaluates its transition stack `m_TransitionStack`, which is implemented in Listing D.4 in Appendix D.

8.6.4. EVC State Machine Implementation

Since the core functionality of the openETCS domain framework or rather the execution of the currently active data flows is located in CDataFlow, the transition between EVC Modes and switching to ETCS Application Levels is mostly implemented in CDataFlow. Also, the `CEVCStateMachine::Start()` should be non-blocking according to the behavioural design in Section 8.4. Hence, the execution of CEVCState must be done in a separated thread that is created and started by that method. The thread is implemented in a method of CEVCStateMachine:

```
void CEVCStateMachine::StateThread()  throw()
```

Similar to the CDataFlow execution by CEVCState, the `::std::thread` is used to start the method in a separated thread. Its implementation is quite simple and can be found in

Listing D.5 in Appendix D.

It only executes a while-loop as long the internal Boolean attribute `m_bStarted` is true. This can only turn into false if the CEVCStateMachine instance is explicitly stopped by the `Stop()` method or if an exception is thrown in the executed current EVC Mode `m_pCurrentState` or rather in its data flows. If the execution of the current EVC Mode in Line 6 is stopped due to a Mode transition or Application Level switch, the while-loop simple starts in its next run the new active CEVCState object or with the new `m_CurrentApplicationLevel` or even with a new combination of both.

To stop the execution, only the attribute `m_bStarted` has to be set to false, which informs the current thread about the stop request and waits for the state thread to terminate by the `::std::thread:join()` method in Listing D.6.

8.6.5. DMI Implementation

The CDMIQWidget implementation is quite simple and mainly treats the update of the displayed widgets for inputs and outputs. Any attached observer is notified by its CEVCStateMachine object via the `Update()` method that the contents or visibility state of one or more CDMIInput or CDMIOutput object of the current subject CDMISubject has changed or that even the complete subject was changed. The latter is the case if the ETCS mode and/or application level was changed. A changed subject always means that the input and output widgets (`m_InputWidgets` and `m_OutputWidgets`) have to be rearranged. Since this includes the deletion of all existing input and output widgets and the creation of the new ones, this process should be avoided in cases, where only the value of CDMIOutput objects of the current CDMISubject were modified by the data flow calculation. Otherwise, this would delete any input of the driver that was already entered but not yet activated via the corresponding QPushButton. Accordingly, the CDMIQWidget observer always stores the pointer to the last used CDMISubject in the `m_pLastSubject` aggregation (see Figure 8.5). Only, if in the execution of `Update()` the last subject is not identical to the current subject, the input and output widgets are rearranged. Otherwise, only the (displayed) values of the `m_OutputWidgets` are updated from `m_Outputs` in the DMI subject.

Another but more minor aspect of the DMI implementation, is the fact that Qt 4 does not accept changes of GUI elements from other threads than the one running the Qt event loop [59]. The event loop is started from the `main()` function because the corresponding openETCS C++ generator (see Chapter 9) only generates the instantiation of domain framework types and the starting of the CEVCStateMachine object. The CDMIQWidget `Update()` method is called from the current data flow thread (see Subsection 8.6.2), which is not the main thread. Therefore, the `Update()` method cannot change GUI elements of CDMIQWidget directly. The solution is again to make use of Qt 4's signal and slot mechanism: The real update implementation is moved to the Qt slot `UpdatedSlot()`, which is connected [59] with the signal `Updated()`. The method `Update()` then only emits the signal `Updated()` to call the slot with the update implementation but then in the thread of the Qt event loop.

8.6.6. Error Handling

Since faults [78, pp. 12-14] not only can occur in concrete hardware components, which means in the PSM, but neither can be avoided completely in the implementation of the PIM, error handling is also an issue for the openETCS domain framework. An error is the incarnation of a fault as a deviation from the normal operation of the framework [78, pp. 12-14]. For fault detection or rather error propagation, exceptions [79] are used. This means if in a method of the domain framework a fault is detected that cannot be handled by the method itself, a corresponding exception is thrown. Mostly, exceptions may occur in data and control flows, which must be handled by the parent CEVCStateMachine object. The ETCS SRS already defines EVC Modes and procedures [88] in the case of an error that lead to a system failure [78, pp. 12-14]. Thus, the error handling is part of the concrete openETCS model while only the propagation is part of the openETCS domain framework. Furthermore, from the view of the domain framework, a system failure[11] may never occur because all errors respectively all thrown exceptions in the execution of CEVCState objects are handled by the parent CEVCStateMachine instance. Of course, from the view of the EVC, those errors lead to system failures because in the resulting failure Modes the EVC cannot perform its required function.

The most relevant origin of possible faults is the PSM or rather the HWSpecificImplementations sources because obviously not any possible specific implementation can be foreseen. On the one hand, faults may occur in the hardware itself. Those faults can be propagated to any connected proxy by a D-Bus signal while the resulting error can be then thrown by the related function block object. This kind of error propagation presumes a correct implementation of the HWSpecificImplementations source code. As already discussed in detail in Chapter 6, this presumption is not valid for the PSM and any supplier implementations in general, which was the reason for the separation of PIM and PSM in different execution environments. A grave issue for the execution of the openETCS domain framework data flow is the execution time of calls to platform specific adaptors since a constant sample time may not be exceeded (see Subsection 8.6.2). To avoid a dead-lock situation in the data flow thread caused by infinite response times of platform-specific implementations, all D-Bus calls are done with a certain time-out value. If this value is overrun, the waiting for the call is aborted and an exception is thrown.

There is a minor set of errors that cannot be handled by CEVCStateMachine class. Those may occur in methods that are called by an external actor, which mainly means the EVC. Nevertheless, those are limited to methods of CEVCStateMachine for starting and stopping the execution, and methods used during the instantiation of the domain framework classes by the generated code from the openETCS model. The latter ones must be avoided by a correct generator implementation and the checking of the static semantics.

In the domain framework, each method is declared with the exception types it can raise [79] to provide a transparent chain for error propagation. This is also done for methods that cannot throw any exception type at all. Additionally, for certain fault and error categories, different C++ types as classes are defined. Those are shown in Figure 8.23 as UML class diagram.

CException is the parent class for all concrete exceptions types in the domain framework.

[11] in the meaning of a unexpected and unwanted process termination

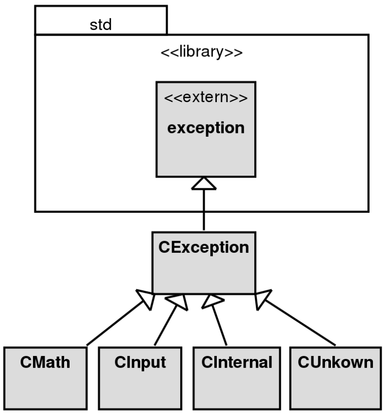


Figure 8.23.: UML class diagram of the openETCS domain framework exception types

It inherits from the `::std::exception` class to provide easier collection of error information, especially during the implementation and testing cycles, because their content is automatically printed (to the console) in cases of an uncaught exception. The concrete domain framework exception types are:

CMath Mathematical errors, e.g. a division by 0.

CInput Errors due to (false) input values in CFunctionBlock objects outside the definition space.

CInternal Internal error in a CFunctionBlock object typically related to an error in a hardware component.

CUnknown Errors for which no category can be determined. This exception should never occur, but also must be included to ensure all error cases are included.

All openETCS domain framework exception classes are located in the `::oECTS::DF::Error` C++ name space.

8.7. Verification

According to Req.5, the verification of the openETCS domain framework is an important issue because it covers most of the source code used for the EVC implementation. Not only to ensure the correctness of the implementation, but also to ensure the correct execution on a target platform, tests are used. Since the domain framework is implemented in an object-oriented manner, unit tests [6] are implemented. In general, unit testing focusses on the individual units of the source code to be tested. For object-oriented programming languages, like C++, those units are typically defined as the methods of classes, but this not explicitly required.

Since C++ does not provide an implicit unit testing mechanism, the tests must be implemented manually or with the support of a unit testing framework. CppUnit [16] is a platform independent framework for creating test suites based on unit tests, which perfectly matches the requirements of the openETCS domain framework. In contrast to complete self implementation of unit tests, CppUnit already provides mechanisms for the test development and the collection and output of test results.

8.7.1. Unit Testing Design

The CppUnit framework defines at its highest level so-called test suites, which are classes. Those test suites combine different (unit) test cases, which are methods of the test suite classes. In these test methods, assertions [79, 16] can be defined as Boolean expressions, which all must evaluate to true if the test method is executed successfully. The complete unit testing application or rather V&V test suite combines and executes all suites respectively all test methods. Since all classes of the domain framework (in name space `::oETCS::DF`) are tested, Figure 8.24 shows only a subset of the complete relations of the unit testing design.

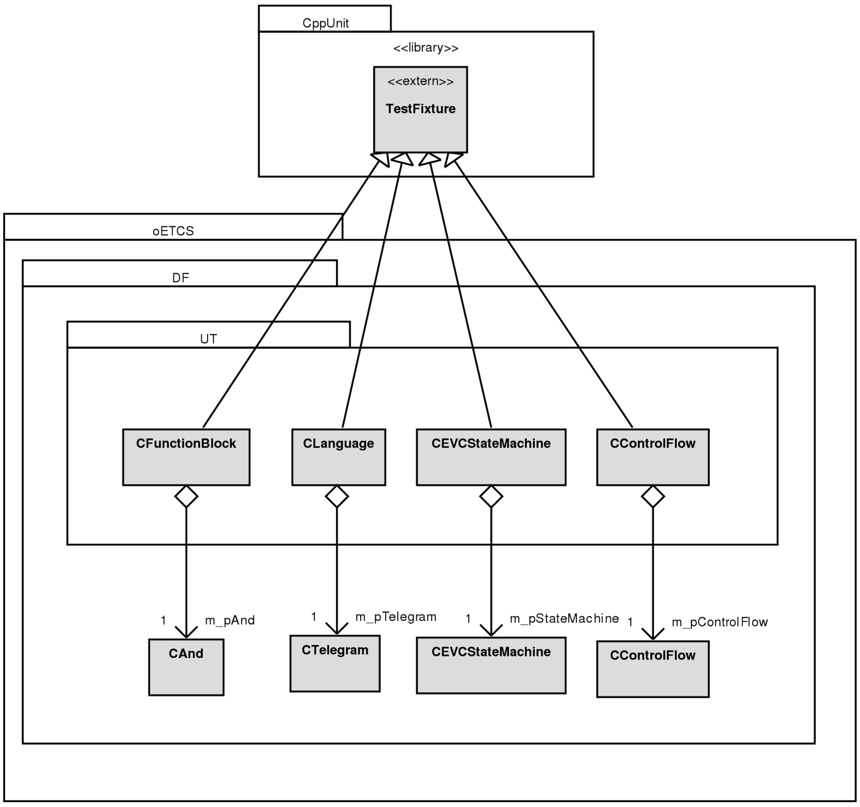


Figure 8.24.: UML class diagram of the basic openETCS domain framework unit testing elements

The classes in `::oETCS::DF::UT` are the test suites, which inherit from TextFixture [16], which is a class of the CppUnit framework. The classes that are tested are indicated by the aggregations `m_pAnd`, `m_pTelegram`, `m_pStateMachine`, and `m_pControlFlow`. Of course, all classes are tested, but, due to the size and complexity of the figure, only some examples are sketched.

The test suite classes must have access to all attributes, which include the private and protected ones, of the tested classes to provide white-box testing [78, p. 312] instead of pure black-box testing [78, p. 312], which only runs tests on the outputs for certain inputs. Since most of those methods are not public visible / accessible, a friend [79] relation between test suite classes and the classes to be tested have to be defined (omitted in the figure). Black-box testing might be sufficient for tests for the CFunctionBlock classes because those are defined as transfer functions for certain inputs, which return certain outputs. Consequently, their internal states are not of interest for the tests. However, this is not the case for all other classes and white-box tests are implemented via direct class attribute access.

8.7.2. Unit Testing Implementation

The implementation of tests for CFunctionBlock blocks is realised as random testing [78, p. 317]. For all inputs of a certain CFunctionBlock class, random input values are generated. Since the the transfer function for the `Execute()` method is always well-defined for each CFunctionBlock class, the resulting outputs can be checked for the expected values.

For classes outside the data flow, such interconnections between inputs and outputs is not defined. Therefore, those have to be tested in a simulative way [78, p. 318]. Especially, the classes for control flows (CControlFlow and CEVCStateMachine) can be additionally tested by state transition testing [78, p. 318]. For those cases, a transition system is instantiated by corresponding state and transition objects and then is tested if all states are reached and executed as expected.

A further, more detailed discussion of test implementation is not in the main focus of this work. The software reference of the tests and their implementation can be found in Appendix G.

8.8. Conclusion

The design, the implementation, and finally the testing of the openETCS domain framework show that all initial defined requirements are fulfilled. Certain, already theoretical introduced issues, like the integration of hardware virtualisation, are dealt with. Furthermore, the domain framework is aligned to the terminology of the OMG's definition of a model-driven architecture (MDA).

The special situations of the behavioural design is described because this is defined by the instantiation of domain framework classes. This can also be called “behaviour through structure”.

In contrast to all other instances of the openETCS architecture, the domain framework has to differentiate between faults, errors, and system failures and has to provide an error handling interface to the meta model respectively model.

9

openETCS Generator Application

This chapter describes the design and implementation of the openETCS code generator. Initially, general requirements for the application are defined. In general, the development description follows the same principles as in Chapter 8, which discussed the openETCS domain framework. Hence, the generator design is illustrated by the used design strategy, the structural design, and the deployment while UML is used as formalism. Also, the implementation is discussed by concrete source code examples. Additionally, the generation of configurations for virtual machines is introduced, which is followed by the description of the V&V strategy of the complete openETCS development process. Finally, the complete employed tool chain down to executable EVC binary is presented.

9.1. Requirements

In a DSL, the generators typically build the link between a model and the domain framework, which was introduced in Chapter 3. Therefore, the process of generating the code that is compiled and linked against the domain framework is also called model-to-text or model-to-code transformation. Since the code generators are not executed together with the target, but their execution result strongly influences the target execution, the requirements for the openETCS generator development are quite different from the ones for the domain framework in Section 8.1 and are defined as follows:

Req.9: full coverage / usage of relevant model elements
Req.10: generation of optimised code
Req.11: generation of a build configuration for compiling and linking the resulting executable binary
Req.12: generation of virtual machine configurations for the PIM and PSM

Req.9 To guarantee that the generated code from the openETCS model is a correct transformation, it must be ensured that all relevant elements in the model are used during the generation process. All relevant elements means any GOPPRR type that influences the dynamic semantics. Those are the most types besides the ones only used for documentation, like oNote.

Req.10 The generated code for the target binary should be optimised in respect to the execution time. Due to Req.5 of the openETCS domain framework, the code generator only creates instances of domain framework classes. Hence, the optimisation is reduced here to the order of the objects in aggregations, which is mostly relevant for data flow creation.

Req.11 Since the pure source code for the instantiation of the domain framework classes has to be compiled to be executable, an appropriate build configuration must be generated. This is especially needed to take certain compiler and linker flags [33] for including the openETCS domain framework and all other required libraries into account. According to Req.7, this build configuration should be also generated in a platform independent way.

Req.12 Since the deployment of the openETCS domain framework (see Section 8.5) defines different execution environments for the domain framework or rather the PIM and possible platform-specific adaptations (for hardware components) or rather the PSM, also the configuration for starting those environments should be generated. The used hypervisor should be available as FLOSS and chosen according to the considerations in Section 6.2.

9.2. Design Strategy

Similar to the openETCS domain framework in Section 8.2, all required classes of the generator must be identified by general use cases in an initial design step. For the use case definition, the tool chain (Figure 4.5 and Figure 5.3) presented already in Section 4.6 and Section 5.3 is also taken into account. Figure 9.1 defines the basic use cases as UML use case diagram. The included external actors and classes are explained in the following:

MERL_Generator Corresponds to the GOPPRR Generator of the tool chain in Figure 4.5 and Figure 5.3. It generates a GOPPRR XML file from a certain model in MetaEdit+.

XML GOPPRR XML file of the corresponding model.

DomParser XML parser class from the libxml++ library [54], which allows the conversion of an XML file to an internal Document Object Model (DOM) [4], which also corresponds to an abstract syntax tree. This can be easier processed than a plain text file.

CGOPPRRTransformer Transforms the DOM of the GOPPRR XML file into a GOPPRR C++ abstract syntax model.

CGOPPRRSyntaxTree Encapsulates a `::GOPPRR:CProject` (in Section 4.2) instance to provide a general interface to any kind of abstract model or tree for GOPPRR. Thus, further abstract syntax trees could be easily integrated. Currently, only the GOPPRR C++ abstract syntax model from Section 4.2 is supported.

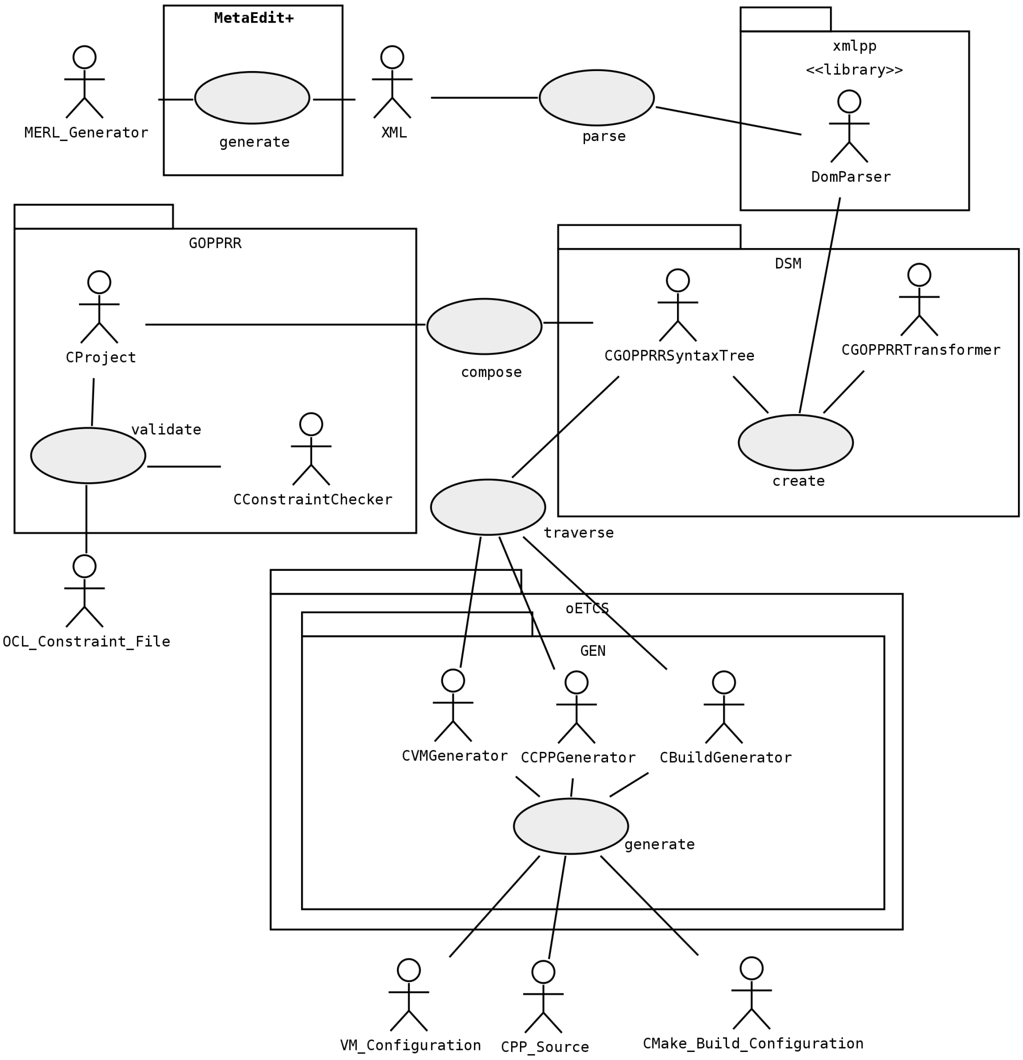


Figure 9.1.: UML use case diagram for the openETCS generator

CProject Root element class for any GOPPRR C++ abstract syntax model.

CConstraintChecker Checks a `::GOPPRR:CProject` instance for certain constraints defined in OCL as static semantics. Its implementation is, like the GOPPRR C++ abstract syntax model in Section 4.2, independent from the used meta model.

OCL_Constraint_File External file with constraints defined by OCL statements as static semantics for a certain meta model in the GOPPRR meta meta model.

CVMGenerator Generates the configuration for the virtual machines by traversing a CGOPPRRSyntaxTree object.

CCPPGenerator Generates the C++ sources to instantiate the openETCS domain framework by traversing a CGOPPRRSyntaxTree object.

CBuildGenerator Generates the build configuration (for cmake [14]) to compile the generated sources by traversing a CGOPPRRSyntaxTree object.

VM_Configuration Generated configuration for virtual machines of a certain hypervisor.

CPP_Source Generated C++ sources.

CMake_Build_Configuration Generated cmake build configuration.

Since the openETCS generator application uses and combines several modules or rather packages [64], the affiliation of the actors in Figure 9.1 is explicitly defined in the diagram. The packages themselves are explained in more detail in the next lines:

xmlpp External libxml++ library for parsing XML files, which is appropriate for the case study because it is licensed as FLOSS and is implemented in C++.

DSM Holds classes that are generally usable for domain-specific modelling and are not limited to a certain meta model. Since in this work only the GOPPRR meta meta model is used, this is mainly a design issue to provide a better expandability for further work with other meta meta models, like MOF (see Appendix A).

GOPPRR Package of the GOPPRR C++ abstract syntax model in Section 4.2, which is independent from the used meta model.

oETCS The superior package for all elements related to the openETCS case study.

GEN Package inside the oETCS package with all classes for the generator application that depend on the openETCS meta model (in Chapter 7).

Since the UML packages are mapped for the implementation to C++ name spaces, those can be directly found in the generator C++ implementation.

9.3. Structural Design

As for the openETCS domain framework in Section 8.3, the classes defined by the use cases in the prior section have to be transferred to a structural design. In contrast to the domain framework, most parts of the generator application are especially developed for the openETCS meta model, which mainly renders the re-usability[1], in most classes, impossible. Thus, the definition of a sophisticated object-oriented design pattern [32] is here not required and may even cause unnecessary workload. Figure 9.2 introduces the structural design for the generator application. The design details are discussed in groups of the surrounding packages in the next paragraphs.

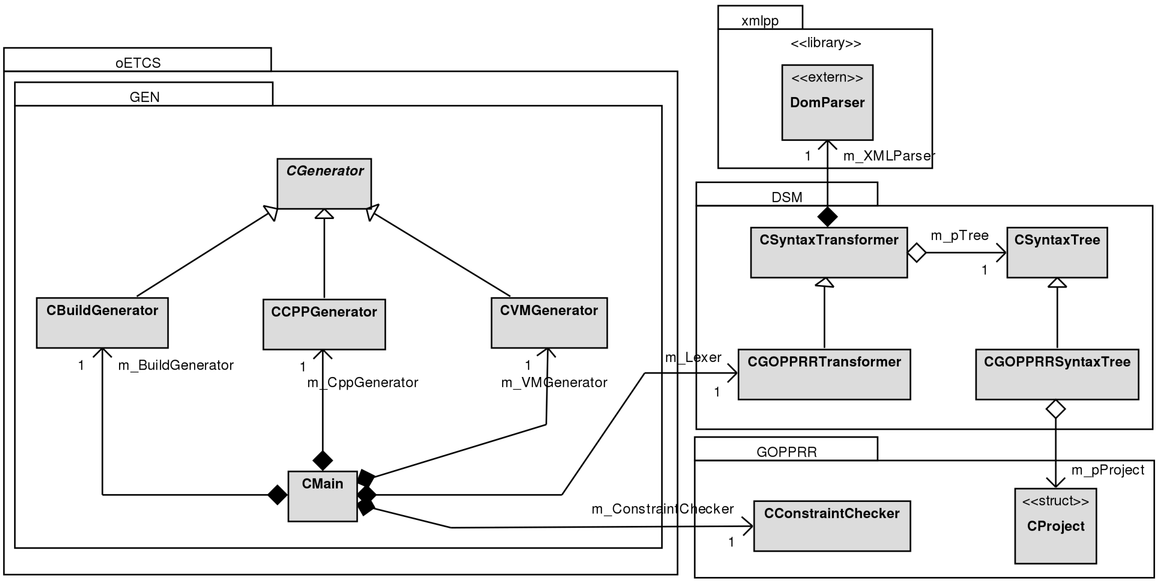


Figure 9.2.: UML class diagram openETCS generator application

GEN The openETCS generator classes do not provide a special software design pattern. Only, CGenerator is used as base class for all concrete generator classes to provide operations and attributes needed for each generator and to define abstract operations [64, 79], which have to be implemented. This could be interpreted as a composite design pattern [32, pp. 163-173], but since CMain only uses compositions of the concrete generator classes, it is not. CMain combines all generator classes and provides an operation, which can be called by the main-function [79] of the generator application. Furthermore, it has a composition `m_Lexer` to

[1] for other meta models

CGOPPRRTransformer to be able to provide a GOPPRR C++ abstract syntax model to the generator classes. The `m_ConstraintChecker` composition is used to check the GOPPRR C++ abstract syntax model validity according to the meta model constraints or rather static semantics.

DSM For general domain-specific modelling issues, this package provides a composite design pattern for transformer and abstract syntax model classes. Generally, transformer classes convert an intermediate model from an XML file to an abstract syntax tree. Therefore, for each one a base class (CSyntaxTransformer and CSyntaxTree) is defined, which corresponds to the component participant [32, p. 165] of the design pattern. CGOPPRRTransformer and CGOPPRRSyntaxTree are the concrete leafs [32, p. 165] for the GOPPRR meta meta model. As already explained above, the integration of this pattern is mainly relevant for possible extensions by further abstract syntax models in further work.

GOPPRR The detailed structure of the GOPPRR C++ abstract syntax model was already introduced and discussed in Section 4.2 and sketched in Figure 4.4. New in this package is the CConstraintChecker class for checking a certain CProject object for a set of constraints. Due to the definition of GOPPRR and its C++ abstract syntax model, also the constraint checker is completely independent from the meta model and can be easily implemented in one class.

xmlpp In the UML model, the xmlpp package only represents the external libxml++ library, which is used for the parsing of XML files.

9.4. Deployment Design

The differentiation by the three MDA model categories, as in Section 8.5, is not necessary for the generator application because it is not part of the model. Neither, the generator has to be executed in a different execution environment, nor an IPC has to be established.

How the different packages are combined or rather deployed together to build an executable binary is of special interest for the generator application development. Thus, the deployment of the GOPPRR and DSM package is introduced first and at last the GEN package, which combines all components.

GOPPRR Corresponding to its definition in Section 4.2, the GOPPRR C++ abstract syntax model is independent from the meta model and accordingly heavily reusable. Hence, its deployment as a library [77] seems to be the most appropriate. The deployment of the libGOPPRR is sketched in Figure 9.3. The artefacts hold the corresponding classes with the "C" prefix and all manifest the libGOPPRR component. The ExceptionTypes artefact holds the exception classes used for the abstraction methods for easier accessing be graph bindings in an abstract syntax model (see Section 4.2). The libGOPPRR component provides all classes to other components, which accordingly have to import or rather link against the libGOPPRR library. Figure 9.4 shows the corresponding component diagram.

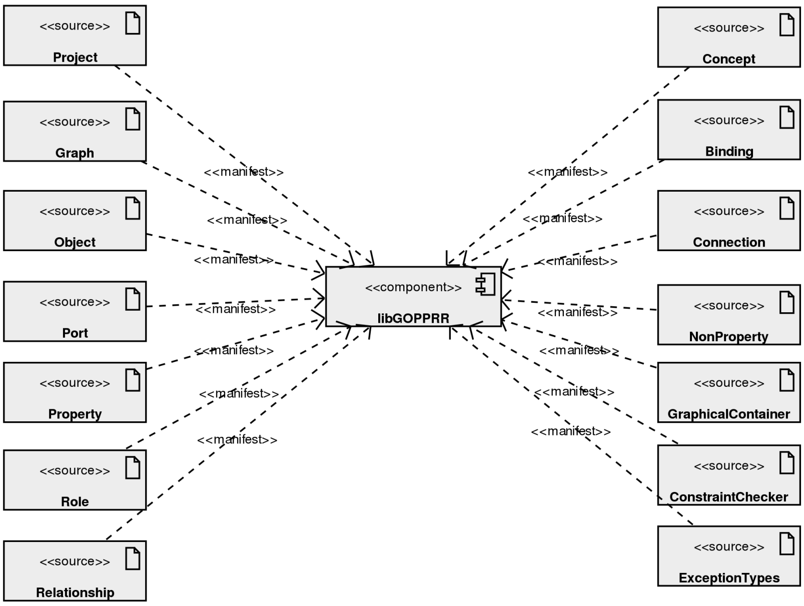


Figure 9.3.: UML deployment diagram for the GOPPRR library

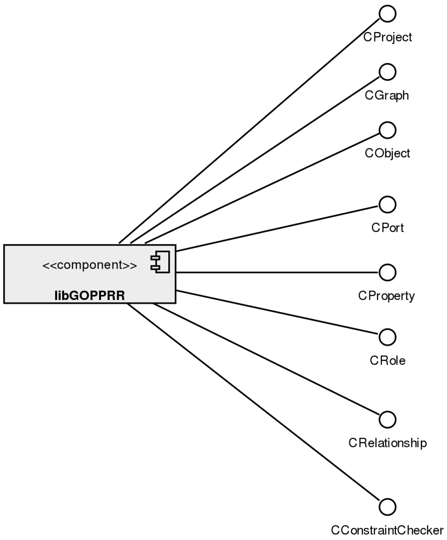


Figure 9.4.: UML component diagram for the GOPPRR library

DSM Similar to the GOPPRR package and the libGOPPRR library, the DSM package is also independent from the used meta model and can be deployed as library. The corresponding deployment diagram can be found in Figure 9.5. Again, the ExceptionTypes artefact holds all types used for raising exceptions. Since the libDSM builds the link between any meta model of GOPPRR and a concrete generator, it also requires and imports the classes provided by the libGOPPRR library. This is sketched in Figure 9.6. The libDSM library itself provides all classes for transforming a GOPPRR XML file, as intermediate model, into objects of the libGOPPRR library classes.

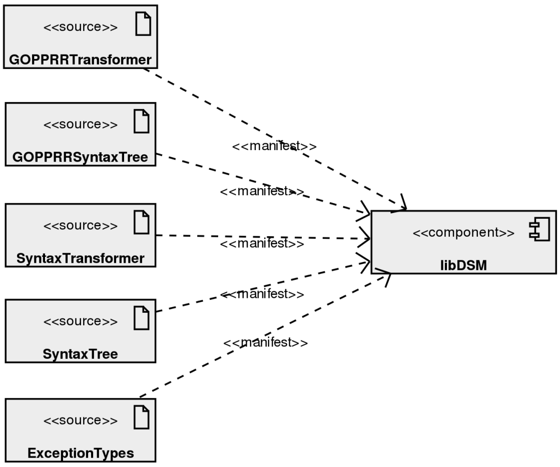


Figure 9.5.: UML deployment diagram for the DSM library

GEN As already described before, the GEN package implements the openETCS generator application as executable binary. Since it is deployed in only one component, the related diagram in Figure 9.7 is as simple as for libGOPPRR and libDSM. In contrast to those two other components, the openETCSGenerator does not provide any classes but requires those from the two libraries. The component diagram in Figure 9.8 shows corresponding import of classes.

9.5. Implementation

Due to the nature of the generator application, the interaction between the generator classes or rather objects is low. Mainly, a CMain object calls a CGOPPRRTransformer instance to convert a GOPPRR XML file into a CGOPPRRSyntaxTree to finally call all generator instances and the constraint checker. Thus, an own section for the behavioural design is spared for the openETCS generator application and the section directly proceeds with the implementation of the structural design.

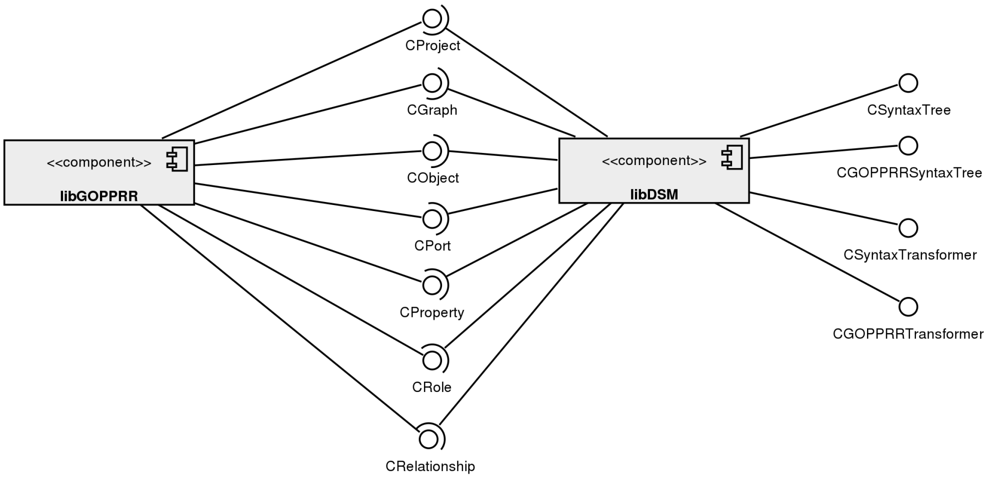


Figure 9.6.: UML component diagram for the DSM library

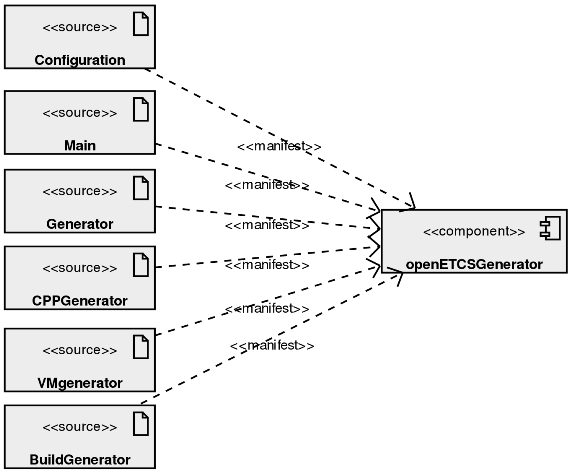


Figure 9.7.: UML deployment diagram for the openETCS generator application

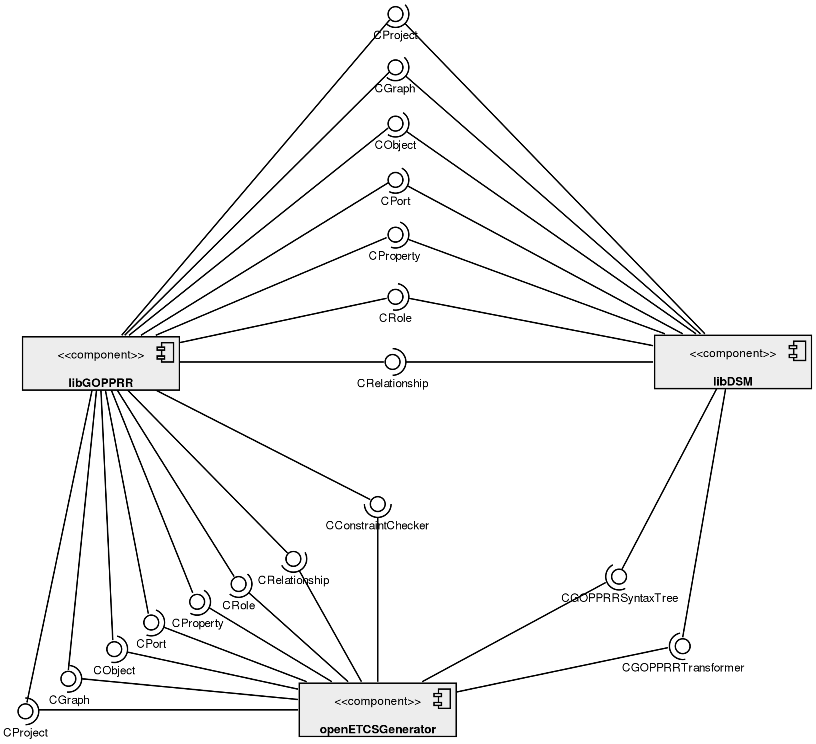


Figure 9.8.: UML component diagram for the openETCS generator application

The openETCS generator application is implemented correspondingly to the methodology of the domain framework implementation. Hence, the general implementation strategy in Section 8.6, the programming language, and target platform in Subsection 8.6.1 are the same and not explicitly discussed here.

9.5.1. libGOPPRR Implementation

Although the structural design of the GOPPRR C++ abstract syntax model was already introduced in Chapter 4, its implementation is discussed in this chapter because it is used by the generator application.

All elements for defining a GOPPRR C++ abstract syntax model are declared as structs, which can be observed in Figure 4.4. In C++, the only difference between classes and structs is that in a class all elements are by default private visible [79]. For structs, the default visibility is public [79]. Since for the GOPPRR C++ abstract syntax model mainly a tree of data structures has to be built, those are declared as structs to provide an easy and direct access to their attributes[2]. Hence, no methods need to be implemented for their access.

The only methods implemented in those structs are the ones used for abstraction, which were already mentioned in Section 4.2. Those are mainly located in the CGraph class[3]. Since they mainly focus on the abstraction of graph bindings (see Section 4.2), they are implemented by iterating over the local bindings `m_BindingSet` and the corresponding sub-structures. As already exemplary introduced in Section 4.2 for the `::GOPPRR::CGraph:Roles()` method, the abstraction methods mainly use a certain GOPPRR C++ abstract syntax element instance (e.g. of `::GOPPRR::CObject`) and the literal name of connected type that should be found.

Generally, such searches can provide more than one object[4]. Hence, the abstraction methods always return a map of pointers to the matching instances. To facilitate the usage in the generator classes, which typically iterate over the returned map or use just the first element, an optional exception was introduced to handle situations if the returned map is empty. The usage of exceptions can be defined in all abstraction methods by the `bUseException` parameter. This means the generator does not have to be implemented by an if-branch for each returned map to check it for non-emptiness. Otherwise, the access to an element of any empty map always causes a segmentation fault [77, pp. 353-453]. Since in the most cases the absence of an element type in a connection means a fault in the model, this is not manageable by the generator classes and the resulting error can be easily propagated by an exception.

Unfortunately, until the end of this work, no general OCL parser and checker was available as OSS or FLOSS library that could have been directly integrated in the CConstraintChecker class implementation. As a work-around for the case study, some exemplary constraints for the static semantics in Section 7.5 were directly implemented in C++. Of course, those are only meaningful for openETCS meta model instances. For other meta models, the static semantics would have to be implemented additionally in C++.

[2] aggregations, compositions, and properties from the design

[3] C++ structs will be also called classes in this document because of the only small difference in C++.

[4] GOPPRR instance

9.5.2. libDSM Implementation

Since the base class CSyntaxTransformer only provides general attributes and methods for all concrete transformer classes, the implementation of CGOPPRRTransformer is explained in detail.

The converting process starts with the parsing of a GOPPRR XML file by the `m_XMLParser` object, which delivers the content of the file as DOM. Furthermore, the XML file is validated according to the GOPPRR XSD from Section 4.3 to ensure that only valid GOPPRR instances are used. Since the DOM is mainly a representation of the XML file in a certain C++ class or rather object structure, the whole document object model has to be processed by the CGOPPRRTransformer.

Due to the structure of the GOPPRR C++ abstract syntax model and the XSD, concrete instances of GOPPRR elements are only used on the project level while on lower levels for graphs, objects, ports, properties, roles, and relationships other elements are only referenced. For the implementation in C++, those are class instances and pointers [79] to them. In the XSD, a special XML element is defined for references to elements, which uses the OID (see Section 4.2) instead of memory addresses [79] for pointers.

A certain issue is the fact that graphs structures can be defined in a quasi unlimited way by decompositions and explosions. Therefore, the first approach was also a recursive implementation of the CGOPPRRTransformer. This means if a reference to any element type was encountered in the DOM, this reference was followed (by the OID) until a concrete instance was met. The big advantage of this approach is that it is ensured that all references are resolved for the GOPPRR C++ abstract syntax model. Unfortunately, this also caused severe performance problems. Mainly, not by the recursion itself but by the usage of methods in the libxml++ library to resolve XPath [42] expressions to find directly the referenced element.

To handle this problem, a certain condition of the GOPPRR C++ abstract syntax model and XSD was used to implement the conversion in an iterative way: Since every GOPPRR element instance must exists in the parent project, all concrete instances can be created in the project by a single iterative step through the complete DOM while all references are left unset. In only one further iterative step, all references can be then easy initialised because all instances are guaranteed to exist, which is enforced by the GOPPRR XSD. This implementation approach provides a much better performance than the recursive one and is finally used in the CGOPPRRTransformer class.

Additionally, it implicitly solves a problem with recursive sub-graph relations if a GOPPRR object has a decomposition to a GOPPRR graph in which this object is again present. Proceeding such a relation in a recursive way would directly lead to an endless execution without termination. Of course, such recursive sub-graph relations are explicitly forbidden for a openETCS model by the static semantics in Subsection 7.5.2, but it must be emphasised that according to the tool chain and the integration of the OCL in Chapter 4, the corresponding constraints are checked on the GOPPRR C++ abstract syntax model. Thus, the GOPPRR XML file can always hold such an invalid relation. Nevertheless, the recursive implementation is still available for references purposes. The issue to insure that the whole intermediate XML model is used is also related to Req.9.

The CGOPPRRSyntaxTree mainly encapsulates a CProject instance and does not have to be discussed in certain detail.

9.5.3. openETCS Generator Implementation

Obviously, CCPPGenerator is the most complex generator class for creating the source code, which instantiates the classes of the openETCS domain framework. Since nearly all objects in the openETCS C++ abstract syntax model are relevant for the domain framework instantiation, the complete C++ model must be traversed. Furthermore, the order of the class instantiations is not arbitrary because some classes need other classes directly during their construction [79]. For example, each CEVCState object needs its parent CEVCStateMachine instance to register itself as new state. Also, for other classes, it is necessary to primary create a vector of pointers to other already existing class instances, which are then used during the object construction. This is the case, for example, for CDataFlow objects, which need a vector of CFunctionBlock pointers, which defines the objects used in the data flow. Therefore, several helper methods were implemented to separately handle the generation of different model parts and to provide a better structured implementation. For clarification, the whole generation process is shown as UML interaction diagram from the view of the CMain class.

In the first step, the GOPPRR C++ abstract syntax model is created by `CreateSyntaxFromFile()` method of the transformation class. This creates the CGOPPRRSyntaxTree object `m_pTree`, which additionally creates the underlying `m_pProject` with the GOPPRR C++ abstract syntax model.

Directly afterwards, the underlying CProject instance is returned by the `GetProject()` method and checked for validity according to the static semantics because the generation process should only be executed for valid models.

