OETCS/WP7/O??

openETCS
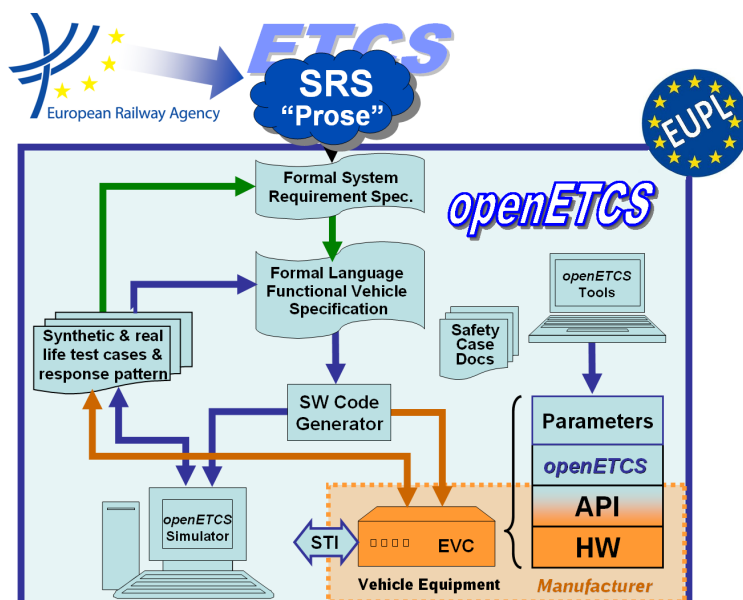
openETCS@ITEA Work Package 7: "Toolchain"

Evaluation model of ETCS using GNATprove

Tool and model presentation

David Mentré

8th of April 2013

This page is intentionally left blank

openETCS@ITEA Work Package 7: "Toolchain"   **OETCS/WP7/O??**
                                                       8th of April 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 1

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.

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

The implementation of this specification is rather obvious.

```
1  package body Example is
2     function Saturated_Sum (X1, X2, Maximum : Natural) return Natural is
3     begin
4       if X1 + X2 <= Maximum then
5          return X1 + X2;
6       else
7          return Maximum;
8       end if;
9     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.

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

```
1  Package ETCS_Level is
2     type ertms_etcs_level_t is range 0..4; -- SUBSET-026-2.6.2.3
3     ertms_etcs_level_ntc : constant ertms_etcs_level_t := 4;
4
5     ertms_etcs_level : ertms_etcs_level_t;
6  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
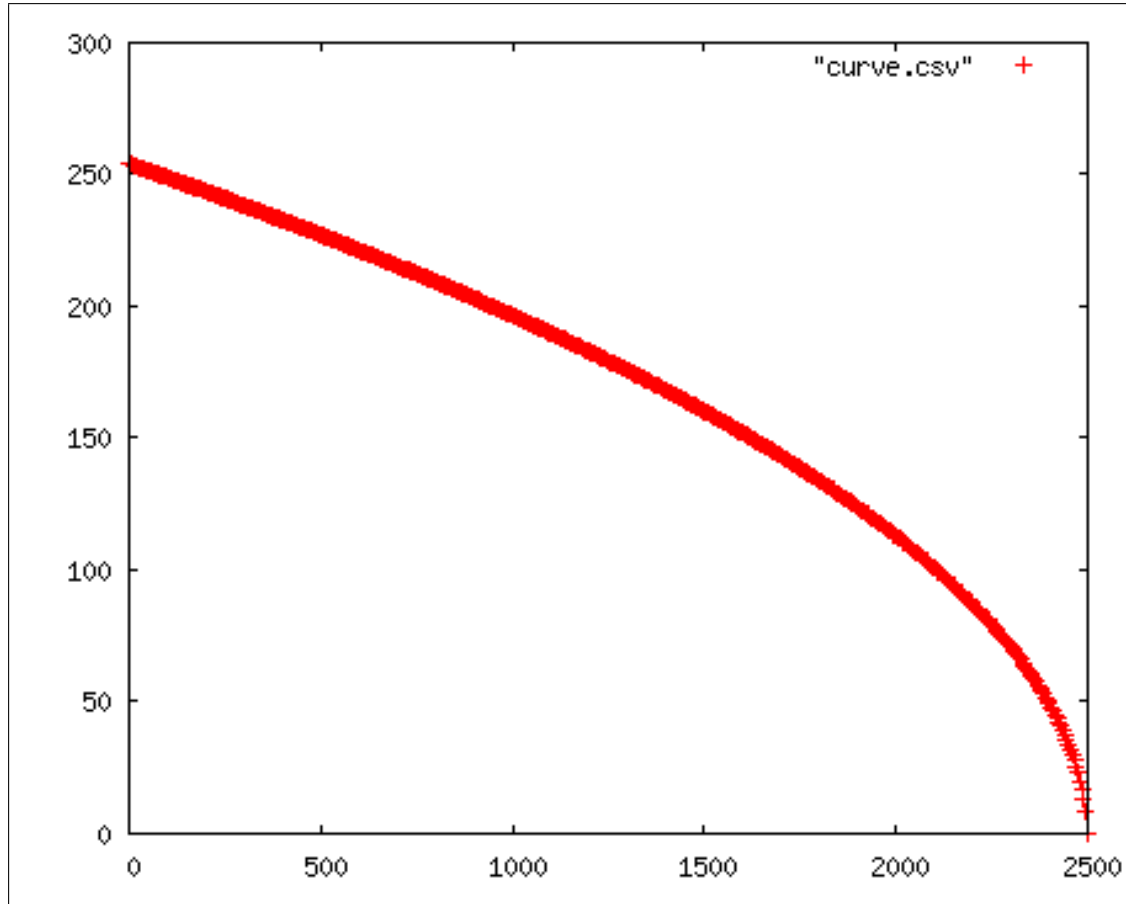
Figure 1. Basic braking curve for EBD, target is at 2,500 m at speed 0 km/h, deceleration is constant $-1m/s^2$

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 affective 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.

Some formalization are generating a lot of VC, e.g. about 9,000 for Step_Function.Restrictive_Merge function. This is going to be fixed in the next release of GNATprove tool in 2013[4].

In case a VC is not proved, the GNATprove error message is sometimes difficult to understand or even missing. This point is going to be significantly improved in next release of GNATprove[5].

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).

---

[4]`http://lists.forge.open-do.org/pipermail/hi-lite-discuss/2013-April/001270.html`
[5]Same reference as footnote 4

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.

```
1  package Data_Types is
2      type Telephone_Number_t is range 0..20_000; -- FIXME refine range
3
4      type RBC_Contact_Action_t is ( Establish_Session , Terminate_Session );
5
6      type RBC_RIU_ID_t is range 1..10_000; -- FIXME: refine range
7  end;
```

```
1
2  -- Reference : UNISIG SUBSET-026-3 v3.3.0
3
4  Package ETCS_Level is
5      type ertms_etcs_level_t is range 0..4; -- SUBSET-026-2.6.2.3
6      ertms_etcs_level_ntc : constant ertms_etcs_level_t := 4;
7
8      ertms_etcs_level : ertms_etcs_level_t;
9  end;
```

```
1
2  -- Reference : UNISIG SUBSET-026-3 v3.3.0
3
4  Package Appendix_A_3_1 is
5      number_of_times_try_establish_safe_radio_connection : constant Integer := 3;
6
7      driver_acknowledgment_time : constant Integer := 5; -- seconds
8      t_ack : constant Integer := driver_acknowledgment_time ;
9
10     M_NVAVADH : constant Float := 0.0;
11 end;
```

5.1.1   SUBSET-026-4.3.2 ETCS modes

```
1
```

```
2   -- Reference: UNISIG SUBSET-026-3 v3.3.0
3
4   Package Section_4_3_2 is
5      type etcs_mode_t is ( Full_Supervision ,
6                            Limited_SUpervision ,
7                            On_Sight ,
8                            Staff_Responsible ,
9                            Shunting ,
10                           Unfitted ,
11                           Passive_Shunting ,
12                           Sleeping ,
13                           Stand_By ,
14                           Trip ,
15                           Post_Trip ,
16                           System_Failure ,
17                           Isolation ,
18                           No_Power,
19                           Non_Leading ,
20                           National_System ,
21                           Reversing );
22
23      type etcs_short_mode_t is (FS,
24                                 LS,
25                                 OS,
26                                 SR,
27                                 SH,
28                                 UN,
29                                 PS,
30                                 SL,
31                                 SB,
32                                 TR,
33                                 PT,
34                                 SF,
35                                 ISo, -- "IS" is a reserved Ada keyword
36                                 NP,
37                                 NL,
38                                 SN,
39                                 RV);
40  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.

