Rail to Digital Automated up to Autonomous Train Operation

D31.2 – Validation of TCMS Data Service concept and formalisation

Due date of deliverable: 26/01/2026

Actual submission date: 26/01/2026

Leader/Responsible of this Deliverable: DB InfraGO AG

Reviewed: Y

Document status		
Revision	**Date**	**Description**
0.1	09/04/2025	Initial Document Structure
0.2	06/05/2025	Early draft version provided for project internal review
0.3	08/05/2025	Sections 3.3.3 and 3.4. added. Document title changed according to amendment WP 31.2 03/24 Bibliography style changed to IEEE
0.4	15/05/2025	Sections 3.4.1 - 3.4.3 completed
0.5	17/05/2025	Section 3.4.4 almost completed. Most comments from DB-internal technical reviews on sections 3.3.3.x, 3.4.1-3.4.3 and chapter 4 were resolved.
0.6	19/05/2025	Completed demonstrator environment description in section 4.5. Updated "actual submission" field in cover page to 21/05/2025.
0.7	20/05/2025	Revision and cleanup of structure and contents from chapter 3 and 5
0.8	21/05/2025	Revision and cleanup of chapters 1, 2, 5, and Executive Summary Document footer updated; most comments closed
0.9	08/07/2025	Updates and improvements according to comments from SMO review
1.0	06/08/2025	Updates of wording and formatting according to comments from SMO Updated due date in document header
1.1	01/09/2025	Updates of used wording in Figure 2, Figure 3 and Figure 33 according to v1.0 review comments from SMO. Removed duplicate remarks in chapter 2, as suggested by SMO in v1.0 review.

		Added two brief examples for operator-specific fault management procedures in chapter 1, as requested by SMO in v1.0 review. Fixed various typos and applied some minor improvements in used wording throughout the whole document. All resolved comments from v1.0 document version have been deleted in the D31.2 MS-Word document.
1.2	27/10/2025	Corrections of document references according to v1.1 review findings from SMO, which also resolves all findings listed in D31.2 v1.1 review sheet from SteCo review. All remaining comments closed and removed from D31.2 MS-Word document. Deliverable submission date updated for v1.2 in doc header. Deliverable code updated for v1.2 in doc footer.
1.3	26/01/2026	Fixes and improvements for spelling and cross-referencing issues from external document review. Deliverable submission date updated for v1.3 in doc header. Deliverable code updated for v1.3 in doc footer.

Project funded from the European Union's Horizon Europe research and innovation programme

Dissemination Level		
PU	Public	x
SEN	Sensitive – limited under the conditions of the Grant Agreement	

Start date: 01/12/2022 Duration: 42 months

ACKNOWLEDGEMENTS

This project has received funding from the Europe’s Rail Joint Undertaking (ERJU) under the Grant Agreement no. 101102001. The JU receives support from the European Union’s Horizon Europe research and innovation programme and the Europe’s Rail JU members other than the Union.

REPORT CONTRIBUTORS

Name	Company	Details of Contribution
Kai Schories	DB	Creator/Leader
Miguel Fernandez	CEDEX	Reviewer
Ignacio Alguacil	INECO	Reviewer
Jan Koning	NS	Reviewer
Thomas, Waschulzik	SMO	Reviewer

EXECUTIVE SUMMARY

The transition towards highly automated and digitalised railway operations introduces a wide range of new onboard functions that are increasingly dependent on sensor- and software-intensive solutions for an automated processing of driver tasks.

Automated processing in unattended driving scenarios includes troubleshooting procedures, that will replace manual actions performed by the train driver or onboard staff. This requires rethinking of onboard fault detection, fault reaction and fault management from the ground on, and implies major architectural changes in today's vehicle systems.

Vehicle-specific fault detection and immediate fault reaction will be always managed by high-integrity onboard functions from TCMS and/or CCS-OB domain. This holds in particular for safety-critical system monitoring and incident prevention, requiring safe (SIL>0) information. However, subsequent operator-specific procedures for post-fault troubleshooting and recovery, can be often performed based on "less safe" (SIL0) information, which might be obtained from common onboard data services.

In this work, the TCMS DataService, as proposed by Eurospec, is considered as viable concept to setup a **service-oriented interface for the provision of vehicle status information** towards operator-specific applications for PIS, fleet management, vehicle maintenance, and potential **future applications for fault management and automated troubleshooting**.

Eurospec TCMS DataService specification [1] defines requirements for operator-specific datasets on system and vehicle level; for example: datasets containing status information from Doors and HVAC systems, as well as datasets containing vehicle global information, such as operating mode, speed and location. To **facilitate the development of client applications**, the TCMS DataService API and hosted datasets have to be specified in a common and concise format.

The main contribution and outcome of Task 31.2 is the **application of formal modelling methods** and model transformation techniques to **provide concise and validatable specifications of TCMS DataService APIs** and hosted vehicle datasets.

Another outcome of Task 31.2 is the **modelling environment and automation workflow**, that has been setup and practically used in a lab demonstrator.

The modelling approach established in this work is using **common open standards**, namely: **YANG 1.1** for the specification of datasets, and **OpenAPI 3.0** for the specification of APIs. The pursued modelling approach has been successfully applied to specify the TCMS DataService APIs and hosted datasets for Brakes and Doors functions. These **specifications have been successfully validated and used in a software prototype**.

Please note, that the actual technologies used in this work to realize the formal data modelling workflow and to implement the TCMS DataService prototype were deliberately chosen to conduct a proof of concept. There is no aim or intent for proposing the developed modelling workflow and used technologies for standardisation or as a standard for submission to TSI.

ABBREVIATIONS AND ACRONYMS

Abbreviation	Definition
API	Application Programming Interface
APM	Automatic Processing Module
ATO	Automatic Train Operation
ATP	Advanced Train Protection
BCU	Brake Control Unit
CCS	Command, Control & Signaling
CCU	Command and Control Unit
DB	Deutsche Bahn
DBS	Digitalisierung Bahnsystem (DB Digital Rail program)
DCU	Door Control Unit
DIA	Diagnostics
DiaLab	DB internal project and demonstrator environment
DIA-VEC	Vehicle Diagnostics, DBS GoA4 function
DS	Data Service
EBA	Eisenbahn-Bundesamt (Germany)
ECU	Electronic Control Unit
ERMTS	European Rail Traffic Management System
ETB	Ethernet Train Backbone (EN IEC 61375-1)
ETCS	European Train Control System
FP	Flagship Program
FRMCS	Future Railway Mobile Communication System
FS	Functional Status
GoA	Grade of Automation
GU(x)	Vendor specific term, synonym for BCU
HM	Health Monitor(ing)
HPC	High-Performance Computing platform
HTTP	Hypertext Transfer Protocol
HVAC	Heating, Ventilation, and Air Conditioning
IM	Infrastructure Manager
IP	Internet Protocol

IP bus	Vendor specific term. Fast Ethernet bus technology used in some TCMS architectures of regional trains for transmission of data between CCUs.
IPM	Incident Prevention and Management (see also APM)
JSON	JavaScript Object Notation
LOC	(Train) Localization
MBD	Model Based Design
MCG	Mobile Communication Gateway (EN IEC 61375-2-6)
MDA	Model-Driven Architecture
MIB	Management Information Base
MVB	Multifunction Vehicle Bus
N/A	Not applicable
OAS	OpenAPI Specification
OB	Onboard
OCORA	Open Control Command and Signalling Onboard Reference Architecture
PC	Personal Computer
PER	Perception function
PIS	Passenger Information System
R2DATO	Rail to Digital Automated up to Autonomous Train Operation
REST	Representational State Transfer
RPC	Remote Procedure Call
RU	Railway Undertaking
SCM	Software Configuration Management
SNMP	Simple Network Management Protocol
SW	Software
TBN	Train Backbone Node (EN IEC 61375-1)
TCN	Train Communication Network (EN IEC 61375-1)
TCMS	Train Control and Management System
TCMS_DS	TCMS Data Service
TRL	Technical Readiness Level (according to [2])
TSI	Technical Specification for Interoperability

URI	Uniform Resource Identifier
VM	Virtual Machine
VLAN	Virtual Local Area Network
WP	Workpackage
WSGI	Web Server Gateway Interface
YANG	Yet Another Next Generation (a modular language representing data structures in an XML tree format)
YAML	YAML Ain't Markup Language

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This document provides a technical report on the implementation and validation of a formal data modelling and workflow automation approach for Eurospec TCMS Data Service (TCMS_DS) [1].

The background and main driver for the work conducted here is fault management for highly Automated Train Operation (ATO) up to Grade of Automation 4 (GoA4, unattended driving) and corresponding onboard architectures for future vehicles [3]. Such onboard architectures have been recently defined and harmonised on European level; particularly in Shift2Rail projects Connecta [4] and X2Rail4 [5], as well as in the OCORA project [6].

Compared to current vehicle systems, future onboard architectures will not only be characterised by increased functional complexity (particularly due to additional CCS-OB components), but they will also be deployed in new operational contexts. These contexts include, for example, remote driving and automated shunting, where neither driver nor staff are on board the vehicle [7].

Automated processing of driver tasks in ATO contexts includes the ability to access and report the health and performance of relevant vehicle systems from the TCMS domain. Furthermore, it includes the ability to support in automated troubleshooting procedures for degraded system states to mitigate negative impacts at the operational level.

Previous research has been conducted in [8] and [9] on new functionalities needed for (1) automated functional tests and (2) permanent health assessment of rolling stock systems during the mission, and degraded mode management in case of failure.

Vehicle-specific fault detection and immediate fault reaction will be always managed by high-integrity onboard functions from TCMS and/or CCS-OB domain. This holds in particular for safety-critical system monitoring and incident prevention, requiring safe (SIL>0) information. However, subsequent operator-specific procedures for post-fault troubleshooting and fault recovery can be often performed based on “less safe” (SIL0) information, which might be obtained from common onboard data services.

For example, in case of a DOORs fault, the operator’s passenger information system might be notified to inform embarked passengers about a degraded boarding service in a particular coach of the train. Another example is forwarding the information about a reduced train speed, due to a degraded BRAKES performance, to the operator’s central fleet management system, for the purpose of operational impact analysis and potential time table adjustments.

The voluntary Eurospec TCMS DataService specification proposes the provision of a centralised onboard data service “as a means for applications to use information generated in a given environment in a consistent manner” [1]. The TCMS_DS scope currently includes several operator-specific datasets defined at the vehicle level and vehicle system level. For example, the TCMS_DS specification defines datasets for the vehicle status (operating mode, position, speed, etc.), as well as datasets with status information from Doors, Lighting, and HVAC systems. These datasets are currently provided for use by operator-specific, non-safety-critical applications concerned with fleet management, passenger information, and vehicle maintenance.

This work evaluates the use of the TCMS_DS concept in ATO contexts, where TCMS_DS acts as a central service for vehicle status information towards operator-specific applications concerned with fault management and automated troubleshooting. Fault management for Brakes and Doors systems were chosen as use cases for problem analysis and conducting proofs of concept.

Formal data modelling methods are here applied to define concise and machine-readable specifications for TCMS_DS datasets and corresponding service APIs for the considered use cases. Based on these specifications, exceutable code is generated and integrated into a software prototype developed for TCMS_DS concept evaluation and model validation.

This work has been carried out in Task 31.2. The primary objectives pursued are:

- Apply formal modelling methods for the definition of TCMS_DS datasets and service APIs.
- Define automation workflows to use TCMS_DS modelling artifacts in software applications.
- Develop a functional TCMS_DS prototype for concept evaluation and validation.
- Demonstrate the practical application of TCMS_DS modelling methods and workflow automation for Brakes and Doors use cases.

The remainder of this document is structured as follows:

- Chapter 2 provides further background on fault management in ATO contexts, the role of TCMS_DS, and the benefits of a formal specification for TCMS_DS datasets and service APIs.
- Chapter 3 describes the formal modelling approach established in this work for TCMS_DS service specification. YANG and OpenAPI have been chosen here as technologies and will be introduced in section 3.1. Data models developed for Brakes and Doors use cases are presented in section 3.2. The model transformation toolchain and automation workflow, established in this work for concept evaluation and model validation, is described in section 3.3.
- Chapter 4 presents the software prototype, which has been developed to evaluate the use of TCMS_DS in ATO onboard contexts. The integration and use of generated artifacts from TCMS_DS data modelling and automation workflow into the software prototype will be described.
- Section 5 concludes the document with lessons learned and further directions.

Please note, that the actual technologies used in this work to realize the formal data modelling workflow and to implement the TCMS DataService prototype were deliberately chosen to conduct a proof of concept. There is no aim or intent for proposing the developed modelling workflow and used technologies for standardisation or as a standard for submission to TSI.

2 BACKGROUND AND MOTIVATION

The European railway sector is massively evolving. New technologies and common standards are being specified and introduced to tackle the challenges of future rail operation. One of these challenges is to increase the network capacity and service quality for passenger and freight transport. Another major challenge is to achieve interoperability among national rail systems across Europe.

The transition towards highly automated and digitalised railway operations introduces a wide range of new onboard functions that are increasingly dependent on sensor- and software-intensive solutions for an automated processing of driver tasks, requiring runtime environments with high-performance computing and networking capabilities.

To address the needs of Automated Train Operation (ATO) up to Grade of Automation 4 (GoA4, unattended driving), new onboard architectures have been recently defined and harmonised on European level; particularly in the Shift2Rail projects Connecta [4] and X2Rail4 [10], as well as in the OCORA project [6]; see Figure 1 for an example.

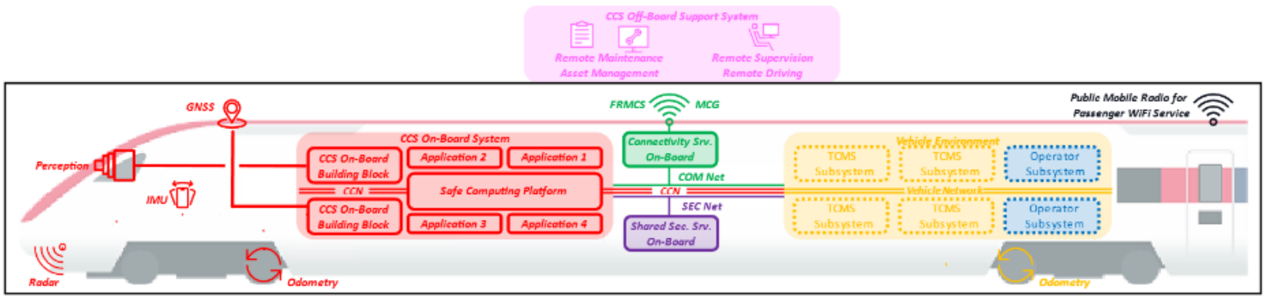


Figure 1. OCORA CCS-OB architecture, long term vision for 2030+ [6]

Onboard functions responsible for automated driving and automated processing are mainly part of the CCS-OB domain, which is shown in the left part of Figure 1 and highlighted in red. Prominent examples, currently specified and harmonised in the rail sector, are the following; see also [10]:

- Functions for Automated Train Operation (ATO) over ETCS
- Functions for Advanced Train Protection (ATP), based on ETCS level 3
- Functions for Perception (PER) and Localization (LOC)
- Functions for Automated Processing (APM)

Use cases for automated processing of driver tasks in ATO contexts have been defined and documented in [11], [12].

Automated processing of driver tasks in ATO contexts includes the ability to access and report the health and performance of relevant vehicle functions from TCMS domain, which is shown in the right part of Figure 1, highlighted in yellow.

In unattended driving scenarios, automated processing further includes the ability to perform operator-specific procedures for fault management and troubleshooting, normally assigned to the train driver or staff. Please note, that operator-specific fault management and troubleshooting is here regarded as complementary to vehicle-specific fault detection and safe reaction.

In the remainder of this document, the terms *fault management* and *automated troubleshooting* (in context of ATO and unattended driving scenarios) will be solely used to refer to operator-specific procedures, performed outside of TCMS domain, and being complementary to vehicle-specifc fault detection and safe reaction.

Operator-specific procedures for fault management and automated troubleshooting rely on the TCMS to provide vehicle status information through a common interface. This status information is here supposed to be provided as *datasets,* containing either system-specific data (e.g. Doors or HVAC status) or vehicle-global data (operating mode, location, speed, etc.).

Specific *health- and performance indicators* on system and vehicle level can be then extracted and aggregated from these TCMS datasets by operator-specific applications, that will feed decision making algorithms for fault management and automated troubleshooting on operator side.

To cope with the task of extracting and aggregating operator-specific health- and performance indicators from TCMS datasets, needed for fault management and automated troubleshooting in unattended driving scenarios, the concept of *agents* has been intoduced in DBS GoA4 onboard architecture [1].

DBS GoA4 onboard architecture defines the *DIA-VEC function* as an agent, which is responsible for providing operator-specific health- and performance indicators about vehicle functions to non-safety-critical applications responsible for fault management and automated troubleshooting on operator side. To do so, DIA-VEC relies on the TCMS to provide vehicle status information through a common interface.

The voluntary Eurospec TCMS DataService specification [1] (TCMS_DS) proposes a centralised onboard data service as means to distribute onboard information from TCMS domain to operator-specific applications. The TCMS_DS scope currently includes several operator-specific datasets defined at the vehicle level and vehicle system level. This data scope is supposed to be configurable and extendable to adapt to different use cases, respectively needs from different client applications, as illustrated by Figure 2 below.

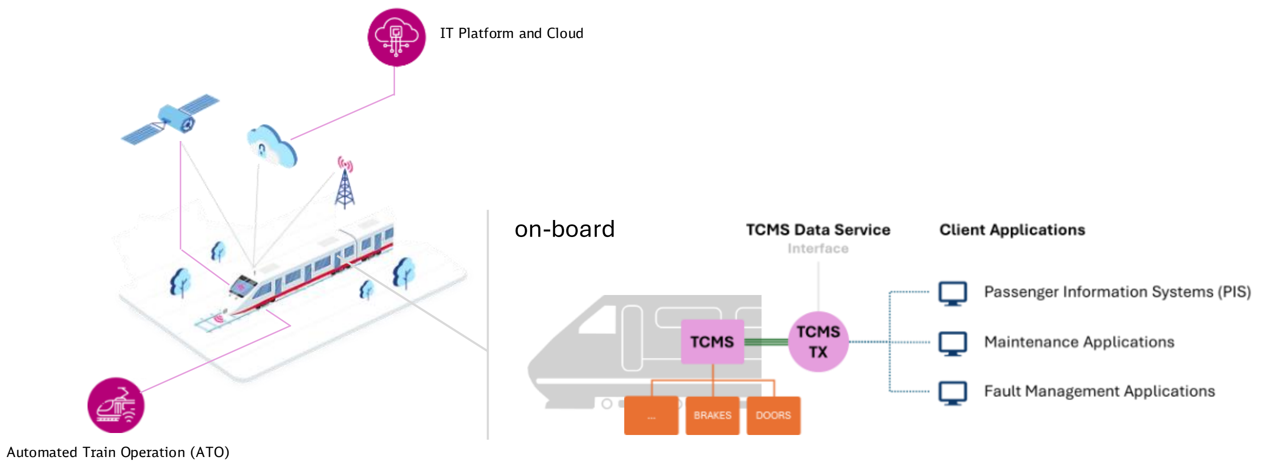


Figure 2. Rolling stock with TCMS DataService

In this work, the TCMS DataService is considered as means for DIA-VEC to receive status information from vehicle systems through a common interface, where the focus is on *availability* (health) and *quality* (performance) of vehicle functions relevant for ATO use cases.

[1] Which is derived from X2Rail-4 specification [10] and consistent to OCORA system architecture [37].

Examples of vehicle functions in scope of DIA-VEC are Brakes and Doors. These vehicle functions were selected as representative use cases for TCMS_DS concept evaluation and formalization towards fault management and automated troubleshooting in ATO contexts. Potential health- and performance indicators for Doors function, needed as input for operator-specific fault management and automated troubleshooting procedures are: individual *door states* (health), and derived *boarding service* quality (performance) per car; see also 3.2.2. Similarly, potential health- and performance indicators for Brakes function are: *SB* and *PB* states (service and parking brakes), and derived *SB* performance; see also section 3.2.1.

When it comes to the software design and implementation of DIA-VEC function, specifications for TCMS_DS datasets and service APIs will be required, in particular those for Brakes and Doors.

In [1], dataset definitions for TCMS_DS Doors are quite informal and imprecise, and dataset definitions for TCMS_DS Brakes are completely missing. Furthermore, there is no proper specification of the service APIs by which client applications like DIA-VEC agents can access these datasets.

The observations from previous paragraph are the main drivers that motivate the work conducted in Task 31.2, namely: the application of formal modelling methods and model transformation techniques to provide concise and validatable specifications of TCMS_DS APIs and hosted vehicle datasets. This topic is covered next in chapter 3, where the formalization approach is introduced and applied for Brakes and Doors use cases.

3 FORMALIZATION OF TCMS_DS DATASETS AND SERVICES

This chapter describes the formal TCMS_DS dataset modelling and service API specification approach, which has been established in this work. The context that was defined to delineate the scope for concept evaluation and validation activities is shown in Figure 3 below.

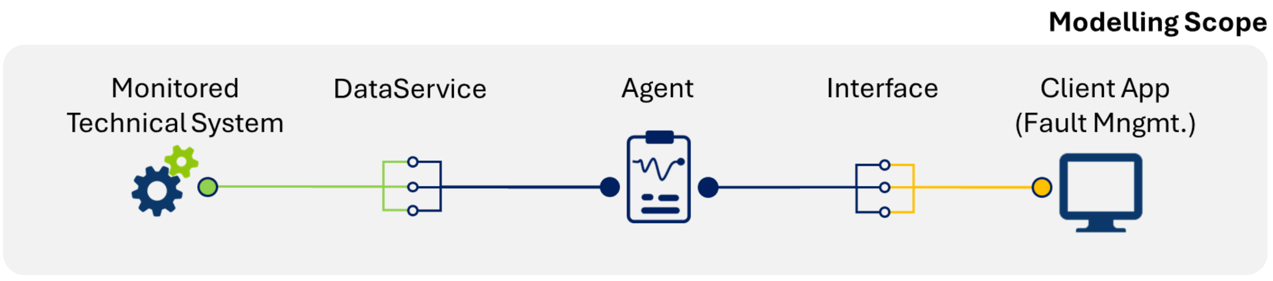


Figure 3. Modelling scope for concept evaluation and validation of the pursued TCMS_DS dataset modelling and service specification approach

The main benefits of applying formal methods in this context are (1) the ability to provide unambiguous definitions (models) for TCMS_DS datasets and (2) the ability to provide machine-readable descriptions for associated service APIs.

The current set of TCMS_DS requirements from [1] is lacking formality when it comes to the definition of data services for relevant vehicle systems from TCMS domain. So, TCMS_DS implementations from different vendors will be likely diverging and/or incompatible [2], which is, from an operator's point of view, an issue for client application development and variant-management (when handling TCMS_DS datasets from different vehicle types).

The use of concise and implementation-agnostic notations (i.e. formal languages), in combination with standardised exchange formats for TCMS_DS dataset models and service API specs, obviously facilitates direct and consistent utilisation of TCMS_DS modelling artifacts in software development projects.

The formalisation approach, presented in this chapter, endorses model-based software design (MBD) and development processes, that use a "spec-first" design-by-contract [3] workflow for the implementation of service-oriented application component interfaces.

The ultimate goal pursued here is to demonstrate the automation of certain coding steps in TCMS_DS software development process [4] by transforming manually authored TCMS_DS dataset models into (1) executable *object models* suitable for an automated datastore setup, and (2) generating machine readable capability descriptions for service APIs, that can be used further to generate executable stubs for both TCMS_DS server and TCMS_DS client applications.

[2] From the perspective of TCMS_DS client applications from operator functional zones.
[3] See https://en.wikipedia.org/wiki/Design_by_contract
[4] According to V-model for classical (non-agile) software development processes, coding steps are part of the implementation phase at the bottom of the "V"; see also: https://en.wikipedia.org/wiki/V-model_(software_development). In real-world software development projects, the implementation phase is often too long. The availability of reusable or auto-generated software libraries for component development helps to improve this.

