Course 02263 Mandatory Assignment 2, 2016

Ali Chegini (s150939)
Quentin Tresontani (s151409)

May 8, 2016

Contents

1 Introduction

This document contains a data structure and requirements for the design and the validation of a tram net and its related timetables.
The first and second parts of the project are proposing a modelling solution for the net and timetable, discuss their related requirements and present their implementation *NET.rsl* and in *TIMETABLE.rsl*. Last part of the project, present the tests we performed in order to validate our design choices and our requirements realization in RSL.

2 Type definition and requirements for NET

This section defines the type structure of a tram net and its related requirements. Their implementation has been done in the file *NET.rsl* and tested in *testNet.rsl*.

2.1 Structure definition of the tram net

The design we chose in order to describe the net of trams, is really close to the project description. As a consequence, a net is composed of a set of stops and a set of connections.
A stop is composed of an ID and a stop capacity which corresponds to the number of parallel lines in the stops. A connection is then composed of a capacity, a driving time, a headway and a set of stops. Actually, since a connection is relating two stations, the set should be only composed of two elements. However, a connection does not have a direction since a connection is composed of two tracks, one for each direction. Choosing a structure without ordering as a Set was a way to reflect this absence of direction. Checking the number of stops in all connections, is one of our requirements in order to get a well formed net.

In order to represent these structures, we used short records types in RSL. The structure of the NET can be found in the RSL implementation.

2.2 Requirements on the tram nets

The following part specifies requirements on nets. We advise the reader to note the correspondence between the numbering of the requirements in the specification and the one in the comments of the RSL implementation.

2.2.1 Requirements on stops

In order to properly define a net, we need to specify requirements on the net stops. These stops should be unique and their parameters should have acceptable values.

1) Unicity of stops Each stop must be only once in the stop set and different stops must have different names.

First part of the requirements is already solved thanks to the set structure we are using inside the NET in order to store the Stops. In fact, the set structure in RSL does not handle duplicates. This way, this requirement can be sum up to all stops of the net should have different names.

2) Validation of stops The capacity of each stop must be strictly higher than zero.

This requirements is there in order to validate stops. In fact, if we have a stop that has a capacity of 0, this stop would not be able to host a tram and this stop would then be useless.

2.2.2 Requirements on connections

In order to properly define a net, we also need to specify requirements on the net connections. These connections are also concerned by unicity and have to fulfil conditions on their parameters values.

3) Unicity of connections There must be at most one connection relating two stops.

In fact, if we had two connections C1 and C2, it would be the same as having a unique connection C that includes both of them. The values of C should be computed based on the values of C1 and C2.

4) Validation of connections stops Each connection must be relating exactly two stops.

Since we used a set of Stops, having two times the same stop is not possible since RSL sets dont have duplicates. RSL would so transform the set {s1, s1} to {s1} which will not satisfy the number of stops anymore.

5) Validation of connections parameters The headway, minimum driving time and capacity of each connection are strictly higher than zero.

Having a headway of 0 means that two train could start at the same time on the same track which is physically not possible. Having a driving time of zero minutes between two different stations is also not possible. Having a capacity of 0 on a connection means that no tram can be on the tracks of this connection. This connection becomes so useless.

2.2.3 Consistency between stops and connections

There should be some consistency between the stops that are represented explicitly in the set of stops and the stops that are represented implicitly in the connections.

6) Use of stops in the net All stops of the net are used at least used once in the connections.

7) Use of stops in connections All stops used in connections should be defined in the net stops.

Requirement 6 ensures that we dont have useless stops defined inside the net. The requirement 7 ensures that all stops implicitly mentioned inside the connections are also explicitly defined in the stop set.

8) Non nullity The set of stops and of connections cannot be empty.

There should be at least 2 stops in the net and 1 connection. Having an empty net is not an interesting case to deal with.

2.3 RSL implementation of a tram net: NET.rsl

```
scheme NET =
class
        type /* Net type */

                Net :: Stops : Stop-set ↔ new_stops
                   Connections : Connection-set ↔ new_connections,

                /* Auxiliary types */

        Stop :: stopId : StopId
                        stopCapacity : Capacity,

        Connection :: capacity : Capacity
                          drivingTime : DrivingTime
                          headway : Headway
                          Stops : Stop-set,

            /* Simple predefined types */

            Headway = Time, −− minimum headways
            DrivingTime = Time, −− minimum driving times
            Time = Nat, −− times in number of minutes
```

```
    Capacity = Nat, −− capacities
    StopId = Text  −− names of stops

value /∗ generators ∗/

        emptyConnections : Connection-set = {},
        emptyStops : Stop-set = {},
    empty : Net = mk_Net(emptyStops, emptyConnections),  −− the empty net

    −− insert a stop with a given name and capacity
    insertStop  :  StopId × Capacity × Net → Net
    insertStop(id,  cap,  net) ≡
    if  (isIn(id,  net)  = false)
                then new_stops(Stops(net) ∪ {mk_Stop(id, cap)}, net)
                else net
                end,

    −− add a connection between given stops,
    −− with the given capacity, minimum driving time and minimum headway
    addConnection : StopId × StopId  × Headway × Capacity × DrivingTime × Net → Net
    addConnection(id1, id2, hw, cap, dt,  net) ≡
   if  (areDirectlyConnected(id1, id2,  net)  = false)
        then new_connections(
        Connections(net) ∪
       {mk_Connection(cap, dt, hw,{getStop(id1, Stops(net)), getStop(id2, Stops(net))})}
                ,  net)
        else net
        end

value /∗ observers ∗/

    −− check whether a stop is in a network
    isIn  :  StopId × Net → Bool
    isIn(id,  net) ≡
       (∃s  :  Stop • s ∈ Stops(net) ∧ stopId(s)  = id),

    −− check whether two stops are directly connected in a network
    areDirectlyConnected : StopId × StopId × Net → Bool
    areDirectlyConnected(id1, id2,  net) ≡
 (∃c  :  Connection •  c ∈ Connections(net) ∧ areConnectedBy(id1, id2, c)),

    −− get minimum driving time between two connected stops
    minHeadway : StopId × StopId × Net ~→ Headway
    minHeadway(id1, id2, net) ≡
  if  areDirectlyConnected(id1, id2,  net)
```

```
      then headway(getConnection(id1, id2, Connections(net)))
      else 0
      end,

  -- get minimum driving time between two connected stops
  minDrivingTime : StopId × StopId × Net $\xrightarrow{\sim}$ DrivingTime
  minDrivingTime(id1, id2, net) ≡
    if areDirectlyConnected(id1, id2, net)
      then drivingTime(getConnection(id1, id2, Connections(net)))
      else 0
      end,

  -- get the capacity for a connection between two connected stops
  capacity : StopId × StopId × Net $\xrightarrow{\sim}$ Capacity
      capacity(id1, id2, net) ≡
if areDirectlyConnected(id1, id2, net)
      then capacity(getConnection(id1, id2, Connections(net)))
      else 0
      end,

  -- get the capacity of a stop
  capacity : StopId × Net $\xrightarrow{\sim}$ Capacity
  capacity(id, net) ≡
 if isIn(id, net)
      then stopCapacity(getStop(id, Stops(net)))
      else 0
      end,

  /* Auxiliary functions for Observers and Generators */

  /* This function returns a connection out of a connection-set that contains
   * the stop, based on the ID of the two stations the connection is relating.
      * This function does not have any precondition, however it is recommanded
      * to perform the areDirectlyConnected(id1, id2, net) in order to know if the
      * there is a connection linking both stops inside the set or not.
      *
      * params
      *             id1 - ID of the first stop
      *             id2 - ID of the second stop
      *             connections - set of connections where the connection is
      *
      * returns
      *             connection - connection relating the two stations.
      */
  getConnection : StopId × StopId × Connection-set $\xrightarrow{\sim}$ Connection
```

```
getConnection(id1, id2, connections) ≡
    let head = hd connections,
            tail = connections \ {head}
    in
            if areConnectedBy(id1, id2, head)
            then head
            else getConnection(id1, id2, tail)
            end
    end,

/* Function used to know a connection is relating two specific stops described
 * by their IDs.
 *
 * params
 *          id1 − ID of the first station
 *          id2 − ID of the second station
 *          connection − connection we want to test
 *
 * returns
 *          boolean − true, if relating, false if not.
 */
    areConnectedBy : StopId × StopId × Connection → Bool
areConnectedBy(id1, id2, connection) ≡
           (∃s1 : Stop • s1 ∈ Stops(connection) ∧ stopId(s1) = id1 ∧
                   (∃s2 : Stop • s2 ∈ Stops(connection) ∧ stopId(s2) = id2
                           ∧ s1 ≠ s2
                    )
            ),

    /* This function returns a stop out of a stop−set that contains the stop
     * based on his ID.
     * This function does not have any precondition, however it is recommanded
     * to perform the function isIn(id, net) in order to know if the stop is
     * inside or not.
     *
     * params
     *              id − ID of the stop
     *              stops − set of stop where the stop is
     *
     * returns
     *              stop − stop corresponding to the ID
     */
    getStop : StopId × Stop-set ⥲ Stop
  getStop(id, stops) ≡
let head = hd stops,
              tail  = stops \ {head}
        in
              if  stopId(head) = id
              then head
              else getStop(id,  tail )
              end
        end

value /* predicates to check nets */

        isWellformed : Net → Bool
     isWellformed(n) ≡
      unicityNetStops(n) ∧ validateStops(n) ∧
        connectionStops(n) ∧ validateConnections(n)∧
        usageStops(n) ∧ usageConnections(n) ∧
        unicityNetConnections(n),

     /* Auxiliary  functions  expressing  requirements on the net */

     −− 1) Unicity of tram stops
     unicityNetStops : Net → Bool
     unicityNetStops(net) ≡
              ∼(∃s1 : Stop • s1 ∈ Stops(net) ∧
                    (∃s2: Stop • s2 ∈ Stops(net) ∧ s1 ≠ s2 ∧ stopId(s1) = stopId(s2))),

        −− 2) Validity of the parameters of the stops
        validateStops  : Net → Bool
        validateStops(net) ≡
             ∼(∃s : Stop •  s ∈ Stops(net) ∧ stopCapacity(s) < 1),

     −− 3) There is at most one connection between two stops
     unicityNetConnections : Net → Bool
     unicityNetConnections(net) ≡
 ∼(∃c1 : Connection • c1 ∈ Connections(net) ∧
              (∃c2  :  Connection • c2 ∈ Connections(net) ∧
                    Stops(c1) = Stops(c2) ∧ c1 ≠ c2)),

     −− 4) All connections should have exactly two stops in  their  stop−set
     connectionStops : Net → Bool
     connectionStops(net) ≡
              ∼(∃c : Connection • c ∈ Connections(net) ∧ card Stops(c) ≠ 2),

        −− 5) Validity of the parameters of connections
        validateConnections : Net → Bool
        validateConnections(net) ≡
              ∼(∃c : Connection • c ∈ Connections(net) ∧
                    (capacity(c)  < 1 ∨ headway(c) < 1 ∨ drivingTime(c) < 1)),

      −− 6) All stops of the net are  least  used once in  a connection
usageStops :  Net → Bool
usageStops(net) ≡
 ∼(∃s : Stop •  s ∈ Stops(net)  ∧ ∼stopUsed(s, net)),

stopUsed : Stop × Net → Bool
stopUsed(s, net) ≡
(∃c  :  Connection • c ∈ Connections(net) ∧ (s ∈ Stops(c ))),

−− 7) All stops used in connections are  defined  in  the  net  stops
usageConnections : Net → Bool
usageConnections(net) ≡
  ∼(∃c : Connection • c ∈ Connections(net) ∧ ∼connectionUsed(c, net)),

connectionUsed : Connection × Net → Bool
connectionUsed(c, net) ≡
  ∼(∃s : Stop •  s ∈ Stops(c)  ∧ (s ∈ Stops(net))  = false)

end
```