```
1
```

```
2   −− Reference : UNISIG SUBSET−026−3 v3.3.0

4   with ETCS_Level ;
5   use ETCS_Level ;

7   with Section_4_3_2 ;
8   use Section_4_3_2 ;

10  Package Section_4_6 is
11      −− SUBSET−026−4.6.3 Transitions Conditions Table
12      −− WARNING: not all conditions are modeled

14      −− Individual condition elements
15      an_acknowledge_request_for_shunting_is_displayed_to_the_driver : Boolean ;

17      driver_acknowledges : Boolean ;

19      driver_isolates_ERTMS_ETCS_on_board_equipment : Boolean ;

21      driver_selects_shunting_mode : Boolean ;

23      ma_ssp_gardient_on_board : Boolean ;

25      no_specific_mode_is_required_by_a_mode_profile : Boolean ;

27      note_5_conditions_for_shunting_mode : Boolean ;

29      reception_of_information_shunting_granted_by_rbc : Boolean ;

31      train_is_at_standstill : Boolean ;

33      valid_train_data_is_stored_on_board : Boolean ;

35      −− Conditions
36      function condition_1 return Boolean is
37        ( driver_isolates_ERTMS_ETCS_on_board_equipment ) ;

39      function condition_5 return Boolean is
40        ( train_is_at_standstill
41         AND ( ertms_etcs_level = 0 OR ertms_etcs_level = ertms_etcs_level_ntc
42              OR ertms_etcs_level = 1)
43         AND driver_selects_shunting_mode ) ;

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

50      function condition_10 return Boolean is
51        ( valid_train_data_is_stored_on_board
52         AND ma_ssp_gardient_on_board
53         AND no_specific_mode_is_required_by_a_mode_profile ) ;

55      function condition_50 return Boolean is
56        ( an_acknowledge_request_for_shunting_is_displayed_to_the_driver
57         AND driver_acknowledges
58         AND note_5_conditions_for_shunting_mode ) ;

60      −− SUBSET−026−4.6.2 Transitions Table
61      type priority_t is range 1..7;
62      priority : priority_t ;

64      function condition_transition_SB_to_SH return Boolean is
```

```
65        (( condition_5 OR condition_6 OR condition_50 ) AND priority = 7);
66
67    function condition_transition_SB_to_FS return Boolean is
68       ( condition_10 AND priority = 7);
69
70    function condition_transition_SB_to_IS return Boolean is
71       ( condition_1 AND priority = 1);
72
73    -- following function is no longer needed
74    function disjoint_condition_transitions return Boolean is
75       (NOT( condition_transition_SB_to_SH = True
76            AND condition_transition_SB_to_FS = True )
77        AND NOT( condition_transition_SB_to_SH = True
78               AND condition_transition_SB_to_IS = True )
79        AND NOT( condition_transition_SB_to_FS = True
80               AND condition_transition_SB_to_IS = True ));
81
82    function transition (mode : etcs_mode_t) return etcs_mode_t
83    with
84 --       Post => ( disjoint_condition_transitions = True );
85       -- SUBSET-026-4.6.1.5: all cases are disjoint
86       Contract_Cases => ( condition_transition_SB_to_SH => True ,
87                           condition_transition_SB_to_FS => True ,
88                           condition_transition_SB_to_IS => True ),
89       Post => True ; -- work around for bug in SPARK Hi-Lite GPL 2013
90 end;
```

```
1
2  -- Reference : UNISIG SUBSET-026-3 v3.3.0
3
4  with Section_4_3_2 ;
5  use Section_4_3_2 ;
6
7  Package body Section_4_6 is
8     function transition (mode : etcs_mode_t) return etcs_mode_t is
9     begin
10       return No_Power ;
11    end;
12 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.

```
1  with Data_Types ;
2
3  package Safe_Radio is
4     type Message_Type_t is ( Initiation_Of_Communication );
5
6     function Setup_Connection (phone : Data_Types . Telephone_Number_t )
7                                  return Boolean ;
8     -- return True if connection is setup , False otherwise
9
10    procedure Send_Message (message : Message_Type_t );
11 end;
```

```
1  package body Safe_Radio is
2
3     ----------------------
4     -- Setup_Connection --
5     ----------------------
6
7     function Setup_Connection
8       (phone : Data_Types.Telephone_Number_t)
9        return Boolean
10    is
11    begin
12       -- Generated stub: replace with real body!
13       raise Program_Error with "Unimplemented function Setup_Connection";
14       return False;
15    end Setup_Connection;
16
17    procedure Send_Message(message : Message_Type_t) is
18    begin
19       -- Generated stub: replace with real body!
20       raise Program_Error with "Unimplemented function Setup_Connection";
21    end Send_Message;
22 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.

