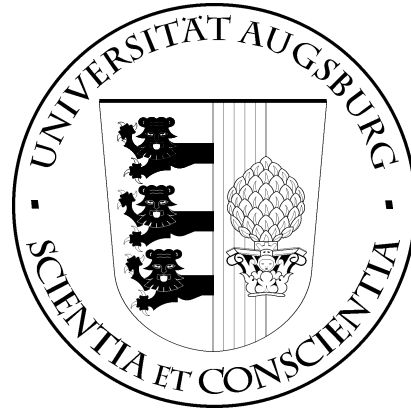


UNIVERSITÄT AUGSBURG

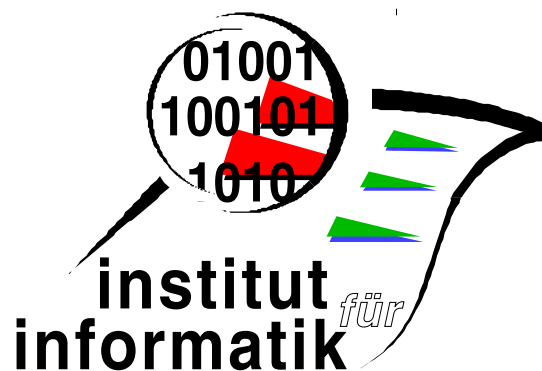


The Statemate Reference Model of the  
Reference Case Study  
'Verkehrsleittechnik'

Jochen Klose, A. Thums

Report 2002-1

Januar 2002



INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

Copyright © Jochen Klose, A. Thums  
Institut für Informatik  
Universität Augsburg  
D-86135 Augsburg, Germany  
<http://www.Informatik.Uni-Augsburg.DE>  
— all rights reserved —

# The Statemate Reference Model of the Reference Case Study ‘Verkehrsleittechnik’

Jochen Klose  
University of Oldenburg  
Jochen.Klose@Informatik.Uni-Oldenburg.de

Andreas Thums  
University of Augsburg  
Andreas.Thums@Informatik.Uni-Augsburg.de

## Abstract

The German railway organization, Deutsche Bahn, prepares a novel technique to control level crossings: the decentralized, radio-based level crossing control. We present a model of this application with activity- and statecharts developed in STATEMATE<sup>1</sup>

The model described in this paper has been developed within the DFG (Deutsche Forschungsgemeinschaft/German Research Foundation) focus area program *Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen*<sup>2</sup> (Integration of Software Specification Techniques for Engineering Applications). The application modeled here is one of the two reference case studies which are provided by this focus area program in order to be able to compare the results of the individual projects. The modelling has been carried out by the projects USE and FORMOSA, other projects (SafeRail and KNOSSOS) have been involved in discussions and in resolving detailed questions about the problem domain.

The case study describes the interaction between trains and level crossings in a radio-based signaling system (funkbasierter Fahrbetrieb, FFB). In order to keep the model sizes manageable several versions of the case study are scheduled. The model presented here has been developed according to the first, simplest version. It implements the control procedure corresponding to the so called ÜS-Prinzip, i.e. the train requests the securing of the crossing and after a certain delay sends another message requesting the status of crossing. This minimal version only considers a restricted set of crossing types:

- single crossings, i.e. an interleaving of two or more securing procedures is not covered.
- single track crossings

In order to further simplify comparing different approaches and techniques the creation of this reference model has been initiated. Even though there exists a reference case study this still allows differing interpretations concerning the details. The reference model presented here has been designed to provide all interested individual projects with a uniform basis for their particular approach. Therefore

---

<sup>1</sup>STATEMATE is a registered trademark of i-Logix Inc.

<sup>2</sup><http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/>

we have tried to narrow down the room for different interpretations without making the design too specific, thereby excluding some techniques from the start. We have chosen the STATEMATE system as a modelling tool, because state charts are used by several projects within the focus area program and it therefore seems to be the most general and useful modelling technique.

## 1 Overall System Architecture (Activity Chart SYSTEM)

First we start with the description of the general design of our reference model for the radio-based crossing control. It provides an overview over the functions and the correlations between the different activities shown in Fig. 1.

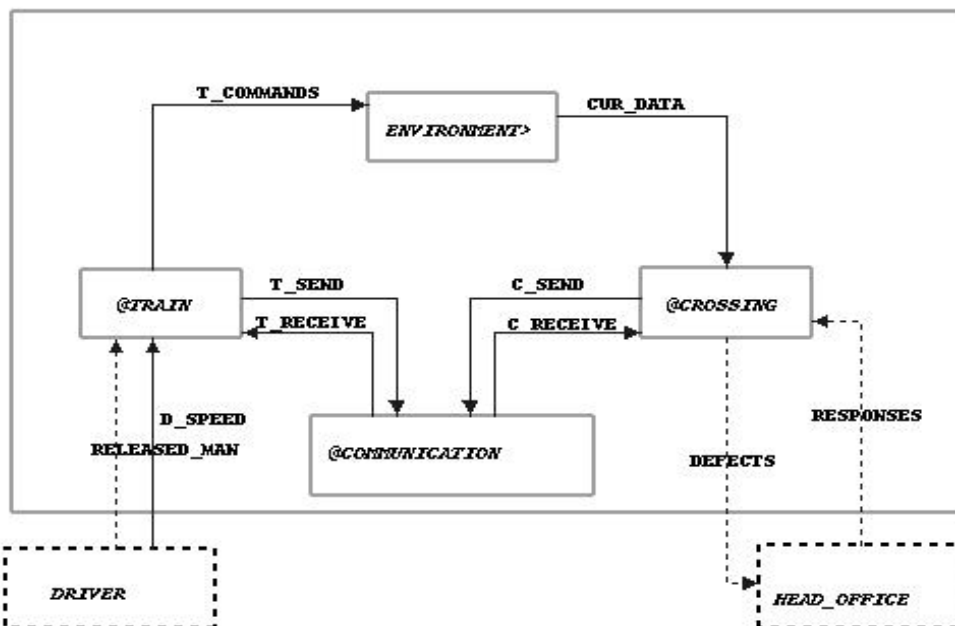


Figure 1: Activity Chart SYSTEM

Having analyzed the requirements on the FFB the separation into six different components has proved to be useful. These are the two main activities TRAIN and CROSSING, the COMMUNICATION between them, the ENVIRONMENT which reflects the real world, and furthermore the ‘human components’ DRIVER and HEAD\_OFFICE, with the latter two not being modeled in detail. The general component SYSTEM consists of these main activities and represents the top level of our model.

The activity TRAIN (see Sect. 4) not fully implements a real train but parts of it which are relevant to the functionality of the FFB. These are the speed control in conjunction with the brake, the odometer for keeping track of the actual position and speed, and the control to enable the crossing.

The implementation of a level crossing in the activity CROSSING (see Sect. 5) contains all its components (the lights, the barrier, a train sensor) and the control units for these components as well as the overall control for closing and opening the crossing.

The physical environment is modeled in the activity ENVIRONMENT (see Sect. 3). Here the position and speed of the train, i.e. the dynamic train data, are described and therefore some calculations have to be made. The environment information is

made available to the activity `CROSSING` via the information flow `CUR_DATA`. In the direction from `ENVIRONMENT` to `TRAIN` there is no data exchange because the train uses the odometer unit to get information about its position and speed, which is modeled independently. In the reverse direction the activity `TRAIN` passes acceleration data (`T_COMMANDS`) on to the `ENVIRONMENT`. Based on this information the speed of the train in the real world is calculated, which thereafter is measured by the odometer in the train.

The information flow between `TRAIN` and `CROSSING` is established by means of the `COMMUNICATION` activity (see Sect. 2). The information exchange takes place via the information flows `T_SEND` and `T_RECEIVE` between `TRAIN` and `COMMUNICATION`, respectively `C_SEND` and `C_RECIEVE` between `CROSSING` and `COMMUNICATION`. Thereby the sent signals (in `T_SEND/C_SEND`) are forwarded by the `COMMUNICATION` activity within the appropriate receive channel (`C_RECIEVE/T_RECEIVE`).

The radio communication has to be explicitly enabled and disabled, which is done by the train. Therefore `T_SEND` also contains the events `ST_COMMUNICATION` and `SP_COMMUNICATION` to start and stop the communication which are not sent to `CROSSING`. The confirmation of the communication establishment (`COMMUNICATION_ESTABLISHED`) is contained in `T_RECEIVE`.

All further exchange of information is related to interaction with the staff at the head office (FFB-Zentrale): The `DRIVER` determines by `D_SPEED` the desired train speed and may re-start the train after a stop (`RELEASED_MAN`). Furthermore the `HEAD_OFFICE` will be informed via `DEFECTS` about any malfunction regarding the components of the crossing and on its part is in the position to ‘repair’ these units (`REPAIRED`) resp. to instruct the software control about the evacuation of the crossing (`CROSSING_VACATED`) by means of `RESPONSES`.

For a comprehensive list of the used information flows see Appendix B.

## 2 Communication

The activity `COMMUNICATION` represents the radio link between `TRAIN` and `CROSSING`. It is explicitly modeled to provide the possibility to simulate errors when establishing the connection and to be able to consider communication delays.

In the current system version it is intended that the train initiates the connection establishment by `ST_COMMUNICATION` whereupon the control of the radio unit (`COMMUNICATION_CTRL`) changes into to state `WAIT_FOR_CONNECTION` (see Fig. 2). This would be the point for possible extensions of the model, in order to model for example the failure of certain communication devices. However at this moment only an ‘establishing lag time’ (ELT) is considered before the radio connection is ready. When the communication is established an acknowledgement is sent to the train (`COMMUNICATION_ESTABLISHED`).

The further interactions take place instantaneously, i.e. requests and acknowledgements are passed on directly. Since the radio mechanism is located between `TRAIN` and `CROSSING`, sent signals are intercepted by it and passed on afterwards. The original signals in `T_SEND` resp. `C_SEND` have thereby the ending `_SND` which becomes `_REC` in `C_RECEIVE` resp. `T_RECEIVE` on the receiver side.

