OETCS/WP3/D3.5.1.1

openETCS

ITEA2 Project
Call 6 11025
2012 – 2015

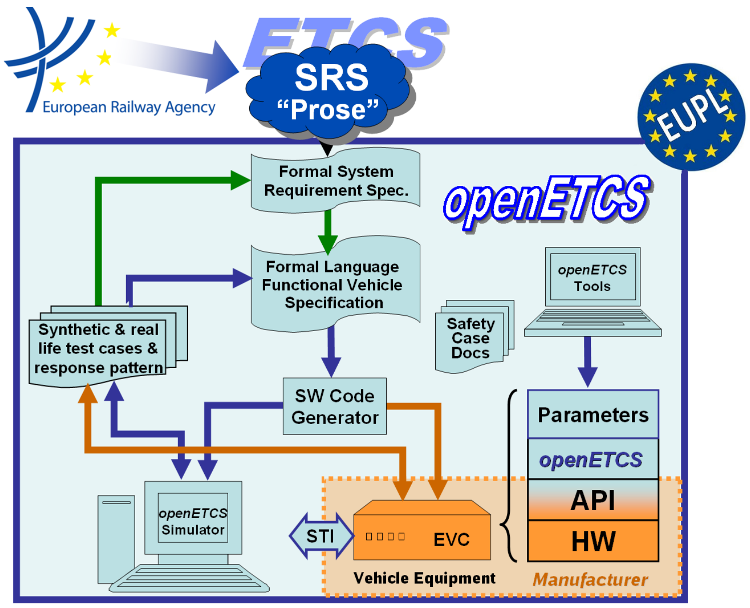
Work-Package 3: “Modeling”

openETCS System Architecture and Design Specification

First Iteration: ETCS Kernel Functions

Bernd Hekele, Peter Mahlmann, Peyman Farhangi, Uwe Steinke, Christian Stahl and David Mentré

October 2014



Funded by:

Federal Ministry of Education and Research
Région de Bruxelles-Capitale
GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, ENERGÍA Y TURISMO

Work-Package 3: “Modeling”

OETCS/WP3/D3.5.1.1
October 2014

openETCS System Architecture and Design Specification

First Iteration: ETCS Kernel Functions

Document approbation

Lead author:	Technical assessor:	Quality assessor:	Project lead:
location / date	location / date	location / date	location / date
signature Bernd Hekele (DB-Netz)	signature Uwe Steinke (Siemens)	signature Izaskun de la Torre (SQS)	signature Klaus-Rüdiger Hase (DB Netz)

Bernd Hekele, Peter Mahlmann, Peyman Farhangi

DB-Netz AG
Völckerstrasse 5
D-80959 München, Germany

Uwe Steinke

Siemens AG

Christian Stahl

TWT-GmbH

David Mentré

Mitsubishi Electric R&D Centre Europe

Architecture and Design Specification

Prepared for openETCS@ITEA2 Project

Abstract: This document gives an introduction to the architecture of the first openETCS iteration, the openETCS kernel functions. It has to be read as an add-on to the models in SysML, Scade and to additional reading referenced from the document.

Modification History

Version	Section	Modification / Description	Author
0.1	all	Initial document providing the structure	Bernd Hekele
0.2	3.5.3	initial contribution and some pretty printing	Christian Stahl
0.3	all	collecting feedback and completion on initial sections	Bernd Hekele
0.4	all	changed document style to openETCS report improved pretty printing	Peter Mahlmann
0.4.1	all	adding content	Bernd Hekele

Table of Contents

Figures and Tables

Figures

Tables

1 Introduction

1.1 Motivation

The openETCS work package WP3 aims to provide the architecture and the design of the openETCS OBU software as mainly specified in UNISIG Subset-026 version 3.3.0 [**?**].

The appropriate functionality has been divided into a list of functions of different complexity (see the WP3 function list [**?**]).

All these functions are object of the openETCS project and have to be analysed from their requirements and subsequently modelled and implemented. With limited manpower, a reasonable selection and order of these functions is required for the practical work that allows the distribution of the workload, more openETCS participants to join and leads to an executable function providing a limited kernel functionality as soon as possible.

While the first version of this document focuses on the first iteration of work, i.e. the limited kernel function, the document is intended to grow in parallel to the growing openETCS software.

1.2 Objectives

The first objective of WP3 software shall be

> “Make the train run as soon as possible, with minimum functionality, and in the form of a rapid prototype”.

Note that this does not contradict the openETCS goal to conform to EN50128. After a phase of prototyping, the openETCS software shall be implemented in compliance to EN50128 for SIL4 systems. The major goals of this document can be summarized as follows:

- Identification of the functions required for a minimum OBU kernel.
- Give an architecture overview regarding the minimum OBU kernel.
- Description of the technical approach, i.e. the process and methods to be used.
- Description of the “road map” of the minimum OBU kernel functions and the road map thereafter.

Note: This document will be extended according to the progress of WP3.

1.3 History

1.4 Goals of the openETCS Modelling Work

1.4.1 Functional Scope: The Minimum OBU Kernel Function

The objective “Make the train run with a very minimum functionality” (see Chapter 1.2) shall be in terms of ETCS OBU translated into

- “The Train moves on a track equipped with balises and determines its position.”

That means, for this very first step, the train shall neither supervise the maximum speed nor activate the brakes. The minimum function set shall be limited to:

- Receive, filter and manage balise information received from track (see `https://github.com/openETCS/SRS-Analysis/issues/12`);
- Calculate the actual train position based on balise and odometry information (see `https://github.com/openETCS/SRS-Analysis/issues/8`);
- Calculate the distances between the actual train position to track elements in its front.

The activities of the first iteration are collected in the openETCS WP3 backlog for the first iteration [**?**].

A more detailed architectural breakdown of these functions is available as a SysML model [**?**]. Diagrams used in the document at hand describing the architecture are taken from the model in [**?**].

The functional design is implemented in the Scade Model [**?**]. Design documents are taken from this model. Design diagrams used in this document are generated from the model. The design documents produced from the scade model are provided in the design location on Github [**?**].

In addition, the work on this minimum functionality requires to be supported by

- The availability of the ETCS language as specified in Subset UNISIG Subset_026, chapters 7 and 8;
- The abiltiy to link intermediate and final results with the requirements of the ETCS specification (subset_026, etc.).

These supporting prerequisites are under construction and therefore currently not completely operable. How to deal with these restrictions, will be outlined in Chapter 2.

1.5 Glossary and Abbreviations

API Application Programming Interface
BTM Balise Transmission Module
EVC European Vital Computer
LRBG Last Relevant Balise Group
SRS System Requirements Specification

2 The openETCS Architecture of the Initial Kernel Functions

2.1 The openETCS Tool Chain and its Impacts on the Actual Model

For understanding the modelling process and the modeling guidelines, we refer to [?].

