OETCS/WP7/O??

openETCS

ITEA2 Project
Call 6 11025
2012 – 2015

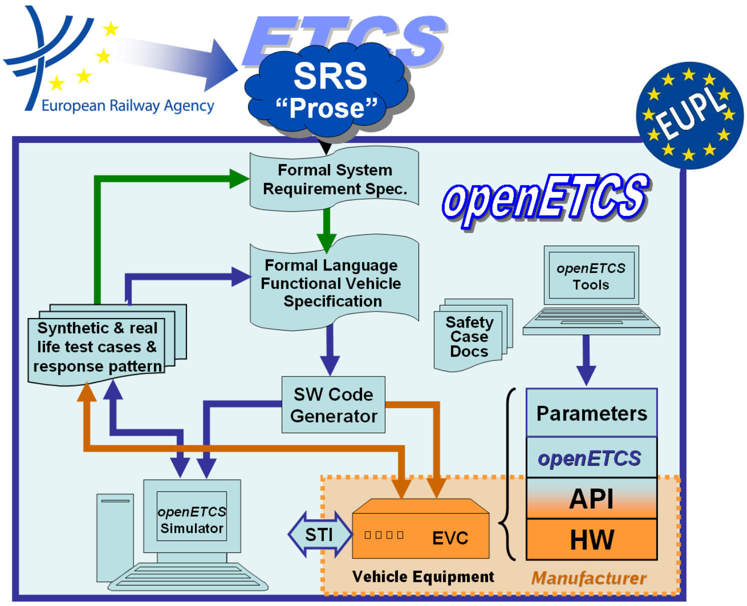
openETCS@ITEA Work Package 7: “Toolchain”

Evaluation model of ETCS using GNATprove

Tool and model presentation

David Mentré

18th of June 2013



Funded by:

Federal Ministry of Education and Research

Région de Bruxelles-Capitale

GOBIERNO DE ESPAÑA

MINISTERIO DE INDUSTRIA, ENERGÍA Y TURISMO

This page is intentionally left blank

openETCS@ITEA Work Package 7: "Toolchain" **OETCS/WP7/O??**
18th of June 2013

Evaluation model of ETCS using GNATprove

Tool and model presentation

David Mentré

Mitsubishi Electric R&D Centre Europe
1 allée de Beaulieu
CS 10806
35708 RENNES cedex 7

email: `d.mentre@fr.merce.mee.com`

Draft Report, version 2

Prepared for openETCS@ITEA2 Project

Abstract: This report outlines and then details the modeling of a subset of SRS SUBSET-026 using the GNATprove proving environment based on Ada 2012 language.

Table of Contents

Figures and Tables

Figures

Tables

1 Introduction

This document presents the use of Ada 2012 language and GNATprove tool to model a subset of ETCS. This subset is included in the subset defined in openETCS D2.5 document[2].

In this report, we firstly present the Ada 2012 language and GNATprove tool, then we give an overview of our model, its benefits and shortcoming to end with the detailed source code of the complete model.

2 Short introduction to Ada 2012 and GNATprove tool

This model is using Ada 2012 language[1]. The Ada language is a well known generic purpose language which first version was published in 1983 and which is normalized by ISO. After several revisions in 1995 and 2005, the latest 2012 revision offers interesting facilities for program verification. More specifically, function contracts can now be declared through pre and post-conditions using Pre and Post Ada annotations.

Those contracts can be compiled in executable code and checked dynamically at execution, or statically verified using a dedicated tool. GNATprove is such a static verification tool. It *automatically*[1] checks Ada 2012 contracts and absence of run-time exception (underflow, overflow, out of bound access, ...) for all possible executions. GNATprove is integrated into AdaCore's GNAT Programming Studio (GPS) environment.

GNATprove and GPS programming environment are Open Source software, licensed under GNU GPL.

All code of this report has been tested using GNAT GPL 2013 and SPARK Hi-Lite GPL 2013[2].

2.1 A short example

As example, here is the Ada 2012 specification of a Saturated_Sum function. The Post-condition states that if the sum of two integers X1 and X2 is below a given Maximum, then this sum is returned, otherwise the Maximum is returned.

The Pre-condition states that both X1 and X2 should be below the biggest Natural divided by 2, such that computation of the sum does not raise an overflow error at execution.

```ada
package Example is
   function Saturated_Sum(X1, X2, Maximum : Natural) return Natural
   with
     Pre => ((X1 <= Natural'Last / 2) and (X2 <= Natural'Last / 2)),
     Post => (if X1 + X2 <= Maximum then saturated_sum'Result = X1 + X2
              else saturated_sum'Result = Maximum);
end;
```

The implementation of this specification is rather obvious.

```ada
package body Example is
   function Saturated_Sum(X1, X2, Maximum : Natural) return Natural is
   begin
      if X1 + X2 <= Maximum then
         return X1 + X2;
      else
         return Maximum;
      end if;
   end Saturated_Sum;
```

[1] Automatic verification is made through several calls to SMT (Satisfiability Modulo Theories) solver *Alt-Ergo*.

[2] http://libre.adacore.com/tools/spark-gpl-edition/

```
10 end Example;
```

Applying GNATprove on above two files generates 5 Verification Conditions (VC) that are all *automatically* proved correct by the tool.

```
example.adb:4:13: info: overflow check proved [overflow_check]
example.adb:5:20: info: overflow check proved [overflow_check]
example.ads:5:14: info: postcondition proved [postcondition]
example.ads:5:21: info: overflow check proved [overflow_check]
example.ads:5:68: info: overflow check proved [overflow_check]
```

The first two VC check the sums X1 + X2 do not overflow on lines 4 and 5 of Saturated_Sum body.

The third VC checks that the post-condition of Saturated_Sum given on line 5 and 6 of its specification can be proved using the function body.

The last two VC check there are no overflow in the annotation X1 + X2 at the line 5 of the post-condition of Saturated_Sum. In fact, as those Pre and Post-conditions could be compiled to assertions checked at run-time, GNATprove checks that they cannot themselves trigger run-time exceptions.

After such verifications, we are sure that for all possible executions no run-time errors are going to be raised and that the Saturated_Sum function will fulfill its contract if called with pre-conditions satisfied.

3 Model overview

The model is organized along SUBSET 026: to each section or subsection corresponds an Ada **package** of the same name[3]. Like Uwe Steinke for its SCADE model, we have tried to formalize each paragraph of SUBSET 026, associating to it some Ada 2012 code.

For each part of the model, a comment gives the paragraph number it corresponds to, thus making a basic traceability.

We have also created additional packages for generic parts of the model or to define data structure used by several parts.

We have created Ada data types to describe specific objects of ETCS specification. We define new integer types, new structured types (arrays or records) and new enumerations. Those data types are in turned used to define Ada entities that represent ETCS specification objects in our model.

For example, the following **package** ETCS_Level defines the five ETCS levels 0 to 4 as **type** ertms_etcs_level_t. Within those levels, the NTC (aka STM) specific level is defined as number 4 with ertms_etcs_level_ntc **constant**. Then this data type is used for definition of ertms_etcs_level variable that describes current ETCS level in the model.

```ada
Package ETCS_Level is
   type ertms_etcs_level_t is range 0..4; -- SUBSET-026-2.6.2.3
   ertms_etcs_level_ntc : constant ertms_etcs_level_t := 4;

   ertms_etcs_level : ertms_etcs_level_t;
end;
```

For propositions in the text, e.g. "Start of Mission", we have used a Boolean variable, e.g. Start_Of_Mission.

[3] We are not sure this approach makes the model easily understandable, at least without a good knowledge of SUBSET 026. Naming packages after described entities (Communication, Balises, etc.) might be a better approach for a full fledged model.

