System Validation
Final Report

Gonzalo Moro Pérez (0926033)
Andrés Gunnarsson (0927231)
Yiting Xu (0925537)
Hanrong Zhou (0925762)

October 24th 2014

Contents

1 Introduction

This document describes a system called Automatic Train Protection (ATP) system. The purpose of the ATP system is to prevent train accidents by detecting the speed limit of the track, show it to the conductor and automatically stop the train if he does not react within a certain time frame. The current speed limit is detected in the form of a pulsed electric signal on the tracks. The frequency of the signal determines the speed limit according to a predefined table. The system also detects if there is a red light ahead and then stops the train. The train can then not continue until a green light is detected.

The ATP system can be turned off and on with certain external signals. If the system is turned off, it should not show the conductor any speed limit, should not warn him of exceeding any limit and in particular should not stop the train under any circumstance. On the other hand, if the system is turned on, it should detect the speed limit and traffic lights, warn the conductor if he is going too fast and automatically brake if he does not react.

In the following chapters, the requirements of the system will be set. The external actions of the system will be determined and the requirements will be expressed in terms of these actions. An overview of the different components in the system will be presented, as well as the internal actions between them. Finally, a model of the system will be developed and the requirements, which will be translated into modal formulas, will be checked.

2 List of requirements

The list of general requirements specifying the behaviour of the system is:

1. If the ATP is not turned on, it will not control the bell or the brakes of the train and it will not show any speed limit.

2. The ATP will order to turn off the bell if the speed of the train is under or at the speed limit. Otherwise, it will order to turn on the bell.

3. When the ATP detects a frequency, then the corresponding speed limit, or speed limit zero must be applied.

4. The ATP will not show the speed limit as zero unless it has detected a red light before, and whenever a red traffic light is detected, the speed limit of the train will be set to zero.

5. If the speed of the train does not get under the speed limit within a time limit after the bell started ringing, then the brakes must be applied.

6. Once the brakes have been applied, they will not be released until the train is at a complete standstill and then only after the ATP is reset.

7. If the ATP is reset and the train is at a complete standstill, then the bell must stop ringing and the brakes must be released.

8. Whenever the train is at a standstill, it is possible to reset.

9. At any time, it is possible within a finite number of steps to receive any frequency from the tracks and receive a red or green light signal. Also, as long as the system is on, it is possible to read the current speed or a stop signal. At any time, as long as the system is off, it is possible receive a start signal.

10. The system will not deadlock.

3 List of external actions

The following table shows all the external actions together with a short description:

No.	Name (data type)	Description	Direction
1	`startATP`	ATP system is started	IN
2	`stopATP`	ATP system is stoped	IN
3	`resetATP`	ATP system is reset after emergency stop	IN
4	`getSpeed(Speed)`	ATP system reads train speed	IN
5	`getPulseFrequency(Freq)`	ATP system reads signal from antenna	IN
6	`lightRed`	ATP system receives notification of red light	IN
7	`lightGreen`	ATP system receives notification of green light	IN
8	`showSpeedLimit(Speed)`	ATP system sends the current speed limit	OUT
9	`bellOn`	ATP system instructs bell to ring	OUT
10	`bellOff`	ATP system instructs bell to stop ringing	OUT
11	`brakesOn`	ATP system instructs brakes to be applied	OUT
12	`brakesOff`	ATP system instructs brakes to be released	OUT

Table 1: List of external actions

The data types *Speed* and *Freq* will be used throughout the whole report. They are defined as follows:

$Speed = struct\ s0 \mid s40 \mid s60 \mid s80 \mid s100 \mid s140;$
$Freq = struct\ f0 \mid f40 \mid f60 \mid f80 \mid f100 \mid f140 \mid specF;$

Besides, the function *freq2speed*, mapping from one data type to the other, is defined as follows:

$freq2speed\colon\ Freq \rightarrow Speed;$

The external actions together with their direction, in or out, are shown in Figure 1:

Figure 1: An overview of the external actions and their directions

4 Requirements expressed in terms of actions

In this section, the requirements listed in section 3 will be translated in terms of the actions described in section 4. Note that the following variables will be used:

f,f1,f2,fOld: Freq

v,v1,v2: Speed

1. (a) Between a `stopATP` action and a `startATP` action, the actions `bellOn` , `bellOff` , `brakesOn` , `brakesOff` and `showSpeedLimit(v)` should not take place.