CCPPGenerator Implementation The generation of the C++ source code for the openETCS domain framework instantiation is started by the `Generate()` method of the `m_CppGenerator` object. Initially, the `GenerateRootGraph()` method generates all sources for the instantiation of the parent CEVCStateMachine object after the source file's initial part[5] was added by the `GenerateHeader()` method. This is followed by the generation of all other objects by the related helper methods. All instantiations are generated surrounded by a `main()` function block [79] in order that the source code file can be directly compiled as executable binary. Since only valid models are used for generation, the CCPPGenerator methods do not need a special error handling for faults related to the model. Only general faults, as segmentations, should be handled or, even better, avoided. Thus, most of the implementation of the CCPPGenerator methods is mostly the iteration over GOPPRR C++ abstract syntax model objects and does not need to be discussed here in more detail or by examples.

More complicate is the generation of the execution order of function block objects, which importance was derived and explained in Section 7.6 and Section 7.7. Simplified expressed, function block objects should be executed in the order of the direction of the data flow. In cases of open loops, this means starting from the objects that only have outputs. For closed loops, the start and end object cannot be defined in a general manner if they do not have any element

[5] with comments about creation time, model name, etc.

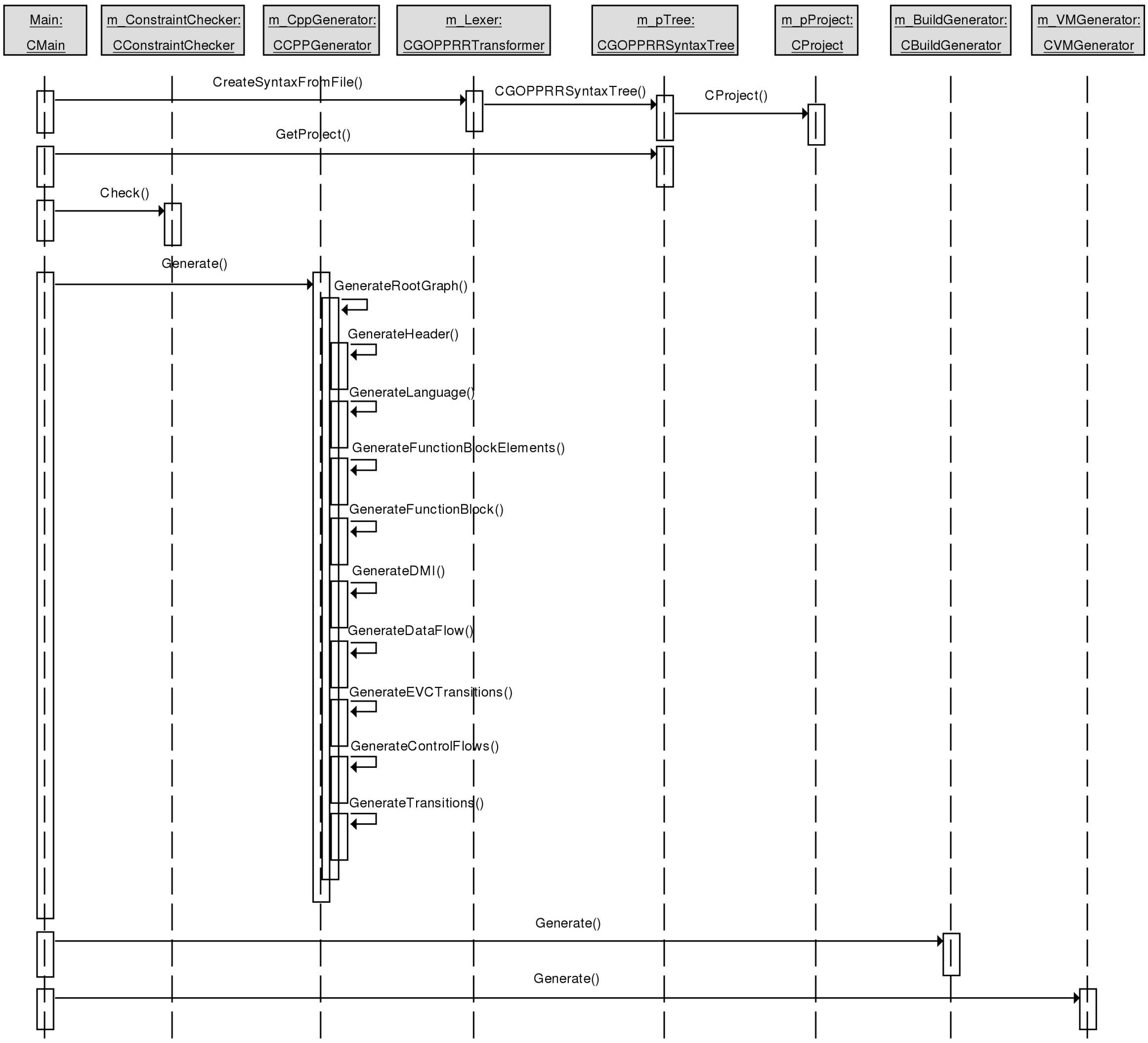


Figure 9.9.: UML interaction diagram of the generation process

outside the closed loop. Hence, the start object can be arbitrarily chosen. Because data flows are modelled as relationships or rather bindings in GOPPRR, the needed processing of objects is quite more complicated than for all other cases in the openETCS meta model. Therefore, a specialised abstract model of data flows is used to determine the execution order in each data flow of the model. The class diagram of this abstract model is shown in Figure 9.10. Due to

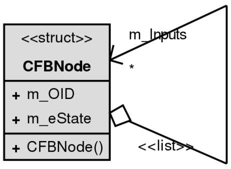


Figure 9.10.: UML class diagram of the abstract model for data flow execution order generation

its specialisation for the data flow order, the abstract model is quite simple and only consists of the struct CFBNode. It holds the attributes `m_OID` which is the OID of the corresponding function block object and `m_eState`, which is an enumeration type used during the creation of the abstract model. The `m_Inputs` aggregation represents the connection from outputs of other function block objects, which is the backward direction of the data flow. The other or rather forward data flow direction, the connection of inputs to outputs, is not relevant for the data flow abstract model because only function block objects without inputs shall be found.

The abstract data flow model is created by the `CCPPGenerator::BuildAbstractModel()` method, which is located in Listing E.1 in Appendix E.

After the execution of this method, the abstract model consists of CFBNode objects, which have the `m_eState` `DEFINED`, which means those do not have to be modified further and are final in the data flow abstract model. In contrast, all objects in each `m_Inputs` are created as `UNDEFINED`. They can be interpreted as place holders and must afterwards be replaced by a reference to the "real" (`DEFINED`) object with the same OID, which can be easily done by the code in Listing E.2.

The execution and also generation order is finally computed by the `CCPPGenerator::-ProcessAbstractModel()` method, which simply generates a list of function block object OIDs in the required execution order in Listing E.3.

Although the openETCS domain framework provides the parallelisation of independent data flows (see Section 8.3), this is currently not taken into account for the C++ code generator because the performance by a plain, serial data flow execution is sufficient for this case study.

CBuildGenerator Implementation In the next step, the build configuration for cmake is generated by the `Generate()` method of the `m_BuildGenerator` object. This is completely independent from the GOPPRR C++ abstract syntax model because it only requires information about the source code file, include directories [79], and libraries for linking.

CVMGenerator Implementation As final generation step, the configuration for the virtual machine(s) is created. This uses only few objects of the model since it only needs information about the reimplementation of certain hardware devices, as COdometer or CServiceBrake, in a PSM. The details of the virtual machine integration and the used hypervisor are discussed in Section 9.6.

9.6. Virtual Machine Usage and Integration

For the openETCS case study, only a simple configuration for the Xen [97] hypervisor is generated. Xen is part of the Linux kernel and accordingly also FLOSS. Furthermore, Xen uses a micro architecture [97], which corresponds to the discussion in Section 6.2 and accordingly fulfils Req.12. On the other hand, the Xen hypervisor is only available on Linux-based operating systems, which opposes Req.7 for the openETCS domain framework. However, the generation of configurations for Xen is used in this case study to exemplary demonstrate the integration of the possible hypervisor usage. Any other hypervisor could be integrated analogously by extending the "VM Generator" or rather the class CVMGenerator from Section 9.3.

Generally, a Xen configuration file is generated for the VM[6] as para-virtualisation [92] for the PIM in Figure 8.16. Additionally to this configuration, at least a root file system is needed, which holds the operating system to be executed in the VM and the EVC binary. Since the EVC binary is compiled and linked from generated code from the CCPPGenerator class, it is not meaningful to provide such a root file system with the software and models without the EVC binary. Furthermore, operating systems executed under a Linux kernel can be created and modified in unlimited ways. Hence, the root file system is not generated by the openETCS generator but can easily be created by established tools like xen-create-image [18].

A second configuration file for a VM for the PSM is created if one ore more objects for hardware interfaces (from Chapter 7) in the openETCS model are defined as external by the Boolean property "IsExternal". Analogue to the VM for the PIM, neither this root file system can be created in advance because the platform specific extensions have to be added and cannot be known before.

9.7. Verification and Validation

The verification of the generator must be divided in several parts because it combines functionality from other classes and name spaces establishing the generation process. Therefore, it is not meaningful to test all those classes together. Since the classes of the `::GOPPRR` name space do not provide much implementation to test, the initial focus lays on the `::DSM` name space, which converts the intermediate GOPPRR XML model into a CProject object.

It would be possible to implement functional tests [78, p. 316] with small, human-interpretable XML models and then check the created CProject instance. Unfortunately, the complete transformation process from the GOPRR model instance in the MetaEdit+ application to the CProject object includes also the generation of the XML file by MetaEdit+ (described

[6] called domain in the terminology of Xen [97]

in Section 4.4). This means even if functional tests show the correct implementation of CGOPPRRTransformer, it is not guaranteed that the XML file is generated correctly by MetaEdit+. Additionally, the implementation of tests for the MERL GOPPRR XML generator is not easy to realise because MERL scripts only may run in the context of certain GOPRR instances. The best possible solution to avoid this problem is to generate the tests directly from the MetaEdit+ model as C++ source code.

Similar to the MERL generator for creating the GOPPRR XML file, this can be done in a general manner not specialised for a certain meta model. Only, all graphs have to be processed to generate for each GOPPRR element test statements to check their existence in a CProject object or rather abstract syntax model. These statements look like the following example, which is only a small excerpt:

```
::GOPPRR::CGraph*                pGraph(0);
::GOPPRR::CObject*               pObject(0);
::GOPPRR::CProperty*             pProperty(0);


// BEGIN: asserts for graph openETCS Case Study with oid 3_256
CPPUNIT_ASSERT(pProject->m_GraphSet.find("3_256") != pProject->m_GraphSet.end());
pGraph = &(pProject->m_GraphSet["3_256"]);
// BEGIN: asserts for object Full Supervision of type Mode with oid 3_283
CPPUNIT_ASSERT(pGraph->m_ObjectSet.find("3_283") != pGraph->m_ObjectSet.end());
pObject = pGraph->m_ObjectSet["3_283"];
CPPUNIT_ASSERT(pObject->m_ID == ::std::string("Full␣Supervision"));
CPPUNIT_ASSERT(pObject->m_Type == ::std::string("Mode"));
CPPUNIT_ASSERT(pObject->m_OID == ::std::string("3_283"));
CPPUNIT_ASSERT(pObject->m_Properties.find("3_285") != pObject->m_Properties.end());
pProperty = pObject->m_Properties["3_285"];
CPPUNIT_ASSERT(pProperty->m_Value == ::std::string("Full␣Supervision"));
CPPUNIT_ASSERT(pProperty->m_Type == ::std::string("ModeName"));
CPPUNIT_ASSERT(pProperty->m_OID == ::std::string("3_285"));
[...]
```

The `CPPUNIT_ASSERT()` [16] macro is part of the CppUnit framework and defines assertions [79], which must be fulfilled for a certain test case. Initially, the existence of the graph instance is checked. Afterwards, objects and their properties are checked and so on. A great advantage is that the OIDs are used as key values for the maps in the GOPPRR C++ abstract syntax model. Thus, the access to the elements is easy definable in the MERL generator. Since a lot of assertions for each graph are generated in this manner, those are grouped in C++ functions for each graph. Finally, a function is generated that calls all those graph functions and can be integrated in any testing application.

Because the test can always be generated along with the GOPPRR XML intermediate model, it can be guaranteed that all GOPRR or rather GOPPRR elements of the model instance in MetaEdit+ also exist in the GOPPRR C++ abstract syntax model or rather in the CProject object. This covers Req.9 for the transformation process and is not limited to the openETCS meta model.

As already explained for the GOPPRR XML generator in Section 4.4, a small drawback is that no assertions can be generated, due to MetaEdit+ / MERL limitations, for properties that are non-properties. For those, always a specialised generator for the concrete meta model has to be added. The full code of the assertion generators can be found in Section F.2.

The realisation of tests to ensure Req.9 for the C++ source code generation by CCPPGenerator is much more complicated. It can also be interpreted as a transformation but not in such a general way as for the internal MetaEdit+ GOPRR representation to the GOPPRR C++ abstract syntax model. Therefore, again generating tests directly from MetaEdit+ is not an option because much more complex and specialised generators in MERL would have to be implemented, which would raise problems already discussed in Chapter 4. On the other hand, it is already assured by the generated tests for the transformation to a CProject object that the GOPPRR C++ abstract syntax model is correct and fully covered / used. Hence, it is sufficient to test if the transformation from CProject object to C++ source code is correct or rather covers the whole model.

The executable EVC binary is the end product of the generation process and consists of linked object or rather binary code. This is very difficult to analyse by parsing since object code is always platform specific. Another possibility would be the generation of test methods / functions that are linked with object code of the EVC binary. Since it is quite complicated to identify the C++ object structure by directly analysing the application memory or rather the heap [77, pp. 353-453], the openETCS domain framework objects would have to be accessed for testing. Unfortunately, according to Section 8.3, class instances of the domain framework cannot be easily compared with instances of GOPPRR C++ abstract syntax model because not all model elements are directly transferred to domain framework instances and not all element properties, including the OID, are used. Therefore, a simple identification of elements by their OID is not available. Furthermore, the CCPPGenerator class generates directly the domain framework instantiation in a main-function, which means that for testing the target code would have to be modified. This might influence the validity of the testing results for the real EVC binary.

Alternatively, instead of examining the object code, the generated C++ source code could be analysed. Since CCPPGenerator uses the OIDs for generation of object instance names / variable declaration, their GOPPRR equivalence can be found much easier by parsing the generated source code file. It is also questionable if it is necessary to again run tests on openETCS domain framework objects since those are already tested independently from the generators (see Section 8.7). To only test the CCPPGenerator class, the analysis of the generated C++ source code file should be sufficient. This means concretely that the generated C++ source code is checked for the existence of certain C++ statements that instantiate the corresponding domain framework classes. In contrast to the verification of the transformation from a GOPRR model instance to GOPPRR C++ abstract syntax model, not all elements can be tested for existence because the openETCS C++ generator does not use all GOPPRR elements to create a certain openETCS domain framework class instances. For example, the roles “DataOutput” and “DataInput” in a data flow are only used to connect a certain function block object output with a certain input, but their OID is never used. On the other hand, the “DataFlow” relationship is directly used to create CFlow or CBitFlow instances, and accordingly all “DataFlow” relationships in the GOPPRR model must exist in the generated C++ code as object. Thus, currently only the existence of instantiation statements of all function blocks, EVC Mode, and language objects and the objects for the “DataFlow” relationship is tested.

9.8. openETCS Tool Chain for Dependable Open Model Software

The resulting concrete openETCS tool chain for the complete development process is sketched in Figure 9.11. The verification of the transformation from the GOPRR to the GOPPRR instance is done, corresponding to the description in Section 9.7, by the generated "Functional Test Cases", which are run on the "GOPPRR Abstract Syntax Tree". The verification of the transformation of the concrete GOPPRR C++ abstract syntax model to the concrete source code for the EVC binary is implemented in the "Functional Testing", which consumes the generated "Source Code" and "GOPPRR Abstract Syntax Tree" artefact. The validation of the generation process refers here to a valid model, which is checked by the constraints of the static semantics. Also, the final build process of the EVC binary and its execution under a Xen hypervisor were added.

Since the extensions or transformations for the PSM cannot be generally foreseen and neither is automatically generated from other artefacts, it is omitted in Figure 9.11.

9.9. Conclusion

The openETCS generator application fulfils all initially defined requirements by combining several C++ classes producing the different generator output.

To ensure that Req.9 is fulfilled, tests can be generated automatically for both models: First, in a general way, independent from the meta model, for the transformation from the GOPRR meta model instance in MetaEdit+ to the GOPPRR C++ abstract syntax model. Second, for the concrete model-to-text transformation to openETCS domain framework objects.

The usage of cmake provides implicitly a platform independent build configuration for compiling and linking the EVC executable binary, which facilitated the implementation of the CBuildGenerator and fulfils Req.11.

CVMGenerator provides an implementation for generating configurations for the Xen hypervisor for para-virtualisations, which fulfils Req.12.

The separation of independent data flows for parallelisation could be future development work for the openETCS generator application because it is already prepared in the domain framework.

Although the checking of the static semantics was only realised by the direct implementation of exemplary constraints in C++, all OCL constraint statements in Section 7.5 could be used in combination with the UML-based Specification Environment (USE) [34] application, which is available as FLOSS, to check the static semantics outside the openETCS tool chain in Figure 9.11. Accordingly, the openETCS tool chain and USE would have to be adapted to allow the export and import of the concrete GOPPRR C++ abstract syntax model instantiation. Hence, an appropriate XMI export must be realised inside the GOPPRR package / name space and the corresponding XMI import mechanism in USE, which neither is currently available.

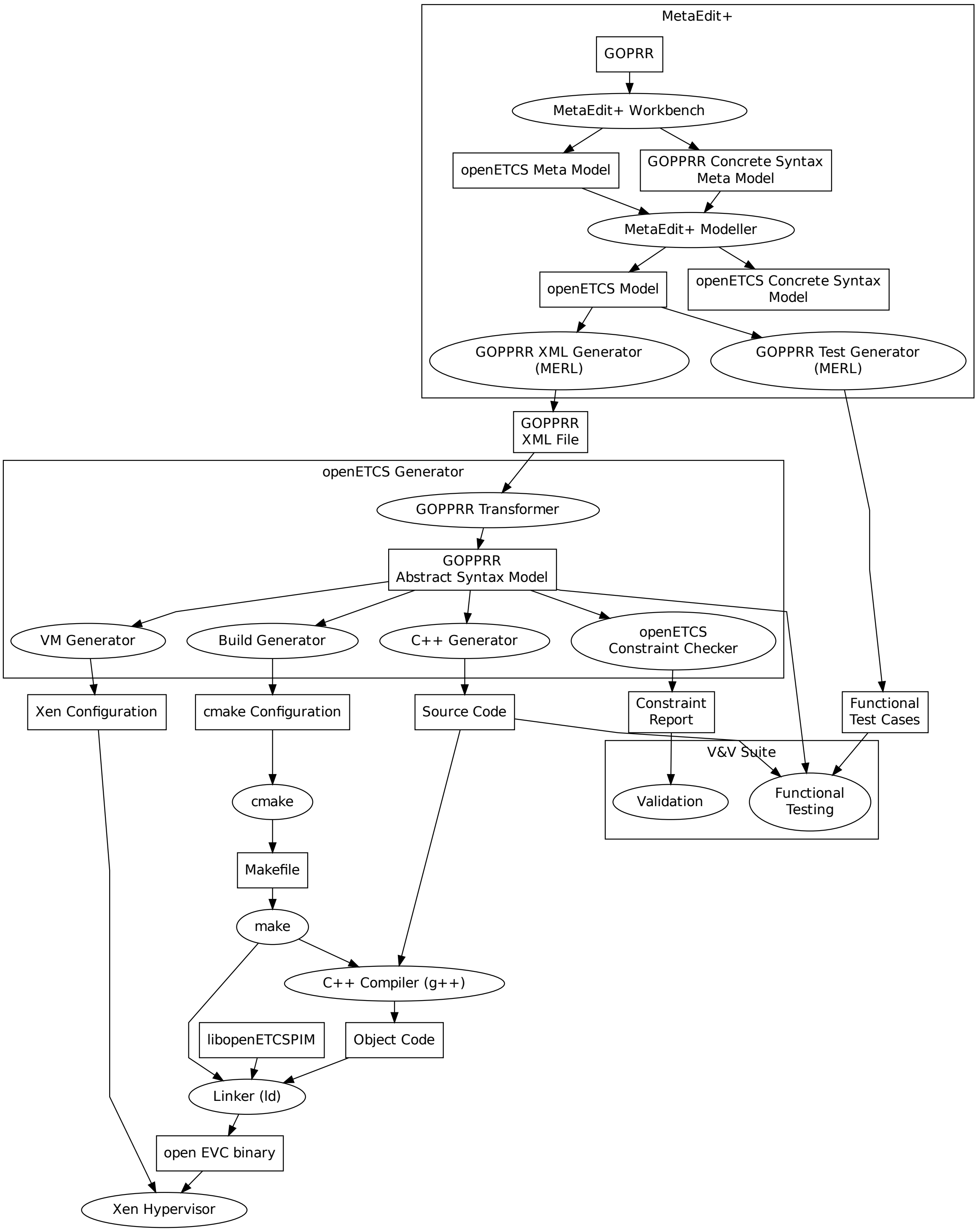


Figure 9.11.: Complete openETCS tool chain for dependable open model software

10
openETCS Model

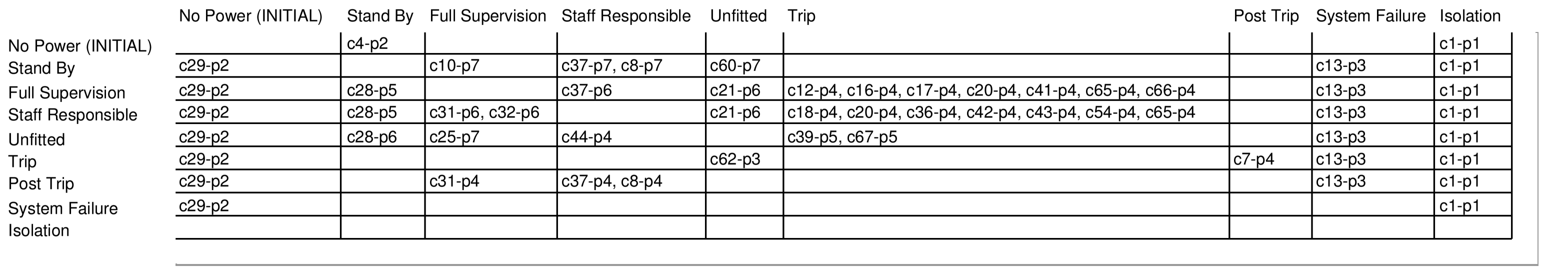
The openETCS model or rather, according to Figure 7.1, the openETCS formal specification is the CIM of the case study. As already discussed in Section 7.1, the modelling of the complete ETCS specification would exceed the limits of this work and this case study by far. Therefore, only a sub-subset of the ETCS SRS is modelled here. Nevertheless, the limitation to certain ETCS Modes and Application Levels emerged to be not sufficient to gain an openETCS model that is small enough to be manageable in this dissertation. Hence, not all functionality specified in the SRS for the used ETCS Modes is modelled. The main goal of the openETCS model is to give an example that is executable in a simple but adequate simulation environment. This simulation is described in Chapter 11. The reduction of used functionality is mainly related to the modelling of data flows.

It should be noted that parts of the SRS that are not modelled for the case study are not always emphasised in this document since ETCS is primarily used as an example of a train control system. Another certain remark is that not all parts of the model are presented in this chapter because those are mostly graphs, which all together would require extensive space. The complete model can be found in Appendix C.

This chapter presents exemplary diagrams of the openETCS model in the order of the top-down structure of the meta model graph types in Figure 7.2. The description starts with the so-called root graph for the ETCS Modes and transitions, which is followed by data and control flows for each combination of ETCS Mode and Application Level. Afterwards, the extraction of data from incoming balise telegrams is described and the models of the used ETCS language elements are presented. Finally, possible concepts of extending models in respect to safety properties are discussed.

10.1. ETCS Mode and Transition Matrix

Although the GOPRR or GOPPRR meta meta model does not define or require a certain root graph, the gEVCStateMachine graph type can be interpreted as such because it must exist exactly once in an openETCS meta model (see Subsection 7.5.1) and all other graph instances are connected with it via decompositions and explosions (see Section 7.2). The modelled transition matrix of the used ETCS Modes (compare Section 7.1) is shown in Figure 10.1.



	No Power (INITIAL)	Stand By	Full Supervision	Staff Responsible	Unfitted	Trip	Post Trip	System Failure	Isolation
No Power (INITIAL)		c4-p2							c1-p1
Stand By	c29-p2		c10-p7	c37-p7, c8-p7	c60-p7			c13-p3	c1-p1
Full Supervision	c29-p2	c28-p5		c37-p6	c21-p6	c12-p4, c16-p4, c17-p4, c20-p4, c41-p4, c65-p4, c66-p4		c13-p3	c1-p1
Staff Responsible	c29-p2	c28-p5	c31-p6, c32-p6		c21-p6	c18-p4, c20-p4, c36-p4, c42-p4, c43-p4, c54-p4, c65-p4		c13-p3	c1-p1
Unfitted	c29-p2	c28-p6	c25-p7	c44-p4		c39-p5, c67-p5		c13-p3	c1-p1
Trip	c29-p2				c62-p3		c7-p4	c13-p3	c1-p1
Post Trip	c29-p2		c31-p4	c37-p4, c8-p4				c13-p3	c1-p1
System Failure	c29-p2								c1-p1
Isolation									

Figure 10.1.: Matrix of the openETCS gEVCStateMachine root graph

The transition matrix holds all transitions that are defined in the ETCS SRS for the subset of modes and can be therefore easily compared with the transition table [88, p. 40] in the specification document. It must be noted that only those conditions are part of the modelled transition matrix that are applicable in ETCS Application Levels 0 and 1 because the selected subset for the case study in Section 7.1 precludes Application Levels 2 and 3.

10.2. Data and Control Flows in ETCS Modes

In contrast to the gEVCStateMachine root graph, a gMainFunctionBlock is modelled for each ETCS Application Level in each Mode. Thus, this section is divided in subsections for each ETCS Mode and, where required, in sub-subsections for different Application Levels.

10.2.1. No Power Mode

Although the No Power Mode is not directly related to a certain Application Level, it is relevant for the model that No Power is modelled for all possible Application Levels. Application Level 0 is defined as default for No Power because it is the initial mode for a newly started EVC binary. Anyway, it is possible that No Power is entered during the runtime from another Mode in Application Level 1. For example, this is the case for the transition Full Supervision $\xrightarrow{\text{c29−p2}}$ No Power.

Application Level 0

The main functionality in No Power is, according to [88, p. 10], the enforcing of the train stop via the emergency brake system. Furthermore, the driver should be able to power up the EVC (via the DMI), which should activate the transition “c4-p2” to the Stand By Mode. The corresponding modelled data flow is sketched in Figure 10.2 as gMainFunctionBlock graph. Of course, in a “real world” implementation of ETCS, powering the system refers to physically powering, but since this case study is realised as pure software, this functionality bases on a simple DMI input object.

The DMI is modelled by a separated oSubFunction object “DMI No Power” to be reusable in other Application Levels. The corresponding model is shown in Figure 10.3. The DMI displays the current ETCS Mode and Application Level to the driver. Additionally, the driver is able to power up the system via the “System On Power” oDMIInput object.

Application Level 1

The only difference between Application Level 0 and 1 in the No Power mode is the assignment of the “Stored ETCS Level” oVariableStorage object, which is therefore omitted. The DMI is modelled by the same oSubFunction object as for Application Level 0 and was already presented in Figure 10.3.

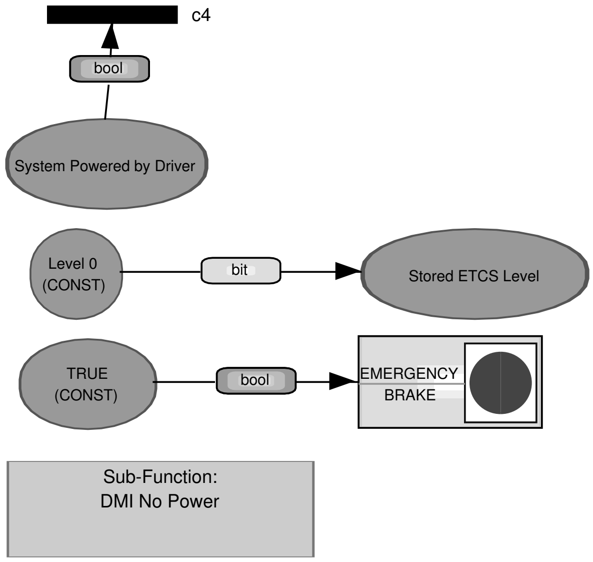


Figure 10.2.: No Power Mode in Application Level 0

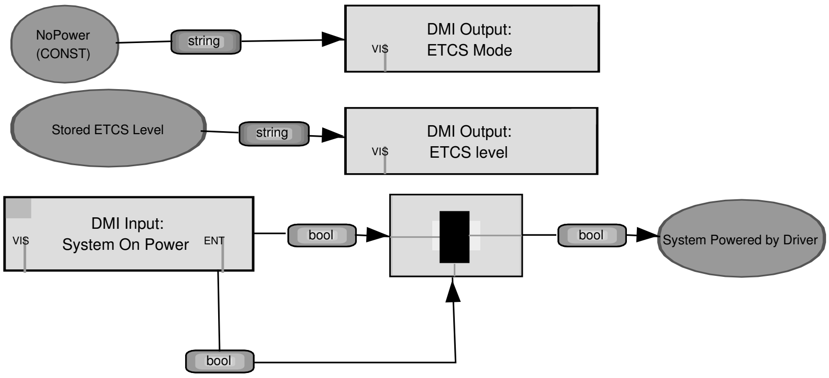


Figure 10.3.: “DMI No Power” gSubFunctionBlock graph

10.2.2. Stand By Mode

Stand By cannot be entered directly by the driver but only by a transition from No Power. The provided functionality is the enforcement of the stand-still of the train and the data collection for the train mission [90, pp. 8-17].

Application Level 0

The data collected from the driver for the train mission is used to determine the next ETCS Mode and, if necessary, an Application Level switch. Figure 10.4 represents the model of the data flow in Application Level 0. According to the transition matrix in Figure 10.1, all guard objects are used besides "c13-p3", which is executed in the case of an error, because the error oModeGuard object is a property of the graph itself.

The primary functionality is modelled in three oSubFunction objects: "Start of Mission in Stand By", "Stand Still Supervision", and "DMI Stand By".

The "Start of Mission in Stand By" function collects, according to [90, pp. 8-17], data from the driver. In the model, the sub-function only holds an oEmbeddedStateMachine object with a Boolean true input to start it. Due to its simplicity, it is omitted in the documentation, and only the related gEmbeddedStateMachine graph is shown in Figure 10.5. States that are prefixed with a "D" or "S" and are followed by a number are directly transferred from the so-called state diagram in [90, p. 18]. Since the state diagram syntax in the SRS is a mixture of states, activities, executions, and decisions, not all of them can be mapped directly to the openETCS model.

It should be remarked that the used state diagram formalism in the SRS was not integrated in the openETCS meta model because first it is a mixture of already existing diagram types, like state machines / diagrams [80] and (UML) activity diagrams [64]. Second, the used formalism for control structure must be combinable with data flows, which was discussed and explained in Section 7.2. This is not the case for the state diagram syntax used in the SRS.

State "S1: Request Driver ID" is the initial state, which is executed until the driver has entered his or her ID in the DMI.

"D1: Check ETCS Level" verifies if the current ETCS Application Level is valid or not.

State "S2: Request ETCS Level" waits for the input of the current Application Level via the DMI if the Application Level is invalid.

In "Check Train Position", the driver can revalidate the current train position or enter a new one.

"S10: Waiting for Driver Selection" offers the driver via the DMI the selection of overriding an existing EoA[1] or the proceeding one with an active mission or the entering of train data.

"Override EoA" is a final state, in which the transition to the Staff Responsible Mode is activated via the "c37" condition with the usage of the "Staff Responsible via Override" oVariableStorage object. Since the Staff Responsible Mode is only available in ETCS

[1] End of Authority

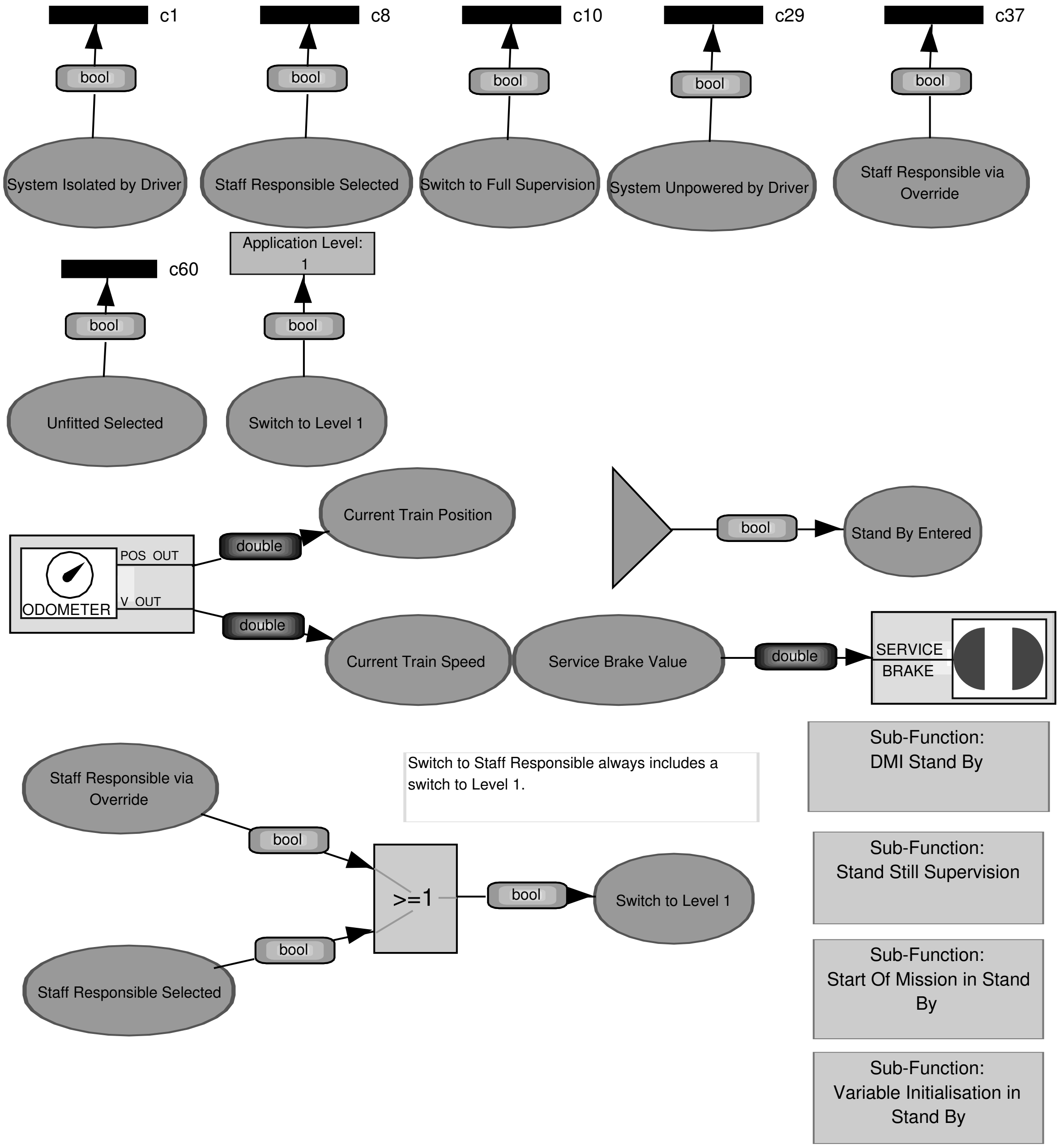

c1
c8
c10
c29
c37
bool
bool
bool
bool
bool
System Isolated by Driver
Staff Responsible Selected
Switch to Full Supervision
System Unpowered by Driver
Staff Responsible via Override
c60
Application Level: 1
bool
bool
Unfitted Selected
Switch to Level 1
Current Train Position
double
POS OUT
ODOMETER
V OUT
double
Current Train Speed
bool
Stand By Entered
Service Brake Value
double
SERVICE BRAKE
Staff Responsible via Override
bool
>=1
bool
Switch to Level 1
bool
Staff Responsible Selected
Switch to Staff Responsible always includes a switch to Level 1.
Sub-Function: DMI Stand By
Sub-Function: Stand Still Supervision
Sub-Function: Start Of Mission in Stand By
Sub-Function: Variable Initialisation in Stand By

Figure 10.4.: Stand By Mode in Application Level 0

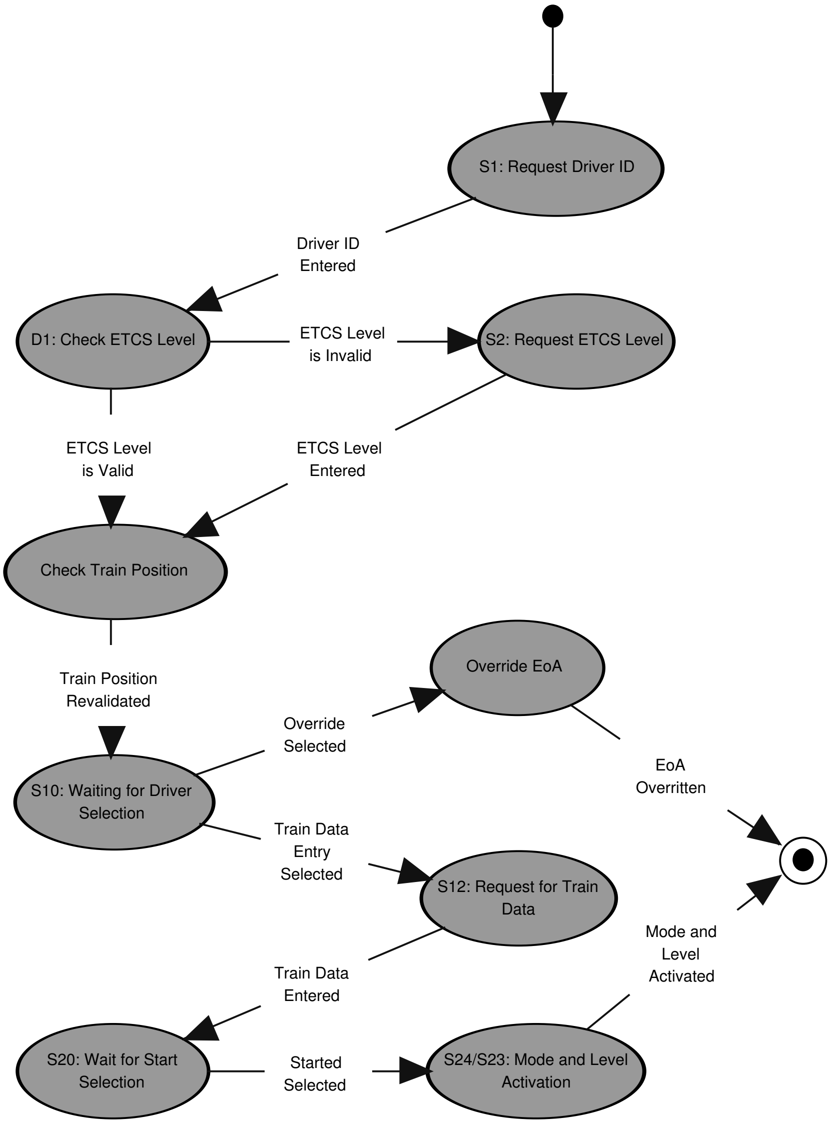


Figure 10.5.: Start of mission procedure as gEmbeddedStateMachine graph

Application Level 1 (and this is Level 0), the switch to Level 1 is activated by the same oVariableStorage object.

State "S12: Request for Train Data" waits for the values for the train length and maximum train speed via the DMI. The SRS defines more values for the train data, but which are not used in this case study.

"S20: Wait for Start Selection" only waits for the driver to select the start of the mission via the DMI.

"S24/S23: Mode and Level Activation" is another final state, which determines the new ETCS Mode based on the data entered. Either the guard "c60" is activated via the "Unfitted Selected" oVariableStorage object or guard "c37" is activated via the "Staff Responsible selected" object. In the latter case, also the switch to Application Level 1 is activated.

Application Level 1

Generally, Application Level 1 does not differ from the functionality in Level 0. Figure 10.6 sketches the corresponding model. In contrast to Figure 10.4, the activation of ETCS Mode Staff Responsible does not require a switch of the Application Level because the EVC is already in Level 1. On the other hand, here the transition to Mode Unfitted requires a switch to Level 0.

10.2.3. Unfitted Mode

Unlike the previous introduced Modes and their models, the Unfitted ETCS Mode is not used for the start-up procedure but for the train movement. Nevertheless, the support of ATP by ETCS (see Chapter 2) in this Mode is little because it is used on tracks that are unfitted or unequipped for ETCS. Thus, Unfitted is only available in Application Level 0. The corresponding gMainFunctionBlock is shown in Figure 10.7. The supervision of the speed according to the train's maximum speed is the only available ATP functionality. The maximum speed can be entered in Mode Stand By as train data. The supervision is modelled in a separate gSubFunctionBlock graph in Figure 10.8.

Since the train can move in Unfitted, balise telegrams can be received and must be evaluated accordingly. Mainly, this is relevant for the Transition Order Package [85, p. 14], which announces a switch to a new Application Level. According to the limits of this case study, this means a switch to Level 1, which also includes a transition to the Mode Staff Responsible or Full Supervision. Also, a Moving Authority (MA) telegram [85, p. 11] can be received, which is not used in Unfitted directly. It enables a transition to the Mode Full Supervision, in which a MA is always needed.

The evaluations of transition order and MA balise telegrams are both modelled by an oEmbeddedStateMachine instance that is located in the oSubFunction object "Telegram Evaluation in Unfitted". Figure 10.9 shows the model for the evaluation of transition orders and Figure 10.10 for MAs.

