Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Rome2Rio

Formal Methods in Software Engineering

Turma 3

Ana Denisa Secuiu - up201803342

Ana Margarida Silva - up201505505

Danny Soares - up201505509

07/01/2019

Contents

1. Informal system description and list of requirements

1.1. Informal system description

The system we developed is a multimodal transport search engine.

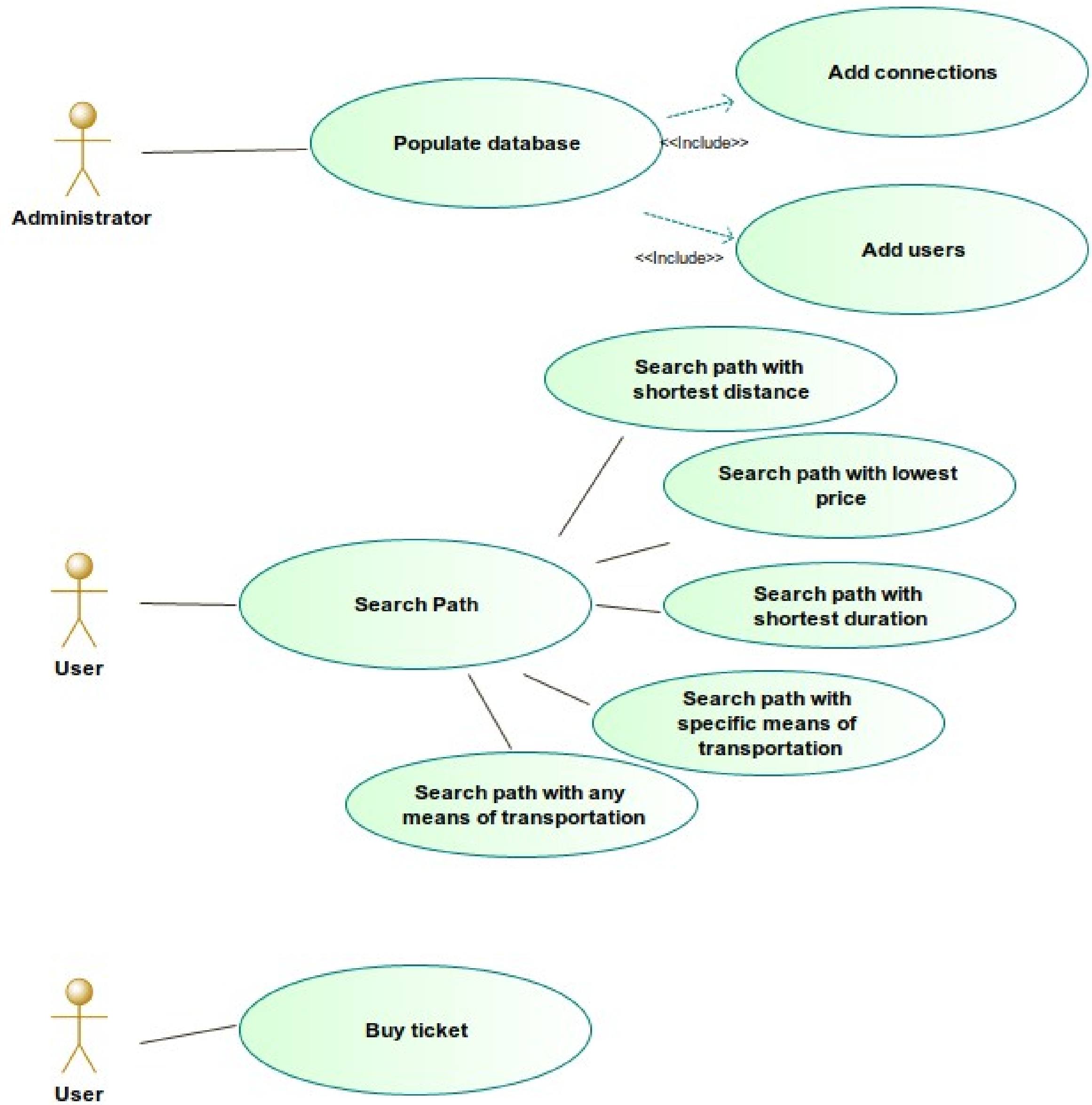
A user can input a starting point and a destination city and the system will try to build the best path according to the user's preference regarding price, distance and duration of the trip. Users can also specify what means of transportation they want (e.g. Bus, Train, etc).