2. (a) Whenever there is a `showSpeedLimit(v1)` action followed by some actions which are not `stopATP` and then followed by an action `getSpeed(v2)` ; if and only if $v2 > v1$, then after a finite number of steps a `bellOn` action will take place except that a `showSpeedLimit(v1)` can happen infinitely often in the meantime.

   (b) Whenever there is a `showSpeedLimit(v1)` action followed by some actions which are not `stopATP` and then followed by an action `getSpeed(v2)` ; if and only if $v2 \leq v1$, then after a finite number of steps a `bellOff` action will take place except that a `showSpeedLimit(v1)` can happen infinitely often in the meantime.

   (c) Whenever there is a `getSpeed(v1)` action followed by some actions which are not `stopATP` and then followed by an action `showSpeedLimit(v2)` ; if and only if $v2 \geq v1$, then after a finite number of steps a `bellOff` action will take place except that a `showSpeedLimit(v2)` can happen infinitely often in the meantime.

   (d) Whenever there is a `getSpeed(v1)` action followed by some actions which are not `stopATP` and then followed by an action `showSpeedLimit(v2)` ; if and only if $v2 < v1$, then after a finite number of steps a `bellOn` action will take place except that a `showSpeedLimit(v2)` can happen infinitely often in the meantime.

3. (a) Whenever an action `getPulseFrequency(fOld)` takes place, followed by some actions which are not `stopATP` , followed by an action `getPulseFrequency(f)` , then after a finite number of steps,

an action `showSpeedLimit(freq2speed(f))` or `showSpeedLimit(s0)` will happen, except that a `showSpeedLimit(freq2speed(fOld))` action can happen infinitely often in the meantime.

4. (a) No action `showSpeedLimit(s0)` will happen if there hasn't been a previous `lightRed` action.

   (b) Whenever an action `lightRed` happens, then after a finite number of steps an action `showSpeedLimit(s0)` or an action `stopATP` will follow, except that an action `showSpeedLimit(v)` can happen infinitely often in the meantime.

5. (a) Whenever an action `showSpeedLimit(v1)` , followed by some actions which are not `stopATP` , followed by `getSpeed(v2)` followed by some actions which are not `stopATP` , followed by an action `timeOut` , then if and only if $v2 > v1$, an action `brakesOn` will eventually happen, except that an action `showSpeedLimit(v1)` can happen infinitely often in the meantime.

   (b) Whenever an action `getSpeed(v1)` , followed by some actions which are not `stopATP` , followed by `showSpeedLimit(v2)` , followed by some actions which are not `stopATP` , followed by an action `timeOut` , then if and only if $v2 < v1$, an action `brakesOn` will eventually happen except, that an action `showSpeedLimit(v2)` can happen infinitely often in the meantime.

6. (a) Whenever there is a `brakesOn` action, a `brakesOff` action cannot happen unless there is a `getSpeed(s0)` action.

   (b) Whenever there is a `brakesOn` action, a `brakesOff` action cannot happen unless there is a `resetATP` action.

7. (a) Whenever there is at least one `getSpeed(s0)` action, followed by no `getSpeed(v)` actions, followed by a `resetATP` action, then after a finite number of steps, a `brakesOff` action will happen.

   (b) Whenever there is at least one `getSpeed(s0)` action, followed by no `getSpeed(v)` actions, followed by a `resetATP` action, then after a finite number of steps, a `bellOff` action will happen.

8. (a) Whenever an action `getSpeed(s0)` occurs, then if no other `getSpeed(v)` action happens, there is a possibility for a `resetATP` action to happen

9. (a) There is always a finite sequence of actions after which the action `getPulseFrequency(f)` is possible.

