OETCS/WP7/T7.1

ITEA2 Project
Call 6 11025
2012 – 2015

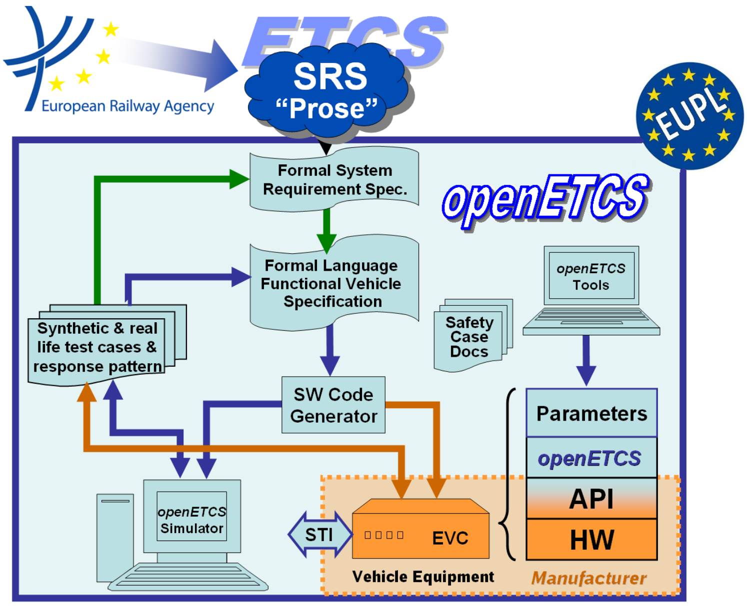
Work-Package 7: “Benchmark”

Classical B models

Introduction and notes on classical B models

Marielle Petit-Doche

May 2013



Funded by:

Federal Ministry of Education and Research

Région de Bruxelles-Capitale

GOBIERNO DE ESPAÑA MINISTERIO DE INDUSTRIA, ENERGÍA Y TURISMO

openETCS

This page is intentionally left blank

Work-Package 7: "Benchmark"

OETCS/WP7/T7.1
May 2013

Classical B models

Introduction and notes on classical B models

Marielle Petit-Doche

Systerel

Notes

Prepared for openETCS@ITEA2 Project

Abstract: This document gives a description of the models developed in Classical B during the benchmark task T7.1.

Table of Contents

Figures and Tables

Figures

Tables

Document information	
Work Package Deliverable ID or doc. ref.	WP7
Document title Document version Document authors (org.)	Classical B models 00.01 Marielle Petit-Doche (Systerel)

Review information	
Last version reviewed	
Main reviewers	

Approbation			
	Name	Role	Date
Written by	Marielle Petit-Doche		
Approved by			

Document evolution			
Version	Date	Author(s)	Justification
00.01	25/05/2013	M. Petit-Doche	Document creation

1 Introduction

1.1 Classical B method

The B-Method is a formal method developed by J-R. Abrial and used in industry, especially in railway industry, to develop complex systems. It covers software development from formal specifications to code level. Proof mechanisms guaranty the consistency of specifications properties and the complete consistency of code regarding its formal specification. It is efficient to model functional elements of a critical software with respect to EN50128 constraints.
Classical B has been used successfully in railway industry (mainly by Alstom, Siemens and AREVA) to develop critical software in urban (CBTC, PMI,...) and mainline domains (KVB, Eurobalise,...). Hundred of different systems are running in the world embedding software developed in B (see `http://www.cs.vu.nl/~wanf/pubs/handbookFFM.pdf`, `http://link.springer.com/chapter/10.1007/3-540-48119-2_22` and `http://web.tiscali.it/chiccoterri/MetodB.htm`).
Classical B is well adapted to describe how a system works and to develop functional critical software. It can be completed with the Event B method to cover system analyses and to explain why a system works.
The main publication on the method is :
[Abrial1996] The B Book, Assigning Programs to Meanings
Language is documented with language manual reference of the tool Atelier B (`http://www.atelierb.eu/ressources/manrefb1.8.6.uk.pdf`). Industrial have developed their own coding rules and guidelines.