1.2. List of requirements

ID	Priority	Description
R1	Mandatory	The administrator can populate the system's database with information.
R2	Mandatory	The user can search for the path with the shortest distance between two cities.
R3	Mandatory	The user can search for the path with the lowest price between two cities.
R4	Mandatory	The user can search for the path with the shortest duration between two cities.
R5	Mandatory	The user can search for paths with the means of transportation that he wants.
R6	Mandatory	The user can search for paths without specifying any means of transportation.
R7	Optional	The user can buy tickets for the path returned by the search engine.

2. Visual UML Model

2.1. Use Case Model



The major use case scenarios are three, the administrator populating the database, the user searching for paths and the user buying tickets for trips. The search case can be split into multiple cases, since the user can search for trips in multiple ways. The search engine accepts searches with or without specification of means of transportation, and accepts searches with one of three factors: shortest distance, lowest price or shortest duration. This cases are described in the following tables.

Scenario	**Populate database (connections) (R1)**
Description	The administrator can add connections to the transport graph.
Pre conditions	The new connection must have a distance greater that 0 and a timetable filled.
Post conditions	A connection is created with distance and duration greater than 0, price greater or equal to 0 and the timetable filled.
Scenario	**Populate database (users) (R1)**
Description	The administrator can add users to the database.
Pre conditions	The new user must have an ID and a cash amount greater or equal than 0 and a password between 0000 and 9999.
Post conditions	(unspecified)

Scenario	**Search path with shortest distance (R2)**
Description	The user searches for a path between two cities, specifying the weight factor to be the distance.
Pre conditions	The weight factor must be set to 1
Post conditions	The maximum duration of the trip is lower than 0 (no maximum specified by the user) or below the maximum duration specified by the user.

Scenario	**Search path with lowest price (R3)**
Description	The user searches for a path between two cities, specifying the weight factor to be the price.
Pre conditions	The weight factor must be set to 2
Post conditions	The maximum duration of the trip is lower than 0 (no maximum specified by the user) or below the maximum duration specified by the user.