To summarize the design process, the following rules are in use:

- Papyrus / SysML is used for modelling the architecture. Functions are visible on this SysML level.
- No behaviour model is allowed on SysML level.
- For referencing the requirements, links from the SysML model to the requirements document (in ProR) are being used.
- Details and especially behaviour is modelled in Scade.
- All interfaces (see also data-dictionary below) are available on bit-level.
- In the architecture model in SysML, all interfaces are available on a functional level for interfaces inside and outside the model and for interfaces between dedicated functions. Due to tool constrains the current model does not show all details for all interfaces (see the data dictionary [?]).

The openETCS tool-chain for doing the modelling work consists of the following components:

Papyrus for modelling the architecture (Kepler version). In this phase only the Kepler version of the tool can be used due to incompatibilities of the Kepler and the Luna version on the SysML model. The SysML models are stored at `https://github.com/openETCS/modeling/tree/master/model/sysml`.

ProR for keeping the requirements (REQIF). The subset 26 is converted into a REQIF-format and also stored in the modeling repository on Github. The openETCS toolchain supports the linking of SysML model parts to SRS-Requirements. These results are also part of the architecture.

Scade for designing and formalising the functions Scade version 15.2 is used. The models are stored at `https://github.com/openETCS/modeling/tree/master/model/Scade`. With the component Scade System Scade also has a component for designing the architecture.

In principle, the synchronisation mechanism of Scade was planned to be used for synchronising the SysML architecture and the Scade models. The idea is to automatically synchronise the SysML types and blocks with the Scade type definitions and the Scade Operators. Unfortunately, with the current set of tools this idea cannot be realised. Instead, we start developing a new

Papyrus plug-in which can be used for generating scade models according to the defined SysML blocks

In addition, faults in the Kepler Papyrus version made it difficult for several members of the team to work on different submodels of the openETCS model. The issue will be solved when changing to the Luna version of Papyrus.

2.2 The openETCS Application Software Architecture

The following diagrams are taken from the SysML model [?].

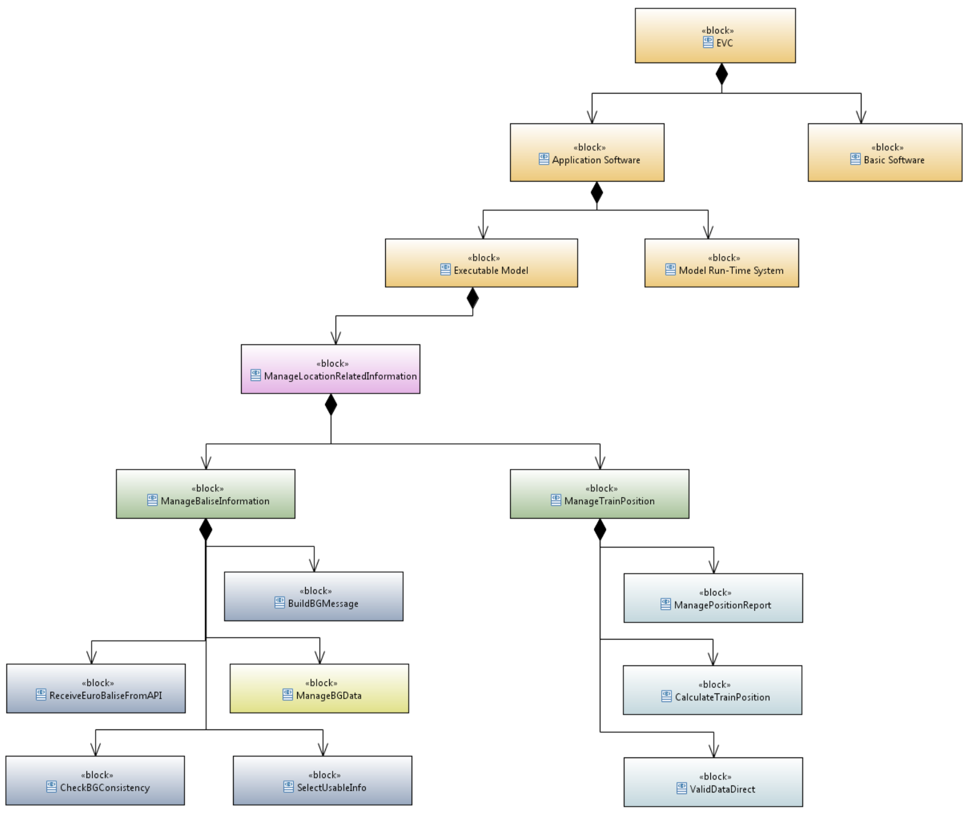

«block» EVC
«block» Application Software
«block» Basic Software
«block» Executable Model
«block» Model Run-Time System
«block» ManageLocationRelatedInformation
«block» ManageBaliseInformation
«block» ManageTrainPosition
«block» BuildBGMessage
«block» ManagePositionReport
«block» ReceiveEuroBaliseFromAPI
«block» ManageBGData
«block» CalculateTrainPosition
«block» CheckBGConsistency
«block» SelectUsableInfo
«block» ValidDataDirect

Figure 1. Block Definition Diagram of the First Iteration Architecture

The diagram shows the hierarchy of the EVC model. The boundaries of the model are given with the API (interfaces into and outside the EVC model), which actually is not part of the diagram. The runtime system of the EVC is also seen as a part outside the model.

Green blocks in this diagram are seen as data collected by the “train” without making use of the function in focus.

Input to the model is via the Model API (see section 3.1).

2.3 openETCS Data Dictionary

In the first iteration, the openETCS data dictionary gives some basic constructs for the project, the definition of interfaces in a central place and the definition of the ETCS language [?].