```
1  -- Commented out because not supported in GNAT GPL 2012
2  --- with Ada.Containers.Formal_Hashed_Maps;
3  --- with Ada.Containers; use Ada.Containers;
4
5  with Data_Types; use Data_Types;
6
7  package Com_Map is
8  -- Commented out because not supported in GNAT GPL 2012
9  --     function RBC_RIU_ID_Hash(id : RBC_RIU_ID_t) return Hash_Type is
10 --        (Hash_Type(id));
11 --
12 --     package Com_To_RBC_Map is new Ada.Containers.Formal_Hashed_Maps
13 --        (Key_Type        => RBC_RIU_ID_t,
14 --         Element_Type    => Boolean, -- False: com being established
15 --                                     -- True : com established
16 --         Hash            => RBC_RIU_ID_Hash,
17 --         Equivalent_Keys => "=",
18 --         "="             => "=");
19    type Com_Element is record
20       Used             : Boolean := False; -- False: element not used
21       Com_Established : Boolean := False; -- False: com being established
22                                            -- True : com established
23    end record;
24
25    type Com_To_RBC_Map is array (RBC_RIU_ID_T) of Com_Element;
26
27    function Contains (Map : Com_To_RBC_Map; Id : RBC_RIU_ID_T) return Boolean
28    is (Map (Id).Used and Map(Id).Com_Established);
29 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.

```
1
2   -- Reference : UNISIG SUBSET-026-3 v3.3.0
3
4   with ETCS_Level; use ETCS_Level;
5
6   with Data_Types; use Data_Types;
7
8   with Com_Map; use Com_Map;
9
10  Package Section_3_5_3 is
11     -- FIXME using SRS sections as package name is probably not the best approach
12
13     -- SUBSET-026-3.5.3.4
14     Start_Of_Mission : Boolean;
15     End_of_Mission : Boolean;
16     Track_Side_New_Communication_Order : Boolean;
17     Track_Side_Terminate_Communication_Order : Boolean;
18     Train_Passes_Level_Transition_Border : Boolean;
19     Train_Passes_RBC_RBC_Border : Boolean;
20     Train_Passes_Start_Of_Announced_Radio_Hole : Boolean;
21     Order_To_Contact_Different_RBC : Boolean;
22     Contact_Order_Not_For_Accepting_RBC : Boolean;
23     Mode_Change_Report_To_RBC_Not_Considered_As_End_Of_Mission : Boolean; -- to be refined
24     Manual_Level_Change : Boolean;
25     Train_Front_Reaches_End_Of_Radio_Hole : Boolean;
26     Previous_Communication_Loss : Boolean;
27     Start_Of_Mission_Procedure_Completed_Without_Com : Boolean;
28
29  --      Connections : Com_To_RBC_Map.Map( Capacity => 10,
30  --                                        Modulus =>
31  --                                           Com_To_RBC_Map. Default_Modulus (10));
32
33     Connections : Com_To_RBC_Map;
34
35     function Authorize_New_Communication_Session return Boolean is
36       ((Start_Of_Mission = True
37         and (ertms_etcs_level = 2 or ertms_etcs_level = 3)) -- SUBSET-026-3.5.3.4.a
38        and Track_Side_New_Communication_Order = True -- SUBSET-026-3.5.3.4.b
39        and (Mode_Change_Report_To_RBC_Not_Considered_As_End_Of_Mission = True
40             and (ertms_etcs_level = 2 or ertms_etcs_level = 3))-- SUBSET-026-3.5.3.4.c
41        and (Manual_Level_Change = True
42             and (ertms_etcs_level = 2 or ertms_etcs_level = 3)) -- SUBSET-026-3.5.3.4.d
43        and Train_Front_Reaches_End_Of_Radio_Hole = True -- SUBSET-026-3.5.3.4.e
44        and Previous_Communication_Loss = True -- SUBSET-026-3.5.3.4.f
45        and (Start_Of_Mission_Procedure_Completed_Without_Com = True
46             and (ertms_etcs_level = 2 or ertms_etcs_level = 3)) -- SUBSET-026-3.5.3.4.g
47       );
48
49     -- SUBSET-026-3.5.3.1 and SUBSET-026-3.5.3.2 implicitly fulfilled as we model on-board
50     procedure Initiate_Communication_Session(destination : RBC_RIU_ID_t;
51                                              phone : Telephone_Number_t)
52     with
53       Pre => ((Authorize_New_Communication_Session = True) -- SUBSET-026-3.5.3.4
54                and (not Contains(Connections, destination)) -- SUBSET-026-3.5.3.4.1
55                -- FIXME: what should we do for cases f and g?
56               ),
57       Post => (Contains(Connections, destination));
58
```

```
59      -- SUBSET-026-3.5.3.3 not formalized (Note)
60
61      -- SUBSET-026-3.5.3.5
62      procedure Contact_RBC(RBC_identity : RBC_RIU_ID_t;
63                            RBC_number : Telephone_Number_t;
64                            Action : RBC_Contact_Action_t;
65                            Apply_To_Sleeping_Units : Boolean);
66
67      -- SUBSET-026-3.5.3.5.1 to be formalized. The content of table SUBSET-026-3.5.3.16 should be
68      -- incorporated as above operation post-condition (if possible)
69
70      -- SUBSET-026-3.5.3.5.3 and SUBSET-026-3.5.3.6 not formalized (FIXME). Should be similar to
71      -- SUBSET-026-3.5.3.5
72
73      -- SUBSET-026-3.5.3.7 see body of Initiate_Communication_Session
74
75      -- SUBSET-026-3.5.3.8 to SUBSET-026-3.5.3.16 not formalized (FIXME)
76  end;
```

```
1
2   -- Reference: UNISIG SUBSET-026-3 v3.3.0
3
4   with Appendix_A_3_1;
5   with Safe_Radio;
6   with ETCS_Level;
7
8   package body Section_3_5_3 is
9      procedure Initiate_Communication_Session(destination : RBC_RIU_ID_t;
10                                              phone : Telephone_Number_t) is
11         connection_attemps : Natural := 0;
12      begin
13         -- SUBSET-026-3.5.3.7.a
14         if Start_Of_Mission then
15            while connection_attemps
16              <= Appendix_A_3_1.number_of_times_try_establish_safe_radio_connection
17            loop
18               if Safe_Radio.Setup_Connection(phone) then
19                  return;
20               end if;
21               connection_attemps := connection_attemps + 1;
22            end loop;
23         else
24            -- not part of on-going Start of Mission procedure
25            loop
26               -- FIXME How following asynchronous events are update within this
27               -- loop? Should be we read state variable updated by external tasks?
28               if Safe_Radio.Setup_Connection(phone)
29                 or End_Of_Mission
30                 or Track_Side_Terminate_Communication_Order
31                 or Train_Passes_Level_Transition_Border
32                 or (Order_To_Contact_Different_RBC -- FIXME badly formalized
33                    and Contact_Order_Not_For_Accepting_RBC)
34                 or Train_Passes_RBC_RBC_Border
35                 or Train_Passes_Start_Of_Announced_Radio_Hole
36                 or (-- FIXME destination is an RIU
37                    ETCS_Level.ertms_etcs_level /= 1)
38               then
39                  return;
40               end if;
41            end loop;
42         end if;
43
44         -- SUBSET-026-3.5.3.7.b
```

```
45        Safe_Radio.Send_Message(Safe_Radio.Initiation_Of_Communication);
46
47        -- SUBSET-026-3.5.3.7.c not formalized (trackside)
48     end;
49
50     procedure Contact_RBC(RBC_identity : RBC_RIU_ID_t;
51                           RBC_number : Telephone_Number_t;
52                           Action : RBC_Contact_Action_t;
53                           Apply_To_Sleeping_Units : Boolean) is
54     begin
55        null;
56     end;
57
58  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.

```
1   package Units is
2      -- For Breaking Curves computation
3      type Speed_t is new Float; -- m/s unit
4      type Speed_km_per_h_t is new Float; -- km/h unit
5      type Acceleration_t is new Float; -- m/s**2 unit
6      type Deceleration_t is new Float range 0.0..Float'Last; -- m/s**2 unit
7      type Distance_t is new Natural; -- m unit
8      type Time_t is new Float; -- s unit
9
10     Maximum_Valid_Speed_km_per_h : constant Speed_km_per_h_t := 500.0;-- 500 km/h
11
12     function Is_Valid_Speed_km_per_h(Speed: Speed_km_per_h_t) return Boolean is
13        (Speed >= 0.0 and Speed <= Maximum_Valid_Speed_km_per_h);
14
15     function m_per_s_From_km_per_h(Speed: Speed_km_per_h_t) return Speed_t
16     is
17        (Speed_t((Speed * 1000.0) / 3600.0))
18     with
19        Pre => Is_Valid_Speed_km_per_h(Speed);
20
21     function Is_Valid_Speed(Speed : Speed_t) return Boolean is
22        (Speed >= 0.0
23         and Speed <= m_per_s_From_km_per_h(Maximum_Valid_Speed_km_per_h));
24
25     function km_per_h_From_m_per_s(Speed: Speed_t) return Speed_km_per_h_t
26     with
```

