OETCS

openETCS

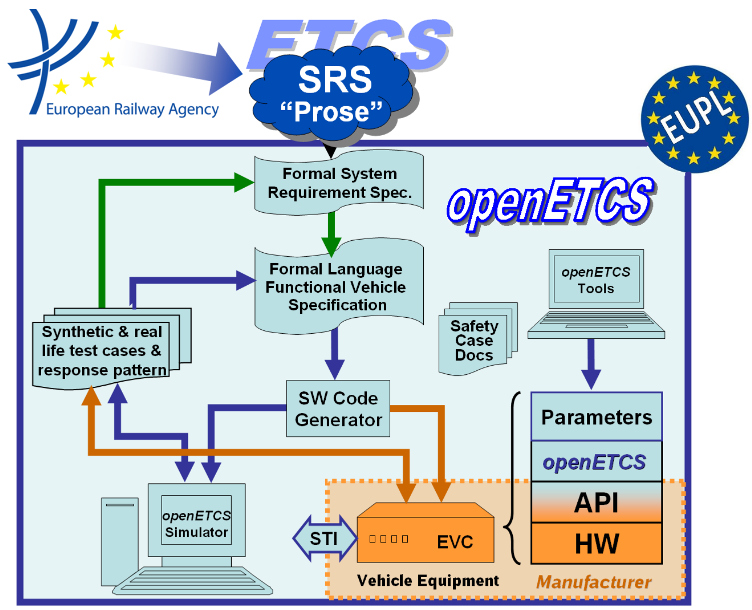
ITEA2 Project
Call 6 11025
2012 – 2015

Work-Package 7: "Toolchain"

Event-B Model of Subset 026, Section 5.9

Matthias Güdemann

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

Work-Package 7: “Toolchain”

OETCS
June 2013

Event-B Model of Subset 026, Section 5.9

Matthias Güdemann

Systerel
Les Portes de l’Arbois, Bâtiment A
1090 rue René Descartes
13857 Aix-en-Provence Cedex 3, France

Model Description

Prepared for openETCS@ITEA2 Project

Table of Contents

Figures and Tables

Figures

Tables

This document describes a formal model of the requirements of section 5.9 of the subset 026 of the ETCS specification 3.3.0 [Eur12]. This section describes the on-sight procedure.

The model is expressed in the formal language Event-B [Abr10] and developed within the Rodin tool [Jas12]. This formalism allows an iterative modeling approach. In general, one starts with a very abstract description of the basic functionality and step-wise adds additional details until the desired level of accuracy of the model is reached. Rodin provides the necessary proof support to ensure the correctness of the refined behavior.

In this document we present an Event-B model of the procedure on-sight. We use the iUML plugin which allows for modeling in UML state-charts to create a graphical model of the procedure which is as close as possible as its description as flowchart in the section 5.9. The state machine is iteratively developed using the refinement feature of Event-B. At each refinement step, we present the reasoning for the step, together with newly introduced variables and events.

For a short introduction on Event-B and the usage of Rodin with models on github see `https://github.com/openETCS/model-evaluation/blob/master/model/B-Systerel/Event_B/rodin-projects-github.pdf?raw=true`

1 Modeling Strategy

The section 5.9 of the SRS describes the procedure on-sight, in particular it describes the sequence of mode changes, necessary driver acknowledge and train brake to enter OS mode, dependent on the current train mode.

For better understanding and to automate many tasks for state based modeling, we use the iUML plugin [**?**] which automatically generates Event-B code representing a state machine specification.

2 Model Overview

Figure 1 shows the structure of the Event-B model. The left column represents the abstract state machines, the right column the contexts. An arrow from one machine to another machine represents a refinement relation, an arrow from a machine to a context represents a sees relation and arrow from one context to another represents an extension relation.

The modeling starts with the very abstract possibility to establish and to terminate a communication session in the machine *m0*, the set of entities is defined in the context *c0*. This basic functionality is refined in the succeeding machines to incorporate a more detailed description of the flowchart.

3 Model Benefits

The Event-B model in Rodin has some interesting properties which are highlighted here. Some stem from the fact that Rodin is well integrated into the Eclipse platform which renders many useful plugins available, both those explicitly developed for integration with Rodin, but also other without Rodin in mind. Other interesting properties stem from the fact that Rodin and Event-B provide an extensive proof support for properties.

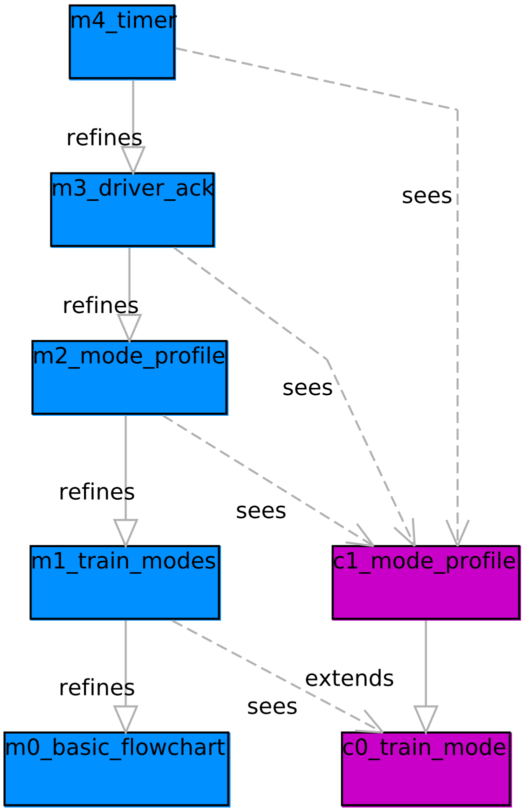


Figure 1. Overview on State Machine and Context Hierarchy

- **Graphical Modeling** Through the iUML plugin, Rodin supports graphical modeling of UML/SysML state machines. Transitions are labeled with events and a fully automatic transformation [SBS09] creates an Event-B representation of the state machine models.
- **Refinement** In addition to the general refinement which is possible in the Event-B approach, the graphical modeling allows to refine the graphical state chart models too. For each refinement step, the new details are graphically emphasized.
- **Model Animation** Through the ProB plugin, the graphical models can be animated just as textual Event-B models. In this case active transitions can be highlighted which helps understand model behavior.
- **Safety Properties** Using Rodin's proof support and the formalization as invariants, it is possible to formalize and prove the identified safety properties of the case study (see *inv2* and *inv10* in Section 4.9).