1.2 Atelier B toolkit

AtelierB is the industrial tool the most used to develop critical software following the Classical approach. The tool is partly open-source, but it is free for use. For more details `http://www.atelierb.eu/outil-atelier-b/`.
Tool is documented with the user manual (`http://www.tools.clearsy.com/resources/User_uk.pdf`).
The version of the tool used to develop the models in the sequel is 4.0.2. However the models can be read on a more recent version of the tool.

1.3 Benchmark activities

The rest of the document describes the models developed in Classical B during the benchmark activities of WP7. The examples specified are those proposed as high priority in D2.5.

State machines procedure On-sight: § 5.9 of subset 026

Time-outs Establishing a communication session: §3.5.3 of subset 026

Arithmetic Braking curves: parts of § 3.13 of subset 026

Truth tables and logical statements Transition tables: parts of § 4.6.2 and § 4.6.3 of subset 026

2 Software Architecture of the model

2.1 Introduction

The first step when developing a Classical B model is to define the architecture of the software to model.
The language allows to give a structure to the model close to the one of the software implemented. Thus functions are shared between components : those in charge of scheduling the functions, those in charge of managing the inputs, those in charge of managing the output, those in charge of describing the functions in details... This allows a modular development of the software.
In Classical B, to describe a function (or a set of functions) we need at least two components : an abstract machine which gives an abstract specification of the functions, and an implementation to detail the functions as an implementable code (by convention the implementation has the same name as the abstract machine with the suffix "_i"). As much refinement components as

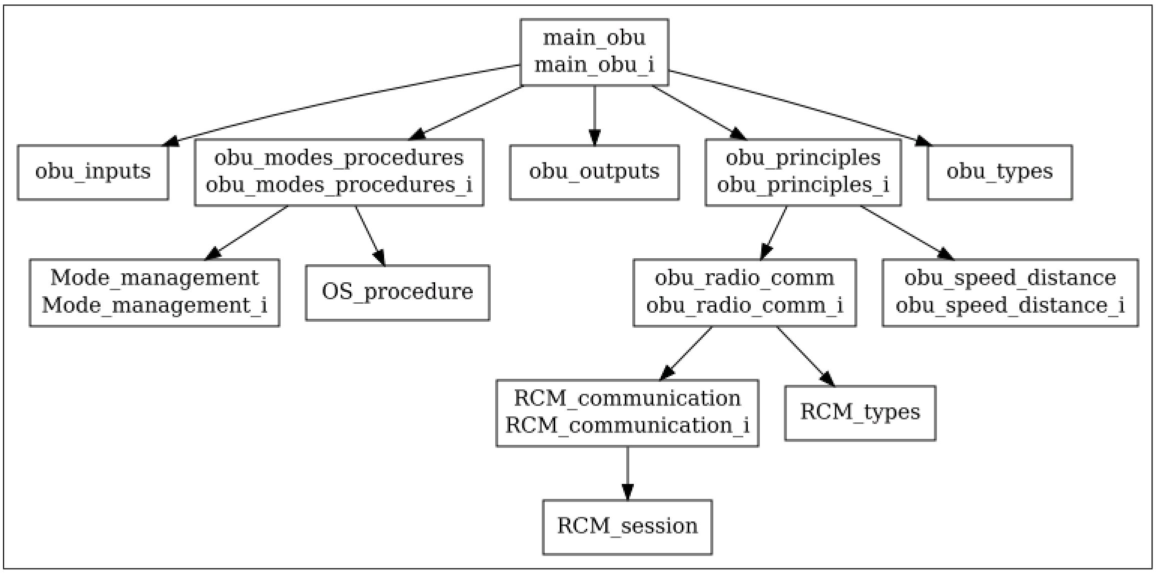


Figure 1. Classical B model architecture

necessary can be added between an abstract machine and an implementation to refine step by step the abstract specification until the implementation and to facilitate traceability. Refinement components are usually used only in the case of complex functions. This is the first mean to structure a Classical B model with abstraction level (this can be compared with the distinction between spec and body components of language such as C, Ada or Java).
The second means to structure the Classical B models is to link the group of components together. Then in a given machine or implementation :

the SEES clause allows to link the components which can be viewed in the component : Variables can be read but not modified (it works as "with" clause in C, Ada or Java software)

the IMPORT clause allows to link the components whom variables can be modified with call of functions (it works as "use" clause in C, Ada or Java software).

