SDL-2000 Design Contest Specification of a Railway Crossing

(extended Abstract)

Manuel Aguilar
Verimag
2, Avenue de la Vignate
38610, Gières, France

Summary

This document presents our rail road system and its verification. For the specification of this system we use SDL96, because we are not aware of any tool for SDL 2000. Therefore, we just mention where the use of SDL 2000 would have allowed a better or more elegant solution.

For modelling the system we use the commercial tool ObjectGeode [Ver96, Ta99]. Indeed, we specified and debugged (by simulation) the system using ObjetcGeode. Nevertheless, we could not "verify" the system (by means of exhaustive simulation or model-checking)as the ObjectGeode tool was not smart enough to generate a sufficiently small graph. Therefore, we use the IF toolset [BGM01, BFG+99] for verification. IF is a powerful validation tool-set, allowing to deal with SDL specification via a translator sdl2if, based on the Geode API.

This document is organized as follow: Section I begins with the description of the system in SDL96. Section II contains the description of the system in SDL2000. Section III describes the generation of the system model. In order to generate this model it is necessary to limit the value of the input parameters to avoid the state explosion. Finally the Section IV describes the verified properties and the obtained result.

Section I. Solution using SDL92

General design problems and decisions

The railway crossing system is of the kind which are usually solved with a synchronous approach. Indeed, the system progresses with time: trains move with time, independently of the speed of the controller. In the standard SDL time semantics, the speed of the system can only be slowed down by explicit waiting, but it is impossible to specify a system which garantees for example that timeouts (and any other signals) are treated within 1 time-unit. This is however, the feature we need here. Fortunately, ObjectGeode and most other tools do not implement the SDL time concept but that of synchronous systems: time only progresses when the system is stable and no system transition is enabled. The IF environment allows to mix both, deterministic and non-deterministic time progress.

1. The controller reacts on signals from the environment (gate and sensors) and must react according to the chosen strategy within a specified delay (which we fixed to one for simplicity)
2. The other parts of the system (the entities to be controlled) are in fact hybrid systems. They have a time driven dynamic behaviour which changes depending on control signals received. We model discretized versions of these entities: for example, trains update their position periodically. We have chosen the period to be 1 time-unit, as in this system it

makes little sense for the validation to suppose trains to have a different reactivity than the controller.

3. All communication delays between different parts of the system are assumed to be zero. This is not a necessary assumption, non-zero-delays just lead to a slower reaction time and thus necessary greater “security distances” In any case there must exist a known maximal transmission delay, as otherwise the system cannot be controlled.

Notice that the solution proposed here with SDL (with ObjectGeode time progress) is not exactly the one suggested by synchronous languages, such as Lustre [HLR92]. The sensors do not periodically verify if there is a train passing, but the train sends an asynchronous signal exactly once when it enters the crossing region; and correspondingly, the sensor sends an asynchronous signal exactly once when it detects a train. Thus, only the dynamic behaviours (the movement of the train and of the gate) are time driven, discrete behaviours are signal driven. This supposes an underlying protocol layer guaranteeing reliable communication in a fixed delay which is independent of the system design.

The architecture of the system

The system has the architecture shown in figure 1. It is structured into 3 main blocks: the Controller, the Gate and the Track blocks. There is no Environment block, as the behaviour of the environment is described within the block Track (there it determines when a new train appears) and within the block Gate (where it determines when cars arrive and disappear)