2.3.1 ETCS Language

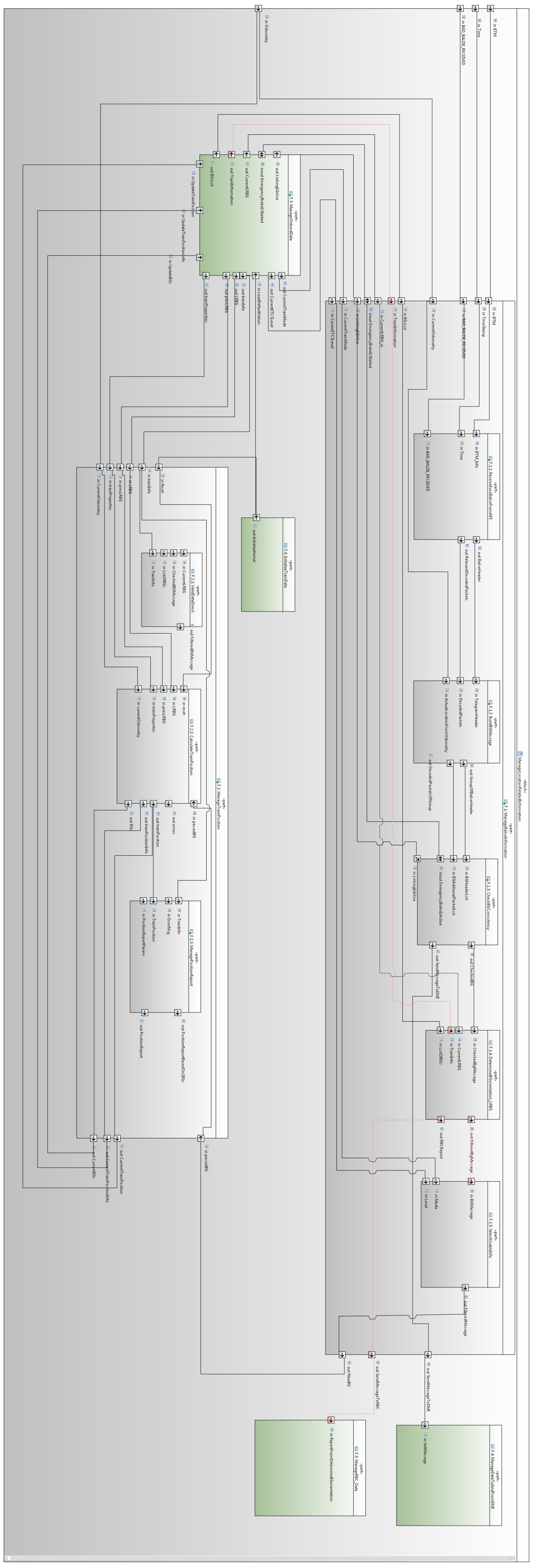


Figure 2. Internal Block Diagram of the First Iteration Architecture

The type definitions of Subset 26 chapters 7 and 8 (called the ETCS language) are provided as SysML resp. Scade types to the openETCS model. For the SysML model the types have been generated based on tools and provided as <>package<> imported to the openETCS toolschain.

2.3.2 openETCS Interfaces

Interfaces used between submodels and interfaces from outside and to outside the EVC kernel are defined as types in the data dictionary.

In the Scade model the ETCS language is available in the oETCS projects S026-7 and S026-8.

2.3.3 Data Dictionary Outlook

In the first iteration the use of the data dictionary concept is reduced to a minimum. The full openETCS process is tailored for a bigger team to cooperate and make use of tools to collect data and generate code.

In the Scade model the types needed to build the interface between models are defined in the projects Obu_Basic_Types.etp, BG_Types.etp, and TrainPosition_Types.etp.

3 openETCS Kernel Functions

3.1 openETCS Model Runtime System

The openETCS model runtime system also provides:

- Input Functions from other Units
  In this entity messages from other connected units are received. maybe use a descr
- Output Functions to other Units
  The entity writes messages to other connected units.
- Conversation Functions for Messages (Bitwalker)
  The conversion function are triggered by Input and Ouput Functions. The main task is to convert input messages from an bit-packed format into logical ETCS messages (the ETCS language) and Output messages from Logical into a bit-packed format. The logical format of the messages is defined for all used types in the openETCS data dictionary.
  Variable size elements in the Messages are converted to fixed length arrays with an used elements indicator.
  Optional elements are indicated with an valid flag. The conversion routines are responsible for checking the data received is valid. If faults are detected the information is passed to the openETCS executable model for further reaction.
- Model Cycle
  The executable model is called in cycles. In the cycle
  - First the received input messages are decoded.
  - The input data is passed to the executable model in a predefined order. **(Details for the interface to be defined)**.
  - Output is encoded according to the SRS and passed to the buffers to the units.

The openETCS system contains two APIs:

1. *openETCS API*: the interface specification between the EVC platform and the openETCS application;
2. *Model API*: the interface between the model itself written in SCADE and the surrounding runtime. Both the SCADE model and the runtime are making the openETCS application.

Figure 3 shows both openETCS API and model API on the software stack.

3.1.1 openETCS API

The openETCS API is currently defined by two documents, one written by Alstom [**?**] and a more abstract specification written by openETCS members [**?**].

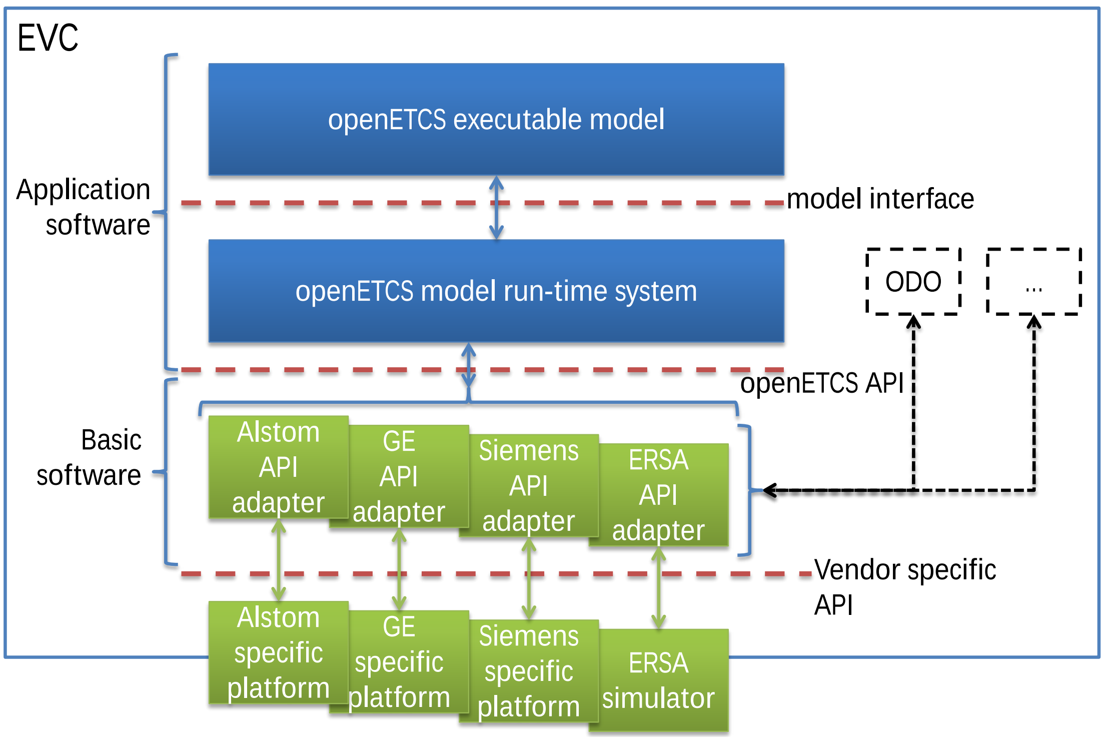


Figure 3. openETCS software architecture

The openETCS API defines the interfaces between the EVC platform and the openETCS application for the following units surrounding the EVC:

- TIU (Train Interface Unit),
- ODO (Odometry),
- DMI (Driver Machine Interface);
- STM (Specific Transmission Module, up to 8 units),
- BTM (Balise Transmission Module),
- LTM (Loop Transmission Module),
- EURORADIO,
- JRU (Juridical Recording Unit), and
- zero or more vendor specific units.