```
27        Pre => Is_Valid_Speed(Speed);
28
29  end Units;
```

```
1   package body Units is
2      function km_per_h_From_m_per_s(Speed: Speed_t) return Speed_km_per_h_t is
3      begin
4         return Speed_km_per_h_t((Speed * 3600.0) / 1000.0);
5      end;
6
7   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.

```
1   package Step_Function is
2      type Num_Delimiters_Range is range 0 .. 10;
3
4      type Function_Range is new Natural;
5
6      type Delimiter_Entry is record
7         Delimiter : Function_Range;
8         Value : Float;
9      end record;
10
11     type Delimiter_Values is array (Num_Delimiters_Range)
12       of Delimiter_Entry;
13
14     type Step_Function_t is record
15        Number_Of_Delimiters : Num_Delimiters_Range;
16        Step : Delimiter_Values;
17     end record;
18
19     function Min(X1, X2 : Float) return Float
20     with Post => (if X1 <= X2 then Min'Result = X1 else Min'Result = X2);
21
```

```ada
22    function Is_Valid (SFun : Step_Function_t) return Boolean is
23      (SFun. Step (0). Delimiter = Function_Range ' First
24       and
25         (for all i in 0..(SFun. Number_Of_Delimiters − 1) =>
26           (SFun. Step ( i +1). Delimiter > SFun. Step ( i ). Delimiter )));
27
28    function Has_Same_Delimiters (SFun1, SFun2 : Step_Function_t) return Boolean
29    is
30      (SFun1. Number_Of_Delimiters = SFun2. Number_Of_Delimiters
31       and (for all i in 1.. SFun1. Number_Of_Delimiters =>
32           SFun1. Step ( i ). Delimiter = SFun2. Step ( i ). Delimiter ));
33
34    function Get_Value (SFun : Step_Function_t; X: Function_Range) return Float
35    with Pre => Is_Valid (SFun) ,
36    Post => ((for some i in
37             Num_Delimiters_Range ' First ..(SFun. Number_Of_Delimiters − 1) =>
38             (SFun. Step ( i ). Delimiter <= X
39              and X < SFun. Step ( i +1). Delimiter
40              and Get_Value ' Result = SFun. Step ( i ). Value ))
41          or
42           (X >= SFun. Step (SFun. Number_Of_Delimiters ). Delimiter
43            and Get_Value ' Result
44            = SFun. Step (SFun. Number_Of_Delimiters ). Value ));
45
46    function Minimum_Until_Point (SFun : Step_Function_t; X: Function_Range)
47                                  return Float
48    with
49      Pre => Is_Valid (SFun) ,
50    Post =>
51    −− returned value is the minimum until the point X
52      (for all i in Num_Delimiters_Range ' First .. SFun. Number_Of_Delimiters =>
53         ( if X >= SFun. Step ( i ). Delimiter then
54           Minimum_Until_Point ' Result <= SFun. Step ( i ). Value ))
55      and
56    −− returned value is a value of the step function until point X
57      ((for some i in Num_Delimiters_Range ' First .. SFun. Number_Of_Delimiters =>
58         (X >= SFun. Step ( i ). Delimiter
59          and
60           (Minimum_Until_Point ' Result = SFun. Step ( i ). Value ))));
61
62    procedure Index_Increment (SFun: Step_Function_t;
63                               i: in out Num_Delimiters_Range;
64                               scan: in out Boolean)
65    with Post =>
66      ( if i ' Old < SFun. Number_Of_Delimiters then
67         ( i = i ' Old + 1 and scan = scan ' Old)
68       else
69         ( i = i ' Old and scan = False ));
70
71    −− Note: In the following Post condition , it would be better to tell that
72    −− Merge is the minimum of both SFun1 and SFun2 for all possible input
73    −− values , but I 'm not sure that can be proved
74    procedure Restrictive_Merge (SFun1, SFun2 : in Step_Function_t;
75                                 Merge : out Step_Function_t)
76    with Pre => Is_Valid (SFun1) and Is_Valid (SFun2)
77      and SFun1. Number_Of_Delimiters + SFun2. Number_Of_Delimiters <=
78       Num_Delimiters_Range ' Last ,
79    Post =>
80    −− Output is valid step function
81    Is_Valid (Merge)
82    −− all SFun1 delimiters are valid delimiters in Merge
83      and (for all i in Num_Delimiters_Range ' First .. SFun1. Number_Of_Delimiters =>
84           (for some j in
```

```
85                    Num_Delimiters_Range'First..Merge.Number_Of_Delimiters =>
86                        (Merge.Step(j).Delimiter = SFun1.Step(i).Delimiter)))
87        -- all SFun2 delimiters are valid delimiters in Merge
88          and (for all i in Num_Delimiters_Range'First..SFun2.Number_Of_Delimiters =>
89                  (for some j in
90                      Num_Delimiters_Range'First..Merge.Number_Of_Delimiters =>
91                        (Merge.Step(j).Delimiter = SFun2.Step(i).Delimiter)))
92        -- for all delimiters of Merge, its value is the minimum of SFun1 and SFun2
93          and (for all i in Num_Delimiters_Range'First..Merge.Number_Of_Delimiters =>
94                  (Merge.Step(i).Value = Min(Get_Value(SFun1,
95                                                        Merge.Step(i).Delimiter),
96                                          Get_Value(SFun2,
97                                                        Merge.Step(i).Delimiter))));
98  end Step_Function;
```

```
1   package body Step_Function is
2       function Min(X1, X2 : Float) return Float is
3       begin
4           if X1 <= X2 then return X1; else return X2; end if;
5       end;
6
7       function Get_Value(SFun : Step_Function_t; X: Function_Range) return Float is
8       begin
9           for i in Num_Delimiters_Range'First..(SFun.Number_Of_Delimiters - 1) loop
10              Pragma Loop_Invariant (for all j in 1..i =>
11                                          X >= SFun.Step(j).Delimiter);
12              if X >= SFun.Step(i).Delimiter and X < SFun.Step(i + 1).Delimiter then
13                  return SFun.Step(i).Value;
14              end if;
15          end loop;
16
17          return SFun.Step(SFun.Number_Of_Delimiters).Value;
18      end Get_Value;
19
20      function Minimum_Until_Point(SFun : Step_Function_t; X: Function_Range)
21                                    return Float is
22      min : Float := SFun.Step(Num_Delimiters_Range'First).Value;
23      begin
24          for i in Num_Delimiters_Range'First .. SFun.Number_Of_Delimiters loop
25              Pragma Loop_Invariant
26                (for all j in Num_Delimiters_Range'First..i-1 =>
27                   (if X >= SFun.Step(j).Delimiter then
28                    min <= SFun.Step(j).Value));
29              Pragma Loop_Invariant
30                (for some j in Num_Delimiters_Range'First..i =>
31                   (X >= SFun.Step(j).Delimiter
32                    and
33                      min = SFun.Step(j).Value));
34
35              if X >= SFun.Step(i).Delimiter then
36                  if SFun.Step(i).Value < min then min := SFun.Step(i).Value; end if;
37              else
38                  Pragma Assert
39                    (for all j in i+1..SFun.Number_Of_Delimiters =>
40                       SFun.Step(j-1).Delimiter < SFun.Step(j).Delimiter);
41              end if;
42          end loop;
43
44          return min;
45      end Minimum_Until_Point;
46
47      procedure Index_Increment(SFun: Step_Function_t;
48                                    i: in out Num_Delimiters_Range;
```

```
49                                      scan : in out Boolean ) is
50      begin
51          if i < SFun . Number_Of_Delimiters then
52              i := i + 1;
53          else
54              scan := False ;
55          end if ;
56      end ;
57
58      procedure Restrictive_Merge (SFun1, SFun2 : in Step_Function_t ;
59                                   Merge : out Step_Function_t) is
60   --      begin
61   --          null ;
62   --      end ;
63          i1 : Num_Delimiters_Range := 0;
64          i2 : Num_Delimiters_Range := 0;
65          im : Num_Delimiters_Range := 0;
66          scan_sfun1 : Boolean := True ;
67          scan_sfun2 : Boolean := True ;
68      begin
69          Pragma Assert (SFun1 . Step (0). Delimiter = SFun2 . Step (0). Delimiter );
70          loop
71              -- im , i1 and i2 bounds
72              Pragma Loop_Invariant ( i1 >= 0);
73              Pragma Loop_Invariant ( i2 >= 0);
74              Pragma Loop_Invariant (im >= 0);
75              Pragma Loop_Invariant ( i1 <= SFun1 . Number_Of_Delimiters );
76              Pragma Loop_Invariant ( i2 <= SFun2 . Number_Of_Delimiters );
77              Pragma Loop_Invariant ( i1 + i2 <= Num_Delimiters_Range ' Last );
78              Pragma Loop_Invariant (im <= Num_Delimiters_Range ' Last );
79              Pragma Loop_Invariant (im <= i1 + i2 );
80
81              -- Merge is a valid step function until im
82              Pragma Loop_Invariant ( for all i in 1.. im−1 =>
83                              Merge . Step ( i −1). Delimiter < Merge . Step ( i ). Delimiter );
84
85              -- All merged delimiters are coming from valid delimiter in SFun1 or
86              -- SFun2
87              Pragma Loop_Invariant
88                ( for all i in 0.. i1 −1 =>
89                    (( for some j in 0.. im−1 =>
90                        SFun1 . Step ( i ). Delimiter = Merge . Step ( j ). Delimiter )));
91              Pragma Loop_Invariant
92                ( for all i in 0.. i2 −1 =>
93                    (( for some j in 0.. im−1 =>
94                        SFun2 . Step ( i ). Delimiter = Merge . Step ( j ). Delimiter )));
95
96              -- Merged value at a delimiter is the minimum of both step functions
97              Pragma Loop_Invariant
98                ( for all i in 0.. im−1 =>
99                    Merge . Step ( i ). Value =
100                   Min ( Get_Value (SFun1, Merge . Step ( i ). Delimiter ),
101                     Get_Value (SFun2, Merge . Step ( i ). Delimiter )));
102
103             if scan_sfun1 and scan_sfun2 then
104                 -- select on delimiter from SFun1 or SFun2
105                 if SFun1 . Step ( i1 ). Delimiter < SFun2 . Step ( i2 ). Delimiter then
106                     Merge . Step (im ). Delimiter := SFun1 . Step ( i1 ). Delimiter ;
107                     Merge . Step (im ). Value :=
108                         Min ( Get_Value (SFun1, Merge . Step (im ). Delimiter ),
109                             Get_Value (SFun2, Merge . Step (im ). Delimiter ));
110                     Index_Increment (SFun1, i1, scan_sfun1 );
111
```

```
112            elsif SFun1.Step(i1).Delimiter > SFun2.Step(i2).Delimiter then
113                Merge.Step(im).Delimiter := SFun2.Step(i2).Delimiter;
114                Merge.Step(im).Value :=
115                  Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
116                      Get_Value(SFun2, Merge.Step(im).Delimiter));
117                Index_Increment(SFun2, i2, scan_sfun2);
118
119            else -- SFun1.Step(i1).Delimiter = SFun2.Step(i2).Delimiter
120                Merge.Step(im).Delimiter := SFun1.Step(i1).Delimiter;
121                Merge.Step(im).Value :=
122                  Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
123                      Get_Value(SFun2, Merge.Step(im).Delimiter));
124                Index_Increment(SFun1, i1, scan_sfun1);
125                Index_Increment(SFun2, i2, scan_sfun2);
126            end if;
127          elsif scan_sfun1 then
128            -- only use SFun1 delimiter
129            Merge.Step(im).Delimiter := SFun1.Step(i1).Delimiter;
130            Merge.Step(im).Value :=
131              Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
132                  Get_Value(SFun2, Merge.Step(im).Delimiter));
133            Index_Increment(SFun1, i1, scan_sfun1);
134          else -- scan_sfun2
135            -- only use SFun2 delimiter
136            Merge.Step(im).Delimiter := SFun2.Step(i2).Delimiter;
137            Merge.Step(im).Value :=
138              Min(Get_Value(SFun1, Merge.Step(im).Delimiter),
139                  Get_Value(SFun2, Merge.Step(im).Delimiter));
140            Index_Increment(SFun2, i2, scan_sfun2);
141          end if;
142
143          Pragma Assert (if scan_sfun1 or scan_sfun2 then im < i1 + i2);
144          if scan_sfun1 or scan_sfun2 then
145            im := im + 1;
146          else
147            exit;
148          end if;
149        end loop;
150
151        Merge.Number_Of_Delimiters := im;
152    end Restrictive_Merge;
153 end Step_Function;
```

```
1  with Step_Function; use Step_Function;
2  with GNAT.IO; use GNAT.IO;
3
4  procedure Step_Function_Test is
5      SFun1 : Step_Function_t :=
6        (Number_Of_Delimiters => 2,
7         Step => ((Delimiter => 0, Value => 3.0),
8                  (Delimiter => 3, Value => 2.0),
9                  (Delimiter => 5, Value => 5.0),
10                 others => (Delimiter => 0, Value => 0.0)));
11
12     SFun2 : Step_Function_t :=
13       (Number_Of_Delimiters => 2,
14        Step => ((Delimiter => 0, Value => 1.0),
15                 (Delimiter => 3, Value => 1.0),
16                 (Delimiter => 5, Value => 3.0),
17                 others => (Delimiter => 0, Value => 0.0)));
18
19     sfun3 : Step_Function_t :=
20       (Number_Of_Delimiters => 5,
```

```
21        Step => (( Delimiter => 0, Value => 1.0),
22                  ( Delimiter => 1, Value => 1.0),
23                  ( Delimiter => 3, Value => 3.0),
24                  ( Delimiter => 5, Value => 5.0),
25                  ( Delimiter => 7, Value => 7.0),
26                  ( Delimiter => 9, Value => 9.0),
27                  others => ( Delimiter => 0, Value => 0.0)));
28
29    sfun4  :  Step_Function_t  :=
30      (Number_Of_Delimiters => 5,
31        Step => (( Delimiter => 0, Value => 10.0),
32                  ( Delimiter => 2, Value => 8.0),
33                  ( Delimiter => 4, Value => 6.0),
34                  ( Delimiter => 6, Value => 4.0),
35                  ( Delimiter => 8, Value => 2.0),
36                  ( Delimiter => 10, Value => 0.5),
37                  others => ( Delimiter => 0, Value => 0.0)));
38
39    sfun_merge  :  Step_Function_t ;
40 begin
41    Pragma Assert ( Is_Valid (SFun1 ));
42    Pragma Assert ( Is_Valid (SFun2 ));
43    Pragma Assert ( Step_Function . Is_Valid ( sfun3 ));
44    Pragma Assert ( Step_Function . Is_Valid ( sfun4 ));
45
46    Pragma Assert ( Get_Value (SFun1 , 0) = 3.0);
47    Pragma Assert ( Get_Value (SFun1 , 1) = 3.0);
48    Pragma Assert ( Get_Value (SFun1 , 3) = 2.0);
49    Pragma Assert ( Get_Value (SFun1 , 4) = 2.0);
50    Pragma Assert ( Get_Value (SFun1 , 5) = 5.0);
51    Pragma Assert ( Get_Value (SFun1 ,
52      Function_Range ' Last ) = 5.0);
53
54    Pragma Assert ( Has_Same_Delimiters (SFun1 , SFun2 ));
55
56    Restrictive_Merge ( sfun3 , sfun4 , sfun_merge );
57
58    for  i  in  Function_Range ' First ..12  loop
59 --        Put ( Float ' Image ( Get_Value ( sfun_merge , i )));
60 --        New_line ;
61      Pragma Assert ( Get_Value ( sfun_merge , i )
62                  = Min ( Get_Value ( sfun3 , i ), Get_Value ( sfun4 , i )));
63    end loop ;
64
65    Pragma Assert ( Minimum_Until_Point ( sfun4 , 1) = 10.0);
66    Pragma Assert ( Minimum_Until_Point ( sfun4 , 5) = 6.0);
67    Pragma Assert ( Minimum_Until_Point ( sfun_merge , 11) = 0.5);
68 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.