3 Type definition and requirements for TIMETABLE

This section defines the type structure of a net time table and its related requirements. Their implementation has been done in the file *TIMETABLE.rsl* and tested in *testTimeTable.rsl.*

3.1 Structure definition of the net time table

The definition of the structure of the time table is a bit more complex than the one of the net. Different possibilities have been discussed.

First approach A plan can be seen as a set of stops: for instance train stayed from 0 to 1 in A, then stayed in B from 12 to 14 which assumes implicitly that the train was travelling from A to B during 1 and 12 since there are no other time in station between them.

Second approach A plan can also be seen as a set of trips: for instance train travelled from A to B, from 1 to 12 and then from B to C from 14 to 28 which assumes implicitly that the train was waiting in B from 12 to C since there are no other trips at that moment.

Third approach A way to mix both approaches is to create a structure considering time of travelling and time of stay before the travel. Structure would then be based on plan element described as following:

1. Arrival time in departure stop
2. Departure time at departure stop
3. Arrival time at arrival stop
4. Departure station
5. Arrival station

Discussion and final decision The third proposition would be the easiest one to deal with while performing tests on the requirements. In fact, we have some overlapping in the information of two consecutive plan element since the arrival time at the arrival stop of one trip is the arrival time in departure stop of the next one. This overlapping would help us finding the next trips in the set by having a fully explicit information.
However it would create some troubles while using the generators template provided in the project description. For instance, using the template of the function to add a stop *addStop* is taking as parameters, a tram ID, a stop ID, an arrival time and a departure time, whereas the data type would also need an other stop ID, and another arrival time if we were using the third approach. We would then having unknown values in our data structure when creating the

stop by using the function.
Considering the functions provided in the template, the best solution for us is to consider the **first approach** that perfectly matches the functions input even if it might include more work during the requirement specification.

Data structure In order to define a time table, we have been using set and short records. A time table is so seen as a set of plans. And each plan is a short record bringing together a tram ID of type text and a set of trips. A trip is a short record combining a station ID, an arrival time and a departure time. We choose to use set and not lists in the plan, since relying on a list would mean to trust the order in which the single element are added to the list. However, we could imagine having a plan where the ordering of the elements is not reflecting the chronological one. Each plan is so composed of a set of trips and methods should be defined in order to know their ordering based on their arrival and departure times.

3.2 Requirements specification on the net time table

The following part specifies requirements on time tables. We advise the reader to note the correspondence between the numbering of the requirements in the specification and the one in the comments of the RSL implementation.

Most of the requirements are checked by contradiction. Instead of saying that a requirement is true for all elements, we check that it doesn't exist element that is not respecting the requirement.

3.2.1 Requirements on the plans and trips

1) Trip times ordering Arrival times should be strictly before the departure times.

The arrival time of each tram in a specific station should be strictly lower than the departure time. We assume that each tram stops in the stations they go through.

The test is performed directly on the values of each trip inside each plan using nested loops.

2) Trip times values All time values must be between zero and sixty.

This requirement is made in order to have values corresponding to the project description.

The test is performed directly on the values of each trip inside each plan using nested loops.

3) Unicity of plans There should be exactly one plan per tram.

Trams are implicitly defined using the tram ID in the plans of the time table. We need to make sure that there is only one plan for each tram.

This requirement has been tested by checking that it does not exist two plans with the same tram ID.

4) Presence of trips There should be at least two trips in each plan.

Since trips, are defining the period where a train is waiting at a specific stop, we need two of our trip objects to define a real trip between two stops. Having one or zero stops inside the plan does not add any value to the tram system.

The test is performed directly on each plan of the time table by checking the value of the cardinal of the trips set related to the current plan.

5) Non nullity of values There should not be any empty values inside the trips and plans.

This requirement is done in order to avoid having empty fields. Having empty fields could be a problem in order to check further requirements and to identify some elements. It means that all stop name should be defined inside the trips (note that this is covered by requirements 6 and 7) and that the tram name should be defined too.

The test is performed directly on the values of each trip and the tramID inside each plan using nested loops.

3.2.2 Consistency with net stops

6) Stop definition in net All stops used inside the time table should also be defined in the stops of the related net.

This requirement is made in order to avoid inconsistency between the time table and the net. Having a stop in the time table that is not defined in the net would prevent us from checking the other requirements including the one on stop capacity.

An auxiliary function has been defined in order to check if the stops of a plan are defined inside the net. The requirement is then made as a loop checking this stop definition on the plans of the time table.

7) Usage of stops in time table All stops of the net should be used at least once in the time table.

This requirement can be seen with two different purposes. On the one hand it has been introduced in order to avoid having a net with useless stops. It can also be seen, on the other hand, as avoiding to forget providing stops with trams while making the time table.

This requirement is directly tested by using a loop on the stops defined inside the net. If one of them, is not used in the timetable, then the requirement is not satisfied.

8) Respect of stop capacity There should not be more trains at the same time in one stop according to the time table than the capacity of this stop allows it.

This requirement is done in order to have a time table following the basic constraints of net definition.

In order to achieve this requirement, we created a function returning the set of trams at a given stop at a given time in a timetable. By using the cardinal of this set and comparing it to the capacity of the stop, we can check if the requirement is respected.

3.2.3 Consistency with net connections

9) Connection definition in net All connections implicitly defined in the time tables should exist in the net.

This requirement is made in order to avoid inconsistency between the time table and the net. Having a connection in the time table that is not defined in the net would prevent us from checking the requirements on capacities, on headway and on driving time.

In order to achieve this requirement, we are checking that he does not exist any plan which as a trip with a trip not defined in the net. A specific auxiliary function has been defined.

10) Usage of connections All connections of the net should be used at least once in the time table.

This requirement could be used in order to enforce the consistency between the net and the time table. It would help us to be sure that we are not defining useless connections inside the net. However, this requirement is not necessary. It is important for a tram station to go to all the stations (as expressed in requirement 7). Having unused connections would not be a problem, for the customer of the network if all other requirements are still satisfied. These unused connections can be analyzed in two ways. It could be a good a thing for the net, if we consider that it would facilitate the creation of new lines of trams.

It can also be a bad one, if we consider the net has been poorly designed since having useless connections means to have paid for something not used.

As a consequence, this requirement has not been implemented.

11) Respect of the connection minimum driving time There should not be any arrival time of the next step that is less than the minimum driving time to get there starting from the step before.

This requirement is done in order to have a time table following the basic constraints of net definition.

In order to achieve this requirement some auxiliary functions were needed. A function returning the next trip of a given trip in a given plan has been created. Using this function we can check that the difference between departure time of the trip and the time of the next one in the plan is bigger than the connection minimum driving time.

12) Respect of connection capacity There should not be more trains on a connection according to the time table than the capacity of the connection allows it.

This requirement is done in order to have a time table following the basic constraints of net definition.

A function returning the set of trams on a specific connection between two given stops at a given time has been created. Using the cardinal of this set and of the set created when permuting the two input stops, we have the number of trains on the connection. Then, we can check the result of the addition of the two cardinals and compare it to the connection capacity.

13) Respect of connection headway There should not be trains starting from the same station on the same connection within a time lapse shorter than the headway of the connection they are on.

This requirement is done in order to have a time table following the basic constraints of net definition.

In order to validate this requirement, we created a function that is returning the next trip in the timetable starting from the same stop as the given trip and going into the same direction. We then have been able to compute the difference between the departure time of these two trips and to compare it with the minimum headway of the connection.