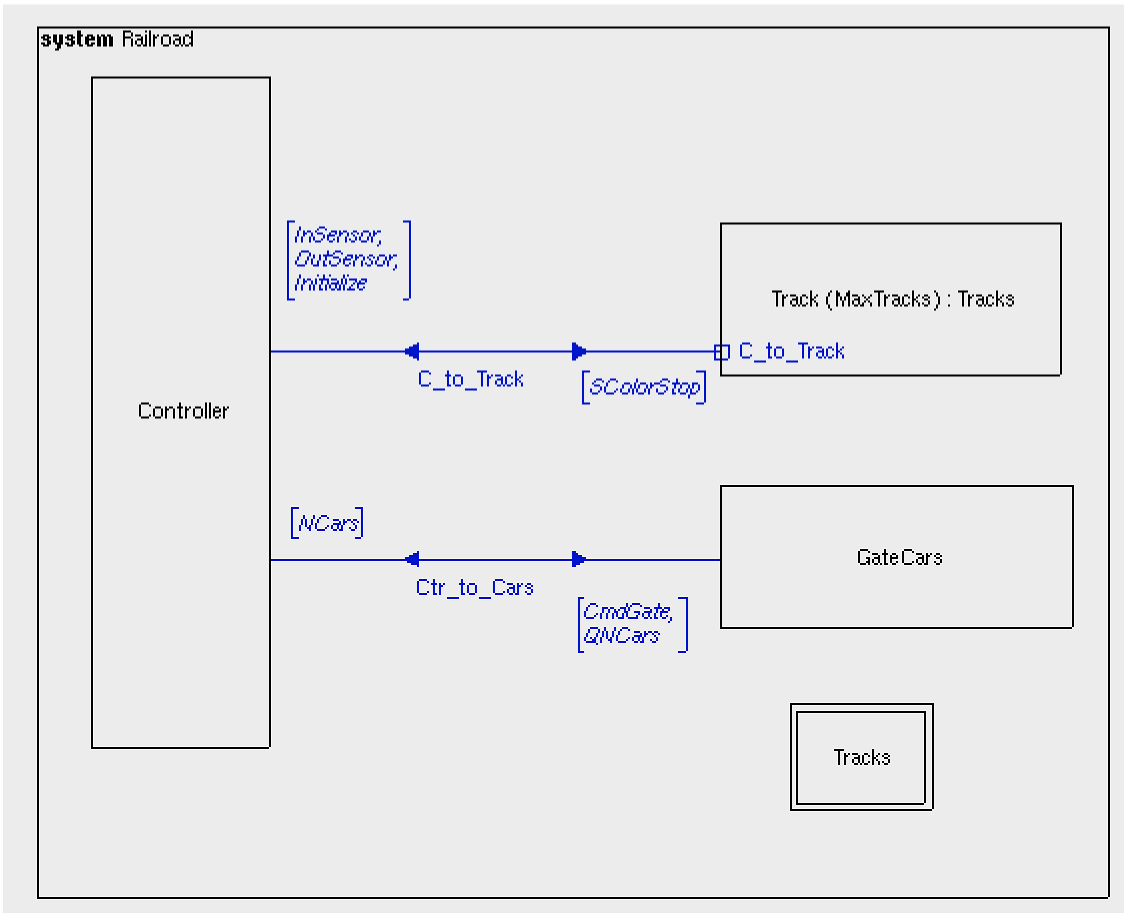


figure 1. System structure.

All the parameters of the system are defined in terms of constants in the main declaration section. Thus, particular instances of the system are obtained by assigning particular values to these constants.

The Controller block

The Controller block contains only one process, the Controller process (in SDL 2000 we would not need the block but only the process). The controller process receives messages from the train sensors (sensor processes are in the Track block) and the gate sensor (sensor is in the gate block). Using this information the controller decides whether it should close or open the gate or change the colour of the red lights of the trains. It never communicates directly with the trains (asking for example their position) otherwise than by means of the train sensor and the red lights. It has a single state init, defining the control loop. The reaction to the different signals is shown in figure 2 (where "do" is a timer set to some value when the gate is open and there are trains waiting; this is the only situation in which the system would deadlock, due to the absence of other signals if no new train arrives, if not reactivated by some timer).