4 Model benefits and shortcomings

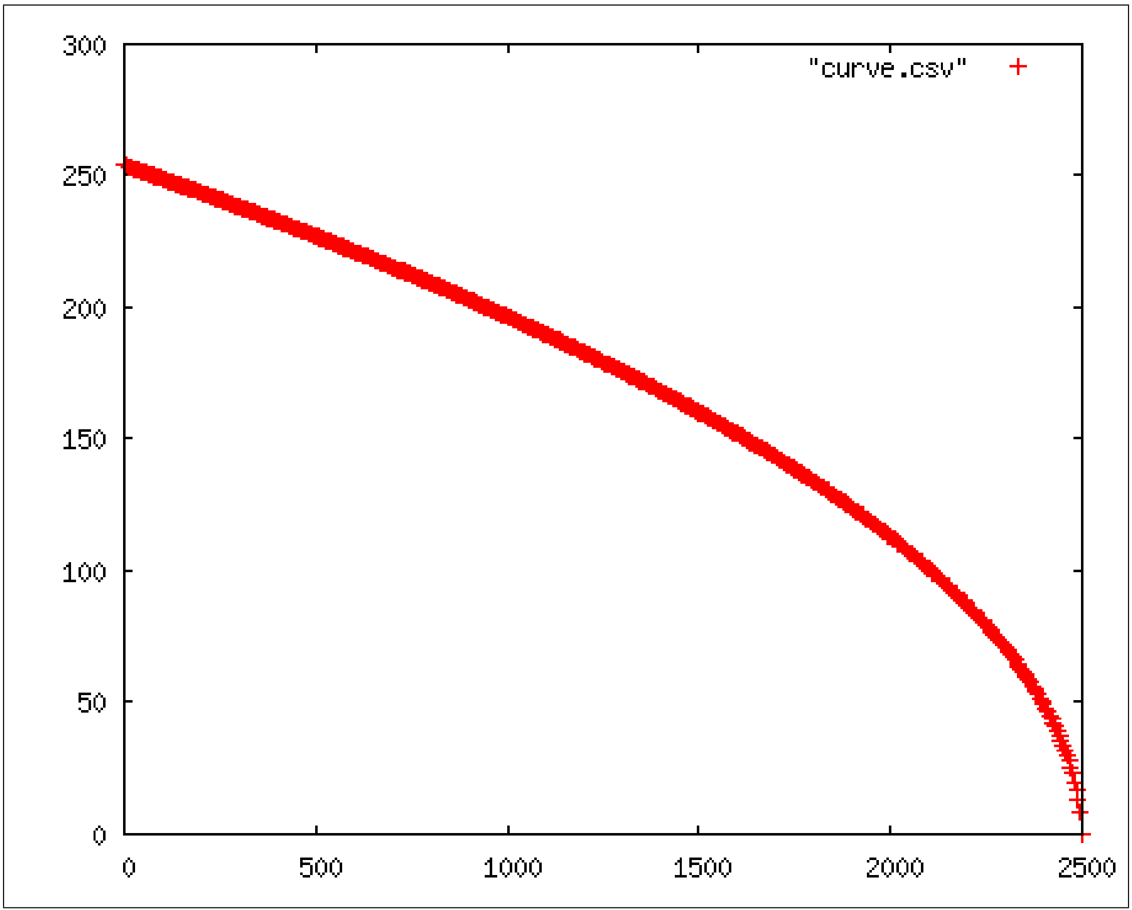


Figure 1. Basic braking curve for EBD, target is at 2,500 m at speed 0 km/h, deceleration is constant $-1m/s^2$

After modeling several chapters of the SRS, we see following advantages to using GNATprove:

- The model relies on the well known and well documented Ada language. It is easy to find documentation and tutorials on the web;
- We were able to describe all the different kinds of entities found in the SRS: transition tables, algorithms, data structures, requirements on functional or data parts, etc. The description of data types is very expressive and thus helps writing a readable model. We are able to compute a basic braking curve (see figure 1), which we consider a rather complex computation;
- The description is rather concise, which we see as a very positive point;
- By writing as comment the reference to the corresponding SRS paragraph, we have a basic but effective traceability. This approach could probably be automated, for example using tools to produce traceability matrix for verification purpose;
- We were able to express formally most of requirements found in the SRS and prove part of those;

- The model can be compiled and is executable. It can be interfaced with external entities like Graphical User Interface.

However some shortcomings have been identified:

- One needs to learn the Ada language in order to understand the model. The Ada language is big. The subset handled by GNATprove is nonetheless smaller and the concepts are quite well known for imperative languages (functions or procedures, variables, loops, records and arrays, ...);
- The model is not graphical. Especially for model overview, a graphical understanding would be helpful. It might be possible to provide such a graphical overview using Ada tools;
- One needs to build needed domain specific abstractions in order to write the model, e.g. the step functions defined in section 5.4.2. But even if this effort is not negligible, once this is done it can be capitalized over several models or model parts;
- The proofs are not complete and some parts cannot be done with the provided automated provers.

4.1 A note on proofs

One main goal of using GNATprove approach is to make proof on parts of the model. We have attempted to make such proof:

- On exclusion property that should be fulfilled by transition table of SRS section 4.6. We can show that the exclusion cannot be proved without using external assumptions;
- On the modeling of Step Functions. On them, we have proved most properties but some Verification Condition (VC) are not proved or cannot be proved.

In case a VC is not proved, the GNATprove message is not very explicit. One should select the line and do “Proof::Prove Line” on it: GNATprove will give more information on the error, for example by splitting conjunctions and detailing specific cases that are not proved.

GNATprove tool is handling quite well integer types. It is therefore relatively easy to prove absence of Run Time Errors like overflow, underflow or out-of-bound accesses. Regarding floating point numbers, the situation is much less rosy. Even if a basic mapping to Mathematical real numbers in the prover is provided, it is much more difficult to prove range checks.

More fundamentally, we are formally specifying or verifying detailed algorithms described in the SRS, but the underlying principles, i.e. those related to safety, are not explicitly stated. Therefore we are not sure such a modeling effort would be worth it for a real development (except maybe for absence of Run Time Error).

5 Detailed model description

The model is organized into a set of ADS (Ada Package Specification) or ADB (Ada Package Body or program) files. The ADS file declare each function and procedure and optionally the contract it should fulfill using Pre => and Post => notation. In those parts, "**for all**" stands for mathematical "∀" and "**for** some" stands for "∃". The ADB file contains code corresponding to the function or procedure declaration. As we are building a *description* of the SRS without making it executable, the ADB file is often empty or missing.

Each Ada package, an Ada ADS and an Ada ADB files, corresponds to a section of the SUBSET-026 SRS. Each paragraph of the SRS is modeled by one or more Ada data type definition or function definition. We have extensively used user defined data type definition to strongly type the different kinds of identifiers in the model. For example, an ertms_etcs_level_t is a different integer that RBC_RIU_ID_t.

5.1 Generic parts of the model

In this part, we define relatively simple data types or model variables reused in other parts of the model.

```
package Data_Types is
   type Telephone_Number_t is range 0..20_000; -- FIXME refine range

   type RBC_Contact_Action_t is (Establish_Session, Terminate_Session);

   type RBC_RIU_ID_t is range 1..10_000; -- FIXME: refine range
end;
```

```

-- Reference: UNISIG SUBSET-026-3 v3.3.0

Package ETCS_Level is
   type ertms_etcs_level_t is range 0..4; -- SUBSET-026-2.6.2.3
   ertms_etcs_level_ntc : constant ertms_etcs_level_t := 4;

   ertms_etcs_level : ertms_etcs_level_t;
end;
```

```

-- Reference: UNISIG SUBSET-026-3 v3.3.0

Package Appendix_A_3_1 is
   number_of_times_try_establish_safe_radio_connection : constant Integer := 3;

   driver_acknowledgment_time : constant Integer := 5; -- seconds
   t_ack : constant Integer := driver_acknowledgment_time;

   M_NVAVADH : constant Float := 0.0;
end;
```

5.1.1 SUBSET-026-4.3.2 ETCS modes

```

```

```
-- Reference: UNISIG SUBSET-026-3 v3.3.0

Package Section_4_3_2 is
   type etcs_mode_t is (Full_Supervision,
                        Limited_SUpervision,
                        On_Sight,
                        Staff_Responsible,
                        Shunting,
                        Unfitted,
                        Passive_Shunting,
                        Sleeping,
                        Stand_By,
                        Trip,
                        Post_Trip,
                        System_Failure,
                        Isolation,
                        No_Power,
                        Non_Leading,
                        National_System,
                        Reversing);

   type etcs_short_mode_t is (FS,
                              LS,
                              OS,
                              SR,
                              SH,
                              UN,
                              PS,
                              SL,
                              SB,
                              TR,
                              PT,
                              SF,
                              ISo, -- "IS" is a reserved Ada keyword
                              NP,
                              NL,
                              SN,
                              RV);
end;
```

5.2 SUBSET-026-4.6 Transitions between modes

Here we define as Boolean variables each condition that should be fulfilled or not to make a transition from one mode to another, e.g. driver_selects_shunting_mode. In a more complete model, those variables would probably be defined and handled in other packages.

We then define a set of functions that correspond to each individual condition defined in table §4.6.2 of the SRS. See for example condition_1.

Those conditions are then in turn combined into a set of functions defining the condition under which a mode transition can occur. See for example condition_transition_SB_to_SH.