4 Detailed Model Description

This section describes in more detail the formal model, beginning from the most abstract Event-B machine. For each refinement, the state machine will be shown and in general only the important manual changes in the model generated from the state machine. The full generated code and the manual changes are available as a Rodin project. At each step the additional modeled functionality and its representation will be described. In particular the initialization event is not shown for the refined machines. If not mentioned explicitly, sets are initialized empty, integers with value 0 and Boolean variables with false.

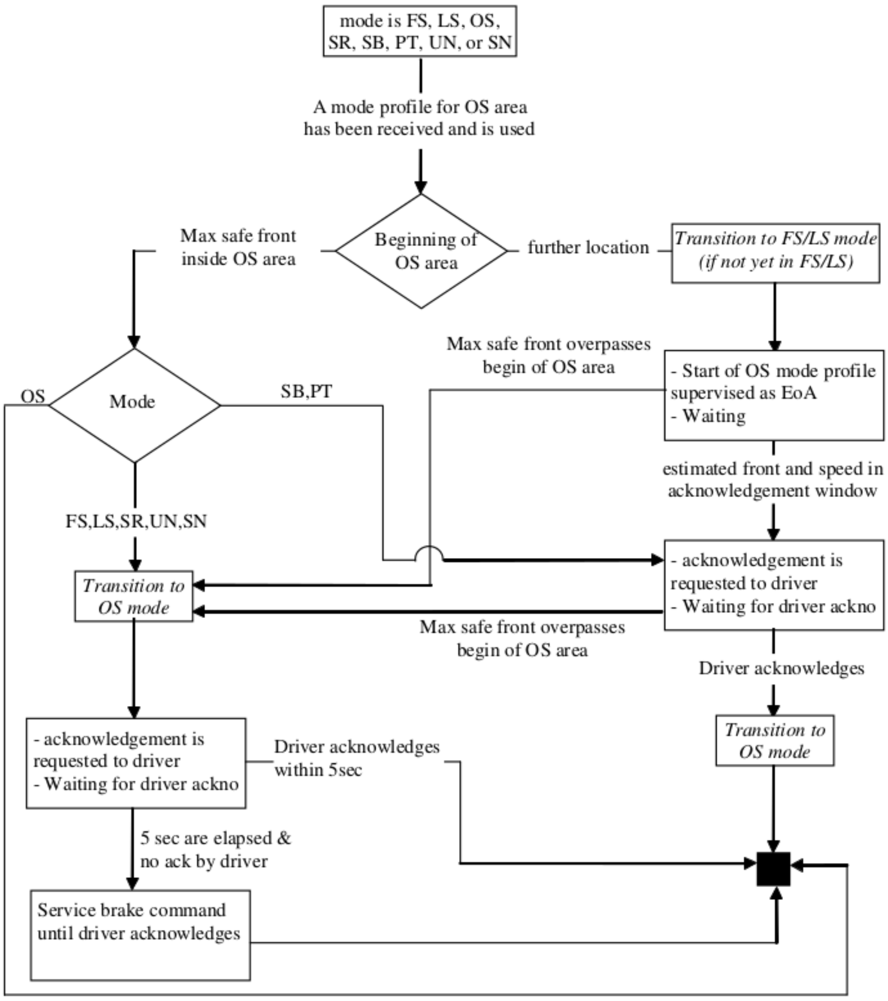


Figure 2. Flowchart for "On-Sight" Procedure [Eur12]

4.1 Machine 0 - Basic Flowchart

The first state machine *m0* (see Fig. 3) represents an abstract view of the flowchart describing the on-sight procedure which is shown in §5.9.7 of the SRS [Eur12] (see Fig. 2).

The flowchart is translated into a iUML state machine as follows: the initial state represents the initial situation of the procedure flowchart. The diamonds of the flowchart represent different cases and are therefore into transitions with different target states in the state chart. The nodes of the flowchart are combined for abstraction by combining nodes with multiple incoming flows (or an initial node) with direct successor nodes.

For example the state *ack_and_transition* can be reached from the initial state via the event *use_profile_OS_inside_area_mode_SB_PT* and corresponds to the two lower right nodes of the flowchart. This is justified, as the flow passes two diamonds in the flowchart, verifying that the i) max safe front of the train is inside the OS area and ii) the train mode is *BS* or *PT*. The complete model is automatically generated from this state machine. Note however, that in this abstraction level, there is no concrete notion of train modes, these appear in the first refinement.

The transitions *switch_to_OS_mode* signal the completion of the on-sight procedure, the internal switch to OS mode in the train happens elsewhere. The state *OS_mode* signals the final state.