3.1.2 Model API

The model API is currently defined by the inputs and outputs of the SCADE model.

FIXME: How to give a precise pointer within the SCADE model? Reference to a specific block within the model?

For the proper working of the SCADE model, a set of assumptions are assumed:

- **Eurobalise (BTM)**: It is assumed that at most one “telegram” is provided per call of the SCADE model. This “telegram” is the merge of the telegrams of the balises making a balise group.

3.2 First Iteration: Model API and Model Runtime System

Note: The basic functions and the API is not implemented in the first iteration. Instead, the Scade test environment is used for demonstration. However, the interface used in the model is described as if they were available.

3.2.1 Short Description of Functionality

- Runtime System
  The Runtime System calls the openETCS kernel model in a cyclic way. Input parameters are updated with every cycle.
- Input
  - Control Interface
    The control interface triggers the reset of the application software. The reset is modelled as reset flag in the Scade model.
  - BTM Services
    The BTM service passes decoded telegrams to the executable model. The telegram is handed over to the API_balise parameter of the model (see Chapter 3.4.4). In each cycle only one telegram is expected to be passed.
  - Odometry
    The input from the odometry is updated with every cycle. The information from the odometry is updated in the parameter actual_odometry of the executable model. cycle=clock?
- Output
  - DMI
    The application may indicate errors at the balise group interface to the driver. The trigger needs to be passed to a corresponding function inside the application software (not part of the kernel).
  - Train Interface Unit (TIU)
    The application may need the service brake function. The trigger needs to be passed to a corresponding function inside the application software (not part of the kernel).
  - EuroRadio
    The application may trigger the position report via radio interface. The message needs to be passed to a corresponding function inside the application software (not part of the kernel).

3.3 First Iteration: Interfaces to other Functions of the Application Software (not part of First Iteration)

This section mainly refers to interfaces replaced by data implemented in the model.

3.3.1 Short Description of Functionality

Train-Info
The parameters of the train are seen as a constant to the kernel model. The data re not changed by any function of the kernel model. Only data used in the kernel are listed here. Used Information: level and mode.

3.4 F.1 Manage Balise Information

3.4.1 Short Description of Functionality

"ManageBaliseInformation" manages information related to balise telegrams received via the API when the train passes a balise. Balise telegrams are collected to build balise group messages. Finally, the message is checked for consistency, the train direction is calculated and the balise group message is passed to the other functions.

Information of the odometer is used to control for the train leaving the expectation window of the balises.

3.4.1.1 Input

- reset (bool) Request a reset of the data in the function. If reset=true no other input to the model is valid.
- API Telegram
  The telegram is build from
  - a present flag (bool)
    Indicates the input decoded telegram parameter is “present”, i.e., the input has been updated by the API. Only if the telegram is present the position information (incenterOfBalise) is to be used.
  - the decoded telegram including optional packets received from the balise.
  - the centerOfBalisePosition parameter. This parameter is used to give the position where the BTM has recognised the center of the balise telegram.
- inActualOdometry
  Actual Information giving the odometry of the train.
- LRBG
  The Last Relevant Balise Group. The information has been collected before by the train position function.

3.4.1.2 Output

- BG-Message
  Information describing the actual balise group just received.

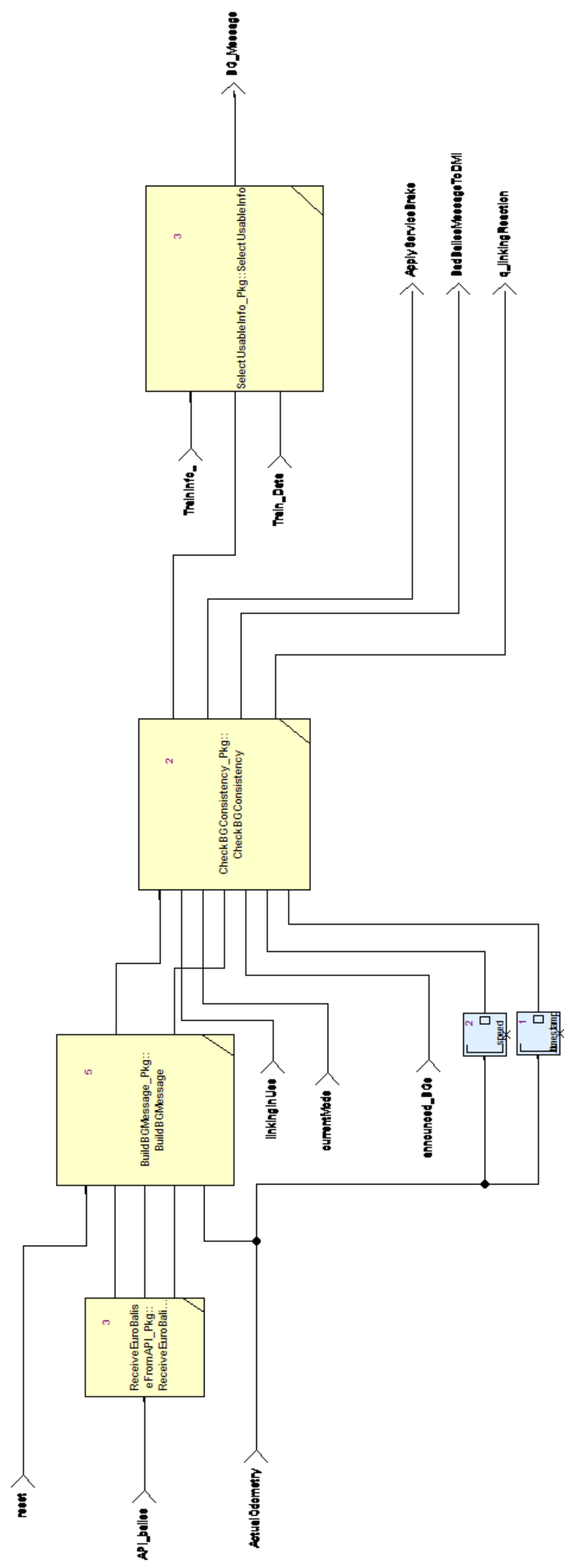


Figure 4. Structure of Manage Balise Information Block

- ApplyServiceBreak
  The flag indicates the balise group the train just passed could not be processed correctly. The check results in the request for a service break.