```
1   with Units; use Units;
2
3   package Deceleration_Curve is
4      Distance_Resolution : constant Distance_t := 5; -- m
5
6      Maximum_Valid_Speed : constant Speed_t :=
7        m_per_s_From_km_per_h(Maximum_Valid_Speed_km_per_h);
8
9      Minimum_Valid_Acceleration : constant Acceleration_t := -10.0; -- FIXME: realistic value?
10
11     type Braking_Curve_Range is range 0..1_000;
12
13     Braking_Curve_Maximum_End_Point : constant Distance_t :=
14       Distance_t(Braking_Curve_Range'Last - Braking_Curve_Range'First)
15       * Distance_Resolution;
16
17     type Braking_Curve_Entry is
18        record
19           location : Distance_t;
20           speed : Speed_t;
21        end record;
22
23     type Braking_Curve_Array is array (Braking_Curve_Range)
24       of Braking_Curve_Entry;
25
26     type Braking_Curve_t is
27        record
28           curve : Braking_Curve_Array;
29           end_point : Distance_t;
30        end record;
31
32     -- SUBSET-026-3.13.8.1.1
33     type Target_t is
34        record
35           supervise : Boolean;
36           location : Distance_t;
37           speed : Speed_t;
38        end record;
39
40     function Distance_To_Speed(Initial_Speed, Final_Speed: Speed_t;
41                               Acceleration: Acceleration_t)
42                               return Distance_t
43     with
44       Pre => (Initial_Speed > 0.0 and Final_Speed >= 0.0
45              and
46                Initial_Speed <= Maximum_Valid_Speed
47              and
48                Initial_Speed > Final_Speed
49              and
50                Acceleration < 0.0
51              and
52                Acceleration >= Minimum_Valid_Acceleration);
53
54     function Curve_Index_From_Location(d : Distance_t)
55                                        return Braking_Curve_Range
56     with
57       Pre => (d <= Braking_Curve_Maximum_End_Point);
58
59     procedure Curve_From_Target(Target : Target_t;
60                                Braking_Curve : out Braking_Curve_t)
61     with
62       Pre => (Target.location <= Braking_Curve_Maximum_End_Point);
```