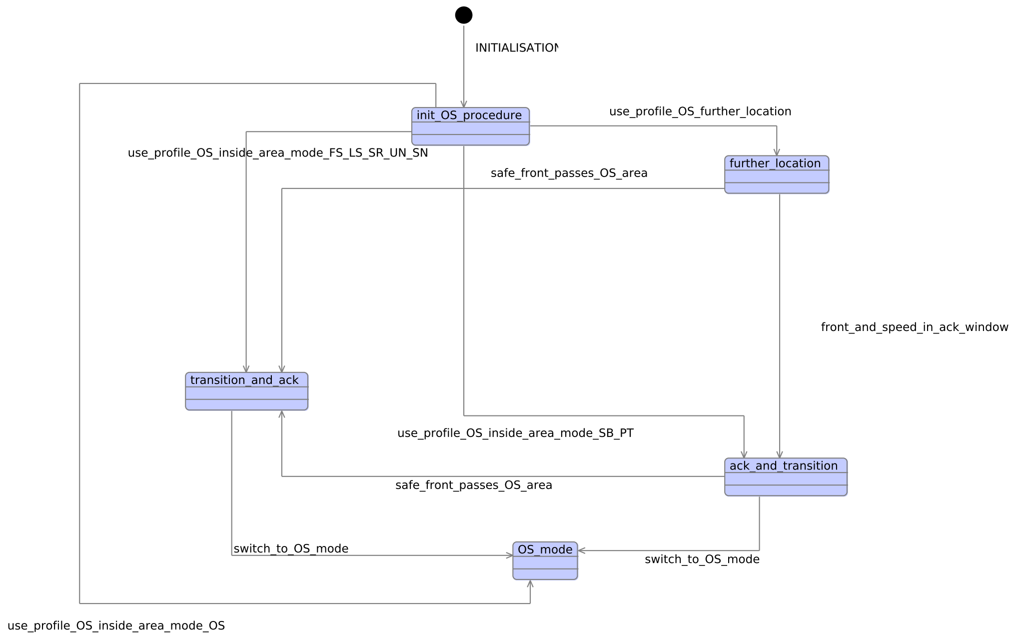


Figure 3. Basic Flowchart Representation

4.2 Context 0 - Train Modes

The first context *c0* specifies the possible modes of the train, these are of type *t_train_modes*. There is one Event-B constant for each possible mode. The constant *c_initial_mode* represents the initial mode of the train when the procedure on-sight is started. The constant *c_supervision_mode* is one mode from the supervision modes.

SETS

t_train_modes

CONSTANTS

c_FS full supervision
c_LS limited supervision
c_OS on sight
c_SR staff responsible
c_SB stand-by
c_PT post-trip
c_UN unfitted
c_SN national system
$c\_initial\_mode$
$c\_supervision\_mode$

AXIOMS

axm1 : $partition(t\_train\_modes, \{c\_FS\}, \{c\_LS\}, \{c\_OS\}, \{c\_SR\}, \{c\_SB\}, \{c\_PT\}, \{c\_UN\}, \{c\_SN\})$
axm2 : $c\_initial\_mode \in \{c\_FS, c\_OS, c\_PT\}$
axm3 : $c\_supervision\_mode \in \{c\_LS, c\_FS\}$

END

4.3 Machine 1 - Train Modes

The first machine refinement adds the variable *current_mode* which tracks the current mode of the train. This variable is initialized with the value of *c_initial_mode*.

The state of this variable is used to constrain the guards of the events that depend on the train modes, i.e., corresponding to those that lead from the "Mode" diamond in the flowchart (see Fig. 2). Its state is changed in the *transition_to_supervision_mode* event which assigns the value of *c_supervision_mode* or in the *transition_to_OS_mode* event which assigns the on-sight mode.

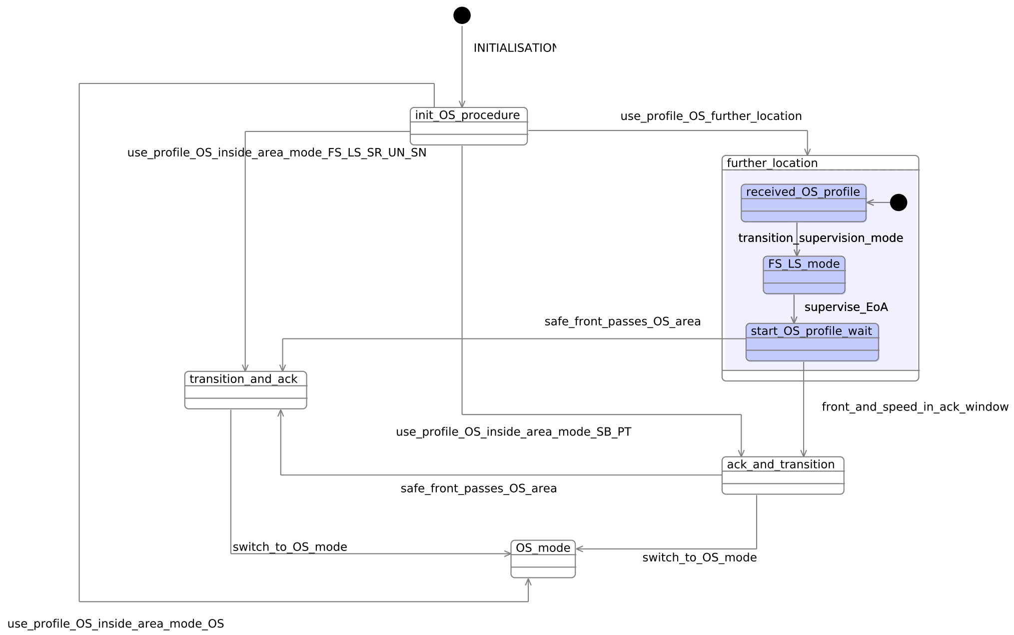


Figure 4. First Refinement with Train Modes

The refined state chart is shown in Fig. 4. The state *further_location* is refined to contain three sub-states and two events. This switches the train to supervision mode, and starts EoA supervision. The train stays in this state until either the maximal safe front passes the OS area or the estimated front and speed leave the acknowledge window. The exiting transitions are changed to originate from the *start_OS_profile_wait* state instead of its super-state. This is possible as the sub-states correctly refine the super-state.

MACHINE m1_train_modes
REFINES m0_basic_flowchart
SEES c0_train_mode
VARIABLES

    current_mode

INVARIANTS

    inv1 : $current\_mode \in t\_train\_modes$

EVENTS
Event *safe_front_passes_OS_area* $\widehat{=}$
extends *safe_front_passes_OS_area*

  when

    isin_ack_and_transition_or_isin_further_location : $ack\_and\_transition = TRUE \lor further\_location = TRUE$
    isin_start_OS_profile_wait : $start\_OS\_profile\_wait = TRUE$

  then

    enter_transition_and_ack : $transition\_and\_ack := TRUE$
    leave_ack_and_transition : $ack\_and\_transition := FALSE$
    leave_further_location : $further\_location := FALSE$
    leave_start_OS_profile_wait : $start\_OS\_profile\_wait := FALSE$

  end

Event *switch_to_OS_mode* $\widehat{=}$

extends *switch_to_OS_mode*

when

`isin_ack_and_transition_or_isin_transition_and_ack` : $ack\_and\_transition = TRUE \lor transition\_and\_ack = TRUE$