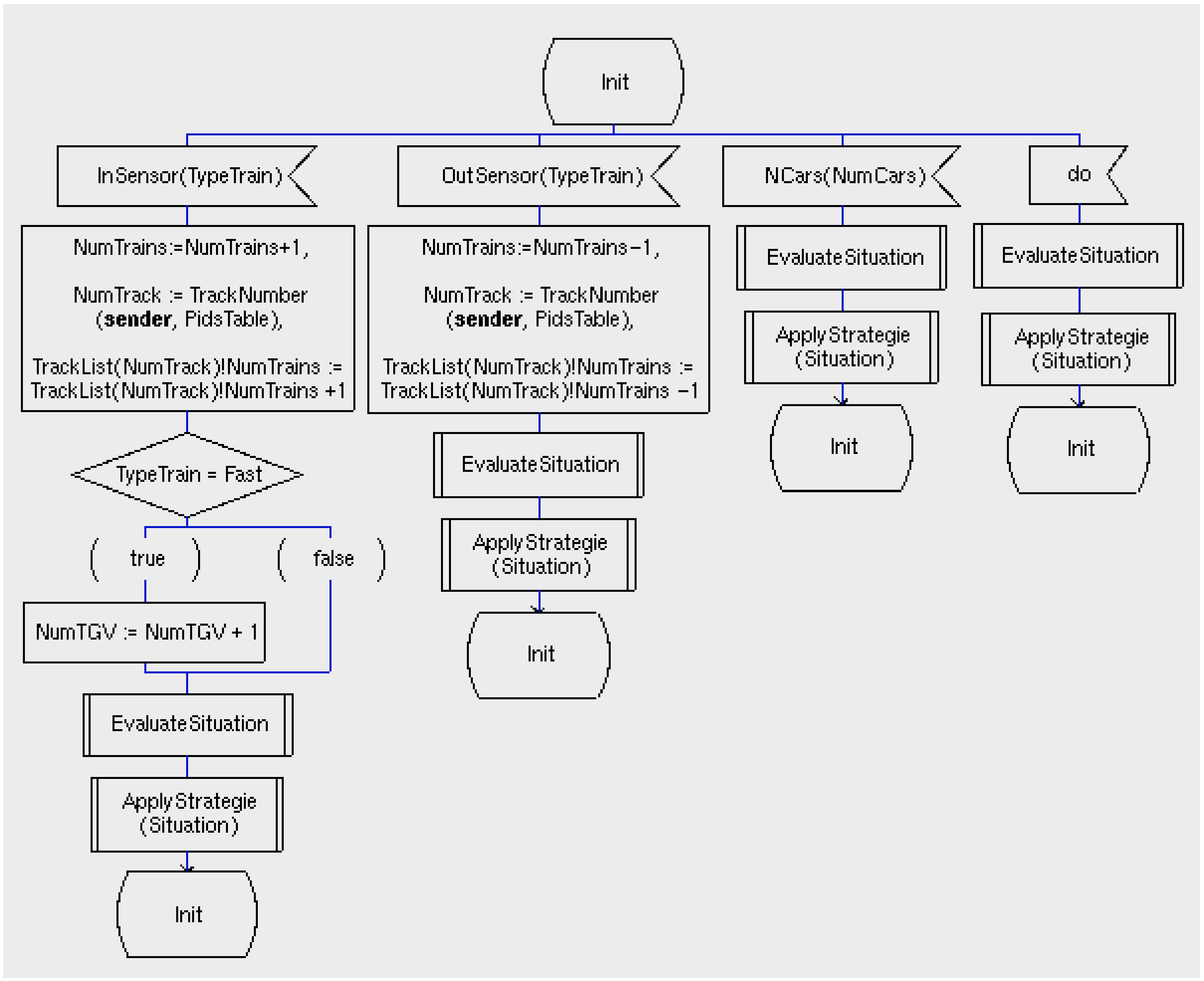

Init
InSensor(TypeTrain)
OutSensor(TypeTrain)
NCars(NumCars)
do
NumTrains:=NumTrains+1,
NumTrack := TrackNumber (sender, PidsTable),
TrackList(NumTrack)!NumTrains := TrackList(NumTrack)!NumTrains +1
NumTrains:=NumTrains-1,
NumTrack := TrackNumber (sender, PidsTable),
TrackList(NumTrack)!NumTrains := TrackList(NumTrack)!NumTrains -1
TypeTrain = Fast
true
false
NumTGV := NumTGV + 1
EvaluateSituation
ApplyStrategie (Situation)
Init

Figure 2. Part of Controller Process.

At each signal reception the controller evaluates the present situation (procedure EvaluateSituation) and acts (procedure ApplyStrategy) according to the strategy requested in the specification of the contest. The chosen reaction depending of the new situation and not of the received signal. The reactions are listed in decreasing priority.

- If there is a fast train in transit and the gate is open, the gate is closed to let pass the train. If there is another train waiting, it will be allowed to pass, thus its light becomes green (if it is not yet).
- If there is more than one train waiting at the closed gate, the gate will be opened, and then the train red lights will become green.
- If there are too many cars waiting, the gate will be opened and the train red light turn red. In this case a timer "do" is set in order not "forget" the trains.
- The controller registers the time at the moment of sending a command (open or close gate) will not send a different command before a certain amount of time
- If there are not trains in transit, the gate is opened.