2.2 ETCS case study

Figure 1 gives the architecture of the ETCS example developed during benchmark activities. Arrows show IMPORT relations.

main_obu is the main sequencer of the software

obu_types contains the definition of types and constants used by all the software components. This machine is seen by all the other components.

obu_inputs stores the input values in different variables. This machine is seen by most of the other components.

obu_outputs manages the outputs.

obu_modes_procedures and branches below are in charge of the management of modes and procedures (subset 26 chapter 4 and 5).

obu_principles and branches below are in charge of the functions (chapter 3 of subset 26).

2.2.1 Main sequencer

This is the main sequencer with two operation called by the hardware. We assume an applicative cycle is defined and at each cycle the main operation "cycle" is called. The operation "power_up" is called when the system is started.

```
MACHINE
    main_obu

    OPERATIONS

    power_up =
    BEGIN
        skip
    END ;

    cycle =
    BEGIN
        skip
    END


END
```

The implementation gives the sequence of the main operations called at each cycle and during the initialisation. first the inputs at read (we assume here the inputs are read and stored at the beginning of each cycle and not on the fly). The called operation are defined in the imported components.

```
IMPLEMENTATION
  main_obu_i
REFINES
  main_obu

IMPORTS
  obu_inputs,
  obu_modes_procedures,
  obu_principles,
  obu_outputs,
  obu_types

OPERATIONS
  power_up =
  BEGIN
    initial_read_inputs ;
    intialize_mode ;
    initialize_data ;
    initial_send_outputs
  END
  ;

  cycle =
  VAR vv IN
    read_inputs ;
    vv ← get_V_train ;
    modes_procedures_management(vv) ;
    principles_management ;
    send_outputs
  END


END
```

2.2.2 Types

The component obu_types defines the types and constants used by the whole system as modes, level,..

MACHINE
obu_types

SETS
$t\_mode = \{FS, LS, OS, SR, SH, UN, PS, SL, SB, TR, PT, SF, ISo, NP, NL, SN, RV\}$
;
$t\_mamode = \{ma\_OS, ma\_SH, ma\_LS, ma\_unknown\}$
;
$t\_level = \{level\_0, level\_1, level\_2, level\_3, level\_NTC\}$
;
$t\_procedure = \{NoProcedure, StartOfMission, EndOfMission, SHInitiatedByDriver,$
$SHOrderFromTrackside, Override, OnSight, LevelTransitions, TrainTrip,$
$ChangeTrainOrientation, TrainReversing, Joining, Splitting, RBCHandover,$
$LevelCrossing, ChangingTrainData, TrackConditions, LimitedSupervision\}$