A transition order or rather the transition to a new Application Level always requires the acknowledgement of the Driver via the DMI within a certain time. If the driver does not

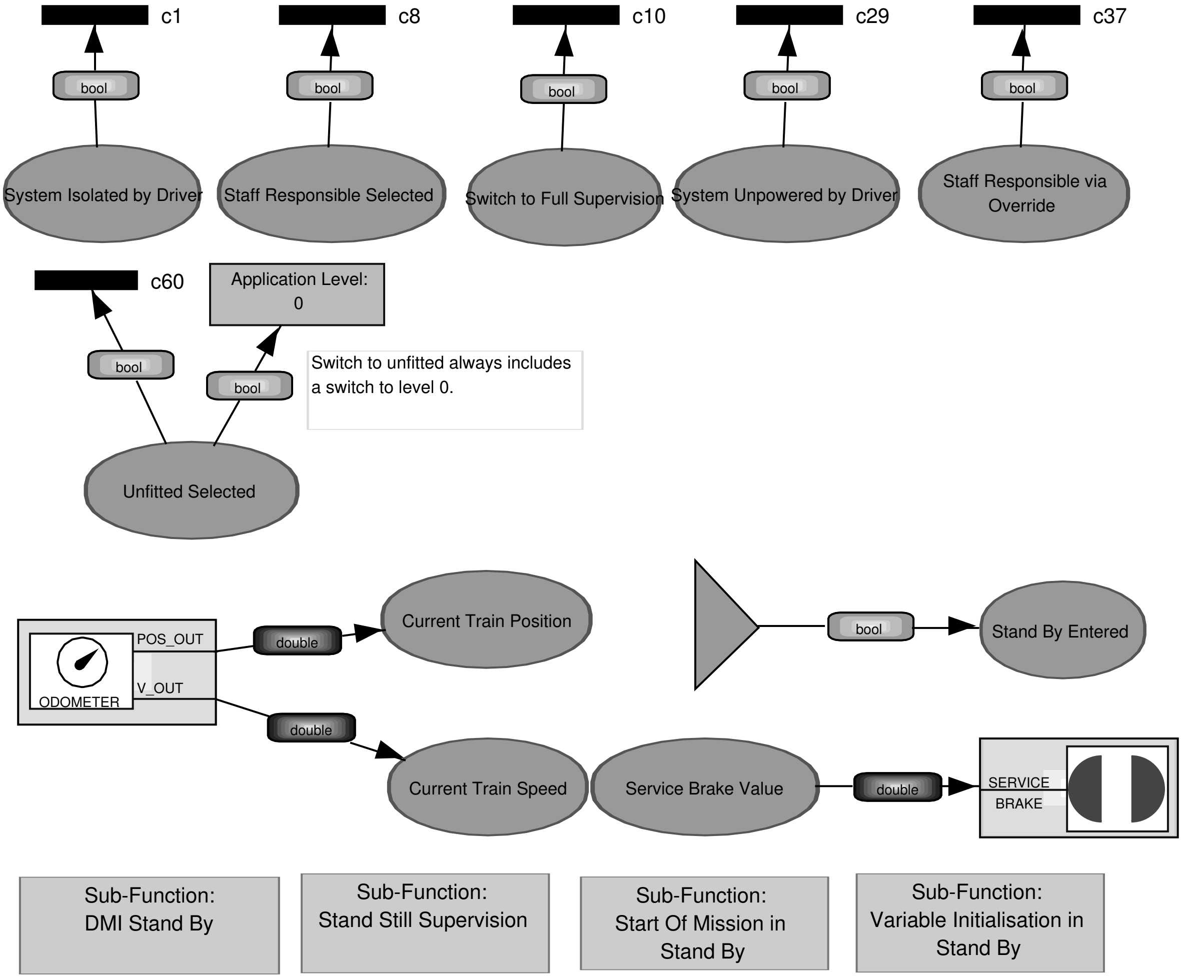

c1
c8
c10
c29
c37
bool
bool
bool
bool
bool
System Isolated by Driver
Staff Responsible Selected
Switch to Full Supervision
System Unpowered by Driver
Staff Responsible via Override
c60
Application Level: 0
bool
bool
Switch to unfitted always includes a switch to level 0.
Unfitted Selected
ODOMETER
POS_OUT
V_OUT
double
double
Current Train Position
Current Train Speed
bool
Stand By Entered
Service Brake Value
double
SERVICE BRAKE
Sub-Function: DMI Stand By
Sub-Function: Stand Still Supervision
Sub-Function: Start Of Mission in Stand By
Sub-Function: Variable Initialisation in Stand By

Figure 10.6.: Stand By Mode in Application Level 1

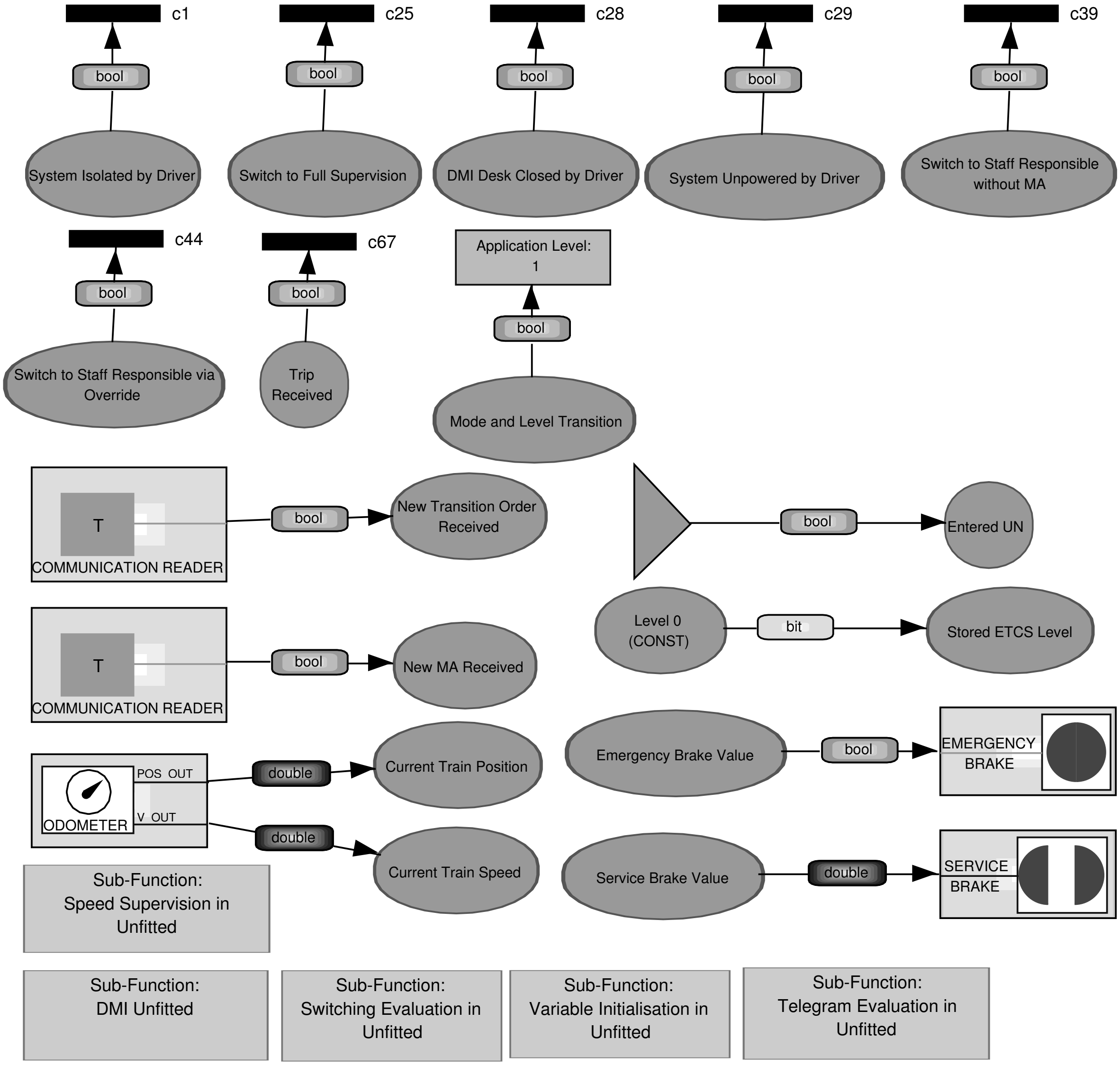

c1
bool
System Isolated by Driver
c25
bool
Switch to Full Supervision
c28
bool
DMI Desk Closed by Driver
c29
bool
System Unpowered by Driver
c39
bool
Switch to Staff Responsible without MA
c44
bool
Switch to Staff Responsible via Override
c67
bool
Trip Received
Application Level: 1
bool
Mode and Level Transition
T
COMMUNICATION READER
bool
New Transition Order Received
bool
Entered UN
Level 0 (CONST)
bit
Stored ETCS Level
T
COMMUNICATION READER
bool
New MA Received
POS OUT
V OUT
ODOMETER
double
Current Train Position
double
Current Train Speed
Emergency Brake Value
bool
EMERGENCY BRAKE
Service Brake Value
double
SERVICE BRAKE
Sub-Function: Speed Supervision in Unfitted
Sub-Function: DMI Unfitted
Sub-Function: Switching Evaluation in Unfitted
Sub-Function: Variable Initialisation in Unfitted
Sub-Function: Telegram Evaluation in Unfitted

Figure 10.7.: Unfitted Mode in Application Level 0

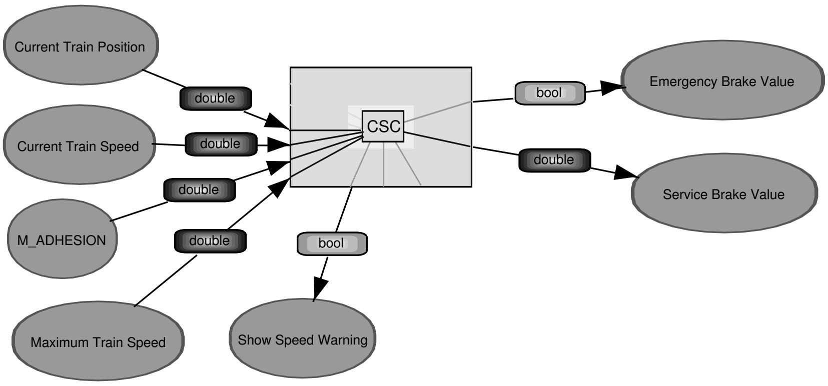


Figure 10.8.: Speed supervision in Unfitted gSubFunctionBlock graph

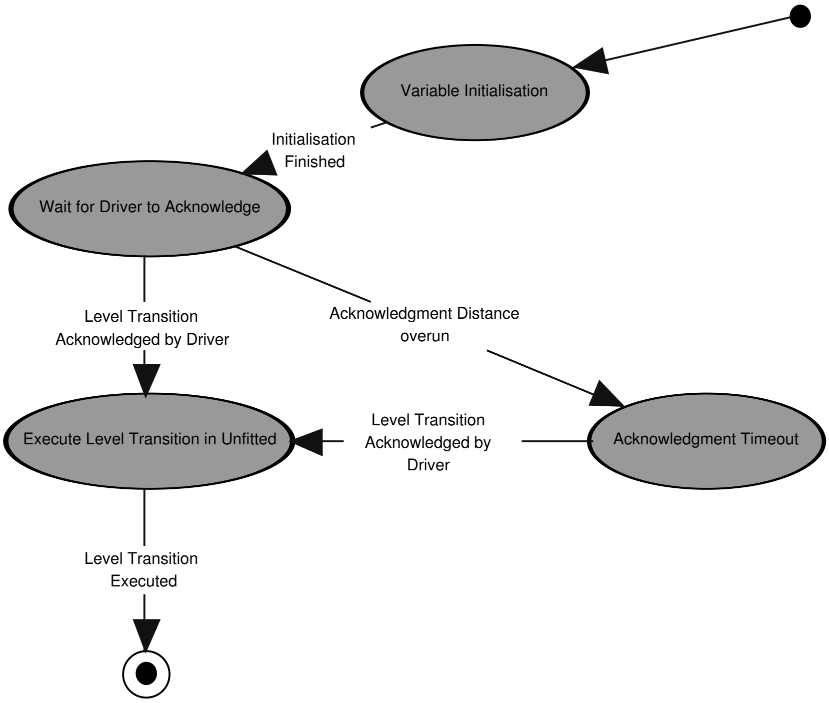

Variable Initialisation
Initialisation Finished
Wait for Driver to Acknowledge
Level Transition Acknowledged by Driver
Acknowledgment Distance overun
Execute Level Transition in Unfitted
Level Transition Acknowledged by Driver
Acknowledgment Timeout
Level Transition Executed

Figure 10.9.: Transition order evaluation in Unfitted as gEmbeddedStateMachine graph

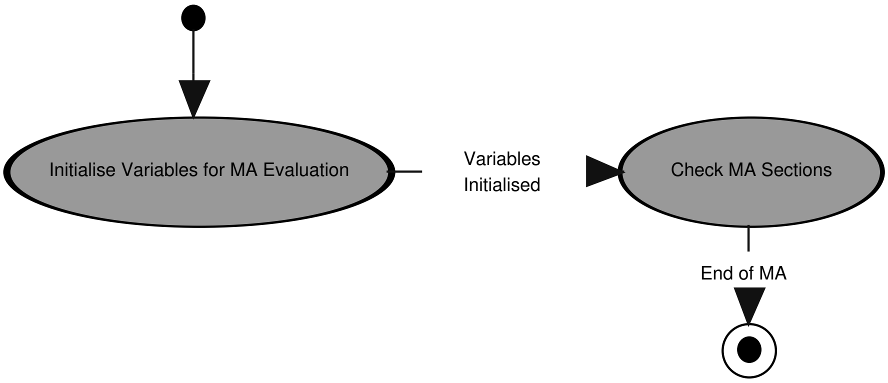


Figure 10.10.: Moving Authority evaluation in Unfitted as gEmbeddedStateMachine graph

acknowledge the transition in this time, the service brake is applied until it is acknowledged. After the acknowledgement, the oVariableStorage objects used for Mode transitions and Level switches are set to true and the transition order evaluation terminates.

Since in Unfitted a received MA is only used to ensure a safe transition to the Mode Full Supervision, the MA evaluation only checks if a received MA still is valid[2]. The result of this evaluation is stored as Boolean in the oVariableStorage object "MA Valid".

10.2.4. Staff Responsible Mode

The Staff Responsible Mode corresponds more or less to Unfitted in Subsection 10.2.3, but is only available in ETCS Application Level 1. The train is mainly moved under the responsibility of the driver and only few ATP functionality is available. Figure 10.11 represents the model of the main functionality. Compared to Unfitted, the following additional sub-functionality is provided:

Distance Supervision Ensures that a certain distance[3] is not overpassed in this Mode. Otherwise, any brake system is activated and oModeGuard "c42" is set to true. See Figure 10.12.

Monitoring of National Values Monitors any incoming balise telegram for new national values by using the "New National Values received" oVariableStorage object. See Figure 10.13.

Balise Linking Supervision Supervises the consistency of the linking of balise groups [89, p. 10ff]. If an error is detected, the oModeGuard "c36" is activated via the "Balise Link Error" oVariableStorage object. See Figure 10.14.

[2] by the submitted section length [85, p. 11]
[3] normally defined by a national value [85]

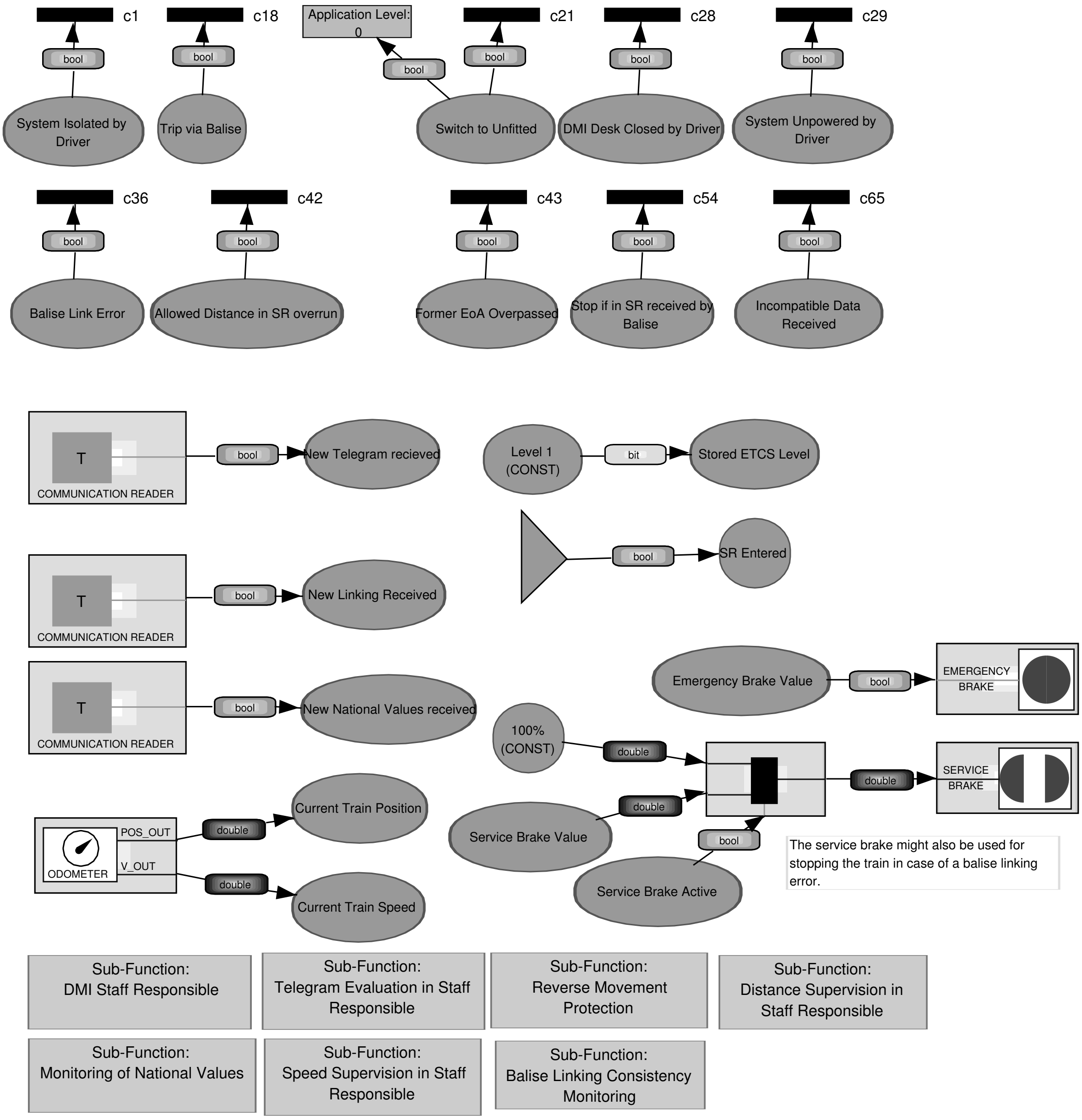

c1
c18
Application Level: 0
c21
c28
c29
bool
System Isolated by Driver
Trip via Balise
Switch to Unfitted
DMI Desk Closed by Driver
System Unpowered by Driver
c36
c42
c43
c54
c65
Balise Link Error
Allowed Distance in SR overrun
Former EoA Overpassed
Stop if in SR received by Balise
Incompatible Data Received
T
COMMUNICATION READER
New Telegram recieved
Level 1 (CONST)
bit
Stored ETCS Level
SR Entered
New Linking Received
Emergency Brake Value
EMERGENCY BRAKE
New National Values received
100% (CONST)
double
SERVICE BRAKE
Current Train Position
POS_OUT
V_OUT
ODOMETER
Service Brake Value
Service Brake Active
Current Train Speed
The service brake might also be used for stopping the train in case of a balise linking error.
Sub-Function: DMI Staff Responsible
Sub-Function: Telegram Evaluation in Staff Responsible
Sub-Function: Reverse Movement Protection
Sub-Function: Distance Supervision in Staff Responsible
Sub-Function: Monitoring of National Values
Sub-Function: Speed Supervision in Staff Responsible
Sub-Function: Balise Linking Consistency Monitoring

Figure 10.11.: Staff Responsible Mode in Application Level 1

Reverse Movement Protection Activates the emergency brake system in the case that the train moves in the reverse direction. See Figure 10.15.

Since different telegram types can be received in Staff Responsible, all of them must be treated in the corresponding sub-function "Telegram Evaluation in Staff Responsible". The evaluation of MAs is modelled analogously to the Unfitted Mode in Subsection 10.2.3 and is only needed for a possible transition to Full Supervision.

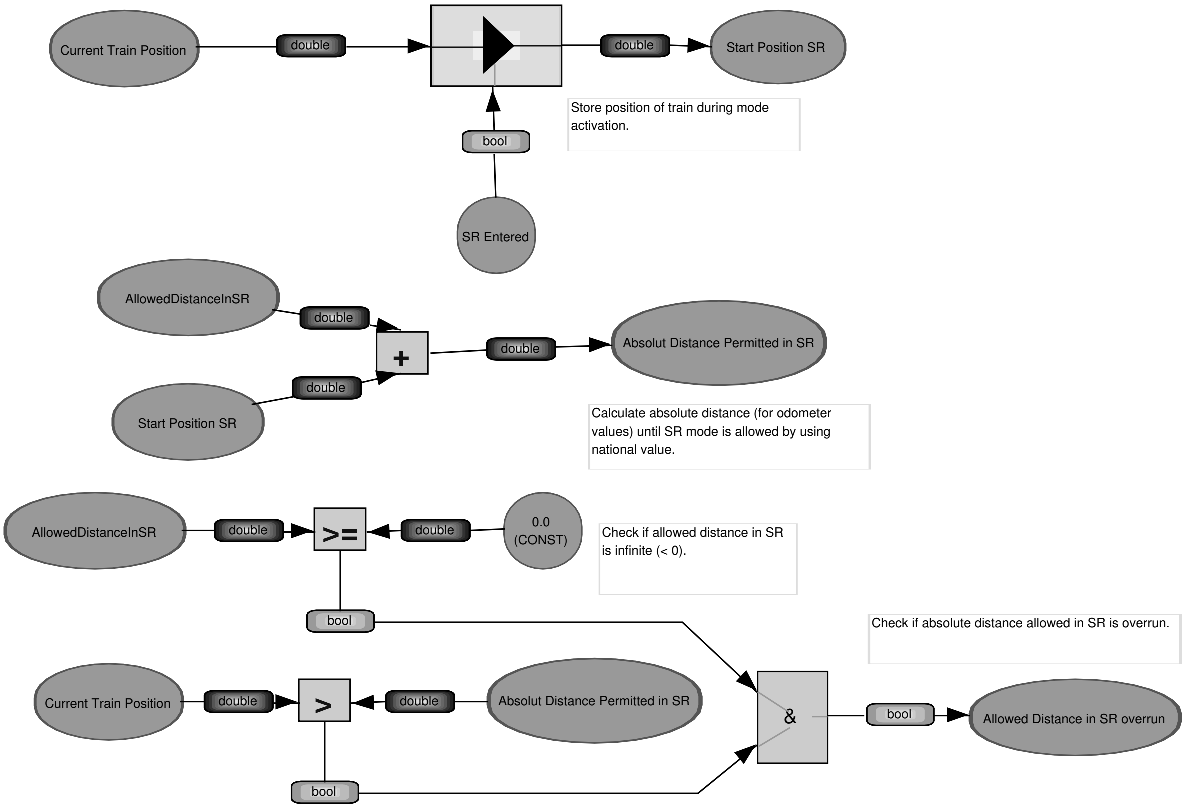

Current Train Position
double
double
Start Position SR
Store position of train during mode activation.
bool
SR Entered
AllowedDistanceInSR
double
+
double
Absolut Distance Permitted in SR
double
Start Position SR
Calculate absolute distance (for odometer values) until SR mode is allowed by using national value.
AllowedDistanceInSR
double
>=
double
0.0
(CONST)
Check if allowed distance in SR is infinite (< 0).
bool
Check if absolute distance allowed in SR is overrun.
Current Train Position
double
>
double
Absolut Distance Permitted in SR
&
bool
Allowed Distance in SR overrun
bool

Figure 10.12.: Distance supervision in Staff Responsible as gSubFunctionBlock graph

10.2.5. Full Supervision Mode

In Full Supervision, the EVC completely supervises the train movement. To ensure that the train only enters allowed areas of the track, Moving Authorities (MA) [89, pp. 35-47] are used. It is ensured that the train never overpasses the end of a MA without a new valid one has been received before. In general, this procedure is done to provide an automatic spacing of trains [67, ch. 3]. The model of the main functionality is sketched in Figure 10.16.

In comparison to Staff Responsible, Full Supervision mainly adds the functionality for automatic train spacing. This is modelled in the "Dynamic Speed Profile Supervision" in Figure 10.17.

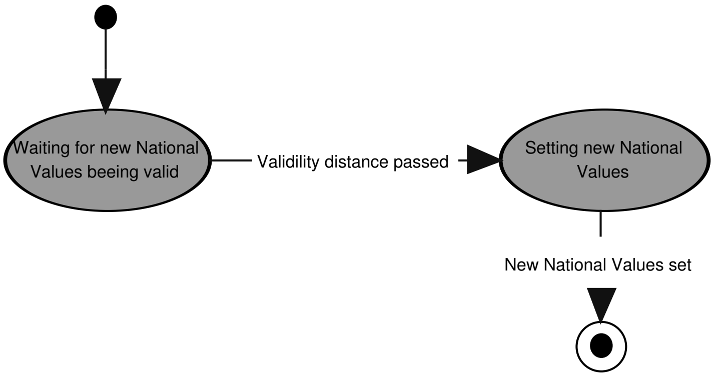

Waiting for new National Values beeing valid
Validility distance passed
Setting new National Values
New National Values set

Figure 10.13.: Evaluation of national values as gEmbeddedStateMachine graph

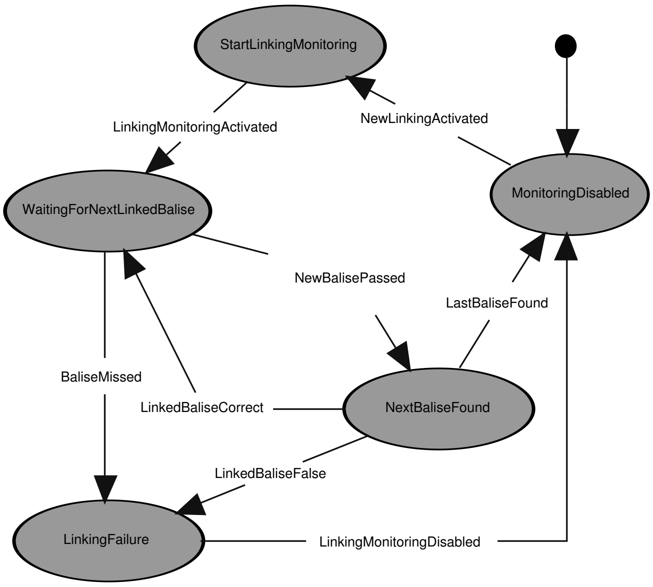


Figure 10.14.: Balise linking supervision as gEmbeddedStateMachine graph

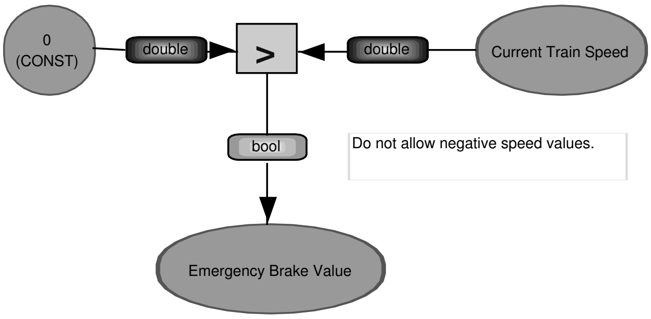


Figure 10.15.: Reverse movement protection as gSubFunctionBlock graph

In contrast to the supervision of train speed in respect to a constant top or ceiling speed in Unfitted or Staff Responsible, here the speed is additionally controlled corresponding to the end of the current valid MA. Furthermore, the allowed speed at the end of a MA must not always be 0. This supervision corresponds to a typically braking curve supervision [67, pp. 95-97] and is represented in the model by an oBrakingToTargetSpeed (BTS) object (see Subsection 7.3.2). “Current V_LOA” (v_{loa}) holds the speed that is currently permitted at the end / limit of authority, and “Distance of EoA” is the absolute position of the end of the authority. In the case that $v_{\mathrm{loa}} > 0$ and the train has overpassed the end of authority, it can move with v_{loa} until a certain overlap distance [89, pp. 35-47] is passed. In this case, the Mode is switched to Trip, which is modelled by the “c12” oModeGuard object in Figure 10.16.

10.2.6. Trip Mode

Trip Mode is always activated if an operational[4] failure occurs that requires the train fully to stop. This can be, for example, the case for an overpassed stop signal and a corresponding balise telegram or if the balise linking supervision (see Subsection 10.2.4) fails.

According to the ETCS SRS, Trip is not available in Application Level 0 [88, p. 23] but in all other levels. This is in conflict with the fact that Trip is also reachable from Unfitted [88, p. 37] (see Figure 10.1 for the corresponding model). Furthermore, successor Modes of Trip can only be Post Trip (oModeGuard “c62”) and Unfitted (oModeGuard “c7”) while condition 62 or rather oModeGuard object “c62” is defined as

> (the driver acknowledges the train trip) AND (the train is at standstill) AND (the ERTMS/ETCS level is 0) [88, p. 58]

Since this is in conflict with definition of available Application Levels for Unfitted in the SRS [88, p. 23], Unfitted is also modelled for Level 0 in this case study.

The main functionality in Trip is to stop the train by applying emergency brakes until the train is fully stopped.

[4] Compared to system failures, operational failures are related to the train operations and not to the train control system.

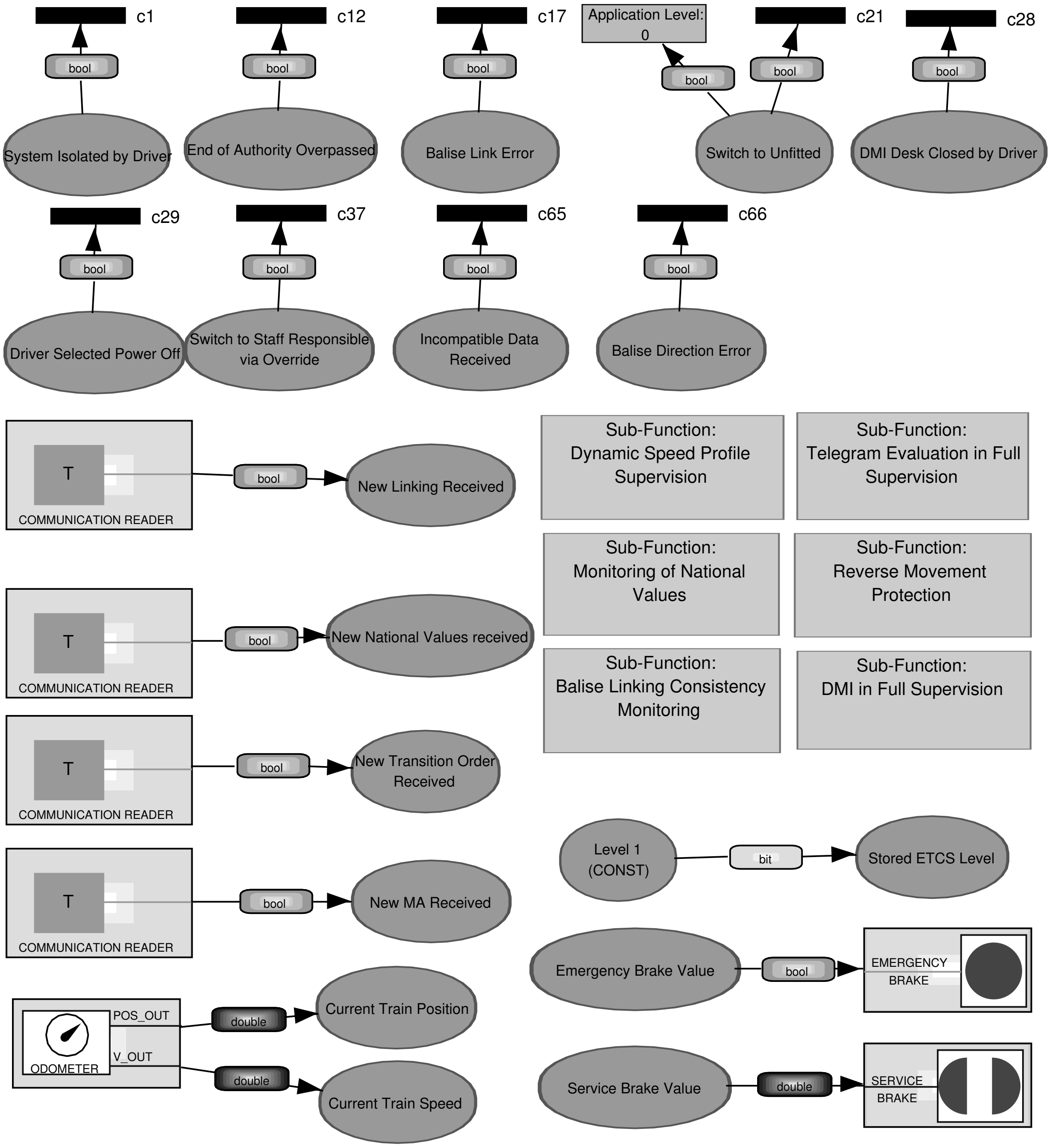

c1
bool
System Isolated by Driver
c12
bool
End of Authority Overpassed
c17
bool
Balise Link Error
Application Level: 0
bool
c21
bool
Switch to Unfitted
c28
bool
DMI Desk Closed by Driver
c29
bool
Driver Selected Power Off
c37
bool
Switch to Staff Responsible via Override
c65
bool
Incompatible Data Received
c66
bool
Balise Direction Error
T
COMMUNICATION READER
bool
New Linking Received
T
COMMUNICATION READER
bool
New National Values received
T
COMMUNICATION READER
bool
New Transition Order Received
T
COMMUNICATION READER
bool
New MA Received
ODOMETER
POS_OUT
V_OUT
double
Current Train Position
double
Current Train Speed
Sub-Function: Dynamic Speed Profile Supervision
Sub-Function: Telegram Evaluation in Full Supervision
Sub-Function: Monitoring of National Values
Sub-Function: Reverse Movement Protection
Sub-Function: Balise Linking Consistency Monitoring
Sub-Function: DMI in Full Supervision
Level 1 (CONST)
bit
Stored ETCS Level
Emergency Brake Value
bool
EMERGENCY BRAKE
Service Brake Value
double
SERVICE BRAKE

Figure 10.16.: Full Supervision Mode in Application Level 1

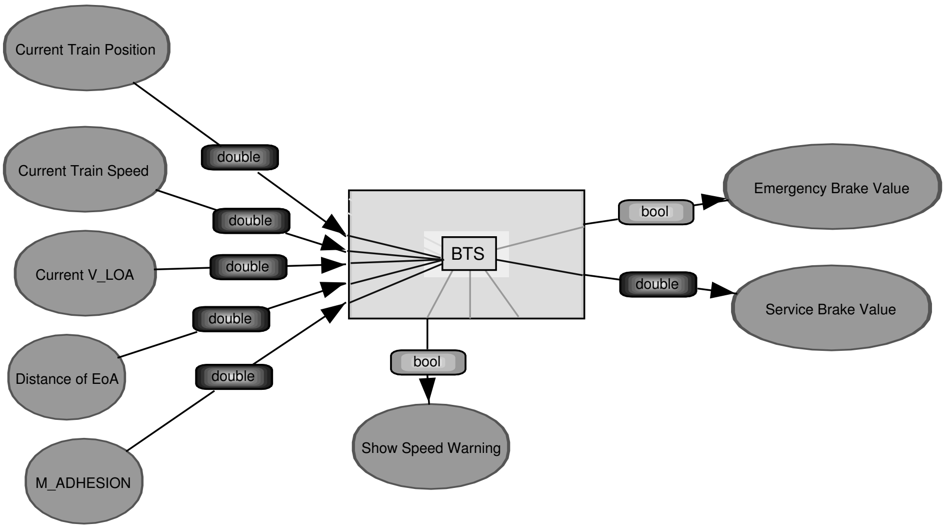


Figure 10.17.: Dynamic speed profile supervision as gSubFunctionBlock graph

10.2.6.1. Application Level 0

Figure 10.18 introduces the model for Trip in Level 0. As already described, the emergency brakes are permanently applied by a data flow from a constant set to true. The supervision of the train stop is modelled in the oSubFunction object “Train Stop Supervision”, which is sketched in Figure 10.19. Accordingly, the oVariableStorage object “Train Stopped” is set to true if the train is fully stopped. In this case, the driver is able to acknowledge the Trip via the DMI. The conjunction of those conditions is used as input for the oModeGuard “c62”, which activates then the transition back to Unfitted.

10.2.6.2. Application Level 1

The model for Trip in Level 1 only differs from the one in Level 0 in the activated oModeGuard object “c7”, which is shown in Figure 10.20. “c7” activates the transition to Post Trip.

10.2.7. Post Trip Mode

The Post Trip Mode is only reachable via Trip and is used to move the train slowly backwards after an operational failure. For example, to get in front of an overpassed stop point. Accordingly, Post Trip is only available in Application Level 1. Its model is shown in Figure 10.21. The reverse movement supervision is modelled in the oSubFunction object “Reverse Movement Supervision in Post Trip”, which is sketched in Figure 10.22. According to the SRS, in Post Trip, the train is allowed to move about a certain distance (a national value [89, pp. 89-90]) in reverse direction. If this distance is overpassed, the service brake is applied to 100%. Any forward movement is always inhibited. In contrast to Trip in Subsection 10.2.6, the emergency brakes are always released.