The strategy of the controller is defined in a procedure which can easily be modified to implement different strategies.

The gate block

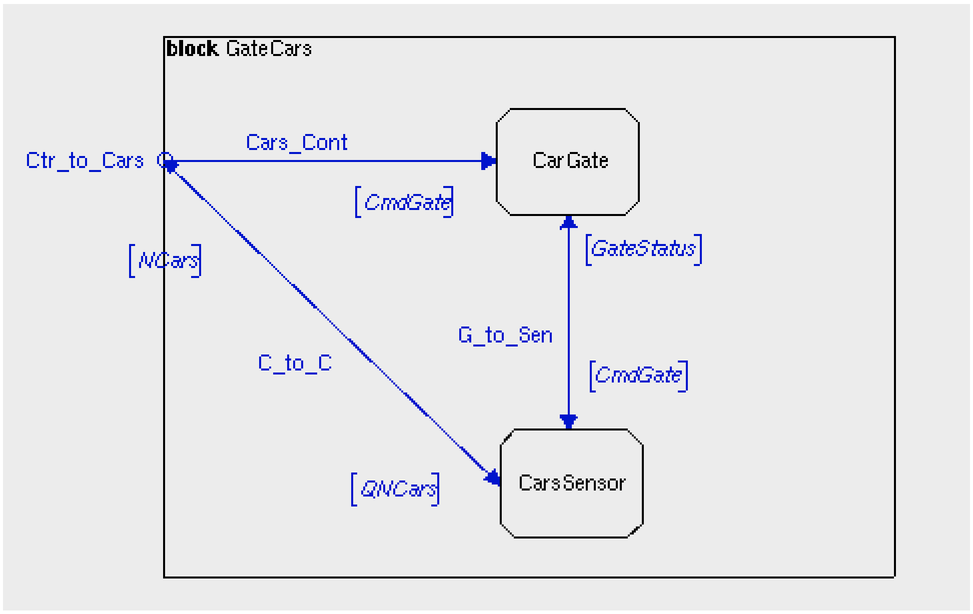


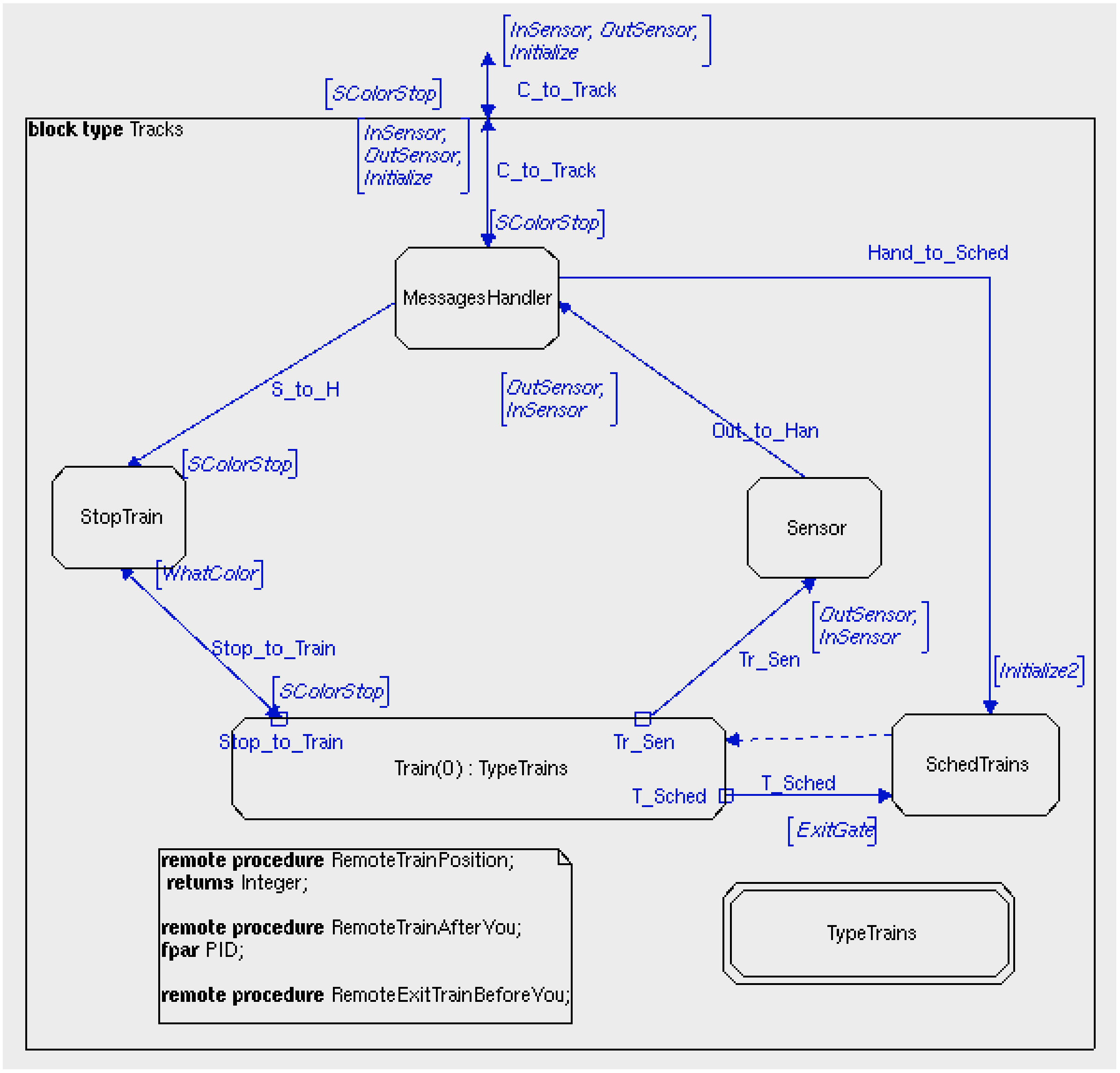
Figure 3. The Block GateCars

The gate block contains the following processes:

CarGate: this process simulates the behavior of the gate. This process reacts to the controller commands, and closes (opens) the gate in several steps.

CarSensor: This process represents the environment, it simulates the number of cars (NumCars) waiting in the gate. If the gate is closed, at each time unit, NumCars is increased by the number "ArrivalCars" (parameter of the system). If the gate is open, each time unit NumCars is decreased by the number "DepartsCars" (parameter of the systems, too).

The type track block



```
remote procedure RemoteTrainPosition;
 returns Integer;

remote procedure RemoteTrainAfterYou;
fpar PID;

remote procedure RemoteExitTrainBeforeYou;
```

Figure 4. Internal structure of the block Track.

The Block type Tracks receives as parameter the number of tracks to create. Notice that creation of a train means that a train enters the zone of the controller; a train dies when it passes the gate.

Each Track is made up of the following processes: Sensor, Train, MessageHandler, StopTrain (the red light) and SchedulerTrains

SchedulerTrains: The SchedulerTrain process represents the environment and creates trains dynamically. It initializes in a non deterministic way the first three parameters of the train processes that it creates (length, speed, and Tgeneration: the time interval between train generations). The last parameter is the PID of the proceeding train (or NULL if it does not exist). This process is illustrated in the next figure.