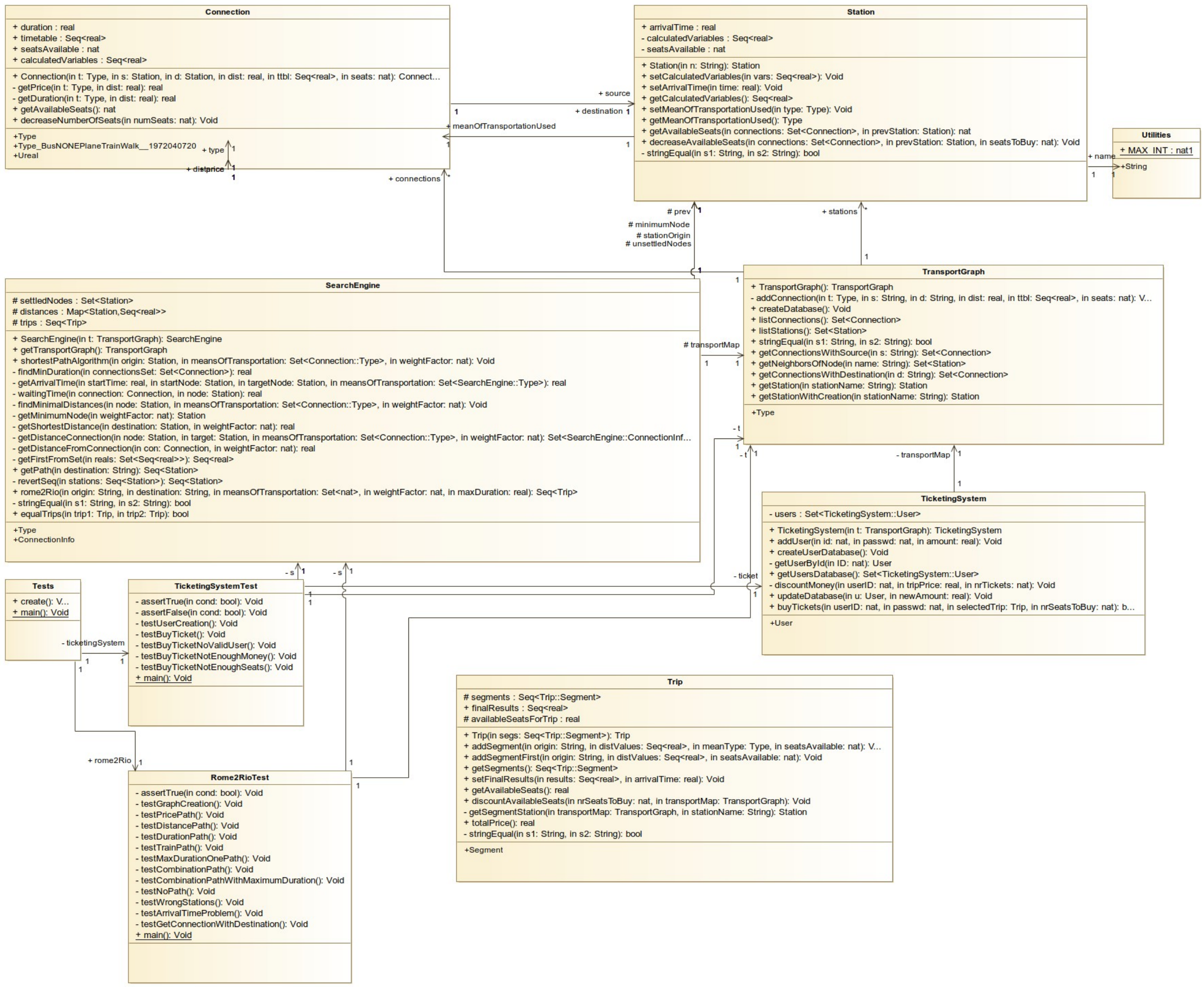
Scenario	**Search path with shortest duration (R4)**
Description	The user searches for a path between two cities, specifying the weight factor to be the trip’s duration.
Pre conditions	The weight factor must be set to 3
Post conditions	The maximum duration of the trip is lower than 0 (no maximum specified by the user) or below the maximum duration specified by the user.

Scenario	**Search path with the means of transportation (R5)**
Description	The user searches for a path between two cities, specifying the means of transportation he wants to take.
Pre conditions	The weight factor must be between 1 and 3 (inclusive)
Post conditions	The maximum duration of the trip is lower than 0 (no maximum specified by the user) or below the maximum duration specified by the user.

Scenario	Search path with shortest distance (R6)
Description	The user searches for a path between two cities, without specifying the means of transportation he wants to take.
Pre conditions	The weight factor must be between 1 and 3 (inclusive)
Post conditions	The maximum duration of the trip is lower than 0 (no maximum specified by the user) or below the maximum duration specified by the user.

Scenario	Buy tickets for a trip (R7)
Description	The user buys one or more tickets for the trip returned by the search engine.
Pre conditions	The user needs to be on the database and have enough money to complete the transaction. The trip’s price must be greater or equal to 0 and the trip’s empty seats must be greater or equal to 1.
Post conditions	(unspecified)

2.2. Class Diagram



2.2.1. Classes

Class	Description
SearchEngine	Core model; defines the variables and operations available for users. One user can choose to get the best path to get from source to destination on the following criteria: price, duration and distance.
Connection	Defines a connection between a source Station and a destination Station, keeping data regarding the mean of transportation, distance and the timetable available in that connection. It also

	contains methods to compute the price and duration of each connection depending of the corresponding mean of transportation.
TransportGraph	Defines the transportation graph, the abstraction of a map containing all the available connections between cities, each connection having information regarding the distance, the means of transportation available and the timetables. Furthermore, this class contains methods to get connections with specific particularities.
Trip	Defines the path to follow from a source to a destination as a succession of segments, with information regarding time duration, distance, price and means of transportation.
Station	Defines a station with name, arrival time in the station, the distance, price and duration of the trip up to that station and the mean of transportation used.
TicketingSystem	Defines the ticketing system, managing the user database and the transactions from users buying tickets for trips.
Utilities	Defines general types and constants used in the model.

