# System Behavior Models and Verification Term project

**Author: Vladimír Rovenský**

**Date: 25.4.2011**

**Topic: Railway station model in PROMELA / Spin**

## **Assignment**

Model a small system of railway stations in the PROMELA language and verify some of its logical properties using the Spin model checker. The model should cover at least stations, tracks and passengers.

## Model Description

The most significant parts of the model are tracks and trains. The tracks represent the static infrastructure that the trains move on, while the trains are the active part of the model (represented by processes) that holds all the decision logic.

The stations are only represented implicitly (station count is defined, but there are no station objects). The reason for this is that all the data needed is stored in the tracks – the whole station graph is represented as a set of links (tracks). I chose this approach merely because it was easier to implement due to the limited expressivity of PROMELA.

### Tracks

The tracks are defined as follows:

typedef Track {

byte station1;

byte waiting1 = 0;

byte station2;

byte waiting2 = 0;

chan track12 = [TRAIN\_COUNT] of {byte};

chan track21 = [TRAIN\_COUNT] of {byte};

}

First there are the IDs of the stations that the track connects (station1, station2) – these should be self-explanatory.

Then there is the representation of waiting people (waiting1, waiting2). These numbers say how many people are waiting on a train on both ends of the track and they are periodically incremented as new people arrive to the stations.

It is assumed that waiting people just want to go to the opposite station of the track, no further. This may seem limiting, since in reality people often want to travel over several stations, however these variables are here only so we can make sure that nobody has to wait forever at any point, accessibility is modeled elsewhere (allStationsAccessible LTL formula).

The channels model the actual track (trains put their ID in them). There is one channel for each direction and we require that at no point in time are there trains in both channels simultaneously – this would be a crash. This condition is checked by the NoCrashMonitor process.

### Trains

The trains hold all the decision logic of the model. Basically every train has a defined path on which it moves while carrying passengers and avoiding other trains going in the opposite direction. The trains are modeled as processes and there are several constraints on when they can be active.

First, the model keeps track of time in a rudimentary way. The time is partitioned into discrete steps and in every step all the trains are able to perform a single move – either from a station to a track or the other way around. Only once all the trains had a chance to make a move can another time step start. Note that this doesn’t mean that the trains actually move, for example if there is another train coming in the opposite direction, one train has to wait and “skip a turn” (become *idle*).

This simple time is there to ensure basic fairness – we don’t want infinite sequences where only one train is active - that is not how railways work in reality.

Another constraint is that the trains move one at a time, in a specific order that doesn’t change, i.e. not randomly.

This is a simple way to model some kind of dispatching service that the trains use to resolve potential collisions, for instance when two trains want to use one track from opposite stations. To prevent a crash, one train has to go first, the other has to wait. Were the order random, there would always be the case when a train never gets to move, since some other train gets chosen instead.

The specific sequence in which the trains act is as follows (determined by their IDs):   
1, 2 … TRAIN\_COUNT, TRAIN\_COUNT, TRAIN\_COUNT-1 … 2, 1, 0, 0, 1, 2 …

In other words the sequence reverses between time steps (remember that the time step ends when all the trains took their turn), so that the train that had to act last in the previous turn can act first in the current turn.

The conflict resolving could probably be done in a more sophisticated manner that introducing a static order on trains, but this didn’t seem to limiting, so I opted for the easy way.

The third and last constraint on train activation is that in every time step, the PassangerSpawner process has to act first, before all the trains.

#### Train logic

The trains always start in the first station on their path. When a train becomes active (after meeting all the above constraints) it does the following:

If the train is in a station, it looks up which station is next on its path and the track that leads to that station. Than it checks that no other train is coming in the other direction of that track and if not, it loads as many passengers as it can and moves on the track. Otherwise the train becomes idle and waits for next time step.

If the train is not in a station (i.e. it is on a track), it checks whether there is enough room in the station where it is heading for another train. If so, the train moves to that station and unloads all the passengers it has there. Otherwise it becomes idle and waits for some train to leave the station.

### PassangerSpawner

This is a simple process that is guaranteed to run at the beginning of every time step, before all the trains. All it does is spawn some more waiting people on all the track stops. This is to model new people arriving periodically – we can check for example whether the trains are big enough to carry all the waiting people etc.