The general software design objectives addressed by the pursued modelling approach are:

- **Functional transparency**: Provide a concise and unambiguous specification of TCMS_DS datasets and service APIs in a machine-readable format
- **Interoperability**: Facilitate integration across multiple vehicle platforms, suppliers, and operator-specific configurations
- **Scalability and modularity**: Enable operators to adopt only relevant subsets of the specification in line with operational and cost constraints

3.1 USED MODELLING METHODS AND TECHNOLOGIES

In this work, the formal modelling of TCMS_DS datasets is realised by using the YANG 1.1 data modelling language, which is specified as RFC 7950 [13]. The formal definition and exchange of TCMS_DS service API for modelled datasets is realised by using the OpenAPI 3.0 standard from [14].

According to [13], YANG is a textual specification language originally designed to model data for the NETCONF protocol [5]. A YANG model comprises one or more *modules* and defines hierarchies of data that can be used in NETCONF-like operations, including configuration, state data, RPCs, and notifications.

According to [14], OpenAPI (OAS) defines a textual document format, enabling the formal specification of RESTful APIs for accessing web services. OpenAPI specs provide human-readable service capability descriptions that can be shared between computers as *.yaml or *.json files. Sharing of OpenAPI specs is usually accomplished by adding a special GET endpoint to the service API, enabling client applications to fetch the OpenAPI spec of provided services from remote nodes.[6]

Shared OpenAPI documents can be processed and interpreted by browser-based client software to interact with remote nodes hosting the specified services. The amount of service-specific implementation logic needed on client side is minimal in this case. Furthermore, shared OpenAPI documents can be used (1) by documentation generation tools to generate navigable API views in browser clients, or (2) by code generation tools to generate API-specific stubs in various programming languages that can be either used as boilerplate in server- or client software development projects, or integrated into tools used for API testing.

Please note, the decision for using YANG 1.1 and OpenAPI 3.0 as modelling technologies in this work has been deliberately made; see also Table 4 from ch. 4.4. The aim of this work is to show how to formalize TCMS_DS datasets and to automate their deployments. We were not aiming for technology and/or toolchain standardization.

3.1.1 Technical overview on the YANG 1.1 data modelling language

YANG is a hierarchical data modelling language that defines data nodes using a tree-like structure. It supports modular design, reusability, and formal validation through constraints and typedefs. Key elements include:

- `container`: A grouping of related data nodes

[5] See: RFC 6241, https://datatracker.ietf.org/doc/html/rfc6241
[6] This is often referred as *self-documenting* service API.

- `leaf / leaf-list`: Basic data fields or enumerations
- `list`: Repeated structures, similar to arrays or tables
- `typedef`: Custom data types for reusability and clarity
- `grouping / uses`: Modular composition of reusable model components
- `rpc`: Remote callable functions with input/output definitions
- `notification`: Asynchronous event signalling (e.g. fault triggers)

Example Specification Snippet:

```
container tcmsds-brakes {
  list brake-unit {
    key "id";
    leaf id {
      type string;
    }
    leaf status {
      type enumeration {
        enum OK;
        enum ERROR;
        enum MAINTENANCE;
      }
    }
  }
}
```

YANG is supported by a diverse tool ecosystem, enabling seamless integration into modern development workflows; see Table 1 below. Such tools accelerate the development cycle and ensure compliance with system requirements defined at the data model level.

Table 1. YANG Ecosystem and Tool Support

Tool	**Purpose**
pyang	Static analysis, linting, code generation
yanglint	Validation and instance checking
OpenAPI generators	RESTful API export from YANG models
NETCONF/RESTCONF clients	Runtime interface testing
Visual Studio Code plugins	Syntax highlighting, schema navigation

3.1.2 Technical overview on the OpenAPI 3.x specification format

OpenAPI 3.1.1 is the latest version of the OpenAPI Specification (OAS), providing a standardised and widely adopted format for describing RESTful APIs. An OpenAPI definition is typically provided in YAML or JSON format and consists of the following key sections:

- `openapi:` – Declares the OAS version (e.g., 3.1.1)
- `info:` – Metadata about the API (title, version, contact, license)
- `servers:` – Base URLs for accessible endpoints
- `paths:` – The heart of the API: lists operations grouped by resource paths (e.g., `/doors`, `/brake/status`)
- `components:` – Reusable elements such as schemas, parameters, request bodies, and responses
- `security:` – Authentication mechanisms (e.g., API keys, OAuth2, JWT)
- `tags:` – Logical grouping of operations for better documentation structure

OpenAPI 3.1.x adopts full alignment with JSON Schema 2020-12, offering enhanced schema reuse, standard `$ref` resolution, and composability across models and components.

Example Specification Snippet:

```yaml
openapi: 3.1.1
info:
  title: TCMS Brake Service
  version: 1.0.0
paths:
  /brake/status:
    get:
      summary: Retrieve current brake status
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/BrakeStatus'
components:
  schemas:
    BrakeStatus:
      type: object
      properties:
        unitId:
          type: string
        status:
          type: string
    enum: [OK, ERROR, MAINTENANCE]
```

The use of OpenAPI for designing and testing service-oriented software component interfaces is supported by a broad range of available tools, which makes it an ideal fit for use an agile and DevOps-based development environments [7]; see Table 2 for some examples.

Table 2. OpenAPI tooling ecosystem

Tool / Platform	Purpose
Swagger UI / ReDoc	Interactive API documentation
OpenAPI Generator	Code generation (clients, servers, stubs) in >50 languages
Swagger Editor / Stoplight	Browser-based and desktop OpenAPI authoring
Postman / Insomnia	API testing based on OpenAPI contracts
Prism	Mock server for API simulation
Spectral / Linters	Specification validation and quality assurance
CI tools (e.g. GitHub Actions, GitLab CI)	OpenAPI linting, bundling, and changelog generation

3.2 YANG MODELS FOR BRAKES AND DOORS DATASETS

As proof of concept for using YANG as formal modelling language for the specification of TCMS_DS datasets and corresponding service extensions, Brakes and Doors systems have been selected as use cases.

Modelling of relevant system resources and corresponding status information has been done here according to the needs for fault management and automated troubleshooting in ATO GoA3/4 contexts, and the use of this information in operator-specific agent applications for the purpose of health- and performance indicator extraction.

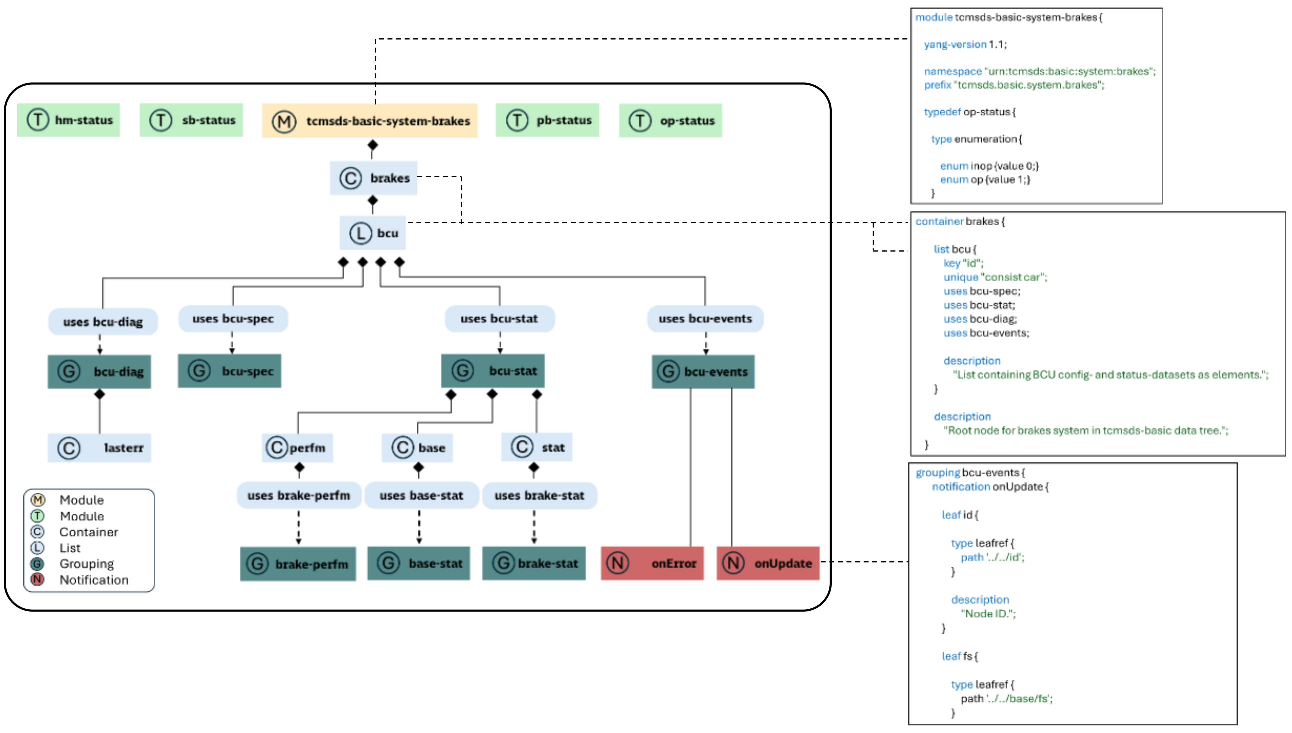
Please note, that modelling aspects related to data quality and data certainty are not considered in this work.

In the following subsection 3.2.1, the YANG model developed here for Brakes use case is explained in detail. The YANG model for Doors use case will be only briefly explained in subsection 3.2.2, since most of the concepts for resource and dataset modelling can be transferred from Brakes use case.

[7] Supporting traceability, testability, and automation across the entire API lifecycle

3.2.1 Brakes data model

Figure 4 provides a graphical representation of the YANG model defined for the Brakes use case.



```
module tcmsds-basic-system-brakes {

  yang-version 1.1;

  namespace "urn:tcmsds:basic:system:brakes";
  prefix "tcmsds.basic.system.brakes";

  typedef op-status {

    type enumeration {

      enum inop {value 0;}
      enum op {value 1;}
    }
```

```
container brakes {

    list bcu {
      key "id";
      unique "consist car";
      uses bcu-spec;
      uses bcu-stat;
      uses bcu-diag;
      uses bcu-events;

      description
        "List containing BCU config- and status-datasets as elements.";
    }

    description
      "Root node for brakes system in tcmsds-basic data tree.";
  }
```

```
grouping bcu-events {
    notification onUpdate {

      leaf id {

        type leafref {
          path '../../id';
        }

        description
          "Node ID.";
      }

      leaf fs {

        type leafref {
          path '../../base/fs';
        }
```

Figure 4. Graphical representation of the YANG data model for the Brakes use case

The data model for the Brakes system is defined as YANG module *tcmsds-basic-system-brakes*.

The module's root entity is *brakes,* which denotes the root node of the resulting data tree for representing and managing the Brakes system states.

The Brakes system states are modelled as a list of individual BCU states with associated datasets for functional status and error information. BCU states are represented in the data model as *bcu* entities.

The data model defines *brakes* as a YANG container with child element *bcu,* defined as YANG list.

This representation allows identifying *bcu* entities and associated datasets as YANG paths of the form */brakes/bcu[id]/,* where *id* is a unique identifier, e.g. GU1 or GU2.[8]

Functional status data associated with *bcu* entities is defined here as a YANG grouping *bcu-stat.*

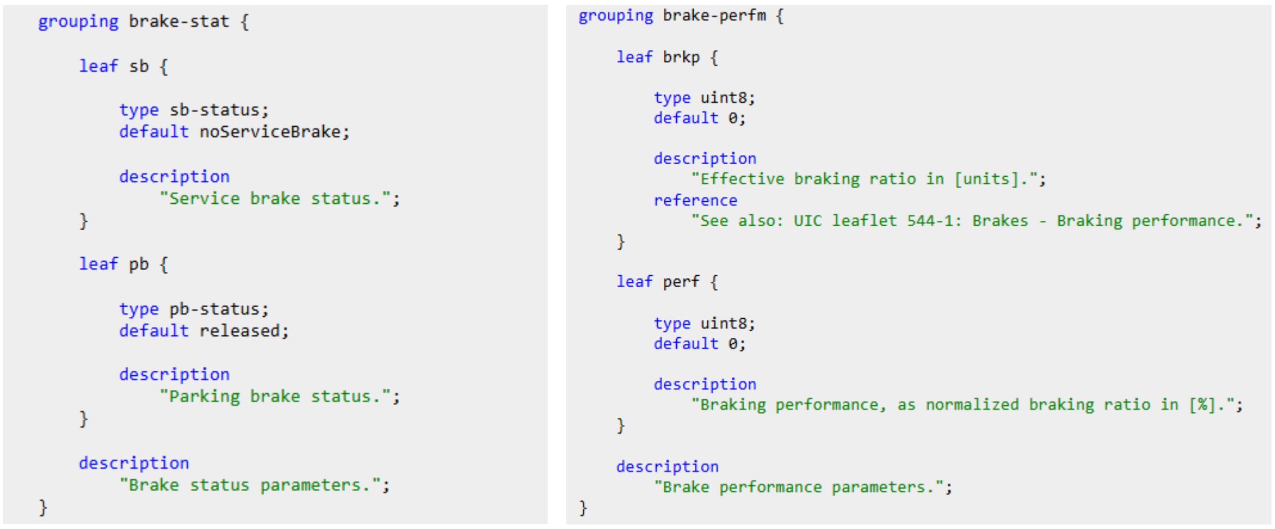
Error information associated with *bcu* entities is defined as YANG grouping *bcu-diag*.

YANG groupings have only syntactical relevance and no representation as nodes in the resulting data tree. In case of *bcu-stat,* the grouping is used to add model elements *base*, *stat,* and *perfm*, as *bcu* child nodes by writing *bcu* { … *uses bcu-stat ;* … *}*; see the second code snippet from Figure 4 above. Adding elements from *bcu-stat* as child nodes of *bcu* creates YANG paths of the form */brakes/bcu[id]/{base,stat,perfm}/* in the resulting data tree.

[8] GU (Gateway Unit) is a vendor-specific term used for BCUs in the vehicle which was used as baseline for system analysis and Brakes fault management in ATO GoA3/4 contexts.

In YANG data models, only the *leaf* elements define actual data points. *Non-leaf* elements are used for structuring and referencing data.

Figure 5 provides code snippets defining the leaf elements, respectively datapoints, for YANG paths */brakes/bcu[id]/stat/* and */brakes/bcu[id]/perfm/*.



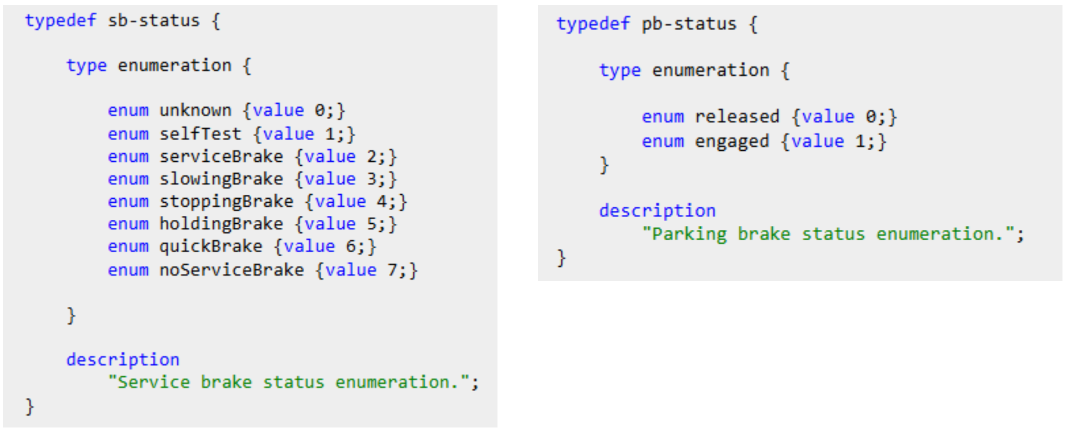
```
grouping brake-stat {

    leaf sb {

        type sb-status;
        default noServiceBrake;

        description
            "Service brake status.";
    }

    leaf pb {

        type pb-status;
        default released;

        description
            "Parking brake status.";
    }

    description
        "Brake status parameters.";
}
```

```
grouping brake-perfm {

    leaf brkp {

        type uint8;
        default 0;

        description
            "Effective braking ratio in [units].";
        reference
            "See also: UIC leaflet 544-1: Brakes - Braking performance.";
    }

    leaf perf {

        type uint8;
        default 0;

        description
            "Braking performance, as normalized braking ratio in [%].";
    }

    description
        "Brake performance parameters.";
}
```

Figure 5. YANG fragments defining data points for brake state and brake performance

By using definitions from Figure 4 and Figure 5, *sb* and *pb* element are added as leafs of *bcu/stat* and can be referenced by their YANG paths as */brakes/bcu[id]/stat/sb* respectively */brakes/bcu[id]/stat/pb.* These paths represent the datapoints for service brake and parking brake states from *bcu* entities.

The data model for Brakes system defines enumeration types *sb-status* and *pb-status* to define discrete value ranges for *bcu/stat/sb* (service brake state) and *bcu/stat/pb* (parking brake state). In the resulting data tree, *sb* and *pb* values can be then assigned and compared using their enum keys (e.g. *bcu/stat/sb == slowingBrake*).[9] The YANG fragments for enumeration typedefs *sb-status* and *pb-status* are shown in Figure 6 below.



```
typedef sb-status {

    type enumeration {

        enum unknown {value 0;}
        enum selfTest {value 1;}
        enum serviceBrake {value 2;}
        enum slowingBrake {value 3;}
        enum stoppingBrake {value 4;}
        enum holdingBrake {value 5;}
        enum quickBrake {value 6;}
        enum noServiceBrake {value 7;}

    }

    description
        "Service brake status enumeration.";
}
```

```
typedef pb-status {

    type enumeration {

        enum released {value 0;}
        enum engaged {value 1;}
    }

    description
        "Parking brake status enumeration.";
}
```

Figure 6. YANG enumerations sb-status and pb-status from the Brakes data model

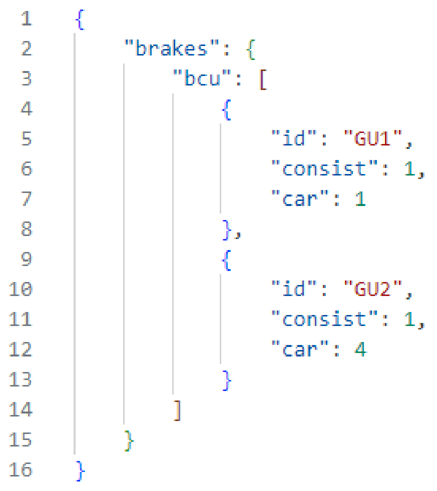
[9] Enumerations for service brake and parking brake states were deliberately defined here for modelling purposes. So far there is no TCN application profile defined for Brakes system in IEC 61375-2-4, or UIC 556.

YANG models are formal specifications of data tree structures representing a hierarchy of data objects that will be created inside a datastore when the data model is instantiated for a particular system configuration. Instantiation of YANG models as runtime datastore for the purpose of managing system entity states cannot be done directly, but requires model transformation into a corresponding *object model* containing executable code that provides data structures and functions for data tree setup and accessing data points.[10] This topic is covered in section 3.3.

Using the automation workflow from section 3.3 for transforming the Brakes data model into a corresponding object model and integrating this object model into TCMS_DS prototype allows instantiation of the Brakes data model as *brakes* data tree in TCMS_DS *datastore*. See also chapter 4 describing the TCMS_DS prototype that was developed and used in this work for validation of data models.

The instantiation of the *brakes* data tree in the TCMS_DS *datastore* is accomplished by sending a HTTP POST request to the *load_state* endpoint from the TCMS_DS backend interface, which uses the default configuration defined for validation and testing Brakes use case; see also section 4.5.

In TCMS_DS prototype, default configurations for Brakes and Doors use cases are realised as JSON files defining the reset default values for the setup of *brakes* and *doors* datastores during TCMS_DS startup. Figure 7 shows the default configuration used to setup the *brakes* datastore.



```
{
    "brakes": {
        "bcu": [
            {
                "id": "GU1",
                "consist": 1,
                "car": 1
            },
            {
                "id": "GU2",
                "consist": 1,
                "car": 4
            }
        ]
    }
}
```

Figure 7. Default configuration for the brakes datastore in the TCMS_DS prototype

By comparing the contents of Figure 7 and Figure 4, one can see how the *brakes* and *bcu* entities from the Brakes data model will be represented as data tree in the TCMS_DS datastore.

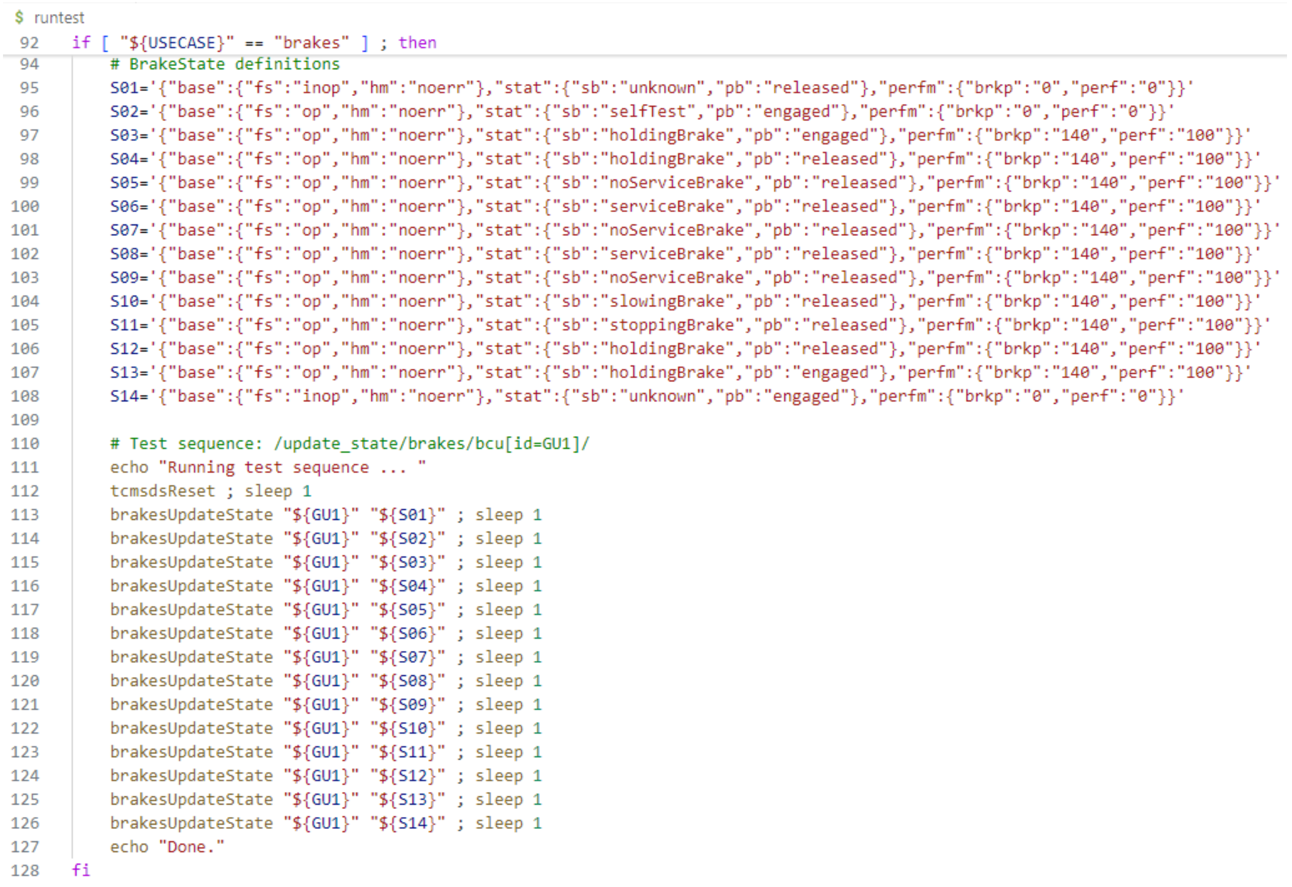
Model configurations used to instantiate YANG models for the datastore setup usually do not contain value assignments for leaf elements with default values. Furthermore, only those leaf elements declared as *mandatory* and *config* in the YANG model must be explicitly initialised through model configurations when no default values are provided.