We end with a function transition of which Contract_Cases expresses that the different transition condition should be disjoint and complete, in order to prove it. This cannot be proved using only the assumptions describe in §4.6 of the SRS. See `https://github.com/openETCS/model-evaluation/wiki/Open-Question-for-Modeling-Benchmark#section-46` for a detailed example.

```
-- Reference: UNISIG SUBSET-026-3 v3.3.0

with ETCS_Level;
use ETCS_Level;

with Section_4_3_2;
use Section_4_3_2;

Package Section_4_6 is
   -- SUBSET-026-4.6.3 Transitions Conditions Table
   -- WARNING: not all conditions are modeled

   -- Individual condition elements
   an_acknowledge_request_for_shunting_is_displayed_to_the_driver : Boolean;

   driver_acknowledges : Boolean;

   driver_isolates_ERTMS_ETCS_on_board_equipment : Boolean;

   driver_selects_shunting_mode : Boolean;

   ma_ssp_gardient_on_board : Boolean;

   no_specific_mode_is_required_by_a_mode_profile : Boolean;

   note_5_conditions_for_shunting_mode : Boolean;

   reception_of_information_shunting_granted_by_rbc : Boolean;

   train_is_at_standstill : Boolean;

   valid_train_data_is_stored_on_board : Boolean;

   -- Conditions
   function condition_1 return Boolean is
     (driver_isolates_ERTMS_ETCS_on_board_equipment);

   function condition_5 return Boolean is
     (train_is_at_standstill
      AND (ertms_etcs_level = 0 OR ertms_etcs_level = ertms_etcs_level_ntc
             OR ertms_etcs_level = 1)
      AND driver_selects_shunting_mode);

   function condition_6 return Boolean is
     (train_is_at_standstill
      AND (ertms_etcs_level = 2 OR ertms_etcs_level = 3)
      AND reception_of_information_shunting_granted_by_rbc);

   function condition_10 return Boolean is
     (valid_train_data_is_stored_on_board
      AND ma_ssp_gardient_on_board
      AND no_specific_mode_is_required_by_a_mode_profile);

   function condition_50 return Boolean is
     (an_acknowledge_request_for_shunting_is_displayed_to_the_driver
      AND driver_acknowledges
      AND note_5_conditions_for_shunting_mode);

   -- SUBSET-026-4.6.2 Transitions Table
   type priority_t is range 1..7;
   priority : priority_t;

   function condition_transition_SB_to_SH return Boolean is
```

```
      ((condition_5 OR condition_6 OR condition_50) AND priority = 7);

   function condition_transition_SB_to_FS return Boolean is
      (condition_10 AND priority = 7);

   function condition_transition_SB_to_IS return Boolean is
      (condition_1 AND priority = 1);

   -- following function is no longer needed
   function disjoint_condition_transitions return Boolean is
      (NOT(condition_transition_SB_to_SH = True
           AND condition_transition_SB_to_FS = True)
       AND NOT(condition_transition_SB_to_SH = True
               AND condition_transition_SB_to_IS = True)
       AND NOT(condition_transition_SB_to_FS = True
               AND condition_transition_SB_to_IS = True));

   function transition(mode : etcs_mode_t) return etcs_mode_t
   with
--         Post => (disjoint_condition_transitions = True);
      -- SUBSET-026-4.6.1.5: all cases are disjoint
      Contract_Cases => (condition_transition_SB_to_SH => True,
                         condition_transition_SB_to_FS => True,
                         condition_transition_SB_to_IS => True),
      Post => True; -- work around for bug in SPARK Hi-Lite GPL 2013
end;
```

```

-- Reference: UNISIG SUBSET-026-3 v3.3.0

with Section_4_3_2;
use Section_4_3_2;

Package body Section_4_6 is
   function transition(mode : etcs_mode_t) return etcs_mode_t is
   begin
      return No_Power;
   end;
end;
```

5.3 SUBSET-026-3.5.3 Establishing a communication session

In this section, we attempt to model SRS §3.5.3.

5.3.1 Safe Radio package

This package emulates an API to open a connection and send messages on it.

```
with Data_Types;

package Safe_Radio is
   type Message_Type_t is (Initiation_Of_Communication);

   function Setup_Connection(phone : Data_Types.Telephone_Number_t)
                              return Boolean;
   -- return True if connection is setup, False otherwise

   procedure Send_Message(message : Message_Type_t);
end;
```

```
package body Safe_Radio is

   ---------------------
   -- Setup_Connection --
   ---------------------

   function Setup_Connection
     (phone : Data_Types.Telephone_Number_t)
      return Boolean
   is
   begin
      -- Generated stub: replace with real body!
      raise Program_Error with "Unimplemented function Setup_Connection";
      return False;
   end Setup_Connection;

   procedure Send_Message(message : Message_Type_t) is
   begin
      -- Generated stub: replace with real body!
      raise Program_Error with "Unimplemented function Setup_Connection";
   end Send_Message;
end Safe_Radio;
```

5.3.2 Com_map utility package

This package defines an abstract table that can be used to search if a connection has already been established, is being established or is not opened.

We have used an array for the definition of Com_To_RBC_Map. We would have preferred to use Ada formal containers, but those cannot be handled by GNATprove in its GPL 2012 edition.

```
-- Commented out because not supported in GNAT GPL 2012
--- with Ada.Containers.Formal_Hashed_Maps;
--- with Ada.Containers; use Ada.Containers;

with Data_Types; use Data_Types;

package Com_Map is
-- Commented out because not supported in GNAT GPL 2012
--     function RBC_RIU_ID_Hash(id : RBC_RIU_ID_t) return Hash_Type is
--        (Hash_Type(id));
--
--     package Com_To_RBC_Map is new Ada.Containers.Formal_Hashed_Maps
--        (Key_Type        => RBC_RIU_ID_t,
--         Element_Type    => Boolean, -- False: com being established
--                                     -- True : com established
--         Hash            => RBC_RIU_ID_Hash,
--         Equivalent_Keys => "=",
--         "="             => "=");
   type Com_Element is record
      Used             : Boolean := False; -- False: element not used
      Com_Established  : Boolean := False; -- False: com being established
                                           -- True : com established
   end record;

   type Com_To_RBC_Map is array (RBC_RIU_ID_T) of Com_Element;

   function Contains (Map : Com_To_RBC_Map; Id : RBC_RIU_ID_T) return Boolean
   is (Map (Id).Used and Map(Id).Com_Established);
end;
```

5.3.3 Section 3.5.3 modeling

The core of the model. We use the body of procedure Initiate_Communication_Session to detail the algorithm specified in the SRS.

```

-- Reference: UNISIG SUBSET-026-3 v3.3.0

with ETCS_Level; use ETCS_Level;

with Data_Types; use Data_Types;

with Com_Map; use Com_Map;

Package Section_3_5_3 is
   -- FIXME using SRS sections as package name is probably not the best approach

   -- SUBSET-026-3.5.3.4
   Start_Of_Mission : Boolean;
   End_of_Mission : Boolean;
   Track_Side_New_Communication_Order : Boolean;
   Track_Side_Terminate_Communication_Order : Boolean;
   Train_Passes_Level_Transition_Border : Boolean;
   Train_Passes_RBC_RBC_Border : Boolean;
   Train_Passes_Start_Of_Announced_Radio_Hole : Boolean;
   Order_To_Contact_Different_RBC : Boolean;
   Contact_Order_Not_For_Accepting_RBC : Boolean;
   Mode_Change_Report_To_RBC_Not_Considered_As_End_Of_Mission : Boolean; -- to be refined
   Manual_Level_Change : Boolean;
   Train_Front_Reaches_End_Of_Radio_Hole : Boolean;
   Previous_Communication_Loss : Boolean;
   Start_Of_Mission_Procedure_Completed_Without_Com : Boolean;

--      Connections : Com_To_RBC_Map.Map(Capacity => 10,
--                                       Modulus =>
--                                         Com_To_RBC_Map.Default_Modulus(10));

   Connections : Com_To_RBC_Map;

   function Authorize_New_Communication_Session return Boolean is
     ((Start_Of_Mission = True
       and (ertms_etcs_level = 2 or ertms_etcs_level = 3)) -- SUBSET-026-3.5.3.4.a
      and Track_Side_New_Communication_Order = True -- SUBSET-026-3.5.3.4.b
      and (Mode_Change_Report_To_RBC_Not_Considered_As_End_Of_Mission = True
           and (ertms_etcs_level = 2 or ertms_etcs_level = 3))-- SUBSET-026-3.5.3.4.c
      and (Manual_Level_Change = True
           and (ertms_etcs_level = 2 or ertms_etcs_level = 3)) -- SUBSET-026-3.5.3.4.d
      and Train_Front_Reaches_End_Of_Radio_Hole = True -- SUBSET-026-3.5.3.4.e
      and Previous_Communication_Loss = True -- SUBSET-026-3.5.3.4.f
      and (Start_Of_Mission_Procedure_Completed_Without_Com = True
           and (ertms_etcs_level = 2 or ertms_etcs_level = 3)) -- SUBSET-026-3.5.3.4.g
     );

   -- SUBSET-026-3.5.3.1 and SUBSET-026-3.5.3.2 implicitly fulfilled as we model on-board
   procedure Initiate_Communication_Session(destination : RBC_RIU_ID_t;
                                            phone : Telephone_Number_t)
   with
     Pre => ((Authorize_New_Communication_Session = True) -- SUBSET-026-3.5.3.4
             and (not Contains(Connections,destination)) -- SUBSET-026-3.5.3.4.1
             -- FIXME: what should we do for cases f and g?
            ),
     Post => (Contains(Connections, destination));