   (b) There is always a finite sequence of actions after which the action `lightRed` is possible.

(c) There is always a finite sequence of actions after which the action `lightGreen` is possible.

(d) After a `startATP` action, there is a sequence of no `stopATP` actions after which the action`getSpeed(v)` is possible.

(e) After a `startATP` action, there is a sequence of no `stopATP` actions after which the action`stopATP` is possible.

10. (a) There is no deadlock

5 Architecture

The ATP consists of three parallel components, namely the ATP++1, the ATP++2 and the GATP. These components communicate with each other and share relevant information. They are organized in such a way that the communication between the three components of the system is very simple. They have the following tasks:

The ATP++1 is the component in charge of reading the signal from the antenna. It gets the frequency of the signal, looks up the corresponding speed limit and sends it to GATP.

The ATP++2 is the component in charge of detecting the signal from the beacon. Thus, it knows if the traffic light is red or green. If the traffic light is red, it will tell the GATP to override a zero km/h speed limit that will finish when the green light is detected.

The GATP is the component in charge of processing the information from the ATP++1 and from the ATP++2 and sending the external actions. It will calculate whether the train is driving at a two high speed and it will react accordingly.

Figure 2 in the following section shows the different components of the system.

6 List of internal actions

The following table shows all the internal actions together with a short description and the both the source and destination component:

No.	Name (data type)	Description	Source	Destination
1	`transmitSpeedLimit(Speed)`	The current speed limit according to track frequency	ATP++1	GATP
2	`transmitRed`	Zero km/h override because of red Light	ATP++2	GATP
3	`transmitGreen`	Stop override because of green Light	ATP++2	GATP

Table 2: List of external actions

Figure 2 shows the internal architecture of the system and the possible communications between the components.

Figure 2: An overview of the internal architecture and communications

Note that the numbers in Figure 2 refer to the number of the internal action in table 2.

7 Model of the system

sort

Mode = struct ON?is_on|OFF?is_off; *%% Two operating modes*

Light = struct GREEN?is_green|RED?is_red; *%% The light can be green or red*

Bellstate = struct SOUND?is_sounded|SILENCE?is_silenced; *%% The bell can be on or off*

Brakestate = struct LOCKED?is_applied|FREE?is_free; *%% The brakes can be applied or released*

Step = struct t0|t1|t2|t3|t4|t5|t6|t7; *%% The steps used in the GATP*

Speed = struct s0|s40|s60|s80|s100|s140; *%% Speed can be either 0, 40, 60, 80, 100 or 140*

Freq = struct f0|f40|f60|f80|f100|f140|specF; *%% Frequency can be 40, 60, 80, 100, 140 or the special frequency*

map
freq2speed: Freq − > Speed; *%% Function that transforms Freq to Speed*
vel: Speed − > Nat; *%% Function that transforms Speed to an actual natural number*
next: Step − > Step; *%% Function that has an input of type step and an output of type step*

eqn

freq2speed(f0) = s40; *%% Define the mapping freq2speed*
freq2speed(f40) = s40;
freq2speed(f60) = s60;
freq2speed(f80) = s80;
freq2speed(f100) = s100;
freq2speed(f140) = s140;
freq2speed(specF) = s140;
vel(s0) = 0; *%% Define the mapping vel*
vel(s40) = 40;
vel(s60) = 60;
vel(s80) = 80;
vel(s100) = 100;

```
vel(s140) = 140;
next(t0) = t1; %% Define the mapping step
next(t1) = t2;
next(t2) = t3;
next(t3) = t4;
next(t4) = t5;
next(t5) = t6;
next(t6) = t7;
next(t7) = t0;

act %% The different actions are defined

%% Actions of the ATP1

sendSpeedLimit : Speed;
getPulseFrequency : Freq;

%% Actions of the ATP2

lightGreen, sendGreen, lightRed, sendRed;

%% Ations of the GATP

startATP,stopATP,requestReset,timeOut;
receiveGreen, receiveRed;
receiveSpeedLimit, getSpeed : Speed;
bellOn,bellOff,brakesOn,brakesOff, resetATP;
showSpeedLimit : Speed;
overSpeed, noResponse;

%% Multi actions
transmitGreen, transmitRed;
transmitSpeedLimit : Speed;

proc

%% The ATP++1 component is responsible for receiving the pulse frequency
from the track and then sending the appropriate speed limit to GATP.
ATP1 =
%% We have to be prepared to receive any frequency
sum f:Freq . (getPulseFrequency(f) . sendSpeedLimit(freq2speed(f))) .
ATP1;

%% The ATP++2 component is responsible for receiving the light signals,
```