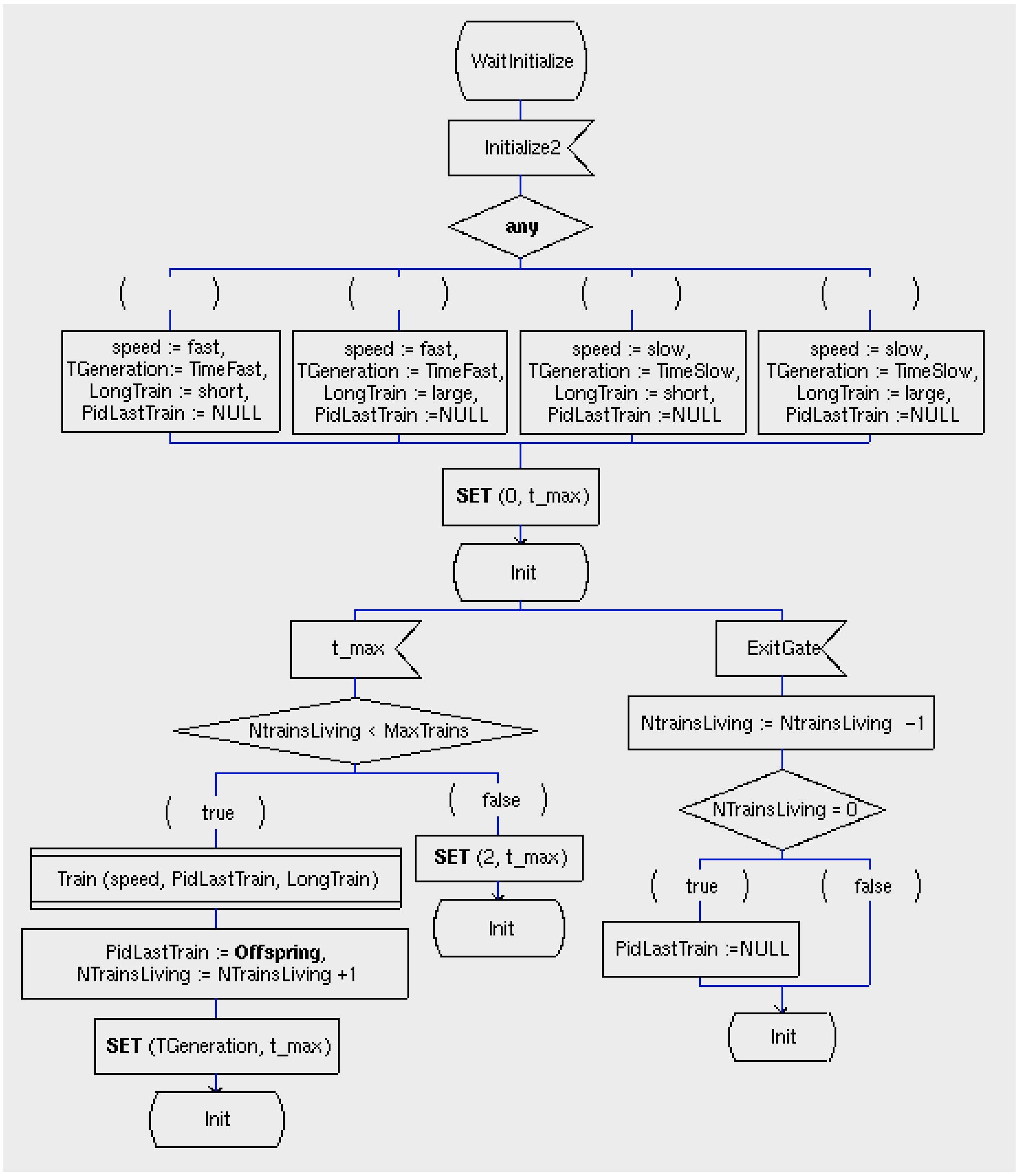

WaitInitialize
Initialize2
any
speed := fast, TGeneration:= TimeFast, LongTrain := short, PidLastTrain := NULL
speed := fast, TGeneration := TimeFast, LongTrain := large, PidLastTrain :=NULL
speed := slow, TGeneration := TimeSlow, LongTrain := short, PidLastTrain :=NULL
speed := slow, TGeneration := TimeSlow, LongTrain := large, PidLastTrain :=NULL
SET (0, t_max)
Init
t_max
ExitGate
NtrainsLiving < MaxTrains
NtrainsLiving := NtrainsLiving -1
NTrainsLiving = 0
true
false
Train (speed, PidLastTrain, LongTrain)
SET (2, t_max)
PidLastTrain := Offspring, NTrainsLiving := NTrainsLiving +1
PidLastTrain :=NULL
SET (TGeneration, t_max)
Init

Figure 5. Scheduler process.

The SchedulerTrains process take into account the number of active train processes (NtrainsLiving) and after each period of time "Tgeneretion" it generates a new train if there are less than MaxTrains (system parameter).

Sensor: The sensor process receives signals from the train and passes them to the controller. The sensors are "located" at the entry point of the trains into the zone of the controller, and at the gate, where the trains leaves the zone of the controller. This process simulates both sensors.

StopTrain: The StopTrain (red light) process, receives signals, "green" or "red", from the controller and changes a local variable for consultation by the train.