then

`leave_ack_and_transition` : $ack\_and\_transition := FALSE$
`enter_OS_mode` : $OS\_mode := TRUE$
`leave_transition_and_ack` : $transition\_and\_ack := FALSE$

end

Event *front_and_speed_in_ack_window* $\widehat{=}$
extends *front_and_speed_in_ack_window*

when

`isin_further_location` : $further\_location = TRUE$
`isin_start_OS_profile_wait` : $start\_OS\_profile\_wait = TRUE$

then

`enter_ack_and_transition` : $ack\_and\_transition := TRUE$
`leave_further_location` : $further\_location := FALSE$
`leave_start_OS_profile_wait` : $start\_OS\_profile\_wait := FALSE$

end

Event *use_profile_OS_further_location* $\widehat{=}$
extends *use_profile_OS_further_location*

when

`isin_init_OS_procedure` : $init\_OS\_procedure = TRUE$

then

`leave_init_OS_procedure` : $init\_OS\_procedure := FALSE$
`enter_further_location` : $further\_location := TRUE$
`enter_received_OS_profile` : $received\_OS\_profile := TRUE$

end

Event *use_profile_OS_inside_area_mode_OS* $\widehat{=}$
extends *use_profile_OS_inside_area_mode_OS*

when

`isin_init_OS_procedure` : $init\_OS\_procedure = TRUE$
`grd1` : $current\_mode = c\_OS$

then

`enter_OS_mode` : $OS\_mode := TRUE$
`leave_init_OS_procedure` : $init\_OS\_procedure := FALSE$

end

Event *use_profile_OS_inside_area_mode_SB_PT* $\widehat{=}$
extends *use_profile_OS_inside_area_mode_SB_PT*

when

`isin_init_OS_procedure` : $init\_OS\_procedure = TRUE$
`grd1` : $current\_mode \in \{c\_SB, c\_PT\}$

then

`enter_ack_and_transition` : $ack\_and\_transition := TRUE$
`leave_init_OS_procedure` : $init\_OS\_procedure := FALSE$

end

Event *use_profile_OS_inside_area_mode_FS_LS_SR_UN_SN* $\widehat{=}$
extends *use_profile_OS_inside_area_mode_FS_LS_SR_UN_SN*

when

`isin_init_OS_procedure` : $init\_OS\_procedure = TRUE$
`grd1` : $current\_mode \in \{c\_FS, c\_LS, c\_SR, c\_UN, c\_SN\}$

then

`leave_init_OS_procedure` : $init\_OS\_procedure := FALSE$
`enter_transition_and_ack` : $transition\_and\_ack := TRUE$

end

Event *transition_supervision_mode* $\widehat{=}$

when

`isin_received_OS_profile` : $received\_OS\_profile = TRUE$

then

`leave_received_OS_profile` : $received\_OS\_profile := FALSE$
`act1` : $current\_mode := c\_supervision\_mode$
`enter_FS_LS_mode` : $FS\_LS\_mode := TRUE$

end

Event $transition\_to\_OS\_mode \widehat{=}$

begin

`act1` : $current\_mode := c\_OS$

end

END

4.4 Context 1 - Mode Profiles

This context extension introduces the type *t_mode_profile* for mode profiles, *t_train_fronts* for train fronts (e.g., max safe front, estimated front), *t_speed* for train speed and *t_locations* for on track locations.

The context also defines several functions, notably one which signals whether a mode profile specifies an OS area, one which signals whether a given train front overpasses the OS area for a specific mode profile, one that signals whether a train front and train speed are in the acknowledge window for a specific mode profile, one that signals whether a given train front is in the OS area of a given mode profile and finally a function that returns the EoA from a given profile.

CONTEXT c1_mode_profile
EXTENDS c0_train_mode
SETS

$t\_mode\_profile$
t_train_fronts
t_speed
$t\_locations$

CONSTANTS

$f\_mode\_profile\_OS\_mode$ indicates whether mode profile demands OS mode
$f\_safe\_train\_front\_overpasses$
$f\_estimated\_train\_front\_speed\_in\_window$
$c\_profile0$
$f\_safe\_front\_in\_OS\_area$
$f\_EoA\_from\_profile$
c_loc0
c_front0

AXIOMS

`axm1` : $f\_mode\_profile\_OS\_mode \in t\_mode\_profile \to BOOL$
`axm2` : $f\_safe\_train\_front\_overpasses \in t\_train\_fronts \times t\_mode\_profile \to BOOL$

train front overpasses begin OS area

`axm3` : $f\_estimated\_train\_front\_speed\_in\_window \in t\_train\_fronts \times t\_mode\_profile \times t\_speed \to BOOL$

est. train front and speed in ack window

`axm4` : $c\_profile0 \in t\_mode\_profile$
`axm5` : $f\_safe\_front\_in\_OS\_area \in t\_train\_fronts \times t\_mode\_profile \to BOOL$
`axm6` : $f\_EoA\_from\_profile \in t\_mode\_profile \to t\_locations$

axm7 : $c\_loc0 \in t\_locations$
axm10 : $c\_front0 \in t\_train\_fronts$
END

4.5 Machine 2 - Mode Profiles

The second refinement of the machine introduces the notion of mode profiles, train fronts (max safe and estimated) and the end of authority (EoA) location into the model. These are represented by the variables *EoA_loc*, *mode_profile_OS*, *safe_train_front* and *estimated_train_front*.

The train fronts can be changed by the events *update_estimated_front* and *update_safe_front*. The current values of the fronts, the current mode profile and its corresponding EoA are used as parameters for the Boolean functions that guard the events, e.g., for the event *safe_front_passes_OS_area* or for the event *front_and_speed_in_ack_window*.