```
63
64      procedure Print_Curve(Braking_Curve : Braking_Curve_t);
65  end Deceleration_Curve;
```

```
1   with Units; use Units;
2   with Ada.Numerics.Generic_Elementary_Functions;
3   with GNAT.IO; use GNAT.IO;
4   with sec_3_13_6_deceleration; use sec_3_13_6_deceleration;
5
6   package body Deceleration_Curve is
7      Minimum_Valid_Speed : constant Speed_t := 0.1; -- m/s
8
9      function Distance_To_Speed(Initial_Speed, Final_Speed: Speed_t;
10                                Acceleration: Acceleration_t)
11                                return Distance_t is
12         speed : Speed_t := Initial_Speed;
13         delta_speed : Speed_t;
14         distance : Distance_t := 0;
15     begin
16        while speed > final_speed and speed > Minimum_Valid_Speed loop
17           Pragma Assert (Minimum_Valid_Acceleration <= Acceleration
18                          and Acceleration < 0.0);
19           Pragma Loop_Invariant
20             (Minimum_Valid_Speed < speed and speed <= Initial_Speed);
21           Pragma Assert (0.0 < 1.0/speed and 1.0/speed < 1.0 / Minimum_Valid_Speed);
22           Pragma assert
23             ((Speed_t(Minimum_Valid_Acceleration) / Minimum_Valid_Speed)
24              <= Speed_t(Acceleration) / speed);
25           Pragma assert
26             ((Speed_t(Minimum_Valid_Acceleration) / Minimum_Valid_Speed)
27              * Speed_t(Distance_Resolution)
28              <= (Speed_t(Acceleration) / speed) * Speed_t(Distance_Resolution));
29
30           delta_speed := (Speed_t(Acceleration) / speed)
31             * Speed_t(Distance_Resolution);
32
33           Pragma Assert
34             ((Speed_t(Minimum_Valid_Acceleration) / Minimum_Valid_Speed)
35              * Speed_t(Distance_Resolution) <= delta_speed
36              and
37                delta_speed < 0.0);
38
39           speed := speed + delta_speed;
40
41           distance := distance + Distance_Resolution;
42        end loop;
43
44        return distance;
45     end;
46
47      function Curve_Index_From_Location(d : Distance_t)
48                                         return Braking_Curve_Range is
49     begin
50        return Braking_Curve_Range(d / Distance_Resolution);
51     end;
52
53      procedure Curve_From_Target(Target : Target_t;
54                                  Braking_Curve : out Braking_Curve_t) is
55         package Speed_Math is
56           new Ada.Numerics.Generic_Elementary_Functions(Speed_t);
57         use Speed_Math;
58
59         speed : Speed_t := Target.speed;
```

```
60        location : Distance_t := Target.location;
61        end_point : constant Braking_Curve_Range :=
62          Curve_Index_From_Location(Target.location);
63     begin
64        Braking_Curve.end_point := Target.location;
65        Braking_Curve.curve(end_point).location := location;
66        Braking_Curve.curve(end_point).speed := speed;
67
68        for i in reverse Braking_Curve_Range'First .. end_point - 1 loop
69           speed :=
70             (speed + Sqrt(speed * speed
71             + (Speed_t(4.0) * Speed_t(A_safe(speed, location)))
72             * Speed_t(Distance_Resolution))) / 2.0;
73           if speed > Maximum_Valid_Speed then
74              speed := Maximum_Valid_Speed;
75           end if;
76
77           location := Distance_t(i) * Distance_Resolution;
78
79           Braking_Curve.curve(i).location := location;
80           Braking_Curve.curve(i).speed := speed;
81        end loop;
82     end Curve_From_Target;
83
84     procedure Print_Curve(Braking_Curve : Braking_Curve_t) is
85     begin
86        for i in Braking_Curve_Range'First ..
87          Curve_Index_From_Location(Braking_Curve.end_point) loop
88           Put(Distance_t'Image(Braking_Curve.curve(i).location));
89           Put(",␣␣␣");
90           Put(Speed_km_per_h_t'Image(
91             km_per_h_From_m_per_s(Braking_Curve.curve(i).speed)));
92           New_Line;
93
94           if Braking_Curve.curve(i).location >= Braking_Curve.end_point then
95              exit;
96           end if;
97        end loop;
98     end Print_Curve;
99  end Deceleration_Curve;
```