- BadBaliseMessageToDMI
  Information to be passed to the DMI to indicate the reception of a "bad balise" to the driver.

3.4.1.3 Data

- The function makes use of internal data for collecting and checking the balise telegrams.

3.4.2 Reference to the SRS (or other requirements)

- Definition of the Balise Telegram: subset 26 section 7 and 8
- Interface to the BTM: Subset 36, section 4.2.2, 4.2.4, 4.2.9
- Handling of Balise Telegrams: Subset 26, sections 3.4.1 - 3.4.3, 3.16.2
- Check of the balise group Subset 26, section 3.16.2
- Determining the Orientation: 3.4.2

3.4.3 Design Constraints and Choices

3.4.4 F.1.1 Receive Eurobalise From API

3.4.4.1 Short Description of Functionality

This function defines the interface of the OBU model to the openETCS generic API for Eurobalise Messages. On the interface, either a valid telegram is provided or a telegram is indicated which could not be received correct when passing the balise. The function passes the telegram without major changes of the information to the next entity for collecting the balise group information.

3.4.4.2 Reference to the SRS (or other requirements)

- Definition of the Balise Telegram: subset 26 section 7 and 8
- Interface to the BTM: Subset 36, section 4.2.2, 4.2.4, 4.2.9

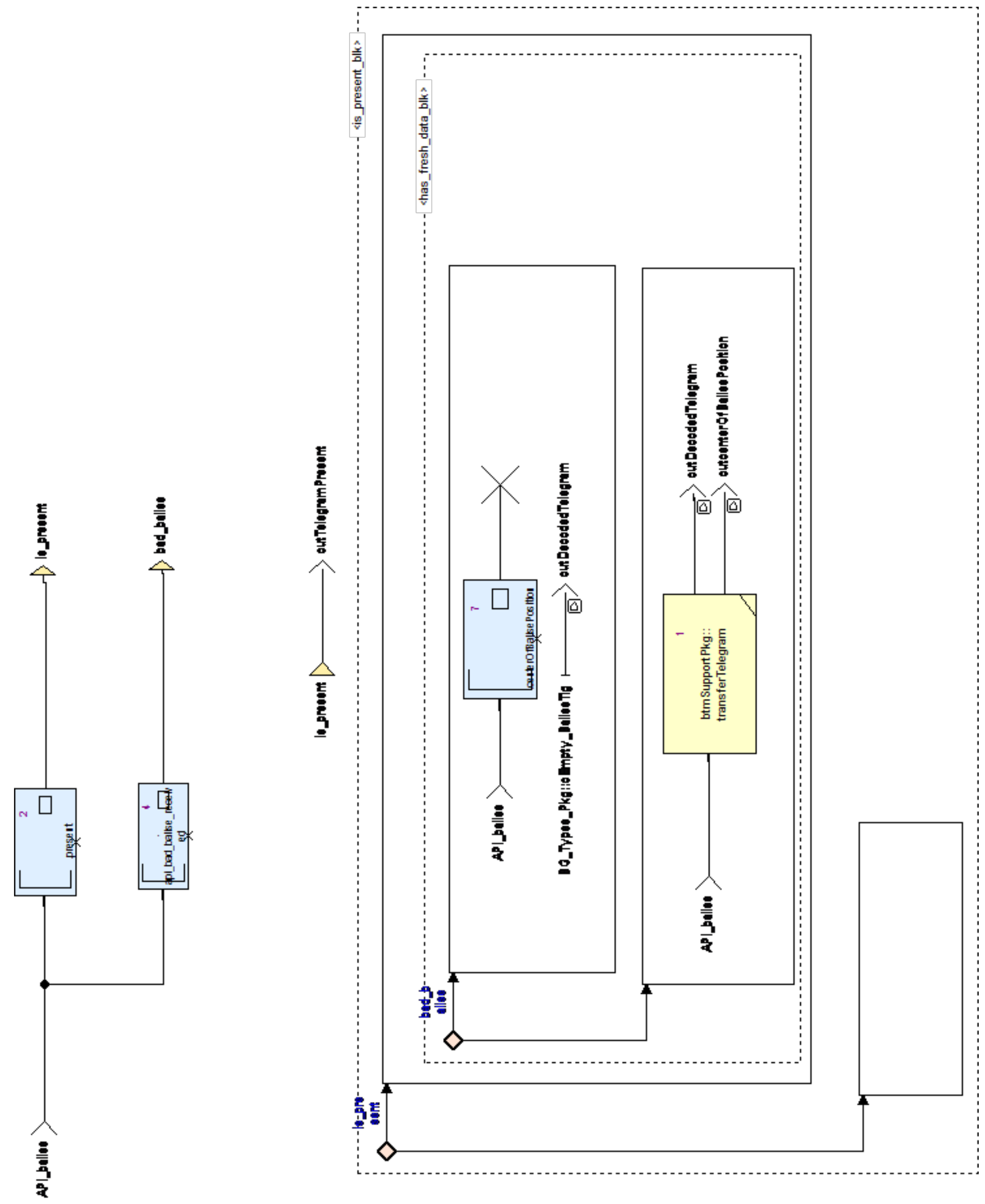


Figure 5. Structure of ReceiveEuroBaliseFromAPI

3.4.4.3 Design Constraints and Choices

1. The decoding of balises is done at the API. Also, packets received via the interface are already transformed into a usable shape.
2. Only packets used inside the current model are passed via the interface:
   Packet 5: Linking Information.
   Linking Information is added to the linking array starting from index 0 without gaps. Used elements are marked as valid. Elements are sorted according to the order given by the telegram sequence.

3.4.5 F.1.2 Build BG Group Message

3.4.5.1 Short Description of Functionality

This entity collects telegrams received via the interface into Balise Group Information.

3.4.5.2 Reference to the SRS (or other requirements)

- Interface to the BTM: Subset 36, section 4.2.2, 4.2.4, 4.2.9

- Handling of Balise Telegrams: Subset 26, sections 3.4.1 - 3.4.3, 3.16.2

3.4.5.3 Design Constraints and Choices

1. Telegrams received as invalid are passed to the “Check-Function” to process errors in communication with the track side according to the requirements and in a single place. Telegrams are added to the telegram array starting from index 0 without gaps. Used elements are marked as valid. Elements are stored according to the order given by the telegram sequence.
2. This function does not process information from the packets. The information is passed to the check without further processing of the values.

3.4.6 F.1.3 Check BG Consistency

3.4.6.1 Short Description of Functionality

This function has the task to verify the completeness and correctness of the received messages from balis-groups.
A message consists of at least a telegram and a maximum of 8 telegrams.

- A message is still complete and correct, if a telegram is missing (or not decoded or incomplete decoded), and this telegram is duplicated within the balise group and the duplicating one is correctly read.