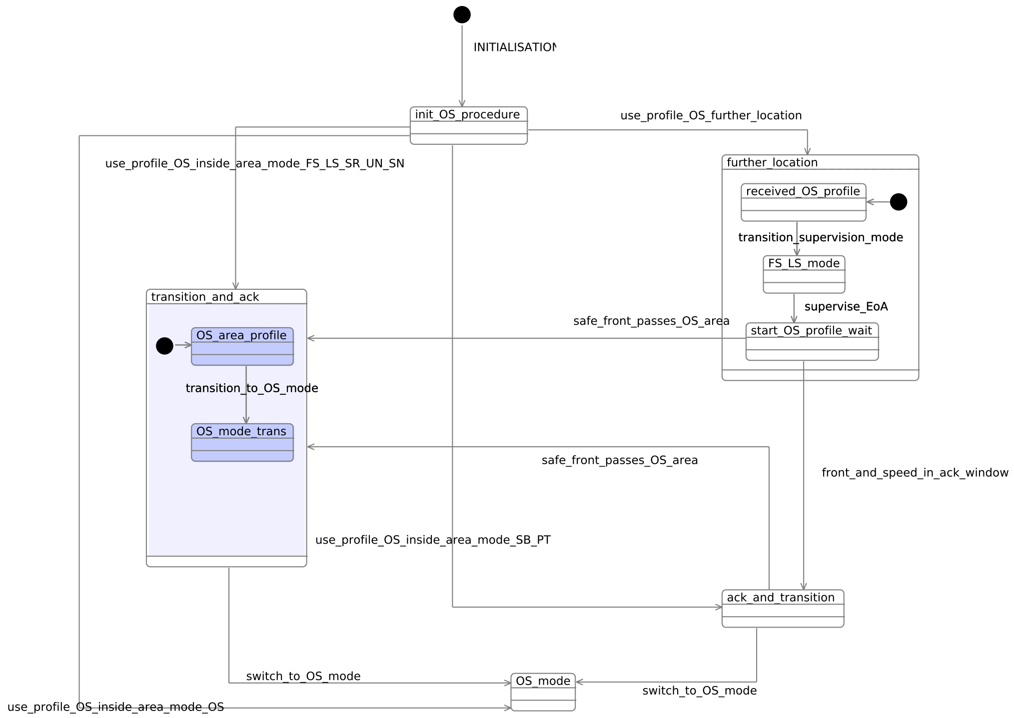


Figure 5. Second Refinement

The second refinement of the state machine is shown in Fig. 5. Here the state *transition_and_ack* is refined with two sub-states. The transition between the two new sub-states is *transition_to_OS_mode* which sets the current train mode to on-sight.

MACHINE m2_mode_profile
REFINES m1_train_modes
SEES c1_mode_profile
VARIABLES

EoA_loc
$mode\_profile\_OS$
$safe\_train\_front$
$estimated\_train\_front$

INVARIANTS

inv1 : $EoA\_loc \in t\_locations$

inv2 : $mode\_profile\_OS \in t\_mode\_profile$
inv3 : $safe\_train\_front \in t\_train\_fronts$
inv4 : $estimated\_train\_front \in t\_train\_fronts$

EVENTS

Event *safe_front_passes_OS_area* $\widehat{=}$
extends *safe_front_passes_OS_area*

where

isin_ack_and_transition_or_isin_further_location : $ack\_and\_transition = TRUE \lor further\_location = TRUE$
isin_start_OS_profile_wait : $start\_OS\_profile\_wait = TRUE$
grd1 : $f\_safe\_train\_front\_overpasses(safe\_train\_front \mapsto mode\_profile\_OS) = TRUE$
grd2 : $l\_safe\_train\_front \in t\_train\_fronts$

then

enter_transition_and_ack : $transition\_and\_ack := TRUE$
leave_ack_and_transition : $ack\_and\_transition := FALSE$
leave_further_location : $further\_location := FALSE$
leave_start_OS_profile_wait : $start\_OS\_profile\_wait := FALSE$
enter_OS_area_profile : $OS\_area\_profile := TRUE$

end

Event *front_and_speed_in_ack_window* $\widehat{=}$
extends *front_and_speed_in_ack_window*

any

l_train_speed

where

isin_further_location : $further\_location = TRUE$
isin_start_OS_profile_wait : $start\_OS\_profile\_wait = TRUE$
grd3 : $l\_train\_speed \in t\_speed$
grd1 : $f\_estimated\_train\_front\_speed\_in\_window(estimated\_train\_front \mapsto mode\_profile\_OS \mapsto l\_train\_speed) = TRUE$

then

enter_ack_and_transition : $ack\_and\_transition := TRUE$
leave_further_location : $further\_location := FALSE$
leave_start_OS_profile_wait : $start\_OS\_profile\_wait := FALSE$

end

Event *update_estimated_front* $\widehat{=}$

any

l_front

where

grd1 : $l\_front \in t\_train\_fronts$

then

act1 : $estimated\_train\_front := l\_front$

end

Event *update_safe_front* $\widehat{=}$

any

l_front

where

grd1 : $l\_front \in t\_train\_fronts$

then

act1 : $safe\_train\_front := l\_front$

end

END

4.6 Machine 3 - Driver Acknowledge

The third machine refinement introduces the driver acknowledgment. In two cases, the driver is asked to acknowledge. This is modeled by additional Boolean variables, two for acknowledging OS mode and two for acknowledging the service brake. Each time, one variable signals that the driver has been informed that he has to acknowledge, e.g., for the service brake this is the *currently_asking_driver_brake_ack* variable, and to signal the completed acknowledge there is the *driver_responded_brake_ack* variable. There is also the Boolean variable *service_brake* which signals the active service brake of the train.