```
1   with GNAT.IO; use GNAT.IO;
2   with Units; use Units;
3   with Deceleration_Curve; use Deceleration_Curve;
4
5   procedure Deceleration_Curve_Test is
6      initial_speed : Speed_t := m_per_s_From_km_per_h(160.0); -- 160 km/h
7
8      target : Target_t := (supervise => True,
9                            location => 2500,
10                           speed => 0.0);
11     braking_curve : Braking_Curve_t;
12  begin
13  --         Put (Distance_t'Image(Distance_To_Speed(initial_speed, 0.0, -1.0)));
14  --         New_line;
15     pragma Assert (Distance_To_Speed(initial_speed, 0.0, -1.0) = 1000);
16
17     Curve_From_Target(target, braking_curve);
18     Print_Curve(braking_curve);
19  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.

```
1   -- Reference: UNISIG SUBSET-026-3 v3.3.0
2
3   with Units; use Units;
4   with Step_Function; use Step_Function;
5
6   package sec_3_13_2_monitoring_inputs is
7      -- *** section 3.13.2.2 Train related inputs ***
8      -- ** section 3.13.2.2.1 Introduction **
9
10     -- SUBSET-026-3.13.2.2.1.1 not formalized (description)
11
12     -- SUBSET-026-3.13.2.2.1.2 not formalized
13
14     -- SUBSET-026-3.13.2.2.1.3
15     type Breaking_Model_t is (Train_Data_Model, Conversion_Model);
16     -- Only Train Data model is modelized
17     Breaking_Model : constant Breaking_Model_t := Train_Data_Model;
18
19
20     -- ** section 3.13.2.2.2 Traction model **
21
22     -- SUBSET-026-3.13.2.2.2.1
23     T_traction_cut_off : constant Time_t := 10.0; -- s -- FIXME: realistic value?
24
25     -- SUBSET-026-3.13.2.2.2.2 not formalized (Note)
26
27
28     -- ** section 3.13.2.2.3 Braking Models **
29
30     -- SUBSET-026-3.13.2.2.3.1.1
31     -- Use Step_Function.Step_Function_t type
32
33     -- SUBSET-026-3.13.2.2.3.1.2
34     -- Note: It would be better to modelize this as Data type invariant
35     function Is_Valid_Deceleration_Model(S : Step_Function_t) return Boolean is
36       (Step_Function.Is_Valid(S)
37        and
38          (S.Number_Of_Delimiters <= 6)); -- 6 delimiters for 7 steps
39
40     -- SUBSET-026-3.13.2.2.3.1.3 not formalized (Note)
41
42     -- SUBSET-026-3.13.2.2.3.1.4
43     -- by definition of Step_Function.Step_Function_t
44
45     -- SUBSET-026-3.13.2.2.3.1.5 not formalized (FIXME?)
46
47     -- SUBSET-026-3.13.2.2.3.1.6
48     A_brake_emergency_model : constant Step_Function_t :=
49       (Number_Of_Delimiters => 0,
50        Step => ((0, 1.0), -- (from 0 m/s, 1 m/s**2)
51                 others => (0, 0.0)));
52
53     A_brake_service_model : Step_Function_t; -- FIXME give value, set constant
54
55     A_brake_normal_service_model : Step_Function_t; -- FIXME give value, set constant
56
57     -- SUBSET-026-3.13.2.2.3.1.7 not formalized (we do not consider regenerative
58     -- brake, eddy current brake and magnetic shoe brake)
```

```
59
60      −− SUBSET−026−3.13.2.2.3.1.8 not formalized (Note)
61
62      −− SUBSET−026−3.13.2.2.3.1.9
63      type Brake_Position_t is (Freight_Train_In_G, Passenger_Train_In_P,
64                                Freight_Train_In_P);
65
66
67      A_SB01 : constant Deceleration_t := 0.1;
68      A_SB02 : constant Deceleration_t := 0.2;
69
70      −− SUBSET−026−3.13.2.2.3.1.10 not formalized FIXME
71  −−       function A_Brake_normal_service(V : Speed_t; position : Brake_Position_t)
72  −−                                       return Deceleration_t;
73
74      −− SUBSET−026−3.13.2.2.3.1.11 not formalized (Note)
75
76      −− SUBSET−026−3.13.2.2.3.2.1 not formalized (description)
77      −− SUBSET−026−3.13.2.2.3.2.2 not formalized (figure)
78
79      −− SUBSET−026−3.13.2.2.3.2.3
80      T_brake_react : constant Time_t := 1.0; −− s
81      T_brake_increase : constant Time_t := 2.0; −− s
82
83      −− SUBSET−026−3.13.2.2.3.2.4
84      T_brake_build_up : constant Time_t := T_brake_react + 0.5 ∗ T_brake_increase;
85
86      −− SUBSET−026−3.13.2.2.3.2.5
87      T_brake_emergency_react : constant Time_t := T_brake_react;
88      T_brake_emergency_increase : constant Time_t := T_brake_increase;
89      T_brake_emergency : constant Time_t :=
90        T_brake_emergency_react + 0.5 ∗ T_brake_emergency_increase;
91
92      T_brake_service_react : constant Time_t := T_brake_react;
93      T_brake_service_increase : constant Time_t := T_brake_increase;
94      T_brake_service : constant Time_t :=
95        T_brake_service_react + 0.5 ∗ T_brake_service_increase;
96
97      −− SUBSET−026−3.13.2.2.3.2.6 not formalized (Note)
98
99      −− SUBSET−026−3.13.2.2.3.2.7 not formalized (Note)
100
101     −− SUBSET−026−3.13.2.2.3.2.8 not formalized (we do not consider regenerative
102     −− brake, eddy current brake and magnetic shoe brake)
103
104     −− SUBSET−026−3.13.2.2.3.2.9 not formalized (Note)
105
106     −− SUBSET−026−3.13.2.2.3.2.10 not formalized (Note)
107
108     −− ∗∗ section 3.13.2.2.4 Brake Position ∗∗
109
110     −− SUBSET−026−3.13.2.2.4.1
111     −− see type Brake_Position_t definition above
112
113     −− SUBSET−026−3.13.2.2.4.2 not formalized (Note)
114
115
116     −− ∗∗ section 3.13.2.2.5 Brake Percentage ∗∗ not formalized (conversion model
117     −− not used)
118
119     −− ∗∗ section 3.13.2.2.6 Special Brakes ∗∗ not formalized (special brake not
120     −− modelized)
121
```

```
122
123      -- ** section 3.13.2.2.7 Service brake interface **
124
125      -- SUBSET-026-3.13.2.2.7.1
126      Service_Brake_Command_Implemented : constant Boolean := True;
127
128      -- SUBSET-026-3.13.2.2.7.2
129      Service_Brake_Feedback_Implemented : constant Boolean := True;
130
131      -- ** section 3.13.2.2.8 Traction cut-off interface **
132
133      -- SUBSET-026-3.13.2.2.8.1
134      Traction_Cut_Off_Command_Implemented : constant Boolean := True;
135
136
137      -- ** section 3.13.2.2.9 On-borad Correction Factors **
138
139      -- SUBSET-026-3.13.2.2.9.1.1 not formalized (description)
140
141      -- SUBSET-026-3.13.2.2.9.1.2
142      Kdry_rst_model : constant Step_Function_t :=
143        (Number_Of_Delimiters => 0,
144         Step => ((0, 1.0), -- (from 0 m/s, 1.0)
145                  others => (0, 0.0)));
146
147      Kwet_rst_model : constant Step_Function_t :=
148        (Number_Of_Delimiters => 0,
149         Step => ((0, 1.0), -- (from 0 m/s, 1.0)
150                  others => (0, 0.0)));
151
152      -- SUBSET-026-3.13.2.2.9.1.3
153      -- FIXME EBCL parameter not formalized
154      function Is_Valid_Kdry_rst return Boolean is
155        (Step_Function.Is_Valid(Kdry_rst_model)
156         and
157           (Has_Same_Delimiters(Kdry_rst_model, A_brake_emergency_model)));
158
159      function Kdry_rst(V: Speed_t) return Float
160      with
161        Pre => Is_Valid_Kdry_rst,
162        Post =>
163          (Kdry_rst'Result
164           = Step_Function.Get_Value(SFun => Kdry_rst_model,
165                                     X    => Function_Range(V)));
166
167      -- SUBSET-026-3.13.2.2.9.1.4 not formalized (FIXME)
168
169      -- SUBSET-026-3.13.2.2.9.1.5
170      function Is_Valid_Kwet_rst return Boolean is
171        (Step_Function.Is_Valid(Kwet_rst_model)
172         and
173           (Has_Same_Delimiters(Kwet_rst_model, A_brake_emergency_model)));
174
175      function Kwet_rst(V: Speed_t) return Float
176      with
177        Pre => Is_Valid_Kwet_rst,
178        Post =>
179          (Kwet_rst'Result
180           = Step_Function.Get_Value(SFun => Kwet_rst_model,
181                                     X    => Function_Range(V)));
182
183      -- SUBSET-026-3.13.2.2.9.2.1
184      type Gradient_Range is new Float range 0.0 .. 10.0; -- m/s**2
```

```
185
186     Kn_Plus : Step_Function_t;
187     Kn_Minus : Step_Function_t;
188
189     -- SUBSET-026-3.13.2.2.9.2.2
190     -- Note: It would be better to modelize this as Data type invariant
191     function Is_Valid_Kn return Boolean is
192       (Step_Function.Is_Valid(Kn_Plus) and Step_Function.Is_Valid(Kn_Minus)
193         and
194           (Kn_Plus.Number_Of_Delimiters <= 4) -- 4 delimiters for 5 steps
195         and
196           (Kn_Minus.Number_Of_Delimiters <= 4)); -- 4 delimiters for 5 steps
197
198     -- SUBSET-026-3.13.2.2.9.2.3 not formalized (Note)
199
200     -- SUBSET-026-3.13.2.2.9.2.4 not formalized (FIXME)
201
202     -- SUBSET-026-3.13.2.2.9.2.5 not formalized (FIXME)
203
204     -- SUBSET-026-3.13.2.2.9.2.6
205     -- By definition of Step_Function_t
206
207     -- ** section 3.13.2.2.10 Nominal Rotating mass **
208
209     -- SUBSET-026-3.13.2.2.10.1 not formalized (FIXME)
210
211     -- ** section 3.13.2.2.11 Train length **
212
213     -- SUBSET-026-3.13.2.2.11.1
214     Train_Length : constant Distance_t := 900; -- m
215
216     -- ** section 3.13.2.2.12 Fixed Values **
217
218     -- SUBSET-026-3.13.2.2.12.1 not formalized (description)
219
220     -- ** section 3.13.2.2.13 Maximum train speed **
221
222     -- SUBSET-026-3.13.2.2.12.1
223     Maximum_Train_Speed : constant Speed_t := m_per_s_From_km_per_h(250.0);
224
225     -- *** section 3.13.2.3 Trackside related inputs ***
226     -- all sections of 3.13.2.3 not formalized
227     procedure dummy;
228 end sec_3_13_2_monitoring_inputs;
```

```
1  -- Reference: UNISIG SUBSET-026-3 v3.3.0
2
3  package body sec_3_13_2_monitoring_inputs is
4  --      function A_Brake_normal_service(V : Speed_t; position : Brake_Position_t)
5  --                                      return Deceleration_t is
6  --      begin
7  --          return 0.0;
8  --      end;
9      function Kdry_rst(V: Speed_t) return Float is
10     begin
11         return Step_Function.Get_Value(SFun => Kdry_rst_model,
12                                        X    => Function_Range(V));
13     end;
14
15     function Kwet_rst(V: Speed_t) return Float is
16     begin
17         return Step_Function.Get_Value(SFun => Kwet_rst_model,
18                                        X    => Function_Range(V));
```

```
19      end ;
20
21      procedure dummy is
22      begin
23         null ;
24      end ;
25
26  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.

