SDL−2000 Design Contest
Specification of a Railway Crossing

Explanations to the specification

Jens Brandt, University of Kaiserslautern

1. Requirements

R1. There are a number of tracks for the trains. The specification should be generic in the number of tracks.

R2. There is a controller, taking into account the number of cars waiting and the trains approaching.

R3. (a) There is a gate for cars, which must be controlled. (b) The pattern of cars ap− proaching is simply given by a constant delay between the cars.

R4. (a) Each track has two sensors: one sensor when a train is approaching and one sensor when the train is leaving the gate. (b) There is also a sensor that indicates when there is more than one car waiting.

R5. There is a signal on each track informing the trains if they are allowed to pass or if they have to stop.

R6. The controller can act either by closing the gate or by stopping a train (setting a closing signal). Each of these actions has to be finished before the train reaches the gate.

R7. Trains: (a) the solution should be generic in the number of trains. (b) A train will start the breaking phase as soon it sees a stopping signal and restart when the stopping signal goes off. (c) All trains on the same track are supposed to run in the same di− rection and with a minimal delay between them depending on their speed (a train must be able to stop even if the preceding one stops immediately) (d) There are regular trains, which have a small (maximal) speed and fast trains, which can run faster. Regular and fast trains never run on the same track.

R8. The active elements of the system are the trains, the sensors, the controller and the gate. The cars are not considered to be in the system.

R9. The solution should allow several strategies of the gate controller to be checked.

R10. At least the following strategies should be possible: (a) trains take precedence − at least those with high speed; (b) cars take precedence if there are too many waiting cars. (c) Furthermore, each specification should provide a strategy.

2. Overview (System Railway Crossing)

The system specifies a railway crossing like the one shown in figure 1.

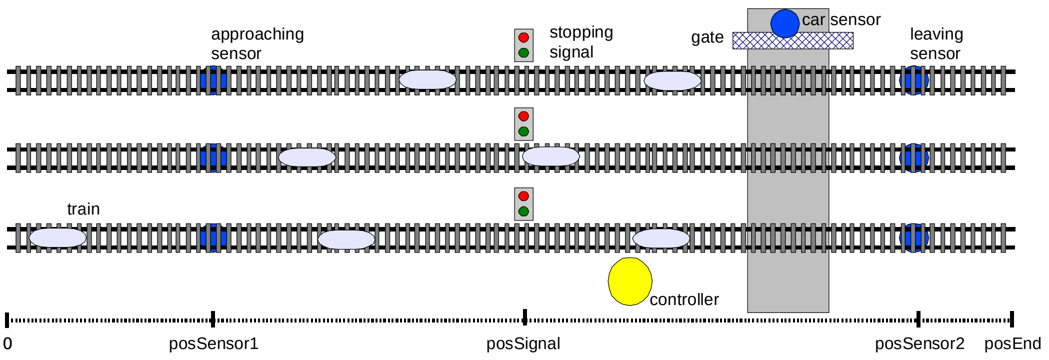

Figure 1: Railway crossing

Looking at the figure, the different parts of the system can be easily recognized:

- the tracks (each having two sensors and several trains running on it)
- the crossing (containing the gate and the car sensor)
- the controller

This corresponds to R8[1], which specifies the active elements: the train sensors, the trains, the gate, the car sensor and the controller.

3. The Tracks (Block aTrack)

3.1. Tracks

The system is generic in the overall number of tracks (R1), so *nrTracks* are instantiated in the *system RailroadCrossing*.

The layout of the track should be generic as well. The positions of the sensors and the stopping signal are given by synonyms in the *package RailroadCrossing* ($posSensor1 < posSignal < posSensor2 < posEnd$). All trains are assumed to approach from the left side. The position of the crossing is implicitly given by $posSignal < posCrossing < posSensor2$.