2.2.2. Test classes

Class	**Description**
Tests	Creates and runs all the tests from Rome2RioTest and TicketingSystemTest classes.
Rome2RioTest	Defines the usage scenarios and test cases for the Rome2Rio search engine.
TicketingSystemTest	Defines the usage scenarios and test cases for the ticketing system.

3. Formal VDM++ Model

3.1. Connection

```
class Connection

types
      public Type = <Plane> | <Bus> | <Train> | <Walk> | <NONE>;
      public Ureal = real
                                                inv n == n >= 0;

values
instance variables
      -- declarations and initializations
      public type: Type;
      public source: Station;
      public destination: Station;
      public distance: Ureal;
      public price: Ureal;
      public duration: real;
      public timetable : seq of real;
      public seatsAvailable : nat := 0;
      public calculatedVariables : seq of real;

      inv source <> destination;
      inv seatsAvailable >= 0;

operations
      -- Constructor
      public Connection: Type * Station * Station * real * seq of real * nat ==>
Connection
      Connection(t, s, d, dist, ttbl, seats) ==
      (
            type := t;
            source := s;
            destination := d;
            distance := dist;
            price := getPrice(t, dist);
            duration := getDuration(t, dist);
            timetable := ttbl;
            calculatedVariables := [distance, price, duration];
            seatsAvailable := seats;

            return self;
      )
      pre ttbl <> []
      post distance > 0 and price >= 0 and duration > 0 and timetable <> [];

      -- calculate price of the connection
      private getPrice: Type * real ==> real
      getPrice(t, dist) ==
      (dcl priceKm: real;
                  cases t:
                        <Plane> -> priceKm := 0.06,
```

```
                        <Bus> -> priceKm := 0.1,
                        <Train> -> priceKm := 0.07,
                        <Walk> -> priceKm := 0
                end;

                return priceKm * dist;
        )
        pre dist > 0;

        -- calculate duration of the connection
        private getDuration: Type * real ==> real
        getDuration(t, dist) ==
        (dcl speed: real;
              cases t:
                        <Plane> -> speed := 760,
                        <Bus> -> speed := 80,
                        <Train> -> speed := 100,
                        <Walk> -> speed := 4
                end;
                return dist / speed;
        )
        pre dist > 0;

        -- returns available seats in this connection
        public getAvailableSeats: () ==> nat
        getAvailableSeats() ==
        (
              return seatsAvailable;
        );

        -- decreases number of seats in the connection
        public decreaseNumberOfSeats: nat ==> ()
        decreaseNumberOfSeats(numSeats) ==
        (
              seatsAvailable := seatsAvailable - numSeats;
        )
        pre seatsAvailable - numSeats >= 0;


functions

traces

end Connection
```