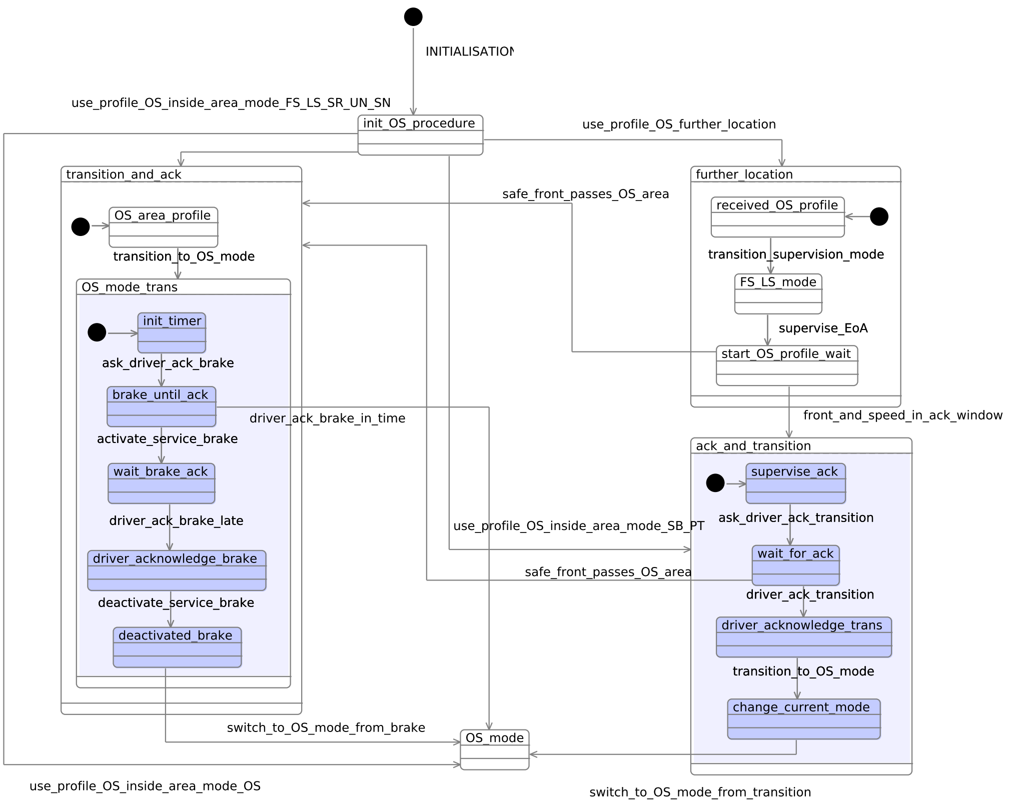


Figure 6. Third Refinement with Driver Acknowledge

The third refinement of the state machine is shown in Fig. 6. Here the two states *ack_and_transition* and *OS_mode_trans* are refined.

For *ack_and_transition* there are four new sub-states defined. There is first an event to ask the driver to acknowledge the switch to on-sight mode, then the OBU waits for the driver acknowledgment. If the max safe train front passes the OS area without driver acknowledge, then the *transition_and_ack* state is entered, else the train mode is switched to on-sight and the final state is entered. Here the outgoing transitions from the abstract *ack_and_transition* state are change to originate from *wait_for_ack* or *change_current_mode* respectively. Again this is possible as the sub-states correctly refine the abstract behavior of the super state.

The abstract *OS_mode_trans* state is refined by five sub-states. First the driver is asked to acknowledge the imminent service brake. If he does so in time, then the procedure finishes and the final state is entered, else the service brake is activated and stays activated until the driver acknowledges and the final state is entered.

At this refinement stage, it is possible to prove that whenever the final state is reached, the mode of the train is on-sight (*inv6*).

MACHINE m3_driver_ack
REFINES m2_mode_profile
SEES c1_mode_profile
VARIABLES

$currently\_asking\_driver\_OS\_ack$
$driver\_responded\_OS\_ack$
$service\_brake\_active$
$currently\_asking\_driver\_brake\_ack$
$driver\_responded\_brake\_ack$

INVARIANTS

inv6 : $OS\_mode = TRUE \Rightarrow current\_mode = c\_OS$

EVENTS

Event $ask\_driver\_ack\_brake \mathrel{\widehat{=}}$

when

isin_init_timer : $init\_timer = TRUE$
grd1 : $currently\_asking\_driver\_brake\_ack = FALSE$

then

act1 : $currently\_asking\_driver\_brake\_ack := TRUE$
enter_brake_until_ack : $brake\_until\_ack := TRUE$
act2 : $driver\_responded\_brake\_ack := FALSE$
leave_init_timer : $init\_timer := FALSE$

end

Event $ask\_driver\_ack\_transition \mathrel{\widehat{=}}$

when

isin_supervise_ack : $supervise\_ack = TRUE$
grd1 : $currently\_asking\_driver\_OS\_ack = FALSE$

then

act1 : $currently\_asking\_driver\_OS\_ack := TRUE$
leave_supervise_ack : $supervise\_ack := FALSE$
enter_wait_for_ack : $wait\_for\_ack := TRUE$
act2 : $driver\_responded\_OS\_ack := FALSE$

end

Event $driver\_ack\_brake\_in\_time \mathrel{\widehat{=}}$
extends $switch\_to\_OS\_mode$

when

isin_ack_and_transition_or_isin_transition_and_ack : $ack\_and\_transition = TRUE \lor transition\_and\_ack = TRUE$
isin_brake_until_ack : $brake\_until\_ack = TRUE$
grd1 : $currently\_asking\_driver\_brake\_ack = TRUE$

then

leave_ack_and_transition : $ack\_and\_transition := FALSE$
enter_OS_mode : $OS\_mode := TRUE$
leave_transition_and_ack : $transition\_and\_ack := FALSE$
leave_OS_mode_trans : $OS\_mode\_trans := FALSE$
leave_OS_area_profile : $OS\_area\_profile := FALSE$
act2 : $driver\_responded\_brake\_ack := TRUE$
leave_brake_until_ack : $brake\_until\_ack := FALSE$
act1 : $currently\_asking\_driver\_brake\_ack := FALSE$