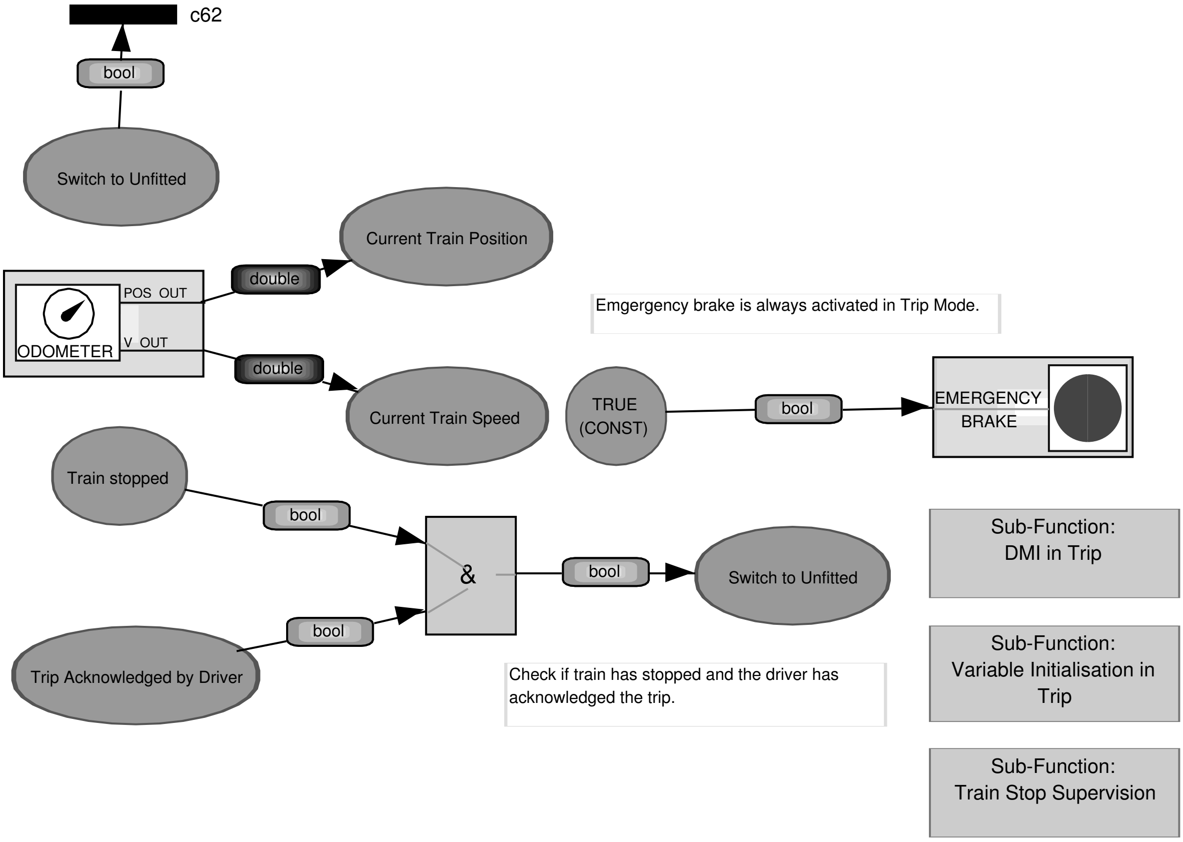


Figure 10.18.: Trip Mode in Application Level 0

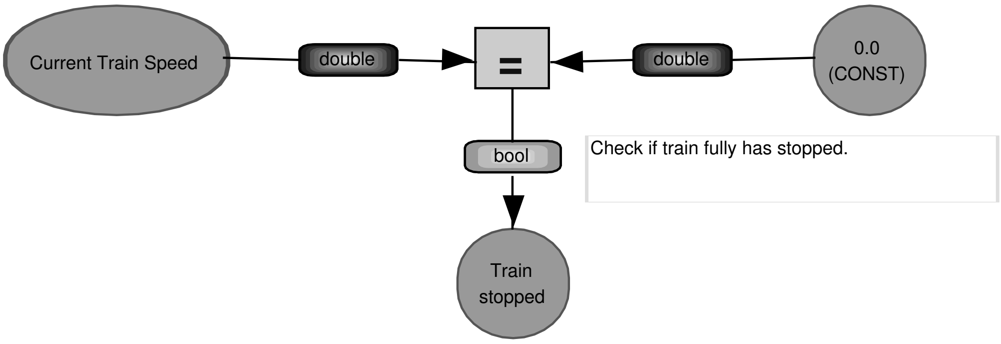


Figure 10.19.: Train stop supervision in Trip as gSubFunctionBlock graph

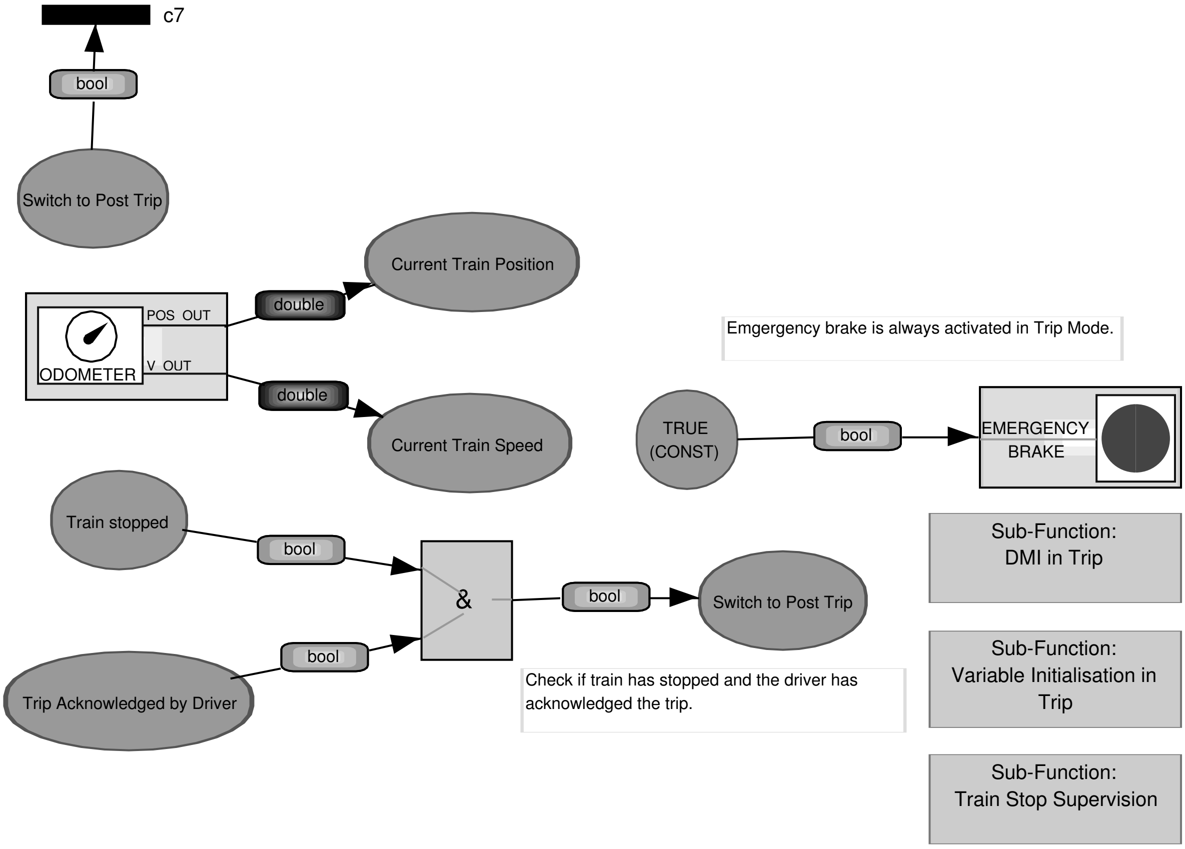


Figure 10.20.: Trip Mode in Application Level 1

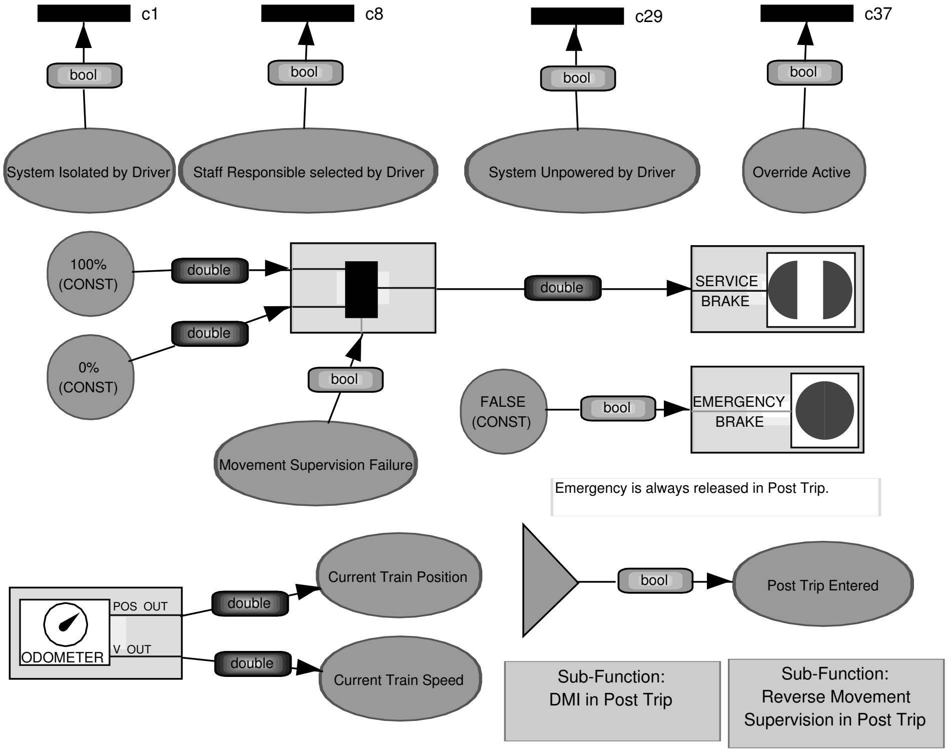


Figure 10.21.: Post Trip Mode in Application Level 1

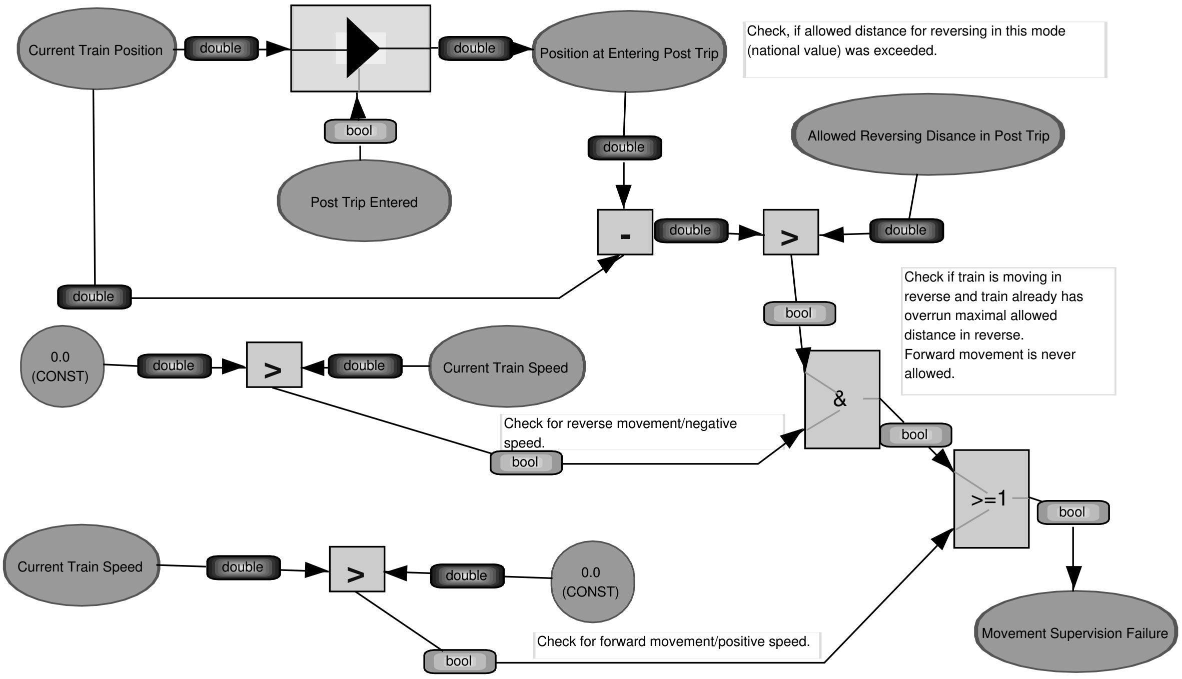

Current Train Position
double
double
Position at Entering Post Trip
Check, if allowed distance for reversing in this mode (national value) was exceeded.
bool
Post Trip Entered
double
Allowed Reversing Disance in Post Trip
-
double
>
double
double
Check if train is moving in reverse and train already has overrun maximal allowed distance in reverse. Forward movement is never allowed.
bool
0.0 (CONST)
double
>
double
Current Train Speed
&
Check for reverse movement/negative speed.
bool
bool
>=1
bool
Current Train Speed
double
>
double
0.0 (CONST)
Movement Supervision Failure
Check for forward movement/positive speed.
bool

Figure 10.22.: Reverse movement supervision in Post Trip as gSubFunctionBlock graph

10.2.8. System Failure Mode

System Failure is activated if an error is detected in the EVC during the execution in other ETCS Modes that causes a system failure [78, p. 12]. As described in Section 7.2, the corresponding oModeGuard object “c13” in Figure 10.1 is typically not activated by a Boolean data flow, but is used as FailureGuard property in the corresponding gEVCStateMachine instances. Since a system failure means that the system or rather the EVC cannot longer perform its required functionality, this is a threat to safety. Correspondingly, the SRS defines for the System Failure Mode that the emergency brakes are permanently activated. Since an error may occur in any Mode and any Application Level, System Failure is modelled for Application Level 0 and 1.

10.2.8.1. Application Level 0

The model of System Failure in Level 0 is sketched in Figure 10.23. Once this Mode is entered, it can only be left by input from the driver via the DMI. It can be either switched to No Power (Subsection 10.2.1) via the “c29” oModeGuard object or to Isolation (Subsection 10.2.9) via the “c1” oModeGuard object.

10.2.8.2. Application Level 1

Since the failure handling does not differ for different Application Levels, the model for Level 1 matches exactly the model of Level 0 and is omitted.

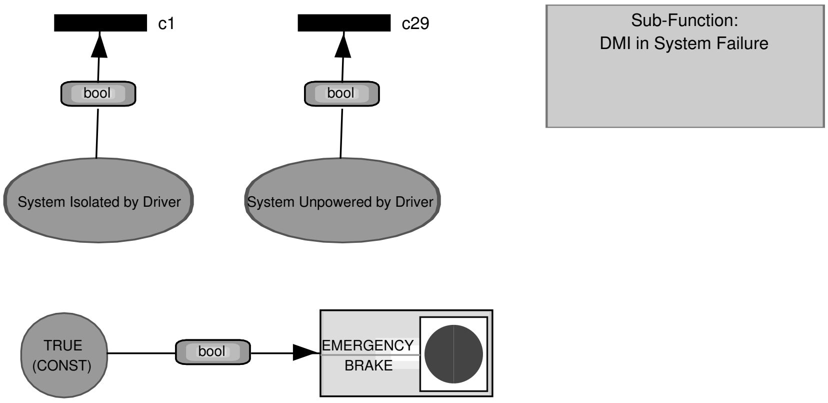


Figure 10.23.: System Failure Mode in Application Level 0

10.2.9. Isolation Mode

The SRS defines that in Isolation the ETCS on-board equipment is isolated from the train and does not have any responsibilities for train safety [88, p. 9]. Since isolation means from the perspective of the openETCS (meta) model that the object types representing hardware interfaces (e.g. oServiceBrake) are not available, the model only consists of oDMIOutput objects for displaying the information about he current state to the driver.

10.2.9.1. Application Level 0

The trivial model for Application Level 0 is shown in Figure 10.24.

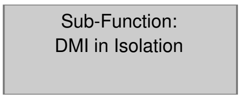


Figure 10.24.: Isolation Mode in Application Level 0

10.2.9.2. Application Level 1

Application Level 1 directly corresponds to Level 0 and is omitted.

10.3. Incoming Balise Telegrams

According to the sub-graph structure of the openETCS meta model in Section 7.2, the gCommunicationReader graph type is a child of the gMainFunctionBlock type, which is referenced by decompositions of oCommunicationReader objects. Thus, the following subsections contain a description of gCommunicationReader instances that are often used in the openETCS model.

The purpose of these graphs is to define how data is extracted for incoming telegrams (see Subsection 7.3.5) but not to define the structure of telegrams. This is done in the graph instances for the ETCS language in the upcoming Section 10.4.

It should be noted that the model of the openETCS case study does not contain any gCommunicationSender graphs because the selected subset of the specification or rather the sub-subset of Subset-026 (of the SRS) does not include sending from train to track-side functionality.

10.3.1. Level Transition Order Reading

The graph for reading a telegram with a level transition order [85, p. 14] is used in most Modes for moving the train: Unfitted, Staff Responsible, and Full Supervision. The corresponding model is displayed in Figure 10.25.

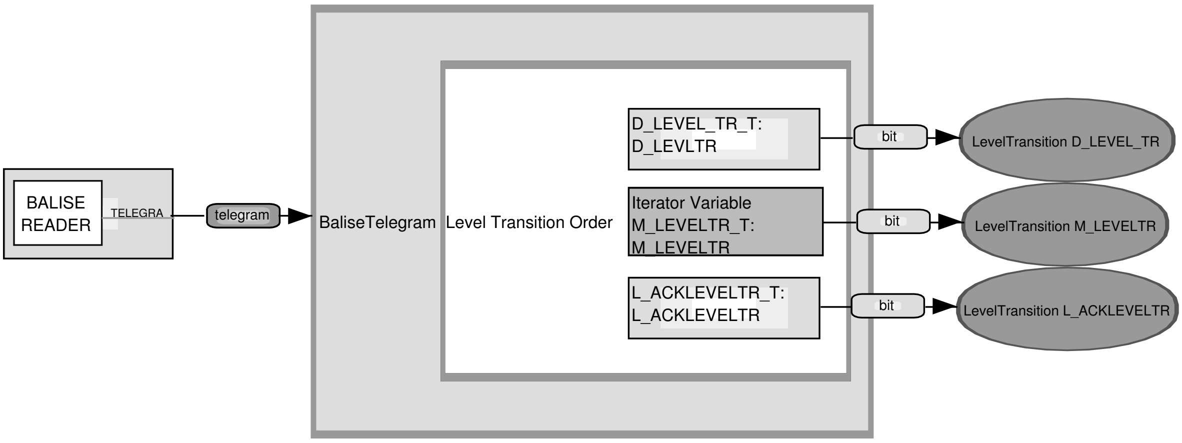

BALISE READER
TELEGRA
telegram
BaliseTelegram
Level Transition Order
D_LEVEL_TR_T: D_LEVLTR
Iterator Variable M_LEVELTR_T: M_LEVELTR
L_ACKLEVELTR_T: L_ACKLEVELTR
bit
bit
bit
LevelTransition D_LEVEL_TR
LevelTransition M_LEVELTR
LevelTransition L_ACKLEVELTR

Figure 10.25.: Reading of a level transition order as gCommunicationReader graph

The currently used data in the case study are the following information:

1. the Application Level to be switched to (“M_LEVELTR”)
2. the distance to the point where the transition takes place (“D_LEVELTR)
3. the distance to the point until the driver has to acknowledge the transition (”L_ACKLEVELTR“)

10.3.2. National Values Reading

This graph describes the reading of national values, which is part of most ETCS Modes in Application Level 1. Its model is shown in Figure 10.26. The meaning of each value can be found in [85, pp. 31-62].

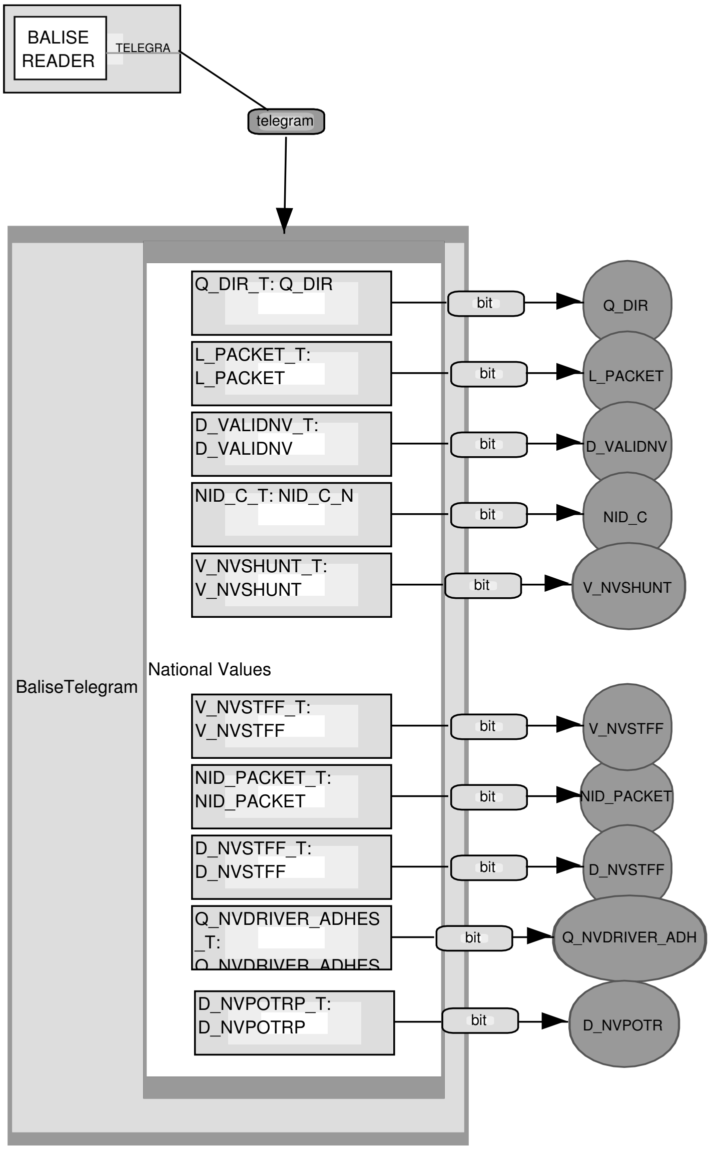


Figure 10.26.: Reading of new national values as gCommunicationReader graph

10.3.3. General Balise Telegram Reading in Staff Responsible

In Staff Responsible, the general balise reading is used to extract the header [87, p. 9] of all incoming balise telegrams. Additionally, the information from a "Stop if in Staff Responsible" [85, p. 24] packet is extracted if it is included.

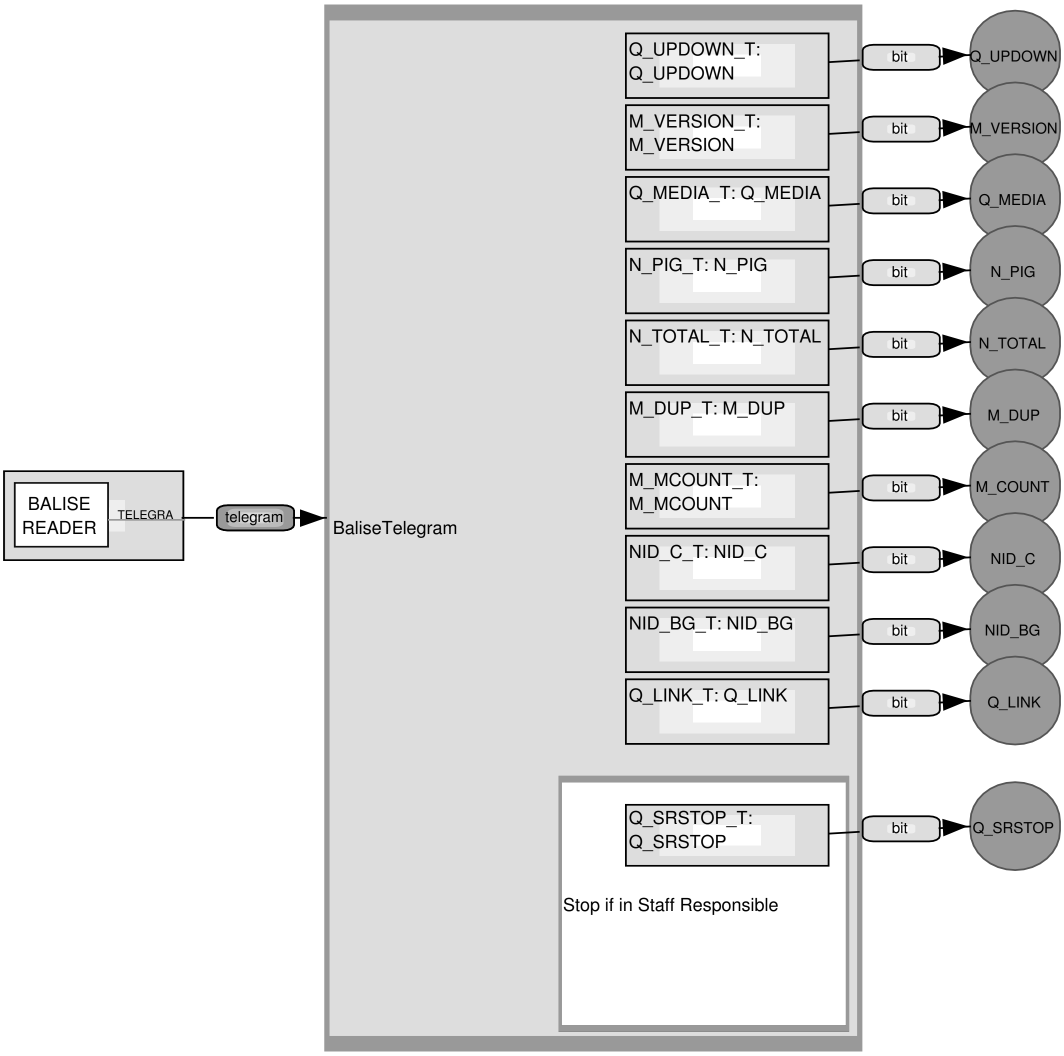


Figure 10.27.: General reading of balise telegrams in Staff Responsible as gCommunicationReader graph

The "Q_SRSTOP" [85, p. 55] oVariableInstance object holds the flag if the train should be stopped if the current Mode is Staff Responsible.

10.4. The ETCS Language

In contrast to all preceding graphs, the language part of the openETCS model describes only data structures and not behaviour.

10.4.1. Balise Telegram

The modelling of the balise telegram is done in a gTelegram graph. Since it defines the structure of any possible telegram, it is only one graph instance required for the complete openETCS model, which is shown in Figure 10.28. The oVariableInstance objects ”Q_UPDOWN“ down

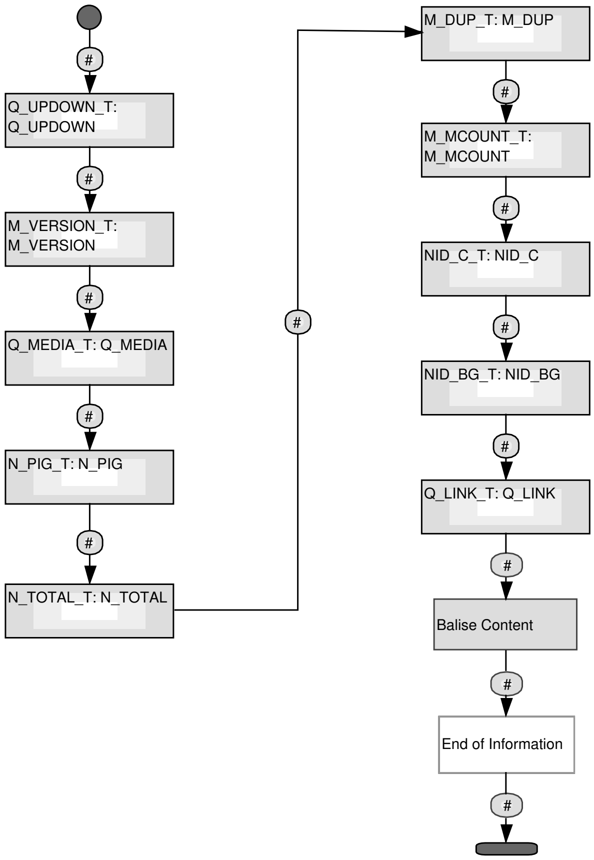


Figure 10.28.: Balise telegram structure as gTelegram graph

to ”Q_LINK“ are used as header for any type of balise telegram [87, p. 9]. They hold the general information about the telegram, like the transmission direction (”Q_UPDOWN” [85, p. 57]), the related ETCS version (“M_VERSION” [85, p. 44]), and the used transmission media (“Q_MEDIA” [85, p. 52]).

The oAnyPacket object labelled “Balise Content” defines all possible ETCS packets that may occur after the header. The concrete structure representation as gAnyPacket graph will be explained in the following Subsection 10.4.2. A balise telegram always terminates with an “End of Information” packet [85, p. 30].

10.4.2. Balise Content

All available ETCS packets are modelled in a gAnyPacket graph. According to the initial description of the gAnyPacket graph type in Subsection 7.3.8, the model in Figure 10.29 only holds a set of oPacket objects without any bindings. The models or rather decompositions of some exemplary packets will be discussed in the following subsections.

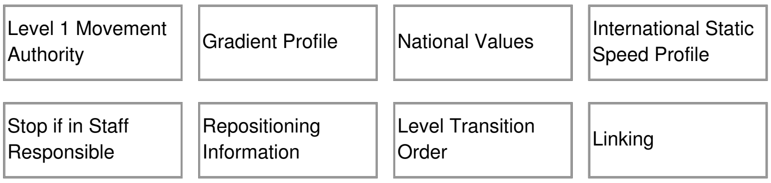

Level 1 Movement Authority
Gradient Profile
National Values
International Static Speed Profile
Stop if in Staff Responsible
Repositioning Information
Level Transition Order
Linking

Figure 10.29.: Available packets as gAnyPacket graph

10.4.3. Level Transition Order Packet

The “Level Transition Order” packet [85, p. 14] is used to order a train to switch to a new Application Level. This was described for the Unfitted Mode in Subsection 10.2.3. The three very first oVariableInstance objects define only general information of the packet. “NID_PACKET” [85, p. 47] is the unique numerical ID of each packet, “Q_DIR” [85, p. 49] defines, which movement directions of the train the packet is valid for, and “L_PACKET” [85, p. 37] is the size of the telegram in bits.

“Q_SCALE” [85, p. 54] is a scaling qualifier, which is applied by the “Scaling” relationship (see Subsection 7.3.7) to all distances and lengths in this packet. “D_LEVELTR” [85, p. 32] holds the distance to the level transition itself. On the one hand, “M_LEVELTR” [85, p. 41] holds the information about the new Application Level to be switched to. On the other hand, it is also a conditional iterator for the oVariableInstance object “NID_STM” [85, p. 37], which holds the identifier to the required STM. Nevertheless, because this case study only includes Application Level 0 and 1, this oVariableInstance object is never used and is only modelled for completeness.

“L_ACKLEVELTR” [85, p. 36] represents the length of the area, in which the driver has to acknowledge the level transition via the DMI.

The oVariableInstance object “N_ITER” [85, p. 4] gives the possibility to define multiple Level transitions in one packet by iterating over oVariableInstance objects of the same oVariableType objects (“M_LEVELTR_T”, “NID_STM_T”, and “L_ACKLEVELTR_T”) that were used before.

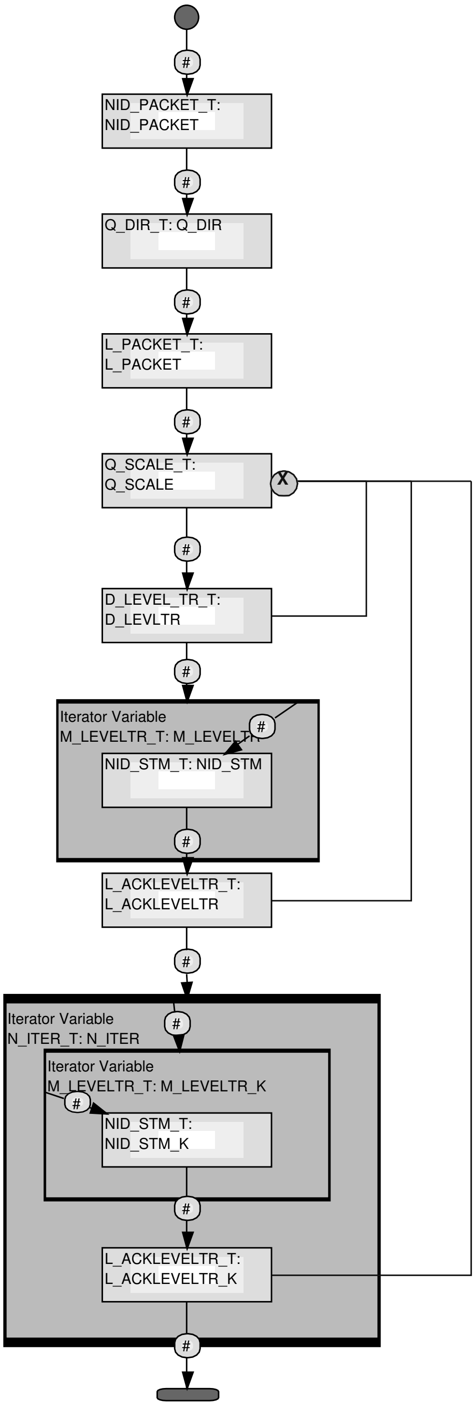


Figure 10.30.: Level transition order as gPacket graph

10.4.4. Stop if in Staff Responsible Packet

The “Stop if in Staff Responsible” packet [85, p. 24-25] is used to stop the train by a transition to the Trip Mode if the current ETCS Mode is Staff Responsible. The corresponding behavioural model was introduced in Subsection 10.2.4 while the model of the packet structure is displayed in Figure 10.31. Compared with the preceding packet in Subsection 10.4.3, this packet is more simple and does not have any scaling factors or iterating variables. The first three oVariableInstance objects are the same as in the the “Level Transition Order” while “Q_SRSTOP” [85, p. 55] is a qualifier to determine if the train should be stopped.

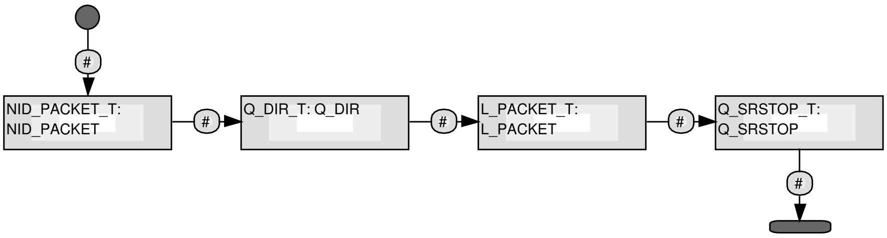


Figure 10.31.: Stop if in Staff Responsible packet as gPacket graph

10.4.5. End of Information Packet

Corresponding to the model in Subsection 10.4.1, every balise telegram terminates with an “End of Information” [85, p. 30] packet to guarantee the integrity of the submitted data. Thus, the packet only contains one oVariableInstance: “NID_PACKET”, which is used in all packet types [85]. Its model is shown in Figure 10.32.

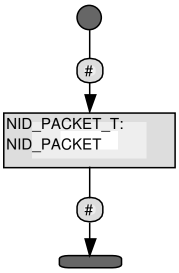


Figure 10.32.: End of information packet as gPacket graph

10.5. Model Extensions

Model extensions are mainly needed in the case that an existing model[5] has to be extended or adapted for a newer version of the ETCS SRS. Ideally, these extensions only have to be done in the model but not in the meta model because any changes there might require additional modifications for the code generators and the domain framework.

On the other hand, it seems doubtful that is possible to foresee any potential changes of the SRS in the future for the design of the meta model or rather the openETCS formal specification Language (in Figure 7.1).

Furthermore, even extensions on the model level should be traceable [78, p. 323][74] because those are always related to the safety of the modelled system. One possibility is to trace changes in the model by identifying the difference (of model elements) between two model / SRS versions. Unfortunately, finding the difference in graphical models is not always trivial because a modification of the graphical representation does not always mean a modification of the abstract syntax model or the dynamic semantics. A loophole to this situation is a transformation to a model representation that only holds the abstract syntax..

The GOPPRR meta model extension is such a model representation, which was introduced in Chapter 4. Especially, the intermediate XML model in Section 4.3 can be directly used to identify the syntactical differences between two models by using standard tools. Additionally, a custom application could be developed that uses the GOPPRR C++ abstract syntax model in Section 4.2.

In contrast to identifying model extensions between different model versions, it is also possible to take model extensions directly into account for the design of the meta model. In other words, the openETCS formal specification could offer a syntax to directly express extensions due to new SRS versions in the graphical model. This would have the advantage of a more abstract representation, compared to building the difference of XML files but also would increase the complexity of the meta model and the corresponding model.

10.6. Conclusion

This chapter introduced the concrete model of the openETCS case study for a sub-subset of the ETCS SRS. Representative examples of the model from all levels respectively graph types were presented and described. The complete openETCS model can be found in Appendix C.

Furthermore, an important issue of model extensions for new ETCS SRS versions was illuminated and a possible outlook on how the traceability of model extensions might be increased in future work was given.

[5] in terms of MDA the CIM

11 openETCS Simulation

Generally, a simulation is applicable to test the behaviour of the generated system to evaluate the openETCS case study and the developed tool chain as proof of concept for model based development and tests for openETCS. According to the MDA principle, which is used for the whole tool chain, also this simulation should be developed based on models.

This chapter starts with an initial description of the simulation methodology, which means how the simulation environment is set up and can interact with a generated EVC binary. Afterwards, the PSM (see Section 8.5) for the simulation is introduced, followed by the simulation model using it.

11.1. Simulation Methodology

The simulation environment should access the generated and compiled binary, which shall be executed on the train on-board unit, the EVC, through special simulative hardware interfaces[1] in the PSM and DMI. The simulation is modelled by two independent state machines: One for simulating the ETCS Modes and Application Levels based on a virtual track. The other for describing the driver's behaviour by the interaction with the DMI. The data flow between simulation and on-board binary or rather PSM is sketched in Figure 11.1.

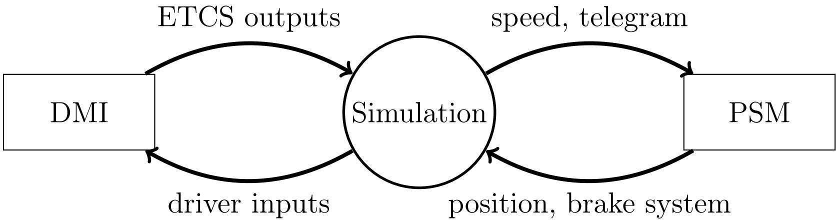


Figure 11.1.: Simulation environment data flow

[1] for odometer, service and emergency brake, and for receiving balise telegrams

The simulation itself is modelled by UML state charts [64] while the source code is generated by the RT-Tester [72] application. The corresponding generation procedure is shown in Figure 11.2. libopenETCSPSMSIM is used to provide the necessary platform specific adaptations by the corresponding D-Bus adaptors, which is explained in detail by the simulation deployment in Subsection 11.2.2.

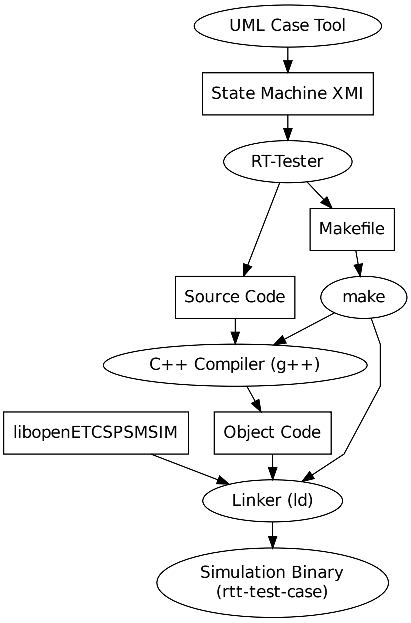


Figure 11.2.: Generators and artefacts for the simulation development

Both simulation state machines are executed together in one thread and share memory for data exchange. The one modelling ETCS Modes and Application Levels should be created directly from the ETCS specification [23].

The idea is to proof the concept of the model based tool chain for openETCS by showing that a simulation representing the ETCS specification for Modes and transitions stimulating the EVC binary, which was generated by a model based on the ETCS specification, produces the same traces of modes and transitions as the EVC binary. In other words, if a Mode transition within the generated binary is executed, this should be also found in the simulation and vice versa. The data flow of the trace generation is shown in Figure 11.3.

11.2. Platform Specific Model for the Simulation

According to Figure 8.15, the adaptations needed for simulation purposes must be realised as a PSM for an openETCS CIM or rather PIM. The special requirements for this PSM for the simulation are defined in the following:

Req.13: transparent integration into the existing openETCS domain framework

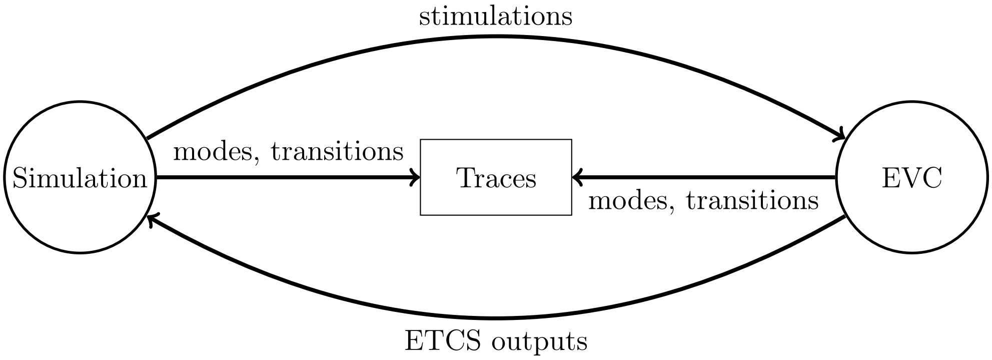


Figure 11.3.: Data flow of the generation of traces for Modes and transitions

Req.14: simulation of the physical behaviour of the train
Req.15: access to DMI inputs and outputs

Req.13 The simulative PSM must be developed using the already proposed platform-specific adaptor stubs in Figure 8.18 to make it directly usable for any openETCS model / CIM. Thus, those stubs or rather their methods only have to be implemented with the simulative functionality by class inheritance.

Req.14 Since the simulation uses no hardware components, the physical behaviour must be simulated, too. This means, for example, if a brake system (service or emergency) is applied, the resulting output value of the odometer should be modified in a physically meaningful manner. Therefore, a physical model for the train movement and for brakes must be implemented in the simulative PSM.

Req.15 Corresponding to Figure 11.1, the simulation does not only need access to hardware interfaces in the PSM because the driver's behaviour should be modelled by manipulating inputs of the DMI. Furthermore, DMI outputs must be used to trace the state and the behaviour of the EVC binary, as it is defined in Figure 11.3. Therefore, an additional interface for accessing the DMI via D-Bus has to be defined and implemented.

11.2.1. Structural Design

According to Req.13, the simulative PSM should implement the provided adaptor stubs. This is done by inheritance from the stub classes, as it is shown in the UML class diagram in Figure 11.4. For simplification, all class names used in this section refer to the simulative classes in the `::oETCS::DF::PS::SIM` UML package. The physical model of the train is located in the CServiceBrake class, which uses a separated thread for the permanent calculation. This is indicated by the composition `m_pPhysicalCalculation` to the ::std::thread class. The COdometer class holds – due to its derivation from the stub class in the PS package – the

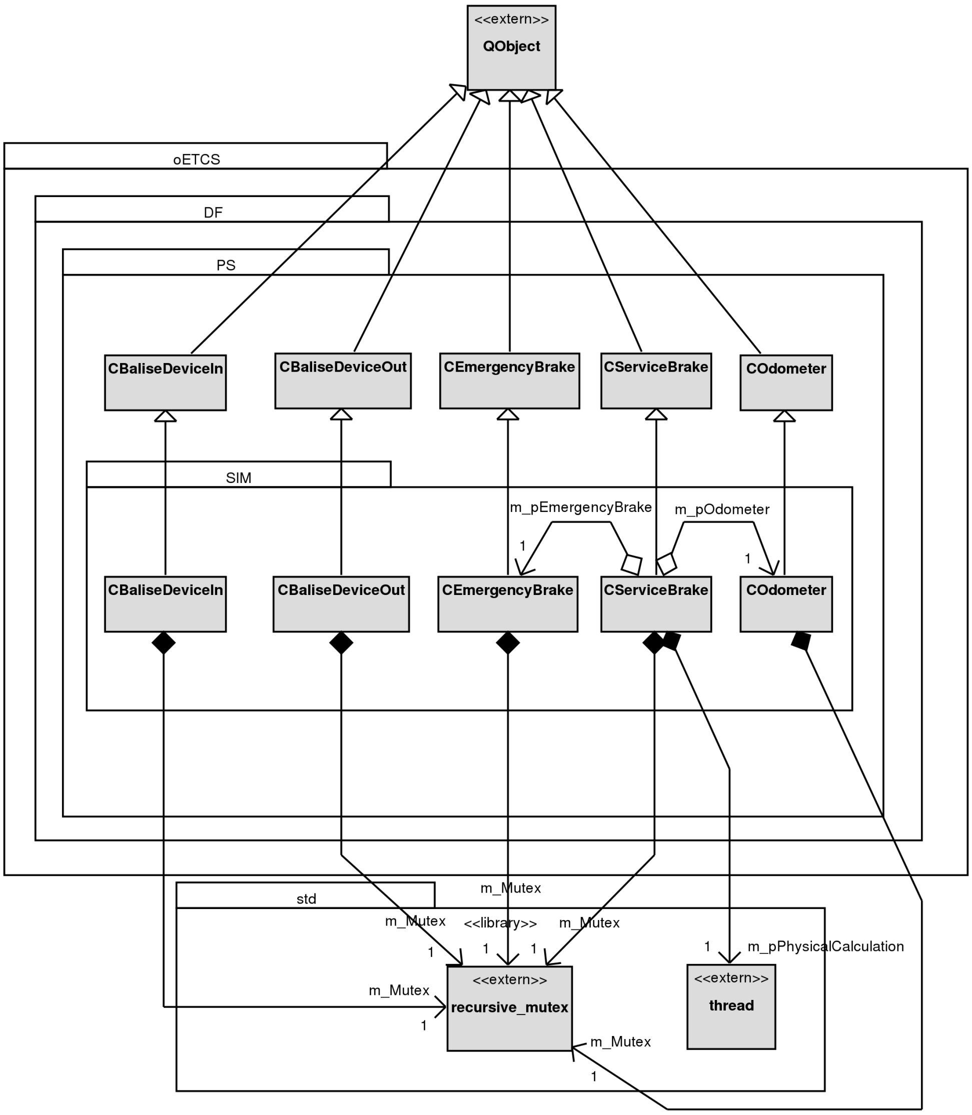


Figure 11.4.: UML class diagram of the simulative PSM

current train speed and the absolute distance. Their simulative calculation will be explained in Subsection 11.2.3, which illustrates details about the implementation.

According to Req.15, the access to the DMI is realised by the definition of another D-Bus interface that is used in the concrete DMI observer CDMIQWidget from Subsection 8.3.2. Since the resulting generated D-Bus adaptor is simply used as composition, an additional UML diagram is omitted. Details about the interface and the adaptor can be found alongside with all other openETCS domain framework description in Appendix D. The access to the DMI from the simulation is in conflict with the general design principle that only accesses by method calls from the PIM to the PSM are allowed. Therefore, the DMI adaptor should only be available for simulation purposes and only be used if required. This is done by the openETCS C++ generator by setting a Boolean property of the gEVCStateMachine root graph. The full documentation of the graph properties can be found in Section B.1.

11.2.2. Deployment Design

The deployment of the PSM for the simulation is shown in Figure 11.5, which is an extension of the PSM execution environment in Figure 8.19. The new artefacts are described below:

Simulation Holds all the PSM implementations needed for providing simulative hardware devices.

WrapperFunctions Provide a C-API for accessing objects of classes of the Simulation artefact because the RT-Tester module used for code generation currently only supports the generation of C source code [72].

SimulationModel Holds the simulation source code generated from the simulation model.

DMIDBusInterface Provides the generated sources for the D-Bus proxy of the additional adaptor for the DMI / CDMIQWidget class.

DMIDBusAdaptor Includes the generated sources for the D-Bus adaptor for the DMI respectively the class CDMIQWidget.

SimulationModel.xmi Exported UML simulation model as XMI file.

DMI.xml Is the D-Bus interface specification for the required DMI adaptor (Req.15).

Details about the simulation model itself and the corresponding code generation can be found in Section 11.3 and Section 11.4.

The libopenETCSPSMSIM component combines the static source code for the simulation environment as library and is imported by the Simulation executable binary.

The manifestation of the libopenETCSPIM component is not complete in Figure 11.5 and only represents the extensions compared to Figure 8.19. Since the usage of the DMI D-Bus adaptor is not defined before the generation of the GeneratedInstantiations artefact (Figure 8.19), the DMIDBusAdaptor source artefact must always be a part of the libopenETCSPIM library component.