3.2. SearchEngine

```
class SearchEngine
types
        public Type = Connection`Type;
        public ConnectionInfo :: con  : seq of Connection
                                                                               type
: Type

weight: real
```

```
instance variables

	protected transportMap: TransportGraph;
	protected settledNodes: set of Station := {};
	protected unsettledNodes: set of Station := {};
	protected distances: map Station to seq of real := {|->}; -- seq of reals --
> 1: distance; 2: price; 3: duration
	protected prev: map Station to Station := {|->};
	protected stationOrigin : Station;
	protected minimumNode : Station;
	protected trips : seq of Trip := [];

	inv visitedImpliesConnected(settledNodes, unsettledNodes, transportMap);

operations
	-- constructor
	public SearchEngine: TransportGraph ==> SearchEngine
	SearchEngine(t) ==
	(
		transportMap := t;
		unsettledNodes := transportMap.listStations();
		return self;
	);

	public getTransportGraph: () ==> TransportGraph
	getTransportGraph() == return transportMap;

	-- shortest path algorithm (losely based on dijkstra) logic and main
function
	public shortestPathAlgorithm: Station * set of Connection`Type * nat ==> ()
	shortestPathAlgorithm(origin, meansOfTransportation, weightFactor) ==
	(

		distances := distances ++ {origin |-> [0,0,0]};
		origin.setArrivalTime(0);

		settledNodes := settledNodes union {origin};

		while(settledNodes inter unsettledNodes <> {}) do (

			dcl minimumNode : Station := getMinimumNode(weightFactor);
			settledNodes := settledNodes union {minimumNode};
			unsettledNodes := unsettledNodes\{minimumNode};

			findMinimalDistances(minimumNode, meansOfTransportation,
weightFactor);

		);
	)
	pre (weightFactor = 1 or weightFactor = 2 or weightFactor = 3) and
validGraph(transportMap) and validStart(stationOrigin, transportMap)
	post IsShortestPath(distances, prev, settledNodes, stationOrigin,
transportMap, meansOfTransportation, weightFactor);

	-- from all possible connections between startNode and targetNode,
```

```
	-- return the duration of the shortest one
	private findMinDuration: set of Connection ==> real
	findMinDuration(connectionsSet) ==
	(
		dcl minDuration : real;

		minDuration := Utilities`MAX_INT;
		for all c in set connectionsSet do (
			if c.duration <= minDuration then (
				minDuration := c.duration;
			);
		);
		return minDuration;
	);

	-- gets the arrivalTime in targetNode from startNode on the shortest route 
regarding distance
	private getArrivalTime: real * Station * Station * set of Type ==> real
	getArrivalTime(startTime, startNode, targetNode, meansOfTransportation) ==
	(
		dcl validConnections : set of Connection;
		dcl connectionsFromSource : set of Connection;
		dcl minArrivalTime : real;

		validConnections := {};
		connectionsFromSource := 
transportMap.getConnectionsWithSource(startNode.name);

		for all c in set connectionsFromSource do (
			if (stringEqual(c.destination.name, targetNode.name)) then ( -- 
if they are connections to target node
				if c.type in set meansOfTransportation then (
					validConnections := validConnections union {c};
				);
			);
		);

		minArrivalTime := startTime + startNode.arrivalTime + 
findMinDuration(validConnections);

		return minArrivalTime;
	);

	private waitingTime: Connection * Station ==> real
	waitingTime(connection, node) ==
	(
		dcl timeDiff : real;
		dcl minDiff : real;

		minDiff := Utilities`MAX_INT;
		for idx = 1 to len connection.timetable do (
			timeDiff := connection.timetable(idx) - node.arrivalTime;
			if (timeDiff >= 0 and timeDiff < minDiff) then (
				minDiff := timeDiff;
			);
```

```
		);

		return minDiff;
	);

	-- function that goes through all the nodes and their neighbours, to get the minimum distances to each of them
	private findMinimalDistances: Station * set of Connection`Type * nat ==> ()
	findMinimalDistances(node, meansOfTransportation, weightFactor) ==
	(
		dcl adjacentNodes: set of Station := transportMap.getNeighborsOfNode(node.name);
		dcl neighborArrivalTime : real;
		dcl startTime : real := 0;

		-- Compute arrivalTime for all neighbors of the source node
		for all neighbor in set adjacentNodes do (
			neighborArrivalTime := getArrivalTime(0, node, neighbor, meansOfTransportation);
			neighbor.setArrivalTime(neighborArrivalTime);
		);

		for all target in set adjacentNodes do (

			dcl cons : set of ConnectionInfo := getDistanceConnection(node, target, meansOfTransportation, weightFactor);
			for all con in set cons do (
				if(con.con <> []) then (

					if(getShortestDistance(target, weightFactor) > getShortestDistance(node, weightFactor) + con.weight) then (

						dcl newArrivalTime : real;
						dcl waitT : real := waitingTime(con.con(1), node);
						if (waitT >= 0 and waitT <> Utilities`MAX_INT) then (