either red or green. It then has to send a message to GATP so it can respond appropriately.
ATP2(light:Light) =
%% If the light is green, then we are ready to receive the lightRed signal
(is_green(light))
− > lightRed . sendRed . ATP2(RED)
+
%%If the light is red, then we are ready to receive the lightGreen signal
(is_red(light))
− > lightGreen . sendGreen . ATP2(GREEN);

%% The GATP will receive inputs from ATP1 and ATP2 and will have to react accordingly. It has six parameters: mode (it can be on or off), step (explained later), light (the traffic light), vmax (the speed limit), vtrain (the speed of the train) and bell (the state of the bell, it can be ringing or silent). In order to make possible that the GATP will listen to both of them, a step parameter is defined to guarantee that in each cycle it will listen to both the ATP1 and ATP2 in case they have to transmit something

GATP(mode:Mode, step:Step, light:Light, vmax:Speed, vtrain:Speed, bell:Bellstate)
=

%% If the GATP is turned off, then we are only ready to receive the startAPT signal
(is_off(mode))
− > startATP . GATP(ON, t0, light, vmax, vtrain, bell)

%% Else we can do all the other things
<> (
%% In step zero, it will listen to a possible red light signal or it will show the speed limit. In case of red signal, the speed limit will be set to zero
(step == t0) − > (
receiveRed . showSpeedLimit(s0) . GATP(mode,next(step),RED,s0,vtrain,bell)
+
showSpeedLimit(vmax) . GATP(mode,next(step),light,vmax,vtrain,bell)
)
+
%% In step one, it will listen to a possible green light signal or it will show the speed limit
(step == t1) − > (
receiveGreen . GATP(mode,next(step),GREEN,vmax,vtrain,bell)
+
showSpeedLimit(vmax) . GATP(mode,next(step),light,vmax,vtrain,bell)
)

+
%% In step two, it will receive a speed limit and show it or it will show the speed limit
(step == t2) − > (
sum vNew:Speed . (
receiveSpeedLimit(vNew) . showSpeedLimit(vNew) . GATP(mode,next(step), light,vNew,vtrain,bell)
)
+
showSpeedLimit(vmax) . GATP(mode,next(step),light,vmax,vtrain,bell)
)
+
%% In step three it will get the current speed of the train
(step == t3) − > (
sum vNew:Speed . (
getSpeed(vNew) . GATP(mode,next(step),light,vmax,vNew,bell)
)
)
+
%% In step four, it will check the speed of the train and the speed limit and decide if the bell should be turned on or off
(step == t4) − > (
(vel(vmax) < vel(vtrain))
− > bellOn . GATP(mode,next(step),light,vmax,vtrain,SOUND)
+
(vel(vmax) >= vel(vtrain))
− > bellOff . GATP(mode,next(step),light,vmax,vtrain,SILENCE)
)
+
%%In step five, there is a timeOut and the bell was ringing then the brakes will be applied, else the speed will be shown
(step == t5) − >(
$timeOut.(is_sounded(bell))$
− > brakesOn . GATP(mode,next(step),light,vmax,vtrain,bell)
<> showSpeedLimit(vmax) . GATP(mode, next(step), light, vmax, vtrain, bell)
+
showSpeedLimit(vmax) . GATP(mode,next(step),light,vmax,vtrain,bell)
)
+