```

```
   -- SUBSET-026-3.5.3.3 not formalized (Note)

   -- SUBSET-026-3.5.3.5
   procedure Contact_RBC(RBC_identity : RBC_RIU_ID_t;
                         RBC_number : Telephone_Number_t;
                         Action : RBC_Contact_Action_t;
                         Apply_To_Sleeping_Units : Boolean);

   -- SUBSET-026-3.5.3.5.1 to be formalized. The content of table SUBSET-026-3.5.3.16 should be
   -- incorporated as above operation post-condition (if possible)

   -- SUBSET-026-3.5.3.5.3 and SUBSET-026-3.5.3.6 not formalized (FIXME). Should be similar to
   -- SUBSET-026-3.5.3.5

   -- SUBSET-026-3.5.3.7 see body of Initiate_Communication_Session

   -- SUBSET-026-3.5.3.8 to SUBSET-026-3.5.3.16 not formalized (FIXME)
end;
```

```

-- Reference: UNISIG SUBSET-026-3 v3.3.0

with Appendix_A_3_1;
with Safe_Radio;
with ETCS_Level;

package body Section_3_5_3 is
   procedure Initiate_Communication_Session(destination : RBC_RIU_ID_t;
                                            phone : Telephone_Number_t) is
      connection_attemps : Natural := 0;
   begin
      -- SUBSET-026-3.5.3.7.a
      if Start_Of_Mission then
         while connection_attemps
           <= Appendix_A_3_1.number_of_times_try_establish_safe_radio_connection
         loop
            if Safe_Radio.Setup_Connection(phone) then
               return;
            end if;
            connection_attemps := connection_attemps + 1;
         end loop;
      else
         -- not part of on-going Start of Mission procedure
         loop
            -- FIXME How following asynchronous events are update within this
            -- loop? Should be we read state variable updated by external tasks?
            if Safe_Radio.Setup_Connection(phone)
              or End_Of_Mission
              or Track_Side_Terminate_Communication_Order
              or Train_Passes_Level_Transition_Border
              or (Order_To_Contact_Different_RBC -- FIXME badly formalized
                  and Contact_Order_Not_For_Accepting_RBC)
              or Train_Passes_RBC_RBC_Border
              or Train_Passes_Start_Of_Announced_Radio_Hole
              or (-- FIXME destination is an RIU
                  ETCS_Level.ertms_etcs_level /= 1)
            then
               return;
            end if;
         end loop;
      end if;

      -- SUBSET-026-3.5.3.7.b
```

```
      Safe_Radio.Send_Message(Safe_Radio.Initiation_Of_Communication);

      -- SUBSET-026-3.5.3.7.c not formalized (trackside)
   end;

   procedure Contact_RBC(RBC_identity : RBC_RIU_ID_t;
                         RBC_number : Telephone_Number_t;
                         Action : RBC_Contact_Action_t;
                         Apply_To_Sleeping_Units : Boolean) is
   begin
      null;
   end;

end;
```

5.4 SUBSET-026-3.13 Speed and distance monitoring

In this section we try to model the complex speed and distance monitoring specified in SRS §3.13.

Contrary to previous parts of the model, some functions or procedures have both a specification and a body thus they can be compiled and executed as well as proved. We have done this work for Step_Function and Deceleration_Curve packages.

5.4.1 Generic package

In this package we define some physic related data types (speed, distance, deceleration, ...) and some utility functions (e.g. to convert from m/s to km/h).

One should notice that in next release of GNAT GPL, it will be possible to use a specific mechanism (Dimension_System aspect) to describe those units, thus enabling more checks by the compiler.

```
package Units is
   -- For Breaking Curves computation
   type Speed_t is new Float; -- m/s unit
   type Speed_km_per_h_t is new Float; -- km/h unit
   type Acceleration_t is new Float; -- m/s**2 unit
   type Deceleration_t is new Float range 0.0..Float'Last; -- m/s**2 unit
   type Distance_t is new Natural; -- m unit
   type Time_t is new Float; -- s unit

   Maximum_Valid_Speed_km_per_h : constant Speed_km_per_h_t := 500.0;-- 500 km/h

   function Is_Valid_Speed_km_per_h(Speed: Speed_km_per_h_t) return Boolean is
     (Speed >= 0.0 and Speed <= Maximum_Valid_Speed_km_per_h);

   function m_per_s_From_km_per_h(Speed: Speed_km_per_h_t) return Speed_t
   is
     (Speed_t((Speed * 1000.0) / 3600.0))
   with
     Pre => Is_Valid_Speed_km_per_h(Speed);

   function Is_Valid_Speed(Speed : Speed_t) return Boolean is
     (Speed >= 0.0
      and Speed <= m_per_s_From_km_per_h(Maximum_Valid_Speed_km_per_h));

   function km_per_h_From_m_per_s(Speed: Speed_t) return Speed_km_per_h_t
   with
```

```
    Pre => Is_Valid_Speed(Speed);

end Units;
```

```
package body Units is
   function km_per_h_From_m_per_s(Speed: Speed_t) return Speed_km_per_h_t is
   begin
      return Speed_km_per_h_t((Speed * 3600.0) / 1000.0);
   end;

end Units;
```

5.4.2 Modeling of step functions

In this package we model the step functions used throughout the SRS. They are defined as an array of delimiters, to each delimiter corresponding the value of the current step.

We define following functions:

- Is_Valid: returns True if the delimiters are in increasing order;
- Has_Same_Delimiters: returns True if two step functions change their steps at the same positions;
- Get_Value: returns the value of a step function at position X;
- Minimum_Until_Point: returns the minimum of a step function until a position X;
- Restrictive_Merge: merges two step functions into a third one, in such a way that the result is always the minimum of the two step functions.

All those functions can be compiled and tested (see examples below).

We have tried to prove them, but except for the simplest ones some unproved and sometimes complex VCs are remaining.