							dcl newPrice : real;
							dcl newDist : real;
							dcl newDuration : real;
							dcl newSeq : seq of real := [];

							newArrivalTime := startTime + node.arrivalTime + getDistanceFromConnection(con.con(1), 3) + waitT;

							newDist := getShortestDistance(node, 1) + getDistanceFromConnection(con.con(1), 1);
							newPrice := getShortestDistance(node, 2) + getDistanceFromConnection(con.con(1), 2);
							newDuration := getShortestDistance(node, 3) + getDistanceFromConnection(con.con(1), 3);

							newSeq := [newDist,newPrice,newDuration];
```

```
                                          distances := distances ++ {target |->
newSeq};
                                          target.setCalculatedVariables(newSeq);
                                          target.setArrivalTime(newArrivalTime);

      target.setMeanOfTransportationUsed(con.type);
                                          prev := prev ++ {target |-> node};

                                          settledNodes := settledNodes union
{target};

                                    );
                              );
                        );
                  );
            );
      )
      pre weightFactor = 1 or weightFactor = 2 or weightFactor = 3;

      -- get node that hasn't been visited with the minimum weight factor
      private getMinimumNode: nat ==> Station
      getMinimumNode(weightFactor) ==
      (
            minimumNode := new Station("");
            for all n in set settledNodes inter unsettledNodes do (
                  if(minimumNode.name = "") then (
                        minimumNode := n;
                  )
                  else (
                        if(getShortestDistance(n, weightFactor) <
getShortestDistance(minimumNode, weightFactor)) then (
                              minimumNode := n;
                        );
                  );
            );

            return minimumNode;
      )
      pre weightFactor = 1 or weightFactor = 2 or weightFactor = 3
      post isMinimumNode(weightFactor, distances, minimumNode, transportMap,
settledNodes, unsettledNodes);

      -- returns the minimum distance (or price or duration) in the distance seq
      private getShortestDistance: Station * nat ==> real
      getShortestDistance(destination, weightFactor) ==
      (
            dcl d: map Station to seq of real := {destination} <: distances;
            dcl d1 : set of seq of real := rng d;
            dcl dist : seq of real := getFirstFromSet(d1);

            return dist(weightFactor);
      );
```

```
	-- returns info about the connection from one node to another, if the
connection is with the correct mean of transporation and if
	-- the origin and destination are correct
	private getDistanceConnection: Station * Station * set of Connection`Type *
nat ==> set of ConnectionInfo
	getDistanceConnection(node, target, meansOfTransportation, weightFactor) ==
	(
		dcl conTmp : Connection;
		dcl connectionInfo : set of ConnectionInfo := {};

		for all con in set transportMap.listConnections() do (
			if(con.type in set meansOfTransportation) then (

				if(stringEqual(con.source.name,node.name) and
stringEqual(con.destination.name, target.name)) then (

					if(weightFactor = 1) then (
						connectionInfo := connectionInfo union
{mk_ConnectionInfo([con], con.type, con.distance)};
					)
					else if (weightFactor = 2) then (
						connectionInfo := connectionInfo union
{mk_ConnectionInfo([con], con.type, con.price)};
					)
					else if (weightFactor = 3) then (
						connectionInfo := connectionInfo union
{mk_ConnectionInfo([con], con.type, con.duration)};
					);
				);
			);
		);

		return connectionInfo;

	);

	-- get distance (or price or duration) from one connection
	private getDistanceFromConnection: Connection * nat ==> real
	getDistanceFromConnection(con, weightFactor) ==
	(
		dcl ret : real := 0;
		if(weightFactor = 1) then (
			ret := con.distance;
		)
		else if (weightFactor = 2) then (
			ret := con.price;
		)
		else if (weightFactor = 3) then (
			ret := con.duration;
		);

		return ret;
	)
	pre weightFactor = 1 or weightFactor = 2 or weightFactor = 3;
```

```
    -- return first element of a set
    private getFirstFromSet: set of seq of real ==> seq of real
    getFirstFromSet(reals) ==
    (
        if(reals <> {}) then (
            for all ds in set reals do return ds;
        )
        else return [1000000, 1000000, 1000000];
    );

    ----------------------------------------------------------------------------
-----------------

    -- Returns seq of nodes from origin to destination
    public getPath: Utilities`String ==> seq of Station
    getPath(destination) ==
    (
        dcl path : seq of Station := [];
        dcl revertedPath: seq of Station := [];