Each track has to be initialized by setting its (unique) identifier *trackId* (so that the controller can distinguish them) and its type (regular or fast track) (R7d). This is done by the *sensor process*, because it is the only one which is initially created. Its start transition performs the necessary settings. A solution to get a unique track identifier is to take the pid of the *sensor process*. The type of the track (regular or fast track) is arbitrarily chosen by a non−deterministic decision.

3.2. Trains (Process aTrain)

Creation of trains

The system is generic in the number of trains (R7a). This is achieved by the dynamic creation of train processes. Initially there are no *train processes*. The first one of each

1 R*x* (resp. C*x*) refer to the requirements and changes of the problem description.

track is created by the sensor. Afterwards each train creates its successor at an arbitrary time during its lifetime.

Model of the physical aspects (position, speed, acceleration)

In order to specify the behaviour of the trains, their movements have to be modelled. As SDL does not support a continuous model, the trains move stepwise: An interval timer *step* triggers every *stepTime* a new step, during which the position, the speed and the acceleration of a train are updated by the *procedure update*. The position of a train is given by two values which represent the boundaries of an interval in which the train is during the current step. The acceleration and speed values are scaled to the *stepTime* and remain constant during a step. An example is given in figure 2.

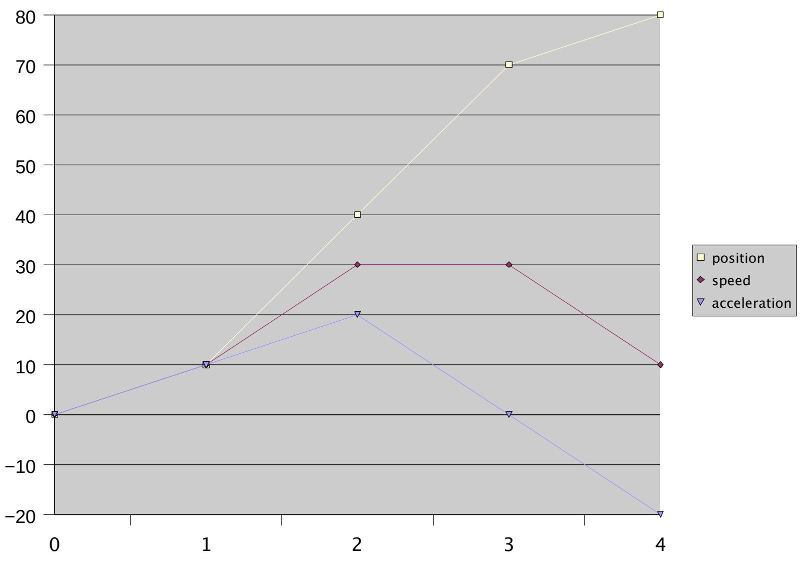

Figure 2: Physical aspects (example run)

Collision prevention

A train must be able to stop even if the preceding one stops immediately (R7c). So every train needs to know the position (and the speed) of its predecessor. As each train knows its successor (because it has created it), it informs it about its current position and speed by sending the signal *position(posX,pos)*. With this information, the following train can determine the maximum acceleration at which no collision will occur:

$$accel := (pos_{pred} - pos) + (speed_{pred} - speed) - minDist$$

Stopping signal

Trains can be controlled by a stopping signal which is on each track (R5). To set and clear this signal, the controller directly sends *trainSignal(sigStatus)* to the first train approaching the stopping signal.

Hence, the controller has to know this train. Initially, the first train announces itself by sending *inSight(track, train)* to the controller. When a train is passing the point at which it is not able to stop in front of the stopping signal anymore (see *procedure brakeDistance*), it hands over the notification to its successor. It does not make sense to stop behind the signal, because the train does not see the stopping signal anymore, and therefore it will not restart when it goes off. If there is no successor at the moment, the next train is told to request the signal notification itself in its start transition (like the first train).

Reaction to the stopping signal

A train immediately (*priority input trainSignal*) stops (resp. restarts) when it gets the signal (R7b). Following trains will automatically stop (resp. restart) due to the collision preven− tion measures (*procedure update*).