The interactions are in detail (see Appendix B.1) the communication `TRAIN` → `CROSSING` by means of `ENABLE_CROSSING` to request the securing of the crossing, `STATUS_RQ` to request a status message, and `CROSSING_FREE` when the crossing has been vacated by the train. On the other side the communication `CROSSING` → `TRAIN` contains the event `ACK` which acknowledges the receipt of the closing request, and the status message `CROSSING_SAFE` to indicate that the crossing has been secured successfully.

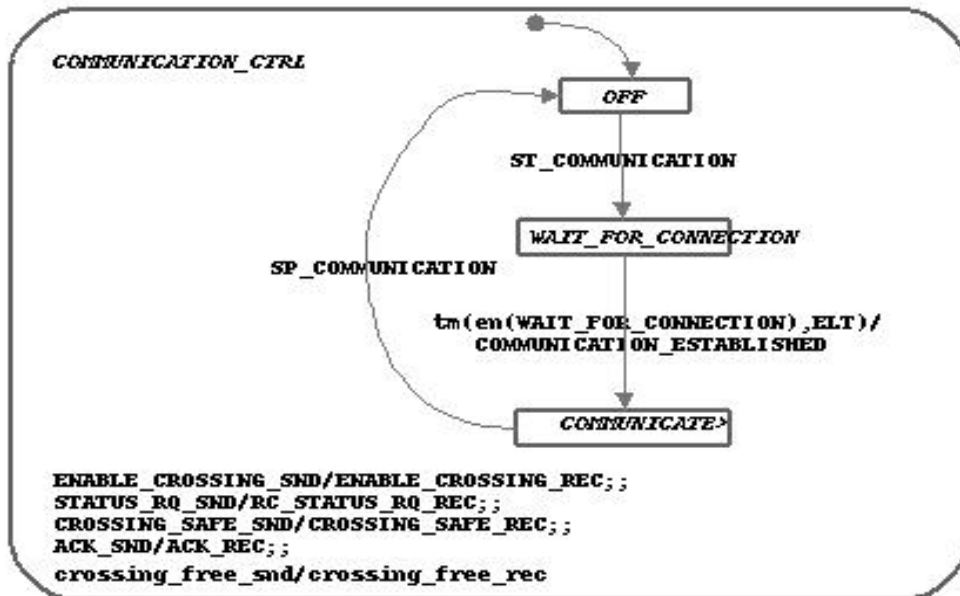


Figure 2: Statechart COMMUNICATION\_CTRL

In the case that the crossing could not secure itself, CROSSING simply ignores the status request (STATUS\_RQ), i.e. an explicit error message is not transmitted!

If the communication between TRAIN and CROSSING is to be terminated, then this is also done by the train via SP\_COMMUNICATION which leads to an immediate abortion of the communication. This is modeled by changing into the state OFF in the COMMUNICATION\_CTRL.

### 3 Environment

The behavior of the ENVIRONMENT is given by state chart ENV\_CTRL (Fig. 3) which keeps track of the train's position (CUR\_DATA.POS) and speed (CUR\_DATA.SPEED) in the real world. Both values are updated in every super-step. This is realized by the expression  $tm(en(POSITION\_UPDATE), 1)$  which is true one super-step after the state POSITION\_UPDATE is entered, i.e. the transition is taken every super-step. This timeout construct is widely used throughout the rest of the model. The new position is calculated by adding the train's current speed to the last position. The new speed is computed by adding the train's current acceleration, resp. subtracting its current retardation value, depending on which field of T\_COMMANDS is set. If both fields are set to non-zero values at the same time, the retardation value has a higher priority. Generally, only one field should contain a non-zero value. The algorithm is based on measuring the distance in meters and the speed in meters per second for simplicity's sake. Other measurement units and/or algorithms can be substituted in more sophisticated versions of the model.

The reason to model this part of the environment is that there has to be a reference point for the train's dynamic data, i.e. here the 'true' position and speed of the train are calculated. Since the train computes these values as well, this allows us to also consider error situations where the data calculated by the train differs from the 'real' value calculated by the environment. The position provided by the ENVIRONMENT serves also as an input for the train sensor (see Sect. 5).

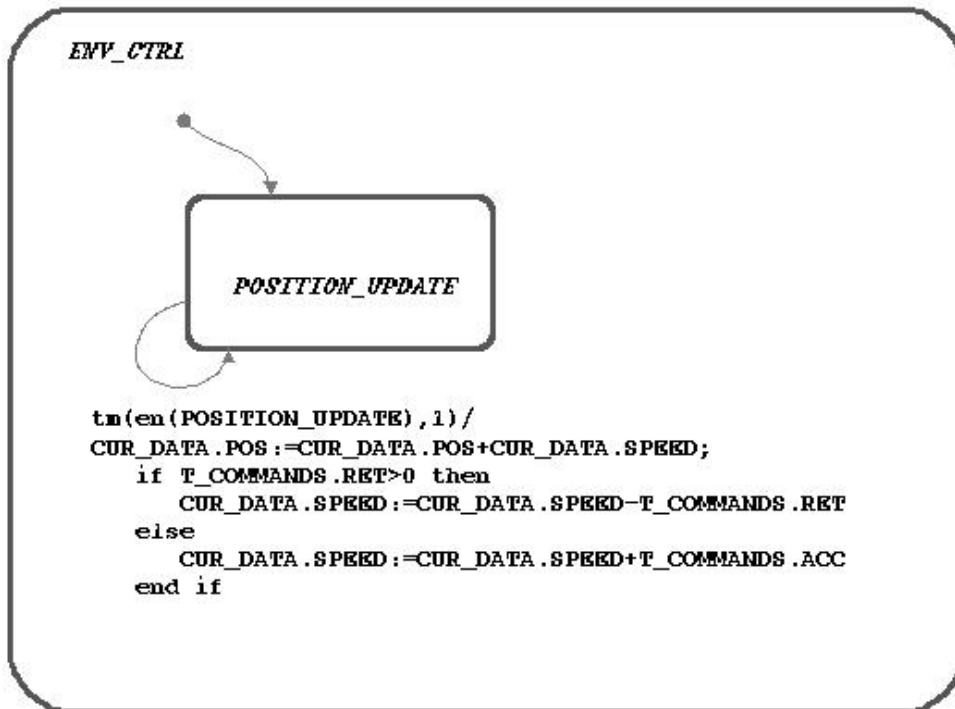


Figure 3: State chart ENV\_CTRL

## 4 Train

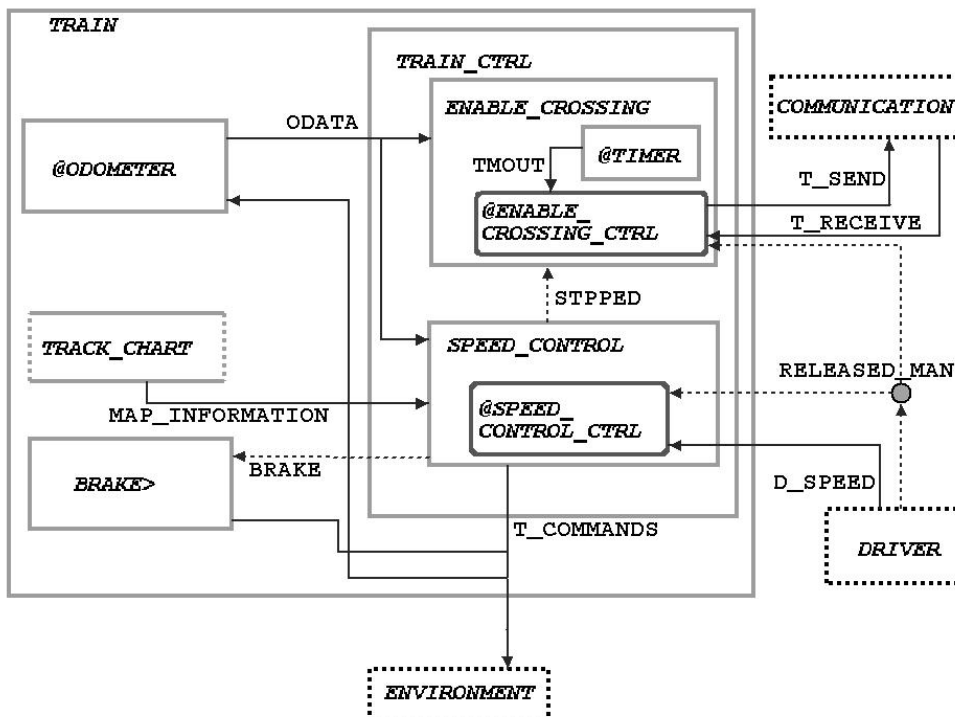


Figure 4: Activity chart TRAIN

The train consists of several sub-components, see figure 4. The activity TRAIN\_CTRL forms the core of the train comprising the two functions of communicating with the crossings (ENABLE\_CROSSING) and controlling the train's speed (SPEED\_CONTROL). The ODOMETER computes the train's speed and position and the BRAKE is activated by the SPEED\_CONTROL if the train is going too fast. The TRACK\_CHART is an empty data store for the moment, but it will hold and provide the static track information in the future.