One will notice that we are using both Assert and Loop_Invariant annotations. Loop_Invariant states a loop invariant, i.e. a property valid at loop entry and should be true at next loop interation. Assert states a property that should be valid at the point it is inserted.

```
package Step_Function is
   type Num_Delimiters_Range is range 0 .. 10;

   type Function_Range is new Natural;

   type Delimiter_Entry is record
      Delimiter : Function_Range;
      Value : Float;
   end record;

   type Delimiter_Values is array (Num_Delimiters_Range)
     of Delimiter_Entry;

   type Step_Function_t is record
      Number_Of_Delimiters : Num_Delimiters_Range;
      Step : Delimiter_Values;
```

```
end record;

function Min(X1, X2 : Float) return Float
with Post => (if X1 <= X2 then Min'Result = X1 else Min'Result = X2);

function Is_Valid(SFun : Step_Function_t) return Boolean is
  (SFun.Step(0).Delimiter = Function_Range'First
   and
     (for all i in 0..(SFun.Number_Of_Delimiters - 1) =>
        (SFun.Step(i+1).Delimiter > SFun.Step(i).Delimiter)));

function Has_Same_Delimiters(SFun1, SFun2 : Step_Function_t) return Boolean
is
  (SFun1.Number_Of_Delimiters = SFun2.Number_Of_Delimiters
   and (for all i in 1.. SFun1.Number_Of_Delimiters =>
        SFun1.Step(i).Delimiter = SFun2.Step(i).Delimiter));

function Get_Value(SFun : Step_Function_t; X: Function_Range) return Float
with Pre => Is_Valid(SFun),
Post => ((for some i in
             Num_Delimiters_Range'First..(SFun.Number_Of_Delimiters - 1) =>
             (SFun.Step(i).Delimiter <= X
              and X < SFun.Step(i+1).Delimiter
              and Get_Value'Result = SFun.Step(i).Value))
         or
           (X >= SFun.Step(SFun.Number_Of_Delimiters).Delimiter
            and Get_Value'Result
            = SFun.Step(SFun.Number_Of_Delimiters).Value));

function Minimum_Until_Point(SFun : Step_Function_t; X: Function_Range)
                             return Float
with
  Pre => Is_Valid(SFun),
Post =>
-- returned value is the minimum until the point X
  (for all i in Num_Delimiters_Range'First..SFun.Number_Of_Delimiters =>
     (if X >= SFun.Step(i).Delimiter then
      Minimum_Until_Point'Result <= SFun.Step(i).Value))
  and
-- returned value is a value of the step function until point X
  ((for some i in Num_Delimiters_Range'First..SFun.Number_Of_Delimiters =>
      (X >= SFun.Step(i).Delimiter
       and
         (Minimum_Until_Point'Result = SFun.Step(i).Value))));

procedure Index_Increment(SFun: Step_Function_t;
                          i: in out Num_Delimiters_Range;
                          scan: in out Boolean)
with Post =>
  (if i'Old < SFun.Number_Of_Delimiters then
     (i = i'Old + 1 and scan = scan'Old)
   else
     (i = i'Old and scan = False));

-- Note: In the following Post condition, it would be better to tell that
-- Merge is the minimum of both SFun1 and SFun2 for all possible input
-- values, but I'm not sure that can be proved
procedure Restrictive_Merge(SFun1, SFun2 : in Step_Function_t;
                            Merge : out Step_Function_t)
with Pre => Is_Valid(SFun1) and Is_Valid(SFun2)
  and SFun1.Number_Of_Delimiters + SFun2.Number_Of_Delimiters <=
    Num_Delimiters_Range'Last,
Post =>
```

```
   -- Output is valid step function
   Is_Valid(Merge)
   -- all SFun1 delimiters are valid delimiters in Merge
     and (for all i in Num_Delimiters_Range'First..SFun1.Number_Of_Delimiters =>
            (for some j in
                 Num_Delimiters_Range'First..Merge.Number_Of_Delimiters =>
                   (Merge.Step(j).Delimiter = SFun1.Step(i).Delimiter)))
   -- all SFun2 delimiters are valid delimiters in Merge
     and (for all i in Num_Delimiters_Range'First..SFun2.Number_Of_Delimiters =>
            (for some j in
                 Num_Delimiters_Range'First..Merge.Number_Of_Delimiters =>
                   (Merge.Step(j).Delimiter = SFun2.Step(i).Delimiter)))
   -- for all delimiters of Merge, its value is the minimum of SFun1 and SFun2
     and (for all i in Num_Delimiters_Range'First..Merge.Number_Of_Delimiters =>
            (Merge.Step(i).Value = Min(Get_Value(SFun1,
                                                 Merge.Step(i).Delimiter),
                                       Get_Value(SFun2,
                                                 Merge.Step(i).Delimiter))));
end Step_Function;
```

```
package body Step_Function is
   function Min(X1, X2 : Float) return Float is
   begin
      if X1 <= X2 then return X1; else return X2; end if;
   end;

   function Get_Value(SFun : Step_Function_t; X: Function_Range) return Float is
   begin
      for i in Num_Delimiters_Range'First..(SFun.Number_Of_Delimiters - 1) loop
         Pragma Loop_Invariant (for all j in 1..i =>
                                  X >= SFun.Step(j).Delimiter);
         if X >= SFun.Step(i).Delimiter and X < SFun.Step(i + 1).Delimiter then
            return SFun.Step(i).Value;
         end if;
      end loop;

      return SFun.Step(SFun.Number_Of_Delimiters).Value;
   end Get_Value;

   function Minimum_Until_Point(SFun : Step_Function_t; X: Function_Range)
                                return Float is
      min : Float := SFun.Step(Num_Delimiters_Range'First).Value;
   begin
      for i in Num_Delimiters_Range'First .. SFun.Number_Of_Delimiters loop
         Pragma Loop_Invariant
           (for all j in Num_Delimiters_Range'First..i-1 =>
              (if X >= SFun.Step(j).Delimiter then
                 min <= SFun.Step(j).Value));
         Pragma Loop_Invariant
           (for some j in Num_Delimiters_Range'First..i =>
              (X >= SFun.Step(j).Delimiter
                 and
                   min = SFun.Step(j).Value));

         if X >= SFun.Step(i).Delimiter then
            if SFun.Step(i).Value < min then min := SFun.Step(i).Value; end if;
         else
            Pragma Assert
              (for all j in i+1..SFun.Number_Of_Delimiters =>
                 SFun.Step(j-1).Delimiter < SFun.Step(j).Delimiter);
         end if;
      end loop;

```

```
         return min;
      end Minimum_Until_Point;

      procedure Index_Increment(SFun: Step_Function_t;
                                i: in out Num_Delimiters_Range;
                                scan: in out Boolean) is
      begin
         if i < SFun.Number_Of_Delimiters then
            i := i + 1;
         else
            scan := False;
         end if;
      end;

      procedure Restrictive_Merge(SFun1, SFun2 : in Step_Function_t;
                                  Merge : out Step_Function_t) is
--       begin
--          null;
--       end;
         i1 : Num_Delimiters_Range := 0;
         i2 : Num_Delimiters_Range := 0;
         im : Num_Delimiters_Range := 0;
         scan_sfun1 : Boolean := True;
         scan_sfun2 : Boolean := True;
      begin
         Pragma Assert (SFun1.Step(0).Delimiter = SFun2.Step(0).Delimiter);
         loop
            -- im, i1 and i2 bounds
            Pragma Loop_Invariant (i1 >= 0);
            Pragma Loop_Invariant (i2 >= 0);
            Pragma Loop_Invariant (im >= 0);
            Pragma Loop_Invariant (i1 <= SFun1.Number_Of_Delimiters);
            Pragma Loop_Invariant (i2 <= SFun2.Number_Of_Delimiters);
            Pragma Loop_Invariant (i1 + i2 <= Num_Delimiters_Range'Last);
            Pragma Loop_Invariant (im <= Num_Delimiters_Range'Last);
            Pragma Loop_Invariant (im <= i1 + i2);

            -- Merge is a valid step function until im
            Pragma Loop_Invariant (for all i in 1..im-1 =>
                             Merge.Step(i-1).Delimiter < Merge.Step(i).Delimiter);

            -- All merged delimiters are coming from valid delimiter in SFun1 or
            -- SFun2
            Pragma Loop_Invariant
              (for all i in 0..i1-1 =>
                 ((for some j in 0..im-1 =>
                     SFun1.Step(i).Delimiter = Merge.Step(j).Delimiter)));
            Pragma Loop_Invariant
              (for all i in 0..i2-1 =>
                 ((for some j in 0..im-1 =>
                     SFun2.Step(i).Delimiter = Merge.Step(j).Delimiter)));

            -- Merged value at a delimiter is the minimum of both step functions
            Pragma Loop_Invariant
              (for all i in 0..im-1 =>
                 Merge.Step(i).Value =
                 Min(Get_Value(SFun1, Merge.Step(i).Delimiter),
                   Get_Value(SFun2, Merge.Step(i).Delimiter)));

            if scan_sfun1 and scan_sfun2 then
               -- select on delimiter from SFun1 or SFun2
               if SFun1.Step(i1).Delimiter < SFun2.Step(i2).Delimiter then
                  Merge.Step(im).Delimiter := SFun1.Step(i1).Delimiter;
```

```
               Merge.Step(im).Value :=
                  Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
                      Get_Value(SFun2, Merge.Step(im).Delimiter));
               Index_Increment(SFun1, i1, scan_sfun1);

            elsif SFun1.Step(i1).Delimiter > SFun2.Step(i2).Delimiter then
               Merge.Step(im).Delimiter := SFun2.Step(i2).Delimiter;
               Merge.Step(im).Value :=
                  Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
                      Get_Value(SFun2, Merge.Step(im).Delimiter));
               Index_Increment(SFun2, i2, scan_sfun2);

            else -- SFun1.Step(i1).Delimiter = SFun2.Step(i2).Delimiter
               Merge.Step(im).Delimiter := SFun1.Step(i1).Delimiter;
               Merge.Step(im).Value :=
                  Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
                      Get_Value(SFun2, Merge.Step(im).Delimiter));
               Index_Increment(SFun1, i1, scan_sfun1);
               Index_Increment(SFun2, i2, scan_sfun2);
            end if;
         elsif scan_sfun1 then
            -- only use SFun1 delimiter
            Merge.Step(im).Delimiter := SFun1.Step(i1).Delimiter;
            Merge.Step(im).Value :=
               Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
                   Get_Value(SFun2, Merge.Step(im).Delimiter));
            Index_Increment(SFun1, i1, scan_sfun1);
         else -- scan_sfun2
              -- only use SFun2 delimiter
               Merge.Step(im).Delimiter := SFun2.Step(i2).Delimiter;
               Merge.Step(im).Value :=
                  Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
                      Get_Value(SFun2, Merge.Step(im).Delimiter));
            Index_Increment(SFun2, i2, scan_sfun2);
         end if;

         Pragma Assert (if scan_sfun1 or scan_sfun2 then im < i1 + i2);
         if scan_sfun1 or scan_sfun2 then
            im := im + 1;
         else
            exit;
         end if;
      end loop;

      Merge.Number_Of_Delimiters := im;
   end Restrictive_Merge;
end Step_Function;
```

```
with Step_Function; use Step_Function;
with GNAT.IO; use GNAT.IO;

procedure Step_Function_Test is
   SFun1 : Step_Function_t :=
     (Number_Of_Delimiters => 2,
      Step => ((Delimiter => 0, Value => 3.0),
               (Delimiter => 3, Value => 2.0),
               (Delimiter => 5, Value => 5.0),
               others => (Delimiter => 0, Value => 0.0)));

   SFun2 : Step_Function_t :=
     (Number_Of_Delimiters => 2,
      Step => ((Delimiter => 0, Value => 1.0),
               (Delimiter => 3, Value => 1.0),
```

```
                (Delimiter => 5, Value => 3.0),
                others => (Delimiter => 0, Value => 0.0)));

   sfun3 : Step_Function_t :=
     (Number_Of_Delimiters => 5,
      Step => ((Delimiter => 0, Value => 1.0),
                (Delimiter => 1, Value => 1.0),
                (Delimiter => 3, Value => 3.0),
                (Delimiter => 5, Value => 5.0),
                (Delimiter => 7, Value => 7.0),
                (Delimiter => 9, Value => 9.0),
                others => (Delimiter => 0, Value => 0.0)));

   sfun4 : Step_Function_t :=
     (Number_Of_Delimiters => 5,
      Step => ((Delimiter => 0, Value => 10.0),
                (Delimiter => 2, Value => 8.0),
                (Delimiter => 4, Value => 6.0),
                (Delimiter => 6, Value => 4.0),
                (Delimiter => 8, Value => 2.0),
                (Delimiter => 10, Value => 0.5),
                others => (Delimiter => 0, Value => 0.0)));

   sfun_merge : Step_Function_t;
begin
   Pragma Assert (Is_Valid(SFun1));
   Pragma Assert (Is_Valid(SFun2));
   Pragma Assert (Step_Function.Is_Valid(sfun3));
   Pragma Assert (Step_Function.Is_Valid(sfun4));

   Pragma Assert (Get_Value(SFun1, 0) = 3.0);
   Pragma Assert (Get_Value(SFun1, 1) = 3.0);
   Pragma Assert (Get_Value(SFun1, 3) = 2.0);
   Pragma Assert (Get_Value(SFun1, 4) = 2.0);
   Pragma Assert (Get_Value(SFun1, 5) = 5.0);
   Pragma Assert (Get_Value(SFun1,
     Function_Range'Last) = 5.0);

   Pragma Assert (Has_Same_Delimiters(SFun1, SFun2));

   Restrictive_Merge(sfun3, sfun4, sfun_merge);

   for i in Function_Range'First..12 loop
--         Put (Float'Image(Get_Value(sfun_merge, i)));
--         New_line;
      Pragma Assert (Get_Value(sfun_merge, i)
                     = Min(Get_Value(sfun3, i), Get_Value(sfun4, i)));
   end loop;

   Pragma Assert (Minimum_Until_Point(sfun4, 1) = 10.0);
   Pragma Assert (Minimum_Until_Point(sfun4, 5) = 6.0);
   Pragma Assert (Minimum_Until_Point(sfun_merge, 11) = 0.5);
end;
```