3.3 RSL implementation of a net time table: TIMETABLE.rsl

```
NET
scheme TIMETABLE = extend NET with
class
        type /∗ Time table type ∗/

        TimeTable = Plan-set,

        /∗ Auxiliary types ∗/

        Plan:: tramId : TramId
            trips  : Trip-set ↔ new_trips,

        −− actually represents when the tram is waiting in a station
        Trip :: stationId  :StopId
                    arrivalTime : Time
                    departureTime: Time,

        /∗ Simple predefined types ∗/

        TramId = Text −− tram names

        value /∗ generators ∗/

        −− the empty timetable
        empty: TimeTable = {},

        −− add to a time table an empty plan for a new tram
        addTram: TramId × TimeTable → TimeTable
        addTram(trId, tt) ≡
        {mk_Plan(trId, {})} ∪ tt,

        −− add a stop with arrival time and departure time to the plan for a given tram
        addStop: TramId × StopId × Time × Time × TimeTable → TimeTable
        addStop(trId, sId, at, dt, tt) ≡
  if isIn (trId, tt)
    then let
                        plan = getPlanByTramId(trId, tt),
                        tt = tt \ {plan}
                in
                        {new_trips({mk_Trip(sId, at, dt)} ∪ trips (plan), plan)} ∪ tt
                end
        else {mk_Plan(trId, {mk_Trip(sId, at, dt)})} ∪ tt
        end
```

```
value /∗ observers ∗/

−− check whether a tram with a given name exists in a given time table
isIn : TramId × TimeTable → Bool
isIn(trId, tt) ≡
  (∃plan : Plan • plan ∈ tt ∧ tramId(plan) = trId),

−− check whether a stop with a given name exists in a given time table
stopIsIn : StopId × TimeTable → Bool
stopIsIn(id, t) ≡
 (∃p : Plan • p ∈ t ∧
              (∃t : Trip • t ∈ trips(p) ∧ stationId(t) = id)),

/∗ Auxiliary functions for the Observers and Generators ∗/

        getPlanByTramId: TramId × TimeTable → Plan
        getPlanByTramId(trId, t) ≡
               let
                        head = hd t,
                tail  = t \ {head}
        in
                if trId  = tramId(head)
                then head
                else if  tail  ≠ {}
                                then getPlanByTramId(trId, tail)
                                else mk_Plan("", {})
                                end
                end
                end

 value /∗ predicates to check time tables ∗/

        isWellformed : TimeTable × Net → Bool
isWellformed(t, n) ≡
      requirementOne(t) ∧ requirementTwo(t) ∧
        requirementThree(t) ∧ requirementFour(t) ∧
        requirementFive(t) ∧ requirementSix(t, n) ∧
        requirementSeven(t, n) ∧ requirementEight(t, n) ∧
        requirementNine(t, n) ∧ requirementTen(t, n) ∧
        requirementEleven(t, n) ∧ requirementTwelve(t, n) ∧
        requirementThirteen(t, n) ∧ isWellformed(n),

        −− 1) The arrival time should be strictly before the departure time
requirementOne: TimeTable → Bool
requirementOne(t) ≡
             ∼(∃plan: Plan • plan ∈ t ∧
                    (∃trip:Trip • trip ∈ trips(plan) ∧
                        arrivalTime(trip) > departureTime(trip))),

−− 2) All values should be between 0 and 60
requirementTwo: TimeTable → Bool
requirementTwo(tt) ≡
    ∼(∃plan : Plan • plan ∈ tt ∧
            (∃trip : Trip • trip ∈ trips(plan) ∧
                (arrivalTime(trip) > 60 ∨ departureTime(trip) > 60))),

−− 3) There should be only one plan per tram
requirementThree: TimeTable → Bool
requirementThree(t) ≡
    ∼(∃p1 : Plan • p1 ∈ t ∧
            (∃p2: Plan • p2 ∈ t
                ∧ p1 ≠ p2 ∧ tramId(p2) = tramId(p1))),

−− 4) There should be at least two trips in each plan
requirementFour : TimeTable → Bool
requirementFour(t) ≡
     ∼(∃p : Plan • p ∈ t ∧ card trips(p) < 2),

−− 5) There should not be any empty values in plans and trips
requirementFive : TimeTable → Bool
requirementFive(t) ≡
     ∼(∃p: Plan • p ∈ t ∧ (tramId(p) = "" ∨
            (∃trip : Trip • trip ∈ trips(p)
                        ∧ stationId(trip) = ""))),

−− 6) All stops of the time table should be defined in the net
requirementSix : TimeTable × Net → Bool
requirementSix(t, n) ≡
    ∼(∃p : Plan • p ∈ t ∧ ∼stopDefinition(p, n)),

−− All stops of the trips of the plan are defined in the net
stopDefinition : Plan × Net → Bool
stopDefinition(p, n) ≡
    ∼(∃t : Trip • t ∈ trips(p) ∧ ∼isIn(stationId(t), n)),

−− 7) All stops of the net are used at least once in the time table
requirementSeven : TimeTable × Net → Bool
requirementSeven(t, n) ≡
  ∼(∃s : Stop • s ∈ Stops(n) ∧ ∼stopIsIn(stopId(s), t)),

−− 8) Respect of stop capacity
requirementEight : TimeTable × Net → Bool
```

```
requirementEight(t, n) ≡
  ∼(∃s : Stop • s ∈ Stops(n) ∧
                    (∃time : Time • time ∈ {0..60} ∧
                              (card getWaitingTrams(t, stopId(s), time) > capacity(stopId(s), n )))),

−− 9) All connections implicitely defined in the time table are also defined
−− in the net, last trip is not checked since same station than first one
requirementNine : TimeTable × Net → Bool
requirementNine(t, n) ≡
   ∼(∃p : Plan • p ∈ t ∧
                    (∃trip : Trip • trip ∈ trips (p) ∧
                              ∼areDirectlyConnected(stationId(trip), stationId (getNextTrip(p, trip )),
                              ∧ departureTime(trip) ≠ getLastTripDepartureTime(trips(p), 0))),

−− 10) All connections of the net are used at least once in the time table
requirementTen : TimeTable × Net → Bool
requirementTen(t, n) ≡
    true, −−Not implemented, see discussion in the report.

−− 11) Respect of connection driving time
requirementEleven : TimeTable × Net → Bool
requirementEleven(t, n) ≡
 ∼(∃p : Plan • p ∈ t ∧
                    (∃trip : Trip • trip ∈ trips (p) ∧ ∼checkMinDrivingTime(trip, p, n))),

checkMinDrivingTime : Trip × Plan × Net → Bool
checkMinDrivingTime(trip, p, n) ≡
if departureTime(trip) ≠ 60
         then minDrivingTime(stationId(trip) , stationId(getNextTrip(p, trip )), n)
                              < (arrivalTime(getNextTrip(p, trip))− departureTime(trip))
         else true
         end,

−− 12) Respect of connection capacity
requirementTwelve : TimeTable × Net → Bool
requirementTwelve(t, n) ≡
∼(∃time : Time • time ∈ {0..60} ∧
                    (∃c : Connection • c ∈ Connections(n)
                              ∧ ∼checkCapacity(c, t, time, n ))),

checkCapacity : Connection × TimeTable × Time × Net → Bool
checkCapacity(c, t , time, n) ≡
    let head = hd Stops(c),
                    tail = hd (Stops(c) \ {head}),
                    id1 = stopId(head),
                    id2 = stopId( tail )
```

```
        in (card getTramsOn(t, id1, id2, time) + card getTramsOn(t, id2, id1, time))
                < capacity(id1, id2, n)
        end,

−− 13) Respect of connection headway
requirementThirteen : TimeTable × Net → Bool
requirementThirteen(t, n) ≡
      ∼(∃p : Plan • p ∈ t ∧
                (∃trip : Trip • trip ∈ trips(p) ∧ ∼checkHeadway(p, trip, t, n))),

−− Return true if the trip and its next departure in the right direction according
−− to the time table and the net are respecting the headway of the connection
checkHeadway : Plan × Trip × TimeTable × Net → Bool
checkHeadway(plan, trip, t, n) ≡
  let station = stationId(trip),
                nextStation = stationId(getNextTrip(plan, trip)),
                (nPlan, nTrip) = nextDeparture(station, nextStation, departureTime(trip), t),
                dtime = departureTime(nTrip),
                headway = dtime − departureTime(trip)
        in (minHeadway(station, nextStation, n) < headway)
        end,

/∗ Auxiliary functions used in order to get the next trip of a given trip in a plan ∗/

/∗ GET NEXT TRIP FUNCTION ∗/

−− Get the next trip of a specific trip in a plan using the arrivalTime
getNextTrip : Plan × Trip → Trip
getNextTrip(p, trip) ≡
    getTripByATime(trips(p), getNextTripArrivalTime(trips(p), trip)),

−− Get the arrival time of a trip
getNextTripArrivalTime : Trip-set × Trip → Time
getNextTripArrivalTime(p, t) ≡
   if (departureTime(t) = getLastTripDepartureTime(p, 0)) −− if last trip of the plan