#### 4.1 Speed Control

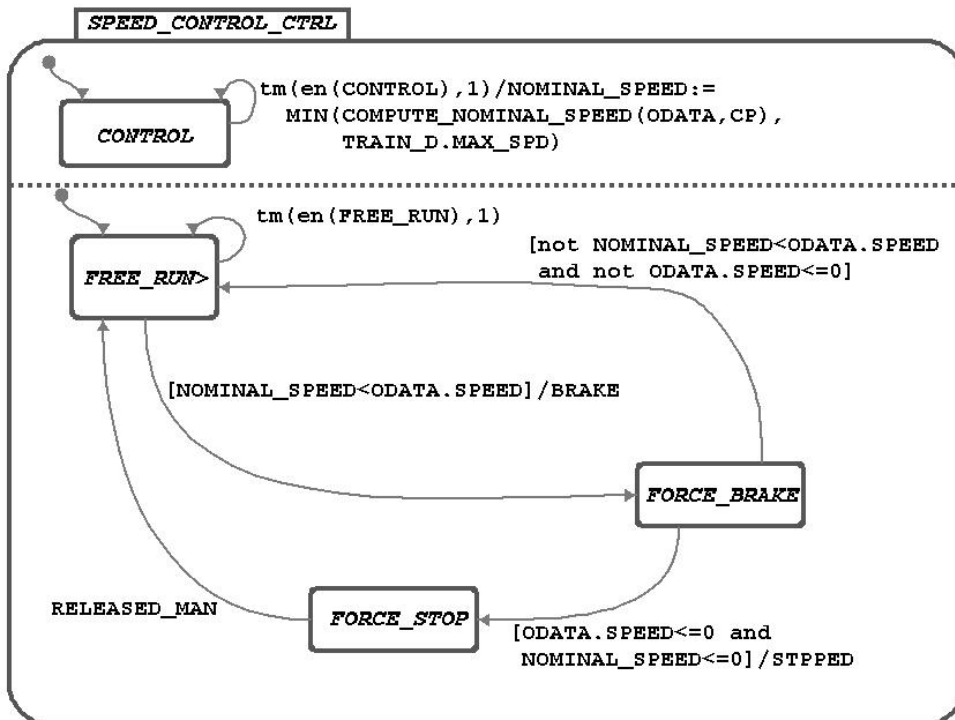


Figure 5: State chart SPEED\_CONTROL\_CTRL

The state chart SPEED\_CONTROL\_CTRL shown in figure 5 consists of two parallel substates; the upper one computes the maximally allowed speed (NOMINAL\_SPEED), whereas the lower one checks if the train speed is below the maximum.

The computation of the NOMINAL\_SPEED is done every super-step utilizing the timeout construct described in section 3. The actual computation is deferred to the function COMPUTE\_NOMINAL\_SPEED which takes the dynamic train data (ODATA) and the next control point (CP) as parameters. A control point is located in front of every crossing and represents a not secured crossing.<sup>3</sup> Control points may not be passed by a train unless the crossing is secured, i.e. the target speed at a control point is zero until the crossing is secured. A control point may be set, i.e. the target speed at its position is set to zero, or deleted, i.e. the target speed at its position is set to the maximal value for this track segment. All control points in a track segment are set when the segment is assigned to a train.

<sup>3</sup>A control point in general represents some sort of hazard. This can be a not secured crossing but also a switch in an unknown position, the end of track segment assigned to the train or a number of other possibly dangerous situations. Here we only consider control points induced by not secured crossings.

The function `COMPUTE_NOMINAL_SPEED` — as all other functions and procedures — is left unspecified here to allow different implementations suiting each user’s needs and preferences. The resulting speed is the maximal speed for the current position and should take into account the train’s own position, its distance to the next crossing, the status of the crossing and track-specific data provided by the track chart. Taking the minimum of the computed value and the train’s maximal speed yields the overall maximal speed.

The lower substate checks if the actual train speed (`ODATA.SPEED`) is within the allowed range. If this is the case, we stay in state `FREE_RUN` and compute the new acceleration and retardation values depending on the speed requested by the driver (`D_SPEED`). A simple algorithm is shown here (The record `T_COMMANDS` contains the current acceleration and deceleration for the train and is also sent to the `ENVIRONMENT`):

```
en(FREE_RUN)/if (D_SPEED-ODATA.SPEED)<0 then
  T_COMMANDS.ACC:=0;
  T_COMMANDS.RET:=MIN(D_SPEED-ODATA.SPEED,TRAIN_D.MAX_RET)
else
  T_COMMANDS.ACC:=MIN(D_SPEED-ODATA.SPEED,TRAIN_D.MAX_ACC);
  T_COMMANDS.RET:=0;
end if
```

If the driver’s speed is greater than the train’s actual speed, the train is accelerated, otherwise it is decelerated. If the two speeds are equal, the train is neither accelerated nor retarded. Again, this is only a simple example of an algorithm which distinguishes only two acceleration/deceleration degrees: none and maximum. More realistic algorithms can be used in more advanced versions of the model.

When the train speed is greater than the allowed speed, state `FREE_RUN` is exited, `FORCE_BRAKE` is entered and the brake is activated. Once the train speed is less than the maximal speed again we return to `FREE_RUN` and resume the above algorithm. If the `NOMINAL_SPEED` and the actual train speed are both zero<sup>4</sup>, the train has stopped, state `FORCE_STOP` is entered and `SPEED_CONTROL_CTRL` emits the event `STPPED` to `ENABLE_CROSSING_CTRL`. Note the not every halting of the train is indicated by a corresponding `STPPED`. Only those stops are considered which result from the `NOMINAL_SPEED` dropping to zero. If the train stops for other reasons, it may resume its course on its own without intervention by the driver. But in the former case the driver has to confirm manually that the crossing may be passed safely (`RELEASE_MAN`) in order to continue. `SPEED_CONTROL_CTRL` then switches back into the normal behavior of state `FREE_RUN`.

## 4.2 Crossing Activation

The activity `ENABLE_CROSSING` (cf. Fig. 4) contains the state chart `ENABLE_CROSSING_CTRL` shown in figure 6, which handles the communication between train and crossing, and a timer (see Sect. 4.2.1) which is used to supervise that the train reaches a secured crossing in time. Here we first consider the communication behavior of the train described by `ENABLE_CROSSING_CTRL`. Once the train reaches the activation point — indicated by the condition `V_ENABLE_POINT_P`<sup>5</sup> — it changes its

<sup>4</sup>Because the braking procedure does not check, if the resulting speed is less than zero when subtracting the retardation value from the current speed, negative speeds can occur. Therefore it is necessary to use  $\leq$  in the condition expression.

<sup>5</sup>Note that the conditions marked by the suffix `_P` represent procedures; in the current version they are modelled as conditions for simplicity’s sake. Thus each user may implement the functionality which is abstracted from by these conditions in his preferred method. The condition `V_ENABLE_POINT_P` abstracts from the computation of the activation point for a crossing.

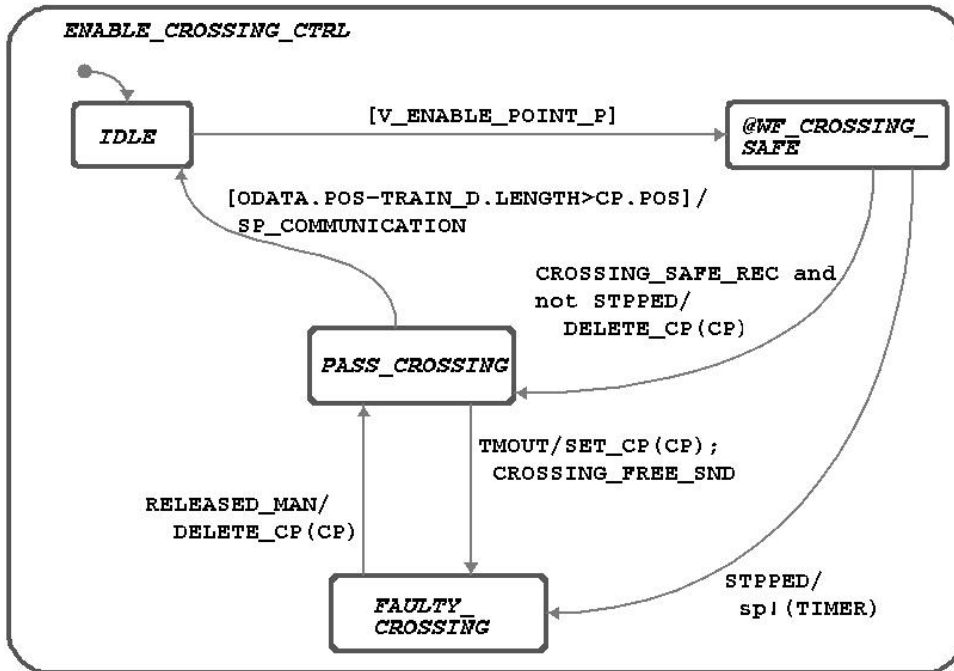


Figure 6: State chart ENABLE\_CROSSING\_CTRL

state from IDLE to WF\_CROSSING\_SAFE which realizes the protocol between the train and the communication channel. The detailed behavior is shown in state chart WF\_CROSSING\_SAFE in figure 7.

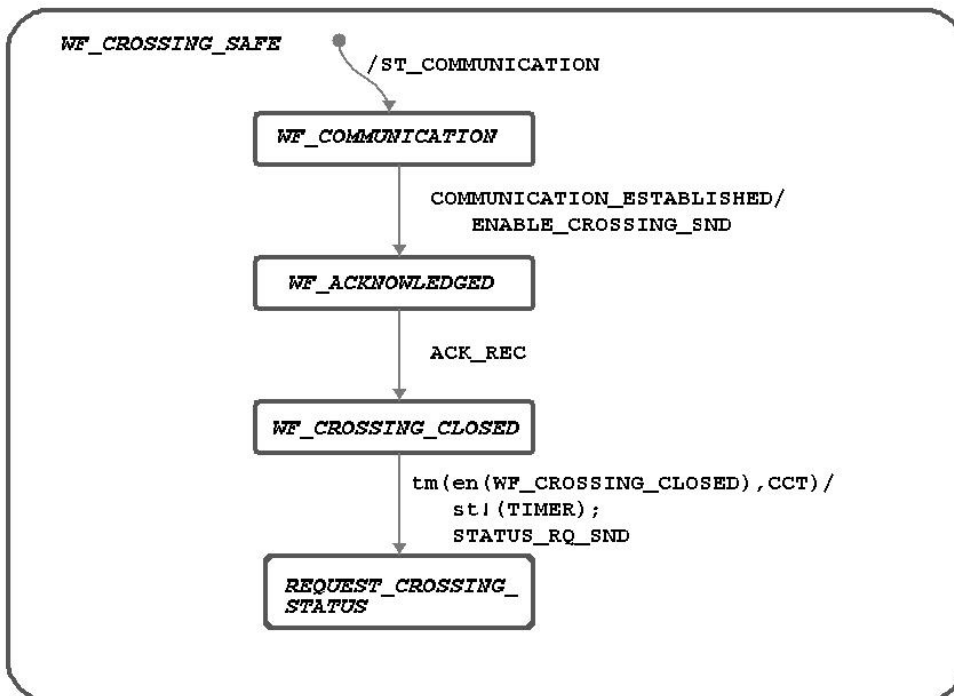


Figure 7: State chart WF\_CROSSING\_SAFE

On entering `WF_CROSSING_SAFE` the communication is started. Once the connection has been established, the train requests the securing of the crossing by emitting the event `ENABLE_CROSSING_SND`. On receiving the acknowledgment from the crossing, the train sends the status request after waiting an amount of time which corresponds to the time needed for the crossing to carry out the securing procedure. This is the crossing closing time `CCT`. Simultaneously with the status request a timer is started which supervises that the train reaches the crossing in time (see Sect. 4.2.1). In the last state of `WF_CROSSING_SAFE` the status message of the crossing is awaited.