For the Brakes system, the YANG model defines the grouping *bcu-spec*, which is used as child of *bcu;* see the third code snippet from Figure 4. *bcu-spec* contains the mandatory config leafs *id*, *consist,* and *car*, defining configuration parameters for *bcu* entities. Since these configuration

[10] This is similar to the principle of document templates. The template (model) defines the document structure and content formats/constraints. Several document instances can be then instantiated and used.

parameters have no assigned default value, they must be assigned through the model configuration, as shown in Figure 7.

After model instantiation and datastore setup in the TCMS_DS prototype, *bcu* nodes from *brakes* data tree can be updated via HTTP/REST request POST /update_state/, passing either of *bcu-stat, bcu-perfm* or *bcu-diag* datasets as payload data (in application/json format). Figure 8 provides a code snippet, taken from a test script used for module testing of the TCMS_DS prototype that implements update requests for */brakes/bcu[GU1]/stat/* and */brakes/bcu[GU1]/perfm/* nodes.



```
$ runtest
if [ "${USECASE}" == "brakes" ] ; then
    # BrakeState definitions
    S01='{"base":{"fs":"inop","hm":"noerr"},"stat":{"sb":"unknown","pb":"released"},"perfm":{"brkp":"0","perf":"0"}}'
    S02='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"selfTest","pb":"engaged"},"perfm":{"brkp":"0","perf":"0"}}'
    S03='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"holdingBrake","pb":"engaged"},"perfm":{"brkp":"140","perf":"100"}}'
    S04='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"holdingBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S05='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"noServiceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S06='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"serviceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S07='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"noServiceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S08='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"serviceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S09='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"noServiceBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S10='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"slowingBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S11='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"stoppingBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S12='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"holdingBrake","pb":"released"},"perfm":{"brkp":"140","perf":"100"}}'
    S13='{"base":{"fs":"op","hm":"noerr"},"stat":{"sb":"holdingBrake","pb":"engaged"},"perfm":{"brkp":"140","perf":"100"}}'
    S14='{"base":{"fs":"inop","hm":"noerr"},"stat":{"sb":"unknown","pb":"engaged"},"perfm":{"brkp":"0","perf":"0"}}'

    # Test sequence: /update_state/brakes/bcu[id=GU1]/
    echo "Running test sequence ... "
    tcmsdsReset ; sleep 1
    brakesUpdateState "${GU1}" "${S01}" ; sleep 1
    brakesUpdateState "${GU1}" "${S02}" ; sleep 1
    brakesUpdateState "${GU1}" "${S03}" ; sleep 1
    brakesUpdateState "${GU1}" "${S04}" ; sleep 1
    brakesUpdateState "${GU1}" "${S05}" ; sleep 1
    brakesUpdateState "${GU1}" "${S06}" ; sleep 1
    brakesUpdateState "${GU1}" "${S07}" ; sleep 1
    brakesUpdateState "${GU1}" "${S08}" ; sleep 1
    brakesUpdateState "${GU1}" "${S09}" ; sleep 1
    brakesUpdateState "${GU1}" "${S10}" ; sleep 1
    brakesUpdateState "${GU1}" "${S11}" ; sleep 1
    brakesUpdateState "${GU1}" "${S12}" ; sleep 1
    brakesUpdateState "${GU1}" "${S13}" ; sleep 1
    brakesUpdateState "${GU1}" "${S14}" ; sleep 1
    echo "Done."
fi
```

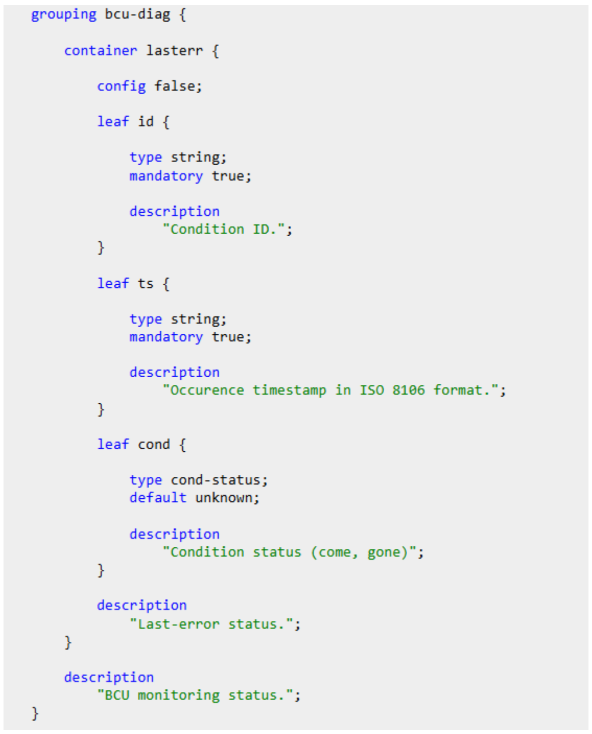
Figure 8. Examples of brakes/bcu updates with bcu-stat and bcu-perfm datasets as payload; taken from test script that implements module tests for the TCMS_DS prototype

Validation tests for the Brakes data model were conducted in lab environments using the TCMS_DS prototype and test scripts like the one from Figure 8. Status updates for simulated Brakes resources were distinguished as *status updates* and *error reports*.

Status updates for simulated Brakes resources were implemented as TCMS_DS update requests for configured *bcu* entities, containing *bcu-base, bcu-stat,* and *bcu-perfm* datasets as payload data; cf. Figure 8.

Error reports for simulated Brakes resources were implemented as TCMS_DS update requests for configured *bcu* entities, containing *bcu-diag* datasets as payload data.

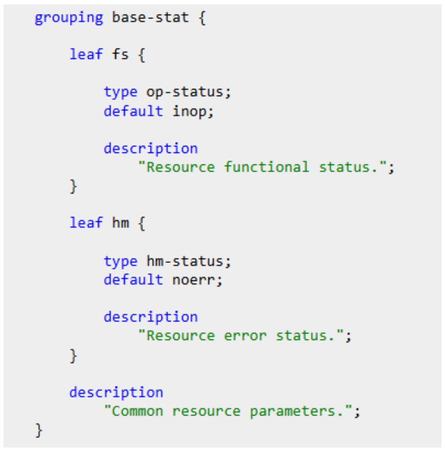
Conceptually, error reports send by system control units, such as BCUs (Brakes) or DCUs (Doors), are supposed here to indicate the monitoring status from these units in terms of a condition identifier, a condition status, and an occurrence timestamp. This monitoring status is represented in the Brakes data model as *bcu-diag* grouping, which is shown in Figure 9 below.[11]



```
grouping bcu-diag {

    container lasterr {

        config false;

        leaf id {

            type string;
            mandatory true;

            description
                "Condition ID.";
        }

        leaf ts {

            type string;
            mandatory true;

            description
                "Occurence timestamp in ISO 8106 format.";
        }

        leaf cond {

            type cond-status;
            default unknown;

            description
                "Condition status (come, gone)";
        }

        description
            "Last-error status.";
    }

    description
        "BCU monitoring status.";
}
```

Figure 9. bcu-diag grouping from the Brakes data model

In Brakes and Doors data models the top-level entities, such as /brakes/*bcu*, contain a *base-stat* grouping, which is used to model indicators for the *availability* and *health* of the represented function or resource. Figure 10 provides the YANG fragment which defines *base-stat*.



```
grouping base-stat {

    leaf fs {

        type op-status;
        default inop;

        description
            "Resource functional status.";
    }

    leaf hm {

        type hm-status;
        default noerr;

        description
            "Resource error status.";
    }

    description
        "Common resource parameters.";
}
```

Figure 10. base-stat grouping used in the Brakes and Doors data models

Figure 8 on p. 24 shows how availability and health indicators from *base-stat* are used in *bcu* status updates. Therein, *fs/inop* indicates that resource GU1 is down (inop) [12], while *fs/op* indicates that

[11] bcu-diag from Brakes data model

[12] This status set as default in TCMS_DS datastore, respectively *brakes* data tree, and will be normally updated to *fs/op* when *GU1* is powered up and ready. However, in case of failure, the loss of a GU1 function might be observed and reported by *GU2,* which is acting as redundant site that will take over in this case.

resource GU1 is up (powered-on). In the latter case, the base/*hm* value must be used for assessing the resource health; see below.

Table 3 lists the interpretations of possible value assignments for *fs* and *hm* in *base-stat* dataset, as used in simulations conducted for validation tests of TCMS_DS prototype and corresponding data models for Brakes and Doors use case.

Table 3. Interpretation of possible value assignments in base-stat dataset

base-stat	***fs***	***hm***
resource down	inop	N/A
resource good	op	noerr
resource bad	op	error

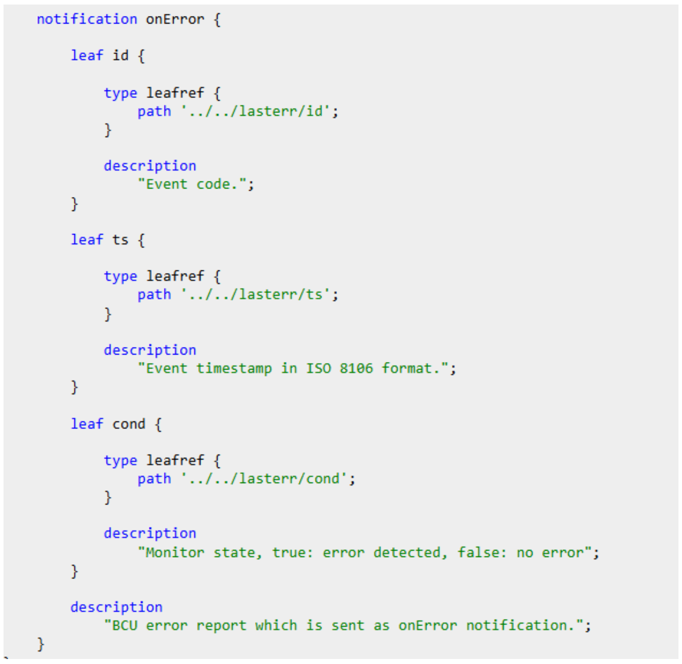
One possible way to see the underlying concept behind using the *base-stat* dataset for resource availability and health indication is to regard *fs* and *hm* as “power” and “self-test” signals driving a status LED on a dashboard. In this case, a possible colouring for *base-stat* assignments from Table 3 would be (in top-down order): *amber*, *green*, and *red*.[13]

The YANG data modelling language provides *rpc*, *action* and *notification* elements to specify remote procedure calls and notifications for network management protocols such as NETCONF.

The YANG models for Brakes and Doors define notification elements *onUpdate* and *onError* for specifying the contents of corresponding events to be triggered by the datastore upon detection of a state change or changed error condition in managed resources.

The *onUpdate* and *onError* notification elements from Brakes data model are shown in Figure 4 on p. 21 as “N”- labelled nodes from *bcu-events* grouping, which is included (used) by /brakes/*bcu* entities. The corresponding YANG fragment for *onError* is exemplarily shown in Figure 11 below.

[13] See https://monroeaerospace.com/blog/an-introduction-to-cockpit-warning-lights/



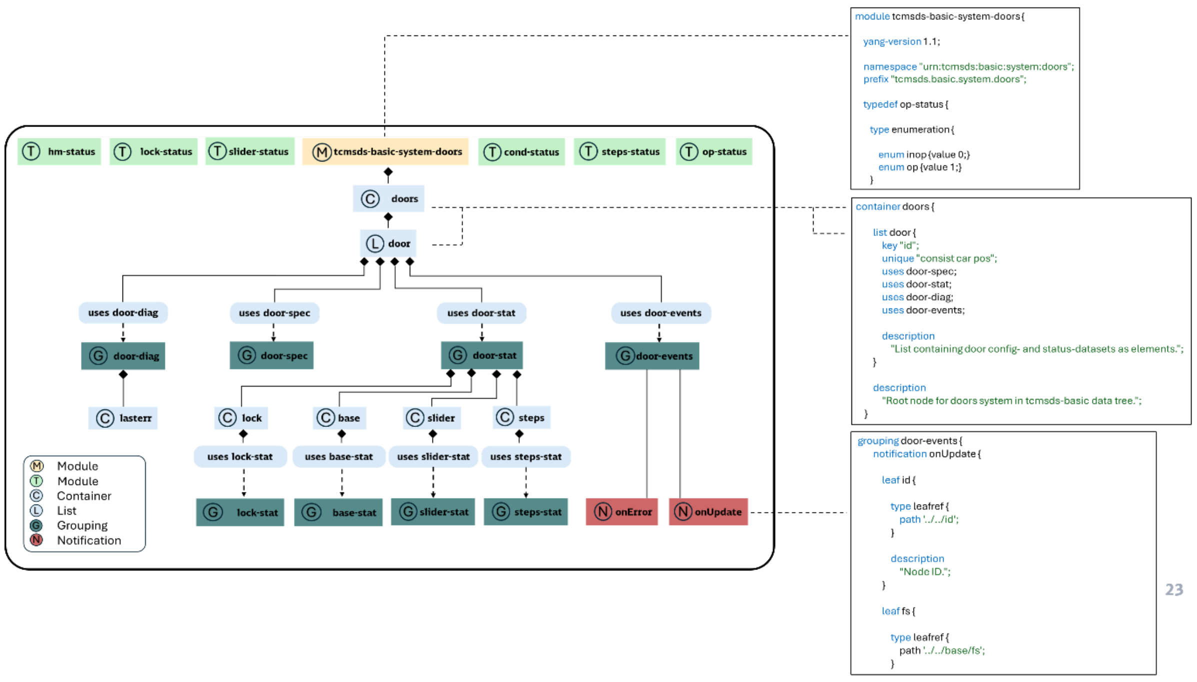
```
notification onError {

    leaf id {

        type leafref {
            path '../../lasterr/id';
        }

        description
            "Event code.";
    }

    leaf ts {

        type leafref {
            path '../../lasterr/ts';
        }

        description
            "Event timestamp in ISO 8106 format.";
    }

    leaf cond {

        type leafref {
            path '../../lasterr/cond';
        }

        description
            "Monitor state, true: error detected, false: no error";
    }

    description
        "BCU error report which is sent as onError notification.";
}
```

Figure 11. YANG fragment for onError notifications

When transforming YANG data models into executable structures, notification elements will be translated into corresponding datastore functions and service API descriptors, used to realize event-streams for managed resources that can be subscribed, for example, by agents. See section 3.3 for details on how the *onUpdate* and *onError* notifications from Brakes data model will be transformed into executable code for the purpose of automated integration and use in TCMS_DS prototype.

3.2.2 Doors data model

The second use case for the evaluation of the TCMS_DS concept and the validation of formal TCMS_DS dataset modelling methods pursued in this work is the Doors system of a regional passenger train. A graphical representation of the developed YANG model is shown in Figure 12.



```
module tcmsds-basic-system-doors {

  yang-version 1.1;

  namespace "urn:tcmsds:basic:system:doors";
  prefix "tcmsds.basic.system.doors";

  typedef op-status {

    type enumeration {

      enum inop {value 0;}
      enum op {value 1;}
    }
```

```
container doors {

    list door {
      key "id";
      unique "consist car pos";
      uses door-spec;
      uses door-stat;
      uses door-diag;
      uses door-events;

      description
        "List containing door config- and status-datasets as elements.";
    }

    description
      "Root node for doors system in tcmsds-basic data tree.";
  }
```

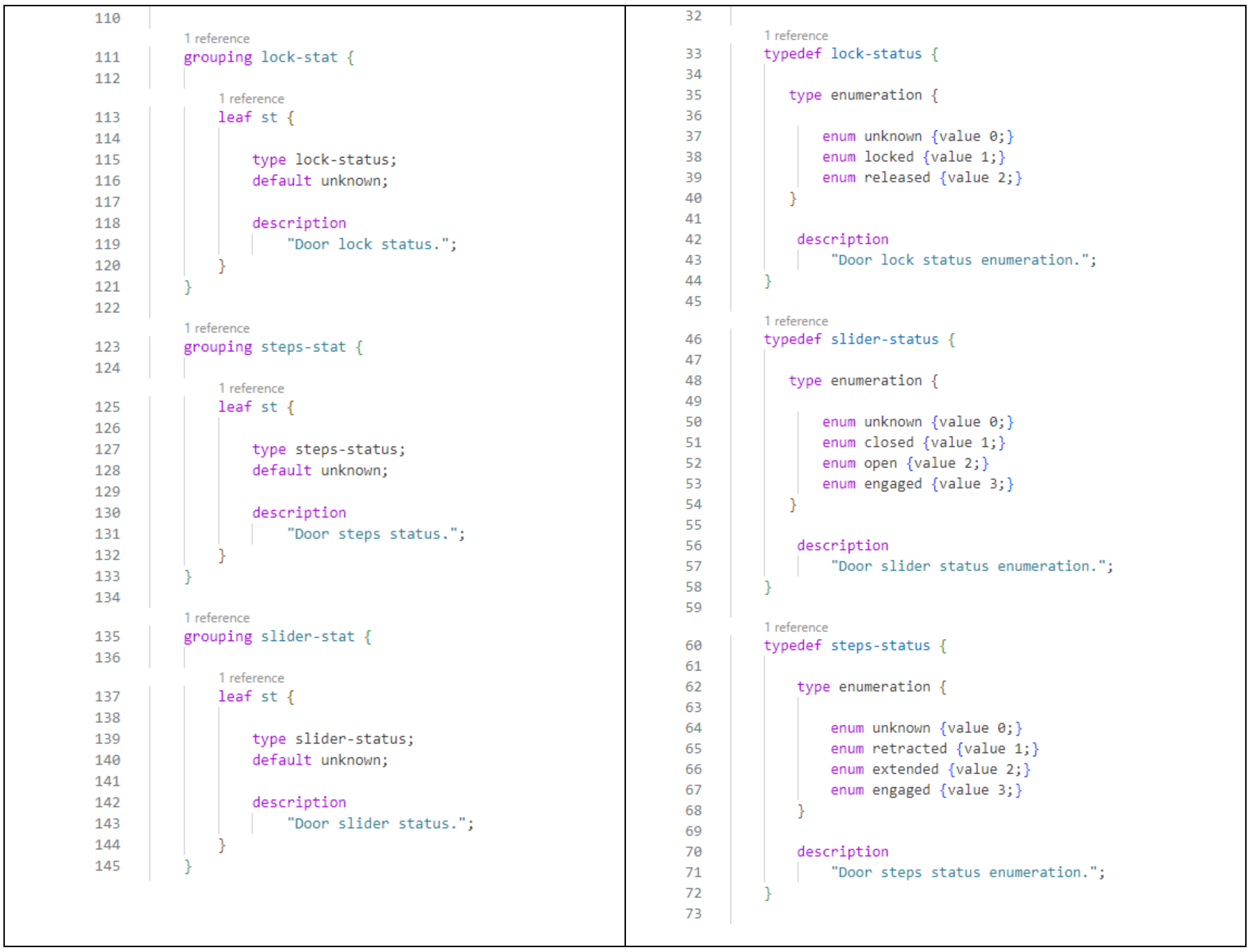
```
grouping door-events {
    notification onUpdate {

      leaf id {

        type leafref {
          path '../../id';
        }

        description
          "Node ID.";
      }

      leaf fs {

        type leafref {
          path '../../base/fs';
        }
```

Figure 12. Graphical representation of the YANG data model for the Doors use case

When comparing data models for Brakes and Doors systems (see Figure 4 on p. 21 and Figure 12), one can see that the same strategy for system abstraction and resource modelling has been applied. For the Doors use case, system resources were modelled as *doors/door* entities, which is a slightly more functional abstraction compared to the Brakes use case (where system resources were modelled as *brakes/bcu* entities).[14] For both, Brakes and Doors use cases, the modelled system resources are supposed to have an individual reporting of functional status updates and error reports (see below for further discussion).

Similar to *brakes/bcu* entities, the functional status for *doors/door* entities is modelled as a YANG grouping *door-stat* (which becomes the *door-stat* dataset in TCMS_DS datastore). The *door-stat* grouping comprises YANG container elements *base*, *lock, steps and slider*. The definition and meaning of a YANG *base* container is the same as for the Brakes use case; see Figure 10 on p. 25. YANG container elements *lock*, *steps* and *slider* represent actuation states of a single exterior door, defined according to YANG fragments shown in Figure 13 below.

[14] In case of the regional train type, which was used in this work as baseline for Brakes and Doors system analysis, at least each exterior door has a dedicated door control unit (DCU), so the main Doors system resources could have been alternatively modelled as *doors/dcu* entities, which would be closer to the level of abstraction used for Brakes use case.



```
1 reference
grouping lock-stat {

  1 reference
  leaf st {

    type lock-status;
    default unknown;

    description
      "Door lock status.";
  }
}

1 reference
grouping steps-stat {

  1 reference
  leaf st {

    type steps-status;
    default unknown;

    description
      "Door steps status.";
  }
}

1 reference
grouping slider-stat {

  1 reference
  leaf st {

    type slider-status;
    default unknown;

    description
      "Door slider status.";
  }
}
```

```
1 reference
typedef lock-status {

  type enumeration {

    enum unknown {value 0;}
    enum locked {value 1;}
    enum released {value 2;}
  }

  description
    "Door lock status enumeration.";
}

1 reference
typedef slider-status {

  type enumeration {

    enum unknown {value 0;}
    enum closed {value 1;}
    enum open {value 2;}
    enum engaged {value 3;}
  }

  description
    "Door slider status enumeration.";
}

1 reference
typedef steps-status {

  type enumeration {

    enum unknown {value 0;}
    enum retracted {value 1;}
    enum extended {value 2;}
    enum engaged {value 3;}
  }

  description
    "Door steps status enumeration.";
}
```

Figure 13. Definition of exterior door actuation states for lock, steps and slider components from door-stat dataset defined for doors/door entities in TCMS_DS doors data tree

The thus modelled *door-stat* dataset for *doors/door* entities is consistent to requirements TCMS_DS.38, TCMS_DS.39, … ,TCMS_DS.43 from [1], p.19 and p.20.

The here made assumption that at each exterior door has a dedicated DCU, by which it is controlled, is consistent to the Doors system breakdown structure from [15], ch. 7.1.2.

For reporting exterior door states and errors from DCUs to TCMS, the distribution network structure from [15], ch. 7.1.2, Figure 22, is assumed to be in place, as well as the Doors system architecture from [15] Figure 23, where a hierarchy of *train DCU*, *consist DCUs* and door *DCUs* is used.[15] In this hierarchy, the TCMS is communicating with the *train DCU* (which is the Doors *function leader*), and the *train DCU* communicates with *consist DCUs* from passenger cars, which in turn communicate with their DCUs responsible for controlling car-local exterior doors.[16] The communication between *train DCU* and *consist DCUs* is realised through the standardised train backbone (ETB+TBNs) from [16]. The consist network technology used for communication between a *consist DCU* and its

[15] The Doors system architecture from [15], ch. 7.1.2 applies to consists (traction units, driving trailers, multiple units, and passenger coaches) that are equipped with a train backbone network in accordance with EN IEC 61375-1 and with exterior doors controlled remotely.

[16] In the Doors system architecture from [15], ch. 7.1.2, there is one dedicated *consist DCU* used per passenger car.

local DCUs for exterior doors is not standardized in [15] and left open for project-specific solutions (e.g. using MVB and/or ECN technologies).[17]

In case of the regional train type used in this work as baseline for the Doors system analysis, assumptions from the Doors application profile in [15], ch. 7.1, are mostly fulfilled. MVB is used as consist network technology for realizing the communication between the *consist DCU* and DCUs for controlling exterior doors.[18] The *train DCU* is realized there as CCU function hosted by TCMS. Communication between *consist DCU* and *train DCU* is realized via IP bus.

In this work we've assumed that the *train DCU* function will be in charge for sending exterior door state updates and error reports to TCMS_DS. This assumption is consistent with [15], ch. 7.1.2, Figure 23, where the *train DCU* is supposed to provide a central diagnostic interface for the Doors system towards TCMS.