        then getFirstTripArrivalTime(p, 61) −− then return the first one
        else getNextTripLoop(p, t, 61) −− else return the right one
                end,

        −− Return arrival time of the first trip
        getFirstTripArrivalTime : Trip-set × Time → Time
        getFirstTripArrivalTime(p, best) ≡
                if p ≠ {}
                then let head = hd p,
                         tail = p \ {head}
                     in getFirstTripArrivalTime(tail, smallestAT(head, best))
```

```
                  end
              else best
              end,

          −− Return the smallest of two values
          smallestAT : Trip × Time → Time
          smallestAT(head, best) ≡
               if arrivalTime(head) < best
              then arrivalTime(head) else best
              end,

          −− Return arrival time of the first trip
          getLastTripDepartureTime : Trip-set × Time → Time
          getLastTripDepartureTime(p, best) ≡
             if p ≠ {}
              then let head = hd p,
                       tail = p \ {head}
                  in getLastTripDepartureTime(tail, biggestDT(head, best))
                  end
              else best
              end,

          −− Return the biggest of two values
          biggestDT : Trip × Time → Time
          biggestDT(head, best) ≡
            if departureTime(head) > best
              then departureTime(head) else best
              end,

          −− Get trip from a set trip using its arrival time
     getTripByATime : Trip-set × Time → Trip
     getTripByATime(p, aTime) ≡
        let head = hd p,
               tail = p \ {head}
          in    if arrivalTime(head) = aTime
              then head
              else getTripByATime(tail, aTime)
              end
          end,

     −− Function used in order to get the next trip of a given trip in a time table
     getNextTripLoop : Trip-set × Trip × Time → Time
     getNextTripLoop(p, t, best) ≡
        if p ≠ {}
          then let head = hd p,
                      tail = p \{head}
```

```
                in insideFunction(t, head, tail, best)
                end
        else best
        end,

-- Function used with getNextTripLoop
insideFunction : Trip × Trip × Trip-set × Time → Time
insideFunction(t, head, tail, best) ≡
     if (departureTime(t) < arrivalTime(head) ∧ arrivalTime(head) < best)
        then getNextTripLoop(tail, t , arrivalTime(head))
        else getNextTripLoop(tail, t , best)
        end,

/* FUNCTION USED IN ORDER TO GET TRAMS WAITING IN A STOP */

-- Returns the set of trams waiting in one stop at a time t from a time table
getWaitingTrams : TimeTable × StopId × Time → Trip-set
getWaitingTrams(t, stopId, time) ≡
       let trips = getAllTrips(t) -- get all trips of time table
        in getWaitingTramsInit(trips, stopId, {}, time)
        end,

-- Returns the list of all trips in the time table
getAllTrips : TimeTable → Trip-set
getAllTrips(t) ≡
  let head = hd t,
              tail = t \ {head}
        in if tail ≠{}
              then trips(head) ∪ getAllTrips(tail)
              else trips(head)
              end
        end,

-- get the set of trams waiting in a stop at a time t from a trip-set
getWaitingTramsInit : Trip-set × StopId × Trip-set × Time → Trip-set
getWaitingTramsInit(tripset, stopId, result, time) ≡
      if tripset ≠{}
        then let
                      head = hd tripset,
                      tail = tripset \ {head}
              in insideGetWaitingTrams(head, tail, stopId, result, time)
              end
        else result
        end,

-- function used for the loop of getWaitingTramsInit
```

```
insideGetWaitingTrams : Trip × Trip-set × StopId × Trip-set × Time → Trip-set
insideGetWaitingTrams(head, tail, stopId, result , time) ≡
 if stationId(head) = stopId
            ∧ (arrivalTime(head) ≤ time)
            ∧ (time ≤ departureTime(head))
      then getWaitingTramsInit(tail, stopId, result ∪ {head}, time)
      else getWaitingTramsInit(tail, stopId, result , time)
      end,

/* FUNCTION USED IN ORDER TO GET TRAMS ON A CONNECTION */

−− Returns the list of trams on a connection at a given time in a time table
getTramsOn : TimeTable × StopId × StopId × Time → Trip-set
getTramsOn(t, id1, id2, time) ≡
  let head = hd t,
             tail = t \ {head}
      in if tail ≠ {}
             then getTramsInPlan(head, trips(head), id1, id2, time, {})
                   union getTramsOn(tail, id1, id2, time)
             else getTramsInPlan(head, trips(head), id1, id2, time, {})
             end
      end,

−− Returns the list of trams on a connection at a given time in a plan
getTramsInPlan : Plan × Trip-set × StopId × StopId × Time × Trip-set → Trip-set
getTramsInPlan(p, trips, id1, id2, time, result ) ≡
      if trips ≠ {}
      then let head = hd trips,
                    tail = trips \ {head}
             in insideGetTramsInPlan(p, tail, id1, id2, time, result , head)
             end
      else result
      end,

−− loop function used to getTramsInPlan
insideGetTramsInPlan : Plan × Trip-set × StopId × StopId × Time
                                                × Trip-set × Trip → Trip-set
insideGetTramsInPlan(p, trips, id1, id2, time, result , head) ≡
   if (stationId(head) = id1) ∧ (stationId(getNextTrip(p, head)) = id2)
            ∧ (departureTime(head) < time)
            ∧ (time < arrivalTime(getNextTrip(p, head)))
      then getTramsInPlan(p, trips, id1, id2, time, result ∪ {head})
      else getTramsInPlan(p, trips, id1, id2, time, result )
      end,

/* FUNCTIONS DEFINING THE NEXT DEPARTURE IN ONE TIME TABLE */
```

```
−− Return the next departure from a given stopId to another one at a given time
−− in a given timetable
nextDeparture : StopId × StopId × Time × TimeTable → (Plan × Trip)
nextDeparture(id1, id2, time, t) ≡
        nextDepartureLoop(id1, id2, time, t, mk_Plan("", {}), mk_Trip("", 61, 61)),

−− loop function used for nextDeparture
nextDepartureLoop : StopId × StopId × Time × TimeTable × Plan × Trip → (Plan × Trip)
nextDepartureLoop(id1, id2, time, t, bestPlan, bestTrip ) ≡
       let head = hd t,
               tail = t \ {head},
               emptyTrip = mk_Trip("", 61, 61),
               formerBestTrip = bestTrip,
               challengerTrip = bestDepartureOfPlan(head, trips(head), time, id1, id2,
                              emptyTrip),
               bestTrip = smallestDT(challengerTrip, bestTrip),
               bestPlan = updateBestPlan(head, bestTrip, bestPlan, formerBestTrip)
        in if tail ≠ {}
               then nextDepartureLoop(id1, id2, time, tail, bestPlan, bestTrip)
               else (bestPlan, bestTrip)
               end
        end,

−− returns the plan related to the best trip
updateBestPlan : Plan × Trip × Plan × Trip → Plan
updateBestPlan(plan, trip, fPlan, fTrip) ≡
        if (departureTime(fTrip) = departureTime(trip))
        then fPlan
        else plan
        end,

−− returns the bestDepartureTrip of a trip set
bestDepartureOfPlan : Plan × Trip-set × Time × StopId × StopId × Trip → Trip
bestDepartureOfPlan(plan, p, time, id1, id2, best) ≡
       let head = hd p,
               tail = p \ {head},
               best = selectTrip(best, head, time, id1, id2, plan)
        in     if tail ≠ {}
               then −− continue recursion
                      bestDepartureOfPlan(plan, tail, time, id1, id2, best)
               else −− stop recursion
                      best
               end
        end,
    selectTrip : Trip × Trip × Time × StopId × StopId × Plan → Trip
    selectTrip (best, challenger, time, id1, id2, plan) ≡
       if (time < departureTime(challenger))
              ∧ (departureTime(challenger) < departureTime(best))
              ∧ (stationId(challenger) = id1)
              ∧ (stationId(getNextTrip(plan, challenger)) = id2)
        then challenger
        else best
        end,

    smallestDT : Trip × Trip → Trip
    smallestDT(trip1, trip2) ≡
        if (departureTime(trip1) ≤ departureTime(trip2))
        then trip1
        else trip2
        end
end
```