Returning to `ENABLE_CROSSING_CTRL`, we first consider the normal case, i.e. the crossing reports itself safe. Thus the train is still moving, the status message `CROSSING_SAFE_REC` arrives, the state `PASS_CROSSING` is entered and the control point is deleted by the function `DELETE_CP`.<sup>6</sup> Note that the functions executing the deleting and setting of control points are left unspecified (see above). When the train now passes the crossing – represented by its control point – the communication channel is closed and the train is ready for the next crossing. For the computation when the train has passed the crossing the train length has to be taken into account in order to avoid that the train dismisses the crossing while having passed it only partially.

During the communication two error situations can arise: First, the crossing may not be able to secure itself. In this case it does not answer the status request and therefore the train cannot delete the control point with the result that it stops in front of the crossing. This is indicated by `SPEED_CONTROL_CTRL` with the event `STPPED` triggering the transition from `WF_CROSSING_SAFE` to `FAULTY_CROSSING`. Since the train has stopped already, it will not reach the crossing in time and therefore the timer is stopped. In this situation the driver has to manually confirm that the crossing can be safely passed (`RELEASE_MAN`) entering state `PASS_CROSSING` and deleting the control point in order to allow the train to pass it.

The second error situation arises when the crossing has answered the status request but the train is unable to pass it within the maximal barrier shut time. This is indicated by the event `TMOUT` sent by the timer resulting in changing from state `PASS_CROSSING` to `FAULTY_CROSSING`, setting the control point again via the function `SET_CP` and notifying the crossing that the train will not reach it in time (event `CROSSING_FREE_SND`). Again this situation has to be resolved by the driver.

#### 4.2.1 Time Supervision

The timer shown in figure 8 supervises that the train reaches an already secured crossing before the maximal barrier closed time elapses, i.e. the maximum amount of time that a crossing may stay closed without a train passing it. It generates a timeout when this property is violated.

The timer implements a simple counter which increments a counter variable `T`. The timeout (`TMOUT`) is generated depending on two conditions, `V_STILL_SAFE_P` and `V_BRAKE_POINT_P`, which again represent procedures (or rather the result of procedures):

`V_STILL_SAFE_P` indicates, if the train will pass an already *secured* crossing within the maximum barrier closed time. The computation of the value of this condition should take the current train speed and position into account.

`V_BRAKE_POINT_P` indicates, if the train has reached the braking point, i.e. the point where it would have to start braking in order to safely stop in front of the crossing. This computation again depends on the current train speed and position.

---

<sup>6</sup>Recall that all control points are assumed to have been set initially.

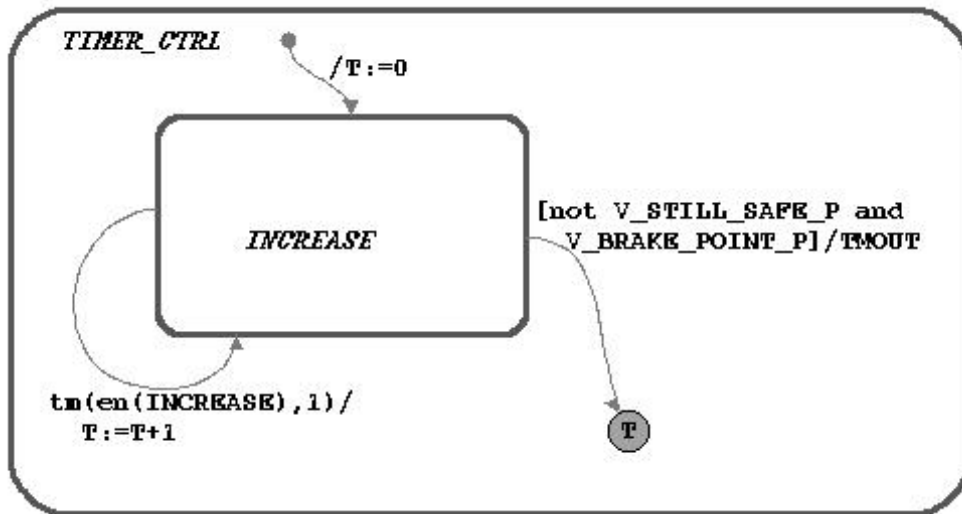


Figure 8: State chart TIMER\_CTRL

The behavior suggested by the transition to the termination connector in figure 8 is thus: If the train reaches its braking point according to the current speed and is unable to reach the crossing before the maximum barrier closed time elapses, the timeout is generated and the timer is stopped. The timeout event is sent to ENABLE\_CROSSING\_CTRL.

### 4.3 Odometer

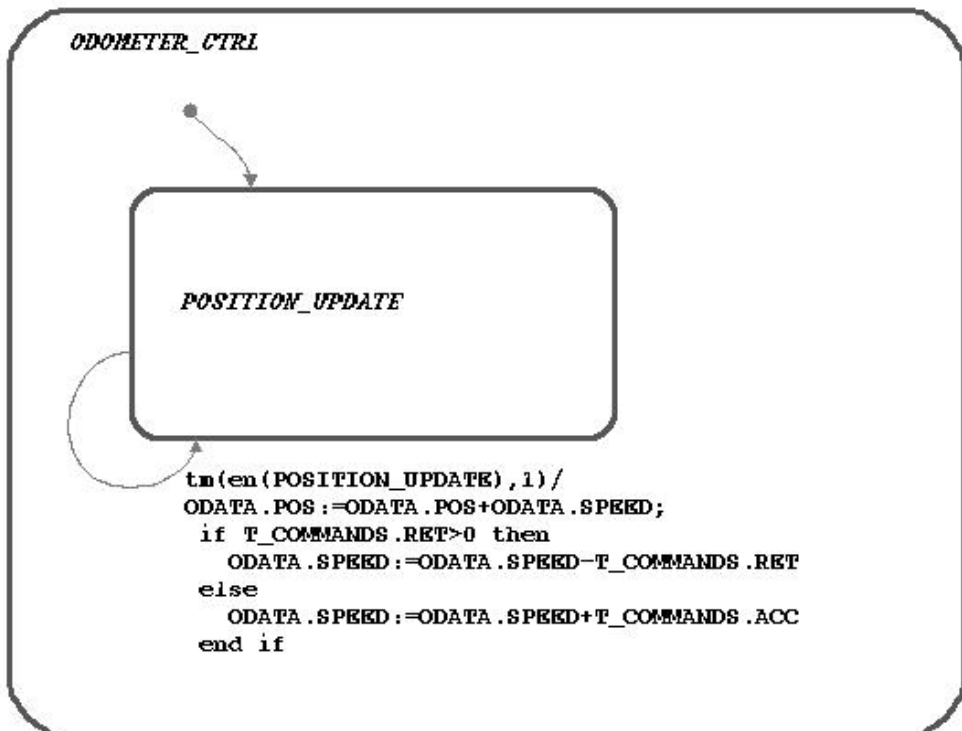


Figure 9: State chart ODOMETER\_CTRL

The ODOMETER (Fig. 9) is responsible for the computation of the actual speed and position of the train. In the present version the algorithm is the same as the one used in the ENVIRONMENT; see section 3 for a detailed description. Future extensions of the model can use different algorithms for the ENVIRONMENT and the ODOMETER to model erroneous speed and position calculations of the train.

#### 4.4 Brake

The BRAKE uses a very simple scheme: when activated it brakes with full force, i.e. the respective fields of T\_COMMANDS are set to the maximum deceleration value of the train and to zero. This is realized by the mini spec:

```

BRAKE/
  T_COMMANDS.ACC:=0;
  T_COMMANDS.RET:=TRAIN_D.MAX_RET

```

### 5 Crossing

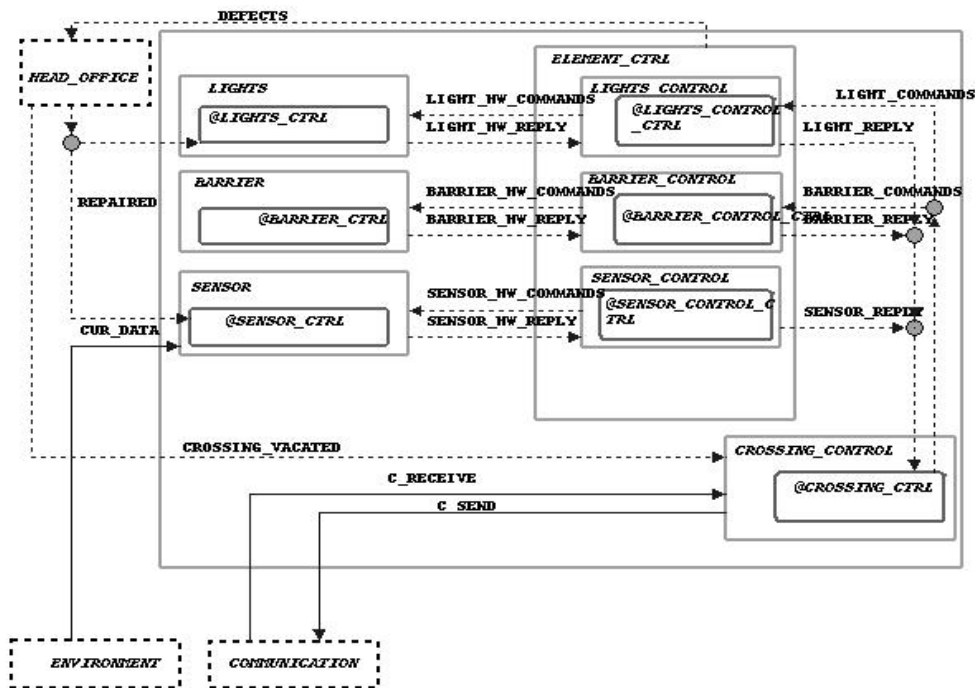


Figure 10: Activity-Chart CROSSING

The activity CROSSING (see Fig. 10) contains the overall software control of the crossing (CROSSING\_CTRL) which is responsible for the coordination of the whole securing process, the component software control (CROSSING\_CTRL) for the involved hardware consisting of the LIGHTS\_CONTROL, the BARRIER\_CONTROL, and the SENSOR\_CONTROL, as well as for each component separately an activity for the description of the hardware functionality, which are the activities LIGHTS, BARRIER, and SENSOR.