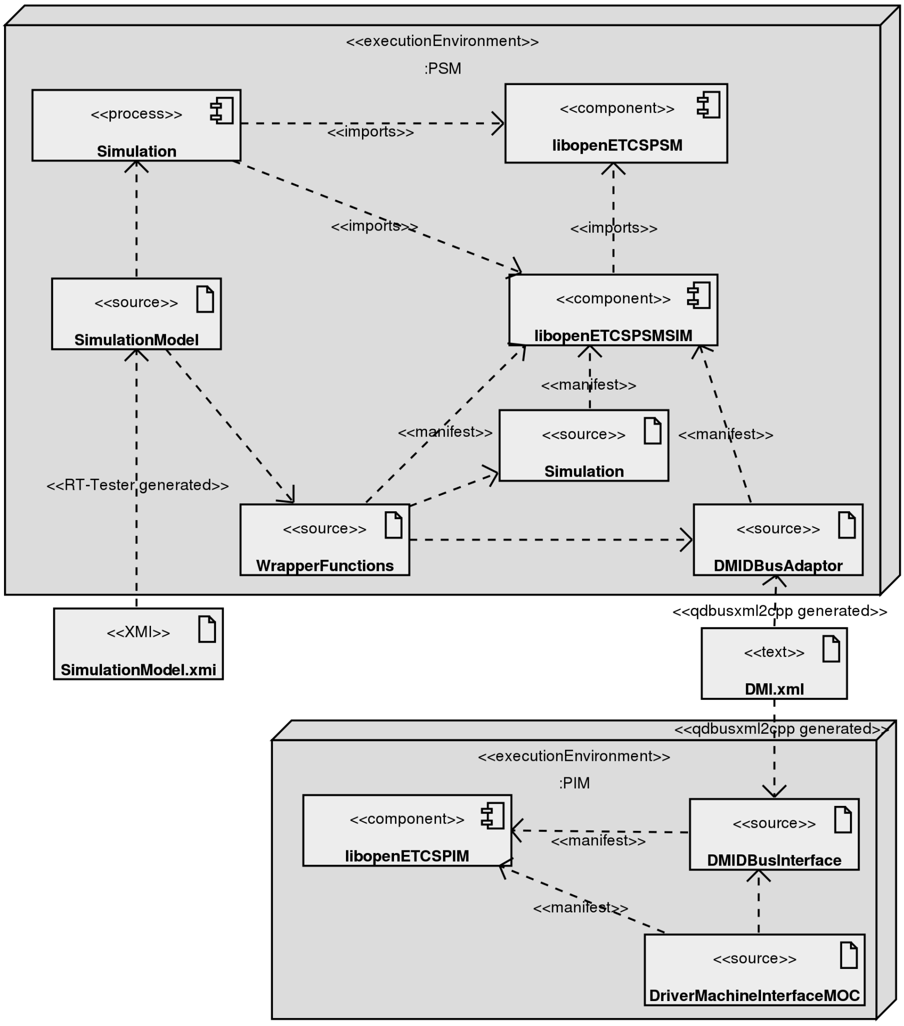


Figure 11.5.: UML deployment diagram of the simulative PSM

11.2.3. Implementation

Train Kinematics As described in the preceding subsections, the main task for the implementation of the simulative PSM is the realisation of the train kinematics, which means the current train speed $v(t)$ and the absolute distance $d(t)$. The latter one is calculated by the simple differential equation [93, p. 15]

$$d(t) = \int_0^\infty v(t)dt \tag{11.1}$$

Since the modelled and implemented system is a sample system (see Section 7.6), this kinematic equation can be expressed with (7.1) as

$$d_k = \sum_{i=0}^{k} v_i T_s \tag{11.2}$$

The speed v_k can only be directly set from the simulation or by the computation of physical model for the brake systems. The computation bases on the following general differential equation [93, p. 15]:

$$v(t) = v_0 + \int_0^\infty a(t)dt \tag{11.3}$$

$a(t)$ is the acceleration or rather the deceleration ($a(t) < 0$) and $v_0 = v(t = 0)$ the initial velocity. Accordingly, expressed as difference equation:

$$v_k = v_0 + \sum_{i=0}^{k} a_k T_s \tag{11.4}$$

The equation system of (11.2) and (11.4) build the train kinematics and are implemented in the physical calculation thread in the CServiceBrake class. Access to the velocity and distance is enabled by the `n_pOdometer` aggregation (in Figure 11.4), and `m_pEmergency` is used to determine the activation flag of the simulative emergency brake system.

The following linear and ideal[2] equation is used to calculate the current deceleration a_k by the service brake intensity s_k and the emergency brake activation e_k:

$$a_k = -fg\mu i_k \tag{11.5}$$

The variables are defined as follows:

$$\begin{aligned}
f &= 0.25 = \text{const} && (11.6)\\
g &= 9.81\frac{\text{m}}{\text{s}^2} = \text{const} && (11.7)\\
\mu &\in \,]0;1] && (11.8)\\
s_k &\in [0;1] && (11.9)\\
e_k &\in \{0;1\} && (11.10)\\
i_k &= \begin{cases} s_k & e_k = 0 \\ 1.2 & e_k = 1 \end{cases} && (11.11)
\end{aligned}$$

[2] Ideal because it is assumed that the mass m influences the friction and deceleration force the same and is accordingly not part of the equation.

f is the maximal friction coefficient [93, pp. 40ff] for metal on metal (track↔wheel), g represents the gravitation acceleration, and μ the adhesion between train and wheels.

The adhesion μ depends on the condition of the track, like wetness caused by rain, and may decrease the maximal possible deceleration: $\mu = 1 \Rightarrow -a_k = \max$
Also, the adhesion is used in the openETCS model in Figure 10.8 and Figure 10.17.

A direct setting of a new speed v' by the simulation environment always means a reset of time $t = 0 \Rightarrow k = 0$ and the initial speed $v_0 = v'$ to have correct kinematics calculations.

It should be emphasised that a more realistic mathematical model is not needed for the simulation because this focuses on the ATP functionality and not the exactly physical behaviour.

Communications In contrast to the devices for train kinematics, the simulative implementation of the communication devices CBaliseDeviceIn and CBaliseDeviceOut is much simpler. Since CBaliseDeviceIn simulates a balise device for reading telegrams from track to train, new telegrams can be set by the simulation, which are then distributed via the D-Bus signalling mechanism [31] to all connected CBaliseDeviceIn objects in the PIM (see Section 8.5 and Figure 8.16).

CBaliseDeviceOut accepts all telegrams sent via the corresponding PIM objects and stores them. These can be retrieved by the simulation if required.

11.3. Simulation Model

Corresponding to the simulation concept in Figure 11.1, the simulation model defines how the PSM of the EVC binary is stimulated by setting the train speed or by sending balise telegrams, and which resulting outputs values of the position and the brake systems are expected from the EVC PSM. Since a driver interaction is always required, the same is modelled for the stimulation via driver inputs to the DMI and for the checking of DMI outputs.

Although the ETCS specification also provides an extensive set of tests sequences in [82], those were not used to model the simulation and expected system behaviour because the tested system is only a sub-subset of the SRS, and therefore most tests are not directly applicable. The simulation model is directly derived from the ETCS SRS (Subset-026) [83] according to the chosen sub-subset for the openETCS case study in Section 7.1 to avoid the extensive work for adaptations and selecting appropriate test sequences from the Subset-076-6-3 [82]. Since the purpose of the simulation is to evaluate the openETCS case study as a proof of concept, this is no limitation to its validity.

The test environment [72] represents the simulation and defines executable state machines and signals between them. Figure 11.6 introduces the high-level model as UML class diagram. Each class represents an independent state machine while attributes of interfaces can be used as signals between the state machines for interaction. Generally, the simulation model is divided in three state machines:

CEVC State machine for modelling a virtual track, which the train is moving on. Additionally, it checks the behaviour of the EVC for correctness.

CDMI State machine that describes the deterministic behaviour of a driver.

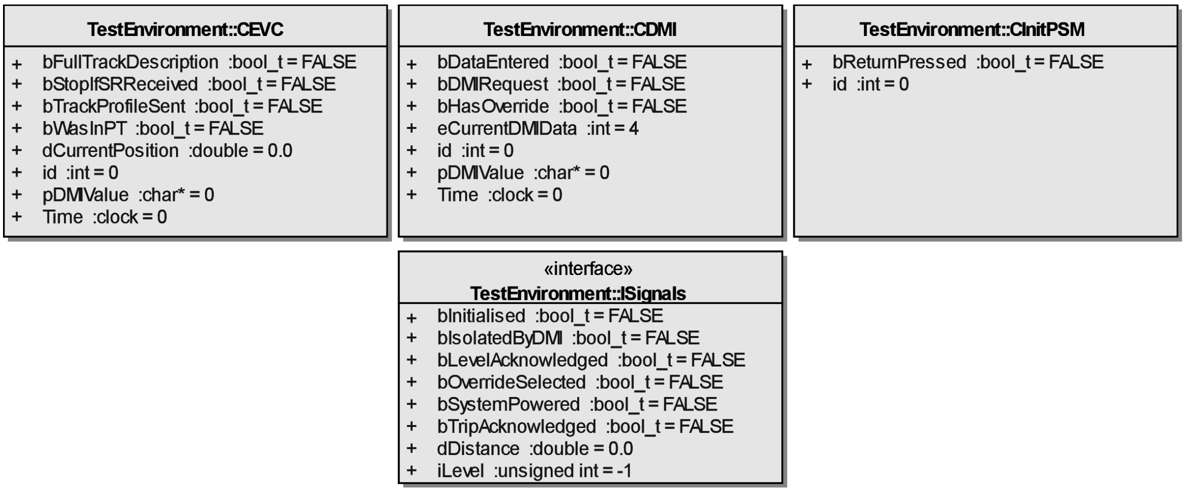

TestEnvironment::CEVC
+ bFullTrackDescription :bool_t = FALSE
+ bStopIfSRReceived :bool_t = FALSE
+ bTrackProfileSent :bool_t = FALSE
+ bWasInPT :bool_t = FALSE
+ dCurrentPosition :double = 0.0
+ id :int = 0
+ pDMIValue :char* = 0
+ Time :clock = 0
TestEnvironment::CDMI
+ bDataEntered :bool_t = FALSE
+ bDMIRequest :bool_t = FALSE
+ bHasOverride :bool_t = FALSE
+ eCurrentDMIData :int = 4
+ id :int = 0
+ pDMIValue :char* = 0
+ Time :clock = 0
TestEnvironment::CInitPSM
+ bReturnPressed :bool_t = FALSE
+ id :int = 0
«interface»
TestEnvironment::ISignals
+ bInitialised :bool_t = FALSE
+ bIsolatedByDMI :bool_t = FALSE
+ bLevelAcknowledged :bool_t = FALSE
+ bOverrideSelected :bool_t = FALSE
+ bSystemPowered :bool_t = FALSE
+ bTripAcknowledged :bool_t = FALSE
+ dDistance :double = 0.0
+ iLevel :unsigned int = -1

Figure 11.6.: UML class diagram of the test environment model

CInitPSM Simple state machine for initialising the PSM in order that the EVC binary can be connected to the required D-Bus adaptors (see Section 8.5). Also, it creates the proxy to the DMI for the CDMI machine.

The interface ISignals provides global signals between the state machine classes and is used in their concrete models, which will be introduced in detail in a subsection each. The complete simulation model can also be found in Section H.3.

11.3.1. CInitPSM – Initialisation Model

The model of the state machine for the PSM initialisation is quite simple and only consists of sequence of states, which is shown in Figure 11.7. The state "Waiting_for_start_of_PSM" creates all simulative adaptors for hardware interfaces and waits for user input because the generated EVC executable binary has to be started manually to connect to those adaptors via proxies.

Afterwards, "Initialising_DMI" creates a proxy to access the DMI in the PSM part of the EVC binary. The Boolean bInitialised signal is set to true to inform the other state machines that all preconditions for the simulation process have been met.

Currently, the RT-Tester does not support the usage of final state objects, which are part of the UML standard for state machines [64]. Therefore, the empty and dead state "Final" is used to model that after the initialisation nothing more is executed by the CInitPSM state machine.

Most states in all state machines generate an output by the `printf()` function, which can be used to generate the simulation traces described in Section 11.1.

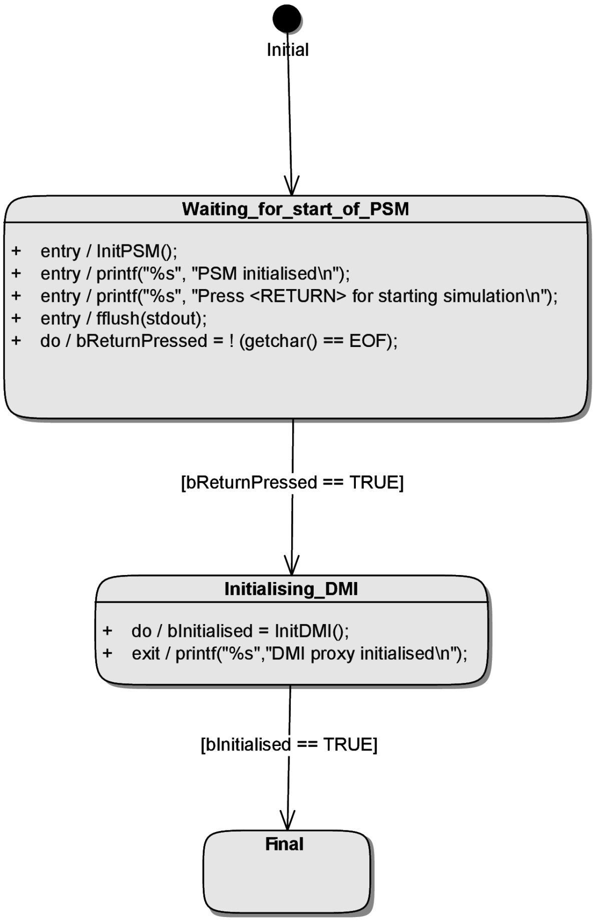

Initial
Waiting_for_start_of_PSM
+ entry / InitPSM();
+ entry / printf("%s", "PSM initialised\n");
+ entry / printf("%s", "Press <RETURN> for starting simulation\n");
+ entry / fflush(stdout);
+ do / bReturnPressed = ! (getchar() == EOF);
[bReturnPressed == TRUE]
Initialising_DMI
+ do / bInitialised = InitDMI();
+ exit / printf("%s","DMI proxy initialised\n");
[bInitialised == TRUE]
Final

Figure 11.7.: UML state machine diagram of the PSM initialisation

11.3.2. CDMI – Driver Model

The model of the driver's behaviour is defined by several decomposition of different states. The parent state machine is displayed in Figure 11.8. On the top level, only two states in a non-terminating state machine are provided:

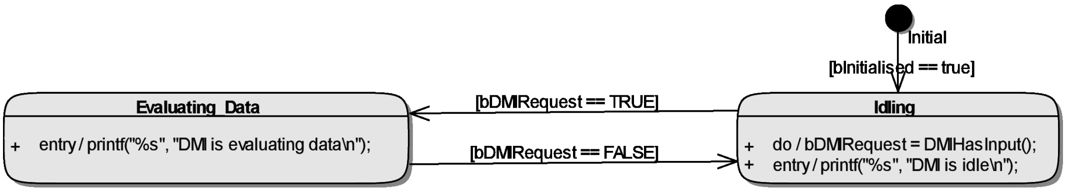

Initial
[bInitialised == true]
Evaluating_Data
+ entry / printf("%s", "DMI is evaluating data\n");
[bDMIRequest == TRUE]
[bDMIRequest == FALSE]
Idling
+ do / bDMIRequest = DMIHasInput();
+ entry / printf("%s", "DMI is idle\n");

Figure 11.8.: UML state machine diagram of the driver's behaviour

Idling No data is currently available in the DMI. Nothing can be entered by the (virtual) driver.

Evaluating_Data The DMI requires data to be entered. The behaviour in this state is refined by a decomposition to a further sub-state machine, which is introduced in a separated section.

11.3.2.1. Evaluating_Data State Machine

The corresponding graph of the UML state machine is shown in Figure 11.9. Its states are explained as follows:

Check_Type_of_Data This initial state checks the type of the available inputs in the DMI by the `EvaluateDMIInputs()` function of the simulative PSM C-API (see Subsection 11.2.2). The full documentation and source code for all those C-API functions is located in the appendix in Section H.2.

Furthermore, it is checked if the system / EVC was initially powered by the "Powering_System" state or the DMI has at least one input[3]. Otherwise, the execution of this state is not meaningful, and the simulation detects an error, which is is evaluated by `@rttAssert`-statement [72]. If the Boolean expression in this statement evaluates to false, the complete simulation is stopped and the error is logged.

Powering_System During an error-free simulation, this state is only executed once, at the beginning of the simulation. It

[3] return value of the `DMIHasInput()` function

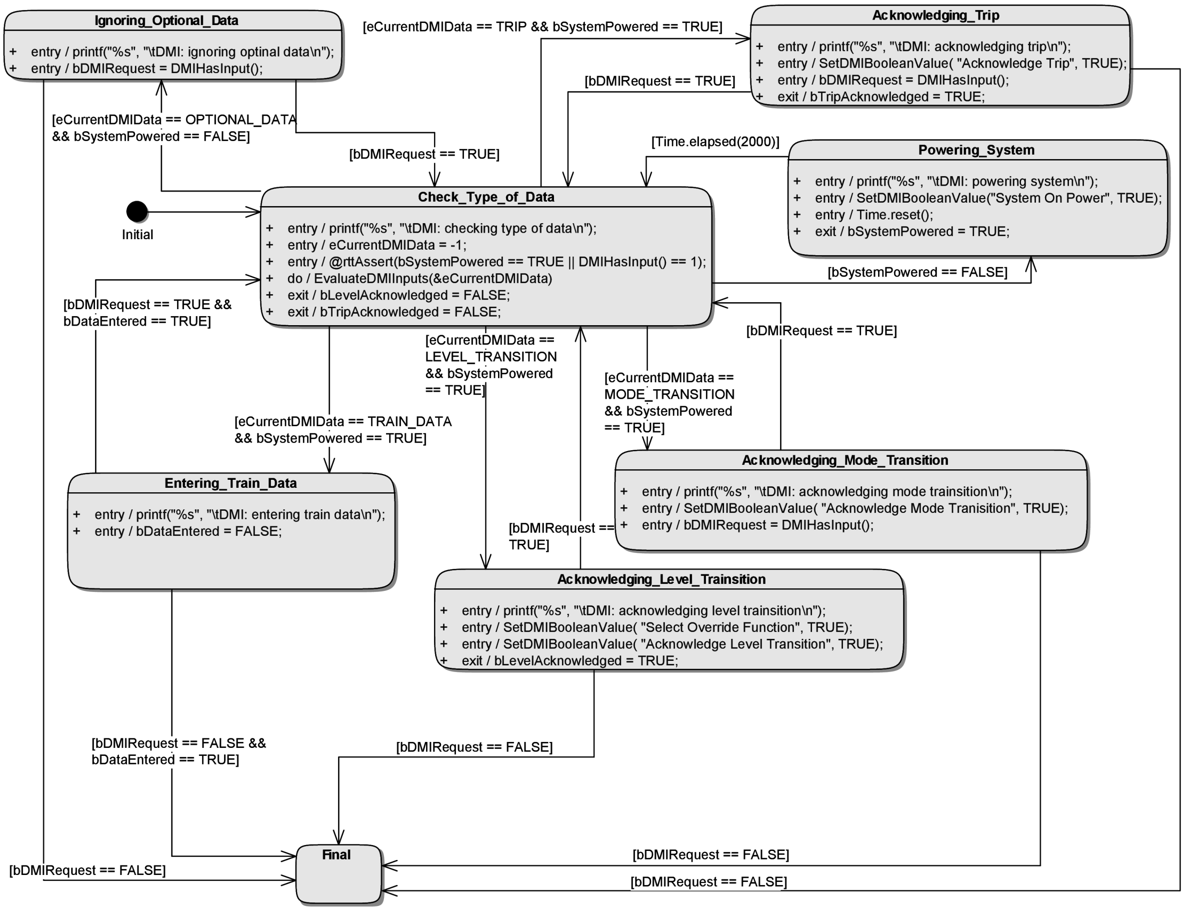

Ignoring_Optional_Data
+ entry / printf("%s", "\tDMI: ignoring optinal data\n");
+ entry / bDMIRequest = DMIHasInput();
[eCurrentDMIData == TRIP && bSystemPowered == TRUE]
Acknowledging_Trip
+ entry / printf("%s", "\tDMI: acknowledging trip\n");
+ entry / SetDMIBooleanValue("Acknowledge Trip", TRUE);
+ entry / bDMIRequest = DMIHasInput();
+ exit / bTripAcknowledged = TRUE;
[bDMIRequest == TRUE]
[eCurrentDMIData == OPTIONAL_DATA
&& bSystemPowered == FALSE]
[bDMIRequest == TRUE]
[Time.elapsed(2000)]
Powering_System
+ entry / printf("%s", "\tDMI: powering system\n");
+ entry / SetDMIBooleanValue("System On Power", TRUE);
+ entry / Time.reset();
+ exit / bSystemPowered = TRUE;
Initial
Check_Type_of_Data
+ entry / printf("%s", "\tDMI: checking type of data\n");
+ entry / eCurrentDMIData = -1;
+ entry / @rttAssert(bSystemPowered == TRUE || DMIHasInput() == 1);
+ do / EvaluateDMIInputs(&eCurrentDMIData)
+ exit / bLevelAcknowledged = FALSE;
+ exit / bTripAcknowledged = FALSE;
[bSystemPowered == FALSE]
[bDMIRequest == TRUE &&
bDataEntered == TRUE]
[bDMIRequest == TRUE]
[eCurrentDMIData ==
LEVEL_TRANSITION
&& bSystemPowered
== TRUE]
[eCurrentDMIData ==
MODE_TRANSITION
&& bSystemPowered
== TRUE]
[eCurrentDMIData == TRAIN_DATA
&& bSystemPowered == TRUE]
Acknowledging_Mode_Transition
+ entry / printf("%s", "\tDMI: acknowledging mode trainsition\n");
+ entry / SetDMIBooleanValue("Acknowledge Mode Tranisition", TRUE);
+ entry / bDMIRequest = DMIHasInput();
Entering_Train_Data
+ entry / printf("%s", "\tDMI: entering train data\n");
+ entry / bDataEntered = FALSE;
[bDMIRequest ==
TRUE]
Acknowledging_Level_Trainsition
+ entry / printf("%s", "\tDMI: acknowledging level trainsition\n");
+ entry / SetDMIBooleanValue("Select Override Function", TRUE);
+ entry / SetDMIBooleanValue("Acknowledge Level Transition", TRUE);
+ exit / bLevelAcknowledged = TRUE;
[bDMIRequest == FALSE &&
bDataEntered == TRUE]
[bDMIRequest == FALSE]
Final
[bDMIRequest == FALSE]
[bDMIRequest == FALSE]
[bDMIRequest == FALSE]

Figure 11.9.: UML state machine diagram for evaluating the data in the DMI

sets the Boolean input "System on Power" (see Subsection 10.2.1) by the `SetDMIBooleanValue()` function. The attribute `Time` is used to determine the elapsed time of 2000ms = 2s as a guard condition.

Entering_Train_Data Similar to "Powering_System", this state is normally executed once to enter the static train data, like train length and static top-speed, in the ETCS Mode Stand By (modelled in Subsection 10.2.2).

Ignoring_Optional_Data Any visible input field in the DMI that is not directly needed to be filled with data for the continuation of the simulation is marked as optional and is in the simulation simply ignored. An example for optional data is the activation of the override function in most ETCS Modes [88, p. 42].

Acknowledging_Trip This state is only used in the case that the EVC is in the Mode Trip (described in Subsection 10.2.6) and this has to be acknowledged by the driver to proceed.

Acknowledging_Level_Transition A transition to a new ETCS Application Level that is propagated to the EVC has always to be acknowledged by the driver, which is executed in this state.

Acknowledging_Mode_Transition Similar to "Acknowledging_Level_Transition", it can be necessary to acknowledge an explicit Mode switch.

Since some states are also refined by a sub-state machine, those are introduced in the following paragraphs.

11.3.2.1.1. Entering_Train_Data The static train data is entered in a simple sequence, which model is sketched in Figure 11.10. On the EVC side, this corresponds to the the "Start of Mission in Stand By" model in Subsection 10.2.2 and Figure 10.5.

Each state in the simulation model sets a certain input value of the DMI. All transitions between the states are activated after 1.5s since a real driver neither can enter data in parallel nor without pause. The data entered in each state is roughly explained in the following:

Entering_Driver_ID The driver ID is a string for identifying the current driver, which is set as "ORA-SIM" for the simulative driver.

Entering_Train_Position The absolute train position can be set to another value than 0 if required. In the simulation, this is not necessary.

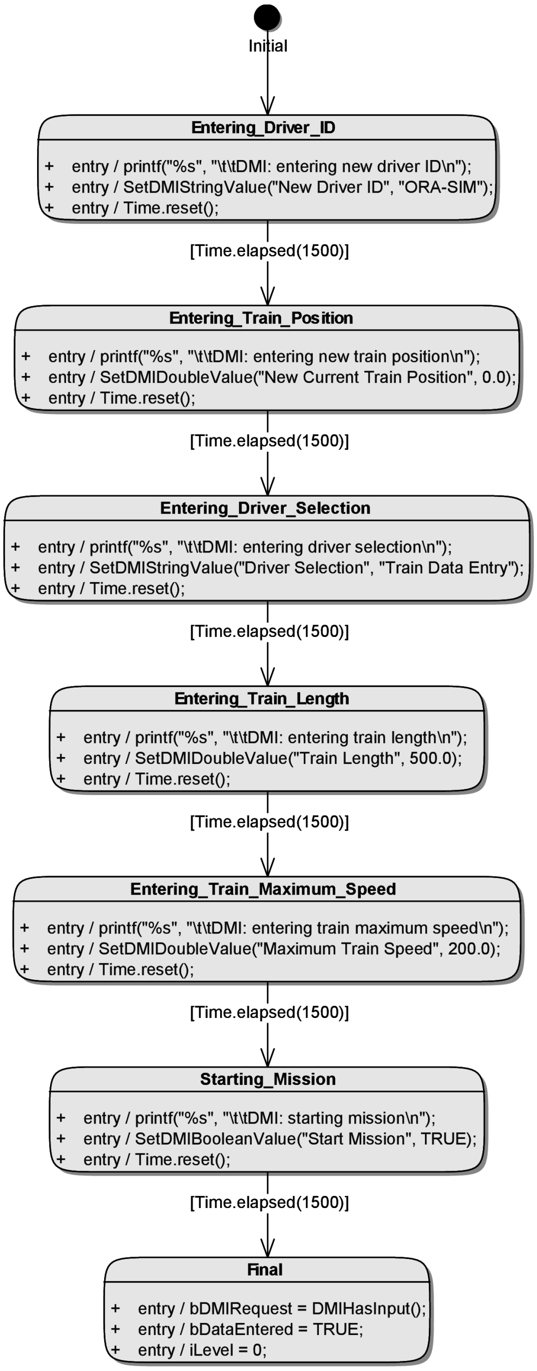

Initial
Entering_Driver_ID
+ entry / printf("%s", "\t\tDMI: entering new driver ID\n");
+ entry / SetDMIStringValue("New Driver ID", "ORA-SIM");
+ entry / Time.reset();
[Time.elapsed(1500)]
Entering_Train_Position
+ entry / printf("%s", "\t\tDMI: entering new train position\n");
+ entry / SetDMIDoubleValue("New Current Train Position", 0.0);
+ entry / Time.reset();
[Time.elapsed(1500)]
Entering_Driver_Selection
+ entry / printf("%s", "\t\tDMI: entering driver selection\n");
+ entry / SetDMIStringValue("Driver Selection", "Train Data Entry");
+ entry / Time.reset();
[Time.elapsed(1500)]
Entering_Train_Length
+ entry / printf("%s", "\t\tDMI: entering train length\n");
+ entry / SetDMIDoubleValue("Train Length", 500.0);
+ entry / Time.reset();
[Time.elapsed(1500)]
Entering_Train_Maximum_Speed
+ entry / printf("%s", "\t\tDMI: entering train maximum speed\n");
+ entry / SetDMIDoubleValue("Maximum Train Speed", 200.0);
+ entry / Time.reset();
[Time.elapsed(1500)]
Starting_Mission
+ entry / printf("%s", "\t\tDMI: starting mission\n");
+ entry / SetDMIBooleanValue("Start Mission", TRUE);
+ entry / Time.reset();
[Time.elapsed(1500)]
Final
+ entry / bDMIRequest = DMIHasInput();
+ entry / bDataEntered = TRUE;
+ entry / iLevel = 0;

Figure 11.10.: UML state machine diagram of the “Entering_Train_Data” state

Entering_Driver_Selection The driver can choose between overriding the input of further static train data or to enter all train data. In the simulation, all data should be entered.

Entering_Train_Length The static train length is entered as 500m

Entering_Train_Maximum_Speed The static maximum velocity of the train is entered. For the simulation, $v_{\max} = 200\frac{\text{km}}{\text{h}}$ is used, which will be also referred to in further state machines.

Starting_Mission The start of mission is activated.

11.3.2.1.2. Acknowledging_Level_Transition In the simulation, any transition from Application Level 0 to Level 1 should be done via the override function in Figure 10.7 and not by a valid MA. Accordingly, if the current Application Level is 0 (`iLevel == 0`), the override function in the DMI should be activated first. This is modelled in Figure 11.11 with the following states:

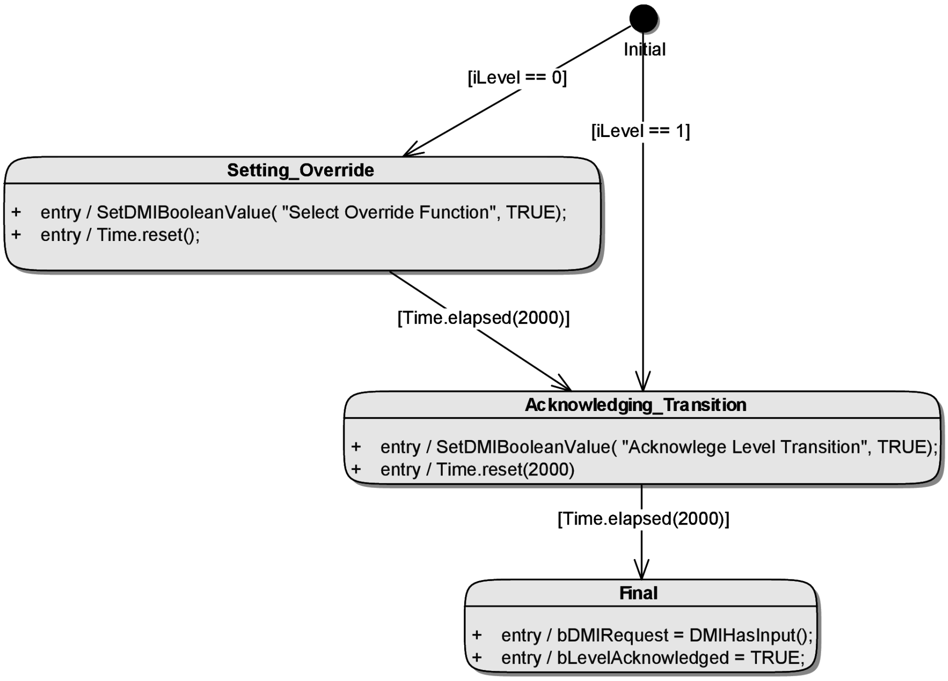

Initial
[iLevel == 0]
[iLevel == 1]
Setting_Override
+ entry / SetDMIBooleanValue("Select Override Function", TRUE);
+ entry / Time.reset();
[Time.elapsed(2000)]
Acknowledging_Transition
+ entry / SetDMIBooleanValue("Acknowlege Level Transition", TRUE);
+ entry / Time.reset(2000)
[Time.elapsed(2000)]
Final
+ entry / bDMIRequest = DMIHasInput();
+ entry / bLevelAcknowledged = TRUE;

Figure 11.11.: UML state machine diagram of the “Acknowledging_Level_Transition” state

Setting_Override Activates the override function in the DMI.

Acknowledging_Transition Acknowledges the Level transition in the DMI.

11.3.3. CEVC – EVC and Virtual Track Model

In contrast to the two preceding models, the CEVC state machine defines the virtual track and also checks for required conditions for the EVC to detect errors during the simulation execution. This means for the model that not only values are set in the simulative PSM hardware devices, but also the `@rttAssert`-statement is used to detect deviations of the EVC from the expected behaviour. On the top level of this state machine, all states correspond to a certain ETCS Mode while the virtual track is modelled in most states as sub-state machines. The correctness of the EVC behaviour is tested on the top level and as well in the decompositions. Obviously, not all ETCS Modes and possible transitions of the openETCS case study model in Chapter 10 were used in the EVC simulation model. This was done to keep a model which complexity and size is still well interpretable in this document. Furthermore, the simulation in the openETCS case study should primary demonstrate the applicability of the approach of a completely model-driven tool chain. A small simulation model does not oppose this conclusion because it can be extended without modifications of any openETCS DSL instance, like meta model, generator, and domain framework.

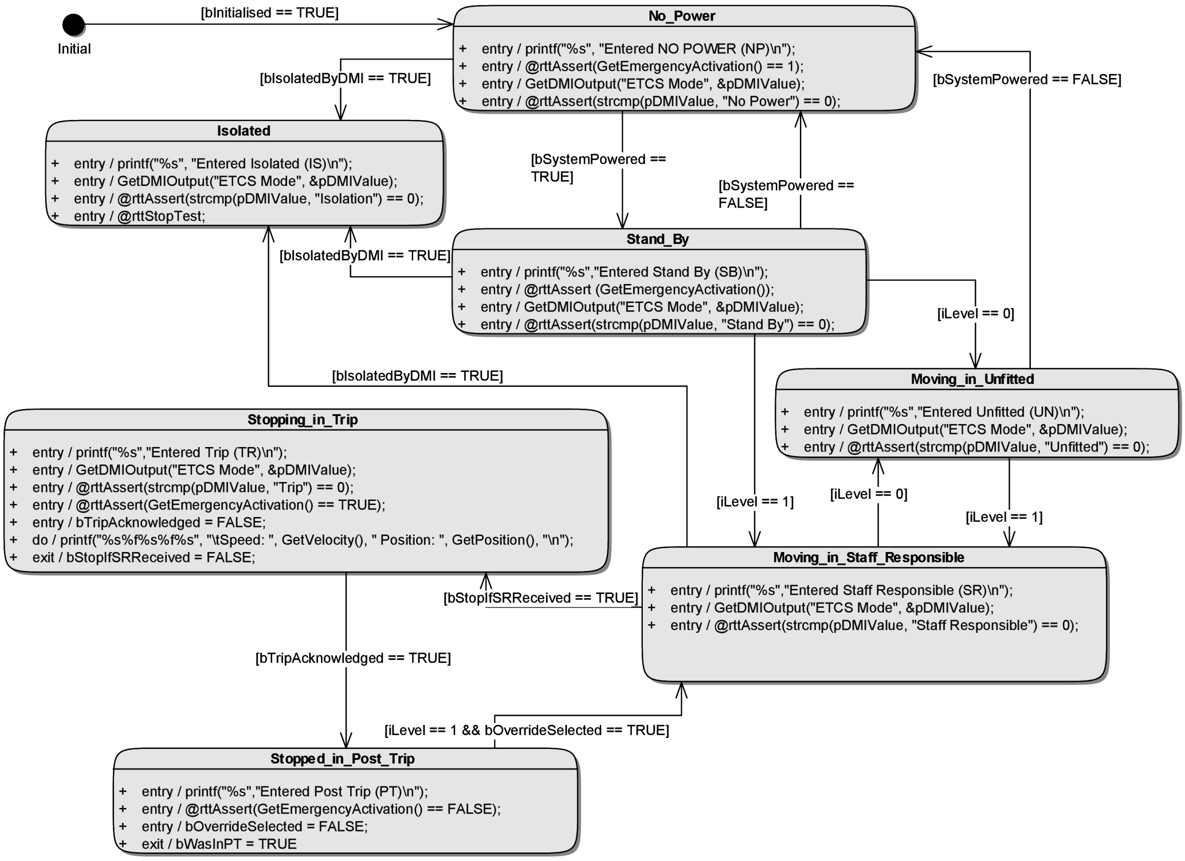

Initial
[bInitialised == TRUE]
No_Power
+ entry / printf("%s", "Entered NO POWER (NP)\n");
+ entry / @rttAssert(GetEmergencyActivation() == 1);
+ entry / GetDMIOutput("ETCS Mode", &pDMIValue);
+ entry / @rttAssert(strcmp(pDMIValue, "No Power") == 0);
[bIsolatedByDMI == TRUE]
[bSystemPowered == FALSE]
Isolated
+ entry / printf("%s", "Entered Isolated (IS)\n");
+ entry / GetDMIOutput("ETCS Mode", &pDMIValue);
+ entry / @rttAssert(strcmp(pDMIValue, "Isolation") == 0);
+ entry / @rttStopTest;
[bSystemPowered == TRUE]
[bSystemPowered == FALSE]
[bIsolatedByDMI == TRUE]
Stand_By
+ entry / printf("%s","Entered Stand By (SB)\n");
+ entry / @rttAssert (GetEmergencyActivation());
+ entry / GetDMIOutput("ETCS Mode", &pDMIValue);
+ entry / @rttAssert(strcmp(pDMIValue, "Stand By") == 0);
[iLevel == 0]
[bIsolatedByDMI == TRUE]
Moving_in_Unfitted
+ entry / printf("%s","Entered Unfitted (UN)\n");
+ entry / GetDMIOutput("ETCS Mode", &pDMIValue);
+ entry / @rttAssert(strcmp(pDMIValue, "Unfitted") == 0);
Stopping_in_Trip
+ entry / printf("%s","Entered Trip (TR)\n");
+ entry / GetDMIOutput("ETCS Mode", &pDMIValue);
+ entry / @rttAssert(strcmp(pDMIValue, "Trip") == 0);
+ entry / @rttAssert(GetEmergencyActivation() == TRUE);
+ entry / bTripAcknowledged = FALSE;
+ do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");
+ exit / bStopIfSRReceived = FALSE;
[iLevel == 1]
[iLevel == 0]
[iLevel == 1]
Moving_in_Staff_Responsible
+ entry / printf("%s","Entered Staff Responsible (SR)\n");
+ entry / GetDMIOutput("ETCS Mode", &pDMIValue);
+ entry / @rttAssert(strcmp(pDMIValue, "Staff Responsible") == 0);
[bStopIfSRReceived == TRUE]
[bTripAcknowledged == TRUE]
[iLevel == 1 && bOverrideSelected == TRUE]
Stopped_in_Post_Trip
+ entry / printf("%s","Entered Post Trip (PT)\n");
+ entry / @rttAssert(GetEmergencyActivation() == FALSE);
+ entry / bOverrideSelected = FALSE;
+ exit / bWasInPT = TRUE

Figure 11.12.: UML state machine diagram of the EVC behaviour

The states of the EVC simulation are introduced below:

No_Power According to the ETCS SRS and the corresponding openETCS model in Subsection 10.2.1, this state only checks if the emergency brakes are activated and if the "ETCS Mode" DMI output field holds the string "No Power". Since no movement can be simulated in this state, transitions to other states can only be done by signals (in Figure 11.6) from the CDMI state machine. For example, the powering of the system / EVC, which is indicated by the `bSystemPowered` variable.

Stand_By This state does not differ much from the (normally) preceding "No_Power" state because again it is checked if the emergency brake is activated and the "ETCS Mode" DMI output field holds the name of the current Mode. Also, the corresponding manipulation of the DMI is done by the CDMI machine from Subsection 11.3.2 by entering the static train data.

Moving_in_Unfitted The train movement starts in the ETCS Mode Unfitted if the Application Level is 0, which is the default case in the simulation. Since various ATP functionalities are tested in this state, which are modelled in a sub-state machine, this state only has an assertion for the correct "ETCS Mode" DMI output field as entry-action [64]. Further assertions are defined in its decomposition.

Moving_in_Staff_Responsible In this state, similar to "Moving_in_Unfitted", only the correct value of the "ETCS Mode" DMI is checked. The concrete functionality and the corresponding correct behaviour is tested in a sub-state machine.

Stopping_in_Trip According to the model in Subsection 10.2.6, the emergency brake must be activated in this Mode and the correct ETCS Mode string must be displayed in the DMI.

Stopped_in_Post_Trip This state can only be executed directly after "Stopping_in_Trip". In contrast to the preceding state, the emergency brake must not be used here, which is explicitly tested by an assert statement. Again, the ATP functionality is tested in a sub-state machine.

Isolated Since ETCS does not provide any ATP functionality in Mode Isolated, only the correct value of the "ETCS Mode" DMI output field is tested. Furthermore, the simulation is stopped in this state by the `@rttStopTest` statement because there is no transition to any other Mode or state from here. Since the failing of any assertion executed before would lead to a stop of the simulation, reaching this stop statement always means a successful execution of the simulation.

The CEVC state machine defines several possible transitions between its states, which are often activated by signals influenced by the CDMI state machine or by sub-state machines of CEVC. Which transitions are used in the current version of the simulation model, can be easily identified in the simulation traces presented in Section 11.5.

Each sub-state machine of CEVC is explained in detail in the following subsections.

11.3.3.1. Moving_in_Unfitted

In the ETCS Mode Unfitted, the ATP functionality is mainly limited to the supervision of train speed in respect to its static top speed (modelled in Subsection 10.2.3), which is entered by the simulated driver. Thus, the simulation model for the movement in this Mode consists of a simple sequence of states, which is displayed in Figure 11.13. Each state is explained in detail in the following:

Moving_with_allowed_speed The train is moving with the speed $v = 100\frac{\text{km}}{\text{h}}$, which is lower than its static top speed $v_{\text{max}} = 200\frac{\text{km}}{\text{h}}$ (see Figure 11.10): $v < v_{\text{max}}$

Accordingly, service or emergency brake may not be applied, which is permanently checked by corresponding do-actions [64].