4 Testing by translation to SML

4.1 Test specifications

4.1.1 Test specifications for NET

In order to perform tests on our methods, we created a class dedicated to perform tests on NET. This class is composed of three different parts. The first tests are dealing with observers of NET and their auxiliary methods. The second part focuses on tests on the two generators of NET. The third part focuses on tests on the predicate and its auxiliary methods.

Each method is tested following this pattern, in order to cover all possibles cases:

1. a test where the output should be a success is performed on the tram net presented in the project description
2. a test where the function should be failing is performed on a specific input designed for the test
3. eventually a third test in performed to show a particular behaviour of the function or a tricky case

Each test is commented in order to explain the expected output. Each test is returning a Boolean, if a test succeed to reach the expected behaviour of the tested method, then the test should return true.

NET

```
/* The following class is composed of a set of unit tests made on functions used
 * in the Scheme NET.
 */
scheme testNET =
extend NET with
        class
                value
                        --Example of the project description for a well defined net
                        basicExample : Unit → Net
                        basicExample() ≡
                         let a = mk_Stop("A", 2),
                                       b = mk_Stop("B", 3),
                                       c = mk_Stop("C", 2),
                                       d = mk_Stop("D", 2),
                                       c1 = mk_Connection(3, 10, 2, {a, b}),
                                       c2 = mk_Connection(3, 12, 4, {b, c}),
                                       c3 = mk_Connection(2, 7, 4, {b, d})
                                in     mk_Net({a, b, c, d}, {c1, c2, c3})
                                end
```

```
          test_case

−−1) Tests on obeservers and auxiliary methods

          [NET_1]
                  ”1) Tests on the observers and their auxiliary functions”,

          /∗ Should return true because b is in the net ∗/

          [IsinStop_1]
                  isIn(”B”, basicExample()) = true,

          /∗ Should return false because s2 is not in the net ∗/

          [IsinStop_2]
                  let s1 = mk_Stop(”Stop1”, 2),
                          s2 = mk_Stop(”Stop1”, 2),
                          c = mk_Connection(1, 1, 1, {s1}),
                          net = mk_Net({s1}, {c})
                  in      isIn(”Stop2”, net) = false
                  end,

          /∗ Should return true since they are connected by c ∗/

          [AreConnectedBy_1]
                  let a = mk_Stop(”A”, 2),
                          b = mk_Stop(”B”, 3),
                          c = mk_Stop(”C”, 2),
                          d = mk_Stop(”D”, 2),
                          c1 = mk_Connection(3, 10, 2, {a, b}),
                          c2 = mk_Connection(3, 12, 4, {b, c}),
                          c3 = mk_Connection(2, 7, 4, {b, d}),
                          net = mk_Net({a, b, c, d}, {c1, c2, c3})
                  in      areConnectedBy(”B”, ”C”, c2) = true
                  end,

          /∗ Should return true since they are connected by c ∗/

          [AreConnectedBy_2]
                  let s1 = mk_Stop(”Stop1”, 2),
                          s2 = mk_Stop(”Stop2”, 2),
                          c = mk_Connection(1, 1, 1, {s1, s2})
                  in      areConnectedBy(”Stop1”, ”Stop3”, c) = false
                  end,
          /* Should return true since they are directly connected */

          [AreDirectlyConnected_1]
                  let a = mk_Stop("A", 2),
                          b = mk_Stop("B", 3),
                          c = mk_Stop("C", 2),
                          d = mk_Stop("D", 2),
                          c1 = mk_Connection(3, 10, 2, {a, b}),
                          c2 = mk_Connection(3, 12, 4, {b, c}),
                          c3 = mk_Connection(2, 7, 4, {b, d}),
                          net = mk_Net({a, b, c, d}, {c1, c2, c3})
                  in      areDirectlyConnected("A", "B", net) = true
                  end,

          /* Should return false since they are not directly connected */

          [AreDirectlyConnected_2]
                  let s1 = mk_Stop("Stop1", 2),
                          s2 = mk_Stop("Stop2", 2),
                          c = mk_Connection(1, 1, 1, {s1, s2}),
                          net = mk_Net({s1, s2}, {c})
                  in      areDirectlyConnected("Stop1", "Stop3", net) = false
                  end,

          /* Should return the connection c2 */

          [GetConnection]
                  let a = mk_Stop("A", 2),
                          b = mk_Stop("B", 3),
                          c = mk_Stop("C", 2),
                          d = mk_Stop("D", 2),
                          c1 = mk_Connection(3, 10, 2, {a, b}),
                          c2 = mk_Connection(3, 12, 4, {b, c}),
                          c3 = mk_Connection(2, 7, 4, {b, d}),
                          net = mk_Net({a, b, c, d}, {c1, c2, c3})
                  in      getConnection("B", "C", Connections(net)) = c2
                  end,

          /* Should return the min headway of the connection c2: 4 */

          [MinHeadway_1]
                  let a = mk_Stop("A", 2),
                          b = mk_Stop("B", 3),
                          c = mk_Stop("C", 2),
                          d = mk_Stop("D", 2),
                          c1 = mk_Connection(3, 10, 2, {a, b}),
                    c2 = mk_Connection(3, 12, 4, {b, c}),
                    c3 = mk_Connection(2, 7, 4, {b, d}),
                    net = mk_Net({a, b, c, d}, {c1, c2, c3})
              in minHeadway("B", "C", net) = 4
              end,

          /* Should return 0 since there is no connection between these stops */

          [MinHeadway_2]
              let a = mk_Stop("A", 2),
                    b = mk_Stop("B", 3),
                    c = mk_Stop("C", 2),
                    d = mk_Stop("D", 2),
                    c1 = mk_Connection(3, 10, 2, {a, b}),
                    c2 = mk_Connection(3, 12, 4, {b, c}),
                    c3 = mk_Connection(2, 7, 4, {b, d}),
                    net = mk_Net({a, b, c, d}, {c1, c2, c3})
              in minHeadway("A", "C", net) = 0
              end,

          /* Should return the min driving time of the connection c2: 3 */

          [MinDrivingTime_1]
              let a = mk_Stop("A", 2),
                    b = mk_Stop("B", 3),
                    c = mk_Stop("C", 2),
                    d = mk_Stop("D", 2),
                    c1 = mk_Connection(3, 10, 2, {a, b}),
                    c2 = mk_Connection(3, 12, 4, {b, c}),
                    c3 = mk_Connection(2, 7, 4, {b, d}),
                    net = mk_Net({a, b, c, d}, {c1, c2, c3})
              in minDrivingTime("B", "C", net) = 12
              end,

          /* Should return 0 since there is no connection between these stops */

          [MinDrivingTime_2]
              let a = mk_Stop("A", 2),
                    b = mk_Stop("B", 3),
                    c = mk_Stop("C", 2),
                    d = mk_Stop("D", 2),
                    c1 = mk_Connection(3, 10, 2, {a, b}),
                    c2 = mk_Connection(3, 12, 4, {b, c}),
                    c3 = mk_Connection(2, 7, 4, {b, d}),
                    net = mk_Net({a, b, c, d}, {c1, c2, c3})
              in minDrivingTime("A", "C", net) = 0
```

```
          end,

/∗ Should return the capacity of the connection c2: 3 ∗/

[Capacity_1]
          let a = mk_Stop("A", 2),
                    b = mk_Stop("B", 3),
                    c = mk_Stop("C", 2),
                    d = mk_Stop("D", 2),
                    c1 = mk_Connection(3, 10, 2, {a, b}),
                    c2 = mk_Connection(3, 12, 4, {b, c}),
                    c3 = mk_Connection(2, 7, 4, {b, d}),
                    net = mk_Net({a, b, c, d}, {c1, c2, c3})
          in  capacity("B", "C", net) = 3
          end,

/∗ Should return 0 since there is no connection between these stops ∗/

[Capacity_2]
          let a = mk_Stop("A", 2),
                    b = mk_Stop("B", 3),
                    c = mk_Stop("C", 2),
                    d = mk_Stop("D", 2),
                    c1 = mk_Connection(3, 10, 2, {a, b}),
                    c2 = mk_Connection(3, 12, 4, {b, c}),
                    c3 = mk_Connection(2, 7, 4, {b, d}),
                    net = mk_Net({a, b, c, d}, {c1, c2, c3})
          in  capacity("A", "C", net) = 0
          end,

/∗ Should return the stop c ∗/

[GetStop]
          let a = mk_Stop("A", 2),
                    b = mk_Stop("B", 3),
                    c = mk_Stop("C", 2),
                    d = mk_Stop("D", 2),
                    c1 = mk_Connection(3, 10, 2, {a, b}),