3.3 Model Transformation and Automation Workflow

The rationale and benefit of formalising TCMS_DS datasets by means of YANG specifications, is to enable automated validation of data models and subsequent code generation of corresponding TCMS_DS extensions, for the purpose of an automated integration of modelled datasets as runtime services.

This section describes the method and automation workflow that was implemented and used in this work for transforming YANG models developed for Brakes and Doors use cases into executable structures for the TCMS_DS prototype (see chapter 4).

3.3.1 Model transformation toolchain

Figure 14 illustrates how YANG models are transformed and integrated into the TCMS_DS prototype.[19]

[17] See [36] for a good overview of recent communication network architectures used for mainline railways, with focus on Train Communication Network (TCN) standard defined by EN IEC 61375.
[18] Two of the DCUs (one per side) for controlling exterior doors have assigned the role of the *consist DCU* that sends door states of the local passenger car to central *train DCU* via TBN (TCN gateway) and ETB (IP bus).
[19] ArchiMate is used in the figure as modelling notation

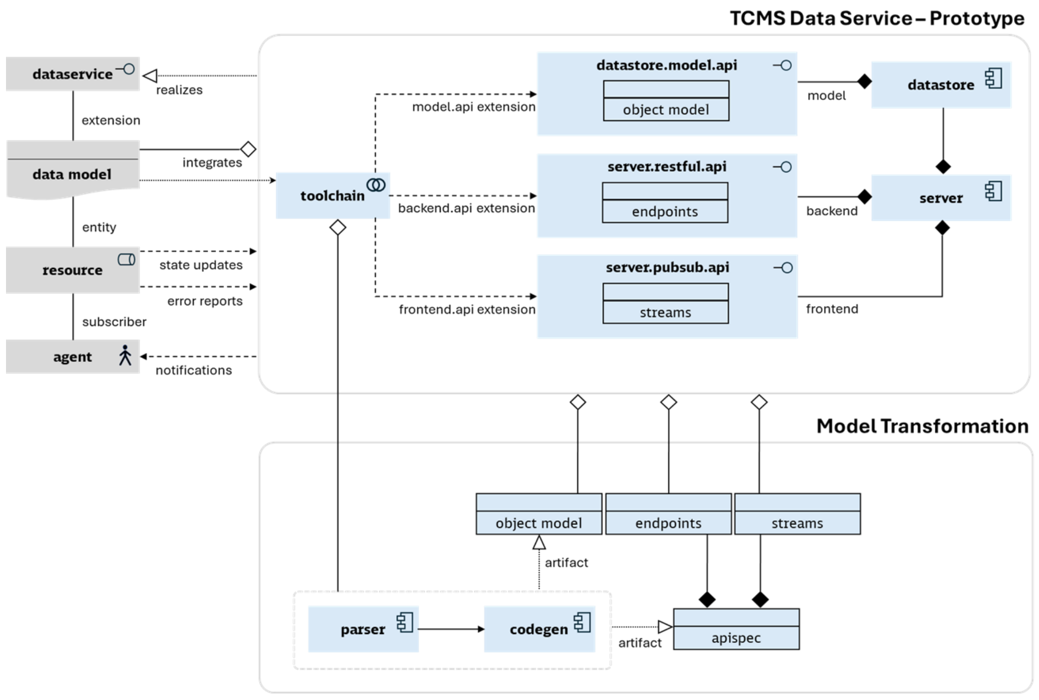


Figure 14. Transforming and integrating YANG models into the TCMS_DS prototype

The grey model elements on the left side of Figure 14 belong to the business view (or problem view), while blue model elements belong to the application model (architecture view), which was implemented in this work for the TCMS_DS prototype.

Transformation of YANG models into TCMS_DS extensions is implemented as a *toolchain* comprising a *parser* and a *codegen* component.

The *parser* component validates and interprets YANG 1.1 compliant data models for TCMS_DS datasets (e.g. Brake and Doors models), provided as textual specifications (*.yang files).

The *codegen* component implements the actual transformation of data model elements into application-specific code. Corresponding *codegen* functions will be called from the *parser* component as parser hooks.

The *codegen* component for generating TCMS_DS extensions has been developed in this work and successfully applied for Brakes and Doors data models. When applied to a particular data model, *codegen* outputs two artifacts; see also the *model transformation* box in Figure 14.

The first *codegen* artifact is the *object model* comprising data structures and functions that realise *model.api extensions* used to integrate *brakes* and *doors* datasets into TCMS_DS *datastore* components. Technically speaking, these are the YANG bindings with data management classes for *brakes* and *doors* datasets in the target programming language used for the TCMS_DS prototype.

The second *codegen* output is the *apispec* containing descriptors for *endpoints* and *streams* to be added to TCMS_DS *server.restful.api* (backend interface) and *server.pubsub.api* (frontend interface)*;* see chapter 4 for details.

The TCMS_DS toolchain uses *pyang* [17] as implementation for the YANG *parser* component. *pyang* is a mature and open source python3 package, that is also referred by vendors like Cisco or Nokia.[20] [21]

The implementation of the *codegen* component from the TCMS_DS toolchain is based on the *pyangbin*d plugin [18] and the *swagger* plugin [19] for the *pyang* parser. The first plugin realises the *object model* generation; the output is a single python3 module (*.py file), containing YANG bindings for *brakes* or *doors* datasets. The second plugin realises the *apispec* generation; the output is a single OpenAPI 3.0 module (*.json file), containing *endpoint* and *stream* descriptors for integration of *brakes* or *doors* datasets into to TCMS_DS *server.restful.api* and *server.pubsub.api.*

Figure 15 shows the YANG bindings generated by *codegen* for the Brakes data model, represented as UML class diagram.[22] For the TCMS_DS prototypical implementation, YANG bindings for Brakes and Doors data models are realised as python3 classes contained within a single python3 module (one module per data model), named *tcmsds_brakes.py* and *tcmsds_doors.py* respectively. These modules are then imported and used by the TCMS_DS *datastore* component, extending the TCMS_DS *datastore.model.api* by corresponding entry points (i.e. bindings for data tree root classes) for managing Brakes and Doors states; cf. Figure 14.

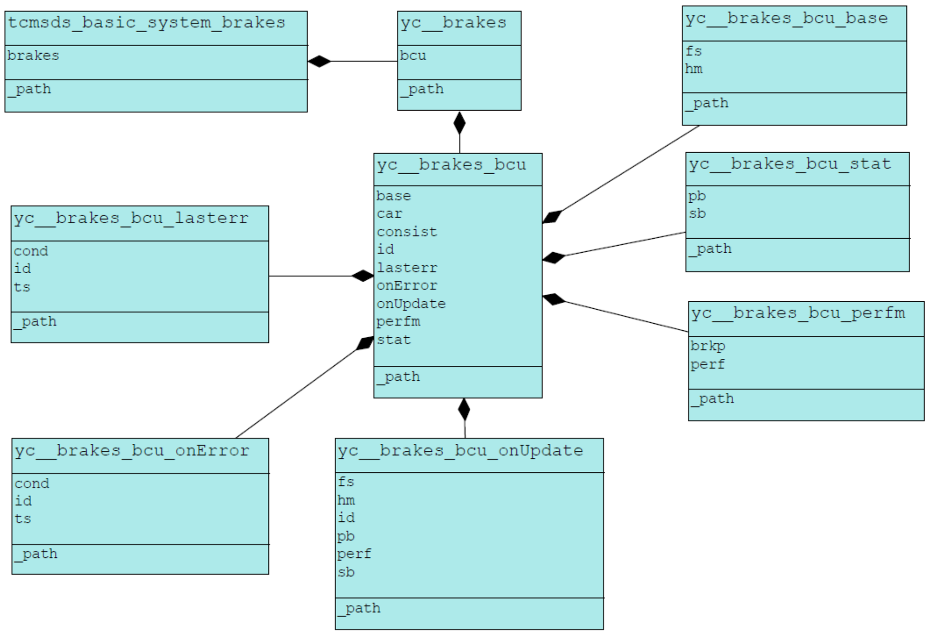


Figure 15. YANG bindings generated for the Brakes data model (as UML class diagram)

The *pyangbind* parser plugin used to transform YANG models for Brakes and Doors into executable code (denoted as *object model*), operates as follows; see [18] for details.

- YANG modules for Brakes and Doors are transformed into corresponding top-level classes named <*module_name*>, where <*module_name*> is either *tcmsds_basic_system_brakes* or *tcmsds_basic_system_doors*. These top-level classes provide the entry points for Brakes and Doors data trees in TCMS_DS *datastore*.model.api

[20] https://developer.cisco.com/docs/nx-os/model-driven-programmability-with-yang/#the-rise-of-network-automation
[21] https://network.developer.nokia.com/sr/learn/yang/manipulating-yang/
[22] Pynsource (https://github.com/abulka/pynsource) was used here for reverse engineering of python3 code as UML class diagrams.

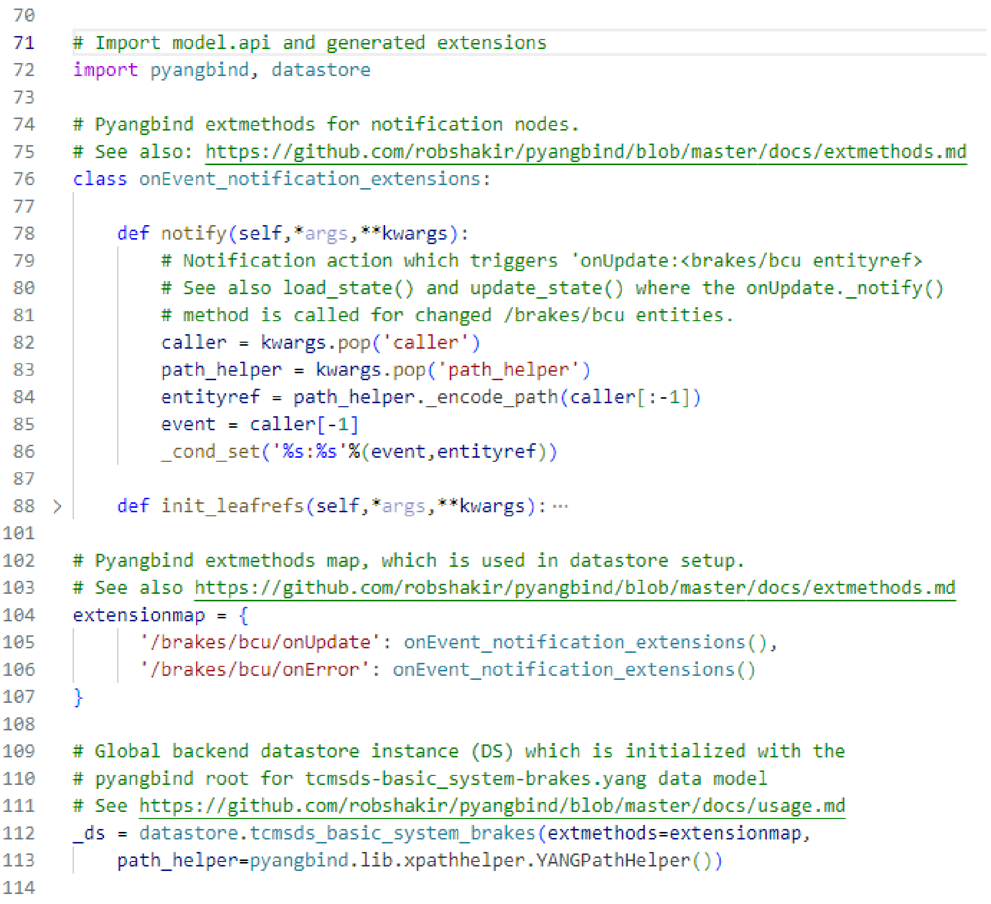
- Non-leaf elements defined by YANG modules are transformed into corresponding python3 classes named *yc_<path-name>*, where *yc_* prefix means "yang class" and *<path-name>* suffix denotes the YANG path of the model element, e.g. */brakes* or */brakes/bcu*, with "/" replaced by "_"
- The data tree structure of the YANG models is reflected in the transformed python3 bindings through composition and nesting of *yc*-childs via corresponding member variables in *yc*-parents. See for example, *yc__brakes.bcu* member variable which holds at runtime one or more *yc__brakes_bcu* objects, depending on system configuration
- Leaf-nodes defined by the YANG modules are transformed into member variables of parent *yc* objects. These member variables have simple python data types, e.g. *int* or *str*, that correspond to assigned YANG built-in types from the data model, e.g. *int32* or *string*
- Access to nested elements of *yc* objects, e.g. access to *yc__brakes.bcu* member, is provided through dedicated accessor and mutator methods (generated by *pyangbind* plugin per *yc* object and contained member variable). For brevity, generated accessor and mutator methods were omitted in the UML class diagram from Figure 15
- In addition to member-specific accessor and mutator methods, all generated *yc* objects include a set of generic methods useful for data tree initialization, -traversal and -change detection; see also [18] docs/generic_methods.md
- Notification elements from the YANG modules are transformed by *pyangbind* plugin into *yc* objects as well; see, for example, *yc__brakes_bcu_onUpdates* and *yc_brakes_bcu_onError* from Figure 15. When using the plugin option "--use-extmethods", additional methods will be added to generated YANG bindings, allowing to add external, user-defined functions to specific nodes in the data tree that can be invoked from those nodes as so called *extmethods*. Such *extmethods* are used as means for implementing notification services in the *datastore* component of TCMS_DS prototype; see also section 4.2, Figure 30, *datastore* service *notify state changes*

3.3.2 Integration of YANG bindings (object model)

The integration of generated YANG bindings (referred to as *object model* in Figure 14) for Brakes and Doors data models into the *datastore* component of the TCMS_DS prototype is realised as follows.

For now, YANG bindings produced by *codegen* as either *tcmsds_brakes.py* module (for Brakes data model) or *tcmsds_doors.py* module (for Doors data model) are, respectively, used directly as the *datastore* component in the TMCS_DS prototype. Integration and use of both Doors and Brakes bindings as single datastore is planned for future revisions of the TCMS_DS prototype software. For the initial validation and evaluation of TCMS_DS formalisation concepts, as described in section 4.5, it was preferred to keep things simple and have separate TCMS_DS instances running per use case.

Figure 16 shows how generated YANG bindings for Brakes data model are integrated in the current TCMS_DS prototype.



```
# Import model.api and generated extensions
import pyangbind, datastore

# Pyangbind extmethods for notification nodes.
# See also: https://github.com/robshakir/pyangbind/blob/master/docs/extmethods.md
class onEvent_notification_extensions:

    def notify(self,*args,**kwargs):
        # Notification action which triggers 'onUpdate:<brakes/bcu entityref>
        # See also load_state() and update_state() where the onUpdate._notify()
        # method is called for changed /brakes/bcu entities.
        caller = kwargs.pop('caller')
        path_helper = kwargs.pop('path_helper')
        entityref = path_helper._encode_path(caller[:-1])
        event = caller[-1]
        _cond_set('%s:%s'%(event,entityref))

    def init_leafrefs(self,*args,**kwargs):···

# Pyangbind extmethods map, which is used in datastore setup.
# See also https://github.com/robshakir/pyangbind/blob/master/docs/extmethods.md
extensionmap = {
      '/brakes/bcu/onUpdate': onEvent_notification_extensions(),
      '/brakes/bcu/onError': onEvent_notification_extensions()
}

# Global backend datastore instance (DS) which is initialized with the
# pyangbind root for tcmsds-basic_system-brakes.yang data model
# See https://github.com/robshakir/pyangbind/blob/master/docs/usage.md
_ds = datastore.tcmsds_basic_system_brakes(extmethods=extensionmap,
    path_helper=pyangbind.lib.xpathhelper.YANGPathHelper())
```

Figure 16. Code snippet from the TCMS_DS prototype module server/backend_brakes.py, containing the code lines that deal with integration of generated datastore extensions

For use cases Brakes and Doors, the TCMS_DS prototype software currently contains two separate python3 modules, named *server/backend_brakes.py* and *server/backend_doors.py*. These modules are almost identical up to some lines of code specific for the integration of *brakes* and *doors* extensions [23]; for example, the code lines defining the *extensionmap* containing use case specific paths for notification nodes to which custom *onEvent_notification_extension* functions shall be added, or the code lines for the construction of use case specific datastore objects for *brakes* and *doors* data tree.

Consider the python3 code snippet from Figure 16. The first section imports the *pyangbind* package and *datastore* module, where the *pyangbind* package provides the *datastore.model.api* from Figure 14, and the *datastore* module provides use case specific *model.api* extensions; see Figure 14 as well. The second section from Figure 16 defines the custom notification handler to be used by the *extensionmap* object, which is defined in the third section of Figure 16. The last section from Figure 16 creates the data tree root node, in this case the root node for the *brakes* data tree. The arguments passed to the data tree constructor are references to *extensionmap* and *xpathhelper* objects, where the latter object implements the mapping of YANG paths to datastore nodes; see also [18], docs/docs/xpathhelper.md.

Figure 17 on p. 36 contains code which is used for the *brakes* data tree initialisation in the TCMS_DS prototype module *server/backend_brakes.py*. Up to the use case specific node references at the

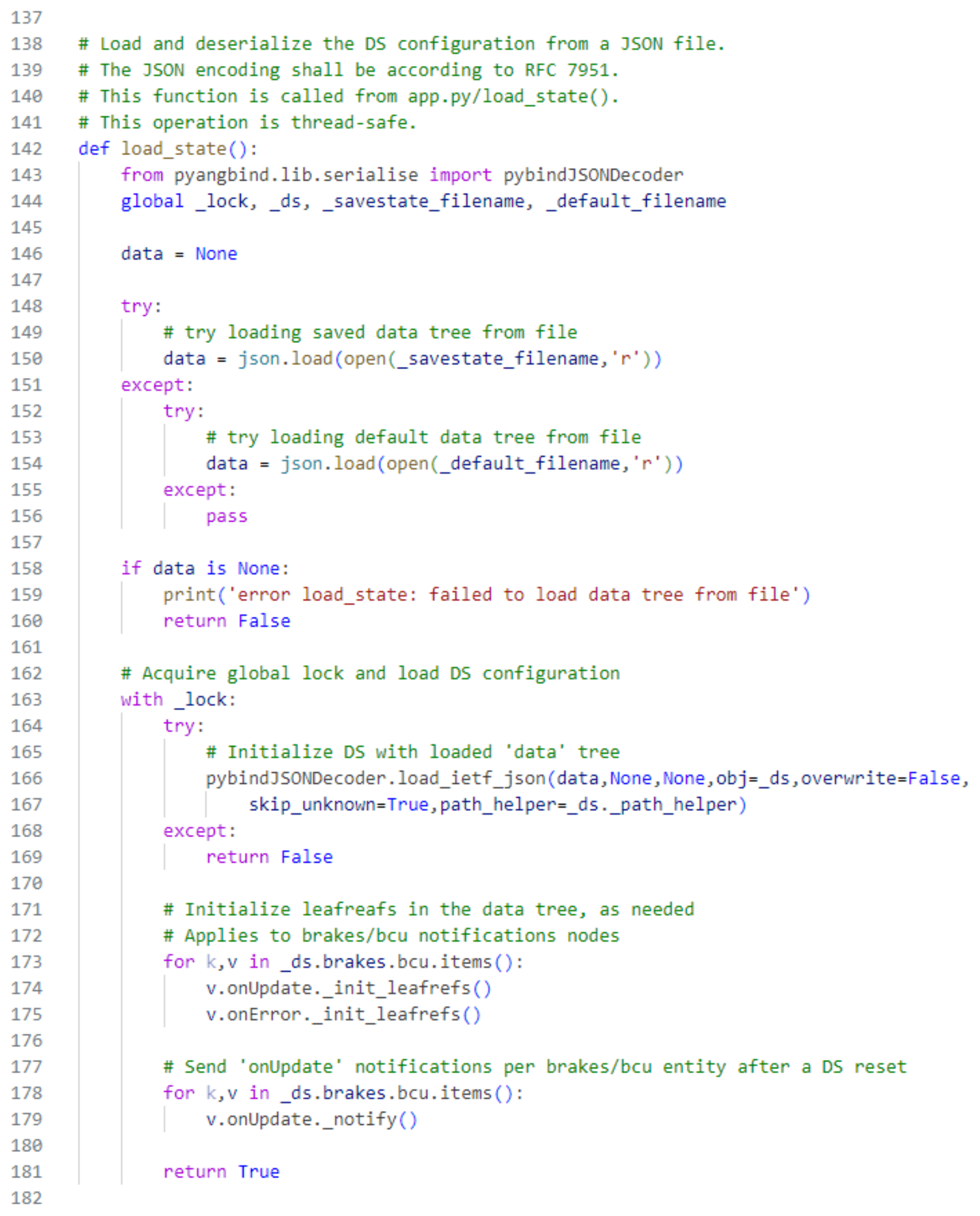
[23] These specifics will be subject to consolidation, which is planned for future revisions of TCMS_DS prototype.

bottom of Figure 17, the same code is used for initialisation of the *doors* data tree in the *server/backend_doors.py* module.

As explained in section 3.2, TCMS_DS default configurations for Brakes and Doors use cases are realised as JSON files defining the reset default values for datastore initialisation during the TCMS_DS startup. Looking at the first code section from Figure 17, one can see how datastore configurations will be loaded from JSON files, when the TCMS_DS backend service *load_state* is called. If no saved state exists, then the default configuration will be loaded as a YANG data tree in JSON format; see Figure 7 on p. 23, containing the default configuration used in TCMS_DS prototype for *brakes* data tree setup.

For serialising/deserialising YANG data trees to/from JSON files, the *pyangbind* package provides helper functions implemented in the *pybindJSONDecoder* module; see [18], docs/serialisation.md for details. Function py*pybindJSONDecoder.load_ietf_json* is used in Figure 17 to deserialise JSON string *data* (that was read from JSON file) as *brakes* data tree, and to insert the *brakes* data tree into *_ds* datastore object.

After initialising the TCMS_DS *datastore* with saved or default states, *onUpdate* notifications for connected TCMS_DS subscribers will be triggered; see the last code section from Figure 17 for how this is done after loading *brakes* data tree. As it was already mentioned above, the TCMS_DS prototype uses the *pyangbind extmethods* interface to inject custom functions into generated YANG bindings that can be called at runtime for specific nodes from hosted data trees; see Figure 16 for how *extmethods* interface is configured in TCMS_DS prototype for notification nodes *brakes/bcu/onUpdate* and *brakes/bcu/onError*.