```
1   with Units ; use Units ;
2
3   package sec_3_13_4_gradient_accel_decel is
4      -- FIXME 3.13.4 not formalized
5
6      function A_gradient (d: Distance_t) return Deceleration_t is
7         (0.0) ;
8
9   end sec_3_13_4_gradient_accel_decel ;
```

```
1   with Units ; use Units ;
2
3   with Step_Function ; use Step_Function ;
4   with Appendix_A_3_1 ; use Appendix_A_3_1 ;
5   with sec_3_13_2_monitoring_inputs ; use sec_3_13_2_monitoring_inputs ;
6   with sec_3_13_4_gradient_accel_decel ; use sec_3_13_4_gradient_accel_decel ;
7
8   package sec_3_13_6_deceleration is
9      -- SUBSET-026-3.13.6.2.1 to 3.13.6.2.1.2 not formalized (FIXME)
10
11     -- SUBSET-026-3.13.6.2.1.5 (Note .5 before .4 for proper definition)
12     -- Note: we are not using specific break configuration so parameter 'd' is
13     -- never used
14     function A_brake_emergency (V: Speed_t; d: Distance_t) return Deceleration_t
15     with
16        Pre => (Is_Valid_Deceleration_Model (A_brake_emergency_model)
17                  and Is_Valid_Speed (V)) ,
18     Post =>
19        (A_brake_emergency ' Result
20        = Deceleration_t (Step_Function. Get_Value (SFun => A_brake_emergency_model ,
21                                                    X    => Function_Range (V)))) ;
22
23     -- SUBSET-026-3.13.6.2.1.4 (Note .4 before .3 for proper definition)
24     function A_brake_safe (V: Speed_t; d: Distance_t) return Deceleration_t is
25        (Deceleration_t ((Kdry_rst (V)
26                       * (KWet_rst (V) + M_NVAVADH * (1.0 - Kwet_rst (V))))
27                       * Float (A_brake_emergency (V,d)))) ;
28
29     -- SUBSET-026-3.13.6.2.1.3
30     -- FIXME reduced adhesion condition not formalized
31     function A_safe (V: Speed_t; d: Distance_t) return Deceleration_t is
32        (A_brake_safe (V, d) + A_gradient (d)) ;
33
34     -- SUBSET-026-3.13.6.2.1.6 not formalized (conversion model not used)
35
36     -- SUBSET-026-3.13.6.2.1.7 not formalized (same requirements as
37     -- SUBSET-026-3.13.2.2.9.1.3)
38
```

```
39    −− SUBSET−026−3.13.6.2.1.8 not formalized (conversion model not used)
40    −− SUBSET−026−3.13.6.2.1.8.1 not formalized (conversion model not used)
41    −− SUBSET−026−3.13.6.2.1.8.2 not formalized (conversion model not used)
42
43    −− SUBSET−026−3.13.6.2.1.9 not formalized (Note)
44
45    −− SUBSET−026−3.13.6.2.2.1 not formalized (description)
46
47    −− SUBSET−026−3.13.6.2.2.2.a not formalized (FIXME?)
48    −− SUBSET−026−3.13.6.2.2.2.b not formalized (conversion model not used)
49    −− SUBSET−026−3.13.6.2.2.2.c not formalized (various brakes not modelled)
50
51    −− SUBSET−026−3.13.6.2.2.3
52    T_be : constant Time_t := T_brake_emergency;
53  end sec_3_13_6_deceleration;
```

```
1   package body sec_3_13_6_deceleration is
2      function A_brake_emergency(V: Speed_t; d: Distance_t) return Deceleration_t
3      is
4      begin
5         return
6            Deceleration_t(Step_Function.Get_Value(SFun => A_brake_emergency_model,
7                                                   X    => Function_Range(V)));
8      end;
9   end sec_3_13_6_deceleration;
```

```
1   with Units; use Units;
2   with sec_3_13_6_deceleration; use sec_3_13_6_deceleration;
3   with Deceleration_Curve; use Deceleration_Curve;
4
5   package sec_3_13_8_targets_decel_curves is
6      −− ** 3.13.8.1 Introduction **
7
8      −− SUBSET−026−3.13.8.1.1
9      −− Defined in Deceleration_Curve package
10
11     −− SUBSET−026−3.13.8.1.2
12     −− Use of package sec_3_13_6_deceleration
13
14     −− SUBSET−026−3.13.8.1.3 not formalized (FIXME?)
15
16     −− ** 3.13.8.2 Determination of the supervised targets **
17
18     −− SUBSET−026−3.13.8.2.1
19     type Target_Type is (MRSP_Speed_Decrease, −− SUBSET−026−3.13.8.2.1.a
20                          Limit_Of_Authority, −− SUBSET−026−3.13.8.2.1.b
21                          −− SUBSET−026−3.13.8.2.1.c
22                          End_Of_Authority, Supervised_Location,
23                          Staff_Responsible_Maximum); −− SUBSET−026−3.13.8.2.1.d
24
25     Target : array (Target_Type) of Target_t;
26
27     −− Note: should be defined as data type invariant
28     function Is_Valid_Target return boolean is
29        ((if Target(End_Of_Authority).speed > 0.0 then
30          Target(Limit_Of_Authority).supervise)
31        and
32          (if Target(End_Of_Authority).speed = 0.0 then
33           Target(End_Of_Authority).supervise
34           and Target(Supervised_Location).supervise)
35        and
36          Target(Staff_Responsible_Maximum).speed = 0.0);
37
```

```
38    -- SUBSET-026-3.13.8.2.1.1 not formalized (Note)
39
40    -- SUBSET-026-3.13.8.2.2 not formalized (FIXME)
41
42    -- SUBSET-026-3.13.8.2.3 not formalized (FIXME)
43
44    -- ** 3.13.8.3 Emergency Brake Deceleration Curve **
45    -- FIXME how to merge EBDs if several targets are active at the same time?
46
47    -- SUBSET-026-3.13.8.3.1 not formalized (FIXME)
48
49    -- SUBSET-026-3.13.8.3.2
50    procedure Compute_SvL_Curve(Braking_Curve : out Braking_Curve_t)
51    with
52      Pre => (Is_Valid_Target and Target(Supervised_Location).speed = 0.0);
53 --        Post => (Curve_From_Target(Target(Supervised_Location), Braking_Curve));
54 --                 True);
55
56    -- SUBSET-026-3.13.8.3.3 not formalized (FIXME)
57 end sec_3_13_8_targets_decel_curves;
```

```
1 package body sec_3_13_8_targets_decel_curves is
2    procedure Compute_SvL_Curve(Braking_Curve : out Braking_Curve_t) is
3    begin
4        Curve_From_Target(Target(Supervised_Location), Braking_Curve);
5    end;
6 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.