### Model properties

The model takes into account the following aspects:

* Train fairness (because of the global time).
* Train capacities – trains have a maximum capacity for passengers.
* Station capacities – stations have a maximum capacity for trains.
* New passengers – people arrive in every time step.
* Train collisions – the model makes sure trains do not crash, a train may have to wait for another train to make a move to accomplish this.

On the other hand, I chose not to model the following:

* Path finding – passengers only travel between adjacent stations. I assume that to travel over a more sophisticated route, the passengers choose the right sequence of stations and trains themselves, as is the case in real life. Instead I verify that (1) trains keep visiting all the stations and (2) no one has to wait forever to get on a train. These properties imply that the passengers can always get from one station to any other, provided the railway network is not partitioned into isolated components.
* Train route durations – I assume the distances between stations to be the same. This obviously doesn’t reflect reality, but it doesn’t add much interesting information in terms of verification.
* Train failures, delays, randomness – I felt like PROMELA wasn’t the right tool to work with these, since we typically want to find out how much of a problem we have given specific probability of the undesired event. The verification would just tell us that some property failed because in some run the delay happened, which is obvious and of little value.  
  Also, to include these would mean that the whole system would have to be designed so that the probability of failure (for example someone not being able to make it to their destination) is zero, otherwise the verification would fail. This is way too strong of a requirement on any real system, since achieving this would typically require an unrealistic amount of redundant resources.

### Model checking

I implemented the verification of five properties on the model, three of them as LTL formulae, two of them as processes.

* The noWaitForever LTL formula checks that no passengers have to wait forever for a train.
* The allTrainsMoved LTL formula checks that all the trains keep moving and no train stays idle forever.
* The allStationsVisited LTL formula checks that all the stations are visited by a train.
* The NoCrashMonitor process makes sure that trains never crash.
* The CapacityMonitor process makes sure that the station’s capacities (that is the number of trains in a station) are not breached.

As for the LTL formulae, they check the properties for all relevant objects in the model (all the tracks / trains / stations). This is done by computing some aggregate results at the end of every step – when all the trains finish, the model checks whether all the people were able to get on a train at some point, whether all trains moved at some point and whether all stations were visited at some point.

The formulae are in form “GF p” where p checks one of these aggregate results. Also, to make sure that the condition holds forever, the model periodically resets the data used to calculate the aggregate results – which stations were visited, where there are still people waiting, which trains moved etc. Therefore in order for the formulae to be true, the model has to behave consistently in time - the property has to hold forever, not just once.

The processes are simpler – they just use the assert construct to check the property. They are one pass only, since the random interleaving of processes should guarantee that the verification runs them in all the possible points in time.

## Model configurability

The model comes with a simple system of stations and trains that meets all the defined properties. It is possible to modify this system – to do this one has to change the following:

1. The constants in the very beginning of the file. This is where we set the number of stations, tracks, trains etc. Particularly note the following:
   * TRACK\_COUNT, TRAIN\_COUNT, STATION\_COUNT – set these to the number of respective objects in the system.
   * PASSANGERS\_PER\_STEP – this is the number of new people that arrive to every track stop in every time step. It can be used to find out how much traffic the system can hold – when the noWaitForever formula fails, some track stop has people arriving more quickly that the trains can manage (simulation will tell which).
   * STATION\_CAPACITY, TRAIN\_CAPACITY – the former says how many trains can a station hold, while the latter says how many passengers can a train hold. Both have dramatic impact on the system performance – smaller stations mean more waiting, smaller trains mean less traffic. Having these too low may eventually produce a deadlock (trainsKeepMoving formula fails).
2. The CreateStationRecords() method that actually builds the system. It uses two other methods: AddStop and CreateTrack. The former is used to build train paths – every call adds another stop to a given train. The latter is used to build tracks – every call adds a track between two stations.

The code is thoroughly commented, so the semantics should be clear.

It is very likely that for certain systems you will need to set some Spin parameters to higher values (maximum search depth in particular). Obviously the model is meant for tiny systems, anything resembling a real-life system of railway stations would have way too many states to fit into memory.

## Preconfigured system

For illustration, this is what the preconfigured system of stations and tracks looks like:



There are five stations (circles), four tracks and two trains going around – one follows the route 0-2-4-2, the other 1-2-3-2.