5.4.3 Modeling of deceleration curves

In this package, we have modeled deceleration curves and functions computing them.

Function Distance_To_Speed is a simple function that returns the distance to reach a final speed, given an initial speed and a constant (and negative) acceleration. We have tried to prove this function.

Procedure Curve_From_Target computes the braking curve given as input a target (speed and location). This computation is closer to SRS SUBSET-026 requirements but albeit is not proved.

Procedure Print_Curve is a utility function to print the curve on the terminal, in order to plot it.

```
with Units; use Units;

package Deceleration_Curve is
   Distance_Resolution : constant Distance_t := 5; -- m

   Maximum_Valid_Speed : constant Speed_t :=
     m_per_s_From_km_per_h(Maximum_Valid_Speed_km_per_h);

   Minimum_Valid_Acceleration : constant Acceleration_t := -10.0; -- FIXME: realistic value?

   type Braking_Curve_Range is range 0..1_000;

   Braking_Curve_Maximum_End_Point : constant Distance_t :=
     Distance_t(Braking_Curve_Range'Last - Braking_Curve_Range'First)
     * Distance_Resolution;

   type Braking_Curve_Entry is
      record
         location : Distance_t;
         speed : Speed_t;
      end record;

   type Braking_Curve_Array is array (Braking_Curve_Range)
     of Braking_Curve_Entry;

   type Braking_Curve_t is
      record
         curve : Braking_Curve_Array;
         end_point : Distance_t;
      end record;

   -- SUBSET-026-3.13.8.1.1
   type Target_t is
      record
         supervise : Boolean;
         location : Distance_t;
         speed : Speed_t;
      end record;

   function Distance_To_Speed(Initial_Speed, Final_Speed: Speed_t;
                              Acceleration: Acceleration_t)
                              return Distance_t
   with
     Pre => (Initial_Speed > 0.0 and Final_Speed >= 0.0
             and
               Initial_Speed <= Maximum_Valid_Speed
             and
               Initial_Speed > Final_Speed
             and
               Acceleration < 0.0
             and
               Acceleration >= Minimum_Valid_Acceleration);

   function Curve_Index_From_Location(d : Distance_t)
                                      return Braking_Curve_Range
   with
   Pre => (d <= Braking_Curve_Maximum_End_Point);

```

```
   procedure Curve_From_Target(Target : Target_t;
                               Braking_Curve : out Braking_Curve_t)
   with
     Pre => (Target.location <= Braking_Curve_Maximum_End_Point);

   procedure Print_Curve(Braking_Curve : Braking_Curve_t);
end Deceleration_Curve;
```

```
with Units; use Units;
with Ada.Numerics.Generic_Elementary_Functions;
with GNAT.IO; use GNAT.IO;
with sec_3_13_6_deceleration; use sec_3_13_6_deceleration;

package body Deceleration_Curve is
   Minimum_Valid_Speed : constant Speed_t := 0.1; -- m/s

   function Distance_To_Speed(Initial_Speed, Final_Speed: Speed_t;
                              Acceleration: Acceleration_t)
                              return Distance_t is
      speed : Speed_t := Initial_Speed;
      delta_speed : Speed_t;
      distance : Distance_t := 0;
   begin
      while speed > final_speed and speed > Minimum_Valid_Speed loop
         Pragma Assert (Minimum_Valid_Acceleration <= Acceleration
                        and Acceleration < 0.0);
         Pragma Loop_Invariant
           (Minimum_Valid_Speed < speed and speed <= Initial_Speed);
         Pragma Assert (0.0 < 1.0/speed and 1.0/speed < 1.0 / Minimum_Valid_Speed);
         Pragma assert
           ((Speed_t(Minimum_Valid_Acceleration) / Minimum_Valid_Speed)
            <= Speed_t(Acceleration) / speed);
         Pragma assert
           ((Speed_t(Minimum_Valid_Acceleration) / Minimum_Valid_Speed)
            * Speed_t(Distance_Resolution)
            <= (Speed_t(Acceleration) / speed) * Speed_t(Distance_Resolution));

         delta_speed := (Speed_t(Acceleration) / speed)
           * Speed_t(Distance_Resolution);

         Pragma Assert
           ((Speed_t(Minimum_Valid_Acceleration) / Minimum_Valid_Speed)
            * Speed_t(Distance_Resolution) <= delta_speed
            and
              delta_speed < 0.0);

         speed := speed + delta_speed;

         distance := distance + Distance_Resolution;
      end loop;

      return distance;
   end;

   function Curve_Index_From_Location(d : Distance_t)
                                      return Braking_Curve_Range is
   begin
      return Braking_Curve_Range(d / Distance_Resolution);
   end;

   procedure Curve_From_Target(Target : Target_t;
                               Braking_Curve : out Braking_Curve_t) is
      package Speed_Math is
```

```
        new Ada.Numerics.Generic_Elementary_Functions(Speed_t);
      use Speed_Math;

      speed : Speed_t := Target.speed;
      location : Distance_t := Target.location;
      end_point : constant Braking_Curve_Range :=
        Curve_Index_From_Location(Target.location);
   begin
      Braking_Curve.end_point := Target.location;
      Braking_Curve.curve(end_point).location := location;
      Braking_Curve.curve(end_point).speed := speed;

      for i in reverse Braking_Curve_Range'First .. end_point - 1 loop
         speed :=
           (speed + Sqrt(speed * speed
            + (Speed_t(4.0) * Speed_t(A_safe(speed, location)))
            * Speed_t(Distance_Resolution))) / 2.0;
         if speed > Maximum_Valid_Speed then
            speed := Maximum_Valid_Speed;
         end if;

         location := Distance_t(i) * Distance_Resolution;

         Braking_Curve.curve(i).location := location;
         Braking_Curve.curve(i).speed := speed;
      end loop;
   end Curve_From_Target;

   procedure Print_Curve(Braking_Curve : Braking_Curve_t) is
   begin
      for i in Braking_Curve_Range'First ..
        Curve_Index_From_Location(Braking_Curve.end_point) loop
         Put(Distance_t'Image(Braking_Curve.curve(i).location));
         Put(",   ");
         Put(Speed_km_per_h_t'Image(
           km_per_h_From_m_per_s(Braking_Curve.curve(i).speed)));
         New_Line;

         if Braking_Curve.curve(i).location >= Braking_Curve.end_point then
            exit;
         end if;
      end loop;
   end Print_Curve;
end Deceleration_Curve;
```

```
with GNAT.IO; use GNAT.IO;
with Units; use Units;
with Deceleration_Curve; use Deceleration_Curve;

procedure Deceleration_Curve_Test is
   initial_speed : Speed_t := m_per_s_From_km_per_h(160.0); -- 160 km/h

   target : Target_t := (supervise => True,
                         location => 2500,
                         speed => 0.0);
   braking_curve : Braking_Curve_t;
begin
--         Put (Distance_t'Image(Distance_To_Speed(initial_speed, 0.0, -1.0)));
--         New_line;
   pragma Assert (Distance_To_Speed(initial_speed, 0.0, -1.0) = 1000);

   Curve_From_Target(target, braking_curve);
   Print_Curve(braking_curve);
```

```
end;
```

5.4.4 Section 3.13.2 Train and Track-side related inputs

This package is the model for all the input parameters used for distance and speed monitoring algorithms.