SearchEngine
        dcl stationDest : Station := transportMap.getStation(destination);

        dcl tmp : map Station to Station := {stationDest} <: prev;
        dcl tmp2 : set of Station := rng tmp;

        if (tmp2 = {}) then (
            return [];
        );

        path := path ^ [stationDest];

        while (tmp2 <> {}) do (

            stationDest := prev(stationDest);
            path := path ^ [stationDest];

            tmp := {stationDest} <: prev;
            tmp2 := rng tmp;
        );

        revertedPath := revertSeq(path);

        return revertedPath;
    );

    -- reverts the order of the sequence
    private revertSeq: seq of Station ==> seq of Station
    revertSeq(stations) ==
    (
        dcl result : seq of Station := stations;
        dcl i : nat := 0;

        for sta in stations do (
             result(len stations - i) := sta;
             i := i + 1;
```

```
		);

		return result;
	);

	-- checks if the user input was with no means of transportation selected or not, if not, calculates all combinations and runs shortest path algorithm and
	-- and gets the shortest path for each combination.
	-- If a combination was provided by the user, the dijkstra algorithm is run and the shortest path related to the weightFactor and with the means of
	-- transportation selected is returned
	public rome2Rio: Utilities`String * Utilities`String * set of nat * nat * real ==> seq of Trip
	rome2Rio(origin, destination, meansOfTransportation, weightFactor, maxDuration) ==
	(
		dcl stationDest : Station := transportMap.getStation(destination);
		trips := [];

		if(stringEqual(stationDest.name, "Error")) then (
				IO`println("There is no destination station with that name.");
				return [];
		);
		stationOrigin := transportMap.getStation(origin);
		if(stringEqual(stationOrigin.name, "Error")) then (
				IO`println("There is no origin station with that name.");
				return [];
		);

		prev := {|->};
		distances := {|->};
		unsettledNodes := transportMap.listStations();
		settledNodes := {};

		if(meansOfTransportation = {}) then (

			dcl answerOne : seq of Station := [];
			dcl means : set of set of Connection`Type := {
				{<Bus>},{<Plane>},{<Train>},{<Walk>},{<Bus>,<Plane>},{<Bus>,<Train>},{<Bus>,<Walk>},{<Plane>,<Train>},{<Plane>,<Walk>},
				{<Train>,<Walk>},{<Bus>,<Train>,<Walk>},{<Bus>,<Train>,<Plane>},{<Plane>,<Train>,<Walk>},{<Bus>,<Plane>,<Walk>},{<Bus>,<Plane>,<Train>,<Walk>}
			};

			for all mean in set means do (

				dcl trip : Trip := new Trip([]);
				prev := {|->};
				distances := {|->};
				unsettledNodes := transportMap.listStations();
				settledNodes := {};
				answerOne := [];
```

```
                        shortestPathAlgorithm(stationOrigin, mean, weightFactor);
                        answerOne := getPath(destination);

                        IO`println(mean);
                        IO`println("\n\n");

                        if(prev <> {|->} and stationDest.getCalculatedVariables()
(1) <> 0
                              and stationDest.getCalculatedVariables()(2) <> 0 and
stationDest.getCalculatedVariables()(3) <> 0
                              and (maxDuration < 0 or stationDest.arrivalTime <=
maxDuration)) then (

                              dcl i : nat := 0;
                              dcl prevStation : Station;
                              for el in answerOne do (

                                    IO`println(el);
                                    if(i = 0) then trip.addSegmentFirst(el.name,
el.getCalculatedVariables(), 0)
                                    else trip.addSegment(el.name,
el.getCalculatedVariables(), el.getMeanOfTransportationUsed(),
el.getAvailableSeats(transportMap.listConnections(), prevStation));

                                    prevStation := el;
                                    i := i + 1;

                              );
                              IO`println("\n\n");

                              if(trip.getSegments() <> []) then (

      trip.setFinalResults(stationDest.getCalculatedVariables(),
stationDest.arrivalTime);
                                    if(trips = []) then (
                                                trips := trips ^ [trip];)
                                    else (
                                                dcl isIn : bool := false;
                                                for t in trips do (
                                                      if(equalTrips(t, trip)
= true) then
                                                                  isIn :=
true;
                                                );
                                                if( isIn = false ) then
                                                      trips := trips ^
[trip];