CONSTANTS
VITESSE

PROPERTIES
$VITESSE \subseteq \mathbf{NAT} \wedge$
$VITESSE = 0 \,..\, 600$

END

3 Procedure On-sight (§5.9)

4 Communication session (§3.5.3)

5 Bracking Curves (§3.13)

6 Transition table (§4.6.2)

this section is fully implemented in Classical B in the component Mode_management and Mode_management_i

```
MACHINE
    Mode_management
SEES
    obu_types,
    obu_inputs
DEFINITIONS
    condition_1 == (driver_isolate = TRUE) ;
    condition_5(v_train, ll) == ( v_train = 0 ∧ ll ∈ {level_0, level_NTC, level_1}
        ∧ driver_ask_SH = TRUE) ;
    condition_6(v_train, ll) == ( v_train = 0 ∧ ll ∈ {level_2, level_3}
        ∧ driver_ask_SH = TRUE) ;
    condition_10 == ( Valid_Train_Data = TRUE ∧ Valid_MA = TRUE
        ∧ Valid_SSP = TRUE ∧ Valid_Grad = TRUE ∧ M_MAMODE = ma_unknown) ;
    condition_50 == (SH_request_accepted = TRUE ∧ driver_ack = TRUE)
OPERATIONS
    mm, ll, pr ← manage_mode(pm, pl, vv, ppr) =
    PRE
        pm ∈ t_mode ∧
        pl ∈ t_level ∧
        vv ∈ VITESSE ∧
        mm ∈ t_mode ∧
        ll ∈ t_level ∧
        pr ∈ t_procedure ∧
        ppr ∈ t_procedure ∧
        ¬ ( (condition_5(vv, pl) ∨ condition_6(vv, pl)) ∧ condition_50 ) ∧
        ¬ ( (condition_5(vv, pl) ∨ condition_6(vv, pl)) ∧ condition_10 ) ∧
        ¬ ( condition_50 ∧ condition_10 )    THEN
        mm, ll, pr :(
            mm ∈ t_mode ∧
            ll ∈ t_level ∧
            pr ∈ t_procedure ∧
            ( condition_1 ⇒ ( mm = ISo ∧ ll = pl ∧ pr = ppr) ) ∧
            ( ¬ (condition_1) ⇒ (
                ( ( condition_5(vv, pl) ∨ condition_6(vv, pl) )
                    ∧ pm = SB ⇒ ( mm = SH ∧ ll = pl ∧ pr = ppr) ) ∧
                ( condition_50
                    ∧ pm = SB ⇒ ( mm = SH ∧ ll = pl ∧ pr = SHInitiatedByDriver) ) ∧
                ( condition_10
                    ∧ pm = SB ⇒ ( mm = FS ∧ ll = pl ∧ pr = ppr) ) ∧
                ( ¬ (condition_5(vv, pl) ∨ condition_6(vv, pl) ∨ condition_50 ∨ condition_10)
                    ∧ pm = SB ⇒ ( mm = pm ∧ ll = pl ∧ pr = ppr) ) ∧
                ( ¬ (pm=SB) ⇒ ( mm = pm ∧ ll = pl ∧ pr = ppr) )
                ) ) ∧
            ( pm = ISo ⇒ mm = ISo )    ∧
            ( (driver_isolate = FALSE ∧ pm ≠ ISo) ⇒ mm ≠ ISo )
            )
    END
END
```

```
IMPLEMENTATION
  Mode_management_i
REFINES
  Mode_management
SEES
  obu_types ,
  obu_inputs
OPERATIONS
  mm , ll , pr ← manage_mode ( pm , pl , vv , ppr) =
  VAR di, daSH, SHra, da, vtd, vma, vssp, vg, mma IN
     mm := pm ;
     ll := pl ;
     pr := ppr ;
     di ← get_driver_isolate ;
     IF di = TRUE
      THEN
        mm := ISo
      ELSE
        IF pm = SB
        THEN
           daSH   ← get_driver_ask_SH;
           IF (vv = 0 ∧ daSH = TRUE)
           THEN
              mm := SH
           ELSE
              da   ← get_driver_ack ;
              SHra   ← get_SH_request_accepted;
              IF (da =   TRUE ∧ SHra = TRUE )
              THEN
                 mm :=   SH ;
                 pr := SHInitiatedByDriver
              ELSE
                  vtd ← get_Valid_Train_Data ;
                  vma ← get_Valid_MA ;
                  vssp ← get_Valid_SSP ;
                  vg ← get_Valid_Grad ;
                  mma ← get_M_MAMODE ;
                  IF vtd =   TRUE ∧ vma = TRUE ∧ vssp = TRUE ∧
                        vg = TRUE ∧ mma = ma_unknown
                  THEN
                     mm := FS
                  ELSE
                     skip
                  END
               END
            END
         END
      END
   END


END
```

The property **PROPERTY_4.6.2_01** "OBU shall never enter in isolated mode if not requested by the driver", is specified as postcondition of the operation of obu_modes_procedures :

$((driver\_isolate = \mathbf{FALSE} \wedge previous\_mode \neq ISo) \Rightarrow \quad M\_mode \neq ISo)$

The property **PROPERTY_4.6.2_02** "OBU shall never leave Isolated mode (no transition from Isolation is specified) ie. if system was previously isolate, it stay isolate", is specified as an invariant of obu_modes_procedures :

$(previous\_mode = ISo \Rightarrow M\_mode = ISo)$