end

Event $driver\_ack\_brake\_late \mathrel{\widehat{=}}$

when

grd1 : $currently\_asking\_driver\_brake\_ack = TRUE$
isin_wait_brake_ack : $wait\_brake\_ack = TRUE$

```
    then
        act1 : currently_asking_driver_brake_ack := FALSE
        enter_driver_acknowledge_brake : driver_acknowledge_brake := TRUE
        act2 : driver_responded_brake_ack := TRUE
        leave_wait_brake_ack : wait_brake_ack := FALSE
    end
Event driver_ack_transition ≙
    when
        grd1 : currently_asking_driver_OS_ack = TRUE
        isin_wait_for_ack : wait_for_ack = TRUE
    then
        act2 : driver_responded_OS_ack := TRUE
        enter_driver_acknowledge_trans : driver_acknowledge_trans := TRUE
        leave_wait_for_ack : wait_for_ack := FALSE
        act1 : currently_asking_driver_OS_ack := FALSE
    end
Event activate_service_brake ≙
    when
        grd1 : service_brake_active = FALSE
        isin_brake_until_ack : brake_until_ack = TRUE
    then
        act1 : service_brake_active := TRUE
        leave_brake_until_ack : brake_until_ack := FALSE
        enter_wait_brake_ack : wait_brake_ack := TRUE
    end
Event deactivate_service_brake ≙
    when
        grd1 : service_brake_active = TRUE
        isin_driver_acknowledge_brake : driver_acknowledge_brake = TRUE
    then
        enter_deactivated_brake : deactivated_brake := TRUE
        act2 : service_brake_active := FALSE
        act3 : driver_responded_brake_ack := FALSE
        leave_driver_acknowledge_brake : driver_acknowledge_brake := FALSE
    end
END
```

4.7 Machine 4 - Timeout

The fourth machine refinement introduces the timer and the timeout event for the acknowledge window for the driver. This refinement is done completely in the textual model, therefore the state machine as shown in Fig. 7 is left unchanged and there is no colored highlighting.

The timeout is represented by the Boolean variable *timer_expired* which is modified by the *expire_timer* event. The variable is used in the guards of *driver_ack_brake_in_time*, *driver_ack_brake_late* and *activate_service_brake*.

At this refinement stage, it is possible to prove that when the timer has expired and the train has transitioned to OS mode then the transition *driver_ack_brake_in_time* cannot be enabled (*inv2*), and in the state *brake_until_ack* the transition *activate_service_brake* is enabled (*inv4*).

```
MACHINE m4_timeout
REFINES m3_driver_ack
SEES c1_mode_profile
```

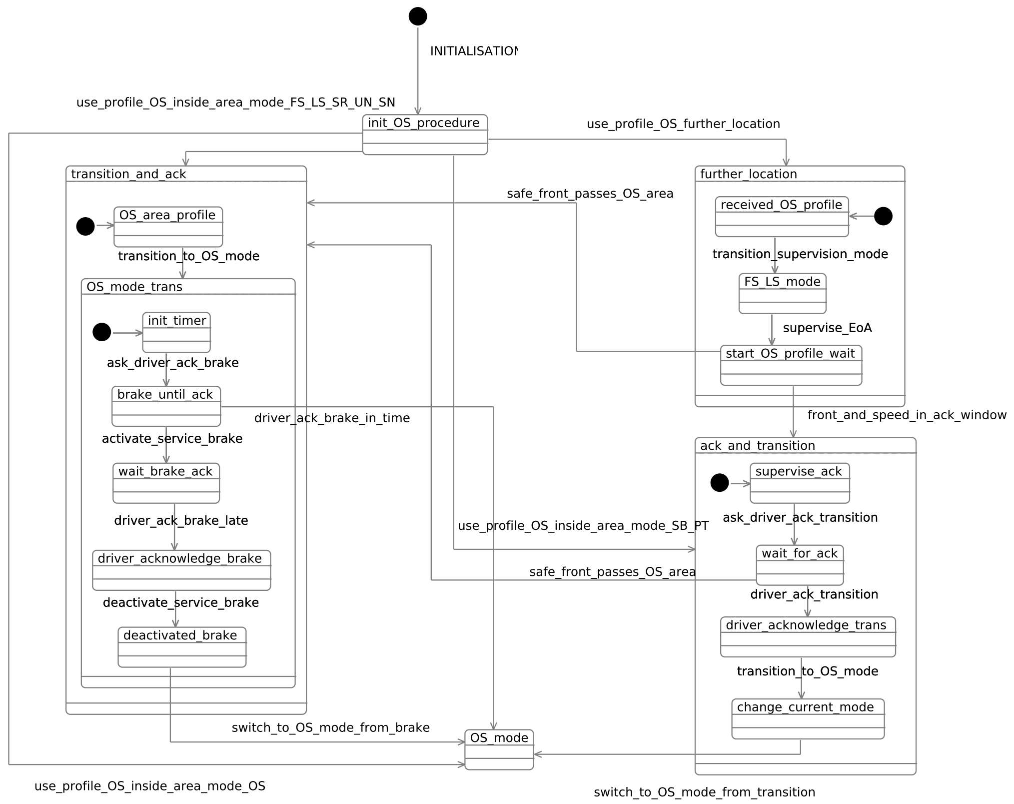


Figure 7. Fourth Refinement State Machine

VARIABLES

timer_expired

INVARIANTS

inv2 : $timer\_expired = TRUE \wedge OS\_mode\_trans = TRUE \Rightarrow$
$\neg((ack\_and\_transition = TRUE \vee transition\_and\_ack = TRUE) \wedge$
$brake\_until\_ack = TRUE \wedge$
$currently\_asking\_driver\_brake\_ack = TRUE \wedge$
$timer\_expired = FALSE)$
PROPERTY_5.9_02_01, timer expired and in OS_mode_trans state disables other transitions

inv4 : $timer\_expired = TRUE \wedge brake\_until\_ack = TRUE \Rightarrow$
$(service\_brake\_active = FALSE \wedge$
$brake\_until\_ack = TRUE \wedge$
$timer\_expired = TRUE)$
PROPERTY_5.9_02_02, transition to activate brake is enabled

EVENTS

Event *driver_ack_brake_in_time* $\widehat{=}$
extends *driver_ack_brake_in_time*

when

isin_ack_and_transition_or_isin_transition_and_ack : $ack\_and\_transition = TRUE \vee transition\_and\_ack = TRUE$
isin_brake_until_ack : $brake\_until\_ack = TRUE$
grd1 : $currently\_asking\_driver\_brake\_ack = TRUE$
grd2 : $timer\_expired = FALSE$

then

leave_ack_and_transition : $ack\_and\_transition := FALSE$
enter_OS_mode : $OS\_mode := TRUE$
leave_transition_and_ack : $transition\_and\_ack := FALSE$
leave_OS_mode_trans : $OS\_mode\_trans := FALSE$
leave_OS_area_profile : $OS\_area\_profile := FALSE$
act2 : $driver\_responded\_brake\_ack := TRUE$