%%In step six, if there is a reset action and the speed is zero, the brakes will be released and the bell will be turned of

(step == t6) − > (
resetATP . (vel(vtrain) == 0)
− > brakesOff . bellOff . GATP(mode,next(step),light,vmax,vtrain,SILENCE)
<> showSpeedLimit(vmax) . GATP(mode, next(step), light, vmax, vtrain, bell)
+
showSpeedLimit(vmax) . GATP(mode,next(step),light,vmax,vtrain,bell)
)
+
%% In step seven, the ATP can be stopped, else it will show the speed limit
(step == t7) − > (
stopATP . GATP(OFF,next(step),light,vmax,vtrain,bell)
+
showSpeedLimit(vmax) . GATP(mode,next(step),light,vmax,vtrain,bell)
)
)
;

init
hide(
transmitSpeedLimit,
transmitGreen,
transmitRed
,

allow(
startATP, stopATP, requestReset,
lightGreen, lightRed, transmitGreen, transmitRed,
brakesOn, brakesOff, resetATP,
getPulseFrequency, transmitSpeedLimit,
getSpeed, showSpeedLimit,
bellOn, bellOff,
timeOut,overSpeed,noResponse
,
comm(

sendSpeedLimit|receiveSpeedLimit − > transmitSpeedLimit,
sendGreen|receiveGreen − > transmitGreen,
sendRed|receiveRed − > transmitRed
,

GATP(OFF,t0,GREEN,s40,s0,SILENCE) || ATP1 || ATP2(GREEN)
)));

8 Requirements expressed in modal formulas

This section contains a list of the initial requirements translated into modal formulas:

1. (a) $forall_{status:TrainSensData}.[true^*.TrainSensData(arrives).\overline{TrainSensData(broken)}^*.bellC$

   (b) $[stopATP.\overline{startATP^*}.bellOff]false$

   (c) $[stopATP.\overline{startATP^*}.brakesOn]false$

   (d) $[stopATP.\overline{startATP^*}.brakesOff]false$

   (e) $\forall_{v1:Speed}[stopATP.\overline{startATP^*}.showSpeedLimit(v)]false$

2. (a) $\forall_{v1,v2:Speed}.[true^*.showSpeedLimit(v1).\overline{stopATP^*}.getSpeed(v2)](v2 > v1) \rightarrow \mu X.\nu Y.\ ([\overline{bellOn}]X \bigwedge < true > true) \bigvee ([showSpeedLimit(v1)]Y \bigwedge < true > true)$

   (b) $\forall_{v1,v2:Speed}.[true^*.showSpeedLimit(v1).\overline{stopATP^*}.getSpeed(v2)](v2 <= v1) \rightarrow \mu X.\nu Y.\ ([\overline{bellOff}]X \bigwedge < true > true) \bigvee ([showSpeedLimit(v1)]Y \bigwedge < true > true)$

   (c) $\forall_{v1,v2:Speed}.[true^*.getSpeed(v1).\overline{stopATP^*}.showSpeedLimit(v2)](v2 >= v1) \rightarrow \mu X.\nu Y.\ ([\overline{bellOff}]X \bigwedge < true > true) \bigvee ([showSpeedLimit(v2)]Y \bigwedge < true > true)$

   (d) $\forall_{v1,v2:Speed}.[true^*.getSpeed(v1).\overline{stopATP^*}.showSpeedLimit(v2)](v2 < v1) \rightarrow \mu X.\nu Y.\ ([\overline{bellOn}]X \bigwedge < true > true) \bigvee ([showSpeedLimit(v2)]Y \bigwedge < true > true)$

3. $\forall_{f,fOld:Freq}.[true^*.getPulseFrequency(fOld).\overline{stopATP^*}.getPulseFrequency(f)]$ $\mu X.\nu Y.([\overline{showSpeedLimit(freq2speed(f))} \bigvee \overline{showSpeedLimit(s0)}]X \bigwedge < true > true) \bigvee ([showSpeedLimit(freq2speed(fOld))]Y \bigwedge < true > true)$