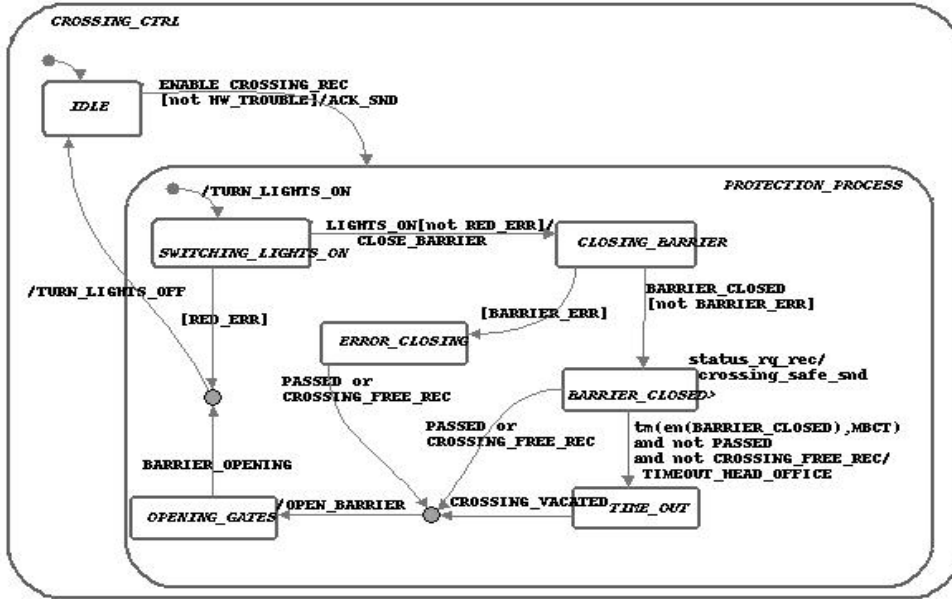


Figure 11: Statechart CROSSING\_CTRL

## 5.1 Overall Software Control (CROSSING\_CONTROL)

The function of this component is the coordination of all measures while closing or opening the crossing. This includes the control of the ‘normal’ securing process (traffic lights on, barrier down, barrier up, lights off), and the consideration of further secondary constraints, for example the maximum time limit for the secure status, as well as error handling.

The initial state `IDLE` remains activated until the control receives the signal `ENABLE_CROSSING_REC`; thereupon if the system is not in an error condition (i.e. `HW_TROUBLE = false`) then the state `PROTECTION_PROCESS` will be entered which encapsulates all further states. A hardware error is present if at least one of the components red light, barrier or sensor operates incorrectly (the condition `HW_TROUBLE = RED_ERR ∨ BARRIER_ERR ∨ SENSOR_ERR` is defined in `CROSSING_CTRL`).

As mentioned above, the securing process itself consists of the handling of the traffic lights and the barriers.

Upon entering the `PROTECTION_PROCESS` the state `SWITCHING_LIGHTS_ON` is activated and the event `TURN_LIGHTS_ON` is sent to `LIGHTS_CONTROL`, where it initiates the signal sequence `YELLOW → RED`. The control remains in this state until either receiving the event `LIGHTS_ON` or a red light malfunction is detected (`RED_ERR`), with both events originating in the activity `LIGHTS_CONTROL`.

**Error Handling:** If during the activation of the traffic light the red light fails, then the rest of the protection process is aborted, the request `TURN_LIGHTS_OFF` is sent to the traffic light control (thus the traffic light device is deactivated) and the initial state `IDLE` is entered. This procedure implies that no `CROSSING_SAFE` message is replied to the approaching train and the crossing is regarded as ‘unsafe’. Also for repeated securing requests the sequence of events is correct, since with a red light defect a hardware error is present (`HW_TROUBLE = true`) and thus the signal `ENABLE_CROSSING_REC` is ignored.

Furthermore no explicit resetting of the overall software control is needed when the damage of the traffic lights device is repaired because being in the `IDLE` state normal operation can take place once the lights are repaired.

After the crossing has been successfully secured by the traffic lights the signal `CLOSE_BARRIER` will be sent to the `BARRIER_CONTROL`. The system remains in the state `CLOSING_BARRIER` until either the barrier closing process is completed (indicated by `BARRIER_CLOSED`) or a barrier hardware error is reported (`BARRIER_ERR`).

Under normal conditions the control stays in the state `BARRIER_CLOSED` until the train has passed (indicated by the signal `PASSED` of the `SENSOR` or the message `CROSSING_FREE` of the `TRAIN`) or the ‘maximum barrier closed time’ (MBCT) has expired. If a status message is requested by means of `STATUS_RQ` the crossing responds with `CROSSING_SAFE`.

**Error Handling** (`ERROR_CLOSING`, `TIME_OUT`):

1. On the other hand if there is some failure (`BARRIER_ERR`) during the closing of the barrier (i.e. control in state `CLOSING_BARRIER`) which will be reported by the `BARRIER_CONTROL` the state `ERROR_CLOSING` will be entered without aborting the overall process. The release of the status message will be suppressed so the crossing remains ‘unsafe’. The error state will only be left when the train has passed whereupon the normal process will be continued by opening the crossing.

If the barrier opens correctly the failure indicator `BARRIER_ERR` will be reset by `BARRIER_CONTROL` and if no further hardware defects are present the crossing will be considered operational.

The traffic lights are turned off in any case and the `CROSSING_CONTROL` enters the state `IDLE`. The further functional behavior depends on the flag `HW_TROUBLE`.

2. Another exception is necessary for the case that the allowed limit MBCT for the crossing to stay in the ‘safe’ state is exceeded (`TIME_OUT`). This is no device failure of the crossing but nevertheless an error message (`TIMEOUT_HEAD_OFFICE`) will be sent to the `HEAD_OFFICE` to inform about this special situation. The crossing remains ‘safe’ until the explicit signal `CROSSING_VACATED` is received from the head office. At that moment the normal event-sequence will be continued just as if the train had passed properly.

After the `CROSSING_CONTROL` has detected the passing of the train (`PASSED`, `CROSSING_FREE_REC` or `CROSSING_VACATED`) an opening request (`OPEN_BARRIER`) will be sent to `BARRIER_CONTROL` while the `CROSSING_CONTROL` stays in the state `OPENING_GATES`. This is indicated by the `BARRIER_OPENING` signal which will be given as soon as the barrier has left the lower position.

Thereupon the lights will be turned off by means of `TURN_LIGHTS_OFF` and the `CROSSING_CONTROL` returns to `IDLE`.

**Error Handling:** There is no explicit error handling during the opening process. The implicit behavior in case of a failure is that with fully closed barriers the traffic lights remain turned on (`=RED`) whereas the lights are turned off when the barriers are not correctly closed any more, i.e. even if the barriers don’t reach their upper position (`OPENED`) the crossing will be opened to traffic. This behavior holds back the traffic no longer than necessary in case of a technical failure.

## 5.2 Traffic Lights

The whole traffic lights device contains – as mentioned above – the software (`LIGHTS_CONTROL`, see Fig. 12) as well as the hardware part (`LIGHTS`, see Fig. 13).

Between the two components there are information flows in each direction (see Appendix B.2). These contain commands from the software to the hardware

(LIGHTS\_HW\_COMMANDS) which are the light control and test request, and in opposite the reply (LIGHT\_HW\_REPLY) with the test results.

Additionally there is an information exchange between LIGHTS\_CONTROL and the CROSSING\_CONTROL for the overall light commands (LIGHT\_COMMANDS) to turn on/off the lights as well as status messages (LIGHT\_REPLY) about command completion and errors.

Furthermore the LIGHTS\_CONTROL sends information about hardware defects (as part of DEFECTS) to the HEAD\_OFFICE which emits the REPAIRED signal to the LIGHTS hardware.

### 5.2.1 Traffic Lights Software (LIGHTS\_CONTROL)

The software control is responsible for the control of the hardware device. Its behavior is defined as follows:

1. It is to be guaranteed that the crossing is open to traffic at least for the ‘minimum green time’ (MGT) before closing it again after it has been opened, i.e. each phase of lights turned off has to be MGT time units or longer.
2. After activation of the device the yellow light is turned on for the ‘minimum yellow time’ (MYT). In case the yellow light is faulty or fails during the yellow phase the remaining amount of MYT is added to the leadtime of the red light.
3. Prior to sending the status LIGHTS\_ON to the CROSSING\_CONTROL the red light has to be turned on for at least ‘minimum red time closing’ (MRTC) time units (i.e. the leadtime of the red light).
4. When the signal TURN\_LIGHTS\_OFF has been received or in case of a failure of the red light the traffic lights are turned off without any delay and the timer which counts the duration of the turned-off phase – ‘green time’ (GT) – starts.

As shown in Fig. 12 the software model is built of two main states: the state OFF to model the traffic lights being turned off, and the state ON which encapsulates all active processes. The following items describe the way our model realizes the above specification:

Entering OFF the SWITCH\_OFF signal is sent to the hardware to ensure that all lights are turned off and to start the GT counter. This timer will be increased each step.

Depending on the GT counter the reaction to the command TURN\_LIGHTS\_ON differs: If the GT counter has not yet reached the MGT the system branches to PENDING where GT is increased further on.