`leave_brake_until_ack` : $brake\_until\_ack := FALSE$
`act1` : $currently\_asking\_driver\_brake\_ack := FALSE$
end

Event *driver_ack_brake_late* $\widehat{=}$
extends *driver_ack_brake_late*
when
`grd1` : $currently\_asking\_driver\_brake\_ack = TRUE$
`isin_wait_brake_ack` : $wait\_brake\_ack = TRUE$
`grd2` : $timer\_expired = TRUE$
then
`act1` : $currently\_asking\_driver\_brake\_ack := FALSE$
`enter_driver_acknowledge_brake` : $driver\_acknowledge\_brake := TRUE$
`act2` : $driver\_responded\_brake\_ack := TRUE$
`leave_wait_brake_ack` : $wait\_brake\_ack := FALSE$
end

Event *activate_service_brake* $\widehat{=}$
extends *activate_service_brake*
when
`grd1` : $service\_brake\_active = FALSE$
`isin_brake_until_ack` : $brake\_until\_ack = TRUE$
`grd2` : $timer\_expired = TRUE$
then
`act1` : $service\_brake\_active := TRUE$
`leave_brake_until_ack` : $brake\_until\_ack := FALSE$
`enter_wait_brake_ack` : $wait\_brake\_ack := TRUE$
end

Event *expire_timer* $\widehat{=}$
when
`grd1` : $timer\_expired = FALSE$
then
`act1` : $timer\_expired := TRUE$
end

END

4.8 Context 2 - Speed Limits

The third context extension adds the notion of the on-sight speed limit which is represented as the constant *c_OS_speed_limit* of type *t_speed*. The context also adds the function *f_speed_exceeds* which compares two speed values and returns *TRUE* if the first argument exceeds the second.

CONTEXT c2_speed_limit
EXTENDS c1_mode_profile
CONSTANTS
$c\_OS\_speed\_limit$
$f\_speed\_exceeds$
AXIOMS
`axm1` : $c\_OS\_speed\_limit \in t\_speed$
constant OS speed limit
`axm2` : $f\_speed\_exceeds \in t\_speed \times t\_speed \rightarrow BOOL$
f_speed_exceeds (x, y) is TRUE if speed x exceeds speed y, else FALSE

END

4.9 Machine 5 - Speed Supervision

The fifth refinement of the machine adds the notion of the current speed of the train to the model. This variable can be updated by two events *update_train_speed_brake* which can only be executed when the brake is active and only accepts speed as argument which does not exceed the current speed and *update_train_speed_no_brake* which requires that the brake is not active and accepts any speed as new value.

Once active, the service brake can now only be activated if the current speed of the train does not exceed the OS speed limit.

MACHINE m5_speed_supervision
REFINES m4_timeout
SEES c2_speed_limit
VARIABLES

$current\_speed$

INVARIANTS

inv2 : $(driver\_acknowledge\_brake = TRUE \wedge f\_speed\_exceeds(current\_speed \mapsto c\_OS\_speed\_limit) = TRUE \wedge driver\_responded\_brake\_ack = TRUE) \Rightarrow service\_brake\_active = TRUE$
PROPERTY_5.9_03 driver acknowledge does not deactivate service brake if train speed exceeds OS speed limit

inv10 : $transition\_and\_ack = TRUE \Rightarrow ((f\_speed\_exceeds(current\_speed \mapsto c\_OS\_speed\_limit) = TRUE \wedge current\_mode = c\_OS) \Rightarrow service\_brake\_active = TRUE)$
PROPERTY_5.9_01, brake is activated if mode is OS and current speed exceeds limit

EVENTS

Event *deactivate_service_brake* $\widehat{=}$
extends *deactivate_service_brake*

when

grd1 : $service\_brake\_active = TRUE$
isin_driver_acknowledge_brake : $driver\_acknowledge\_brake = TRUE$
grd2 : $f\_speed\_exceeds(current\_speed \mapsto c\_OS\_speed\_limit) = FALSE$

then

enter_deactivated_brake : $deactivated\_brake := TRUE$
act2 : $service\_brake\_active := FALSE$
act3 : $driver\_responded\_brake\_ack := FALSE$
leave_driver_acknowledge_brake : $driver\_acknowledge\_brake := FALSE$

end

Event *update_train_speed_brake* $\widehat{=}$
if brake is on new speed cannot exceed current speed

any

l_speed

where

grd1 : $l\_speed \in t\_speed$
grd2 : $service\_brake\_active = TRUE$
grd3 : $f\_speed\_exceeds(l\_speed \mapsto current\_speed) = FALSE$
grd4 : $init\_OS\_procedure = TRUE \vee OS\_mode = TRUE$

then

act1 : $current\_speed := l\_speed$

end

Event *update_train_speed_no_brake* $\widehat{=}$

any

l_speed

where

grd1 : $service\_brake\_active = FALSE$
grd2 : $l\_speed \in t\_speed$
grd3 : $driver\_acknowledge\_brake = FALSE$
grd4 : $init\_OS\_procedure = TRUE \lor OS\_mode = TRUE$

then

act1 : $current\_speed := l\_speed$

end

END

At this refinement stage, it is possible to prove the two remaining safety properties from the D2.5 document: If the train is in the *driver_responded_brake_ack* state, then even if the driver has acknowledged the service brake activation, the service brake will not be deactivated if the *current_speed* exceeds the OS speed limit. And, if the train is in OS mode and the *current_speed* exceeds the speed limit, then the service brake is active.

References

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[Eur12] European Railway Agency (ERA). System Requirements Specification - ETCS Subset 026. http://www.era.europa.eu/Document-Register/Documents/Index00426.zip, 2012.

[Jas12] Michael Jastram, editor. *Rodin User's Handbook*. DEPLOY Project, 2012.

[SBS09] Mar Yah Said, Michael Butler, and Colin Snook. Language and tool support for class and state machine refinement in uml-b. In *FM 2009: Formal Methods*, pages 579–595. Springer, 2009.