3.3. Train sensors (Process theSensor)

Each track has two sensors: one sensor detecting approaching trains and another detecting leaving ones (R4a). These two sensors are modelled by the *sensor process*. In order to detect the trains, all send their current *position* to the process. On the basis of the received values, the process determines if a sensor has been passed and the signal *trainApproaching* or *trainLeaving* has to be sent to the controller. The sensor signals are parameterized with the track identifier, so that the controller knows on which track the event has occurred. (Currently, the track identifier happens to be the *pid* of the sensor. Hence, it would not be necessary to pass this parameter, but it gives some flexibility for changing the track identifier in later versions of the specification).

4. The Crossing (Block theCrossing)

4.1 Gate (Process theGate)

The crossing has a gate which can be opened or closed. (R3a) Opening and closing needs some time, so there are two additional states beside *open* and *closed* which represent the transitions between them. The signal *gateClosed* is emitted to the controller and the car sensor at different times, because the cars cannot pass the crossing during the opening and the closing of the gate, whereas the controller is informed when the action has been completed.

The opening and the closing of the gate can be done by a one or two way handshake with the controller. The two way handshake prevents unsafe system states (Otherwise the controller may set the signals to green before the closing of the gate is finished.), the one way handshake avoids a possible blocking of the controller.

4.2. Car sensor (Process theCarSensor)

There is a car sensor that indicates when there is “more than one car” or when there are “too many cars” waiting (R4b, R10b). To send the signals *carsWaiting* and *manyCarsWaiting* to the controller, the car sensor must keep an account of the cars waiting at the gate. Each time a new one approaches a closed gate (which is simulated by a repeatedly winded up timer (R3b)), it increases the number of waiting cars *(waitingCars)*. If the gate is opened, all cars may pass. To perform this task, the car sensor has to know, whether the gate is open or closed (*gateOpen*, *gateClosed* by *theGate*).

5. The Controller (Block theController)

5.1. Controller (Process theController)

The controller is the core of the system specification. Its task is to prevent unsafe situa−tions by closing the gate or stopping the trains (R6). To perform this it has some informa−tion about the trains approaching and the cars waiting (R2).

As the specification should allow several strategies of the controller to be checked (R9), several process types of the controller are created.

Process Type BasicController

This process defines the common parts of all controllers, which are in detail:

- To control the crossing, each controller must know its tracks. These are stored in a string *TrackLst*. The controllers get to know them by receiving *trackAnnounce* signals which are emitted by the tracks.
- It also has to collect information about the tracks: its type (regular or fast speed), the number of trains between the sensors, the status of the signal and the first train approaching the signal. This information is stored in the array *trackTbl*.
- Unless no other behaviour is specified, the controller simply updates the number of trains between the sensors in the *trackTbl* when receiving a signal from the sensor.
- All controllers have to perform common tasks like setting all signals or controlling the gate. Therefore, procedures (*switchTrains*, *switchBoth*, *switchCars*) are provided, which transform the system to well−defined and commonly usable states. These are in detail:
    1. *trains*: The gate is closed and the stopping signals are green.
    2. *cars*: The gate is open and the stopping signals are red.
    3. *both*: The gate is open and the stopping signals are green. There are no trains between the sensors. (This state is potentially unsafe, but it is very useful: On one side, it is necessary to set the signals to green to keep the trains running. (Otherwise all trains would stop and no sensor event could be detected[2]). On the other side, it would be annoying for the drivers when there is no train on the track and the gate is closed. If this state is used, the controller has to react fast enough to close the gate or stop the train before a train reaches the gate. Systems which should be absolutely safe at whatever delay of the controller should not use the state *both*.)
- All transitions concerning the sensor events are *virtual* so that they can be redefined in the controller to implement its strategy.
- The Basic Controller does not perform any control of the crossing. It remains in its initial (and safe) state (*trains*).