However, when at last GT has reached the MGT the command SWITCH\_ON is sent to the hardware to activate the yellow light and the state YELLOW is entered. This starts the counter ‘elapsed yellow time’ (EYT). EYT measures the period of time the yellow light is on.

Sending the command SWITCH\_OVER which causes the lights to switch from yellow to red the state RED is entered. This will be done when EYT has reached MYT or if the yellow light fails. As mentioned above the signal LIGHTS\_ON will only be sent when the remaining yellow time ( $= MYT - EYT$ ) and the red light leadtime (MRTC) have passed.

As soon as the command TURN\_LIGHTS\_OFF has been received or a failure of the red light has been detected (RED\_ERR) the light control switches back to OFF and the traffic lights device is turned off.

The traffic light software control contains in addition to the software control of the sensor (see 5.4.1) a parallel process for the purpose of cyclic self diagnosis.

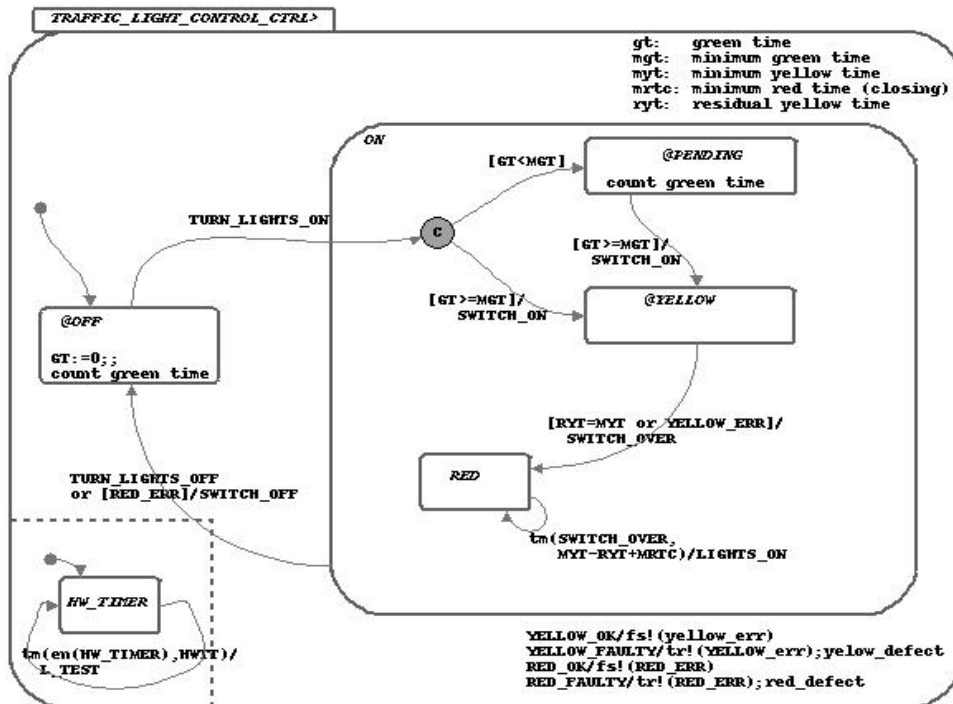


Figure 12: Statechart LIGHTS\_CONTROL\_CTRL

It consists of only a single state (HW\_TIMER) with a loop executed infinitely with a delay defined by the 'hardware test time' (HWTT). When the loop is executed the hardware is asked by means of L\_TEST to check its functionality.

Depending on the response in LIGHT\_HW\_REPLY the variables YELLOW\_ERR and RED\_ERR are set and if necessary the HEAD\_OFFICE is informed via the error messages YELLOW\_DEFECT and/or RED\_DEFECT. This is done by the static reactions defined in LIGHTS\_CONTROL\_CTRL (see Fig. 12).

### 5.2.2 Traffic Lights Hardware (LIGHTS)

The model of the traffic lights hardware describes the behavior of the traffic lights in the real world. As for our purposes only the functional aspects of the devices are of interest physical details are left aside.

The statechart LIGHTS\_CTRL consists of states to indicate which light is on (see Fig. 13). Changing from one state to another is controlled by the commands received from the software control (see Sect. 5.2.1). Initially all lights are off (OFF).

Analogously to the software test cycle described above, which is identical in all software controls (except for the barrier software), all hardware statecharts have in common the existence of parallel processes to simulate hardware failures. Because we are only interested in whether a component works or not, the parts of the statecharts simulating failures all look the same:

Starting with an initial state (`xxx_OK`<sup>7</sup>) indicating no hardware problems, in each step an error of the device may occur. This is modeled through an indeterministic choice of the transition out of the `xxx_OK` state. Either the state in the next step is `xxx_OK` or `xxx_FAULTY`, which represents the error state. The replies to the software control in case of the hardware test `xxx.TEST` are depending on these two states.

<sup>7</sup>'xxx' standing for the hardware device in discussion

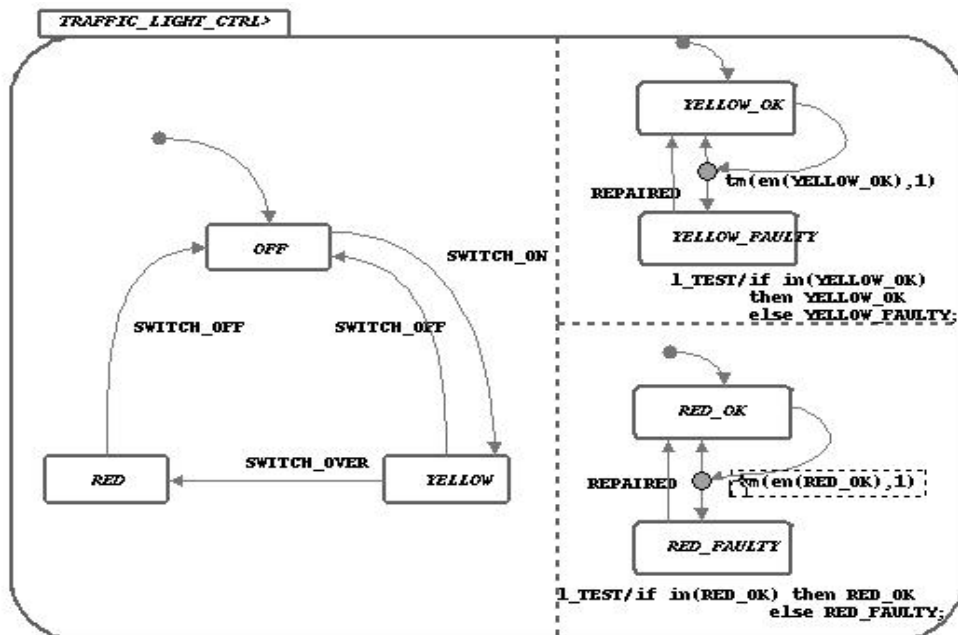


Figure 13: Statechart LIGHTS\_CTRL

Except for the barrier, the error state is only left when the REPAIRED signal from the HEAD\_OFFICE is received. The REPAIRED signal is sent to all hardware devices.

Since there are two independent parts of the traffic lights which can fail – the red and the yellow light – each of them is modeled separately. The initial states are labeled YELLOW\_OK and RED\_OK respectively, the error states are YELLOW\_FAULTY and RED\_FAULTY, and a test report is requested by the L\_TEST command.

### 5.3 Barriers

As the other components of the crossing the barriers consist of a software control (BARRIER\_CONTROL, see Fig. 14) containing the logic and the hardware (BARRIER, see Fig. 15) simulating the physical aspects.

The closing and opening of the barriers is controlled by the barrier software module BARRIER\_CONTROL which receives the appropriate commands (CLOSE\_BARRIER, OPEN\_BARRIER) from the overall software control (see Sect. 5.1) and translates them into action by commands (LOWER, RAISE) to the hardware device BARRIER. The software replies to the CROSSING\_CONTROL by means of BARRIER\_CLOSED, BARRIER\_OPENING, and the failure report BARRIER\_ERR, while itself getting responses from the hardware (CLOSED, OPENED). If errors occur these are signaled to the HEAD\_OFFICE by the software control via BARRIERS\_DEFECT.

#### 5.3.1 Barrier Software (BARRIER\_CONTROL)

The behavior of the barrier software control (see Fig. 14) has been modeled corresponding to the following description (for an overview of the control and information flows see Appendix B.3):

1. The crossing is closed and opened on request of the overall software control (CROSSING\_CONTROL) without delay by lowering and raising the barriers. Further commands given during one of these phases (CLOSING, OPENING) are ignored. Only error exceptions are handled.

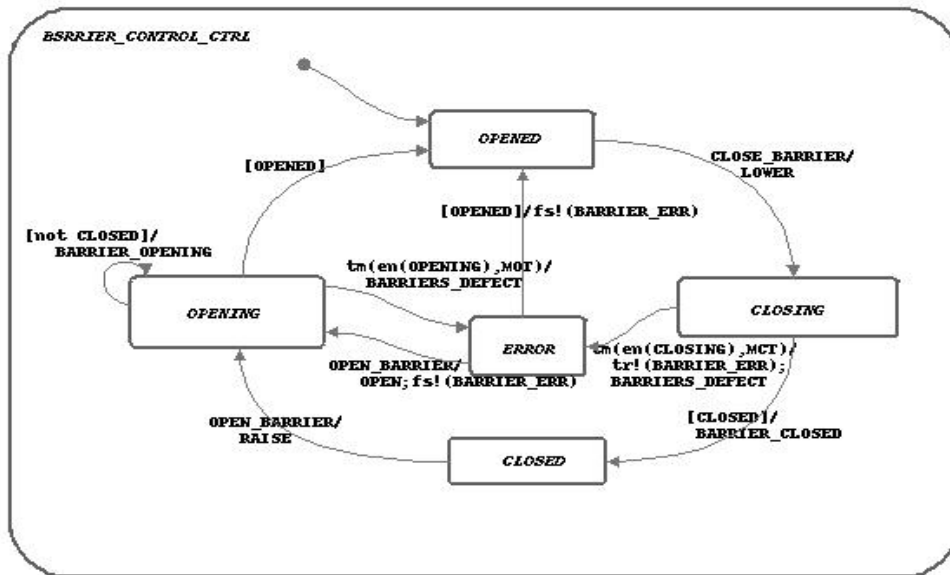


Figure 14: Statechart BARRIER\_CONTROL\_CTRL

- The barrier device informs the CROSSING\_CONTROL of the successful closing of the barriers by means of BARRIER\_CLOSED (initiated by the condition CLOSED being set to *true* by the hardware) and of the beginning of the opening by means of BARRIER\_OPENING (initiated by the condition CLOSED being set to *false*). There is no explicit message to CROSSING\_CONTROL about having opened the barriers successfully.
- There are maximum time limits for the closing ('maximum closing time' (MCT)) as well as the opening process ('maximum opening time' (MOT)). If the execution time for at least one of the operating phases exceeds its limit a hardware failure is assumed.