Moving_with_warning_speed The train speed $v = 1.025v_{\text{max}} = 205\frac{\text{km}}{\text{h}}$ exceeds the static top speed v_{max} but not the hard limit for brake intervention $v_{\text{max,int}} = 1.05v_{\text{max}} = 210\frac{\text{km}}{\text{h}}$: $v_{\text{max}} \leq v \leq v_{\text{max,int}}$

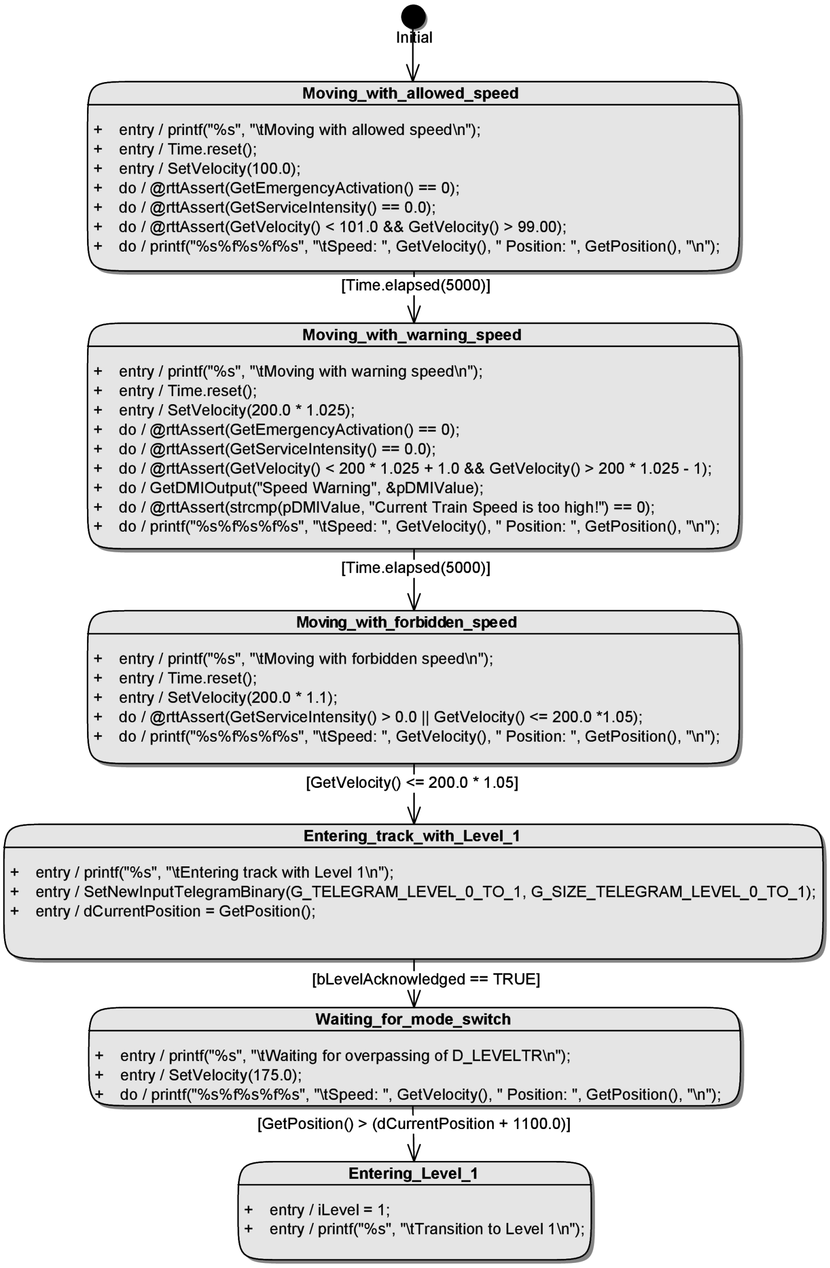
Details can be found in the implementation and documentation of the CBrakingToTargetSpeed class in Appendix D.

In this state, still no brakes may be applied, but a warning in the "Speed Warning" DMI output field must be displayed, which is checked by an assert statement.

Moving_with_forbidden_speed The train speed of $v = 1.1v_{\text{max}} = 220\frac{\text{km}}{\text{h}}$ exceeds the hard limit $v_{\text{max,int}}$: $v > v_{\text{max,int}}$

Thus, the service brake s_k must be applied until the speed v is smaller than the hard limit $v_{\text{max,int}}$: $s_k > 0 \vee v \leq v_{\text{max,int}}$

Entering_track_with_Level_1 A switching of the ETCS Application Level to 1 is propagated to the EVC by a balise telegram that holds a "Level Transition Order" packet in Subsection 10.4.3. The binary



```
Moving_with_allowed_speed
+   entry / printf("%s", "\tMoving with allowed speed\n");
+   entry / Time.reset();
+   entry / SetVelocity(100.0);
+   do / @rttAssert(GetEmergencyActivation() == 0);
+   do / @rttAssert(GetServiceIntensity() == 0.0);
+   do / @rttAssert(GetVelocity() < 101.0 && GetVelocity() > 99.00);
+   do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");

[Time.elapsed(5000)]

Moving_with_warning_speed
+   entry / printf("%s", "\tMoving with warning speed\n");
+   entry / Time.reset();
+   entry / SetVelocity(200.0 * 1.025);
+   do / @rttAssert(GetEmergencyActivation() == 0);
+   do / @rttAssert(GetServiceIntensity() == 0.0);
+   do / @rttAssert(GetVelocity() < 200 * 1.025 + 1.0 && GetVelocity() > 200 * 1.025 - 1);
+   do / GetDMIOutput("Speed Warning", &pDMIValue);
+   do / @rttAssert(strcmp(pDMIValue, "Current Train Speed is too high!") == 0);
+   do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");

[Time.elapsed(5000)]

Moving_with_forbidden_speed
+   entry / printf("%s", "\tMoving with forbidden speed\n");
+   entry / Time.reset();
+   entry / SetVelocity(200.0 * 1.1);
+   do / @rttAssert(GetServiceIntensity() > 0.0 || GetVelocity() <= 200.0 *1.05);
+   do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");

[GetVelocity() <= 200.0 * 1.05]

Entering_track_with_Level_1
+   entry / printf("%s", "\tEntering track with Level 1\n");
+   entry / SetNewInputTelegramBinary(G_TELEGRAM_LEVEL_0_TO_1, G_SIZE_TELEGRAM_LEVEL_0_TO_1);
+   entry / dCurrentPosition = GetPosition();

[bLevelAcknowledged == TRUE]

Waiting_for_mode_switch
+   entry / printf("%s", "\tWaiting for overpassing of D_LEVELTR\n");
+   entry / SetVelocity(175.0);
+   do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");

[GetPosition() > (dCurrentPosition + 1100.0)]

Entering_Level_1
+   entry / iLevel = 1;
+   entry / printf("%s", "\tTransition to Level 1\n");
```

Figure 11.13.: UML state machine diagram of the “Moving_in_Unfitted” state

telegram itself consists of 121 bits and is therefore not printed here explicitly but can be found in the C-API of the simulative PSM in Section H.2 of the appendix.

Waiting_for_mode_switch After the level transition was acknowledged by CDMI state machine, the relative distance $d_{\text{trans}} = 1000\text{m}$ to the transition point must be passed. The speed is decreased below the static top speed because here the Level transition and the speed supervision should not be tested in parallel: $v = 175\frac{\text{km}}{\text{h}} < v_{\text{max}}$

Entering_Level_1 The guard to this final state requires that 1100m instead of $d_{\text{trans}} = 1000\text{m}$ were passed because the EVC is a sample system (see Section 7.6). Therefore, it cannot be ensured that the Level transition happens at exactly d_{trans} and $d_{\text{tol}} = 100\text{m}$ is used as tolerance value in the simulation.

This state itself only sets the global signal `iLevel = 1` (from Figure 11.6) to inform all state machines about the new ETCS Application Level.

11.3.3.2. Moving_in_Staff_Responsible

The simulated virtual track for the ETCS Mode Staff Responsible does not differ much from the one for Unfitted in Subsection 11.3.3.1, besides that in Staff Responsible a certain value for the top speed in this Mode is taken into account, which is normally a national value. In this simulation, it is initialised with $v_{\text{SR,max}} = 180\frac{\text{km}}{\text{h}}$. The state machine model is shown in Figure 11.14 and its states are explained below.

Moving_with_allowed_speed The train is moving with a speed $v = 100\frac{\text{km}}{\text{h}}$, which is lower than the top speed in Staff Responsible $v_{\text{SR,max}}$: $v < v_{\text{SR,max}}$

Service or emergency brake may not be applied in this state.

Moving_with_warning_speed The train speed $v = 1.025 v_{\text{SR,max}} = 184.5\frac{\text{km}}{\text{h}}$ exceeds the top speed $v_{\text{SR,max}}$ but not the hard limit for brake intervention $v_{\text{SR,max,int}} = 1.05 v_{\text{SR,max}} = 189\frac{\text{km}}{\text{h}}$: $v_{\text{SR,max}} \leq v \leq v_{\text{SR,max,int}}$

In this state still no brakes may be applied, but a warning in the “Speed Warning” DMI output field must be displayed, which is checked by an assert statement.

Moving_with_forbidden_speed The train speed of $v = 1.1 v_{\text{SR,max}} = 220\frac{\text{km}}{\text{h}}$ exceeds the hard limit $v_{\text{SR,max,int}}$: $v > v_{\text{max,int}}$

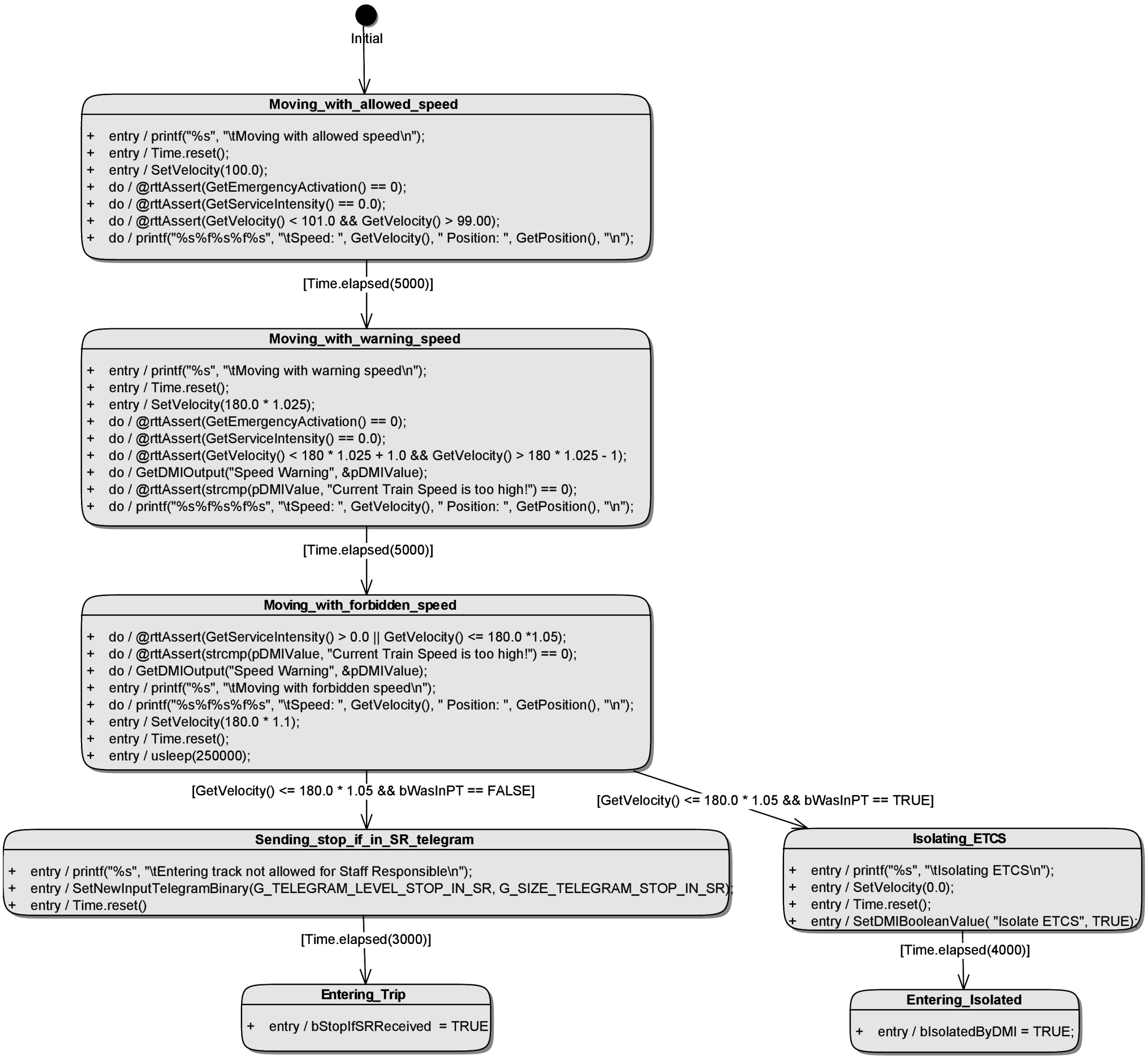

Initial
Moving_with_allowed_speed
+ entry / printf("%s", "\tMoving with allowed speed\n");
+ entry / Time.reset();
+ entry / SetVelocity(100.0);
+ do / @rttAssert(GetEmergencyActivation() == 0);
+ do / @rttAssert(GetServiceIntensity() == 0.0);
+ do / @rttAssert(GetVelocity() < 101.0 && GetVelocity() > 99.00);
+ do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");
[Time.elapsed(5000)]
Moving_with_warning_speed
+ entry / printf("%s", "\tMoving with warning speed\n");
+ entry / Time.reset();
+ entry / SetVelocity(180.0 * 1.025);
+ do / @rttAssert(GetEmergencyActivation() == 0);
+ do / @rttAssert(GetServiceIntensity() == 0.0);
+ do / @rttAssert(GetVelocity() < 180 * 1.025 + 1.0 && GetVelocity() > 180 * 1.025 - 1);
+ do / GetDMIOutput("Speed Warning", &pDMIValue);
+ do / @rttAssert(strcmp(pDMIValue, "Current Train Speed is too high!") == 0);
+ do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");
[Time.elapsed(5000)]
Moving_with_forbidden_speed
+ do / @rttAssert(GetServiceIntensity() > 0.0 || GetVelocity() <= 180.0 *1.05);
+ do / @rttAssert(strcmp(pDMIValue, "Current Train Speed is too high!") == 0);
+ do / GetDMIOutput("Speed Warning", &pDMIValue);
+ entry / printf("%s", "\tMoving with forbidden speed\n");
+ do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");
+ entry / SetVelocity(180.0 * 1.1);
+ entry / Time.reset();
+ entry / usleep(250000);
[GetVelocity() <= 180.0 * 1.05 && bWasInPT == FALSE]
[GetVelocity() <= 180.0 * 1.05 && bWasInPT == TRUE]
Sending_stop_if_in_SR_telegram
+ entry / printf("%s", "\tEntering track not allowed for Staff Responsible\n");
+ entry / SetNewInputTelegramBinary(G_TELEGRAM_LEVEL_STOP_IN_SR, G_SIZE_TELEGRAM_STOP_IN_SR);
+ entry / Time.reset()
Isolating_ETCS
+ entry / printf("%s", "\tIsolating ETCS\n");
+ entry / SetVelocity(0.0);
+ entry / Time.reset();
+ entry / SetDMIBooleanValue("Isolate ETCS", TRUE);
[Time.elapsed(3000)]
[Time.elapsed(4000)]
Entering_Trip
+ entry / bStopIfSRReceived = TRUE
Entering_Isolated
+ entry / bIsolatedByDMI = TRUE;

Figure 11.14.: UML state machine diagram of the “Moving_in_Staff_Responsible” state

Thus, the service brake s_k must be applied until the speed v is smaller than the hard limit $v_{\text{SR,max,int}}$: $s_k > 0 \lor v \leq v_{\text{SR,max,int}}$

Sending_stop_if_in_SR_telegram A "Stop if in Staff Responsible" packet from Subsection 10.4.4 is sent to the train, which should trigger a transition to the Trip Mode in the EVC to test the additional functionality of the EVC in Staff Responsible.

Entering_Trip After giving the EVC some time to react, the switch to Trip is propagated to the parent state machine by the local signal `bStopIfSRReceived = TRUE`.

Isolating_ETCS This state is only entered if the states "Stopping_in_Trip" and "Stopped_in_Post_Trip" in the parent state machine in Figure 11.12 were executed before. This is indicated by the `bWasInPT == TRUE` conjunction in the corresponding transition guard.

The train is stopped and the EVC is isolated from the train via the DMI "Isolate ETCS" input field.

Entering_Isolated All state machines are informed by the global signal `bIsolatedByDMI = TRUE` that the EVC was isolated.

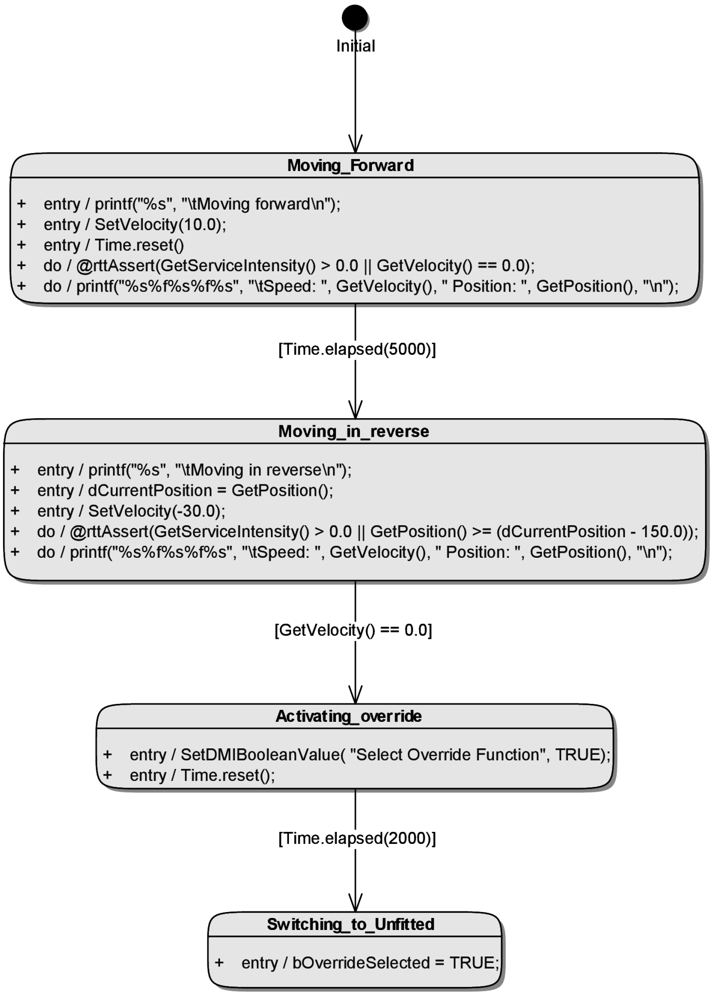
11.3.3.3. Stopped_in_Post_Trip

According to the model in Subsection 10.2.6, the ETCS Mode Post Trip (in Application Level 1) is entered after the train was completely stopped in Trip and this is acknowledged by the driver via the DMI. In Post Trip itself, the emergency brakes should not be applied and any forward movement of the train must be inhibited by the service brakes (see Subsection 10.2.7). Furthermore, the reverse movement about a constance distance[4] is allowed. The simulation model for testing these ATP functionalities is sketched in Figure 11.15. The executed simulation is explained for each state in detail in the following:

Moving_forward The train speed is set to $v = 10\frac{\text{km}}{\text{h}}$, which simulates a slow forward movement. Accordingly, the service brakes should be applied ($s_k > 0$) until the train is again fully stopped ($v \overset{!}{=} 0$). This is checked by an assert statement.

Moving_in_reverse The train speed is set to $v = -30\frac{\text{km}}{\text{h}}$, which means a slow backward movement. This is allowed until the distance $d_{\text{rev}} = 150\text{m}$ is overpassed. Then, the backward movement should be also inhibited by the service brakes until the train fully stops.

[4] national value



```
+   entry / printf("%s", "\tMoving forward\n");
+   entry / SetVelocity(10.0);
+   entry / Time.reset()
+   do / @rttAssert(GetServiceIntensity() > 0.0 || GetVelocity() == 0.0);
+   do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");
```

```
+   entry / printf("%s", "\tMoving in reverse\n");
+   entry / dCurrentPosition = GetPosition();
+   entry / SetVelocity(-30.0);
+   do / @rttAssert(GetServiceIntensity() > 0.0 || GetPosition() >= (dCurrentPosition - 150.0));
+   do / printf("%s%f%s%f%s", "\tSpeed: ", GetVelocity(), " Position: ", GetPosition(), "\n");
```

```
+   entry / SetDMIBooleanValue( "Select Override Function", TRUE);
+   entry / Time.reset();
```

```
+   entry / bOverrideSelected = TRUE;
```

Figure 11.15.: UML state machine diagram of the “Stopped_in_Post_Trip” state

Activating_override Similar to the Mode switch from Unfitted to Staff Responsible in Figure 11.13, the override function is also used here to switch Staff Responsible using the “c37” oModeGuard in Figure 10.21. Thus, the corresponding input field “Select Override Function” in the DMI is set to true.

Switching_to_Unfitted The switch to the ETCS Mode Staff Responsible is propagated to all state machines by the global signal `bOverrideSelected = TRUE`.

11.4. Code Generation

The simulation model of UML state machines introduced in Section 11.3 cannot be directly executed. Therefore, compilable code must be generated (Figure 11.2) or rather a model-to-text transformation must be applied. The RT-Tester application was chosen as generator because its ORA-SIM [72] component provides a complete generator for C to build executable simulations from UML models. Alternatively, custom code generators could have be implemented for the UML state machines. Though, as already discussed in Section 3.4, complete tool development is not in the main focus of this work and using already available solutions is preferred. Furthermore, the RT-Tester application already provides an infrastructure for evaluating test results.

Since the RT-Tester is currently not available under a OSS or FLOSS license, the generation process cannot be published, like other parts of the case study, as FLOSS within this document. Nevertheless, this restriction is not applicable for the generated source code, which can be found in Section H.4.

The generated source code must be linked against libraries provided by RT-Tester, which neither are publicly available. Therefore, the simulation source code cannot be currently compiled and executed without a valid RT-Tester license. Nevertheless, the generated source code is used as reference in this work.

11.5. Simulation Execution Results

The simulation execution generates two types of logs or rather traces:

1. output of the state machines
2. output of the RT-Tester

The output of the state machines is produced by the `printf()` statements in the actions of states and can be found accordingly in the simulation model. It reflects the simulated, virtual track for the train and the expected ETCS Mode switches of the EVC. Errors cannot be found directly in this trace, only the activation of the State “Isolated” of the CEVC machine in Figure 11.12 refers to a successful execution.

The RT-Tester produces for each state machine or rather abstract machine a separated log file. This holds the information about the executed asserts and the test result. Thus, this type is not very qualified for comprehending the virtual track but for the identification of errors or the successful execution of the simulation. Each log file contains at its very end a summary about warnings, failures, and the test verdict [72], which should be “PASS”[5]. The traces are located in the appendix in Section H.5 and Section H.6.

The simulation state machines and data flows of the EVC binary are executed both with the sample time $T_s = 1\text{s}$ but are not especially synchronised. Nevertheless, this apparently big T_s is not a limitation to the validity of the simulation because simply the time scale is increased for the whole simulation process. Furthermore, a big sample time provides the advantage that (local) drifts (see Subsection 8.6.2) in the EVC and the simulation could be better avoided because the simulation was not executed in a hard real-time operating system. Local drifts do not necessarily lead to errors in the simulation but may falsify the results. Since any local drifts are logged, it is ensured that during the generation of the simulation traces neither local drifts occurred in the simulation nor in the EVC binary.

11.6. Conclusion

This chapter illustrated the usage of a simulation to verify the generated EVC binary of the openETCS case study. To follow the MDA principle, the simulation is generated from formal models, which enables the verification of the correctness of the simulation model. Accordingly, the successful execution of the simulation for the generated EVC binary implies a proof of concept for the MDA approach for the development of safety-critical systems in the railway domain as open model software.

In contrast to modelling the simulation in a different meta model than the system, for future work, the simulation models could also be integrated in the openETCS model. This would probably enhance the possibility of detecting errors during the modelling phase but also would increase the complexity of the meta model and the model. Also, the (automatic) generation of simulative tests from the Subset-076-6-3 [82] could be investigated, but therefore the full Subset-026 [83] would have to be modelled.

[5] “NOT TESTED” for CInitPSM because it does not contain any asserts.

12 Conclusion and Outlook

This chapter provides the overall conclusion of this dissertation and proposes possible future work for related research.

The chapters in the Background part (Part I) provided several concepts that are relevant for the development of OSS/FLOSS for safety-critical train control applications. It was identified that the current and typical principles of software development of OSS / FLOSS are not sufficient for the certification for applicable safety standards. Thus, the principles of DSM were discussed by examples of several state-of-the-art meta meta models. Additionally, an own extension of the existing GOPRR meta meta model was introduced, which complies with all requirements for defining a completely formal meta model. Besides a tree- and graph-based graphical formalism for defining the concrete syntax of a meta model, also the definition of an abstract syntax model and the integration of OCL for defining constraints as static semantics were developed.

Part Dependability (Part II) dealt with issues related to safety and security in connection with the idea of developing safety-critical software as OSS / FLOSS. Applicable safety standards for the railway domain were illuminated and a possible concept developed to integrate the usage of DSM in the software life cycles, which is defined by these standards. This also included the new integration of V&V based on model properties. Furthermore, the new term open model software was raised, which, in contrast to the the traditional open source software, refers to a MDA developed under the principles and licenses of OSS / FLOSS. New security problems caused by the usage of open model software were identified and discussed. The usage of hardware virtualisation, in contrast to traditional operating system strategies, to oppose those security risks was investigated and elaborated as possible strategy to be used together with open model software.

To proof the correctness and applicability of the developed concepts, part openETCS Case Study (Part III) introduced a completely developed case study for ETCS. This included all required instances for a DSL: Starting from the meta model as formal specification language, to the concrete formal model of a sub-subset of the ETCS SRS, and finally down to the generated source code. Furthermore, all new generators between those instances or rather artefacts were developed and discussed. The meta model description was identified as the most crucial part of the development because any errors made in the definition of the concrete syntax or the static semantics can have an erroneous impact to all lower instances. Accordingly, the concrete syntax – especially the graph bindings – and the static semantic were extensively documented. Furthermore, mathematical models were developed and introduced to also formalise the dynamic semantics of the openETCS meta model.

All static source code that does not have to be generated from a concrete model was designed and implemented as domain framework for the openETCS meta model. An object-oriented design was chosen because this could be aligned to the object-oriented meta model syntax, which simplified the later generator development. Hence, the openETCS domain framework provides classes, which only have to be instantiated from the concrete openETCS model by the generator. All implementations of the dynamic semantics were transferred accordingly to the domain framework libraries. Additionally, functional tests were provided to verify the correctness of the openETCS domain framework implementation.

The openETCS generator, as link between the concrete openETCS model as formal specification and the source code, was developed and its software design was discussed. Additionally, the developed strategies for the verification of the used model transformation for the generation of the openETCS source code for the EVC were described: Assert statements are generated from the GOPRR instance, which can be executed on the transformed GOPPRR model to ensure that all model elements for the model-to-model transformation from GOPRR to GOPPRR model are correctly converted. The final model-to-text transformation from GOPPRR model to the instantiation of the openETCS domain framework is verified by tests for the existence of certain domain framework objects in the generated source code. Furthermore, the openETCS generator provides the possibility to generate the build configuration needed for creating the EVC binary from the generated source code and the configuration for executing the PIM and PSM of the openETCS model in separated virtual machines under a Xen hypervisor.

The ETCS SRS or rather Subset-026 [83] was partly[1] modelled and discussed by exemplary diagrams of the corresponding openETCS model for all developed graph types. Additionally, the possibility of tracing the modification of safety properties due to model extensions was illuminated.

A simulation for the model or rather the generated openETCS EVC binary was developed under the MDA principle to validate the complete openETCS case study. Hence, a special simulative PSM had to be realised to provide the interconnection between EVC binary and simulation. The successful execution of the simulation shows that the proposed and developed concepts for developing safety-critical software for train control applications as open model software can be seen as a proof-of-concept. The reduction of model size and complexity is no limitation to this statement because the influence of further model extensions was also illuminated and the case study is a valid sub-subset of the ETCS SRS.

The concepts for dependability were only exemplary realised by generating configurations for VMs, which was also applied in the simulation execution. The certification of the developed openETCS case study software for the EN 50128 and SWSIL 4 is obviously out of the scope of this work because this always has to be done by an external party. Since neither a concrete hardware target platform was available during this work nor the proposed minimal host operating system, all tests (including the simulation) are pure software tests and no system

[1] according to the scope of the openETCS case study

integration tests.

Future work would primary focus on the extension of the openETCS meta model and model to completely cover the ETCS SRS, which also would render extensions of the domain framework and the generator necessary. The direct integration of the simulation models and the Subset-076-6-3 [82] in the openETCS model and meta model seems promising to enhance the modelling process related to safety properties of the system. Furthermore, it might be necessary to investigate if the tracing of model extensions and safety properties by the identification of differences in model instances is sufficient for SWSIL 4 conformance or if model extensions must also be added to the meta model capabilities.

Currently, not all parts of the developed tool chain conform to the Open Proofs concept, which requires all elements to be FLOSS. Primary, this is related to meta modelling and modelling application because with MetaEdit+ only a closed source application is available for the GOPRR meta meta model. Thus, it would be necessary to develop the openETCS meta model under a different meta meta model that has FLOSS tool support. Alternatively, the openETCS meta model could be simply transformed to avoid a complete redevelopment, which was developed and discussed for the MOF meta meta model. While this transformation only have to be proceeded, the main future work would have to focus on the development of the modelling tool because related Eclipse plug-ins currently do not provide a stable performance. Furthermore, the openETCS model or rather the formal specification has to be remodelled in the new meta model or an appropriate model-to-model transformation has to be found.

Neither, the simulation generation by the RT-Tester application is aligned to the idea of open proofs and could accordingly be extended in further work. In contrast to the development of new FLOSS generators for the simulation model, it seems more promising to directly integrate the simulation specification in the openETCS meta model, which means another extension of the openETCS specification language.

Part IV.

Appendix

A

GOPPRR to MOF Transformation

Although the selection of the GOPRR meta meta model was well reasoned in Section 3.4, the proprietary properties of the modelling application MetaEdit+ is an obstacle to the idea of publishing all generators and artefacts under the principles of FLOSS. Additionally, the application causes some drawbacks that are not related to the meta meta model.

The usage and integration of MetaEdit+ is adequate for the case study presented in this work, but for further work going beyond a case study it might be necessary to use an alternative application. The main problem of using another application is the integration of a new meta meta model because currently no other meta (meta) modelling application that uses GOPRR is available. Then, most elements would have to be manually redeveloped. Instead, a solution that obtains a better re-usability by a transformation from GOPPRR to another meta meta model that has FLOSS tool support could be found.

The most reasonable choice for the new meta meta model is MOF. For example, the Ecore meta meta model is used in Eclipse for meta modelling and modelling [9]. Fortunately, the C++ abstract syntax model is already defined as UML class diagram (see Figure 4.4) and uses the same syntax as MOF [62][76, pp. 64-71]. Figure 4.4 could also be interpreted as a template for GOPPRR meta models represented in MOF. Concrete GOPPRR elements only have to be added to the structure by inheriting from their super type[1] for any existing GOPPRR meta model.

This approach arises the problem that all associations between the GOPPRR elements are defined between the super types and are inherited by all concrete types of the original GOPPRR meta model. As a consequence, for example, all sub types of `CGraph` could have any sub type of `CObject`, which is normally not desired and does not correspond to a unique transformation.

The redefinition of all associations would be a possible solution. In other words, besides the associations between the supertypes, all subtypes or rather concrete meta model types have the concrete associations to other concrete meta model types. An example of this strategy was modelled in Eclipse with EMF and with types from the case study in Part III and is shown in Figure A.1.

[1] CProject, CGraph, CObject, CProperty, CPort, CRole, CRelationship, CBinding, CConnection, CCall, and CGraphicalContainer

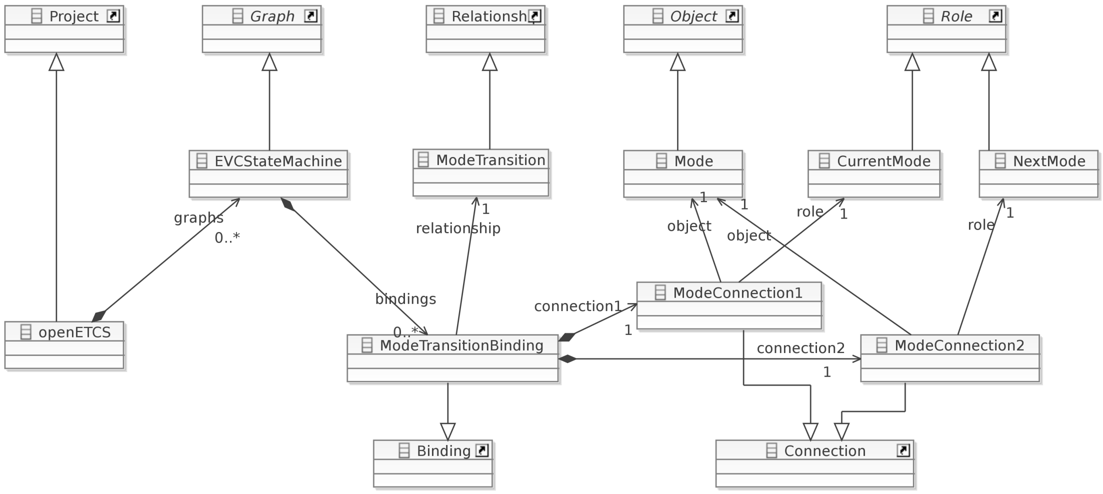


Figure A.1.: A possible transformation from GOPPRR to MOF using redefinitions of associations

This leaves the main problem that associations to different types have to be defined separately and therefore named differently. For example, in `ModelTransitionBinding` the compositions `connection1` and `connection2`. Since those associations can be then named more less arbitrary, model and generator development might emerge as more difficult and even non-unique.

To avoid this problem, constraints could be used instead of the redefinition of associations to define the allowed association ends for certain sub types of a meta model. Fortunately, OCL can also be used for meta models or rather concrete models of MOF. In Figure A.1, to only allow "Mode" objects to be in a "CurrentMode" or "NextMode" tole, the following OCL statement could be used:

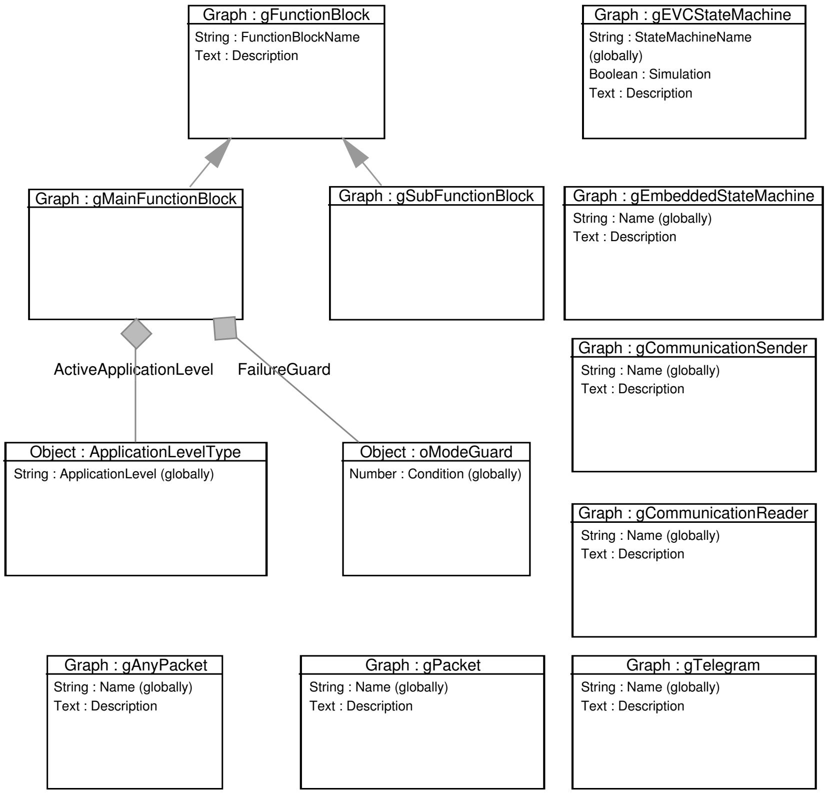
```
context GOPPRR::CProject
inv: m_GraphSet->select(m_Type = 'EVCStateMachine')->forAll(
  graph |
  graph.m_Connection.m_Calls->forAll(
    call |
    call.m_pObject.oclIsTypeOf(Mode)
    and
    (call.m_pRole.oclIsTypeOf(CurrentMode) or call.m_pRole.oclIsTypeOf(NextMode))
  )
)
```

This example shows that it not only renders the redefinitions of associations unnecessary but also the intermediate connection types `ModeConnection1` and `ModeConnection2` because the OCL can be directly defined for the each call in `m_Calls`. A minor disadvantage is that, besides the graphical modelling of sub-types, the bindings are specified by textual OCL statements outside the graphical model, which are not as intuitively understandable, like the GOPPRR abstract syntax description.