                    c2 = mk_Connection(3, 12, 4, {b, c}),
                    c3 = mk_Connection(2, 7, 4, {b, d}),
                    net = mk_Net({a, b, c, d}, {c1, c2, c3})
          in        getStop("C", Stops(net)) = c
          end,

/∗ Should return the capacity of the stop C: 2 ∗/
```

```
        [Stop_Capacity_1]
                let a = mk_Stop("A", 2),
                        b = mk_Stop("B", 3),
                        c = mk_Stop("C", 2),
                        d = mk_Stop("D", 2),
                        c1 = mk_Connection(3, 10, 2, {a, b}),
                        c2 = mk_Connection(3, 12, 4, {b, c}),
                        c3 = mk_Connection(2, 7, 4, {b, d}),
                        net = mk_Net({a, b, c, d}, {c1, c2, c3})
                in  capacity("C", net) = 2
                end,

        /* Should return 0 since there is no stop with this name */

        [Stop_Capacity_2]
                let a = mk_Stop("A", 2),
                        b = mk_Stop("B", 3),
                        c = mk_Stop("C", 2),
                        d = mk_Stop("D", 2),
                        c1 = mk_Connection(3, 10, 2, {a, b}),
                        c2 = mk_Connection(3, 12, 4, {b, c}),
                        c3 = mk_Connection(2, 7, 4, {b, d}),
                        net = mk_Net({a, b, c, d}, {c1, c2, c3})
                in  capacity("E", net) = 0
                end,

−−2) Tests on generators

        [NET_2]
                "2) Tests on the generators",

        /* Should insert the stop, then should be 2 stops in the net */

        [InsertStop_1]
                let s1 = mk_Stop("Stop1", 2),
                        c = mk_Connection(1, 1, 1, {s1}),
                        net = mk_Net({s1}, {c})
                in  card Stops(insertStop("Stop2", 3, net)) = 2
                end,

        /* Should not insert the stop since he is already in the net */

        [InsertStop_2]
                let s1 = mk_Stop("Stop1", 2),
                        c = mk_Connection(1, 1, 1, {s1}),
                        net = mk_Net({s1}, {c})
```

```
            in      card Stops(insertStop("Stop1", 3, net)) = 1
            end,

        /* Should insert the connection, then should be 2 connections in the net */

        [AddConnection_1]
            let s1 = mk_Stop("Stop1", 2),
                s2 = mk_Stop("Stop2", 2),
                s3 = mk_Stop("Stop3", 2),
                c = mk_Connection(1, 1, 1, {s1, s2}),
                net = mk_Net({s1, s2, s3}, {c})
            in  card Connections(addConnection("Stop3", "Stop2", 3, 15, 7, net)) = 2
            end,

        /* Should not insert the connection since he is already in the net */

        [AddConnection_2]
            let s1 = mk_Stop("Stop1", 2),
                s2 = mk_Stop("Stop2", 2),
                c = mk_Connection(1, 1, 1, {s1, s2}),
                net = mk_Net({s1, s2}, {c})
            in  card Connections(addConnection("Stop1", "Stop2", 3, 15, 7, net)) = 1
            end,

--3) Test on the predicate and auxiliary functions used

        [NET_3]
            "3) Tests on the predicate and its auxiliary functions",

        /* Should return true since all stops have different ids */

        [UnicityNetStops_1]
            unicityNetStops(basicExample()) = true,

        /* Should return false since s1 and s2 have the same id */

        [UnicityNetStops_2]
            let s1 = mk_Stop("Stop1", 1),
                s2 = mk_Stop("Stop1", 2),
                c = mk_Connection(1, 1, 1,{s1, s2}),
                net = mk_Net({s1, s2}, {c})
            in  unicityNetStops(net) = false
            end,

        /* Should return 1 since their parameters have all the same values */
```

```
[UnicityNetStops_3]
        let s1 = mk_Stop("Stop1", 2),
                s2 = mk_Stop("Stop1", 2),
                c = mk_Connection(1, 1, 1,{s1, s2}),
                net = mk_Net({s1, s2}, {c})
        in  card Stops(net) = 1
        end,

/* Should return true, since all connections have two stops */

[ConnectionStops_1]
        connectionStops(basicExample()) = true,

/* Should return false since the connection is linking three stops */

[ConnectionStops_2]
        let s1 = mk_Stop("Stop1", 2),
                s2 = mk_Stop("Stop2", 2),
                s3 = mk_Stop("Stop3", 2),
                c = mk_Connection(1, 1, 1, {s1, s2, s3}),
                net = mk_Net({s1, s2}, {c})
        in  connectionStops(net) = false
        end,

/* Should return true since all stops have their capacity different from 0 */

[ValidateStops_1]
        validateStops(basicExample()) = true,

/* Should return false since the capacity of s2 is 0 */

[ValidateStops_2]
        let s1 = mk_Stop("Stop1", 2),
                s2 = mk_Stop("Stop2", 0),
                c = mk_Connection(1, 1, 1, {s1, s2}),
                net = mk_Net({s1, s2}, {c})
        in  validateStops(net) = false
        end,

/* Should return true since all connections have their parameters different from 0 */

[ValidateConnections_1]
        let s1 = mk_Stop("Stop1", 2),
                s2 = mk_Stop("Stop2", 2),
                c = mk_Connection(1, 1, 1, {s1, s2}),
                net = mk_Net({s1, s2}, {c})
        in  validateConnections(net) = true
        end,

/* Should return false since the driving time of c is 0 */

[ValidateConnections_2]
        let s1 = mk_Stop("Stop1", 2),
                s2 = mk_Stop("Stop2", 2),
                c = mk_Connection(1, 0, 1, {s1, s2}),
                net = mk_Net({s1, s2}, {c})
        in  validateConnections(net) = false
        end,

/* Should return true since stop b is used in c1, c2 and c3 */

[StopUsed_1]
        let a = mk_Stop("A", 2),
                b = mk_Stop("B", 3),
                c = mk_Stop("C", 2),
                d = mk_Stop("D", 2),
                c1 = mk_Connection(3, 10, 2, {a, b}),
                c2 = mk_Connection(3, 12, 4, {b, c}),
                c3 = mk_Connection(2, 7, 4, {b, d}),
                net = mk_Net({a, b, c, d}, {c1, c2, c3})
        in  stopUsed(b, net) = true
        end,

/* Should return false since s3 is not used in the only connection c */

[StopUsed_2]
        let s1 = mk_Stop("Stop1", 2),
                s2 = mk_Stop("Stop2", 2),
                s3 = mk_Stop("Stop3", 2),
                c = mk_Connection(1, 1, 1, {s1, s2}),
                net = mk_Net({s1, s2}, {c})
        in  stopUsed(s3, net) = false
        end,

/* Should return true since all stops are used at least once in a connection */

[UsageStops_1]
        usageStops(basicExample()) = true,

/* Should return false since s3 is not used in a connection */

[UsageStops_2]
```

```
    let s1 = mk_Stop("Stop1", 2),
        s2 = mk_Stop("Stop2", 2),
        s3 = mk_Stop("Stop3", 2),
        c = mk_Connection(1, 1, 1, {s1, s2}),
        net = mk_Net({s1, s2, s3}, {c})
    in  usageStops(net) = false
    end,

/* Should return true since the stops of c2 are in the spots of the net */

[ConnectionUsed_1]
    let a = mk_Stop("A", 2),
        b = mk_Stop("B", 3),
        c = mk_Stop("C", 2),
        d = mk_Stop("D", 2),
        c1 = mk_Connection(3, 10, 2, {a, b}),
        c2 = mk_Connection(3, 12, 4, {b, c}),
        c3 = mk_Connection(2, 7, 4, {b, d}),
        net = mk_Net({a, b, c, d}, {c1, c2, c3})
    in  connectionUsed(c2, basicExample()) = true
    end,

/* Should return false since s3 is not in the stops of the net */

[ConnectionUsed_2]
    let s1 = mk_Stop("Stop1", 2),
        s2 = mk_Stop("Stop2", 2),
        s3 = mk_Stop("Stop3", 2),
        c = mk_Connection(1, 1, 1, {s1, s2}),
        net = mk_Net({s1, s3}, {c})
    in  connectionUsed(c, net) = false
    end,

/* Should return true since all stops of the connections are in the stops of the net */

[UsageConnections_1]
    usageConnections(basicExample()) = true,

/* Should return false, since s3 of the connection c is not in the net */

[UsageConnections_2]
    let s1 = mk_Stop("Stop1", 2),
        s2 = mk_Stop("Stop2", 2),
        s3 = mk_Stop("Stop3", 2),
        c = mk_Connection(1, 1, 1, {s1, s2}),
        net = mk_Net({s1, s3}, {c})
```

```
          in  usageConnections(net) = false
          end,

/* Should return true since all connections are relating different stops */

[UnicityNetConnections_1]
          unicityNetConnections(basicExample()) = true,

/* Should return false since c1 and c2 are relating the same stops */

[UnicityNetConnections_2]
          let s1 = mk_Stop("Stop1", 2),
                    s2 = mk_Stop("Stop2", 2),
                    s3 = mk_Stop("Stop3", 2),
                    c1 = mk_Connection(1, 1, 1, {s1, s2}),
                    c2 = mk_Connection(1, 2, 5, {s1, s2}),
                    net = mk_Net({s1, s3}, {c1, c2})
          in  unicityNetConnections(net) = false
          end,

/* Should return true, since the net example is well formed */

[IsWellFormed]
          isWellformed(basicExample()) = true

end
```