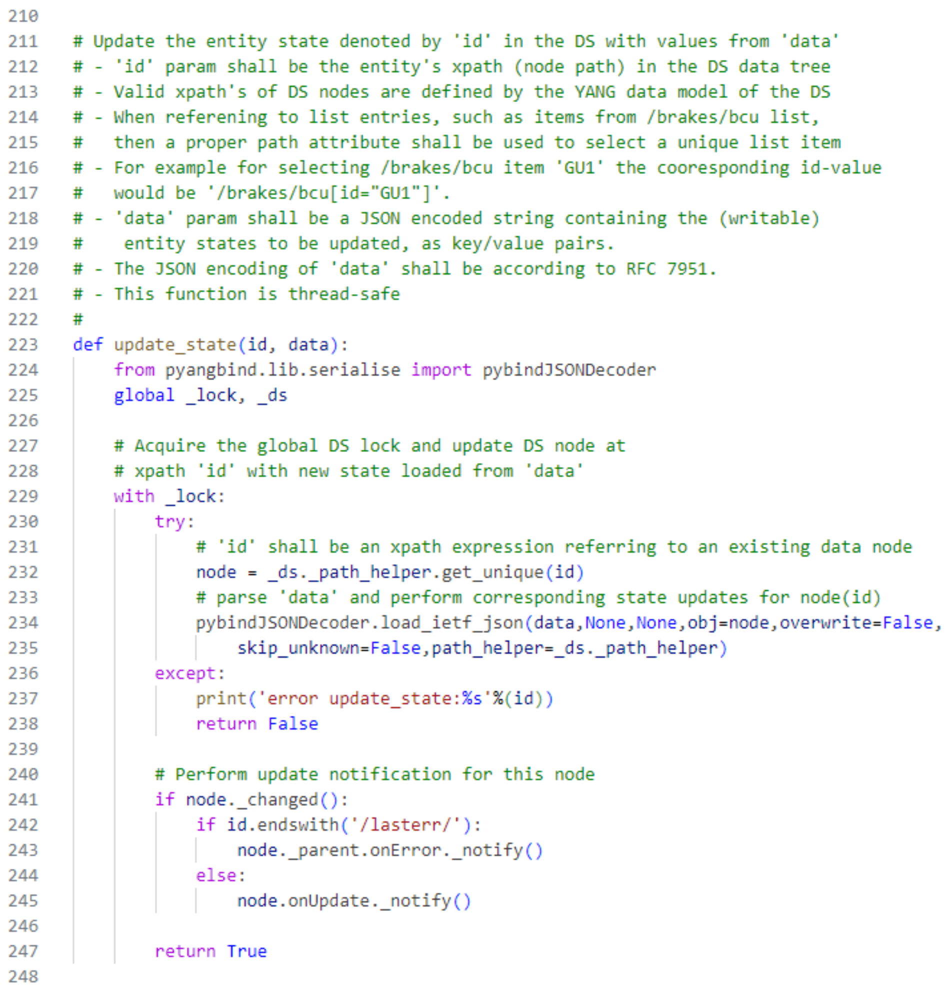


```python

# Load and deserialize the DS configuration from a JSON file.
# The JSON encoding shall be according to RFC 7951.
# This function is called from app.py/load_state().
# This operation is thread-safe.
def load_state():
    from pyangbind.lib.serialise import pybindJSONDecoder
    global _lock, _ds, _savestate_filename, _default_filename

    data = None

    try:
        # try loading saved data tree from file
        data = json.load(open(_savestate_filename,'r'))
    except:
        try:
            # try loading default data tree from file
            data = json.load(open(_default_filename,'r'))
        except:
            pass

    if data is None:
        print('error load_state: failed to load data tree from file')
        return False

    # Acquire global lock and load DS configuration
    with _lock:
        try:
            # Initialize DS with loaded 'data' tree
            pybindJSONDecoder.load_ietf_json(data,None,None,obj=_ds,overwrite=False,
                skip_unknown=True,path_helper=_ds._path_helper)
        except:
            return False

        # Initialize leafreafs in the data tree, as needed
        # Applies to brakes/bcu notifications nodes
        for k,v in _ds.brakes.bcu.items():
            v.onUpdate._init_leafrefs()
            v.onError._init_leafrefs()

        # Send 'onUpdate' notifications per brakes/bcu entity after a DS reset
        for k,v in _ds.brakes.bcu.items():
            v.onUpdate._notify()

        return True

```

Figure 17. Code snippet from the TCMS_DS prototype module server/backend_brakes.py, containing the code lines that deal with initialisation of generated datastore extensions

So far, we have shown how generated YANG bindings for *brakes* and *doors* data models are integrated as TCMS_DS *datastore* extensions, and how these bindings are initialised in the TCMS_DS *datastore* component using corresponding default configurations. What remains here is to show how the state of individual nodes from *brakes* and/or *doors* data tree can be updated in the TCMS_DS prototype, and how connected subscribers will be informed through corresponding notifications.

Figure 18 on p. 37 contains the code which is used for updating individual *brakes/bcu* nodes from the *brakes* data tree in the TCMS_DS prototype module *server/backend_brakes.py*.



```python

# Update the entity state denoted by 'id' in the DS with values from 'data'
# - 'id' param shall be the entity's xpath (node path) in the DS data tree
# - Valid xpath's of DS nodes are defined by the YANG data model of the DS
# - When referening to list entries, such as items from /brakes/bcu list,
#   then a proper path attribute shall be used to select a unique list item
# - For example for selecting /brakes/bcu item 'GU1' the cooresponding id-value
#   would be '/brakes/bcu[id="GU1"]'.
# - 'data' param shall be a JSON encoded string containing the (writable)
#    entity states to be updated, as key/value pairs.
# - The JSON encoding of 'data' shall be according to RFC 7951.
# - This function is thread-safe
#
def update_state(id, data):
    from pyangbind.lib.serialise import pybindJSONDecoder
    global _lock, _ds

    # Acquire the global DS lock and update DS node at
    # xpath 'id' with new state loaded from 'data'
    with _lock:
        try:
            # 'id' shall be an xpath expression referring to an existing data node
            node = _ds._path_helper.get_unique(id)
            # parse 'data' and perform corresponding state updates for node(id)
            pybindJSONDecoder.load_ietf_json(data,None,None,obj=node,overwrite=False,
                skip_unknown=False,path_helper=_ds._path_helper)
        except:
            print('error update_state:%s'%(id))
            return False

        # Perform update notification for this node
        if node._changed():
            if id.endswith('/lasterr/'):
                node._parent.onError._notify()
            else:
                node.onUpdate._notify()

        return True

```

Figure 18. Code snippet from TCMS_DS prototype module server/backend_brakes.py, dealing with update of individual nodes from generated datastore extensions

The TCMS_DS prototype provides a HTTP/REST-based *backend* interface with service endpoint */update_state* for receiving state updates and error reports from (simulated) system resources; see sections 4.2 and 4.5 for details. For the Brakes use case, system resources are modelled as *brakes/bcu* entities in the corresponding YANG data model; cf. Figure 4 on p. 21. State updates for *brakes/bcu* entities will be received by the TCMS_DS prototype as POST requests to the *backend* interface with service URI */update_state/brakes/bcu[id]/*. Similarly, error reports will be received as POST requests to the *backend* interface with service URI */update_state/brakes/bcu[id]/lasterr/*.

Depending on the request type (either state update or error report), the *update_state* payload contains either functional status information or error status information, reported for a particular system resource (respectively YANG model entity), such as *brakes/bcu[id=GU1]/*; see also section 3.2.1. Functional status information for *brakes/bcu* entities will be formatted according to the *bcu-stat* dataset from Brakes data model. Similarly, error status information for *brakes/bcu* entities will be formatted according to the *bcu-diag* dataset from Brakes data model. In both cases, the payload data (either *bcu-stat*, or *bcu-diag*) will be serialised as JSON string.[24]

Upon reception of an *update_state* request from a (simulated) system resource, the TCMS_DS prototype internally calls the corresponding request handler from the *server/backend* module. The

[24] Technically speaking, the Content-Type for POST *update_state* requests is required to be *application/json.*

implementation of the *update_state* request handler in the TCMS_DS prototype *server/backend_brakes.py* module is shown in Figure 18.

The processing of *update_state* requests for *brakes/bcu* entities, as implemented in the TCMS_DS prototype *server/backend_brakes.py* module, is as follows:

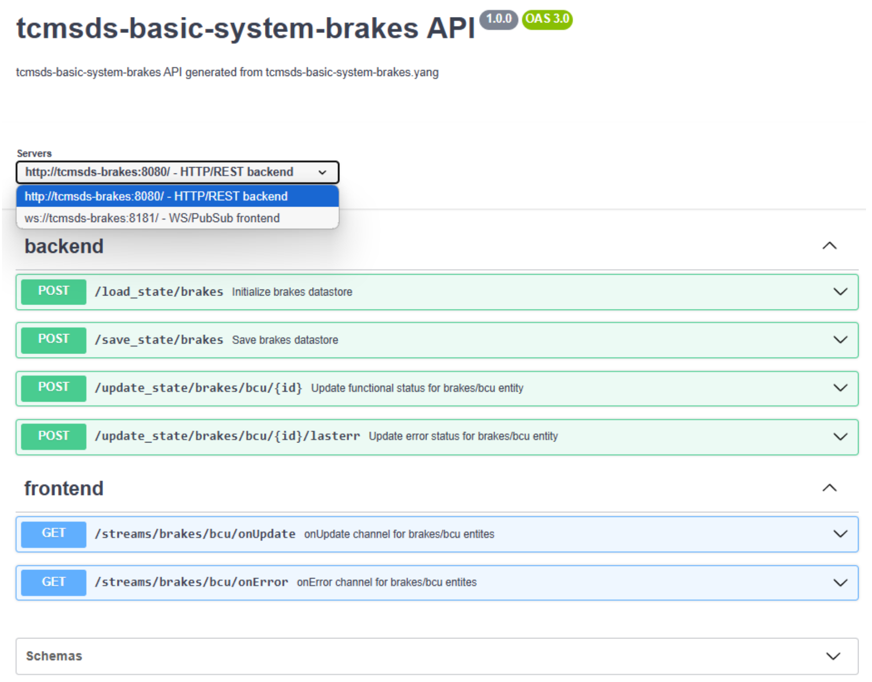
- Search the *brakes* data tree from TCMS_DS *datastore* to find the *brakes/bcu* node that uniquely matches the entity path that was passed in *update_state* request URI. In the code from Figure 18, the entity path will be passed to *update_state* request handler as *id* parameter. The *pyangbind xpathhelper* function (see [18], docs/xpathhelper.md) is used here to map entity paths (i.e. YANG paths, provided in xpath format [20]) to corresponding TCMS_DS *datastore* nodes-)
- If no matching *brakes/bcu* node could be found in the TCMS_DS *datastore*, an exception will be raised, and the processing terminates with an error
- Deserialise the *update_state* payload, which is passed as JSON string parameter *data*, and apply the deserialised payload data to the *brakes/bcu* node. Depending on whether *id* has the form */brakes/bcu/* or */brakes/bcu/lasterr/*, the deserialised payload data is either a *bcu-stat* dataset or a *bcu-diag* dataset, represented as *pyangbind* specific runtime object. *pybindJSONDecoder* module is used here to transparently perform the TCMS_DS *datastore* update, hiding the *pyangbind* specific details for data tree iteration and node updates
- After applying the update for the matching *brakes/bcu* node, check whether its state has changed. If so, trigger a corresponding update notification for that node, which causes the TCMS_DS PubSub frontend service to push either an *onUpdate* or *onError* message to connected subscribers; see sections 4.2 and 4.3 for details

3.3.3 Integration of API extensions (apispec)

This section describes the approach pursued in this work to realise an automated integration of YANG models on TCMS_DS interface level. With regard to the model transformation workflow from Figure 14 on p. 31, the goal is here to show how a model-generated *apispec* in OpenAPI format is used to setup corresponding *backend.api* and *frontend.api* extensions in TCMS_DS prototype.

Figure 19 provides a rendered view of the model-generated *apispec* for Brakes use case. As explained in section 3.3.1, *apispec* is an output artifact produced by *codegen*, which is part of the YANG model transformation toolchain developed in this work. The *codegen* component uses a modified version of the *pyang/swagger* plugin [19], which produces an OpenAPI 3.0 compliant specification of TCMS_DS backend and frontend service descriptors.

Generated *apispec* artifacts are text files in OpenAPI/JSON format, where service descriptors for TCMS_DS backend and frontend interface are contained as OpenAPI *path* elements. Figure 20 exemplarily shows the textual specification of backend service POST */update_state/brakes/bcu*, which was taken from the model-generated *apispec* for Brakes use case.



tcmsds-basic-system-brakes API 1.0.0 OAS 3.0

tcmsds-basic-system-brakes API generated from tcmsds-basic-system-brakes.yang

Servers

http://tcmsds-brakes:8080/ - HTTP/REST backend

http://tcmsds-brakes:8080/ - HTTP/REST backend

ws://tcmsds-brakes:8181/ - WS/PubSub frontend

backend

POST `/load_state/brakes` Initialize brakes datastore

POST `/save_state/brakes` Save brakes datastore

POST `/update_state/brakes/bcu/{id}` Update functional status for brakes/bcu entity

POST `/update_state/brakes/bcu/{id}/lasterr` Update error status for brakes/bcu entity

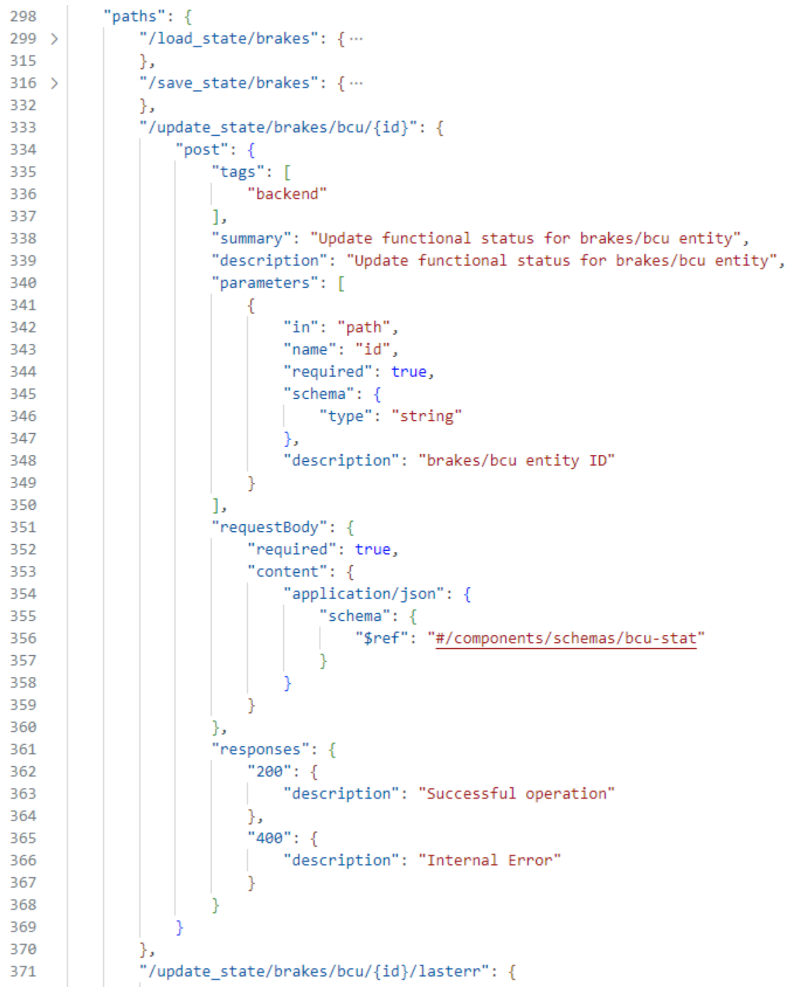
frontend

GET `/streams/brakes/bcu/onUpdate` onUpdate channel for brakes/bcu entites

GET `/streams/brakes/bcu/onError` onError channel for brakes/bcu entites

Schemas

Figure 19 Rendered view of generated TCMS_DS apispec artifact for Brakes use case

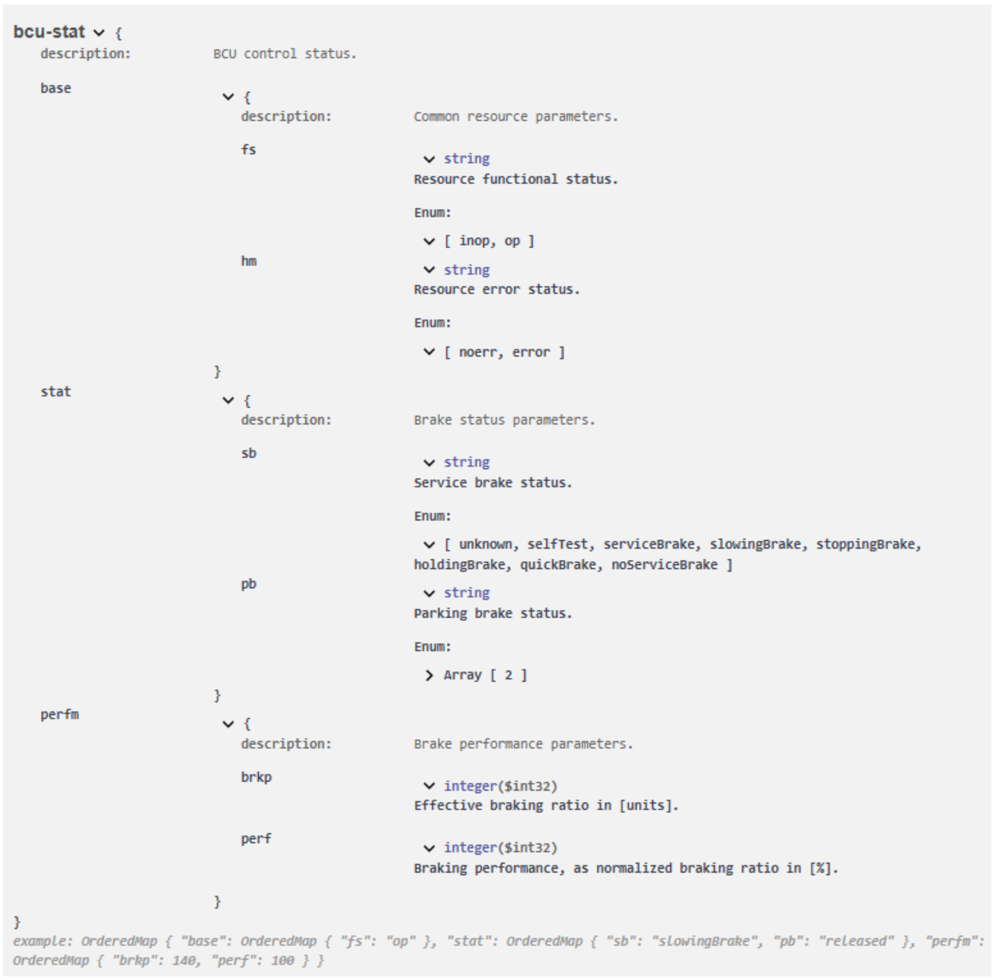


```
"paths": {
    "/load_state/brakes": {···
    },
    "/save_state/brakes": {···
    },
    "/update_state/brakes/bcu/{id}": {
        "post": {
            "tags": [
                "backend"
            ],
            "summary": "Update functional status for brakes/bcu entity",
            "description": "Update functional status for brakes/bcu entity",
            "parameters": [
                {
                    "in": "path",
                    "name": "id",
                    "required": true,
                    "schema": {
                        "type": "string"
                    },
                    "description": "brakes/bcu entity ID"
                }
            ],
            "requestBody": {
                "required": true,
                "content": {
                    "application/json": {
                        "schema": {
                            "$ref": "#/components/schemas/bcu-stat"
                        }
                    }
                }
            },
            "responses": {
                "200": {
                    "description": "Successful operation"
                },
                "400": {
                    "description": "Internal Error"
                }
            }
        }
    },
    "/update_state/brakes/bcu/{id}/lasterr": {
```

Figure 20. OpenAPI 3.0 path spec for the backend service POST /update_state/brakes/bcu

OpenAPI 3.0 path elements allow to formally define the required request parameters and payload format of service endpoints. For service descriptor POST */update_state/brakes/bcu* , the path spec from Figure 20 defines a required path-parameter, named *id,* as last part of the request URI, where *id* acts as placeholder for actual *brakes/bcu* entity IDs from the *brakes* datastore [25].

Furthermore, the path spec from Figure 20 defines the payload for POST */update_state/brakes/bcu* requests as content type *application/json*, with a content structure according to *bcu-stat* schema. The rendered view of *bcu-stat* schema definition from *apispec* is shown in Figure 21; compare this to the original YANG fragments from Figure 5, Figure 6 and Figure 10 in section 3.2.1 to see that the *bcu-stat* schema from *apispec* is content-wise identical to the *bcu-stat* grouping (respectively *bcu-stat* dataset) from the YANG model for the Brakes use case.



```
bcu-stat ⌄ {
   description:         BCU control status.
   base
                        ⌄ {
                        description:         Common resource parameters.
                        fs
                                             ⌄ string
                                             Resource functional status.
                                             Enum:
                                             ⌄ [ inop, op ]
                        hm
                                             ⌄ string
                                             Resource error status.
                                             Enum:
                                             ⌄ [ noerr, error ]
                        }
   stat
                        ⌄ {
                        description:         Brake status parameters.
                        sb
                                             ⌄ string
                                             Service brake status.
                                             Enum:
                                             ⌄ [ unknown, selfTest, serviceBrake, slowingBrake, stoppingBrake,
                                             holdingBrake, quickBrake, noServiceBrake ]
                        pb
                                             ⌄ string
                                             Parking brake status.
                                             Enum:
                                             > Array [ 2 ]
                        }
   perfm
                        ⌄ {
                        description:         Brake performance parameters.
                        brkp
                                             ⌄ integer($int32)
                                             Effective braking ratio in [units].
                        perf
                                             ⌄ integer($int32)
                                             Braking performance, as normalized braking ratio in [%].
                        }
}
example: OrderedMap { "base": OrderedMap { "fs": "op" }, "stat": OrderedMap { "sb": "slowingBrake", "pb": "released" }, "perfm":
OrderedMap { "brkp": 140, "perf": 100 } }
```

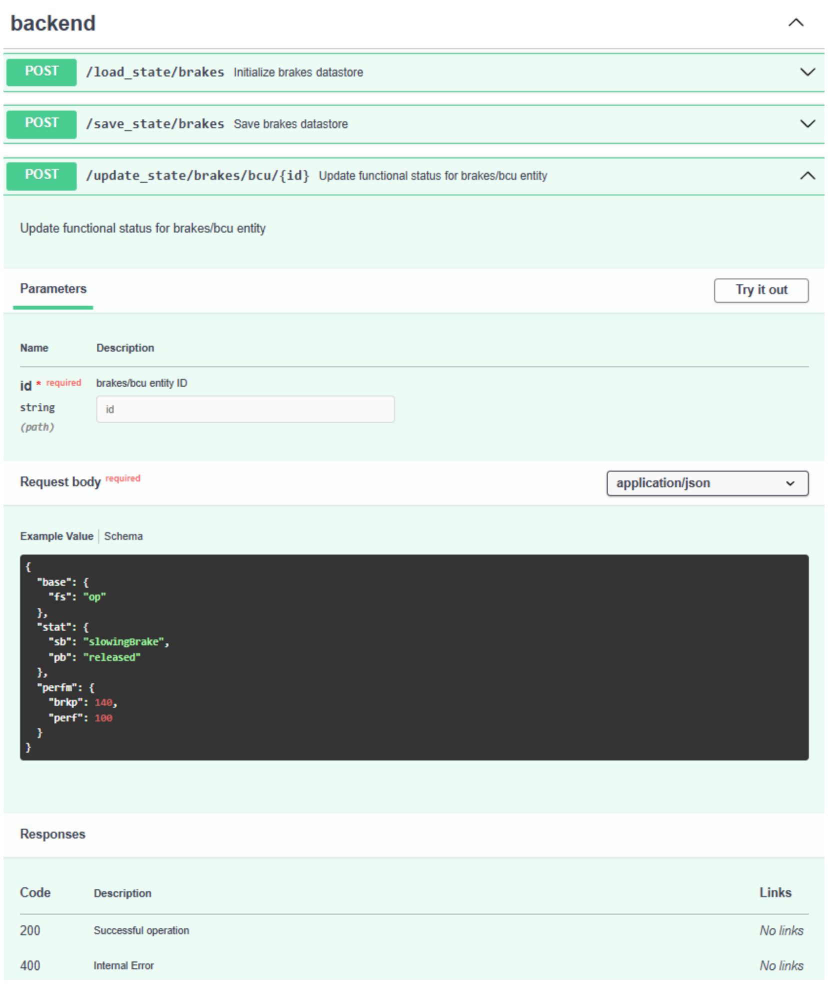
Figure 21. Rendered view of the transformed OpenAPI schema definition for bcu-stat dataset defined in the YANG data model for the Brakes use case

Generated TCMS_DS *apispec* artifacts can be used as formal *contract*s, documenting the TCMS_DS backend and frontend services provided for use case specific datasets. These contracts are particularly required for TCMS_DS integration and testing, as well as for client application development.

[25] For Brakes use case, either id=GU1 or id=GU2; see also Figure 7, on p. 24.

Generating TCMS_DS *apispec* artifacts as OpenAPI documents allows specifying service API contracts in a common and language-agnostic format. Moreover, there is a broad range of tools [26] facilitating the exchange and use of OpenAPI contracts in different environments.