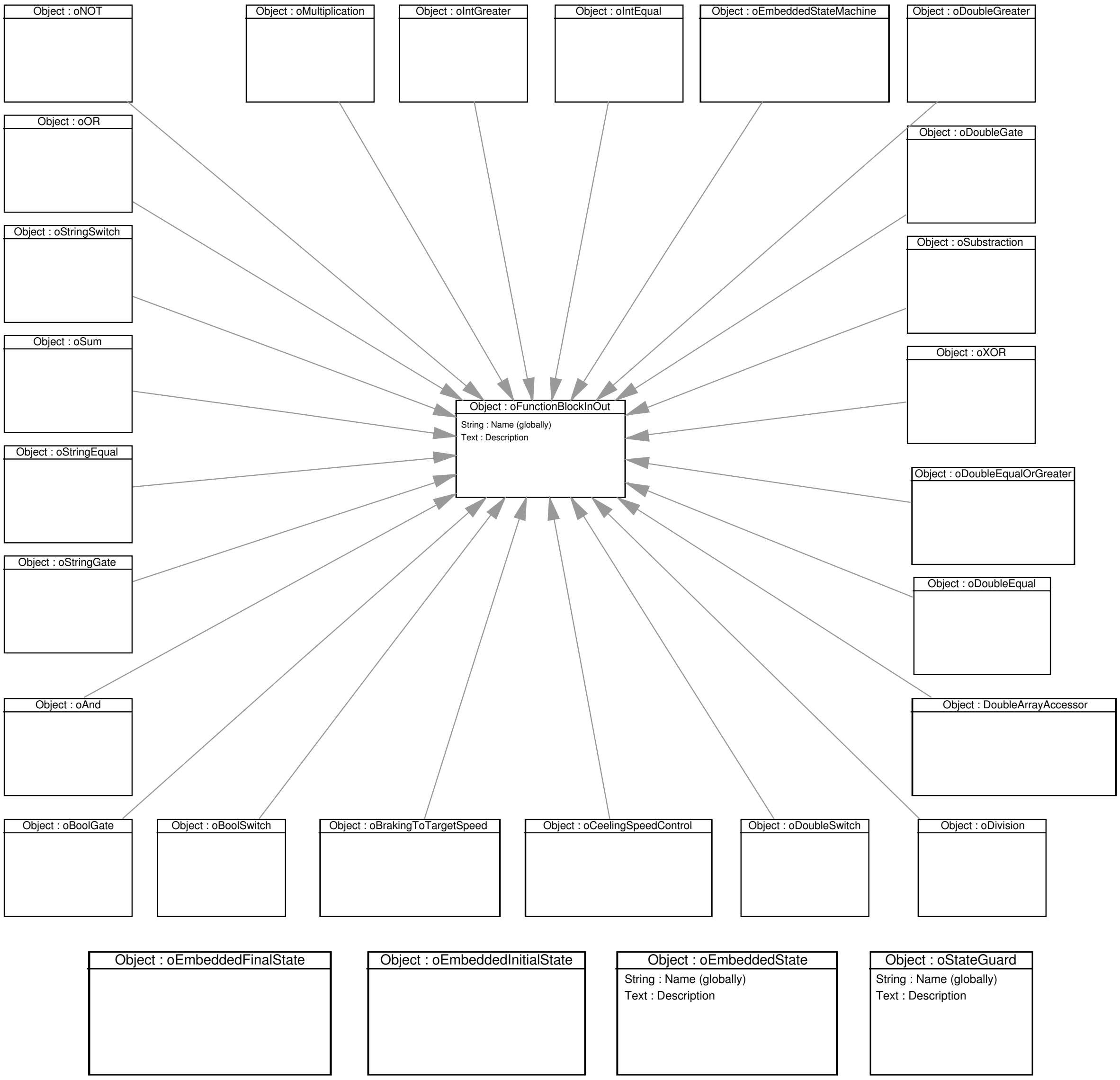
B

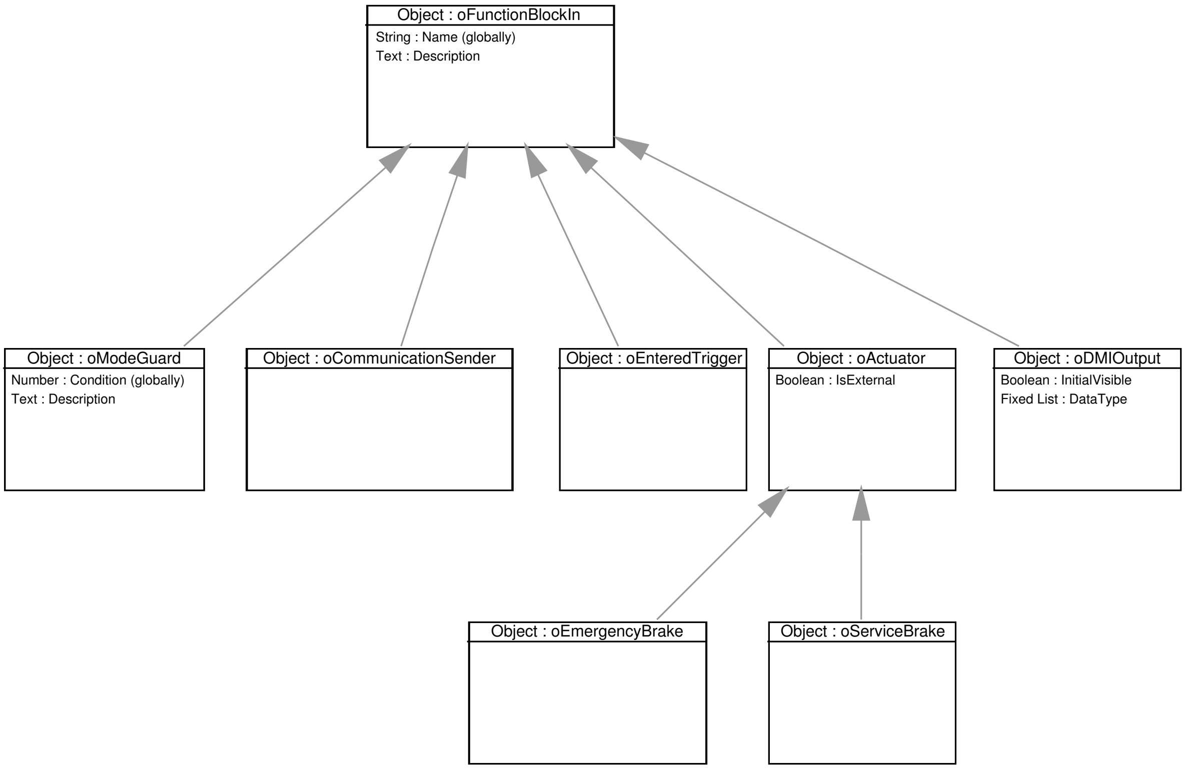
openETCS Meta Model Concrete Syntax

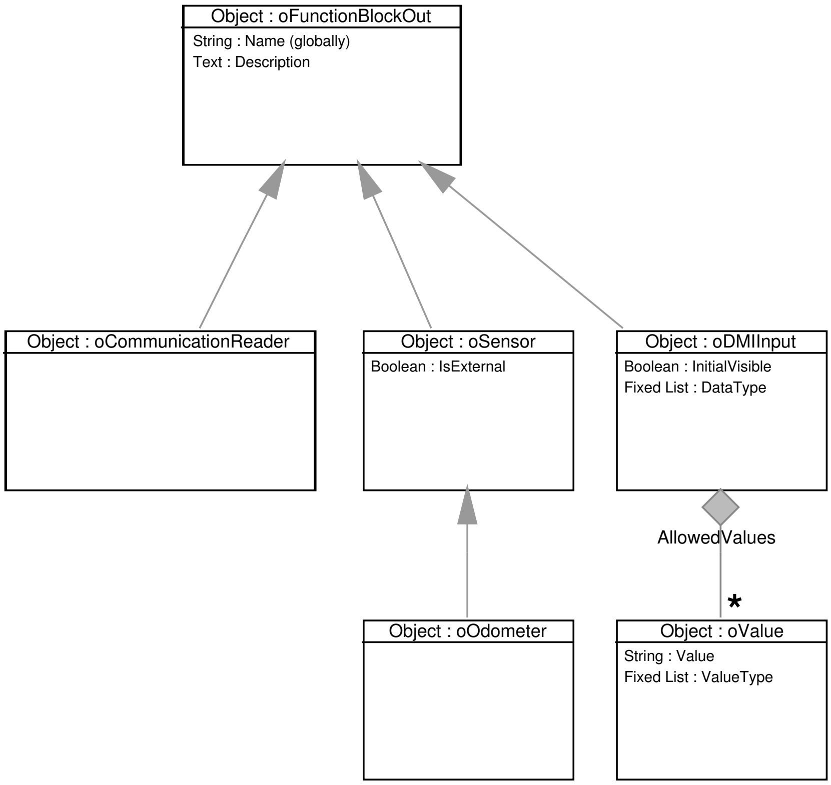
B.1. Graph Type Properties

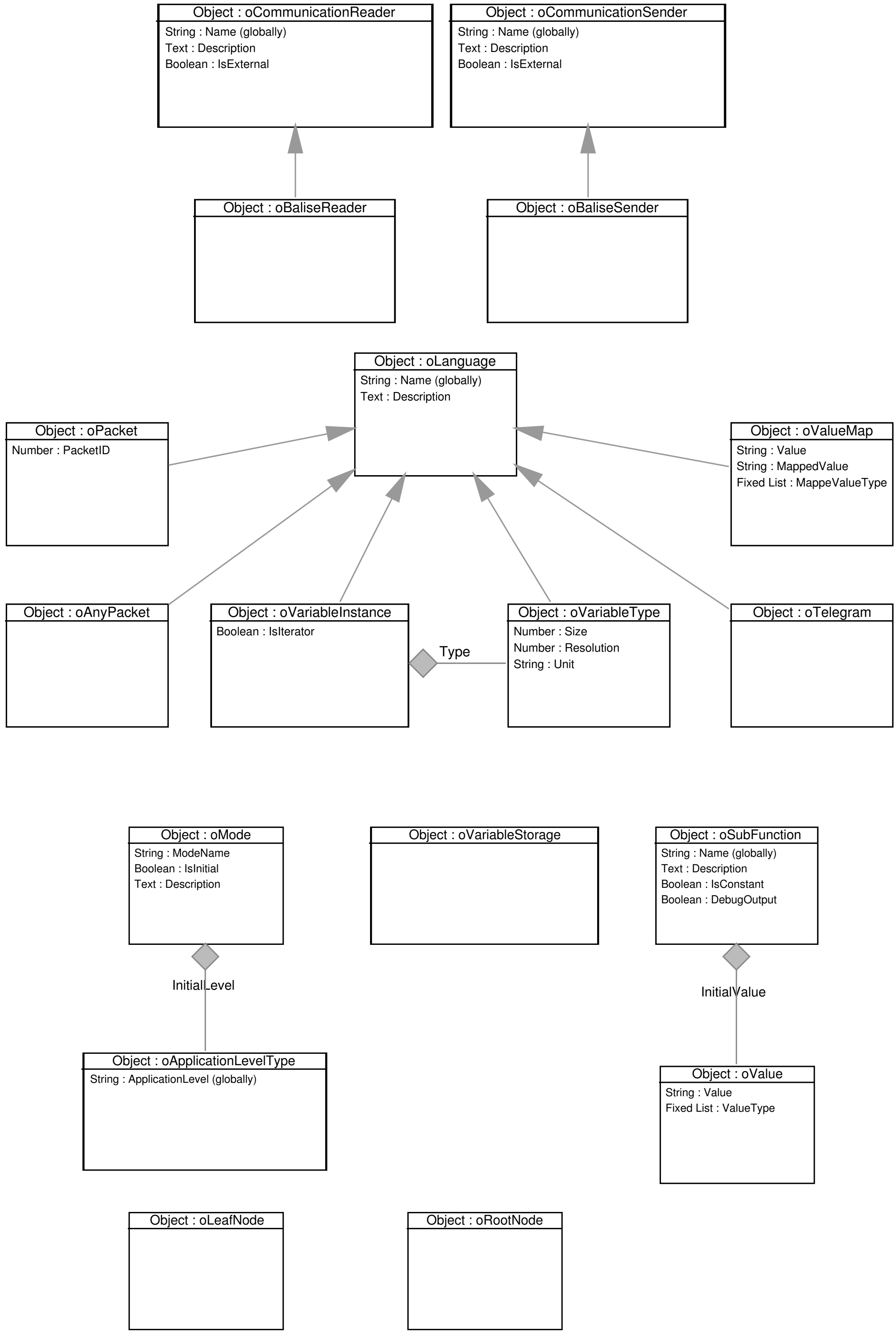


B.2. Object Type Properties

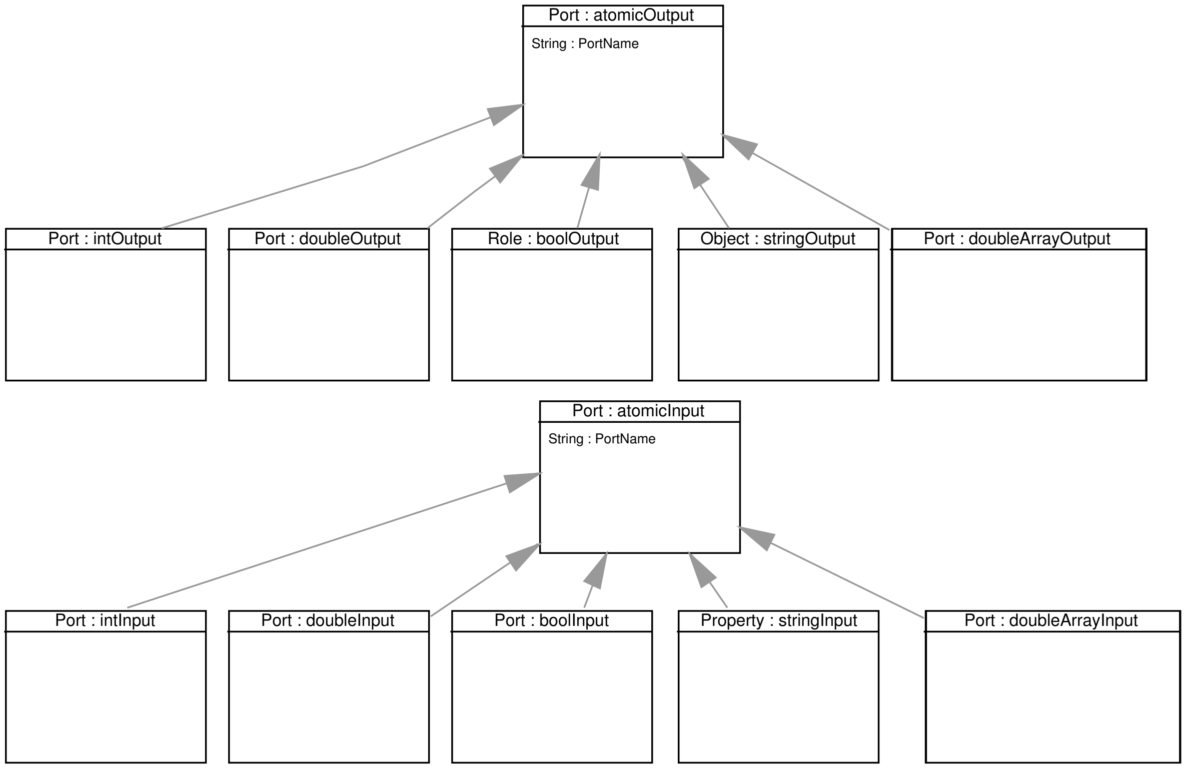



Object : oFunctionBlockIn
String : Name (globally)
Text : Description
Object : oModeGuard
Number : Condition (globally)
Text : Description
Object : oCommunicationSender
Object : oEnteredTrigger
Object : oActuator
Boolean : IsExternal
Object : oDMIOutput
Boolean : InitialVisible
Fixed List : DataType
Object : oEmergencyBrake
Object : oServiceBrake

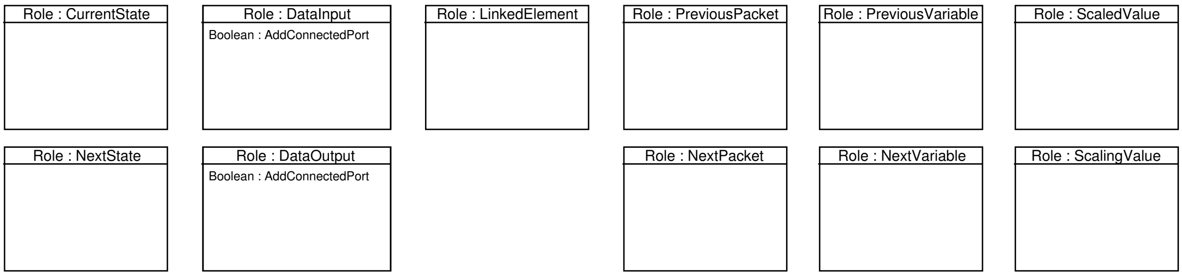

Object : oFunctionBlockOut
String : Name (globally)
Text : Description
Object : oCommunicationReader
Object : oSensor
Boolean : IsExternal
Object : oDMIInput
Boolean : InitialVisible
Fixed List : DataType
AllowedValues
*
Object : oOdometer
Object : oValue
String : Value
Fixed List : ValueType


Object : oCommunicationReader
String : Name (globally)
Text : Description
Boolean : IsExternal
Object : oCommunicationSender
String : Name (globally)
Text : Description
Boolean : IsExternal
Object : oBaliseReader
Object : oBaliseSender
Object : oLanguage
String : Name (globally)
Text : Description
Object : oPacket
Number : PacketID
Object : oValueMap
String : Value
String : MappedValue
Fixed List : MappeValueType
Object : oAnyPacket
Object : oVariableInstance
Boolean : IsIterator
Type
Object : oVariableType
Number : Size
Number : Resolution
String : Unit
Object : oTelegram
Object : oMode
String : ModeName
Boolean : IsInitial
Text : Description
Object : oVariableStorage
Object : oSubFunction
String : Name (globally)
Text : Description
Boolean : IsConstant
Boolean : DebugOutput
InitialLevel
InitialValue
Object : oApplicationLevelType
String : ApplicationLevel (globally)
Object : oValue
String : Value
Fixed List : ValueType
Object : oLeafNode
Object : oRootNode

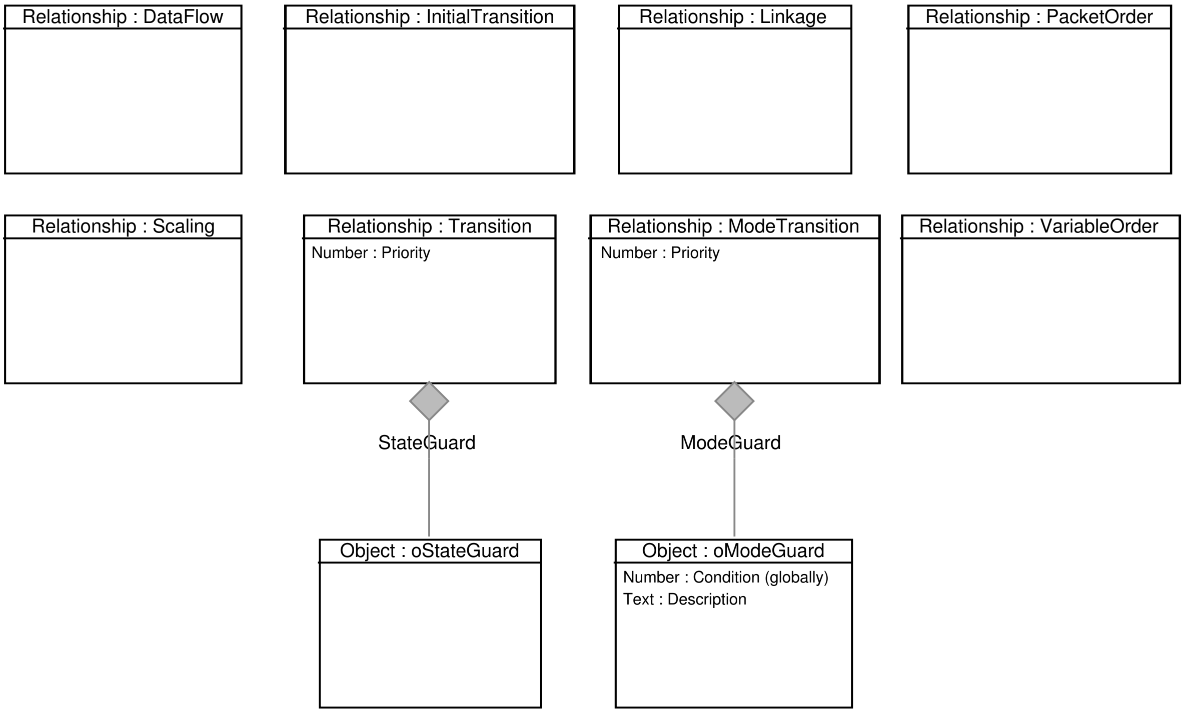
B.3. Port Type Properties



B.4. Role Type Properties



B.5. Relationship Type Properties



B.6. openETCS Meta Model Concrete Syntax Model

The complete concrete syntax model of the openETCS meta model can be accessed as a separated MetaEdit+ patch [56], and MetaEdit+ XML file at
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/GOPPRR-MetaEdit.met` and
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/GOPPRR-MetaEdit.mxt`,
and together with the openETCS model as MetaEdit+ repository [56] in Appendix C.

C
openETCS Model

The complete openETCS model can be accessed at
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/openETCS-Repo.tar.bz2`,
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/openETCS-MetaEdit.mxt`, and
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/openETCS.xml`
as MetaEdit+ repository [56], as MetaEdit+ XML file [96], and as GOPPRR XML file. The MetaEdit+ XML file should not be directly used and only serves here as a reference because it does not contain any port types.

The generated source code from the openETCS model is kept separately at
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/openETCS-GaDF.tar.bz2`.

D
openETCS Domain Framework

D.1. Domain Framework Software Reference

The reference documentation for the openETCS domain framework sources can be found at `http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/doxygen`.

D.2. Domain Framework Source Code

The complete source code of the openETCS domain framework is located at `http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/openETCS-GaDF.tar.bz2`.
The following C++ source code listings are used as examples for the main part of this document.

Listing D.1: Thread creation in C++11

```
int y;

// start all independent data flows threads
for (y = 0; y < m_CurrentDataFlow.size(); y++)
{

  // create new thread
  m_Threads[y] = (new ::std::thread(&::oETCS::DF::CEVCStateMachine::CEVCState::DataFlowThread,
       this, y));

} // for (y = 0; y < m_CurrentDataFlow.size(); y++)
```

Listing D.2: Thread joining in C++11

```
// wait for all threads to finish and delete them
for (y = 0; y < m_Threads.size(); y++)
{
  // check if current thread is joinable
  if (m_Threads[y]->joinable())
  {
    // wait for current thread
    m_Threads[y]->join();

  } // if (m_Threads[y]->joinable())

  // delete current thread
  delete m_Threads[y];

} // for (y = 0; y < m_Threads.size(); y++)
```

Listing D.3: Data flow thread synchronisation

```
// synchronise with all other threads at start
// check if this is already last thread for synchronisation
if (m_iLockedThreads == (m_Threads.size() - 1))
{

  // notify all other threads to proceed
  m_Barrier.notify_all();

} // if (m_iLockedThreads == (m_Threads.size() - 1))
else
{
  // increase number of locked threads
  m_iLockedThreads++;

  // wait for all remaining executing threads
  m_Barrier.wait(Lock);

} // else

// compute start time once
StartTime = ::std::chrono::high_resolution_clock::now();

// execute thread until a state and/or level switch
while (!m_bStateLevelSwitch)
{
  try
  {
    // execute related data flow once
    m_CurrentDataFlow[iIndex]->Execute();

  } // try
  catch (const ::oETCS::DF::Error::CException& Exception)
  {
    // create and print error message to stderr
    ::std::cerr << "Error␣while␣executing␣data␣flow␣#␣" << iIndex << ":␣" << Exception.what() << "␣in␣" << __FILE__ << "␣at␣line␣" << __LINE__ << ::std::endl;

    // store exception on stack
    m_DataFlowExceptions.push_back(Exception);

    // stop execution of all threads by using the switch flag
    m_bStateLevelSwitch = true;

  } // catch (const ::oETCS::DF::Error::CException& Exception)

  // check if this is already last thread for synchronisation
  if (m_iLockedThreads == (m_Threads.size() - 1))
  {
    // check if any transitions were placed on stack
    if (m_TransitionStack.size() > 0)
    {
      // store very first pointer
      pStateTransition = m_TransitionStack[0];

      // search all other transition on stack for higher priority
      for (x = 1; x < m_TransitionStack.size(); x++)
      {
        // compare current priority with stored one
        if (pStateTransition->GetPriority() < m_TransitionStack[x]->GetPriority())
        {
          // store current transition
          pStateTransition = m_TransitionStack[x];

        } // if (pStateTransition->GetPriority() < m_TransitionStack[x]->GetPriority())

      } // for (x = 0; x < m_TransitionStack.size(); x++)

      // do sanity check of transition object
      if (pStateTransition->GetStartState() != this)
      {
        // print error message to stderr
        ::std::cerr << "In␣file␣" << __FILE__ << "␣at␣line␣" << __LINE__ << ":␣activated␣transition␣object␣has␣false␣start␣state" << ::std::endl;

      } // if (pStateTransition->GetStartState() != this)

      // set pointer to new state in parent state machine
      m_pParent->m_pCurrentState = pStateTransition->GetTargetState();

      // activate flag for state/level switch
      m_bStateLevelSwitch = true;

      // clear transition stack
      m_TransitionStack.clear();

    } // if (m_TransitionStack.size() > 0)

    // reset the number of locked threads
    m_iLockedThreads = 0;

    // notify all other threads to proceed
    m_Barrier.notify_all();

  } // if (m_iLockedThreads == (m_Threads.size() - 1))
  else
  {
    // increase number of locked threads
    m_iLockedThreads++;

    // wait for all remaining executing threads
      m_Barrier.wait(Lock);

  } // else


  // compute sleep time by execution time until now
  SleepTime = SAMPLE_TIME - (::std::chrono::high_resolution_clock::now() - StartTime);

  // check if sample time was no exceeded
  if (SleepTime.count() >= 0 )
  {
    // put thread to sleep
    ::std::this_thread::sleep_for(SleepTime);

  } // if (SleepTime.count() >= 0 )
  else
  {
    // print warning to stderr
    ::std::cerr << "In file " << __FILE__ << " in line " << __LINE__ << ": sample time exceeded in thread" << std::endl;

  } // else

  // calculate next start time point to avoid global drifts
  StartTime += SAMPLE_TIME;

} // while (!m_bStateLevelSwitch)
```

Listing D.4: Control flow execution

```
void CControlFlow::CState::Execute() throw(::oETCS::DF::Error::CException)
{
  unsigned int                        x(0);
  const unsigned int                  NUMBER_FUNCTION_BLOCKS(m_FunctionBlocks.size());
  ::oETCS::DF::CTransition*           pTransition(0);


  try
  {
    // execute all related function block objects
    for (x = 0; x < NUMBER_FUNCTION_BLOCKS; x++)
    {
      // calculate current function block
      m_FunctionBlocks[x]->Calculate();

    } // for (x = 0; x < NUMBER_FUNCTION_BLOCKS; x++)


    // check if transition stack is not empty
    if (!m_TransitionStack.empty())
    {
      // store first transition on stack
      pTransition = m_TransitionStack[0];

      // find transition with highest priority
      for (x = 0; x < m_TransitionStack.size(); x++)
      {
        // compare priorities
        if (pTransition->GetPriority() < m_TransitionStack[x]->GetPriority())
        {
          // store current transition
          pTransition = m_TransitionStack[x];

        } // if (pTransition->GetPriority() < m_TransitionStack[x]->GetPriority())

      } // for (x = 0; x < m_TransitionStack.size(); x++)

      // execute transition
      m_pParent->m_pCurrentState = pTransition->GetTargetState();

    } // if (!m_TransitionStack.empty())

    // check if this state is a final state
    if (m_bIsFinal)
    {
      // tell parent control flow to terminate
      m_pParent->m_pCurrentState = 0;

    } // if (m_bIsFinal)

  } // try
  catch (const ::oETCS::DF::Error::CException&)
  {
    // re-throw exception
    throw;

  } // catch (const ::oETCS::DF::Error::CException&)

} // void CControlFlow::CState::Execute()
```

Listing D.5: EVC state machine execution

```
while (m_bStarted)
{
  try
  {
    // start execution of current state
    m_pCurrentState->Start();

  } // try
  catch (const ::oETCS::DF::Error::CException& Exception)
  {
    // set start flag to stopped
    m_bStarted = false;

    // create and print error message to stderr
    ::std::cerr << "Error␣while␣executing␣current␣EVC␣state:␣" << Exception.what() << "␣in␣" <<
        __FILE__ << "␣at␣line␣" << __LINE__ << "␣and␣no␣fault␣state␣available!" << ::std::endl;

  } // catch (const ::oETCS::DF::Error::CException& Exception)

} // while (m_bStarted)
```

Listing D.6: EVC state machine termination

```
void CEVCStateMachine::Stop()  throw(::oETCS::DF::Error::CException)
{
  // check if a current state is available
  if (m_pCurrentState == 0)
  {
    // throw exception
    throw (::oETCS::DF::Error::CInternal("no␣current␣state␣available"));

  } // if (m_pCurrentState == 0)

  // check if current state is already running
  if (m_pCurrentState->IsRunning())
  {
    // explicitly set the state machine to stop
    m_bStarted = false;

    // start execution of current state
    m_pCurrentState->Stop();

    // check if thread is joinable
    if (m_pThread->joinable())
    {
      // wait for thread to terminate
      m_pThread->join();

    } //if (m_pThread->joinable())

    // finally delete thread object
    delete m_pThread;

    // reset thread pointer
```

```
      m_pThread = 0;

  } // if (m_pCurrentState->IsRunning())

} // void CEVCStateMachine::Stop()
```

D.3. Domain Framework Model

The UML model of the openETCS domain framework is located at
http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/openETCS-DF-Model.tar.bz2.

E

openETCS Generator

E.1. GOPPRR C++ Abstract Syntax Source Reference

The reference documentation for GOPPRR C++ abstract syntax implementation is also part of the openETCS domain framework reference documentation in Section D.1.

E.2. GOPPRR XML Schema Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" targetNamespace="http://www.GOPPRR.org"
  xmlns:gopprr="http://www.GOPPRR.org">


  <xs:element name="project">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:graph" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:object" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:property" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:port" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:role" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:relationship" />
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:Name" />
    </xs:complexType>

    <xs:key name="graph-oid">
      <xs:selector xpath="./gopprr:graph" />
      <xs:field xpath="@oid" />
    </xs:key>

    <xs:keyref name="graph-oid-ref" refer="gopprr:graph-oid">
      <xs:selector
        xpath="./gopprr:object/gopprr:decomposition | ./gopprr:object/gopprr:explosions |
             ./gopprr:role/gopprr:explosions | ./gopprr:relationship/gopprr:explosions |
             ./gopprr:property/gopprr:graph-reference" />
      <xs:field xpath="@oid" />
    </xs:keyref>

    <xs:key name="relationship-oid">
      <xs:selector xpath="./gopprr:relationship" />
      <xs:field xpath="@oid" />
    </xs:key>

    <xs:keyref name="relationship-oid-ref" refer="gopprr:relationship-oid">
      <xs:selector
        xpath="./gopprr:graph/gopprr:relationship-reference |
             ./gopprr:graph/gopprr:binding/gopprr:relationship-reference |
             ./gopprr:property/gopprr:relationship-reference |
             ./gopprr:graph/gopprr:graphical-content/gopprr:relationship-reference" />
      <xs:field xpath="@oid" />
    </xs:keyref>

    <xs:key name="role-oid">
      <xs:selector xpath="./gopprr:role" />
      <xs:field xpath="@oid" />
    </xs:key>

    <xs:keyref name="role-oid-ref" refer="gopprr:role-oid">
      <xs:selector
        xpath="./gopprr:graph/gopprr:binding/gopprr:connection/gopprr:role-reference |
            ./gopprr:property/gopprr:role-reference" />
      <xs:field xpath="@oid" />
    </xs:keyref>

    <xs:key name="port-oid">
      <xs:selector xpath="./gopprr:port" />
      <xs:field xpath="@oid" />
    </xs:key>

    <xs:keyref name="port-oid-ref" refer="gopprr:port-oid">
      <xs:selector
        xpath="./gopprr:graph/gopprr:binding/gopprr:connection/gopprr:port-reference |
            ./gopprr:property/gopprr:port-reference" />
      <xs:field xpath="@oid" />
    </xs:keyref>

    <xs:key name="property-oid">
      <xs:selector xpath="./gopprr:property" />
      <xs:field xpath="@oid" />
    </xs:key>

    <xs:keyref name="property-oid-ref" refer="gopprr:property-oid">
      <xs:selector
        xpath="./gopprr:graph/gopprr:property-reference | ./gopprr:object/gopprr:property-reference
            | ./gopprr:port/gopprr:property-reference | ./gopprr:role/gopprr:property-reference |
            ./gopprr:relationship/gopprr:property-reference" />
      <xs:field xpath="@oid" />
    </xs:keyref>

    <xs:key name="object-oid">
      <xs:selector xpath="./gopprr:object" />
      <xs:field xpath="@oid" />
    </xs:key>

    <xs:keyref name="object-oid-ref" refer="gopprr:object-oid">
      <xs:selector
        xpath="./gopprr:graph/gopprr:object-reference |
            ./gopprr:graph/gopprr:binding/gopprr:connection/gopprr:object-reference |
            ./gopprr:property/gopprr:object-reference |
            ./gopprr:graph/gopprr:graphical-content/gopprr:object-reference" />
      <xs:field xpath="@oid" />
    </xs:keyref>

  </xs:element>


  <xs:complexType name="concept">
    <xs:attribute name="oid" use="required" type="xs:NMTOKEN" />
    <xs:attribute name="type" use="required" type="xs:string" />
  </xs:complexType>


  <xs:complexType name="non-property">
    <xs:complexContent>
      <xs:extension base="gopprr:concept">
        <xs:attribute name="name" use="required" type="xs:string" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>


  <xs:complexType name="reference">
    <xs:attribute name="oid" use="required" type="xs:NMTOKEN" />
  </xs:complexType>


  <xs:element name="graph">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="gopprr:non-property">
          <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded"
              ref="gopprr:object-reference" />
            <xs:element minOccurs="0" maxOccurs="unbounded"
              ref="gopprr:property-reference" />
            <xs:element minOccurs="0" maxOccurs="unbounded"
              ref="gopprr:port-reference" />
            <xs:element minOccurs="0" maxOccurs="unbounded"
              ref="gopprr:role-reference" />
            <xs:element minOccurs="0" maxOccurs="unbounded"
              ref="gopprr:relationship-reference" />
            <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:binding" />
            <xs:element minOccurs="0" maxOccurs="unbounded"
              ref="gopprr:graphical-content" />
          </xs:sequence>
          <xs:attribute name="root" use="required" type="xs:boolean"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>


  <xs:element name="object">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="gopprr:non-property">
          <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded"
              ref="gopprr:property-reference" />
            <xs:element minOccurs="0" maxOccurs="1"
              ref="gopprr:decomposition" />
            <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:explosions" />
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>


  <xs:element name="property">
    <xs:complexType mixed="true">
      <xs:complexContent>
        <xs:extension base="gopprr:concept">
          <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="gopprr:graph-reference" />
            <xs:element ref="gopprr:object-reference" />
            <xs:element ref="gopprr:port-reference" />
            <xs:element ref="gopprr:role-reference" />
            <xs:element ref="gopprr:relationship-reference" />
          </xs:choice>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>


  <xs:element name="port">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="gopprr:non-property">
                    <xs:sequence>
                        <xs:element minOccurs="0" ref="gopprr:property-reference" />
                    </xs:sequence>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>


  <xs:element name="role">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="gopprr:non-property">
                    <xs:sequence>
                        <xs:element minOccurs="0" ref="gopprr:property-reference" />
                        <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:explosions" />
                    </xs:sequence>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>


  <xs:element name="relationship">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="gopprr:non-property">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
            ref="gopprr:property-reference" />
          <xs:element minOccurs="0" maxOccurs="unbounded" ref="gopprr:explosions" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>


<xs:element name="graphical-content">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded"
        ref="gopprr:object-reference" />
      <xs:element minOccurs="0" maxOccurs="unbounded"
                  ref="gopprr:relationship-reference" />
    </xs:sequence>
  </xs:complexType>
</xs:element>


<xs:element name="decomposition">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1"
        ref="gopprr:graph-reference" />
    </xs:sequence>
  </xs:complexType>
</xs:element>


<xs:element name="explosions">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded"
        ref="gopprr:graph-reference" />
    </xs:sequence>
  </xs:complexType>
</xs:element>


<xs:element name="binding">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1"
        ref="gopprr:relationship-reference" />
      <xs:element minOccurs="1" maxOccurs="1" ref="gopprr:connection" />
    </xs:sequence>
  </xs:complexType>
</xs:element>


<xs:element name="connection">
  <xs:complexType>
    <xs:sequence minOccurs="2" maxOccurs="unbounded">
      <xs:element minOccurs="1" maxOccurs="1"
        ref="gopprr:object-reference" />
      <xs:element minOccurs="1" maxOccurs="1"
        ref="gopprr:role-reference" />
      <xs:element minOccurs="0" maxOccurs="1"
        ref="gopprr:port-reference" />
    </xs:sequence>
  </xs:complexType>
</xs:element>


<xs:element name="graph-reference" type="gopprr:reference" />


<xs:element name="object-reference" type="gopprr:reference" />


<xs:element name="property-reference" type="gopprr:reference" />


<xs:element name="port-reference" type="gopprr:reference" />


<xs:element name="role-reference" type="gopprr:reference" />


<xs:element name="relationship-reference" type="gopprr:reference" />
```

```


</xs:schema>
```

E.3. Generator Application Source Reference

The source code reference documentation is also part of the openETCS domain framework reference documentation in Section D.1.

E.4. Generator Model

The UML model is also part of the openETCS domain framework model in Section D.3.

E.5. Generator Source Code

The source code is also part of the openETCS domain framework source code in Section D.2. The following C++ source code listings are used as examples for the main part of this document.

Listing E.1: Abstract data flow model initial creation

```
void CCPPGenerator::BuildAbstractModel(GOPPRR::CGraph * const pFunctionBlock, ::std::map<
    ::std::string, oETCS::GEN::CFBNode >& FBNodes) throw(::GOPPRR::Error::CException)
{
  decltype (::GOPPRR::CGraph::m_ObjectSet.begin())            Object;
  decltype (::GOPPRR::CGraph::m_RoleSet)                      Inputs;
  decltype (::GOPPRR::CGraph::m_RoleSet.begin())              Input;
  ::GOPPRR::CRole*                                            pOutput(nullptr);
  ::GOPPRR::CObject*                                          pOutputNode(nullptr);


  // process all objects in sub function block
  for (Object = pFunctionBlock->m_ObjectSet.begin(); Object != pFunctionBlock->m_ObjectSet.end();
      Object++)
  {
    // check if current object is a concrete function block element
    if (m_FBMap.find(Object->second->m_Type) != m_FBMap.end())
    {
      // add current object to node list
      FBNodes[Object->second->m_OID] = ::oETCS::GEN::CFBNode(Object->second->m_OID,
          ::oETCS::GEN::CFBNode::DEFINED);

      // get all input roles of current function block object
      Inputs = pFunctionBlock->Roles(Object->second, "DataInput", false, false);

      // process all inputs
      for (Input = Inputs.begin(); Input != Inputs.end(); Input++)
      {
        // get connected output role
        pOutput = pFunctionBlock->Roles(Input->second, "DataOutput", false, true).begin()->second;

        // get connected function block object
        pOutputNode = pFunctionBlock->Objects(pOutput, true).begin()->second;

        // add input object as undefined node in outputs
        FBNodes[Object->second->m_OID].m_Inputs.push_back(new
            ::oETCS::GEN::CFBNode(pOutputNode->m_OID, ::oETCS::GEN::CFBNode::UNDEFINED)); // TODO:
            handle exceptions

      } // for (Input = Inputs.begin(); Input != Inputs.end(); Input++)

    } // if (m_FBMap.find(Object->second->m_Type) != m_FBMap.end())
    else if (Object->second->m_Type == "SubFunction")
    {
      // check if object has a decomposition
      if (Object->second->m_pDecomposition != 0)
      {
        // call this method recursively on decomposition
        this->BuildAbstractModel(Object->second->m_pDecomposition, FBNodes);

      } // if (Object->second->m_pDecomposition != 0)

    } // else if (Object->second->m_Type == "SubFunction")

  } // for (Object = pFunctionBlock->m_ObjectSet.begin(); Object !=
      pFunctionBlock->m_ObjectSet.end(); Object++)

} // void CCPPGenerator::BuildAbstractModel()
```

Listing E.2: Abstract data flow model full creation

```
// process all nodes found (including sub-graphs)
for (n = FBNodes.begin(); n != FBNodes.end(); n++)
{
  // process all inputs of current node
  for (i = n->second.m_Inputs.begin(); i != n->second.m_Inputs.end(); i++)
  {
    // store OID of current output node
    OID = (*i)->m_OID;

    // delete current undefined object
    delete (*i);

    // use pointer to actual node at iterators place
    *i = &FBNodes[OID];

  } // for (i = n->second.m_Inputs.begin(); i != n->second.m_Inputs.end(); i++)

} // for (n = FBNodes.begin(); n != FBNodes.end(); n++)
```

Listing E.3: Abstract data flow model processing

```
void CCPPGenerator::ProcessAbstractModel(const oETCS::GEN::CFBNode & Node, ::std::list<
    ::std::string >& ExecutionOrder, ::std::vector< const oETCS::GEN::CFBNode * >& NodeStack)
    throw()
{
  decltype (Node.m_Inputs.begin())        i;
  decltype (ExecutionOrder.begin())       s;
  decltype (NodeStack.begin())            ns;
  bool                                    bFound(false);
  bool                                    bOnStack(false);


  // check if this node is already on node stack
  for (ns = NodeStack.begin(); ns != NodeStack.end() && not bOnStack; ns++)
  {
    // check if current node on stack is current node
    bOnStack = (*ns == &Node);

  } // for (ns = NodeStack.begin(); ns != NodeStack.end() && not bOnStack; ns++)


  // only process node if it is not on stack to avoid infinite recursions on data flow loops
  if (not bOnStack)
  {
    // check if current node already exists on execution order stack
    for (s = ExecutionOrder.begin(); s != ExecutionOrder.end() && not bFound; s++)
    {
      // check if current stack element corresponds to current node's OID
      bFound = (*s == Node.m_OID);

    } // for (s = ExecutionOrder.begin(); s != ExecutionOrder.end() && not bFound; s++)


    // check if  node has any input
    if (Node.m_Inputs.empty())
    {
        // check if node is already on stack
      if (not bFound)
      {
        // node is a flow-chain start point and is directly added to the execution order
        ExecutionOrder.push_back(Node.m_OID);

      } // if (not bFound)

    } // if (Node.m_Inputs.empty())
    else
    {
      // process all input nodes
      for (i = Node.m_Inputs.begin(); i != Node.m_Inputs.end(); i++)
      {
        // place current node on stack
        NodeStack.push_back(&Node);

        // call this method recursively for current node
        ::oETCS::GEN::CCPPGenerator::ProcessAbstractModel((**i), ExecutionOrder, NodeStack);

        // remove node again from stack
        NodeStack.pop_back();

      } // for (i = Node.m_Inputs.begin(); i != Node.m_Inputs.end(); i++)

      // check if node is already on stack
      if (not bFound)
      {
        // add current node after all inputs node on execution order stack
        ExecutionOrder.push_back(Node.m_OID);

      } // if (not bFound)

    } // else

  } // if (not bOnStack)

} // void CCPPGenerator::ProcessAbstractModel() throw()
```

F

MetaEdit+ Generators

F.1. GOPPRR XML Generators

F.1.1. EVCStateMachine XML Generator

The openETCS EVCStateMachine XML MERL generator can be accessed at http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generateEVCStateMachineXML.merl.

F.1.2. GraphI XML Generator

The GOPPRR graph XML MERL generator can be accessed at http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generateGraphXML.merl.

F.1.3. Object XML Generator

The GOPPRR object XML MERL generator can be accessed at http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generateObjectXML.merl.

F.1.4. Property XML Generator

The GOPPRR property XML MERL generator can be accessed at http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generatePropertyXML.merl.

F.1.5. Non-Property XML Generator

The GOPPRR non-property XML MERL generator can be accessed at http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generateNonPropertyXML.merl.

F.1.6. Port XML Generator

The GOPPRR port XML MERL generator can be accessed at http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generatePortXML.merl.

F.1.7. Role XML Generator

The GOPPRR role XML MERL generator can be accessed at
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generateRoleXML.merl`.

F.1.8. Relationship XML Generator

The GOPPRR relationship XML MERL generator can be accessed at
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generateRelationshipXML.merl`.

F.2. GOPPRR CppUnit Assertion Generators

The GOPPRR CppUnit assertion MERL generator can be accessed at
`http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/generateCppAsserts.merl`.

G openETCS Unit Testing

G.1. Unit Testing Source Reference

The source code reference documentation is also part of the openETCS domain framework reference documentation in Section D.1.

G.2. Unit Testing Model

The UML model is also part of the openETCS domain framework model in Section D.3.

G.3. Unit Testing Code

The source code is also part of the openETCS domain framework source code in Section D.2.

H openETCS Simulation

H.1. Simulation Platform Specific Model

The the complete PSM for the simulation is part of the domain framework implementation in Appendix D.

H.2. Simulation Platform Specific C-API

The the complete C-API of the PSM for the simulation is part of the domain framework implementation in Appendix D.

H.3. Simulation Model

The complete simulation model as XMI is located at
http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/openETCS-Simulation.xml.

H.4. Simulation Source Code

The complete generated source code generated from the simulation model in Section H.3 can be found at
http://www.informatik.uni-bremen.de/agbs/jfeuser/openETCS/openETCS-Simulation-src.tar.bz2.

H.5. Simulation Execution Trace

```
QAppThread: Executing adaptor event loop ...
PSM initialised
Press <RETURN> for starting simulation

SUT started
Entered NO POWER (NP)
DMI is idle
DMI proxy initialised
DMI is evaluating data
        DMI: checking type of data
        DMI: powering system
        DMI: checking type of data
Entered Stand By (SB)
```

```
        DMI: entering train data
                DMI: entering new driver ID
                DMI: entering new train position
                DMI: entering driver selection
                DMI: entering train length
                DMI: entering train maximum speed
                DMI: starting mission
Entered Unfitted (UN)
        Moving with allowed speed
        DMI: checking type of data
        Speed: 100.000000 Position: 27.754528
        Speed: 100.000000 Position: 55.524556
        Speed: 100.000000 Position: 83.312250
        Speed: 100.000000 Position: 111.079583
        Moving with warning speed
        Speed: 205.000000 Position: 195.936306
        Speed: 205.000000 Position: 252.817428
        Speed: 205.000000 Position: 309.776165
        Speed: 205.000000 Position: 366.708822
        Moving with forbidden speed
        Speed: 219.117100 Position: 484.670926
        Speed: 217.429251 Position: 545.385042
        Speed: 215.924388 Position: 605.673038
        Speed: 214.549457 Position: 665.523256
        Speed: 213.293239 Position: 725.005800
        Speed: 212.145485 Position: 784.141271
        Speed: 211.096829 Position: 843.034789
        Speed: 210.138715 Position: 901.533198
        Entering track with Level 1
        DMI: acknowledging level trainsition
        DMI: checking type of data
        Waiting for overpassing of D_LEVELTR
        Speed: 175.000000 Position: 1474.422800
        Speed: 175.000000 Position: 1523.031578
        Speed: 175.000000 Position: 1571.648862
        Speed: 175.000000 Position: 1620.258078
        Speed: 175.000000 Position: 1668.869286
        Speed: 175.000000 Position: 1717.467709
        Speed: 175.000000 Position: 1766.079161
        Speed: 175.000000 Position: 1814.685897
        Speed: 175.000000 Position: 1863.295647
        Speed: 175.000000 Position: 1911.904473
        Speed: 175.000000 Position: 1960.512133
        Speed: 175.000000 Position: 2009.122515
        Speed: 175.000000 Position: 2057.731876
        Transition to Level 1
Entered Staff Responsible (SR)
        Moving with allowed speed
        Speed: 100.000000 Position: 2182.766876
        Speed: 100.000000 Position: 2210.549932
        Speed: 100.000000 Position: 2238.327904
        Speed: 100.000000 Position: 2266.097432
        Speed: 100.000000 Position: 2293.881959
        Moving with warning speed
        Speed: 184.500000 Position: 2372.967443
        Speed: 184.500000 Position: 2424.261620
        Speed: 184.500000 Position: 2475.471389
        Speed: 184.500000 Position: 2526.718212
        Moving with forbidden speed
        Speed: 197.117100 Position: 2632.987851
        Speed: 195.394606 Position: 2687.569965
        Speed: 193.843213 Position: 2741.702481
        Speed: 192.448156 Position: 2795.452495
        Speed: 191.193687 Position: 2848.811497
        Speed: 190.065637 Position: 2901.763424
        Speed: 189.051267 Position: 2954.487330
        Entering track not allowed for Staff Responsible
Entered Trip (TR)
        Speed: 135.165121 Position: 3286.444740
        Speed: 124.570321 Position: 3323.075907
        Speed: 113.975521 Position: 3356.763916
        Speed: 103.380721 Position: 3387.521565
        Speed: 92.785921 Position: 3415.323837
        Speed: 82.191121 Position: 3440.192337
        Speed: 71.596321 Position: 3462.110898
        Speed: 61.001521 Position: 3481.089896
        Speed: 50.406721 Position: 3497.125772
        Speed: 39.811921 Position: 3510.219971
        Speed: 29.217121 Position: 3520.368447
        Speed: 18.622321 Position: 3527.575910
        Speed: 8.027521 Position: 3531.843242
        Speed: 0.000000 Position: 3533.207930
        Speed: 0.000000 Position: 3533.207930
        Speed: 0.000000 Position: 3533.207930
        Speed: 0.000000 Position: 3533.207930
        Speed: 0.000000 Position: 3533.207930
        DMI: acknowledging trip
        Speed: 0.000000 Position: 3533.207930
        DMI: checking type of data
Entered Post Trip (PT)
        Moving forward
        Speed: 5.585500 Position: 3535.913686
        Speed: 0.000000 Position: 3536.761126
        Speed: 0.000000 Position: 3536.761126
        Speed: 0.000000 Position: 3536.761126
        Moving in reverse
        Speed: −30.000000 Position: 3528.427501
        Speed: −30.000000 Position: 3520.094484
        Speed: −30.000000 Position: 3511.761284
        Speed: −30.000000 Position: 3503.427509
        Speed: −30.000000 Position: 3495.092584
        Speed: −30.000000 Position: 3486.759926
        Speed: −30.000000 Position: 3478.414576
        Speed: −30.000000 Position: 3470.091109
        Speed: −30.000000 Position: 3461.761218
        Speed: −30.000000 Position: 3453.428826
        Speed: −30.000000 Position: 3445.094676
        Speed: −30.000000 Position: 3436.762693
        Speed: −30.000000 Position: 3428.429384
        Speed: −21.171000 Position: 3420.853403
        Speed: −12.342000 Position: 3415.729461
        Speed: −3.513000 Position: 3413.057916
Entered Staff Responsible (SR)
        Moving with allowed speed
        Speed: 100.000000 Position: 3440.370770
        Speed: 100.000000 Position: 3468.152020
        Speed: 100.000000 Position: 3495.956187
        Speed: 100.000000 Position: 3523.708270
        Moving with warning speed
        Speed: 184.500000 Position: 3602.781962
        Speed: 184.500000 Position: 3654.048926
        Speed: 184.500000 Position: 3705.285191
        Speed: 184.500000 Position: 3756.545749
        Moving with forbidden speed
        Speed: 197.117100 Position: 3862.832663
        Speed: 195.394606 Position: 3917.376523
        Speed: 193.843213 Position: 3971.513661
        Speed: 192.448156 Position: 4025.226868
        Speed: 191.193687 Position: 4078.585802
        Speed: 190.065637 Position: 4131.583487
        Speed: 189.051267 Position: 4184.295925
        Isolating ETCS
Entered Isolated (IS)

real    2m47.682s
user    0m23.005s
sys     2m19.725s
```

H.6. Simulation Abstract Machine Logs

H.6.1. CInitPSM Logs