Those functions can be compiled. No proof attempt has been made.

```
-- Reference: UNISIG SUBSET-026-3 v3.3.0

with Units; use Units;
with Step_Function; use Step_Function;

package sec_3_13_2_monitoring_inputs is
   -- *** section 3.13.2.2 Train related inputs ***
   -- ** section 3.13.2.2.1 Introduction **

   -- SUBSET-026-3.13.2.2.1.1 not formalized (description)

   -- SUBSET-026-3.13.2.2.1.2 not formalized

   -- SUBSET-026-3.13.2.2.1.3
   type Breaking_Model_t is (Train_Data_Model, Conversion_Model);
   -- Only Train Data model is modelized
   Breaking_Model : constant Breaking_Model_t := Train_Data_Model;


   -- ** section 3.13.2.2.2 Traction model **

   -- SUBSET-026-3.13.2.2.2.1
   T_traction_cut_off : constant Time_t := 10.0; -- s -- FIXME: realistic value?

   -- SUBSET-026-3.13.2.2.2.2 not formalized (Note)


   -- ** section 3.13.2.2.3 Braking Models **

   -- SUBSET-026-3.13.2.2.3.1.1
   -- Use Step_Function.Step_Function_t type

   -- SUBSET-026-3.13.2.2.3.1.2
   -- Note: It would be better to modelize this as Data type invariant
   function Is_Valid_Deceleration_Model(S : Step_Function_t) return Boolean is
     (Step_Function.Is_Valid(S)
      and
        (S.Number_Of_Delimiters <= 6)); -- 6 delimiters for 7 steps

   -- SUBSET-026-3.13.2.2.3.1.3 not formalized (Note)

   -- SUBSET-026-3.13.2.2.3.1.4
   -- by definition of Step_Function.Step_Function_t

   -- SUBSET-026-3.13.2.2.3.1.5 not formalized (FIXME?)

   -- SUBSET-026-3.13.2.2.3.1.6
   A_brake_emergency_model : constant Step_Function_t :=
     (Number_Of_Delimiters => 0,
      Step => ((0, 1.0), -- (from 0 m/s, 1 m/s**2)
               others => (0, 0.0)));

```

```
  A_brake_service_model : Step_Function_t; -- FIXME give value, set constant

  A_brake_normal_service_model : Step_Function_t; -- FIXME give value, set constant

  -- SUBSET-026-3.13.2.2.3.1.7 not formalized (we do not consider regenerative
  -- brake, eddy current brake and magnetic shoe brake)

  -- SUBSET-026-3.13.2.2.3.1.8 not formalized (Note)

  -- SUBSET-026-3.13.2.2.3.1.9
  type Brake_Position_t is (Freight_Train_In_G, Passenger_Train_In_P,
                            Freight_Train_In_P);


  A_SB01 : constant Deceleration_t := 0.1;
  A_SB02 : constant Deceleration_t := 0.2;

  -- SUBSET-026-3.13.2.2.3.1.10 not formalized FIXME
--    function A_Brake_normal_service(V : Speed_t; position : Brake_Position_t)
--                                    return Deceleration_t;

  -- SUBSET-026-3.13.2.2.3.1.11 not formalized (Note)

  -- SUBSET-026-3.13.2.2.3.2.1 not formalized (description)
  -- SUBSET-026-3.13.2.2.3.2.2 not formalized (figure)

  -- SUBSET-026-3.13.2.2.3.2.3
  T_brake_react : constant Time_t := 1.0; -- s
  T_brake_increase : constant Time_t := 2.0; -- s

  -- SUBSET-026-3.13.2.2.3.2.4
  T_brake_build_up : constant Time_t := T_brake_react + 0.5 * T_brake_increase;

  -- SUBSET-026-3.13.2.2.3.2.5
  T_brake_emergency_react : constant Time_t := T_brake_react;
  T_brake_emergency_increase : constant Time_t := T_brake_increase;
  T_brake_emergency : constant Time_t :=
    T_brake_emergency_react + 0.5 * T_brake_emergency_increase;

  T_brake_service_react : constant Time_t := T_brake_react;
  T_brake_service_increase : constant Time_t := T_brake_increase;
  T_brake_service : constant Time_t :=
    T_brake_service_react + 0.5 * T_brake_service_increase;

  -- SUBSET-026-3.13.2.2.3.2.6 not formalized (Note)

  -- SUBSET-026-3.13.2.2.3.2.7 not formalized (Note)

  -- SUBSET-026-3.13.2.2.3.2.8 not formalized (we do not consider regenerative
  -- brake, eddy current brake and magnetic shoe brake)

  -- SUBSET-026-3.13.2.2.3.2.9 not formalized (Note)

  -- SUBSET-026-3.13.2.2.3.2.10 not formalized (Note)

  -- ** section 3.13.2.2.4 Brake Position **

  -- SUBSET-026-3.13.2.2.4.1
  -- see type Brake_Position_t definition above

  -- SUBSET-026-3.13.2.2.4.2 not formalized (Note)
```

```
-- ** section 3.13.2.2.5 Brake Percentage ** not formalized (conversion model
-- not used)

-- ** section 3.13.2.2.6 Special Brakes ** not formalized (special brake not
-- modelized)


-- ** section 3.13.2.2.7 Service brake interface **

-- SUBSET-026-3.13.2.2.7.1
Service_Brake_Command_Implemented : constant Boolean := True;

-- SUBSET-026-3.13.2.2.7.2
Service_Brake_Feedback_Implemented : constant Boolean := True;

-- ** section 3.13.2.2.8 Traction cut-off interface **

-- SUBSET-026-3.13.2.2.8.1
Traction_Cut_Off_Command_Implemented : constant Boolean := True;


-- ** section 3.13.2.2.9 On-borad Correction Factors **

-- SUBSET-026-3.13.2.2.9.1.1 not formalized (description)

-- SUBSET-026-3.13.2.2.9.1.2
Kdry_rst_model : constant Step_Function_t :=
  (Number_Of_Delimiters => 0,
   Step => ((0, 1.0), -- (from 0 m/s, 1.0)
            others => (0, 0.0)));

Kwet_rst_model : constant Step_Function_t :=
  (Number_Of_Delimiters => 0,
   Step => ((0, 1.0), -- (from 0 m/s, 1.0)
            others => (0, 0.0)));

-- SUBSET-026-3.13.2.2.9.1.3
-- FIXME EBCL parameter not formalized
function Is_Valid_Kdry_rst return Boolean is
  (Step_Function.Is_Valid(Kdry_rst_model)
   and
     (Has_Same_Delimiters(Kdry_rst_model, A_brake_emergency_model)));

function Kdry_rst(V: Speed_t) return Float
with
  Pre => Is_Valid_Kdry_rst,
  Post =>
    (Kdry_rst'Result
     = Step_Function.Get_Value(SFun => Kdry_rst_model,
                               X    => Function_Range(V)));

-- SUBSET-026-3.13.2.2.9.1.4 not formalized (FIXME)

-- SUBSET-026-3.13.2.2.9.1.5
function Is_Valid_Kwet_rst return Boolean is
  (Step_Function.Is_Valid(Kwet_rst_model)
   and
     (Has_Same_Delimiters(Kwet_rst_model, A_brake_emergency_model)));

function Kwet_rst(V: Speed_t) return Float
with
  Pre => Is_Valid_Kwet_rst,
  Post =>
```

```
        (Kwet_rst'Result
         = Step_Function.Get_Value(SFun => Kwet_rst_model,
                                   X    => Function_Range(V)));

   -- SUBSET-026-3.13.2.2.9.2.1
   type Gradient_Range is new Float range 0.0 .. 10.0; -- m/s**2

   Kn_Plus : Step_Function_t;
   Kn_Minus : Step_Function_t;

   -- SUBSET-026-3.13.2.2.9.2.2
   -- Note: It would be better to modelize this as Data type invariant
   function Is_Valid_Kn return Boolean is
     (Step_Function.Is_Valid(Kn_Plus) and Step_Function.Is_Valid(Kn_Minus)
      and
        (Kn_Plus.Number_Of_Delimiters <= 4) -- 4 delimiters for 5 steps
      and
        (Kn_Minus.Number_Of_Delimiters <= 4)); -- 4 delimiters for 5 steps

   -- SUBSET-026-3.13.2.2.9.2.3 not formalized (Note)

   -- SUBSET-026-3.13.2.2.9.2.4 not formalized (FIXME)

   -- SUBSET-026-3.13.2.2.9.2.5 not formalized (FIXME)

   -- SUBSET-026-3.13.2.2.9.2.6
   -- By definition of Step_Function_t

   -- ** section 3.13.2.2.10 Nominal Rotating mass **

   -- SUBSET-026-3.13.2.2.10.1 not formalized (FIXME)

   -- ** section 3.13.2.2.11 Train length **

   -- SUBSET-026-3.13.2.2.11.1
   Train_Length : constant Distance_t := 900; -- m

   -- ** section 3.13.2.2.12 Fixed Values **

   -- SUBSET-026-3.13.2.2.12.1 not formalized (description)

   -- ** section 3.13.2.2.13 Maximum train speed **

   -- SUBSET-026-3.13.2.2.12.1
   Maximum_Train_Speed : constant Speed_t := m_per_s_From_km_per_h(250.0);

   -- *** section 3.13.2.3 Trackside related inputs ***
   -- all sections of 3.13.2.3 not formalized
   procedure dummy;
end sec_3_13_2_monitoring_inputs;
```

```
-- Reference: UNISIG SUBSET-026-3 v3.3.0

package body sec_3_13_2_monitoring_inputs is
--      function A_Brake_normal_service(V : Speed_t; position : Brake_Position_t)
--                                      return Deceleration_t is
--      begin
--         return 0.0;
--      end;
   function Kdry_rst(V: Speed_t) return Float is
   begin
      return Step_Function.Get_Value(SFun => Kdry_rst_model,
                                     X    => Function_Range(V));
```

```
   end;

   function Kwet_rst(V: Speed_t) return Float is
   begin
      return Step_Function.Get_Value(SFun => Kwet_rst_model,
                                     X    => Function_Range(V));
   end;

   procedure dummy is
   begin
      null;
   end;

end sec_3_13_2_monitoring_inputs;
```