One example is the swagger.io editor [27], which was used in this work for visualisation and validation of generated *apispec* artifacts. Figure 22 shows a typical validation scenario where the swagger editor was used (1) to check syntactical correctness of *apispec*, and (2) to "try out" things by submitting example requests, defined by *apispec*, to service endpoints implemented by the TCMS_DS prototype; e.g. sending a POST *update_state/brakes/bcu* with exemplary payload data as shown in Figure 22.


backend

POST `/load_state/brakes` Initialize brakes datastore

POST `/save_state/brakes` Save brakes datastore

POST `/update_state/brakes/bcu/{id}` Update functional status for brakes/bcu entity

Update functional status for brakes/bcu entity

Parameters — Try it out

Name	Description
id * required string *(path)*	brakes/bcu entity ID id

Request body required — application/json

Example Value | Schema

```
{
  "base": {
    "fs": "op"
  },
  "stat": {
    "sb": "slowingBrake",
    "pb": "released"
  },
  "perfm": {
    "brkp": 140,
    "perf": 100
  }
}
```

Responses

Code	Description	Links
200	Successful operation	*No links*
400	Internal Error	*No links*

Figure 22. Validation of generated TCMS_DS apispec artifact for the Brakes use case

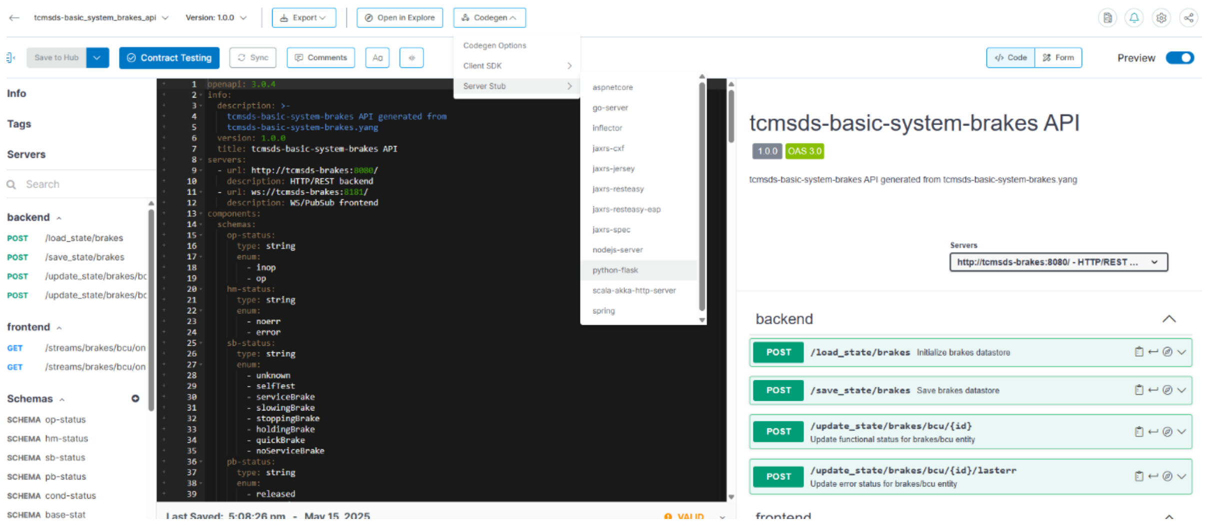
Besides formal contract specification, generated *apispec* artifacts can be used as well to automate the setup and configuration of TCMS_DS backend and frontend services.[28]

The current implementation of the TCMS_DS prototype uses generated *apispec* artifacts for Brakes and Doors use cases to configure the PubSub interface (server frontend) during server startup [29]. The REST interface (server backend) of the TCMS_DS prototype does not need to be configured yet, as the differences in backend API contracts for the Brakes and Doors use cases are minimal.

[26] See for example: https://openapi.tools/
[27] See https://swagger.io/tools/open-source/, and https://editor.swagger.io/
[28] Considering OpenAPI document sections: *servers, paths and components/schema*
[29] For example, setting up the list of channel URIs for hosted event streams

So, routing of *load_state*, *save_state* and *update_state* requests can be accomplished by common service handlers, and the processing of use case specific payload data is solely managed by the TCMS_DS internal *datastore* component that is configured differently; see section 3.3.2.

More advanced TCMS_DS implementations could use generated server stubs. For example, the TCMS_DS prototype uses Python/Flask for the implementation of the *server* component; cf. section 4.2. The generation of Python/Flask server stubs from OpenAPI contracts is supported by tools like *swagger-codegen* [30], or *openapi-generator* [31]. As a basic proof of concept, the generated TCMS_DS *apispec* for Brakes use case has been used as input for *swagger-codegen* tool to generate Python/Flask- server stubs as shown in Figure 23 and Figure 24.



```
openapi: 3.0.4
info:
  description: >-
    tcmsds-basic-system-brakes API generated from
    tcmsds-basic-system-brakes.yang
  version: 1.0.0
  title: tcmsds-basic-system-brakes API
servers:
  - url: http://tcmsds-brakes:8080/
    description: HTTP/REST backend
  - url: ws://tcmsds-brakes:8181/
    description: WS/PubSub frontend
components:
  schemas:
    op-status:
      type: string
      enum:
        - inop
        - op
    hm-status:
      type: string
      enum:
        - noerr
        - error
    sb-status:
      type: string
      enum:
        - unknown
        - selfTest
        - serviceBrake
        - slowingBrake
        - stoppingBrake
        - holdingBrake
        - quickBrake
        - noServiceBrake
    pb-status:
      type: string
      enum:
        - released
```

Last Saved: 5:08:26 pm - May 15 2025

tcmsds-basic-system-brakes API

1.0.0 OAS 3.0

tcmsds-basic-system-brakes API generated from tcmsds-basic-system-brakes.yang

Servers: http://tcmsds-brakes:8080/ - HTTP/REST ...

backend

POST /load_state/brakes Initialize brakes datastore

POST /save_state/brakes Save brakes datastore

POST /update_state/brakes/bcu/{id} Update functional status for brakes/bcu entity

POST /update_state/brakes/bcu/{id}/lasterr Update error status for brakes/bcu entity

GET /streams/brakes/bcu/on

GET /streams/brakes/bcu/on

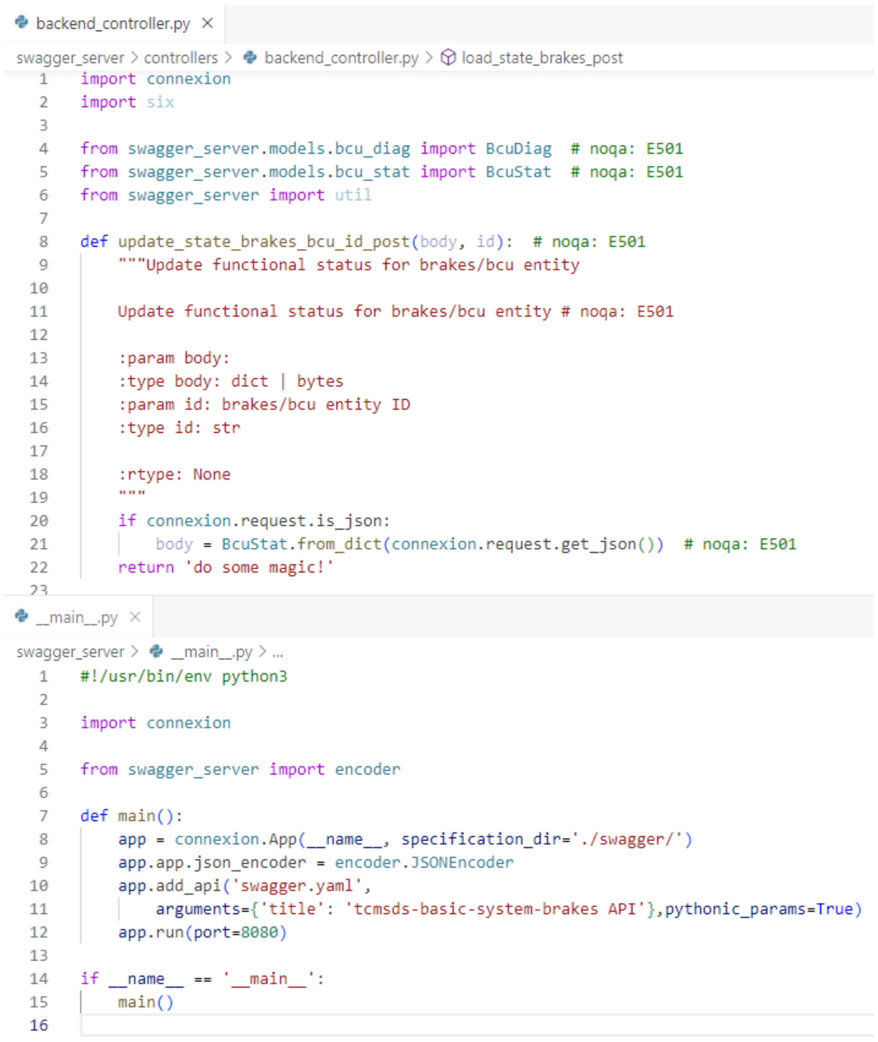
Figure 23 PoC: generating TCMS_DS server stubs, swagger-codegen usage

The *swagger-codegen* output from Figure 23 is a complete python3 boilerplate for a HTTP/REST server application that implements the service endpoints defined by *apispec* for Brakes use; see also Figure 24 for a code extract that shows the generated code modules implementing the server main and service request handlers. The generated boilerplate not only includes code modules, but also unit test templates and even a Dockerfile template.

The code from Figure 24 uses the Python/Connexion package [32], which allows a direct use of OpenAPI-based contracts in Python/Flask applications to accomplish an automated setup of specified REST endpoints in the server application; see the article from [21] for an in-depth discussion of this *design-by-contract* driven approach with some handy examples.

The use of model-generated OpenAPI contracts, such as the YANG-based TCMS_DS *apispec* artifacts, further automates and simplifies things, as it relieves software engineers form manual authoring and maintaining OpenAPI contracts.

[30] See https://github.com/swagger-api/swagger-codegen
[31] See https://openapi-generator.tech/docs/generators/python-flask
[32] See https://github.com/spec-first/connexion



backend_controller.py

swagger_server > controllers > backend_controller.py > load_state_brakes_post

```python
import connexion
import six

from swagger_server.models.bcu_diag import BcuDiag  # noqa: E501
from swagger_server.models.bcu_stat import BcuStat  # noqa: E501
from swagger_server import util

def update_state_brakes_bcu_id_post(body, id):  # noqa: E501
    """Update functional status for brakes/bcu entity

    Update functional status for brakes/bcu entity # noqa: E501

    :param body:
    :type body: dict | bytes
    :param id: brakes/bcu entity ID
    :type id: str

    :rtype: None
    """
    if connexion.request.is_json:
        body = BcuStat.from_dict(connexion.request.get_json())  # noqa: E501
    return 'do some magic!'
```

__main__.py

swagger_server > __main__.py > ...

```python
#!/usr/bin/env python3

import connexion

from swagger_server import encoder

def main():
    app = connexion.App(__name__, specification_dir='./swagger/')
    app.app.json_encoder = encoder.JSONEncoder
    app.add_api('swagger.yaml',
        arguments={'title': 'tcmsds-basic-system-brakes API'},pythonic_params=True)
    app.run(port=8080)

if __name__ == '__main__':
    main()
```

Figure 24. PoC: Generating TCMS_DS server stubs and generated python3 code

3.3.4 Workflow automation environment

The previous sections of chapter 3 described the YANG-based data modelling approach pursued in this work towards formalisation of TCMS_DS datasets, as well as the methods applied to transform YANG models into executable code (*object model*) and API contracts (*apispec*) towards integration as service extensions into TCMS_DS prototype.

This section outlines the authoring tools and automation scripts used in this work to streamline the data modelling and model transformation activities into a practical workflow, which has been applied here for Brakes and Doors use cases.

Figure 25 illustrates the applied modelling and automation workflow in terms of actors, activities and tools contributing to the authoring of TCMS_DS data models and the generation of corresponding *object model* and *apispec* artifacts for integration into the TCMS_DS prototype software.

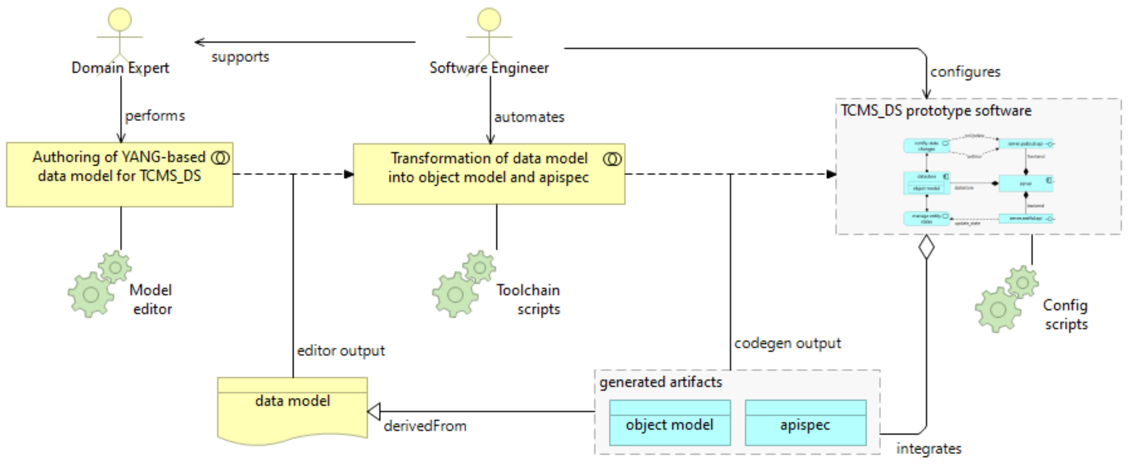


Figure 25. Applied modelling and automation workflow

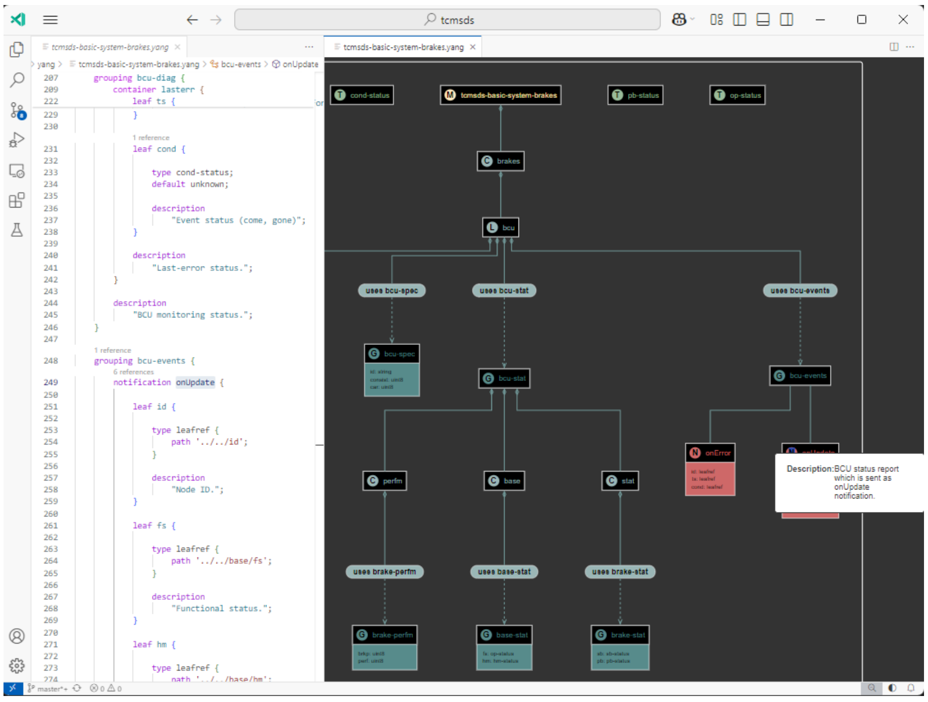
According to Figure 25, domain experts with dedicated background in TCMS, vehicle systems and system diagnostics are supposed to perform the definition of TCMS_DS datasets and the authoring of corresponding data models in YANG. Software engineers with dedicated background in YANG modelling might support in the authoring process when it comes to the practical use of YANG modelling language and corresponding tools.

The basic authoring tool used in the workflow is the *model editor*, which could be, in general, the standard text editor from an Office-PC, or the text editor from an IDE. However, it is recommended to choose a text editor for the model authoring, which provides at least a basic syntax highlighting and syntax checking functionality for YANG modules (*.yang text files).

In this work, VSCode [33] in combination with the Yangster extension [34] has been used as model editor. VSCode is a free, extensible IDE that is frequently used in various development contexts [35]. The Yangster extension adds syntax highlighting and syntax-checking features for YANG language to VSCode, and also provides rendering of textual YANG specifications as interactive diagrams; see Figure 26 below.

In addition to the YANG authoring support, the model editor should also facilitate the integration of SCM tools, like git, for tracking model changes and managing model revisions. This is particularly needed when model authoring is jointly performed in a team. In this work, YANG models for Brakes and Doors use cases have been authored by a team of two domain experts and one software developer, where the collaborative authoring has been mostly done remotely, and model revisions have been maintained and exchanged through a central git repository. The fact that YANG models are pure text files makes comparing and merging different model versions via git quite easy. VSCode, which was chosen here as model editor, has a built-in git support.

[33] See https://code.visualstudio.com/
[34] See https://github.com/TypeFox/yang-vscode
[35] See also https://www.techradar.com/reviews/microsoft-vs-code



```
    grouping bcu-diag {
        container lasterr {
            leaf ts {
            }

            1 reference
            leaf cond {

                type cond-status;
                default unknown;

                description
                    "Event status (come, gone)";
            }

            description
                "Last-error status.";
        }

        description
            "BCU monitoring status.";
    }

    1 reference
    grouping bcu-events {
        6 references
        notification onUpdate {

            leaf id {

                type leafref {
                    path '../../id';
                }

                description
                    "Node ID.";
            }

            leaf fs {

                type leafref {
                    path '../../base/fs';
                }

                description
                    "Functional status.";
            }

            leaf hm {

                type leafref {
                    path '../../base/hm';
```

Figure 26. Use of VSCode with Yangster extension as model editor

To streamline model transformation activities, a number of batch scripts (referred to as *toolchain scripts* in Figure 25) have been developed in this work. The main purpose of these scripts is to simplify the invocation of the model transformation toolchain (cf. section 3.3.1). Furthermore, these batch scripts were added as *build tasks* to VSCode from where they can be run for predefined modelling uses cases via editor shortcut CTRL+SHIFT+B <build task>. Figure 27 illustrates the integration of model batch processing tasks in the model editor.

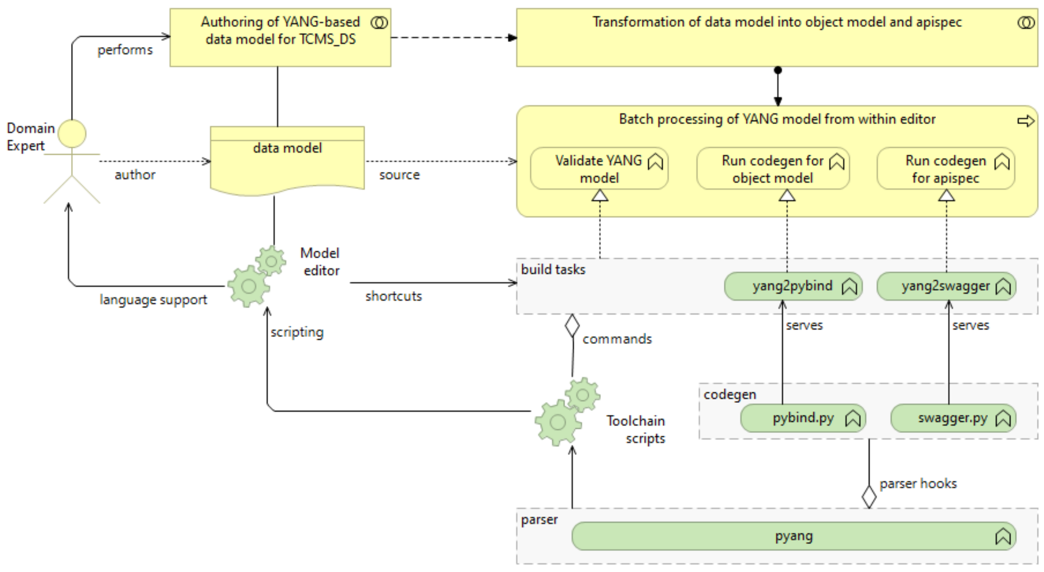
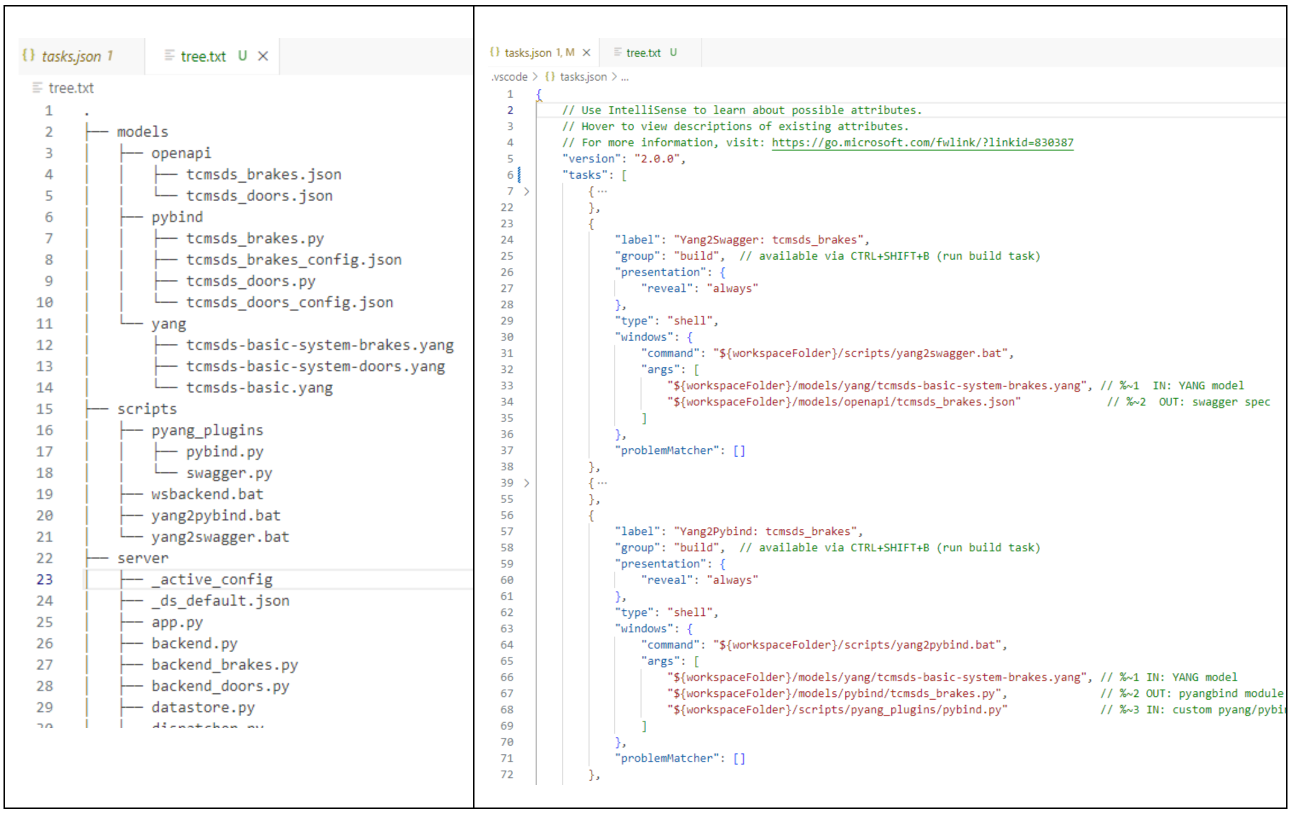