4.1.2 Test specifications for TIMETABLE

We have split the tests on the TIMETABLE in two different files. The first one *testTIMETABLE.rsl* contains all the tests made on the final requirements for time tables. The other one *testTIMETABLEAux.rsl* contains the tests made on auxiliary functions used to define the requirements.

We used the same pattern as explained for the tests on the requirements test for time table. However, we did not use it on the auxiliary functions that have been tested on some specific data sets.

TIMETABLE

```
/∗ The following class is composed of a set of unit tests made on the
 ∗ requirements functions used in the Scheme TIMETABLE.
 ∗/
scheme testTIMETABLE =
extend TIMETABLE with
class
        value

        −−Example of the project description for a well defined net
        basicExample : Unit → Net
        basicExample() ≡
         let
                a = mk_Stop("A", 2),
                b = mk_Stop("B", 3),
                c = mk_Stop("C", 2),
                d = mk_Stop("D", 2),
                c1 = mk_Connection(3, 10, 2, {a, b}),
                c2 = mk_Connection(3, 12, 4, {b, c}),
                c3 = mk_Connection(2, 7, 4, {b, d})
        in
                mk_Net({a, b, c, d}, {c1, c2, c3})
        end,

        −−Example of the project description for a well defined net
        basicTimetable : Unit → TimeTable
        basicTimetable() ≡
                let
                        −−Creating the time table for tram 1
                        trip_one = mk_Trip("A", 0, 1),
                        trip_two = mk_Trip("B", 12, 14),
                        trip_three = mk_Trip("C", 28, 32),
                        trip_four = mk_Trip("B", 46, 48),
                        trip_five = mk_Trip("A", 59, 60),
                trips = {trip_one, trip_two, trip_three, trip_four, trip_five },
                plan_one = mk_Plan("tram1", trips),

                --Creating the time table for tram 2
                trip_a = mk_Trip("D", 0, 2),
                trip_b = mk_Trip("B", 11, 13),
                trip_c = mk_Trip("A", 28, 32),
                trip_d = mk_Trip("B", 47, 50),
                trip_e = mk_Trip("D", 58, 60),
                trips_bis = {trip_a, trip_b, trip_c, trip_d, trip_e },
                plan_two = mk_Plan("tram2", trips_bis)
        in
                {plan_one, plan_two}
        end

test_case

[TIMETABLE_1]
        "1) Tests on the predicate and their auxiliary functions",

/* Should return true because we want to get plan 2 */

[getPlanByTramId_1]
        let plan1 = mk_Plan("TramA",{mk_Trip("A",0,2)}),
                plan2 = mk_Plan("TramB",{mk_Trip("B",1,5)}),
                tt = {plan1, plan2}
        in  getPlanByTramId("TramB", tt) = plan2
        end,

/* Should return an empty plan because we want to get a tram without plan */

[getPlanByTramId_2]
        let plan1 = mk_Plan("TramA",{mk_Trip("A",0,2)}),
                plan2 = mk_Plan("TramB",{mk_Trip("B",1,5)}),
                tt = {plan1, plan2}
        in      getPlanByTramId("TramC", tt) = mk_Plan("", {})
        end,

[TIMETABLE_2]
        "2) Tests on the generators and their auxiliary functions",

/* Adding a plan for a tram to time table, it should
        return true because we have to have 2 elements after adding*/

[addTram_1]
        let trId = "TramB",
                tt = {mk_Plan("TramB",{mk_Trip("A",0,2)})}
        in  card addTram(trId, tt) = 2
        end,

/∗ Should return true because TramB is in time table ∗/

[ isIn_1 ]
        let  tt = {mk_Plan("TramB",{mk_Trip("A",0,2)})}
        in   isIn("TramB", tt) = true
        end,

/∗ Should return false because TramB is not in time table ∗/

[ isIn_2 ]
        let  tt = {mk_Plan("TramB",{mk_Trip("A",0,2)})}
        in   isIn("TramA", tt) = false
        end,

/∗ Should return true because becuase we add stop to plan
        and we should have two trip in the plan after adding∗/
[addStop_1]
        let  tt = {mk_Plan("TramA",{mk_Trip("A",0,2)})},
                expected = {mk_Plan("TramA", {mk_Trip("B",12,24),
                                                                        mk_Trip("A",0,2)})}
        in  addStop("TramA", "B", 12, 24, tt) = expected
        end,

/∗ Returns the initial time table since the trip already exists in plan ∗/

[addStop_2]
        let  tt = {mk_Plan("TramA",{mk_Trip("C", 12, 24)})}
        in  addStop("TramA", "C", 12, 24, tt)= tt
        end,

/∗ Should return true, since the stop C is in the time table ∗/

[ stopIsIn_1 ]
        stopIsIn("C", basicTimetable()) = true,

/∗ Should return false, since the stop A is not in the time table ∗/

[ stopIsIn_2 ]
        stopIsIn("E", basicTimetable()) = false,

[TIMETABLE_3]
        "3) Tests on the predicate, auxiliary functions are in testTIMETABLEAux",
/∗ Should return true since basic timetable has correct values ∗/

[RequirementOne_1]
        requirementOne(basicTimetable())= true,

/∗ Should return false because departure time is before arrival time ∗/

[RequirementOne_2]
        let plan1 = mk_Plan(”TramA”,{mk_Trip(”A”,5,3), mk_Trip(”B”,10,20)}),
                t = {plan1}
        in      requirementOne(t) = false
        end,

/∗ Should return true because values are between 0 and 60 ∗/

[RequirementTwo_1]
        let t = {mk_Plan(”TramA”,{mk_Trip(”A”,0,2)})},
                plan = mk_Plan(”TramA”,{mk_Trip(”A”,0,2)})
        in      requirementTwo(t) = true
        end,

/∗ Should return false because values are not between 0 and 60 ∗/

[RequirementTwo_2]
        let t = {mk_Plan(”TramA”,{mk_Trip(”A”,61,62), mk_Trip(”A”,0,2)})}
        in requirementTwo(t) = false
        end,

/∗ Should return true because we have only one plan for TramA ∗/

[RequirementThree_1]
        requirementThree(basicTimetable()) = true,

/∗ Should return false because we have two plan for tram A ∗/

[RequirementThree_2]
        let plan1 = mk_Plan(”TramA”,{mk_Trip(”A”,0,3), mk_Trip(”B”,10,20)}),
                plan2 = mk_Plan(”TramA”,{mk_Trip(”B”,1,5)}),
                t = {plan1, plan2}
        in      requirementThree(t) = false
        end,

/∗ Should return true since all plans have trips ∗/

[RequirementFour_1]
```

```
        requirementFour(basicTimetable()) = true,

/* Should return true since all plans have trips */

[RequirementFour_2]
        let t = {mk_Plan("TramA",{})}
        in      requirementFour(t) = false
        end,

/* Should return true since all values of the basic time table are defined */

[RequirementFive_1]
        requirementFive(basicTimetable()) = true,

/* Should return false since the first plan does not have a tram ID */

[RequirementFive_2]
        let     plan1 = mk_Plan("",{mk_Trip("A",0,2)}),
                plan2 = mk_Plan("TramB",{mk_Trip("E",1,5)}),
                t = {plan1, plan2}
        in      requirementFive(t) = false
        end,

/* Should return false since one trip does not have a stop ID */

[RequirementFive_3]
        let     plan1 = mk_Plan("TramA",{mk_Trip("",0,2)}),
                plan2 = mk_Plan("TramB",{mk_Trip("E",1,5)}),
                t = {plan1, plan2}
        in      requirementFive(t) = false
        end,

/* Should return true since all stops defined in trips are also defined in the net */

[RequirementSix_1]
        requirementSix(basicTimetable(), basicExample()) = true,

/* Should return false since stop E is not defined in the net */

[RequirementSix_2]
        let     plan1 = mk_Plan("TramA",{mk_Trip("A",0,2)}),
                plan2 = mk_Plan("TramB",{mk_Trip("E",1,5)}),
                t = {plan1, plan2}
        in      requirementSix(t, basicExample()) = false
        end,
/* Should return true since all stops of the net are used in the time table */

[RequirementSeven_1]
        requirementSeven(basicTimetable(), basicExample()) = true,

/* Should return false since the stop D of the net is not used in the time table */

[RequirementSeven_2]
        let plan1 = mk_Plan("TramA",{mk_Trip("A",0,2), mk_Trip("C",1,5)}),
                plan2 = mk_Plan("TramB",{mk_Trip("B",1,5)}),
                t = {plan1, plan2}
        in requirementSeven(t, basicExample()) = false
        end,

/* Should return true since this example is respecting the stops capacities */

[RequirementEight_1]
        requirementEight(basicTimetable(), basicExample()) = true,

/* Should return false since the capacity of C is 2 and there are 3 trams
 * at the time 25 */

[RequirementEight_2]
        let plan1 = mk_Plan("TramA",{mk_Trip("B",0,3), mk_Trip("C",15,20)}),
                plan2 = mk_Plan("TramB",{mk_Trip("B",1,5), mk_Trip("C",12, 15)}),
                plan3 = mk_Plan("TramC",{mk_Trip("C",12,22)}),
                t = {plan1, plan2, plan3}
        in  requirementEight(t, basicExample()) = false
        end,

/* Should return true since all connection implicitly defined in the time table are
 * also defined inside of the net */

[RequirementNine_1]
        requirementNine(basicTimetable(), basicExample()) = true,

/* Should return false since there is no connection between A and D */

[RequirementNine_2]
        let plan1 = mk_Plan("TramA",{mk_Trip("A",0,3), mk_Trip("D",15,20)}),
                plan2 = mk_Plan("TramB",{mk_Trip("B",1,5)}),
                t = {plan1, plan2}
        in  requirementNine(t, basicExample()) = false
        end,

/* Should return true since the example respects the minimum driving time */
```

```
[RequirementEleven_1]
        requirementEleven(basicTimetable(), basicExample()) = true,

/* Should return false since the minimum driving time between A and B is
 * 10 and here the table tables gives a time of 7 */

[RequirementEleven_2]
        let plan1 = mk_Plan("TramA",{mk_Trip("A",0,3), mk_Trip("B",10,20)}),
                plan2 = mk_Plan("TramB",{mk_Trip("B",1,5)}),
                t = {plan1, plan2}
        in requirementEleven(t, basicExample()) = false
        end,

/* Should return true since the example respects the connection capacities */

[RequirementTwelve_1]
        requirementTwelve(basicTimetable(), basicExample()) = true,

/* Should return false since it does not respect the capacity (3) of B to C */

[RequirementTwelve_2]
        let plan1 = mk_Plan("TramA",{mk_Trip("B",0,3), mk_Trip("C",15,20)}),
                plan2 = mk_Plan("TramB",{mk_Trip("B",1,5), mk_Trip("C",12, 15)}),
                plan3 = mk_Plan("TramC",{mk_Trip("B",1,4), mk_Trip("C",15, 20)}),
                t = {plan1, plan2, plan3}
        in  requirementTwelve(t, basicExample()) = false
        end,

/* Should return true since the example respects the connection headways */

[RequirementThirteen_1]
        requirementThirteen(basicTimetable(), basicExample()) = true,

/* Should return false since the min headway between B and C is 4 and there
 * is only 2 minutes here */

[RequirementThirteen_2]
        let plan1 = mk_Plan("TramA",{mk_Trip("B",0,3), mk_Trip("C",15,20)}),
                plan2 = mk_Plan("TramB",{mk_Trip("B",1,5), mk_Trip("C",12, 15)}),
                t = {plan1, plan2}
        in  requirementThirteen(t, basicExample()) = false
        end,

[IsWellFormed]
        isWellformed(basicTimetable(), basicExample()) = true
```

```
end
```