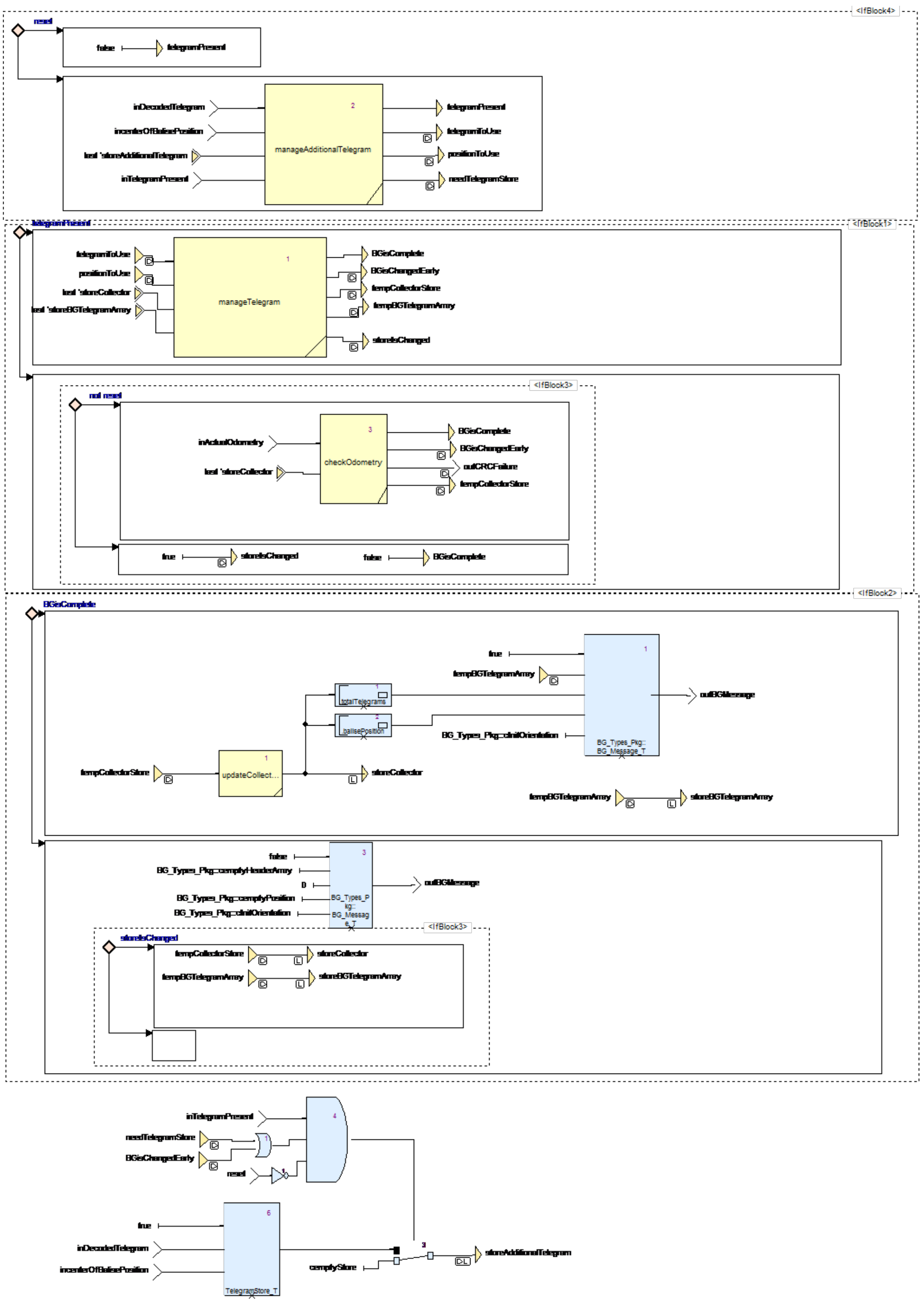


Figure 6. Structure of BuildBGMessage

- By more than one telegram, the order of the telegrams must be either ascending (nominal) or Descending(reverse).

- A message is correct, if all message counters (M MCUNT) do not equal 254 (that means: The telegram never fits any message of the group).
  A message counter can be equal 255 (that means: The telegram fits with all telegrams of the same balise group) and all other values must be the same.

For a correct balise group the balise group message is generated and passed to the system. In error situations the triggers for the driver and the breaking system are generated. I

3.4.6.2 Reference to the SRS (or other requirements)

- Check of the balise group Subset 26, section 3.16.2

- Determining the Orientation: 3.4.2

- Active Functions Table: 4.5.2

3.4.6.3 Design Constraints and Choices

This function is active in certain modes and the output and reactions are dependent on if the linking information is used.
The orientation of the BG will also be calculated in this block. The check, if the message has been received in due time and the right at the right expected location, will be performed in "Calculate Train Position".
The checks on the validity of the data in the packets and the validity with respect to the direction of motion will be performed in other modules, e.g. "Validate Data Direction" .

3.4.7 F.1.4 Determine BG- Orientation and LRBG

The orientation of the Balise Group is already determined in the check procedure. Due to the relocation of functions this block is not used any more.

3.4.8 F.1.5 Select Usable Info

Remark 1. *This function has to be seen as a separate part of the system, since the filter of this function is not limited to balise messages but also filters radio messages. Thus, the architecture design will be corrected accordingly in the next iteration of the modelling activities.*

3.4.8.1 Short Description of Functionality

The function Select Usable Info filters information received from balises that have been passed, radio messages, and EUROLOOP messages. Filtering is done depending on the mode of the train, the current ETCS level, the type/content of the information, and the transition media of the information. As neither radio messages nor EUROLOOP are part of the first iteration of work, not all functionality of the filter described in the specification is currently implemented.

3.4.8.2 Reference to the SRS (or other requirements)

The functionality of Select Usable Info is described in Chapter 4.8 of subset-026 [**?**]. The following list gives an overview of the most important sections for each of the blocks in the model.

First filter The first filter, i.e. the filter on the level, is described in [**?** , Chapter 4.8.3].

Second filter The second filter, i.e. the filter on the transition media, is described in[**?** , Chapter 4.8.3].

Third filter The third filter, i.e. the filter on the modes, is described in [**?** , Chapter 4.8.4].

Transition buffers Details on the handling of the transition buffers used in the first and the second filter are described in [**?** , Chapter 4.8.5].

3.4.8.3 Design Constraints and Choices

The first iteration of the model takes only balise group messages into account. This implies that a large part of the specification of this function described in subset-026 [**?**] is not relevant for the first iteration. This in particular applies to the second filter, i.e. filter on the transition media, because radio messages are not part of the model so far. Moreover, the functionality of the first filter, i.e. filter on the level, is currently limited because the first iteration of the model implements ETCS level 1 only.