```
TM 00000000021 AM   3 N ****************************** STARTUP OF AM   3 *****************************
                        * PROJECT          : RTT–ORA–SIM–openETCS
                        * COMPONENT        : oEVC
                        * TEST PROCEDURE : P1
                        * TOOL VERSION     : RT–Tester 6.0−4.9.3
                        * AUTHOR           : rtt−mbt
                        * START OF TEST   : 2012:10:06:13:30:12  (local time)
                        * TIME SCALE       : millisecond
                        * SCHEDULING       : AM 3 − am__sim_CInitPSM, LWP 0 − Simulation
                        ************************************************************************
TM 00000167061 AM   3 R **************************** TERMINATION OF AM   3 ****************************
                        * WARNINGS: 0
                        * FAILURES: 0
                        * VERDICT : NOT TESTED
                        ************************************************************************
```

H.6.2. CDMI Logs

```
TM 00000000018 AM   1 N ****************************** STARTUP OF AM   1 *****************************
                        * PROJECT          : RTT–ORA–SIM–openETCS
                        * COMPONENT        : oEVC
                        * TEST PROCEDURE : P1
                        * TOOL VERSION     : RT–Tester 6.0−4.9.3
                        * AUTHOR           : rtt−mbt
                        * START OF TEST   : 2012:10:06:13:30:12  (local time)
                        * TIME SCALE       : millisecond
                        * SCHEDULING       : AM 1 − am__sim_CDMI, LWP 0 − Simulation
                        ************************************************************************
TM 00000010971 AM   1 P PASS: @rttAssert expression in file _sim_CDMI.rts, line 709 evaluates to 1:
                        ((_bSystemPowered == TRUE) || (DMIHasInput() == 1))
TM 00000014969 AM   1 P PASS: @rttAssert expression in file _sim_CDMI.rts, line 690 evaluates to 1:
                        ((_bSystemPowered == TRUE) || (DMIHasInput() == 1))
TM 00000030970 AM   1 P PASS: @rttAssert expression in file _sim_CDMI.rts, line 563 evaluates to 1:
                        ((_bSystemPowered == TRUE) || (DMIHasInput() == 1))
TM 00000056969 AM   1 P PASS: @rttAssert expression in file _sim_CDMI.rts, line 45 evaluates to 1:
                        ((_bSystemPowered == TRUE) || (DMIHasInput() == 1))
TM 00000115969 AM   1 P PASS: @rttAssert expression in file _sim_CDMI.rts, line 211 evaluates to 1:
                        ((_bSystemPowered == TRUE) || (DMIHasInput() == 1))
TM 00000167018 AM   1 R **************************** TERMINATION OF AM   1 ****************************
                        * WARNINGS: 0
                        * FAILURES: 0
                        * VERDICT : PASS
                        ************************************************************************
```

H.6.3. CEVC Logs

```
TM 00000000009 AM   2 N ****************************** STARTUP OF AM   2 *****************************
                        * PROJECT          : RTT–ORA–SIM–openETCS
                        * COMPONENT        : oEVC
                        * TEST PROCEDURE : P1
                        * TOOL VERSION     : RT–Tester 6.0−4.9.3
                        * AUTHOR           : rtt−mbt
                        * START OF TEST   : 2012:10:06:13:30:12  (local time)
                        * TIME SCALE       : millisecond
                        * SCHEDULING       : AM 2 − am__sim_CEVC, LWP 0 − Simulation
                        ************************************************************************
TM 00000008969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 20 evaluates to 1:
                        (GetEmergencyActivation() == 1)
TM 00000008971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 22 evaluates to 1:
                        (strcmp(ctx−>pDMIValue, "No Power") == 0)
TM 00000015969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 722 evaluates to 1:
                        GetEmergencyActivation()
TM 00000015970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 724 evaluates to 1:
                        (strcmp(ctx−>pDMIValue, "Stand By") == 0)
TM 00000030970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 763 evaluates to 1:
                        (strcmp(ctx−>pDMIValue, "Unfitted") == 0)
TM 00000031969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 560 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000031969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 561 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
```

```
TM 00000031969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 562 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000032969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 560 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000032969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 561 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000032969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 562 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000033969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 560 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000033969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 561 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000033970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 562 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000034969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 560 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000034969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 561 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000034969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 562 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000036969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 659 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000036969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 660 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000036970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 661 evaluates to 1:
                        ((GetVelocity() < ((200 * 1.025000e+00) + 1.000000e+00))
                                     && (GetVelocity() > ((200 * 1.025000e+00) - 1)))
TM 00000036971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 664 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000037969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 659 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000037969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 660 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000037969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 661 evaluates to 1:
                        ((GetVelocity() < ((200 * 1.025000e+00) + 1.000000e+00))
                                     && (GetVelocity() > ((200 * 1.025000e+00) - 1)))
TM 00000037970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 664 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000038969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 659 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000038969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 660 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000038969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 661 evaluates to 1:
                        ((GetVelocity() < ((200 * 1.025000e+00) + 1.000000e+00))
                                     && (GetVelocity() > ((200 * 1.025000e+00) - 1)))
TM 00000038971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 664 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000039969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 659 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000039969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 660 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000039969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 661 evaluates to 1:
                        ((GetVelocity() < ((200 * 1.025000e+00) + 1.000000e+00))
                                     && (GetVelocity() > ((200 * 1.025000e+00) - 1)))
TM 00000039970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 664 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000041969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 610 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (2.000000e+02 * 1.050000e+00)))
TM 00000042969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 610 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (2.000000e+02 * 1.050000e+00)))
TM 00000043969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 610 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (2.000000e+02 * 1.050000e+00)))
TM 00000044969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 610 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (2.000000e+02 * 1.050000e+00)))
TM 00000045969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 610 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (2.000000e+02 * 1.050000e+00)))
TM 00000046969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 610 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (2.000000e+02 * 1.050000e+00)))
TM 00000047969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 610 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (2.000000e+02 * 1.050000e+00)))
TM 00000048969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 610 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (2.000000e+02 * 1.050000e+00)))
TM 00000072971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 449 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Staff Responsible") == 0)
TM 00000073969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000073969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000073969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000074969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000074969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000074969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000075969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000075969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000075970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000076969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
```

```
TM 00000076969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000076969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000077969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000077969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000077969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000079969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 376 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000079969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 377 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000079970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 378 evaluates to 1:
                        ((GetVelocity() < ((180 * 1.025000e+00) + 1.000000e+00))
                                      && (GetVelocity() > ((180 * 1.025000e+00) - 1)))
TM 00000079970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 381 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000080969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 376 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000080970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 377 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000080970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 378 evaluates to 1:
                        ((GetVelocity() < ((180 * 1.025000e+00) + 1.000000e+00))
                                      && (GetVelocity() > ((180 * 1.025000e+00) - 1)))
TM 00000080971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 381 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000081969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 376 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000081969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 377 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000081969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 378 evaluates to 1:
                        ((GetVelocity() < ((180 * 1.025000e+00) + 1.000000e+00))
                                      && (GetVelocity() > ((180 * 1.025000e+00) - 1)))
TM 00000081971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 381 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000082969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 376 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000082969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 377 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000082969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 378 evaluates to 1:
                        ((GetVelocity() < ((180 * 1.025000e+00) + 1.000000e+00))
                                      && (GetVelocity() > ((180 * 1.025000e+00) - 1)))
TM 00000082970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 381 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000084971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000084971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000085971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000085971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000086971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000086971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000087971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000087971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000088970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000088971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000089970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000089970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000090971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000090971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000096970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 99 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Trip") == 0)
TM 00000096970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 100 evaluates to 1:
                        (GetEmergencyActivation() == TRUE)
TM 00000116969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 940 evaluates to 1:
                        (GetEmergencyActivation() == FALSE)
TM 00000117969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 868 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00) || (GetVelocity() == 0.000000e+00))
TM 00000118969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 868 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00) || (GetVelocity() == 0.000000e+00))
TM 00000119969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 868 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00) || (GetVelocity() == 0.000000e+00))
TM 00000120969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 868 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00) || (GetVelocity() == 0.000000e+00))
TM 00000122969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000123969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000124969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
```

```
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000125969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000126969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000127969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000128969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000129970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000130969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000131969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000132969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000133969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000134969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000135969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000136969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000137969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 904 evaluates to 1:
                        ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetPosition() >= (ctx->dCurrentPosition - 1.500000e+02)))
TM 00000142970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 919 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Staff Responsible") == 0)
TM 00000143969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000143969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000143969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000144969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000144969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000144969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000145969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000145969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000145970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000146969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 236 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000146969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 237 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000146969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 238 evaluates to 1:
                        ((GetVelocity() < 1.010000e+02) && (GetVelocity() > 9.900000e+01))
TM 00000148969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 376 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000148969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 377 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000148969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 378 evaluates to 1:
                        ((GetVelocity() < ((180 * 1.025000e+00) + 1.000000e+00))
                                     && (GetVelocity() > ((180 * 1.025000e+00) - 1)))
TM 00000148970 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 381 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000149969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 376 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000149969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 377 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000149969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 378 evaluates to 1:
                        ((GetVelocity() < ((180 * 1.025000e+00) + 1.000000e+00))
                                     && (GetVelocity() > ((180 * 1.025000e+00) - 1)))
TM 00000149971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 381 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000150969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 376 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000150969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 377 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000150969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 378 evaluates to 1:
                        ((GetVelocity() < ((180 * 1.025000e+00) + 1.000000e+00))
                                     && (GetVelocity() > ((180 * 1.025000e+00) - 1)))
TM 00000150971 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 381 evaluates to 1:
                        (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000151969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 376 evaluates to 1:
                        (GetEmergencyActivation() == 0)
TM 00000151969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 377 evaluates to 1:
                        (GetServiceIntensity() == 0.000000e+00)
TM 00000151969 AM   2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 378 evaluates to 1:
                        ((GetVelocity() < ((180 * 1.025000e+00) + 1.000000e+00))
                                     && (GetVelocity() > ((180 * 1.025000e+00) - 1)))
```

```
TM 00000151971 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 381 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000153971 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000153971 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                         ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000154970 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000154971 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                         ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000155970 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000155971 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                         ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000156970 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000156970 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                         ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000157971 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000157971 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                         ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000158970 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000158970 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                         ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000159970 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 312 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Current Train Speed is too high!") == 0)
TM 00000159970 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 313 evaluates to 1:
                         ((GetServiceIntensity() > 0.000000e+00)
                                      || (GetVelocity() <= (1.800000e+02 * 1.050000e+00)))
TM 00000165971 AM    2 P PASS: @rttAssert expression in file _sim_CEVC.rts, line 41 evaluates to 1:
                         (strcmp(ctx->pDMIValue, "Isolation") == 0)
TM 00000166975 AM    2 R *************************** TERMINATION OF AM    2 ***************************
                         * WARNINGS: 0
                         * FAILURES: 0
                         * VERDICT : PASS
                         ******************************************************************************
```

Glossary

ANTLR ANother Tool for Language Recognition is the name of a parser generator that uses LL(*) parsing. 36

API An application programming interface is a specification intended to be used as an interface by software components to communicate with each other. 145, 147, 215, 222, 229, 269

ARINC The Aeronautical Radio, Incorporated is a major provider of transport communications and systems engineering solutions for eight industries: aviation, airports, defence, government, healthcare, networks, security, and transportation. 67

ATP Automatic Train Protection provides automatic train stopping if active limits (speed or distance) are ignored. 11, 12, 14, 20, 35, 79, 186, 190, 218, 227, 232, 280

Backdoor A Backdoor is a hidden method for bypassing normal computer authentication systems. 20, 66

BNF The Backus-Naur Form is a meta syntax that can be used to describe context-free grammars. 30, 31, 36, 278

C In computing, C is a general-purpose programming language. 96, 215, 222, 229, 233

C++ C++ is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. 23, 40, 43, 44, 46–48, 50, 52–54, 74, 80, 96, 107, 122, 131, 132, 134, 147–149, 151, 154–156, 160, 162–164, 169–171, 173, 175, 176, 178, 209, 215, 241, 251, 257, 261, 282

CASE-Tool A Computer-Aided Software Engineering Tool is used for assistance in the development cycle in software development. 21

CIM In software engineering, a Computational-Independent Model is a model of a software system that is independent of any computational formalisms or aspects. 141, 148, 179, 209, 212, 213

CMOF The Complete Meta Object Facility describes the complete set of elements of MOF. 25

CORBA The Common Object Request Broker Architecture is a standard defined by the OMG that enables software components written in multiple computer languages and running on multiple computers to work together, i.e. it supports multiple platforms. 74, 75, 147, 279

CPU The Central Processing Unit or the processor is the portion of a computer system that carries out the instructions of a computer program and is the primary element carrying out the computer's functions. 69, 71–74, 280

DIN Deutsches Institut für Normung e.V. is the German national organization for standardisation and is the ISO member body. 58, 59, 62

DMI A Driver Machine Interface is an ETCS vehicle device. 15, 87, 89, 122, 127, 128, 131, 154, 181, 183, 186, 196, 200, 206, 211, 213, 215, 218, 219, 221–223, 225, 227, 229, 231, 233

DOM The Document Object Model is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML, and XML documents. 160, 170

DSL A Domain-Specific Language is a software modelling language that is highly specialized for a certain problem domain. 2, 3, 6, 21–23, 26, 28, 30, 35–37, 39, 62, 65, 79–81, 120, 121, 141, 159, 225, 235, 279–281, 283

DSM Domain-Specific Modelling is a modern software engineering method related to a certain problem domain. 2, 3, 21–23, 26, 28, 30, 31, 35, 37, 43, 61, 62, 65, 76, 80, 235, 281

EBNF The Extended Backus-Naur Form extends the BNF about operators to define the occurrence of expressions. 31

Eclipse Eclipse is a popular FLOSS IDE. 36, 61, 279

Ecore Ecore is a meta meta model very similar to MOF that is used by EMF. 28, 36, 37, 241

EMF The Eclipse Modelling Framework is a Java-based software framework for model driven software development. 28, 36, 241, 278, 279

EMOF Essential Meta Object Facility describes the essential set of elements of MOF. 25, 26

EN European standards for products and services. 58–62, 66, 236

ERA The European Railway Agency is one of the official agencies in the European Union. 13

ETCS The European Train Control System is a component of the standardized European train traffic control system. 1–3, 6, 13–20, 35, 36, 52, 65, 79–81, 83, 85, 89, 91, 95, 96, 98, 103, 105, 107, 108, 110, 120, 122, 126, 127, 132, 133, 137, 139, 141, 143, 153, 154, 179, 181, 183, 186, 190, 194, 200–202, 205, 206, 208, 209, 211, 212, 218, 222, 223, 225, 227, 229–231, 233, 235–237, 278–282

EUPL The European Union Public License is a FLOSS software license that has been created and approved by the European Commission. 20

Eurobalise Discrete data transfer element used in ETCS similar to a transponder in the rails. 14, 16, 17

Euroloop Semi-continuous data transfer element used in ETCS. 14, 16

Euroradio Standard for data transfer encryption. 14–16

EVC An European Vital Computer is an ETCS vehicle device. 15, 16, 18, 20, 52, 80, 81, 83–85, 93, 98, 99, 122, 126–128, 132, 137, 139, 141, 143, 145, 153–156, 159, 174, 176, 178, 181, 186, 192, 200, 211–213, 218, 219, 222, 223, 225, 229, 231, 233, 234, 236

FAA The Federal Aviation Administration is an agency of the United States Department of Transportation with authority to regulate and oversee all aspects of civil aviation in the U.S. 281

FLOSS Free/Libre Open-source software is, additionally to OSS, distributed under licenses that allow users to use, study, and change the software / source code. 4, 5, 12, 13, 19, 20, 37, 57, 61, 62, 65, 69, 76, 122, 149, 160, 162, 169, 174, 178, 233, 235, 237, 241, 278, 281

FRS The Function Requirements Specification describes all functional requirements of a system and is typically used as input for the SRS. 13

FS Full Supervision is an ETCS Mode. 17, 80

GEF The Graphical Editing Framework is an Eclipse plug-in, which provides graphical functionality. 36

GMF The Graphical Modelling Framework is an Eclipse plug-in that combines EMF and GMF for creating DSLs. 36, 279

GOPPRR The Graph, Objects, Properties, Ports, Roles, and Relationships is an extension of the GOPRR meta meta model. 2, 3, 40, 42–44, 46–48, 50, 52–54, 61, 62, 79, 80, 85, 97, 107, 108, 120, 148, 159, 160, 162–164, 166, 169–171, 173–176, 178, 179, 209, 236, 241, 242, 249, 257, 265, 266

GOPRR Graphs, Objects, Relationships, and Roles is a meta meta model. 2, 31, 32, 34–37, 39, 40, 43, 44, 46, 47, 50, 52, 53, 174–176, 178, 179, 235–237, 241, 279, 281

GSM-R Global System for Mobile Communications is an international wireless communications standard for railway communication and applications. 14–16

GUI In computing, a graphical user interface is a type of user interface that allows users to interact with electronic devices with images rather than text commands. 23, 122, 128, 132, 154

hypervisor In computing, a hypervisor, also called virtual machine monitor (VMM), allows multiple operating systems to run concurrently on a host computer. 71–76, 178, 236

IDE An integrated development environment also known as integrated design environment or integrated debugging environment is a software application that provides comprehensive facilities to computer programmers for software development. 5, 36, 278

IOR An Interoperable Object Reference is a CORBA or RMI-IIOP reference that uniquely identifies an object on a remote CORBA server. 75

IPC In computing, Inter-process communication is a set of methods for the exchange of data among multiple threads in one or more processes. 121, 122, 143, 164

IS Isolation is an ETCS Mode. 18, 80

Java Java is a modern, object-oriented programming language. 22, 28, 36, 74, 280

Java bytecode Java bytecode is generated by Java compilers and interpreted by the Java virtual machine and therefore is (mostly) platform independent. 22

LZB Linienförmige Zugbeeinflussung is a cab signalling and train protection system used on selected German and Austrian railway lines as well as the AVE in Spain. 12

MA In ATP, a Moving Authority allows a train to move about a certain distance. 186, 190, 192, 223

MDA Model-driven architecture is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of specifications, which are expressed as models. 3, 39, 50, 158, 164, 209, 211, 234–236, 281

MERL The MetaEdit+ Reporting Language is a script language used in MetaEdit+ to define code generators of graphical models. 37, 43, 44, 47, 50, 107, 108, 175, 176, 265, 266

meta meta model Model language that can be used to describe or define a DSL / meta model. 2, 3, 6, 22, 23, 25, 28, 29, 32, 34–37, 39, 40, 43, 44, 46, 47, 50, 52–54, 62, 65, 79, 97, 148, 162, 164, 179, 235, 237, 241, 278–280

meta model Domain-Specific model language / DSL. 2, 3, 6, 7, 23, 25, 28, 29, 31, 32, 35–37, 39, 40, 43, 46, 47, 50, 52–54, 61, 62, 65, 66, 79–81, 83–85, 97, 98, 108–110, 115, 116, 119–122, 124, 126–128, 130, 131, 137, 149, 158, 162–164, 166, 169, 173, 175, 178, 179, 183, 201, 209, 225, 234–237, 241, 242, 248, 280, 281

MMU The Memory Management Unit is a computer component responsible for handling memory accesses requested by the CPU. 69, 74

moc The metaobject compiler is a tool that is run on the sources of a Qt program. It interprets certain macros from the C++ code as annotations and uses them to generate added C++ code with Meta Information about the classes used in the program. 143

MOF The Meta Object Facility is a meta meta model using the UML. 23, 25, 26, 28, 29, 36, 40, 50, 54, 162, 237, 241, 242, 277, 278

NL Non Leading is an ETCS Mode. 19

NP No Power is an ETCS Mode. 18, 80, 84

OCL The Object Constraint Language is used to define constraints for UML models. 47, 48, 50, 52–54, 80, 97, 100, 107, 120, 162, 169, 170, 178, 235, 242

OEM An original equipment manufacturer manufactures products or components that are purchased by another company and retailed under that purchasing company's brand name. 5

OID The Object Identifier is formatted as a special string and used within MetaEdit+, which is always unique within a GOPRR project. 44, 170, 173, 175, 176

OMG The Object Management Group is a consortium that originally aimed at setting standards for distributed object-oriented systems and is now focused on modelling (programs, systems and business processes) and model-based standards. 23, 158, 277, 282

open model An open model is in a DSL or a MDA a concrete meta model instance / model that is developed and published under the principles of OSS / FLOSS. 65, 66, 69, 72, 74, 75, 79, 80, 281

open model software Software developed by DSM or in a MDA based on open models. 2, 7, 66, 69, 75, 76, 234–236

openETCS openETCS is the concept to use and provide ETCS software under the principles of FLOSS. 3, 5–7, 19, 20, 36, 46, 52, 53, 61, 66, 79–81, 83, 84, 98, 108–110, 115, 116, 119–122, 124, 126–128, 130, 131, 133, 134, 136, 141, 145, 147–149, 153–156, 158–160, 162, 163, 166, 169–171, 173–176, 178, 179, 183, 201, 202, 205, 209, 211–213, 215, 218, 223, 225, 234, 236, 237, 248, 249, 251, 255, 257, 261, 265, 267

OS On Sight is an ETCS Mode. 17

OSS Open-source software is distributed with public source code. 12, 13, 19, 36, 57, 61, 65, 69, 76, 169, 233, 235, 279, 281

PIM In software engineering, a Platform-Independent Model is a model of a software system that is independent of the specific technological platform used to implement it. 141, 143, 145, 147–149, 151, 154, 155, 159, 160, 174, 212, 215, 218, 236

PSM A platform-specific model is a model of a software that is linked to a specific technological platform. For example, a specific programming language, operating system, document file format or database.. 141, 145, 147, 148, 151, 154, 155, 159, 160, 174, 178, 211–213, 215, 217–219, 222, 223, 229, 236, 269

PT Post Trip is an ETCS Mode. 18, 80

PZB Punktförmige Zugbeeinflussung is an intermittent cab signalling system and train protection system used in Germany, Austria, Slovenia, Croatia, Romania, Israel and on one line in Canada. 12

Radio-Infill Radio based semi-continuous data transfer facility. 14

RTCA The Radio Technical Commission for Aeronautics develops guidance documents related to the FAA. 58, 60

RV Reversing is an ETCS Mode. 19

SB Stand By is an ETCS Mode. 18, 80, 84

SE STM European is an ETCS Mode. 19

SF System Failure is an ETCS Mode. 18, 80

SH Shunting is an ETCS Mode. 17

SIL The safety-integrity-level describes the performance of a safety-function. 59

SL Sleeping is an ETCS Mode. 17

SN STM national is an ETCS Mode. 19

SR Staff Responsible is an ETCS Mode. 17, 80

SRS The System Requirements Specification describes all necessary requirements of a system. 13, 35, 79–81, 83, 96, 97, 120, 137, 143, 154, 179, 181, 183, 186, 194, 196, 200–202, 209, 218, 225, 235–237, 279

STL The Standard Template Library is a C++ software library that later evolved into the C++ Standard Library. 44

STM Specific Transmission Module is an ETCS Mode used for compatibility with national train control systems or a corresponding hardware component connection to a national train control system with ETCS. 14–17, 19, 206

SWSIL The software safety-integrity-level describes the performance of a software safety-function. 59, 60, 236, 237

TOPCASED TOPCASED is a toolkit plug-in for Eclipse for the development of safety-critical systems in the avionics domain. 61

TR Trip is an ETCS Mode. 18, 80

UML The Unified Modelling Language can describe parts of object-oriented software. 7, 21, 23, 25, 26, 29, 43, 47, 50, 61, 121, 122, 124, 126, 131, 133, 134, 136, 137, 141, 143, 147–149, 155, 159, 160, 163, 164, 171, 178, 183, 212, 213, 215, 218, 219, 221, 233, 241, 255, 261, 267, 280

UN Unfitted is an ETCS Mode. 17, 80

VM A virtual machine is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine. 52, 53, 69, 71–74, 174, 236

XMI The XML Metadata Interchange is an OMG standard for exchanging meta data information via XML. 37, 61, 178, 215, 269

XML The Extensible Markup Language is a set of rules for encoding documents electronically. 28–30, 37, 40, 43, 46, 47, 50, 52–54, 75, 108, 145, 147, 160, 162, 164, 166, 170, 174, 175, 209, 248, 249, 265, 266, 278, 282, 283

XSD The XML Schema Definition defines grammar for XML. 28–30, 46, 47, 61, 170

Xtext Xtext is a plug-in for Eclipse for creating textual DSLs. 36

Bibliography

[1] "AMD-V^{TM} Nested Paging," Advanced Micro Devices, Inc., July 2008, white paper.

[2] "AFDX - End System Detailed Functional Specification," Airbus France, AIRBUS FRANCE, 316, route de Bayonne, 31060 TOULOUSE CEDEX 03 - FRANCE, October 2001.

[3] "Avionics Application Software Interface, Part 1, Required Services," ARINC, AERONAUTICAL RADIO, INC., 2551 Riva Road, Annapolis, Maryland 21401-7435, December 2005.

[4] P. L. H. Arnaud Le Hors, "Document Object Model (DOM) Level 3 Core Specification," W3C, W3C Recommendation 07 April 2004 Version 1.0, April 2004, accessed October 23rd, 2012. [Online]. Available: http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/

[5] "Autosar – automotive open system architecture," autosar.org, accessed October 23rd, 2012. [Online]. Available: http://www.autosar.org/

[6] I. Bashir, A. Goel, L. Int, and V. McLean, "Testing C++ classes," in *Software Testing, Reliability and Quality Assurance, 1994. Conference Proceedings., First International Conference on*, 1994, pp. 43–48.

[7] P. Becker, "Working Draft, Standard for Programming Language C++," JTC1/SC22/WG21, Tech. Rep. N3242=1-0012, February 2011, revises N3225. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

[8] G. Berry, "The effectiveness of synchronous languages for the development of safety-critical systems," *white paper, Esterel Technologies*, 2003, accessed October 23rd, 2012. [Online]. Available: http://www.esterel-technologies.com/DO-178B/files/The-Effectiveness-of-Synchronous-Languages-for-the-Development-of-Safety-Critical-Systems.pdf

[9] F. Budinsky, S. Brodsky, and E. Merks, *Eclipse modeling framework.* Pearson Education, 2003.

[10] "EN 50126 - Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)," CENELEC, CENELEC European Committee for Electrotechnical Standardization, Central Secretariat: rue de Stassart 35, B - 1050 Brussels, September 1999.

[11] "EN 50128 - Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems," CENELEC, CENELEC European Committee for Electrotechnical Standardization, Central Secretariat: rue de Stassart 35, B - 1050 Brussels, March 2001.

[12] "EN 50128 - Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems," CENELEC, CENELEC European Committee for Electrotechnical Standardization, Central Secretariat: rue de Stassart 35, B - 1050 Brussels, pp. 45 – 59, March 2001, annex A.

[13] "EN 50129 - Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling," CENELEC, CENELEC European Committee for Electrotechnical Standardization, Central Secretariat: rue de Stassart 35, B - 1050 Brussels, February 2003.

[14] "CMake - Cross Platform Make," cmake, accessed October 23rd, 2012. [Online]. Available: http://www.cmake.org/

[15] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "Vcc: A practical system for verifying concurrent c," in *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, ser. Lecture Notes in Computer Science, vol. 5674. Springer.

[16] "SourceForge.net: cppunit," cppunit, accessed October 23rd, 2012. [Online]. Available: http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page

[17] "Rascal - meta programming language," CWI - Software Analysis and Transformation, accessed February 25th, 2013. [Online]. Available: http://www.rascal-mpl.org/

[18] "Xen - Debian Wiki," Debian, accessed October 10, 2012. [Online]. Available: http://wiki.debian.org/Xen

[19] "DIN EN 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems," Deutsches Institut für Normung e.V., August 2005.

[20] "Eclipse - the eclipse foundation open source community website," eclipse.org, accessed October 23rd, 2012. [Online]. Available: http://www.eclipse.org/

[21] "The european railway agency at a glance," ERA, 2012, accessed October 23rd, 2012. [Online]. Available: http://www.era.europa.eu/

[22] European Railway Agency, "ERTMS/ETCS Functional Requirements Specification FRS," July 2007, version 5.0.

ERA/ERTMS/00304

[23] ——, "List of Mandatory Specifications," October 2012, accessed October 23rd, 2012. [Online]. Available: http://www.era.europa.eu/Core-Activities/ERTMS/Pages/List-Of-Mandatory-Specifications-and-Standards.aspx

[24] "IDABC - European Union Public Licence - EUPL v.1.1 ," European Union, 2007, accessed October 23rd, 2012. [Online]. Available: https://joinup.ec.europa.eu/software/page/eupl/introduction-eupl-licence

[25] L. Ferier, S. Pinte, and D. Blasband, “ERTMS Formal Specs: a domain specific language to formalize ERTMS specifications for onboard unit development,” ERTMS Solutions, Tech. Rep., 2011, accessed October 23rd, 2012. [Online]. Available: http://www.ertmssolutions.com/files/ERTMSFormalSpecs_WCRR2011.pdf

[26] J. Feuser and J. Peleska, “Security in Open Model Software with Hardware Virtualisation–The Railway Control System Perspective,” in *Foundations and Techniques for Open Source Software Certification 2010*, vol. 33, 2010, accessed October 23rd, 2012. [Online]. Available: http://journal.ub.tu-berlin.de/index.php/eceasst/issue/view/44

[27] ——, “Model Based Development and Tests for openETCS Applications – A Comprehensive Tool Chain,” in *FORMS/FORMAT 2012*, E. Schnieder and G. Tarnai, Eds., 12 2012.

[28] ——, “Dependability in Open Proof Software with Hardware Virtualization – The Railway Control Systems Perspective,” *Science of Computer Programming*, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642313002001

[29] F. Flammini, Ed., *Railway Safety, Reliability and Security: Technologies and Systems Engineering*. IGI Global, 2012.

[30] O. Föllinger, F. Dörrscheidt, and M. Klittich, *Regelungstechnik*. Hüthig, 1992.

[31] “freedesktop.org - software/dbus,” freedesktop.org, accessed October 23rd, 2012. [Online]. Available: http://www.freedesktop.org/wiki/Software/dbus

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addision-Wesley, 2007.

[33] “GCC, the GNU Compiler Collection,” GNU, accessed October 23rd, 2012. [Online]. Available: http://gcc.gnu.org/

[34] M. Gogolla, F. Büttner, and M. Richters, “USE: A UML-Based Specification Environment for Validating UML and OCL,” *Science of Computer Programming*, no. 69, pp. 27–34, 2007.

[35] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel, “Text-based modeling,” in *4th International Workshop on Software Language Engineering*, 2007, accessed February 27th, 2013. [Online]. Available: http://www.se-rwth.de/~rumpe/publications20042008/Groenniger_et_al_ATEM_07.pdf

[36] D. Harel and A. Naamad, “The statemate semantics of statecharts,” *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293–333, October 1996.

[37] K. R. Hase, “openETCS - Ein Vorschlag zur Kostensenkung und Beschleunigung der ETCS-Migration,” *SIGNAL + DRAHT*, vol. 10, October 2009.

[38] ——, “openETCS - Open Source Software für ETCS-Fahrzeugausrüstung,” *SIGNAL + DRAHT*, vol. 12, December 2009.

[39] ——, "“Open Proof” for Railway Safety Software-A Potential Way-Out of Vendor Lock-in Advancing to Standardization, Transparency, and Software Security," *FORMS/FORMAT 2010*, pp. 5–38, 2011.

[40] K. Hase, "SIGNALLING & TELECOMMUNICATIONS SUPPLEMENT – openETCS: Applying 'Open Proofs' to European Train Control," *European Railway Review*, vol. 18, no. 3, p. 30, 2012.

[41] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Addision-Wesley Publishing Company, 1999.

[42] M. M. Henry S. Thompson, David Beech and N. Mendelsohn, "XML Schema Part 1: Structures Second Edition," W3C, Tech. Rep. W3C Recommendation 28 October 2004, October 2004, accessed October 23rd, 2012. [Online]. Available: http://www.w3.org/TR/xmlschema-1/

[43] "ITEA2 - information technology for european advancement," ITEA, accessed October 23rd, 2012. [Online]. Available: http://www.itea2.org/

[44] J. M. Joaquin Miller, "MDA Guide – Version 1.0.1," June 2003, accessed October 23rd, 2012. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01

[45] S. Kelly, "Towards a comprehensive metacase and came environment," Ph.D. dissertation, University of Jyväskylä, Jyväskylä, 1997.

Conceptual, Architectural, Functional and Usability Advances in MetaEdit+

[46] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling*. JOHN WILEY & SONS, INC., 2008.

[47] "The linux kernel archives," kernel.org, accessed October 23rd, 2012. [Online]. Available: http://www.kernel.org/

[48] D. E. Knuth, "Backus normal form vs. backus naur form," *Communications of the ACM 7*, vol. 12, p. 735–736, 1964.

[49] H. Kopetz and G. Grunsteidl, "TTP - a time-triggered protocol for fault-tolerant real-timesystems," in *The Twenty-Third International Symposium on Fault-Tolerant Computing, Digest of Papers, Toulouse, France*, IEEE. FTCS-23, 1993.

[50] H. Krahn, B. Rumpe, and S. Völkel, "Integrated definition of abstract and concrete syntax for textual languages," *Model Driven Engineering Languages and Systems*, pp. 286–300, 2007, accessed February 25th, 2013. [Online]. Available: http://www.se-rwth.de/~rumpe/publications20042008/KRV_Models_07.pdf

[51] ——, "Monticore: Modular development of textual domain specific languages," *Objects, Components, Models and Patterns*, pp. 297–315, 2008, accessed February 25th, 2013. [Online]. Available: http://www.se-rwth.de/~rumpe/publications20042008/KRV_TOOLS_08.pdf

[52] "Linux advanced Routing & Traffic Control HOWTO," LARTC, accessed October 23rd, 2012. [Online]. Available: http://lartc.org/

[53] N. G. Leveson, *Safeware*. Addison-Wesley, 1995.

[54] "libxml++: libxml++ Reference Manual," libxml++, accessed October 23rd, 2012. [Online]. Available: http://developer.gnome.org/libxml++/stable/

[55] LynuxWorks, "Embedded Virtual Machines, Embedded Hypervisor and Separation Kernel for Operating-system Virtualization: LynxSecure," accessed October 23rd, 2012. [Online]. Available: http://www.lynuxworks.com/virtualization/lynxsecure-hypervisor.pdf

[56] "MetaEdit+ Workbench User's Guide," MetaCASE, accessed October 23rd, 2012. [Online]. Available: http://www.metacase.com/support/45/manuals/mwb/Mw.html

[57] K. Mewes, "Domain-specific modelling of railway control systems with integrated verification and validation," Ph.D. dissertation, University of Bremen, 2009.

[58] E. F. Moore, "Gedanken-experiments on sequential machines," *Annals of Mathematical Studies*, no. 34, p. 129–153, 1956.

[59] Nokia, "Qt — Qt - A cross-platform application and UI framework," 2012. [Online]. Available: http://qt.nokia.com/products/

[60] "Meta Object Facility (MOF) Core Specification," Object Management Group, January 2006, accessed October 23rd, 2012.

[61] "XML Metadata Exchange," Object Management Group, December 2007, accessed October 23rd, 2012. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/2007-12-01

[62] "Object management group - UML," Object Management Group, 2010, accessed October 23rd, 2012. [Online]. Available: http://www.uml.org

[63] "Object constraint language," Object Management Group, January 2012, accessed October 23rd, 2012. [Online]. Available: http://www.omg.org/spec/OCL/2.3.1

[64] OMG, "Uml 2.0 superstructure specification, omg adopted specification," August 2005, accessed October 23rd, 2012. [Online]. Available: http://www.omg.org/spec/UML/2.0/Superstructure/PDF/

[65] "Open proofs," openproofs.org, accessed October 23rd, 2012. [Online]. Available: http://www.openproofs.org

[66] "VirtualBox," Oracle, accessed October 23rd, 2012. [Online]. Available: http://www.virtualbox.org/

[67] J. Pachl, *Railway operation and control*. VTD Rail Publ., 2009.

[68] ——, *Systemtechnik des Schienenverkehrs*, 3rd ed. Teubner, 2002.

[69] T. J. Parr and R. W. Quong, “Antlr: A predicated-ll (k) parser generator,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[70] A. M. Paul V. Biron, “XML Schema Part 2: Datatypes Second Edition,” W3C, Tech. Rep. W3C Recommendation 28 October 2004, October 2004, accessed October 23rd, 2012. [Online]. Available: http://www.w3.org/TR/xmlschema-2/

[71] J. Peleska, J. Feuser, and A. E. Haxthausen, *Railway Safety, Reliability and Security: Technologies and Systems Engineering.* IGI Global, 2012, ch. The Model-Driven openETCS Paradigm for Secure, Safe and Certifiable Train Control Systems, pp. 22–52.

[72] J. Peleska, E. Vorobev, F. Lapschies, and C. Zahlten, “Automated model-based testing with RT-Tester,” University of Bremen, Tech. Rep., 2011, accessed October 23rd, 2012. [Online]. Available: http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/tool/rtt-mbt.pdf

[73] “About - QEMU,” QEMU, accessed October 23rd, 2012. [Online]. Available: http://wiki.qemu.org/Main_Page

[74] “DO-178B/ED-12B - Software Considerations in Airborne Systems and Equipment Certification,” RTCA, Inc., 1140 Connecticut Avenue, N. W., Suite 1020, Washington, D. C. 20036.

[75] O. Schulz, “Entwurf und Analyse sicherheitsrelevanter Kommunikationsarchitekturen,” Ph.D. dissertation, University of Bremen, 2011.

[76] T. Stahl, M. Völter, J. Bettin, and B. Stockfleth, *Model-driven software development: technology, engineering, management.* John Wiley, 2006.

[77] W. Stallings, *Operating systems: internals and design principles.* Prentice Hall, 2008.

[78] N. Storey, *Safety critical computer systems.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.

[79] B. Stroustrup, *The C++ Programming Language. Second Edition.* Addision-Wesley Publishing Company, 1998.

[80] A. Tiwari, “Formal Semantics and Analysis Methods for Simulink Stateflow Models,” Tech. Rep., accessed October 23rd, 2012. [Online]. Available: http://www.csl.sri.com/users/tiwari/papers/stateflow.pdf

[81] “TOPCASED - The Open-Source Toolkit for Critical Systems,” topcased.org, accessed October 23rd, 2012. [Online]. Available: http://www.topcased.org

[82] “Test Sequences,” UNISIG, 2009, Issue 2.3.1.

[83] “ERTMS/ETCS – Class 1 System Requirements Specification,” UNISIG, February 2010, Issue 2.3.0.

[84] “ERTMS/ETCS – Class 1 System Requirements Specification: Basic System Description,” UNISIG, February 2010, ch. 2.

[85] “ERTMS/ETCS – Class 1 System Requirements Specification: ERTMS / ETCS Language,” UNISIG, February 2010, ch. 6.

[86] “ERTMS/ETCS – Class 1 System Requirements Specification: Introduction,” UNISIG, February 2010, ch. 1.

[87] “ERTMS/ETCS – Class 1 System Requirements Specification: Messages,” UNISIG, February 2010, ch. 8.

[88] “ERTMS/ETCS – Class 1 System Requirements Specification: Modes and Transitions,” UNISIG, February 2010, ch. 4.

[89] “ERTMS/ETCS – Class 1 System Requirements Specification: Principles,” UNISIG, February 2010, ch. 3.

[90] “ERTMS/ETCS – Class 1 System Requirements Specification: Procedures,” UNISIG, February 2010, ch. 5.

[91] “The User-mode Linux Kernel Home Page,” User Mode Linux, accessed October 23rd, 2012. [Online]. Available: http://user-mode-linux.sourceforge.net/

[92] “Understanding Full Virtualization, Paravirtualization, and Hardware Assist,” vmware, August 2007, white paper.

[93] H. Vogel, *Gerthsen Physik*, 19th ed. Springer, 1995.

[94] “Open-source software - Wikipedia,” Wikipedia, accessed October 23rd, 2012. [Online]. Available: http://en.wikipedia.org/wiki/Open_source_software

[95] “Free and open software - Wikipedia,” Wikipedia, October 2012, accessed October 23rd, 2012. [Online]. Available: http://en.wikipedia.org/wiki/FLOSS

[96] “XML - extensible markup language (xml) 1.0,” World Wide Web Consortium, March 2010, accessed October 23rd, 2012. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/

[97] “Welcome to xen.org, home of the Xen® hypervisor, the powerful open source industry standard for virtualization,” Xen, accessed October 23rd, 2012. [Online]. Available: http://www.xen.org/

Index