Process Type TrainPrecedenceController

This controller inherits all the described parts of the basic controller. All what is left to specify is its strategy to control the gate (which is quite simple): Every time a train is leaving, it is checked if there are still trains between the sensors. If not, the gate is

2 Here, it is assumed that the signal can be seen from any distance. Alternatively, signal visibility could be restricted.

opened. It will be closed again when the first trains reenters the region. (R10a)

Process Type ManyCarsPrecedenceController

This controller is a subtype of the previous one. The only difference is its reaction to the signal *manyCarsWaiting*. If there are too many cars waiting, it closes the gate and re−opens it after a constant time. (R10b)

Process Type FastTrainsPrecedenceController

Another strategy is to prefer only the fast trains: Each time a car is waiting in front of the gate, regular trains are stopped. When no trains are between the sensors, approaching ones are stopped and the gate is closed. After a constant time, the controller reopens it and clears all stopping signals. (R10c)

6. Chosen parameters

In order to get useful results, the parameters of the system should be chosen carefully. It has been tried to set the parameters to “realistic” values. (Assume that the duration parameters are given in seconds and the distances in metres.)

7. Changes in the requirements

C1. It is not allowed to set the stopping signal for a track when a train is between the two sensors.

C2. It should be possible to manually control the crossing. In this case, unsafe actions should be rejected.

C3. A sensor does not produce one signal, but a sequence of signals (one for each of the wheels). The end of the sequence is given by a delay of a certain size after the last signal.

All three changes affect the controller. The communication with its environment has become more complex due to the bouncing train sensors and the temporarily not settable signals. In order to keep the main control as simple as possible, the controller process is divided up into services (see also: 9. Notes about the usage of SDL 2000). One service performs the actual control (like the former version of the controller process), two new services handle the modifications in the environment. This means that all the process types defining the controllers are converted to service types.

Setting the signals

The setting of the signals (C1) is done by a signal control service. The controller instructs it to set the signals for a certain list of tracks. The service sets the signals of all the tracks where it is possible: If not all can be set, it waits until a train leaves the region between the sensors and tries again.[3] By sourcing out this task, the controller does not block and can concurrently handle sensor events without any changes to its specification.

Control the gate manually

To manually control the gate (C2), a communication path from the system environment to the controller is needed, and the behaviour of the *process type BasicController* has to be

3 Note that there is a possible starvation of the signal controller. A signal may never be set, when there are always trains between the sensors of a track.

extended (The gate should be manually controlled, whatever strategy the controller uses). This change is rather simple: Each time a manual closing of the gate is requested, it is immediately performed, whereas a manual opening is only possible when there are not any trains between the sensors.

Debouncing the sensor signal

As the sensor produces a sequence of signals (C3), it has to be debounced. In order to perform this task, the service winds up a timer for each sensor each time a trigger is detected. After a certain delay without any signals for this sensor, the signal is passed to the main controller. As the input signal sets of all services have to be disjoint , the signals emitted by the train sensor have to be renamed.

8. Used Tools

To build the SDL specification Cinderella 1.3 has been used.

Notes:

- Unused service types in the controller process have to be deleted to start a simulation. Otherwise, Cinderella is not able to start the simulation and will report an error ("Failed to simulate").
- The controller process is not directly defined but by a type. Direct definition will make Cinderella not to find a receiver for the signals that are sent by the processes in the track block (*trackAnnounce, inSight, trainApproaching, trainLeaving*), if there are multiple track blocks. Curiously, it seems to be possible if only one track exists.

9. Notes about the usage of SDL 2000

- Cinderella offers only limited support of SDL 2000. Unfortunately, it does not support composite states, which would have been very useful to realize the controller and especially the changes (sensor debouncing and signal setting). So the obsolete concept of services has been used.
- The new data syntax has been used, but no new data concepts. There is no need for *object types* or complex new data types in this specification.
- Exceptions (which are supported Cinderella) are currently not used as well. There seems to be no necessity.