Because the barriers are mechanical devices without any electrical equipment to control the functionality this kind of error detection is necessary. Whether the mechanics works correctly can only be determined in action. So the software control of the barriers contains no cyclic self diagnosis routines.

- If a failure has been detected (i.e. the state ERROR has been activated), then the normal processing can only be continued when the barriers are fully opened (switch to the state OPENED) or the OPEN\_BARRIER command is received. Each try to open the barriers again has to preserve the MOT limit.

Leaving the ERROR state the error flag BARRIER\_ERR is reset so that the barrier device is again available for the further operation. However because each entering of ERROR causes a message (BARRIERS\_DEFECT) being sent to the HEAD\_OFFICE, the staff is informed about each failure even if the mechanics works well in the meantime.

### 5.3.2 Barrier Hardware (BARRIER)

Because when describing the physical behavior of barriers time is relevant the model of the barrier hardware (BARRIER\_CTRL) is not as simple as providing states for 'up' and 'down', even though these two states (OPEN and CLOSE) also exist in our model, of course. The additional states CLOSING and OPENING model the temporal aspects (see Fig. 15).

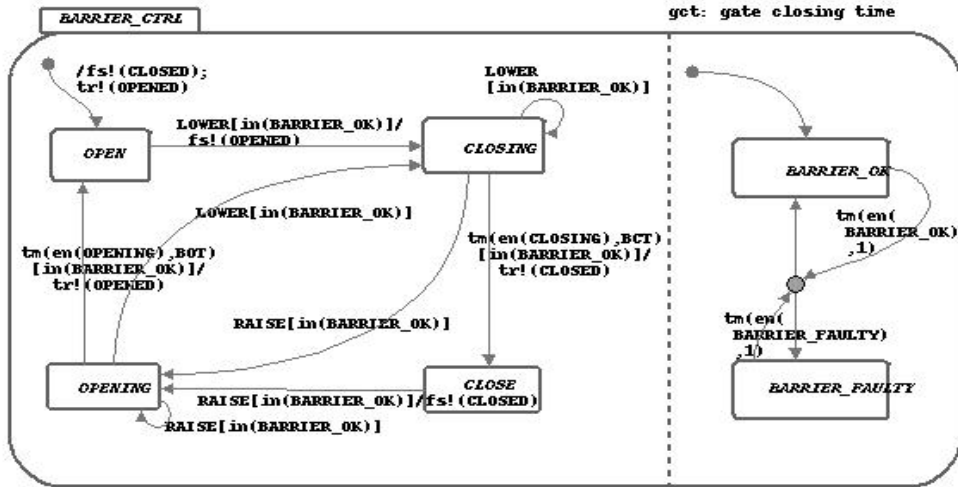


Figure 15: Statechart BARRIER\_CTRL

Initially the conditions **CLOSED** and **OPENED** are initialized (**CLOSED** := *false*, **OPENED** := *true*). The further operation is as follows:

1. Receiving the **LOWER** command in the **OPEN** state with no hardware failure present the model switches to the **CLOSING** state setting **OPENED** to *false*. This means the barriers are no longer open but have not yet been closed.
2. After the ‘barrier closing time’ (BCT) has passed and the device is still ‘operational’ the **CLOSE** state is reached. The condition **CLOSED** is set to *true*; the barriers are closed.

If instead a hardware failure recurs (i.e. the state **BARRIER\_FAULTY** of the failure simulation is active) the **CLOSING** state remains active even if the BCT has passed. So the condition **CLOSED** is not changed.

The third possibility while closing the barriers, is that the **RAISE** command is received. During normal operation this can never happen according to the model of the barrier software control (see 5.3.1). But if the hardware fails during the closing operation this is the way to get back to an operational state.

Furthermore while being in the **CLOSING** state a **LOWER** command can occur. This may be when after a first **LOWER** command a barrier device failure occurs and the barriers are not closed. Then the device stays in the state **CLOSING** and when a second **LOWER** command occurs the barrier may in the meantime be operational and a ‘closing’ cycle should be possible. Therefore the transition from **CLOSING** to **CLOSING** resets the ‘barrier closing time’.

3. With the barriers closed successfully and operational the command **RAISE** initiates the opening phase. The variable **CLOSED** being reset to *false* indicates that the barriers are no longer closed.

Similar to the lowering process also when raising the barriers the **OPENING** state stays active until – normally after a delay determined by the ‘barrier opening time’ (BOT) – the barriers have reached the upper position. With this step the condition **OPENED** is set to *true* and the **OPEN** state is entered.

In the real world the upper and lower position of a barrier have sensors to signal contact with the gate. This model does without them.

As far as the simulation of failures is concerned it is the same with the `BARRIER_CTRL` as with the other hardware devices (see the paragraph about the hardware failure simulation, page 15) with the difference that there is no `REPAIRED` signal from the `HEAD_OFFICE` to leave the faulty state. Instead the `BARRIER_FAULTY` state behaves as the `BARRIER_OK` state with also a functionality decision each step depending on `BARRIER_OUT_OF_ORDER`.

## 5.4 Sensor

A sensor in the context of a level crossing is a piece of equipment installed at the railroad tracks to detect passing trains. This reference model only provides a simple view of a sensor which detects a train when it is located above the sensor.

The communication between the software (`SENSOR_CONTROL`) which implements the logic of the sensor and the hardware (`SENSOR`) contains on one side the raw information whether the hardware detects a train (`SENSOR_ON`) and on the other side the request for a status message (`S_TEST`) and its answer (`SENSOR_OK`, `SENSOR_FAULTY`) respectively.

The software control of the sensor emits the `PASSED` signal indicating that a train has passed to the `CROSSING_CONTROL` as well as an error message (`SENSOR_ERR`) if necessary. The latter message is also sent to the `HEAD_OFFICE` by means of `SENSOR_DEFECT`. As to the traffic lights hardware the `REPAIRED` signal is also sent to the sensor hardware `SENSOR`.

Unique to this device is the data transfer from the `ENVIRONMENT` activity to the `SENSOR` providing it with the information about the actual train position. This information is needed to determine whether the train is located above the sensor.

(A list of the flows connected to the sensor component can be found in the appendix B.4.)

### 5.4.1 Sensor Software (`SENSOR_CONTROL`)

The sensor software (see Fig. 16) works as follows: As soon as the hardware of the sensor is activated (condition `SENSOR_ON`) the initial state `IDLE` is left and instead the `ACTIVATED` state is entered. This is interpreted as the train starting to pass the sensor.

When the `SENSOR_ON` condition is reset to *false* which means that the train has fully passed the corresponding signal `PASSED` is sent to the `CROSSING_CONTROL` and the software control changes back to the `IDLE` state.

Analogous to the traffic lights (see the paragraph about the software self diagnosis cycle, page 14) the sensor software control performs a cyclic self test requesting a status message of the hardware device by means of `S_TEST`. In case of a hardware failure the `SENSOR_ERR` condition used by the `CROSSING_CONTROL` is set and the `SENSOR_DEFECT` message is reported to the `HEAD_OFFICE`.

### 5.4.2 Sensor Hardware (`SENSOR`)

The structure of the sensor hardware model is quite simple but the details are more complicated as for the other hardware components. At a glance (see Fig. 17) the hardware control contains the two states `OFF` and `ON` as well as the parallel failure simulating process that is typical for all hardware units (see the paragraph about the hardware failure simulation, page 15).

But the decision when to switch from `OFF` to `ON` is not as easy as might be expected. In this model the behavior of the sensor is defined as follows:

- An operational sensor is activated as soon as the last position of the head of the train is located before the sensor position and the actual position

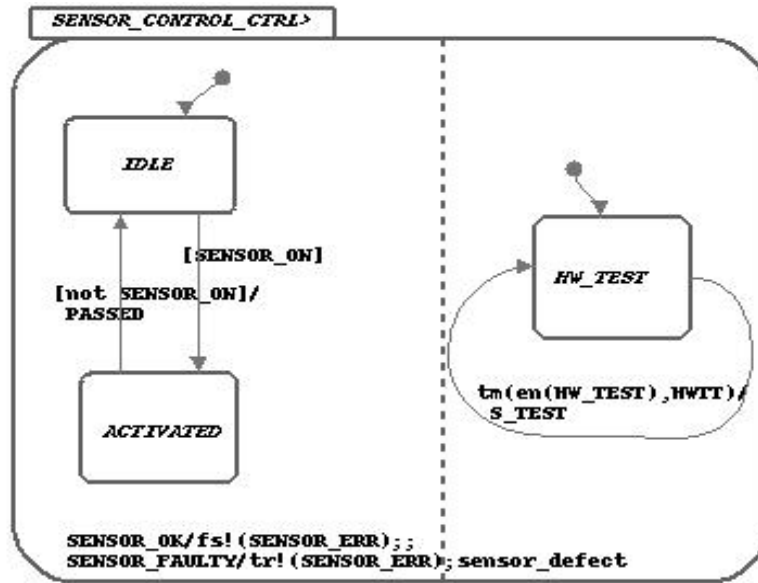


Figure 16: Statechart SENSOR\_CONTROL\_CTRL

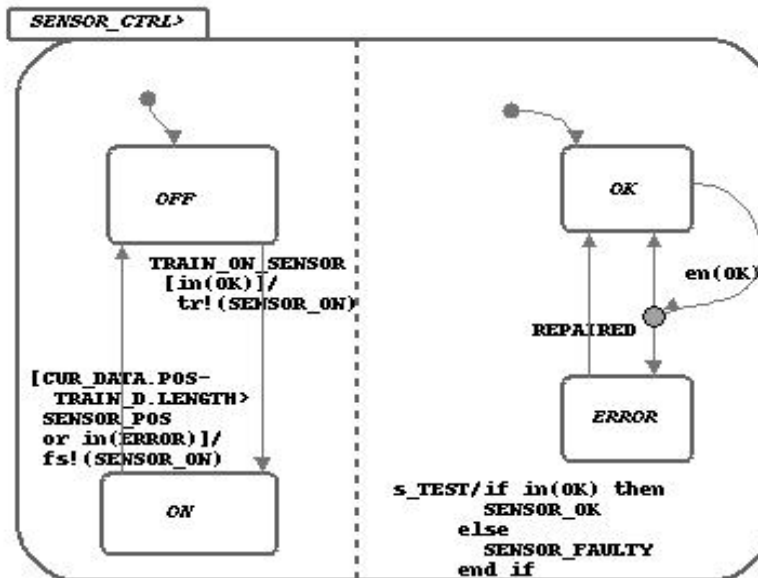


Figure 17: Statechart SENSOR\_CTRL

behind or above it. For this purpose the variable **LAST\_POS** is introduced which is updated each step with the last position of the train taken from **CUR\_DATA.POS**. The event **TRAIN\_ON\_SENSOR** is raised if the above condition is true. This condition is checked in a separate parallel process with the single state **TRAIN\_PASSED**.