3.5 F.2 Manage Train Position

3.5.1 F.2.1 Validate Data Direction

3.5.1.1 Short Description of Functionality

This function determines for direction information of the LRBG or an (ordinary) balise group whether this information is valid or not. The function takes as an input the LRBG and the balise groups passed and outputs the input extended with validity information.

3.5.1.2 Reference to the SRS (or other requirements

The functionality is mainly described in [**?** , Chapter 3.6.3].

3.5.1.3 Design Constraints and Choices

none

3.5.2 F.2.2 Calculate Train Position

3.5.2.1 Short Description of Functionality

The main purpose of the function is to calculate the locations of linked and unlinked balise groups (BGs) and the current train position while the train is running along the track.

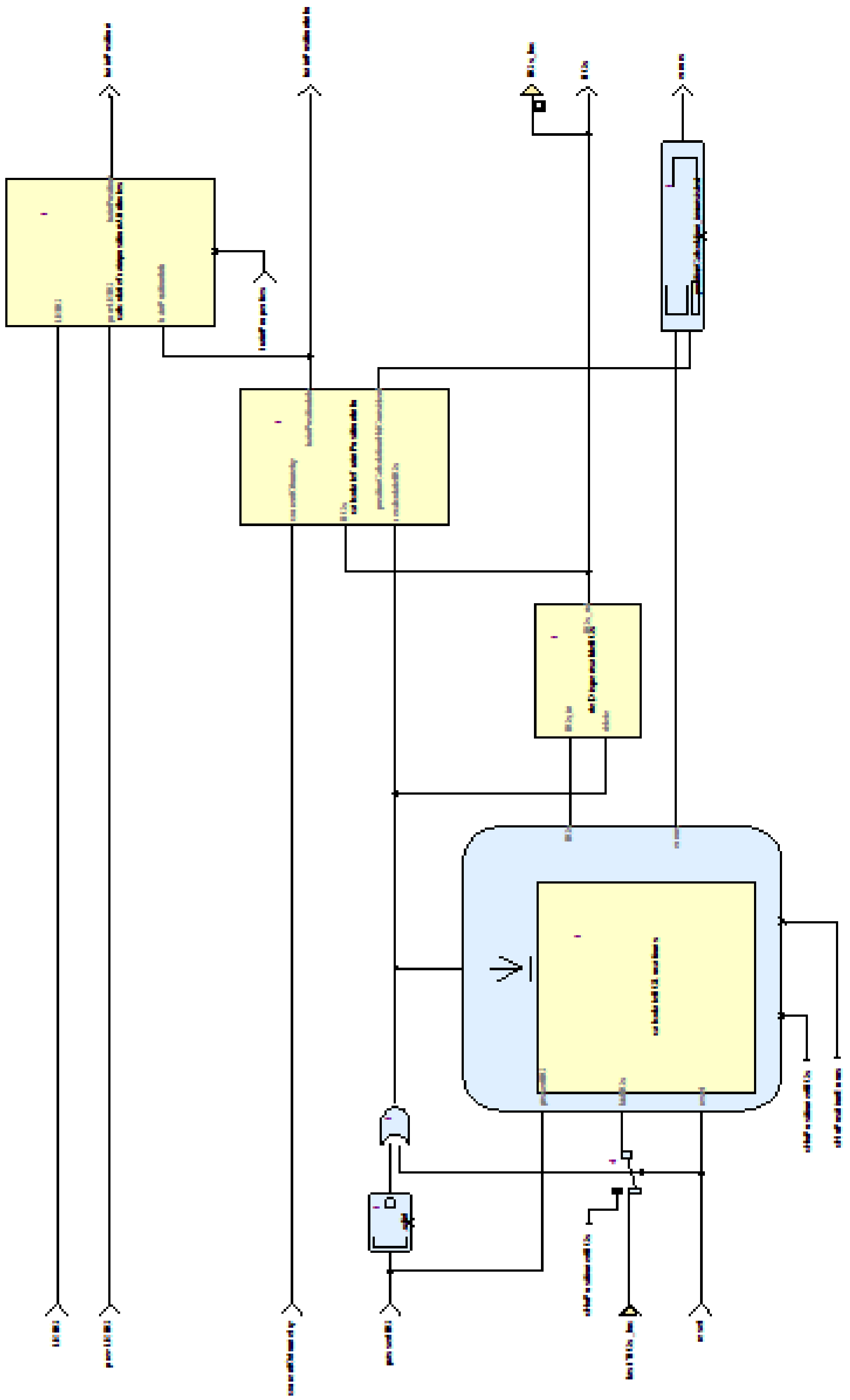


Figure 7. Structure of calculateTrainPosition

Functional Structure in Stages

The function calculateTrainPosition is divided into the following four functions, which are being performed sequentially:

1. ***calculateBGLocations***: Calculate the balise group locations
   The first stage is triggered each time the train passes a balise group (input *passedBG*). It takes the balise group header with the BG identification, the linking information (Subset 26, packet 5) and the current odometry values as inputs and calculates the location of the the passed balise group. If the passed BG has been announced via linking information previously, it takes into account the linking as well as the odometry information. If the passed BG does not meet the tolerance window announced by linking, an error flag is set. If the passed BG is an unlinked BG, its location is determined by odometry only, but related to the next previously passed linked BG, if there is one.
   Then, if the passed BG is a linked BG comprising linking information for BGs ahead, the linking information is evaluated by creating the announced BGs and computing their locations from the linking distances.
   The passed and the announced BGs are stored in a list *BGs*, ordered by their nominal location on the track.
   Afterwards the locations of all BGs are further improved by re-adjusting their locations with reference to the just passed BG. This optimizes the BG location inaccuries around the current train position (= location of the passed BG).

2. ***delDispensableBGs***: Delete dispensable balise groups
   The second stage removes balise groups supposed not to be needed any longer from the list of *BGs*.
   If the number of stored passed linked BGs exceeds the maximum number of eight as specified in [**?** , Chapter 3.6.2.2.2 c], all BGs astern are deleted. If only (passed) unlinked BGs are in the list and exceed the number of *cNoOfAtLeast_x_unlinkedBGs*, all passed BGs astern to those are removed from the list.

3. ***calculateTrainPositionInfo***: Calculate train position information.
   This stage take the list of stored BGs and the current odometry values as inputs and steadily provides the current train position.

4. ***calculateTrainpositionAttributes***: Calculate train position attribute information.
   This stage provides several additional position related attributes that might conveniently be used by subsequent consumers in the architecture. It requires the actual LRBG and the previous LRBG to be assigned external from the list *BGs*.

3.5.2.2 Reference to the SRS (or other requirements)

The component calculateTrainPosition determines the location of linked and unlinked balise groups and the current train position during the train trip as specified mainly in [**?** , Chapter 3.6].