5.4.5 Sections 3.13.4 to 3.13.8 Braking curves computation

The following packages contains the modeling of the braking curves computation, as close as possible to SRS §3.13.4 to §3.13.8.

```
with Units; use Units;

package sec_3_13_4_gradient_accel_decel is
   -- FIXME 3.13.4 not formalized

   function A_gradient(d: Distance_t) return Deceleration_t is
     (0.0);

end sec_3_13_4_gradient_accel_decel;
```

```
with Units; use Units;

with Step_Function; use Step_Function;
with Appendix_A_3_1; use Appendix_A_3_1;
with sec_3_13_2_monitoring_inputs; use sec_3_13_2_monitoring_inputs;
with sec_3_13_4_gradient_accel_decel; use sec_3_13_4_gradient_accel_decel;

package sec_3_13_6_deceleration is
   -- SUBSET-026-3.13.6.2.1 to 3.13.6.2.1.2 not formalized (FIXME)

   -- SUBSET-026-3.13.6.2.1.5 (Note .5 before .4 for proper definition)
   -- Note: we are not using specific break configuration so parameter 'd' is
   -- never used
   function A_brake_emergency(V: Speed_t; d: Distance_t) return Deceleration_t
   with
     Pre => (Is_Valid_Deceleration_Model(A_brake_emergency_model)
             and Is_Valid_Speed(V)),
   Post =>
     (A_brake_emergency'Result
      = Deceleration_t(Step_Function.Get_Value(SFun => A_brake_emergency_model,
                                               X    => Function_Range(V))));

   -- SUBSET-026-3.13.6.2.1.4 (Note .4 before .3 for proper definition)
   function A_brake_safe(V: Speed_t; d: Distance_t) return Deceleration_t is
     (Deceleration_t((Kdry_rst(V)
                      * (KWet_rst(V) + M_NVAVADH * (1.0 - Kwet_rst(V))))
                     * Float(A_brake_emergency(V,d))));

   -- SUBSET-026-3.13.6.2.1.3
   -- FIXME reduced adhesion condition not formalized
   function A_safe(V: Speed_t; d: Distance_t) return Deceleration_t is
     (A_brake_safe(V, d) + A_gradient(d));
```

```
    -- SUBSET-026-3.13.6.2.1.6 not formalized (conversion model not used)

    -- SUBSET-026-3.13.6.2.1.7 not formalized (same requirements as
    -- SUBSET-026-3.13.2.2.9.1.3)

    -- SUBSET-026-3.13.6.2.1.8 not formalized (conversion model not used)
    -- SUBSET-026-3.13.6.2.1.8.1 not formalized (conversion model not used)
    -- SUBSET-026-3.13.6.2.1.8.2 not formalized (conversion model not used)

    -- SUBSET-026-3.13.6.2.1.9 not formalized (Note)

    -- SUBSET-026-3.13.6.2.2.1 not formalized (description)

    -- SUBSET-026-3.13.6.2.2.2.a not formalized (FIXME?)
    -- SUBSET-026-3.13.6.2.2.2.b not formalized (conversion model not used)
    -- SUBSET-026-3.13.6.2.2.2.c not formalized (various brakes not modelled)

    -- SUBSET-026-3.13.6.2.2.3
    T_be : constant Time_t := T_brake_emergency;
end sec_3_13_6_deceleration;
```

```
package body sec_3_13_6_deceleration is
    function A_brake_emergency(V: Speed_t; d: Distance_t) return Deceleration_t
    is
    begin
       return
          Deceleration_t(Step_Function.Get_Value(SFun => A_brake_emergency_model,
                                                  X    => Function_Range(V)));
    end;
end sec_3_13_6_deceleration;
```

```
with Units; use Units;
with sec_3_13_6_deceleration; use sec_3_13_6_deceleration;
with Deceleration_Curve; use Deceleration_Curve;

package sec_3_13_8_targets_decel_curves is
    -- ** 3.13.8.1 Introduction **

    -- SUBSET-026-3.13.8.1.1
    -- Defined in Deceleration_Curve package

    -- SUBSET-026-3.13.8.1.2
    -- Use of package sec_3_13_6_deceleration

    -- SUBSET-026-3.13.8.1.3 not formalized (FIXME?)

    -- ** 3.13.8.2 Determination of the supervised targets **

    -- SUBSET-026-3.13.8.2.1
    type Target_Type is (MRSP_Speed_Decrease, -- SUBSET-026-3.13.8.2.1.a
                         Limit_Of_Authority, -- SUBSET-026-3.13.8.2.1.b
                         -- SUBSET-026-3.13.8.2.1.c
                         End_Of_Authority, Supervised_Location,
                         Staff_Responsible_Maximum); -- SUBSET-026-3.13.8.2.1.d

    Target : array (Target_Type) of Target_t;

    -- Note: should be defined as data type invariant
    function Is_Valid_Target return boolean is
      ((if Target(End_Of_Authority).speed > 0.0 then
         Target(Limit_Of_Authority).supervise)
        and
```

```
        (if Target(End_Of_Authority).speed = 0.0 then
         Target(End_Of_Authority).supervise
         and Target(Supervised_Location).supervise)
      and
        Target(Staff_Responsible_Maximum).speed = 0.0);

   -- SUBSET-026-3.13.8.2.1.1 not formalized (Note)

   -- SUBSET-026-3.13.8.2.2 not formalized (FIXME)

   -- SUBSET-026-3.13.8.2.3 not formalized (FIXME)

   -- ** 3.13.8.3 Emergency Brake Deceleration Curve **
   -- FIXME how to merge EBDs if several targets are active at the same time?

   -- SUBSET-026-3.13.8.3.1 not formalized (FIXME)

   -- SUBSET-026-3.13.8.3.2
   procedure Compute_SvL_Curve(Braking_Curve : out Braking_Curve_t)
   with
     Pre => (Is_Valid_Target and Target(Supervised_Location).speed = 0.0);
--         Post => (Curve_From_Target(Target(Supervised_Location), Braking_Curve));
--                  True);

   -- SUBSET-026-3.13.8.3.3 not formalized (FIXME)
end sec_3_13_8_targets_decel_curves;
```

```
package body sec_3_13_8_targets_decel_curves is
   procedure Compute_SvL_Curve(Braking_Curve : out Braking_Curve_t) is
   begin
      Curve_From_Target(Target(Supervised_Location), Braking_Curve);
   end;
end sec_3_13_8_targets_decel_curves;
```

References

[1] Ada 2012 language reference manual. ISO/IEC 8652:2012(E) standard. `http://www.ada-auth.org/standards/ada12.html`.

[2] D. Mentré, S. Pinte, G. Pottier, and WP2 participants. D2.5 methods and tools benchmarking methodology. Technical report, openETCS, 2013.