Figure 27. Integration of model batch processing tasks into model editor

Editor configuration files for use case specific build tasks and associated toolchain scripts were deployed in this work through a central git repository, alongside with YANG model sources for the Brakes and Doors use cases, and Python sources for the TCMS_DS prototype software. This git repository obeys a predetermined workspace structure, which is assumed by configured build tasks to find associated YANG models and toolchain scripts, and to output associated build artifacts (i.e. YANG bindings and swagger specs) to a specific workspace path. The used repository structure is shown below on the left side of Figure 28. By opening VSCode in the top-level git repository folder, the git repository becomes the editor's *workspaceFolder,* which is used as path variable in *build task* configurations, as shown on the right side of Figure 28. This is how the integration of toolchain scripts as editor build tasks has been realised in this work. Adding build tasks for additional TCMS_DS modelling use cases (e.g. YANG models for Lights or HVAC datasets) is just a matter of copy pasting existing build configurations and adjusting paths for associated YANG models and build artifacts.



```
.
├── models
│   ├── openapi
│   │   ├── tcmsds_brakes.json
│   │   └── tcmsds_doors.json
│   ├── pybind
│   │   ├── tcmsds_brakes.py
│   │   ├── tcmsds_brakes_config.json
│   │   ├── tcmsds_doors.py
│   │   └── tcmsds_doors_config.json
│   └── yang
│       ├── tcmsds-basic-system-brakes.yang
│       ├── tcmsds-basic-system-doors.yang
│       └── tcmsds-basic.yang
├── scripts
│   ├── pyang_plugins
│   │   ├── pybind.py
│   │   └── swagger.py
│   ├── wsbackend.bat
│   ├── yang2pybind.bat
│   └── yang2swagger.bat
├── server
│   ├── _active_config
│   ├── _ds_default.json
│   ├── app.py
│   ├── backend.py
│   ├── backend_brakes.py
│   ├── backend_doors.py
│   ├── datastore.py
```

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "2.0.0",
    "tasks": [
        {…
        },
        {
            "label": "Yang2Swagger: tcmsds_brakes",
            "group": "build",  // available via CTRL+SHIFT+B (run build task)
            "presentation": {
                "reveal": "always"
            },
            "type": "shell",
            "windows": {
                "command": "${workspaceFolder}/scripts/yang2swagger.bat",
                "args": [
                    "${workspaceFolder}/models/yang/tcmsds-basic-system-brakes.yang", // %~1  IN: YANG model
                    "${workspaceFolder}/models/openapi/tcmsds_brakes.json"            // %~2  OUT: swagger spec
                ]
            },
            "problemMatcher": []
        },
        {…
        },
        {
            "label": "Yang2Pybind: tcmsds_brakes",
            "group": "build",  // available via CTRL+SHIFT+B (run build task)
            "presentation": {
                "reveal": "always"
            },
            "type": "shell",
            "windows": {
                "command": "${workspaceFolder}/scripts/yang2pybind.bat",
                "args": [
                    "${workspaceFolder}/models/yang/tcmsds-basic-system-brakes.yang", // %~1 IN: YANG model
                    "${workspaceFolder}/models/pybind/tcmsds_brakes.py",             // %~2 OUT: pyangbind module
                    "${workspaceFolder}/scripts/pyang_plugins/pybind.py"             // %~3 IN: custom pyang/pybi
                ]
            },
            "problemMatcher": []
        },
```

Figure 28. TCMS_DS workspace structure (left) and build task configuration (right)

The TCMS_DS modelling and workflow automation environment described above has been setup and tested on Windows 11 workstations with recent a VSCode Insiders [36] installation; VSCode/Yangster extension v2.3.4 or higher is required for YANG language support. Toolchain scripts used by editor build tasks were developed as batch scripts for Windows *cmd32* interpreter. MSYS2 [37] has been used as build platform for hosting the python3 based model transformation toolchain, comprising the *pyang* parser package with *pybind* and *swagger* plugins (cf. section 3.3.1), providing model validation and code generation functions. See also section 4.4 for a complete overview of technologies and tools used in this work for the TCMS_DS prototype development.

Generated artifacts produced by the modelling and automation workflow from Figure 25, i.e. *object model* and *apispec*, are deployed through the central git repository as well. Generated *object models* containing the TCMS_DS *datastore* extensions for supported datasets (i.e. datasets for Brakes or Doors use case) are located in the *models/pybind* folder of the git repository and provided as python3

[36] See https://code.visualstudio.com/insiders/; v.1.97-insiders was installed at the time of writing this document
[37] See https://www.msys2.org/

modules, named *tcmsds_<usecase>.py* (e.g. *tcmsds_brakes.py*), alongside with their default configurations used for datastore setup in the TCMS_DS prototype; see the left side of Figure 28 as well as section 3.3.2. Similarly, generated *apispec* artifacts are located in the *models/openapi* folder of the git repository and provided as OpenAPI/JSON modules, named *tcmsds_<usecase>.json*.

Maintaining different model variants and/or default configurations of TCMS_DS datasets for specific projects might be accomplished by using dedicated branches or tags of the TCMS_DS git repository. Another option would be to maintain the *models* folder as separate, project-specific git repository, and include it in the editor workspace as git submodule. However, none of these options for TCMS_DS configuration management were practically tested yet.

4 IMPLEMENTATION OF THE TCMS_DS PROTOTYPE

A prototypical implementation of the TCMS_DS has been developed for the validation of data modelling and workflow automation concepts from chapter 3. Data models for Brakes and Doors status have been used to test the integration of model-generated TCMS_DS extensions. Furthermore, the developed prototype has been used in lab environments to evaluate pursued TCMS_DS deployments for use by agents in ATO GoA3/4 contexts and will be integrated into the Onboard Platform Demonstrator in WP 36.

4.1 IMPLEMENTATION CONCEPT

Figure 29 illustrates the general lab setup used to evaluate and test TCMS_DS deployments for Brakes and Doors systems.

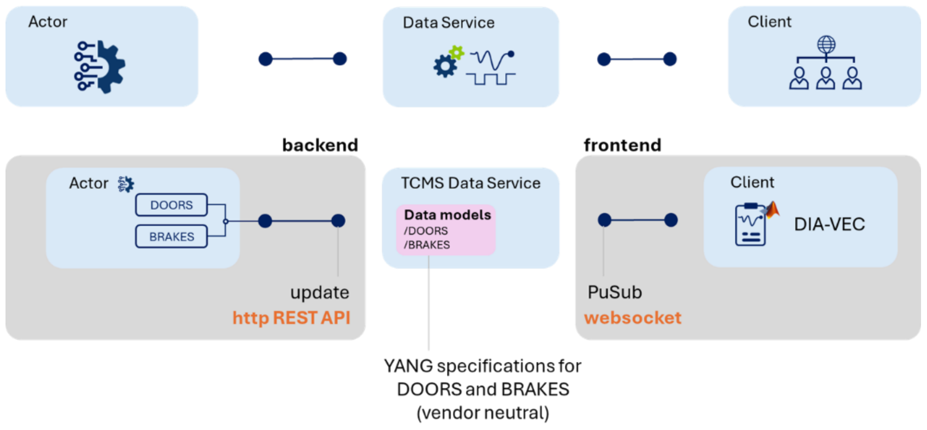


Figure 29. TCMS_DS prototype – Implementation concept

In the setup shown in Figure 29, behavioural actor models for Brakes and Doors are used to simulate system states and error conditions. These simulations feed the TCMS_DS backend and trigger corresponding update notifications in the TCMS_DS frontend.

The TCMS_DS prototype is implemented as a web service that provides a RESTful backend API [22] for posting updates. The frontend interface is realised as PubSub/WebSocketServer [23] [24], enabling TCMS_DS to publish hosted datasets as event streams (denoted as *channels*), and to push update notifications to subscribed clients in near real-time.

Simulated system states are sent to the TCMS_DS backend interface via HTTP-POST requests to endpoint */update_state/<entity>/* with payload format *application/json*. Similarly, simulated error conditions are sent to endpoint */update_state/<entity>/lasterr/*. The *<entity>* placeholder denotes the YANG path of the managed resource to which the *update_state* request applies. Channels published by the TCMS_DS frontend can be subscribed from WebSocket clients [38] by connecting to the channel Uniform Resource Identifier (URI) */tcmsds/<entity>/<stream>/.*

As already stated earlier in this document (see ch. 1, p. 12), the actual technologies used in this work to realize the formal TCMS data modelling workflow, and to implement the TCMS_DS software prototype, were deliberately chosen to conduct a proof of concept. Product-ready TCMS_DS implementations for next-generation onboard architectures must comply with future iterations of the onboard network communication constraints defined by ERTMS/ETCS SUBSET-147 [25] (One

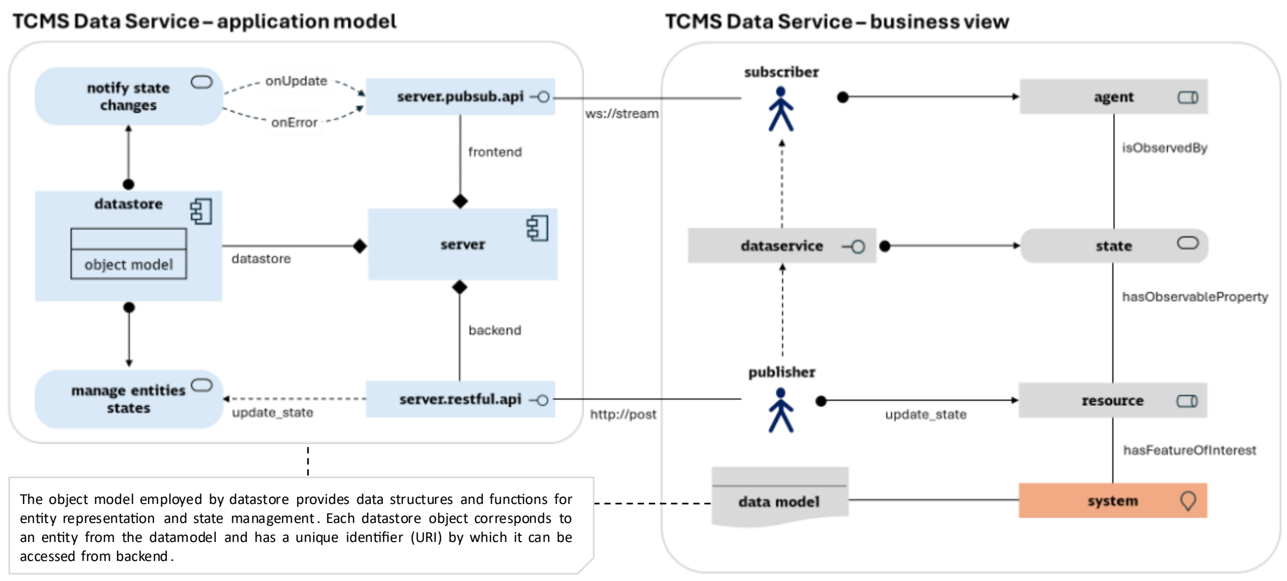
[38] Can be browser clients or dedicated applications such as fault management agents.

Common Bus). R2DATO Task 23 is specifying such a proposal for the next iteration of SUBSET-147. For further details refer to deliverable D23.4 [26].

Moreover, the harmonization of middleware technologies to be used for realizing data services and data distribution in future onboard platforms is currently defined in R2DATO Task 23.2 and Task 23.3; see [27] and [28] for details.

4.2 Software Architecture

The intended usage and derived software architecture of the TCMS_DS prototype is outlined in Figure 30 in the form of an application model and business view [39].



The object model employed by datastore provides data structures and functions for entity representation and state management. Each datastore object corresponds to an entity from the datamodel and has a unique identifier (URI) by which it can be accessed from backend.

Figure 30. TCMS_DS prototype - Software architecture in the form of application model and business view

Considering the business view from the right side of Figure 30, TCMS_DS is acting as a proxy for general system status information modelled here as *dataservice* interface hosting *states* of one or more *system resources*. Available system resources and associated status data are defined in the *system data model*, which is used as an input for dataservice configuration. Actors responsible for providing status updates are modelled as *publishers*. Technically, publishers are supposed to be system control units responsible for hosting one or more resources; for example, BCUs and DCUs in case of Brakes and Doors systems.

The derived application model for the TCMS_DS prototype is shown on the left side of Figure 30. Therein, the main application component is the *server*, which is composed of the *backend* and *frontend* interface, and the *datastore* component. As already mentioned in section 4.1, the TCMS_DS prototype is implemented as a webservice providing a RESTful *backend* interface for receiving system status updates, and a PubSub *frontend* interface for pushing update notifications to connected clients (subscribers). The *server*-internal *datastore* component employs an *object model* for entity- and state-representation of managed resources. The code for data structures and functions that implements the object model is generated from YANG data models of systems in scope of the TCMS_DS prototype, using the automation workflow described in section 3.3.4.

[39] ArchiMate is used in Figure 30 as modelling notation.

Accessing datastore entities from the backend interface, for the purpose of state retrieval and state update, is accomplished through the datastore management API, which is modelled in Figure 30 as service *manage entity states*. Forwarding state updates for managed resources from the datastore to the frontend interface, for the purpose of sending update notifications to connected clients, is realised through the datastore notification API, which is modelled in Figure 30 as a service *notify state changes*.

4.3 Processing model

Processing of state updates for managed resources is implemented in the backend part of the TCMS_DS prototype. State updates are sent by system actors (publishers) to the TCMS_DS backend interface, as HTTP-POST requests to */update_state/* endpoint. The reception of an update request in the TCMS_DS backend interface triggers the processing sequence shown in Figure 31.

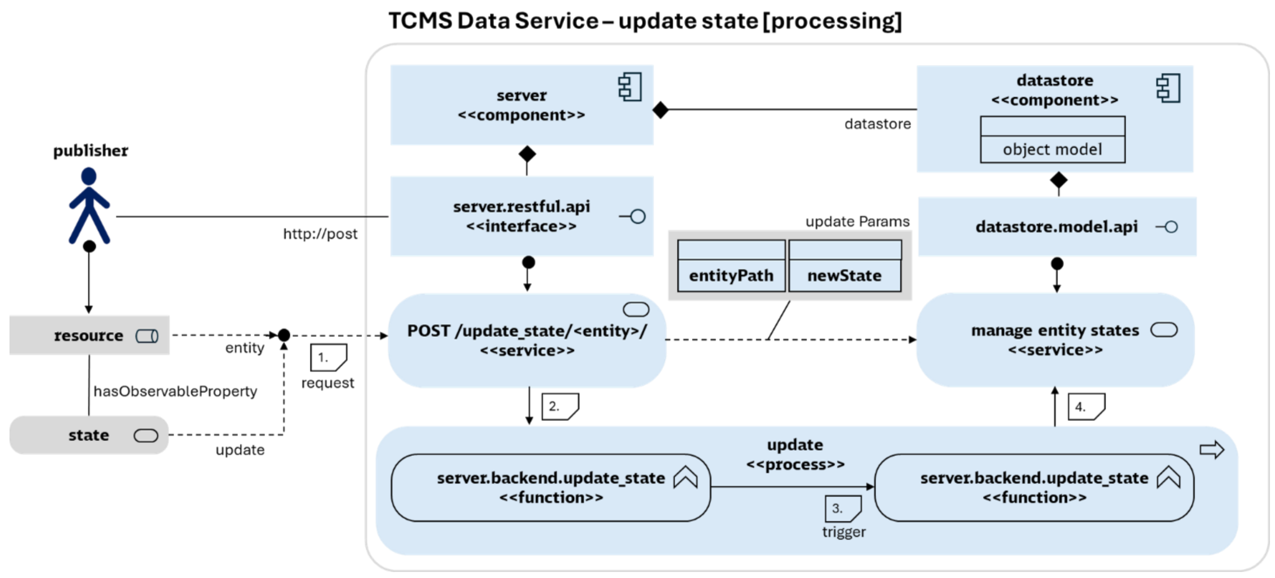
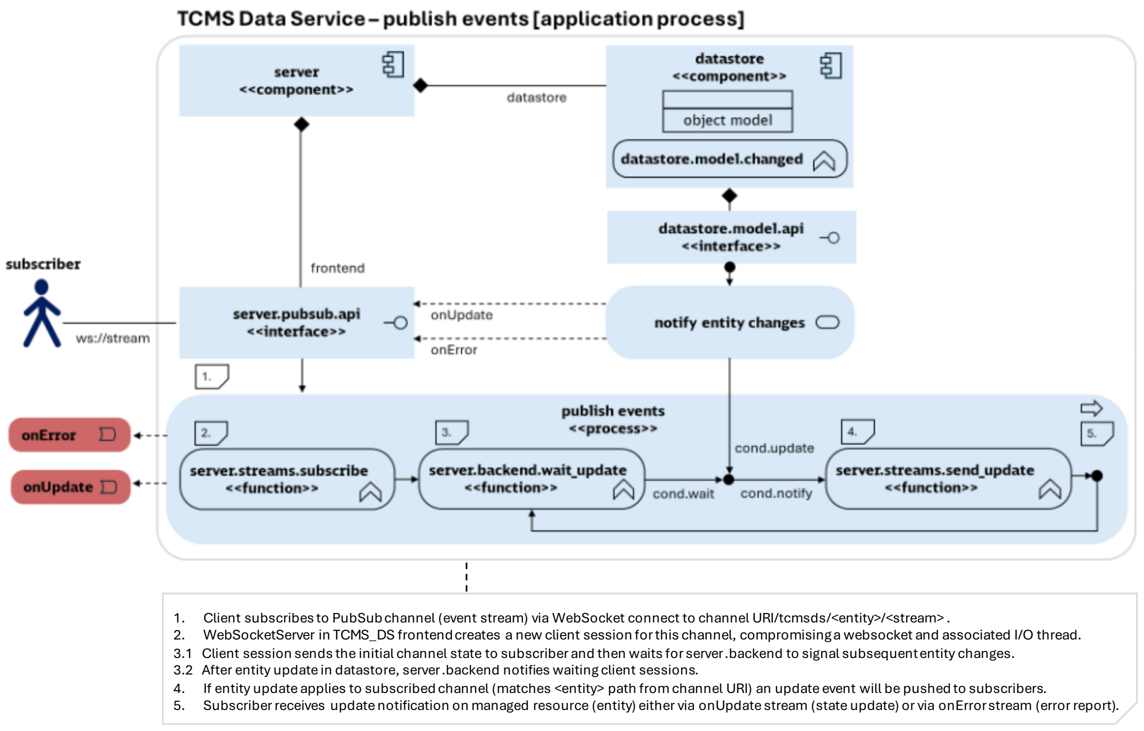

TCMS Data Service – update state [processing]
server <<component>>
datastore <<component>>
object model
datastore
publisher
server.restful.api <<interface>>
http://post
update Params
datastore.model.api
entityPath
newState
resource
entity
POST /update_state/<entity>/ <<service>>
manage entity states <<service>>
1. request
hasObservableProperty
2.
4.
state
update
update <<process>>
server.backend.update_state <<function>>
server.backend.update_state <<function>>
3. trigger

Figure 31. TCMS_DS prototype – update processing in server backend

Notifications to external actors (i.e. subscribers, such as agents) about state updates in managed resources are implemented in the frontend part of the TCMS_DS prototype. Notifications are realised, per managed resource (entity), as a corresponding pair of PubSub channels (event streams) hosted by the frontend interface. The format of PubSub channel-URIs used in the TCMS_DS prototype is */tcmsds/<entity>/<stream>/,* where *<entity>* denotes the resource path from the YANG data model, and *<stream>* denotes the published event stream, which is either *onUpdate* or *onError.* The former event stream provides notifications on regular state updates (*functional status* changes), while the latter event stream provides notifications about changed error conditions (error reports). Splitting notifications on managed resources into *onUpdate* and *onError* streams allows separate processing of functional states and error conditions, which is supposed to simplify the software design and implementation of TCMS_DS client applications (subscribers). Figure 32 shows the event processing and notification of subscribers, as implemented in the TCMS_DS prototype.


TCMS Data Service – publish events [application process]
server <<component>>
datastore
datastore <<component>>
object model
datastore.model.changed
datastore.model.api <<interface>>
frontend
subscriber
ws://stream
server.pubsub.api <<interface>>
onUpdate
onError
notify entity changes
1.
publish events <<process>>
onError
onUpdate
2.
3.
4.
5.
server.streams.subscribe <<function>>
server.backend.wait_update <<function>>
cond.wait
cond.update
cond.notify
server.streams.send_update <<function>>

1. Client subscribes to PubSub channel (event stream) via WebSocket connect to channel URI/tcmsds/<entity>/<stream>.
2. WebSocketServer in TCMS_DS frontend creates a new client session for this channel, compromising a websocket and associated I/O thread.
3.1 Client session sends the initial channel state to subscriber and then waits for server.backend to signal subsequent entity changes.
3.2 After entity update in datastore, server.backend notifies waiting client sessions.
4. If entity update applies to subscribed channel (matches <entity> path from channel URI) an update event will be pushed to subscribers.
5. Subscriber receives update notification on managed resource (entity) either via onUpdate stream (state update) or via onError stream (error report).

Figure 32. TCMS_DS prototype – processing of update notifications in the server frontend

4.4 Technology Stack and Tools

An overview about the main technologies and tools used for the implementation of the TCMS_DS prototype is provided in Table 4 below. Comments explaining and justifying design decisions were added to make ideas transparent and to facilitate technical discussions.

Table 4. TCMS-DS prototype – Technology stack and tools

Purpose	Used technologies and tools	Comments
Data modelling language and authoring tools for formal modelling and automated code generation of TCMS_DS datastore extensions (comprising the object model and management API)	• YANG 1.1 (RFC 7950) • VSCode editor with Yangster extension for syntax highlighting, visualization and pretty printing of textual YANG specifications • pyang parser tools in combination with pybind parser plugin for generation of TCMS_DS datastore extensions from textual YANG specifications of datasets for relevant system status information from TCMS domain, e.g. system status and error reports for Brakes and Doors systems.	• YANG language fits all needs for formal modelling of TCMS_DS datasets • YANG is the designated successor for SNMP/MIB industry standard for remote network configuration and network management applications • Model editor and parser tools are open source and actively maintained • Python as target language for parser- and code-generation tools is well-suited for use in simulation and prototyping environments (e.g. for

		integration and usage in Matlab Simulink)
Technologies used for implementation of the TCMS_DS backend and frontend interface	• HTTP/REST in combination with OpenAPI as means for realising TCMS_DS backed interface for receiving status updates and error reports on managed resources from system-side actors (publishers) • WebsocketServer protocol according to RFC 6455 in combination with PubSub messaging pattern as means for pushing update notifications on managed resources to one more clients (subscribers) from operator zone (according to Eurospec TCMS_DS use cases), or clients from CCS-OB domain (pursued in ATO GoA 3/4 onboard use cases).	• Considering future TCMS and CCS-OB system architectures and pursued use of TCMS_DS as provider for onboard data in ATO GoA3/4 contexts, the realization of TCMS_DS backend interface as RESTful webservice with an OpenAPI based endpoint specification provides maximum flexibility and scalability for adapting to different usage scenarios and onboard architecture variants. • Considering the “General System Environment” from section 4.5 and the broad spectrum of potential TCMS_DS client applications (mentioned in Eurospec TCMS_DS specification and in chapter 2 of this document), the use of Websockets protocol as session layer and network transport for TCMS_DS frontend interface provides a robust and scalable foundation for implementing PubSub channels (event streams) to push update notifications to connected clients in near realtime; see [23] and [29] for discussions on using WebSockets for real-time applications, and potential alternatives. • Websockets protocol is directly supported by HTML5 standard and all modern browsers, which facilitates TCMS_DS usage from browser-based, interactive client applications; see also [30] • For using TCMS_DS PubSub channels from native clients, e.g. agents, existing implementations of WebSocket protocol for all mainstream programming languages (e.g. C/C++, java, javascript, python, go, etc.) exist. For example, a

		minimal and performant C/C++ based implementation of WebSockets protocol, for potential use in embedded TCMS_DS client applications, is mongoose [40] library.
API specification format for TCMS_DS webservices	• OpenAPI 3.0 (also known as Swagger) for specification of restful TCMS_DS backend interface • JSON representations of OpenAPI data descriptors are directly used to define the message data for TCMS_DS update_state requests (backend interface), and TCMS_DS update notifications (frontend interface)	• OpenAPI is widely used and becomes more and more a de-facto standard • Examples and tools exist (both open source and commercial) for automated transformation of YANG models into corresponding OpenAPI specifications for webservice endpoints
Coding language and Webservice frameworks used for TCMS_DS server component	• Python 3 has been mostly used for development of TCMS_DS prototype components • The developed server code and used webservice frameworks (see below) have been tested with latest releases for Python 3.7 and Python 3.9 • Python/Flask WSGI web application framework for implementation of HTTP server app and RESTful backend interface • Python/Autobahn WebSocket server framework for implementation of PubSub frontend interface	• Using Python as coding language for TCMS_DS prototype was preferred to allow direct integration and usage of generated model extensions (see above) • Decisions for using Python/Flask and Python/Autobahn as frameworks for implementing TCMS_DS backend and frontend services were deliberately made by developers. There might be alternative/better tools to accomplish this. However, for TCMS_DS prototype, performance and security requirements from Eurospec TCMS_DS were not in scope.
Supported runtime environments for testing TCMS_DS prototype	• Win10/Win11, where TCMS_DS is running as native Windows application, or inside WSL2 VM • Standard Linux distros; tested with Ubuntu 18.04 and 22.04 LTS VMs • TCMS_DS deployments as Docker container; tested with recent Docker Desktop versions for Win11/WSL2	

[40] See https://mongoose.ws/, and https://github.com/cesanta/mongoose

4.5 Validation and Evaluation

Figure 33 outlines the general system environment used as reference for validation and evaluation of TCMS_DS prototype and modelled datasets.

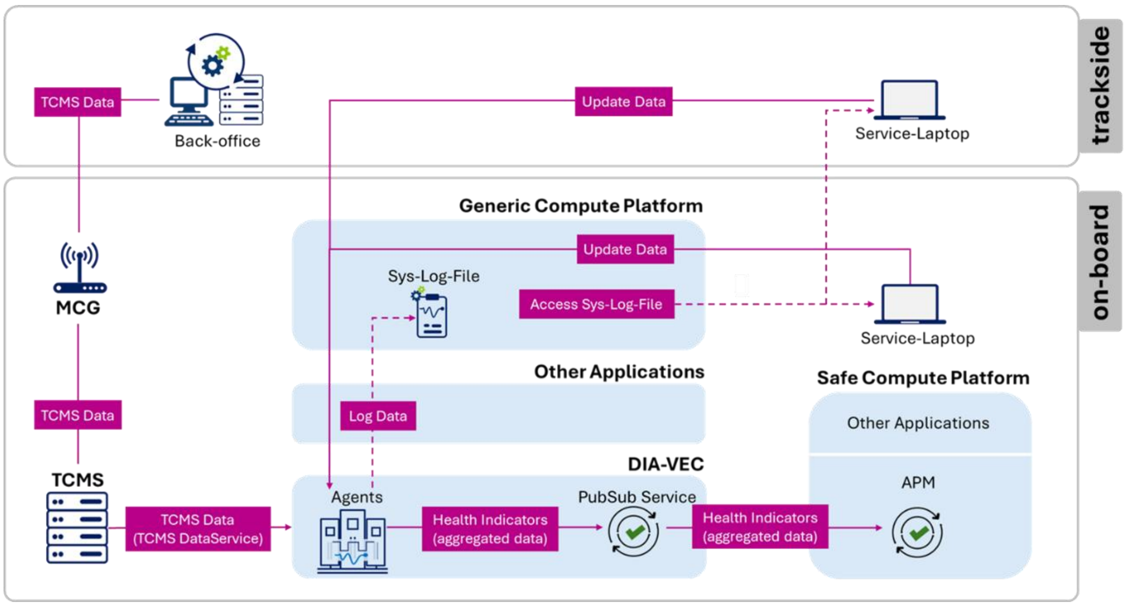


Figure 33. General system environment used as reference for validation and evaluation of TCMS_DS as central provider for common, operator specific vehicle data sets

In ATO GoA3/4 contexts, TCMS_DS is particularly needed to feed agents from CCS-OB domain responsible for processing and aggregating onboard system states to provide health- and performance indicators for automated processing functions concerned with problem solving and automated troubleshooting. DIA-VEC function, which was introduced in section 2, and which is shown in the onboard zone part of Figure 33, is an example for such agents. DIA-VEC is defined in DBS GoA4 onboard system architecture, and is responsible for supporting operator-specific procedures for fault management and automated troubleshooting in ATO contexts.

ATO-unspecific TCMS_DS deployments supporting operator needs for fleet management and vehicle maintenance, as defined in [1], are implicitly addressed in the general system environment from Figure 33 by the train-to-ground transmission of TCMS data to operator trackside-systems (back-office) via Mobile Communication Gateway (MCG) [31]. These maintenance-oriented use cases are, however, not considered here for validation and evaluation of the TCMS_DS prototype.

In the remainder of this section, we will focus on the onboard part from Figure 33 and TCMS_DS deployments for DIA-VEC specific use cases.

The Generic Compute Platform (GCP, [32] and [33]) is used in the reference environment as node for hosting non-safety-critical applications from CCS-OB domain with moderate performance constraints, such as DIA-VEC.

From the perspective of separation principles between TCMS and the “Rest of the train”, as defined in [1] Ch. 6, the GCP and hosted DIA-VEC agents belong to the operator functional zone, comprising non-safety-critical vehicle functions and operator-specific applications; see below for an example, which has been directly copied from Eurospec TCMS_DS document [1].

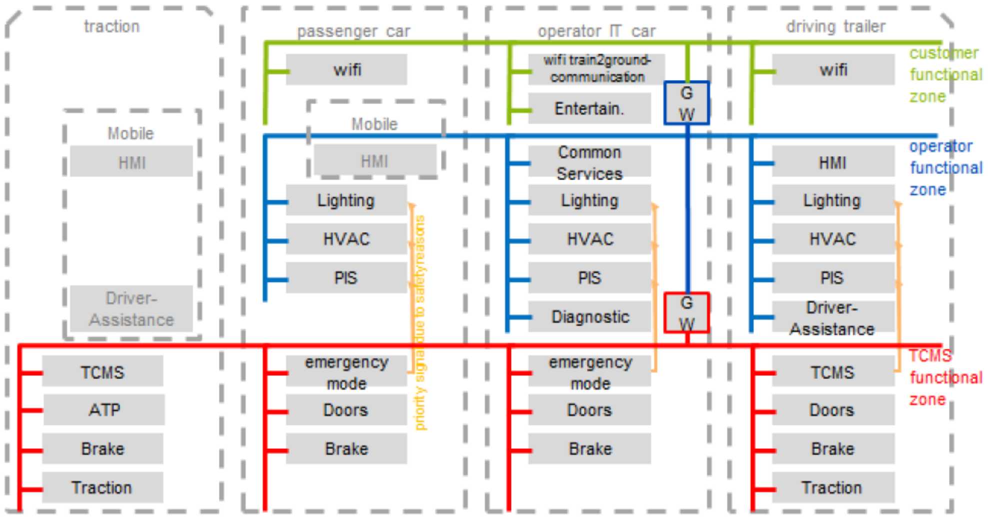


Figure 34. Separation of onboard functions into different functional zones [1]

Eurospec TCMS_DS specification proposes four possible methods for realising functional separation to achieve safety objectives. One of the proposed methods is *logical separation* via VLANs [34] in Ethernet-based onboard networks. This method will be assumed for the reference environment from Figure 33 to restrict communication between the TCMS node (used for hosting TCMS_DS prototype) and the GCP node (used for hosting DIA-VEC agents).

The implications of logical separation requirements for validation and evaluation of TCMS_DS prototype are as follows:

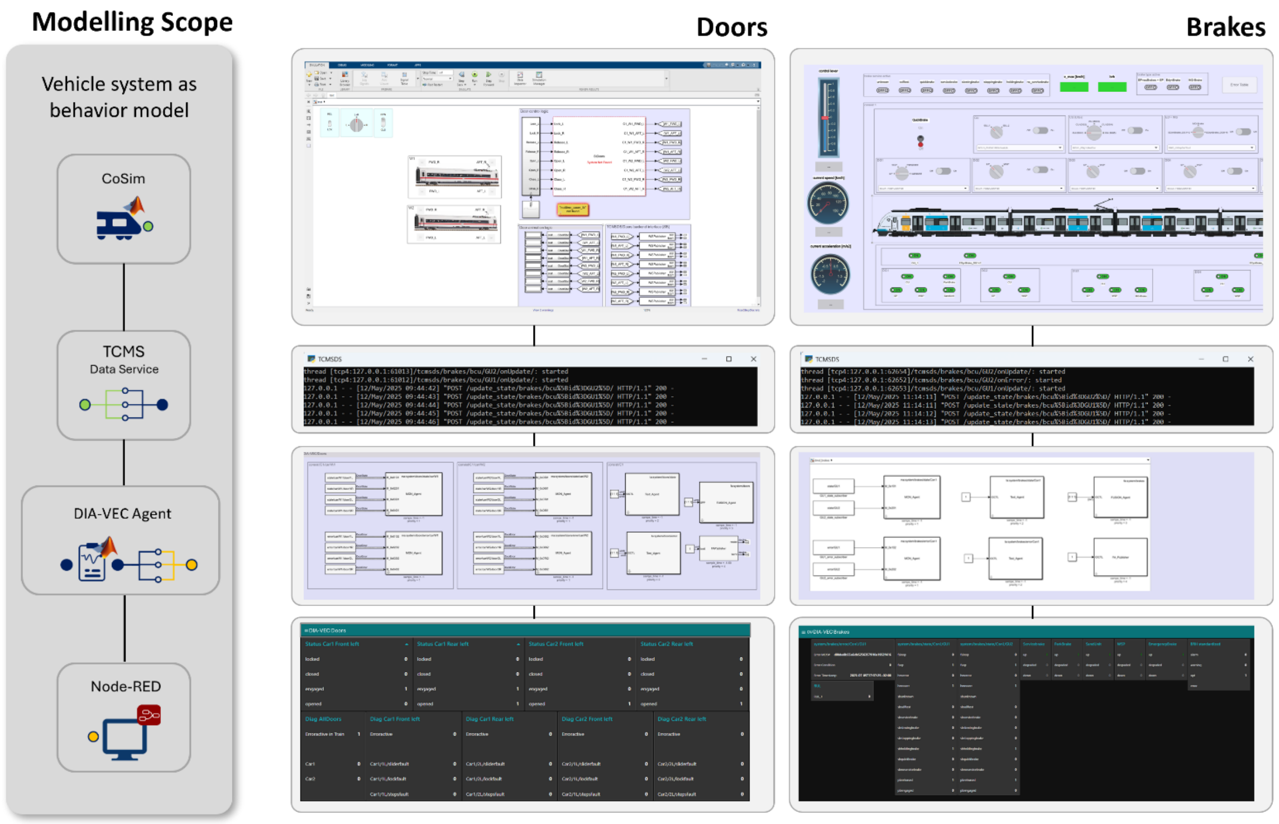
- The TCMS_DS prototype is supposed to act as data gateway between the TCMS functional zone and the operator functional zone (according to Eurospec TCMS_DS specification)
- TCMS_DS *backend* interface and TCMS_DS *frontend* interface are connected to different VLANs (used to realise logical separation between TCMS and operator functional zones). This might have implications for configuring network services in TCMS_DS software
- Since TCMS_DS software is considered *basic integrity* [41], the backend interface and hosted update service must be "receive-only". Transmission of data to TCMS nodes is not possible due to safety constraints. Only reception of state updates from TCMS nodes will be allowed[42]

So far, validation of TCMS_DS architecture and evaluation of TCMS_DS deployments with respect to separation requirements from Eurospec TCMS_DS specification has not been performed, since the focus was on formal data modelling and implementation of TCMS_DS in the form of a functional software prototype.

Planned usage and test cases of the TCMS_DS prototype in the R2DATO WP36 onboard platform demonstrator [43] won't address validation of safety-related aspects either. So, this topic is left open here and might be concluded in upcoming projects.

[41] Defined according to functional safety objectives defined in EN 50126-2, ch. 10.2, [35]
[42] In general, homologation of TCMS_DS deployments according to EN 50126-2 requires the attestation of interference-free operation, to proof that potential malfunctions cannot harm the train integrity.
[43] WP36 Onboard Platform Demonstrator; https://projects.rail-research.europa.eu/eurail-fp2/structure/

Functional testing of the TCMS_DS prototype with generated dataset extensions for Doors and Brakes use case was mainly done, so far, in DB DiaLab environments, as illustrated in Figure 35.