4. (a) $[\overline{lightRed^*}.showSpeedLimit(s0)]false$

(b) $\forall_{v:Speed}.[true^*.lightRed]\mu X.\nu Y.([\overline{showSpeedLimit(s0)} \bigwedge \overline{stopATP}]X \bigwedge < true > true) \bigvee ([showSpeedLimit(v)]Y \bigwedge < true > true)$

5. (a) $\forall_{v1,v2:Speed}.[true^*.showSpeedLimit(v1).\overline{stopATP^*}.getSpeed(v2).\overline{stopATP^*}.timeOut](v2 > v1) \Rightarrow \mu X.\nu Y.([\overline{brakesOn \bigwedge showSpeedLimit(v1)}]X) \bigwedge ([showSpeedLimit(v1)]Y \bigvee < true > true)$

   (b) $\forall_{v1,v2:Speed}.[true^*.getSpeed(v1).\overline{stopATP^*}.showSpeedLimit(v2).\overline{stopATP^*}.timeOut](v2 < v1) \Rightarrow \mu X.\nu Y.([\overline{brakesOn}]X \bigvee < true > true) \bigwedge ([showSpeedLimit(v2)]Y \bigvee < true > true)$

6. (a) $[true^*.brakesOn.\overline{getSpeed(s0)}^*.brakesOff]false$

   (b) $[true^*.brakesOn.\overline{resetATP}^*.brakesOff]false$

7. (a) $\forall_{v:Speed}.[true^*.getSpeed(s0)^+.\overline{getSpeed(v)^*}.resetATP\mu X.([\overline{brakesOff}]X \bigwedge < true > true)$

   (b) $\forall_{v:Speed}.[true^*.getSpeed(s0)^+.\overline{getSpeed(v)^*}.resetATP\mu X.([\overline{bellOff}]X \bigwedge < true > true)$

8. (a) $\forall_{v:Speed}.[true^*.getSpeed(s0)] < [\overline{getSpeed(v)}^*.resetATP > true$

9. (a) $\forall_{f:Freq}.[true^*]\mu X.(< true > X \bigwedge < getPulseFrequency(f) > true)$

   (b) $[true^*]\mu X.(< true > X \bigwedge < lightRed > true)$

   (c) $[true^*]\mu X.(< true > X \bigwedge < lightGreen > true)$

   (d) $\forall_{v:Speed}.[true^*]\mu X.(< \overline{stopATP^*} > X \bigwedge < getSpeed(v) > true)$

   (e) $[true^*]\mu X.(< \overline{stopATP^*} > X \bigwedge < stopATP > true)$

   (f) $[true^*]\mu X.(< \overline{startATP^*} > X \bigwedge < startATP > true)$

10. (a) $[true^*] < true > true$

9 Verification

The safety requirements were translated into modal formulas in order to verify them. Each of them will be checked by using the mcrl2 toolset. The verification process follows the following steps:

1. Install the mcrl2 toolset.

2. Save the model with .mcrl2 extension (f.x. model.mcrl2).

3. Save all the modal formulas with .mcf extension (f.x. req.mcf).

4. Comment all the formulas except the one that will be verified.

5. Create a parameterized boolean equation system (req.pbes) with the command: lps2pbes model.mcrl2 -f req.mcf req.pbes

6. Check the validity of the parameterized boolean equation system: bpes2bool req.pbes

7. If the formula is valid, an output true will be obtained

10 Conclusions

All the fomulated safety requirements listed in Section 8 have got *true* outputs in the verification process by using the mcrl2 toolset. Therefore, the Automatic Train Protection system satisfies with all the requirements. The system was thus verified.

11 References

1. J. F. Groote and M. R. Mousavi Modelling and Analysis of Communicating Systems 2013.

2. mCRL2 201210.1 documentation `http://www.mcrl2.org/release/user_manual/user.html`