4.1.3 Test specifications for auxiliary functions for TIMETABLE

TIMETABLE

```
/* The following class is composed of a set of unit tests made on auxiliary
 * functions used in the Scheme TIMETABLE.
 */
scheme testTIMETABLEAux =
extend TIMETABLE with
class
        value

        −−Example of the project description for a well defined net
        basicExample : Unit → Net
        basicExample() ≡
         let      a = mk_Stop("A", 2),
                        b = mk_Stop("B", 3),
                        c = mk_Stop("C", 2),
                        d = mk_Stop("D", 2),
                        c1 = mk_Connection(3, 10, 2, {a, b}),
                        c2 = mk_Connection(3, 12, 4, {b, c}),
                        c3 = mk_Connection(2, 7, 4, {b, d})
                in      mk_Net({a, b, c, d}, {c1, c2, c3})
                end,

        −−Example of the project description for a well defined net
        basicTimetable : Unit → TimeTable
        basicTimetable() ≡
                let −−Creating the time table for tram 1
                        trip_one = mk_Trip("A", 0, 1),
                        trip_two = mk_Trip("B", 12, 14),
                        trip_three = mk_Trip("C", 28, 32),
                        trip_four = mk_Trip("B", 46, 48),
                        trip_five = mk_Trip("A", 59, 60),
                        trips = {trip_one, trip_two, trip_three , trip_four , trip_five },
                        plan_one = mk_Plan("tram1", trips),

                        −−Creating the time table for tram 2
                        trip_a = mk_Trip("D", 0, 2),
                        trip_b = mk_Trip("B", 11, 13),
                        trip_c = mk_Trip("A", 28, 32),
                        trip_d = mk_Trip("B", 47, 50),
                        trip_e = mk_Trip("D", 58, 60),
                    trips_bis  = {trip_a, trip_b , trip_c , trip_d , trip_e },
                    plan_two = mk_Plan("tram1", trips_bis)
          in  {plan_one, plan_two}
          end,

-- Elements used in tests
time : Nat = 4,
emptyPlan : Plan-set = {}

test_case

[TIMETABLE_AUX]
          "Tests on auxiliary  functions  used for the time table  predicate",

[GetTripByATime_1]
          let trip_one  = mk_Trip("A", 0, 1),
                    trip_two  = mk_Trip("B", 12, 14),
                    trip_three  = mk_Trip("C", 28, 32),
                    trip_four  = mk_Trip("B", 46, 48),
                    trips  = {trip_one, trip_two , trip_three , trip_four },
                    plan = mk_Plan("tram1", trips)
          in        getTripByATime(trips, 0) = trip_one
          end,

[GetTripByATime_2]
          let trip_one  = mk_Trip("A", 0, 1),
                    trip_two  = mk_Trip("B", 12, 14),
                    trip_three  = mk_Trip("C", 28, 32),
                    trip_four  = mk_Trip("B", 46, 48),
                    trips  = {trip_one, trip_two , trip_three , trip_four },
                    plan = mk_Plan("tram1", trips)
          in        getTripByATime(trips, 28) = trip_three
          end,

[GetLastTripDepartureTime_1]
          let trip_one  = mk_Trip("A", 0, 1),
                    trip_two  = mk_Trip("B", 12, 14),
                    trip_three  = mk_Trip("C", 28, 32),
                    trip_four  = mk_Trip("B", 46, 48),
                    trips  = {trip_one, trip_two , trip_three , trip_four }
          in  getLastTripDepartureTime(trips, 0) = 48
          end,

[GetLastTripDepartureTime_2]
          let trip_one  = mk_Trip("A", 0, 1),
                    trip_two  = mk_Trip("B", 12, 14),
              trip_three  = mk_Trip("C", 28, 32),
              trip_four  = mk_Trip("B", 46, 48),
              trip_five  = mk_Trip("A", 55, 60),
              trips  = {trip_one, trip_two, trip_three , trip_four , trip_five },
              plan = mk_Plan("tram1", trips)
       in  getLastTripDepartureTime(trips, 0) = 60
       end,

[GetWaitingTrams]
       card getWaitingTrams(basicTimetable(), "A", 1) = 1
       ∧ card getWaitingTrams(basicTimetable(), "A", 29) = 1
       ∧ card getWaitingTrams(basicTimetable(), "A", 47) = 0,

[GetNextTrip_1]
       let trip_one  = mk_Trip("A", 0, 1),
              trip_two  = mk_Trip("B", 12, 14),
              trip_three  = mk_Trip("C", 28, 32),
              trip_four  = mk_Trip("B", 46, 48),
              trips  = {trip_one, trip_two, trip_three , trip_four },
              plan = mk_Plan("tram1", trips)
       in  getNextTrip(plan, trip_two)  = trip_three
       end,

[GetNextTrip_2]
       let trip_one  = mk_Trip("A", 0, 1),
              trip_two  = mk_Trip("B", 12, 14),
              trip_three  = mk_Trip("C", 28, 32),
              trip_four  = mk_Trip("B", 46, 48),
              trips  = {trip_one, trip_two, trip_three , trip_four },
              plan = mk_Plan("tram1", trips)
       in  getNextTrip(plan, trip_one)  = trip_two
       end,

[GetNextTrip_3]
       let trip_one  = mk_Trip("A", 0, 1),
              trip_two  = mk_Trip("B", 12, 14),
              trip_three  = mk_Trip("C", 28, 32),
              trip_four  = mk_Trip("B", 46, 48),
              trip_five = mk_Trip("A", 55, 60),
              trips  = {trip_one, trip_two, trip_three , trip_four , trip_five },
              plan = mk_Plan("tram1", trips)
       in  getNextTrip(plan,  trip_five )  = trip_one
       end,

[GetAllTrips_1]
       card getAllTrips(basicTimetable()) = 10,
[GetAllTrips_2]
        let plan1 = mk_Plan(”TramA”,{mk_Trip(”A”,0,3), mk_Trip(”D”,15,20)}),
                plan2 = mk_Plan(”TramB”,{mk_Trip(”B”,1,5)}),
                t = {plan1, plan2}
        in  card getAllTrips(t) = 3
        end,

[NextDepartureLoop]
        let plan1 = mk_Plan(”TramA”,{mk_Trip(”A”,0,3), mk_Trip(”B”,15,20)}),
                trip = mk_Trip(”A”,1,5),
                plan2 = mk_Plan(”TramB”,{trip, mk_Trip(”B”,15,20)}),
                t = {plan1, plan2}
        in  nextDepartureLoop(”A”, ”B”, time, t, mk_Plan(””, {}), mk_Trip(””, 61, 61))
                 = (plan2, trip)
        end,

[NextDeparture_1]
        let plan1 = mk_Plan(”TramA”,{mk_Trip(”A”,0,3), mk_Trip(”B”,15,20)}),
                trip = mk_Trip(”A”,1,5),
                plan2 = mk_Plan(”TramB”,{trip, mk_Trip(”B”,15,20)}),
                t = {plan1, plan2}
        in  nextDeparture(”A”, ”B”, time, t) = (plan2, trip)
        end,

[NextDeparture_2] −− time = 4
        let     plan = mk_Plan(”tram1”,
                        {mk_Trip(”D”,58,60), mk_Trip(”B”,47,50),
                        mk_Trip(”A”,28,32), mk_Trip(”B”,11,13),
                        mk_Trip(”D”,0,2)}),
                trip = mk_Trip(”A”,28,32)
        in nextDeparture(”A”, ”B”, time, basicTimetable()) = (plan, trip)
        end,

[NextDeparture_3] −− time = 4
        let plan = mk_Plan(”tram1”,
                        {mk_Trip(”D”,58,60), mk_Trip(”B”,47,50),
                        mk_Trip(”A”,28,32), mk_Trip(”B”,11,13),
                        mk_Trip(”D”,0,2)}),
                trip = mk_Trip(”B”,11,13)
        in      nextDeparture(”B”, ”A”, time, basicTimetable()) = (plan, trip)
        end,

[CheckMinDrivingTime_1]
        let one = mk_Trip(”A”, 0, 1),
                two = mk_Trip(”B”, 12, 14),
                 trips  = {one, two},
                 plan = mk_Plan("tram1", trips),
                 n = basicExample()
        in       checkMinDrivingTime(one, plan, basicExample())
        end,

    [CheckMinDrivingTime_2]
        let one = mk_Trip("A", 0, 1),
                 two = mk_Trip("B", 9, 14),
                 trips  = {one, two},
                 plan = mk_Plan("tram1", trips)
        in checkMinDrivingTime(one, plan, basicExample()) = false
        end,

    [getTramsInPlan_1]
        let one = mk_Trip("A", 0, 1),
                 two = mk_Trip("B", 15, 20),
                 trips  = {one, two},
                 plan = mk_Plan("tram1", trips)
        in getTramsInPlan(plan, trips(plan), "A", "B", 10, {})
                 = {mk_Trip("A",0,1)}
        end,

    [getTramsOn_1]
        getTramsOn(basicTimetable(), "A", "B", 7) = {mk_Trip("A",0,1)},

    [getTramsOn_2]
        getTramsOn(basicTimetable(), "C", "B", 40) = {mk_Trip("C",28,32)}

end
```

4.2 Test results

The following sections shows the results of the tests contained in the files *test-Net.sml.results* and *testTimeTable.sml.results*

4.2.1 Results for NET

[NET_1] "1) Tests on the observers and their auxiliary functions"
[IsinStop_1] **true**
[IsinStop_2] **true**
[AreConnectedBy_1] **true**
[AreConnectedBy_2] **true**
[AreDirectlyConnected_1] **true**
[AreDirectlyConnected_2] **true**
[GetConnection] **true**
[MinHeadway_1] **true**
[MinHeadway_2] **true**
[MinDrivingTime_1] **true**
[MinDrivingTime_2] **true**
[Capacity_1] **true**
[Capacity_2] **true**
[GetStop] **true**
[Stop_Capacity_1] **true**
[Stop_Capacity_2] **true**
[NET_2] "2) Tests on the generators"
[InsertStop_1] **true**
[InsertStop_2] **true**
[AddConnection_1] **true**
[AddConnection_2] **true**
[NET_3] "3) Tests on the predicate and its auxiliary functions"
[UnicityNetStops_1] **true**
[UnicityNetStops_2] **true**
[UnicityNetStops_3] **true**
[ConnectionStops_1] **true**
[ConnectionStops_2] **true**
[ValidateStops_1] **true**
[ValidateStops_2] **true**
[ValidateConnections_1] **true**
[ValidateConnections_2] **true**
[StopUsed_1] **true**
[StopUsed_2] **true**
[UsageStops_1] **true**
[UsageStops_2] **true**
[ConnectionUsed_1] **true**
[ConnectionUsed_2] **true**
[UsageConnections_1] **true**

[UsageConnections_2] **true**
[UnicityNetConnections_1] **true**
[UnicityNetConnections_2] **true**
[IsWellFormed] **true**

4.2.2 Results for TIMETABLE

[TIMETABLE_1] "1) Tests on the predicate and their auxiliary functions"
[getPlanByTramId_1] **true**
[getPlanByTramId_2] **true**
[TIMETABLE_2] "2) Tests on the generators and their auxiliary functions"
[addTram_1] **true**
[isIn_1] **true**
[isIn_2] **true**
[addStop_1] **true**
[addStop_2] **true**
[stopIsIn_1] **true**
[stopIsIn_2] **true**
[TIMETABLE_3] "3) Tests on the predicate, auxiliary functions are **in** testTIMETABLEAux"
[RequirementOne_1] **true**
[RequirementOne_2] **true**
[RequirementTwo_1] **true**
[RequirementTwo_2] **true**
[RequirementThree_1] **true**
[RequirementThree_2] **true**
[RequirementFour_1] **true**
[RequirementFour_2] **true**
[RequirementFive_1] **true**
[RequirementFive_2] **true**
[RequirementFive_3] **true**
[RequirementSix_1] **true**
[RequirementSix_2] **true**
[RequirementSeven_1] **true**
[RequirementSeven_2] **true**
[RequirementEight_1] **true**
[RequirementEight_2] **true**
[RequirementNine_1] **true**
[RequirementNine_2] **true**
[RequirementEleven_1] **true**
[RequirementEleven_2] **true**
[RequirementTwelve_1] **true**
[RequirementTwelve_2] **true**
[RequirementThirteen_1] **true**
[RequirementThirteen_2] **true**
[IsWellFormed] **true**

4.2.3 Results for auxiliary functions for TIMETABLE

[TIMETABLE_AUX] "Tests on auxiliary functions used **for** the time table predicate"
[GetTripByATime_1] **true**
[GetTripByATime_2] **true**
[GetLastTripDepartureTime_1] **true**
[GetLastTripDepartureTime_2] **true**
[GetWaitingTrams] **true**
[GetNextTrip_1] **true**
[GetNextTrip_2] **true**
[GetNextTrip_3] **true**
[GetAllTrips_1] **true**
[GetAllTrips_2] **true**
[NextDepartureLoop] **true**
[NextDeparture_1] **true**
[NextDeparture_2] **true**
[NextDeparture_3] **true**
[CheckMinDrivingTime_1] **true**
[CheckMinDrivingTime_2] **true**
[getTramsInPlan_1] **true**
[getTramsOn_1] **true**
[getTramsOn_2] **true**

5 Conclusion

In this project we started by analysing the project description in order to define a data structure that can be used to model the tram net and its related time tables. We continued then by specifying requirements in order to check if a net and time tables are well formed and that they are consistent together. After having implemented them into RSL, we have been able to check these requirements, to translate them to SML and to perform test to check their validity. All of them have been tested on small specific data set in order to check the behaviour of our functions. Afterwards, these requirements have all been tested on the examples of net and timetable provided in the project description. From these tests, we can conclude that we should be able to define a consistent time table and net based on our requirements functions.