Train: The train process is created by the SchedulerTrain with its parameters length, speed and PID of the preceding train. It does the following:

- At creation, it sends a signal to the sensor
- After every time unit:
  1. The speed is regulated such that
     - It does not exceed SpeedMax.
     - The distance to the preceding train (if exists) always allows to stop before reaching its position.
  2. If the traffic light is red, it calculates, based on its position and its speed, if it can stop before de gate. If yes, it stops; otherwise it ignores the traffic light.
- It sends a signal to the sensor, when it leaves the zone controlled by the controller.

Section II. Description of the system in SDL 2000

Using SDL 2000, we would have been able to produce a more elegant solution. SDL 2000 is fully object oriented.

1. It allows parameterised blocks, where parameters can be fixed at initialisation of the system.
   a. In this version we define a single type Track using a non deterministic choice (short or long) at initialisation of each track leading to unnecessary state explosion (in fact such a specification describes not a single instance of a crossing but 2^k, where k is the number of tracks).
   b. A better solution defines two types Track, one for short and one for long Tracks. In SDL 2000, we define single type Track receiving the speed as a parameter at creation
2. SDL 2000 allows to mix blocks and processes in a superblock. It would be more elegant to put the "environment processes" of the gate and each track at the highest level of the Gate, respectively the Track block.
3. Instead of using RPC for getting the values of redlights and sensors, we could have used exported variables. Also in SDL 96, we could have used the export and reveal constructs, but this is not simpler from the point of view of the modelling effort, than RPC.
4. SDL 2000 allows composite states, which would have allowed to define more elegant state machines.

As the system is of a rather static architecture, we did not miss too much the new possibilities of SDL 2000. Also, we did not miss the concept of exception handler, as we use no "dangerous operations". Moreover, in such a safety critical system, it is almost always better to catch possible problems explicitly.

SDL 2000 has the same time semantics as SDL 96, and the same problems, mentioned in section I, occur here to.

Section III Scenarios and properties considered for validation

The model presented so far is highly parameterised, and describes therefore as huge set of possible systems instead of a single system. Moreover, in order to be able to do an exhaustive exploration of the system, we need to restrict also some parameters (such as the possible lengths of trains) which could be (almost) unrestricted in a real system, but keeping them as parameters just introduces a huge state explosion without much added value for verification.

We have made the following restrictions:

1. In the design, the number of tracks is a parameter of the system, nevertheless each instance of a crossing has a fixed number of tracks. We consider systems with one or two tracks.
2. As recommended in the specification we consider two types of trains, fast and slow, each one with a different nominal speed, for which we use fixed values. Moreover, we suppose that the trains always enter with there nominal speed into the crossing
3. Each train has a length, but for validation, all trains have the same length within each scenario. Nevertheless, we consider scenarios with long trains and with short trains.
4. All other restrictions concern the environment: cars and trains are generated in a deterministic and periodic manner. We consider several scenarios with different inter arrival times of cars and trains.

Section IV Verification results.

The commercial SDL tools (like Objectgeode [Ver96], sdt [Ta99], …) provide several development facilities, like editing, code generation and testing. However, they are usually restricted to basic verification techniques (exhaustive simulation, deadlock detection, …)

This situation motivated the use of the IF validation toolbox, able to support:

- translation of the SDL systems to a labelled transition system via an “IF Intermediate Representation Language”.
- many static analysis techniques on the IF intermediate representation language (constant propagation, dead variables, slicing, etc.),
- automatic property checking and
- test case generation.

The verification results are still very preliminary and the full results will be provided at the SAM workshop.

Bibliography

[BFG+99] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, J. Sifakis, *IF: An Intermediate Representation for SDL and its Applications*, Proceedings of SDL-Forum'99 (Montreal, Canada) June 1999.

[BGM01], Marius Bozga, Susanne Graf, and Laurent Mounier, *Automated validation of distributed software using the IF environment*, In Scott D. Stoller and Willem Visser, editors, Workshop on Software Model-Checking, associated with CAV'01 (Paris, France) July 2001 Volume 55 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers.

[HLR92] N. Halbwachs, F. Lagnier and C. Ratel. *Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre*. IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems. September 1992.

[Ta99] Sweden Telelogic A.B., Malmo. *Telelogic TAU SDL suit Reference Manuals*. http://www.telelogic.se, 1999.

[Ver96] Verilog, *OjectGEODE SDL Simulator – Reference Manual* http://www.telelogic.com/products/geode.html 1996.