The definition implicitly contains the fact that the sensor is only activated (**ON**) if the train passes in forward direction (i.e. in direction with greater position values).

- If there are no failures the sensor is deactivated (**OFF**) when the tail of the

train, which is located at `CUR_DATA.POS - TRAIN_D.LENGTH`, has passed the sensor.

This implies that if a train starts passing the sensor in a forward direction, stops and at the end leaves the sensor backwards, then the sensor will stay activated infinitely.

- A faulty sensor always switches to the `OFF` state, i. e. no train will be detected. This can cause a `PASSED` signal being submitted to `CROSSING_CONTROL` by `SENSOR_CONTROL` even though a train may still be located above the sensor. The actual model does not take care of that possibility.

## 6 Acknowledgement

We would like to thank the projects `KNOSSOS` and `SafeRail` for the serious discussions on the radio-base crossing control case study. These discussions were necessary for the understanding of case study. Further we would thank Alexander Puschnig who helped us in modelling and documenting the case study.

## A Data Types

Here follows a short description of the data types used for modelling the radio-based crossing control system.

- **NATS\_T**  
The *natural numbers* are the types for almost all variables which represent numbers in the model.
- **DYN\_DATA\_T**  
This data type is a record for the dynamic train datas and contains the current position (**POS**) and speed (**SPEED**).
- **STATIC\_DATA\_T**  
The static train information stored in **STATIC\_DATA\_T**, also a record. This data type contains entries for the length of the train (**LENGTH**), the maximal retardation (**MAX\_RET**), the maximal acceleration (**MAX\_ACC**), the minimal speed (**MIN\_SPD**, when the train drives backwards), and the maximal speed (**MAX\_SPD**).

## B Information Flows

Here is an enumeration of the most important control and information flows of our reference model. The arrow ( $\rightarrow$ ) indicates the direction of data exchange.

### B.1 Communication

For the control of the communication there are commands and responses between TRAIN and COMMUNICATION:

- TRAIN  $\rightarrow$  COMMUNICATION
  - ST\_COMMUNICATION  
initiates the communication between TRAIN and CROSSING by establishing a radio link
  - SP\_COMMUNICATION  
aborts the communication between TRAIN and CROSSING
- COMMUNICATION  $\rightarrow$  TRAIN
  - COMMUNICATION\_ESTABLISHED  
status message ‘Communication established’

Between the TRAIN and the CROSSING there are the following messages which are exchanged via the COMMUNICATION:

- TRAIN  $\rightarrow$  CROSSING (in T\_SEND/C\_RECEIVE):
  - ENABLE\_CROSSING\_SND  $\mapsto$  ENABLE\_CROSSING\_REC  
request to secure the crossing
  - STATUS\_RQ\_SND  $\mapsto$  STATUS\_RQ\_REC  
request for status message
  - CROSSING\_FREE\_SND  $\mapsto$  CROSSING\_FREE\_REC  
message about the vacation of the crossing
- CROSSING  $\rightarrow$  TRAIN (in C\_SEND/T\_RECEIVE):
  - ACK\_SND  $\mapsto$  ACK\_REC  
acknowledgement for closing request
  - CROSSING\_SAFE\_SND  $\mapsto$  CROSSING\_SAFE\_REC  
status message ‘Crossing safe’

As mentioned in Section 2 the transformation ( $\mapsto$ ) of a sent message ( $\_SND$ ) to a received message ( $\_REC$ ) is done by the COMMUNICATION activity.

### B.2 Traffic Lights

This section describes the command and information flows concerning the traffic lights. The functionality of the lights is controlled by the overall crossing control by means of these messages:

- CROSSING\_CONTROL  $\rightarrow$  LIGHTS\_CONTROL (in LIGHT\_COMMANDS):
  - TURN\_LIGHTS\_ON  
request to turn on the traffic lights by switching on the yellow light
  - TURN\_LIGHTS\_OFF  
request to turn off the traffic lights

- LIGHTS\_CONTROL → CROSSING\_CONTROL (in LIGHT\_REPLY):
  - LIGHTS\_ON  
indicates that the turn-on process is finished
  - YELLOW\_ERR  
indicates a failure of the yellow light
  - RED\_ERR  
indicates a failure of the red light

The hardware device is controlled by the commands of the LIGHTS\_CONTROL:

- LIGHTS\_CONTROL → LIGHTS (in LIGHT\_HW\_COMMANDS):
  - SWITCH\_ON  
request to turn on the yellow light
  - SWITCH\_OVER  
request to switch off the yellow and turn on the red light
  - SWITCH\_OFF  
request to turn off all lights
  - L\_TEST  
request for test report
- LIGHTS → LIGHTS\_CONTROL (in LIGHT\_HW\_REPLY):
  - RED\_OK  
test report 'Red light ok'
  - RED\_FAULTY  
test report 'Red light faulty'
  - YELLOW\_OK  
test report 'Yellow light ok'
  - YELLOW\_FAULTY  
test report 'Yellow light faulty'

There is also some interaction with the HEAD\_OFFICE:

- LIGHTS\_CONTROL → HEAD\_OFFICE (in DEFECTS):
  - YELLOW\_DEFECT  
error message 'Yellow light faulty'
  - RED\_DEFECT  
error message 'Red light faulty'
- HEAD\_OFFICE → LIGHTS:
  - REPAIRED  
the fact that all hardware devices are repaired

### B.3 Barriers

The control and information flows concerning the barriers are described in this section. Analogous to the lights (see Appendix B.2) the actions of the barriers are controlled by the overall crossing control:

- **CROSSING\_CONTROL** → **BARRIER\_CONTROL** (in **BARRIER\_COMMANDS**):
  - **CLOSE\_BARRIER**  
request to lower the barriers
  - **OPEN\_BARRIER**  
request to raise the barriers
- **BARRIER\_CONTROL** → **CROSSING\_CONTROL** (in **BARRIER\_REPLY**):
  - **BARRIER\_OPENING**  
indicates that the opening process has started
  - **BARRIER\_CLOSED**  
indicates that the barriers are closed
  - **BARRIER\_ERR**  
indicates a failure of the barriers

The hardware device is controlled by the commands of the **BARRIER\_CONTROL**:

- **BARRIER\_CONTROL** → **BARRIER** (in **BARRIER\_HW\_COMMANDS**):
  - **LOWER**  
request to lower the barriers
  - **RAISE**  
request to raise the barriers
- **BARRIER** → **BARRIER\_CONTROL** (in **BARRIER\_HW\_REPLY**):
  - **CLOSED**  
indicates that the barriers are closed
  - **OPENED**  
indicates that the barriers are opened

For the purpose of error notification there is some communication with the **HEAD\_OFFICE**:

- **BARRIER\_CONTROL** → **HEAD\_OFFICE** (in **DEFECTS**):
  - **BARRIERS\_DEFECT**  
error message ‘Barriers faulty’

There is no explicit control flow **HEAD\_OFFICE** → **BARRIER** to indicate the repair of the barriers as it exists at the traffic lights and the sensor. This is explained in the barrier hardware section (see Sect. 5.3.2).

## B.4 Sensor

The following list shows the control and information flows connected to the sensor. Since the sensor is not an interactive device there are no commands from the overall crossing control. Instead there is an information flow from the sensor control to `CROSSING_CONTROL` containing status messages:

- `SENSOR_CONTROL` → `CROSSING_CONTROL` (in `SENSOR_REPLY`):
  - `PASSED`  
indicates that the train has passed
  - `SENSOR_ERR`  
indicates a failure of the sensor

The software control of the sensor (`SENSOR_CONTROL`) receives information from the hardware to interpret them and requests test reports from the hardware:

- `SENSOR_CONTROL` → `SENSOR` (in `SENSOR_HW_COMMANDS`):
  - `S_TEST`  
request for test report
- `SENSOR` → `SENSOR_CONTROL` (in `SENSOR_HW_REPLY`):
  - `SENSOR_ON`  
indicates that the train is on the sensor
  - `SENSOR_OK`  
test report ‘Sensor ok’
  - `SENSOR_FAULTY`  
test report ‘Sensor faulty’

Analogous to the traffic lights (see Appendix B.2) there is some communication with the `HEAD_OFFICE`:

- `SENSOR_CONTROL` → `HEAD_OFFICE` (in `DEFECTS`):
  - `SENSOR_DEFECT`  
error message ‘Sensor faulty’
- `HEAD_OFFICE` → `SENSOR`:
  - `REPAIRED`  
the fact that all hardware devices are repaired

Because the behavior of the sensor depends on the position of the train this information has to be passed to the sensor:

- `ENVIRONMENT` → `SENSOR`:
  - `CUR_DATA`  
the dynamic train data, therein the actual position of the train