3.5.2.3 Design Constraints and Choices

The following constraints and prerequisites apply:

1. The input data received from the balises groups must have been checked and filtered for validity, consistency and the appropriate train orientation before delivering them to calculate-TrainPosition.

2. The storage capacity for balise groups is finite. calculateTrainPosition will raise an error flag when a balise group cannot be stored due to capacity limitations.
3. calculateTrainPosition will raise an error flag if a just passed balise group is not found where announced by linking information. It will not (yet) detect when an announced balise group is missing.
4. calculateTrainPosition is not yet prepared for train movement direction changes.
5. calculateTrainPosition does not yet consider repositioning information.

3.5.3 Provide Position Report

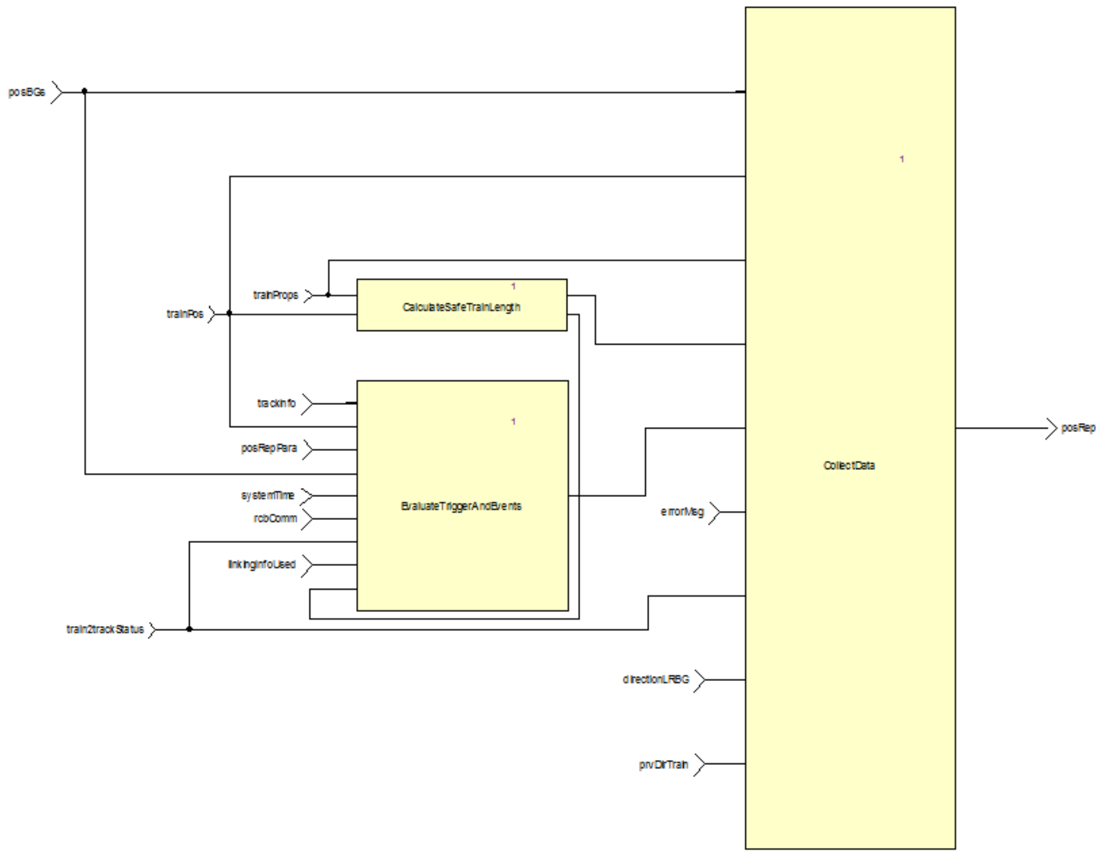


Figure 8. Structure of component ProvidePositionReport

3.5.3.1 Short Description of Functionality

This function takes the current train position and generates a position report which is sent to the RBC. The point in time when such a report is sent is determined by events, on the one hand, and position report parameters—which are basically triggers—provided by the RBC or a balise group passed, on the other hand. The functionality is modeled using three operations, as shown in Figure 8, which are explained below.

CalculateSafeTrainLength Calculates the the safeTrainLength and the MinSafeRearEnd according to [**?** , Chapter 3.6.5.2.4/5].
`safeTrainLength = absolute(EstimatedFrontEndPosition - MinSafeRearEnd)`,
where `MinSafeRearEnd = minSafeFrontEndPosition - L_TRAIN`.

EvaluateTriggerAndEvents Returns a Boolean modelling whether the sending of the next position report is triggered or not. This value is the conjunction of the evaluation of all triggers (PositionReportParameters, i.e., Packet 58) and events (see [**?** , Chapter 3.6.5.1.4]).

CollectData This operation aggregates data of Packet 0, . . . , Packet 5 and the header to a position report.

3.5.3.2 Reference to the SRS (or other requirements

Most of the functionality is described in [**?** , Chapter 3.6.5].

3.5.3.3 Design Constraints and Choices

1. The message length (i.e., attribute `L_MESSAGE`) is by default set to 0; the actual value will be set by the Bitwalker/API.
2. The attribute `Q_SCALE` is assumed to be constant; that is, all operations using this attribute do not convert between different values of that attribute.
3. *PositionReportHeader*: The time stamp (i.e., attribute `T_TRAIN`) is not set; this should be done once the message is being sent by the API.
4. *Packet 4*: When aggregating data for this packet, an error message might be overwritten by a succeeding error message. Because the specification allows only to sent one error in one position report, errors are not being stored in a queue, for instance.
5. *Packet 44*: This packet is currently not contained in a position report as it is not part of the kernel functions.
6. The usage of attributes `D_CYCLOC` and `T_CYCLOC` as part of the triggers specified by the position report parameters (i.e., Packet 58 sent by the RBC) may lead to unexpected results if a big clock cycle together with small values for the attributes is used. The cause is that at every clock cycle the current model increments the reference value for the distance and time by at most `D_CYCLOC` and `T_CYCLOC`, respectively and not a factor of it.

3.5.3.4 Open Issues

1. The specification requires to store the last eight balise groups for which a position report has been sent (see [**?** , Chapter 3.6.2.2.2.c]).
2. For all reports that contain Packet 1 (i.e., report based on two balise groups), the RBC sends a coordinate system. It is unclear where this has to be stored (i.e., somehow the balise groups have to be stored in a database which has then to be updated), see [**?** , Chapter 3.4.2.3.3.6]. Moreover, such a coordination system can be invalid and then has to be rejected (see [**?** , Chapter 3.4.2.3.3.7-8]). On a more abstract level, we need to think about the interface between the RBC and the OBU or a proper abstraction thereof.