```
thread [tcp4:127.0.0.1:61013]/tcmsds/brakes/bcu/GU2/onUpdate/: started
thread [tcp4:127.0.0.1:61012]/tcmsds/brakes/bcu/GU1/onUpdate/: started
127.0.0.1 - - [12/May/2025 09:44:42] "POST /update_state/brakes/bcu%5Bid%3DGU2%5D/ HTTP/1.1" 200 -
127.0.0.1 - - [12/May/2025 09:44:43] "POST /update_state/brakes/bcu%5Bid%3DGU1%5D/ HTTP/1.1" 200 -
127.0.0.1 - - [12/May/2025 09:44:44] "POST /update_state/brakes/bcu%5Bid%3DGU1%5D/ HTTP/1.1" 200 -
127.0.0.1 - - [12/May/2025 09:44:45] "POST /update_state/brakes/bcu%5Bid%3DGU1%5D/ HTTP/1.1" 200 -
127.0.0.1 - - [12/May/2025 09:44:46] "POST /update_state/brakes/bcu%5Bid%3DGU1%5D/ HTTP/1.1" 200 -
```

```
thread [tcp4:127.0.0.1:62654]/tcmsds/brakes/bcu/GU2/onUpdate/: started
thread [tcp4:127.0.0.1:62652]/tcmsds/brakes/bcu/GU2/onError/: started
thread [tcp4:127.0.0.1:62653]/tcmsds/brakes/bcu/GU1/onUpdate/: started
127.0.0.1 - - [12/May/2025 11:14:11] "POST /update_state/brakes/bcu%5Bid%3DGU2%5D/ HTTP/1.1" 200 -
127.0.0.1 - - [12/May/2025 11:14:11] "POST /update_state/brakes/bcu%5Bid%3DGU1%5D/ HTTP/1.1" 200 -
127.0.0.1 - - [12/May/2025 11:14:12] "POST /update_state/brakes/bcu%5Bid%3DGU1%5D/ HTTP/1.1" 200 -
127.0.0.1 - - [12/May/2025 11:14:13] "POST /update_state/brakes/bcu%5Bid%3DGU1%5D/ HTTP/1.1" 200 -
```

Figure 35. TRL-3 demonstrators for DIA-VEC/Doors and DIA-VEC/Brakes, where the TCMS_DS prototype and the generated dataset extensions have been tested

DB DiaLab project aims to evaluate and demonstrate technical concepts for vehicle agents, pursued for DIA-VEC function.[44] The DiaLab project uses a Matlab/Simulink-based platform for TRL-3 demonstrator development, where external components, such as co-simulations or browser-based dashboards, can be integrated through custom interface blocks.

Behavioural models of Brakes and Doors systems (denoted as *cosim* in the left part of Figure 35) are used here to simulate relevant system resources for Brakes and Doors use cases in near real-time, generating synthetic state updates and error reports for configured *brakes/bcu* and *doors/door* entities sent to TCMS_DS prototype as HTTP/REST-based POST *update_state* requests.

For the integration and usage of published TCMS_DS *brakes* and *doors* datasets in Matlab/Simulink, configurable interface blocks have been developed that realise subscriber functionality for connecting to the TCMS_DS PubSub frontend.

The TCMS_DS server application, used in the DiaLab TRL-3 demonstrator, is running as host-local console process and is started/stopped by agent models.

[44] DIA-VEC is a function defined in DBS GoA4 CCS-OB architecture. The technical concept for DIA-VEC implementation is a set of agents for vehicle system data relevant for automated trouble-shooting procedures to be performed by CCS-OB functions like APM. TCMS_DS concept is pursued as central data provider for vehicle data from TCMS domain.

DiaLab TRL-3 demonstrators for DIA-VEC/Brakes and DIA-VEC/Doors have been further developed towards TRL-4 (using a Processor-In-The-Loop (PIL) setup [45]), where DIA-VEC agent models were deployed to speedgoat realtime hardware [46]. Thanks to the use of common webservice technologies and protocols for TCMS_DS prototype backend and frontend services, the reuse of TCMS_DS TRL- 3 prototype in the TRL-4 demonstrator setup was possible without any changes.

For usage in FA2 R2DATO WP 36 Onboard Platform Demonstrator, the DiaLab demonstrator components (in particular the TCMS_DS prototype and DIA-VEC agent software) have been even further developed towards demonstrating technical readiness according to TRL4 and TRL-5, respectively. The documentation of the deployment and usage of DiaLab demonstrator components in the WP 36 Onboard Platform Demonstrator will be described separately in R2DATO deliverable document D31.3.

[45] See https://de.mathworks.com/help/ecoder/processor-in-the-loop.html
[46] See: https://www.speedgoat.com/products-services/real-time-target-machines

5 CONCLUSION

The results presented in this report demonstrate that the implementation of a **model-driven, service-oriented architecture** for the **TCMS Data Service (TCMS_DS)** is both technically feasible and highly beneficial for the future of railway operations.

By leveraging open standards such as **YANG** for formal dataset modeling, **OpenAPI** for unified service capability description, and the use of state-of-the-art web technologies such as **HTTP/REST** and **PubSub/WebSocketServer,** a robust, scalable and real-time capable, implementation of a TCMS_DS prototype could be developed as prototype and validated in a lab demonstrator environment.

The **prototypical implementation** confirms that TCMS_DS can serve as an **effective integration layer**, enabling:

- Vendor-independent data access and interoperability
- Seamless integration of onboard and landside applications (via MCG)
- Support of fault management and automated troubleshooting use cases for ATO GoA 3/4

The successful validation underlines that a **standardised, event-driven vehicle data platform** is a **key enabler** for achieving the goals of **automated and efficient railway operations**, particularly in the context of the Europe's Rail innovation program. Furthermore, it is essential to ensure that **Railway Undertakings (RU)** and **Infrastructure Managers (IM)** are able to **access, interpret, and analyse vehicle data over extended lifecycle periods**, typically exceeding **+30 years**, to support operational safety, compliance tracking, asset management, and predictive maintenance strategies.

To further strengthen and industrialise this approach, the following steps are recommended:

- Extending the modelling and validation of TCMS_DS datasets to other use cases from [1], in particular those for serving onboard data for trackside clients in operator IT backend
- Evaluate TCMS_DS prototype architecture with respect to **separation**, **security, reliability, and scalability** requirements from [1]
- Extend and refine TCMS_DS prototype software up to TRL-7 (pilot for vehicle integration)

The application of formal modelling methods for the specification of vehicle datasets and the automated deployment of these models as corresponding data services, as presented in this work, directly contributes to the broader objectives of **Digital Rail initiatives** and supports the long-term evolution towards **autonomous** and **resilient railway systems**.

REFERENCES

[1] Eurospec, “Specification TCMS Data Service, 1st edition,” June 2019. [Online]. Available: https://eurospec.eu/tcms-data-service/.

[2] Horizon Europe NCP Portal, “TRL Assessment tool,” 2022.

[3] CER, “Position paper, Operators's requirements for ATO development,” CER, 2016.

[4] *Shift2Rail projects Connecta,* European Union’s Horizon 2020 research and innovation programme under grant agreement No: 730539, 2016.

[5] *X2RAIL-4,* European Union’s Horizon 2020 research and innovation programme under grant agreement No: 881806, 2019.

[6] OCORA, “Introduction to OCORA,” 2025. [Online]. Available: https://github.com/OCORA-Public/Publications/tree/master/00_OCORA%20Latest%20Publications/Latest%20Release. [Accessed 27 10 2025].

[7] DB InfraGO AG - Digitale Schiene Deutschland, “Digitale Schiene Deutschland – Zukunftstechnologien für das Bahnsystem,” *EISENBAHN INGENIEUR KOMPENDIUM,* vol. EIK 2024, pp. 189-208, 2024.

[8] TAURO, “D3.1 – Contribution to enhanced TCMS for automatic diagnostic functionality regarding autonomous train. ATO automatic functional test.,” TAURO project, 2023.

[9] TAURO, “D3.2 – Contribution to enhanced TCMS for automatic diagnostic functionality regarding autonomous train. ATO running capability.,” TAURO project, 2023.

[10] X2Rail-4, “GoA3/4 Specification v1.0.0,” S2R-IP2 X2Rail-4, 2023.

[11] R2DATO, “D5.1 - Documentation of use cases for automating functions,” R2DATO project, 2023.

[12] R2DATO, “D5.2 - Documentation of use cases for Perception system,” R2DATO project, 2023.

[13] M. Björklund, “The YANG 1.1 Data Modeling Language,” August 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7950.

[14] OpenAPI Initiative, “OpenAPI Specification v3.0.3,” 2020.

[15] IEC, EN IEC 61375-2-4:2017, Electronic railway equipment – Train communication network (TCN) - Part 2-4: TCN application profile, 2017.

[16] IEC, EN IEC 61375-1:2015, Electronic railway equipment – Train communication network (TCN) – Part 1: General architecture, 2015.

[17] M. Bjorklund, “An extensible YANG validator and converter in python,” 08 05 2025. [Online]. Available: https://github.com/mbj4668/pyang/blob/master/doc/pyang.1.md.

[18] R. Shakir, “A plugin for pyang that creates Python bindings for a YANG model.,” 08 05 2025. [Online]. Available: https://github.com/robshakir/pyangbind/blob/master/docs/README.md. [Accessed 08 05 205].

[19] netgroup-polito, “pyang plugin for swagger,” 08 05 2025. [Online]. Available: https://github.com/netgroup-polito/pyang-swagger. [Accessed 08 05 2025].

[20] W3C, “XML Path Language (XPath), Version 1.0,” W3C, 06 11 2016. [Online]. Available: https://www.w3.org/TR/1999/REC-xpath-19991116/. [Accessed 14 05 2025].

[21] P. Acsany, “Python REST APIs With Flask, Connexion, and SQLAlchemy,” Real Python, 15 05 2025. [Online]. Available: https://realpython.com/flask-connexion-rest-api/. [Accessed 15 05 2025].

[22] L. Richardson, RESTful Web Services, O'Reilly, 2007.

[23] ably.com, “What is Pub/Sub? Publish/Subscribe model explained,” ably, 02 05 2025. [Online]. Available: https://ably.com/topic/pub-sub. [Accessed 02 05 2025].

[24] P. Bisht, “Pub/Sub and WebSockets,” medium.com, 02 05 2025. [Online]. Available: https://blog.devops.dev/pub-sub-and-websockets-d8d180c2b9e8. [Accessed 02 05 2025].

[25] UNISIG, “SUBSET-147 - 1.0.0; ERTMS Data Applications, FFFIS part: CCS Consist Network Communication Layers,” 2023.

[26] R2DATO, “D23.4 – Proposal on TSI202x, SS147,” R2DATO project, 2025.

[27] R2DATO, “D23.2 – Definitive and aligned requirements set for Onboard Communication Network and Basic Services,” R2DATO project, 2024.

[28] R2DATO, “D23.3 – List of Solution Candidates,” R2DATO project, 2024.

[29] ably.com, “Alternatives to WebSockets for realtime features,” 10 08 2023. [Online]. Available: https://dev.to/ably/alternatives-to-websockets-for-realtime-features-4mkp. [Accessed 05 05 2025].

[30] F. S. a. P. M. Vanessa Wang, The Definitve Guide to HTML5 WebSocket, Apress Berkeley, CA, 2013.

[31] IEC, EN IEC 61375-2-6:2018, Electronic railway equipment – Train communication network (TCN) - Part 2-6: On-board to ground communication, 2018.

[32] P. Dr. Marsch and A. Heine, “IT Platforms for future Railway Systems,” in *Digital Rail Summer School*, Jöhrstadt, 2022.

[33] P. Dr. Marsch, S. Steffens, T. Suess and F. Dr. Eschmann, “SIL4 Data Center – a new platform architecture for safety-relevant railway applications,” *SIGNAL+DRAHT,* vol. 113, pp. 41-48, 10 2021.

[34] IEEE, IEEE Std. 802.1Q-2014, Bridges and Bridged Networks, IEEE, 2014.

[35] CENELEC, EN 50126-2, Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 2: Systems Approach to Safety, CENELEC, 2017.

[36] D. Ludicke and A. Lehner, "Train Communication Networks - Status and Prospect," IEEE, 2019.

[37] OCORA, "CCS-OB System Architecture," 2025. [Online]. Available: https://github.com/OCORA-Public/Publications/tree/master/00_OCORA%20Latest%20Publications/Latest%20Release. [Accessed 27 10 2025].