                                    );
                              );
                        );

                  );

                  if(trips = []) then (
```

```
                              IO`println("There are no possible paths for your 
options.");
                  );

                  return trips;


            )
            else (

                  dcl answerSeq : seq of Station := [];
                  dcl trip : Trip := new Trip([]);
                  dcl means : set of Connection`Type := {};
                  dcl i : nat := 0;
                  dcl prevStation : Station;

                  for all m in set meansOfTransportation do(
                        if( m = 1 ) then(
                        means := means union {<Bus>})
                        else if( m = 2 ) then(
                        means := means union {<Plane>})
                        else if( m = 3 ) then(
                        means := means union {<Train>})
                        else if( m = 4 ) then(
                        means := means union {<Walk>})
                  );

                  shortestPathAlgorithm(stationOrigin, means, weightFactor);
                  answerSeq := getPath(destination);

                  for el in answerSeq do (

                              if(i = 0) then trip.addSegmentFirst(el.name, 
el.getCalculatedVariables(), 0)
                              else trip.addSegment(el.name, 
el.getCalculatedVariables(), el.getMeanOfTransportationUsed(), 
el.getAvailableSeats(transportMap.listConnections(), prevStation));

                              prevStation := el;
                              i := i + 1;
                  );

                  if(maxDuration > 0 and stationDest.arrivalTime > maxDuration) 
then (
                        IO`println("There is no path with the configurations given 
that has a smaller duration than the one given");
                        return [];
                  );

                  if(trip.getSegments() <> []) then (

      trip.setFinalResults(stationDest.getCalculatedVariables(), 
stationDest.arrivalTime);
                              trips := trips ^ [trip];
                  );
```

```
			if(trips = []) then (
					IO`println("There are no possible paths for your 
options.");
			);

			return trips;
		);

	)
	pre weightFactor = 1 or weightFactor = 2 or weightFactor = 3
	post maxDuration < 0 or checkMaximumDuration(trips,maxDuration);

	-- checks if a string is equal
	private stringEqual: Utilities`String * Utilities`String ==> bool
	stringEqual(s1, s2) ==
	(
	 if len s1 <> len s2 then
		return false;
	 for idx = 1 to len s1 do
		if s1(idx) <> s2(idx) then return false;

	 return true;
	);

	-- compare trips
	public equalTrips: Trip * Trip ==> bool
	equalTrips(trip1, trip2) == (
			dcl i : nat := 1;
			dcl seg2 : seq of Trip`Segment := trip2.getSegments();

			for s1 in trip1.getSegments() do (
					if(s1.startCity <> seg2(i).startCity or 
s1.timeDuration <> seg2(i).timeDuration or s1.distance <> seg2(i).distance or 
s1.price <> seg2(i).price or s1.meanOfTransport <> seg2(i).meanOfTransport) then
					 return false;
					i := i+1;
			);

			return true;
	);


functions

	-- checks if the graph is valid, meaning that for each connection, the 
distance and duration can't be equal to 0, the price is >= 0 and
	-- the destination and origins of each connection have to be stations in the 
transportMap
	validGraph(g : TransportGraph) res: bool ==
	(

		forall e in set g.connections & (e.distance <> 0 and e.price >= 0 and 
e.duration <> 0 and e.timetable <> []
		and e.source in set g.stations and e.destination in set g.stations)
```

```
      );

      -- checks if the station given as origin is a station in the transportMap
      validStart(sta : Station, g : TransportGraph) res: bool ==
      (
            sta in set g.stations
      );

      -- verifies if the distances are being calculated and that all stations
(nodes) have at least on connection related to them
      IsShortestPath(distances : map Station to seq of real, prev: map Station to
Station, settledNodes: set of Station, stationOrigin : Station, transportMap:
TransportGraph, meansOfTransportation : set of Type, weightFactor: nat) res: bool
==
            definesShortestDist(distances, prev, settledNodes,
stationOrigin,transportMap, meansOfTransportation, weightFactor)
            and setOfLinkedVertices(settledNodes, stationOrigin, transportMap,
meansOfTransportation);

      -- checks if the algorithm is doing the calculus related to the distances
correctly and that the for each connection there is a prev that is
      -- the option with the shortest path related to the weightFactor
      definesShortestDist(distances : map Station to seq of real, prev: map
Station to Station, settledNodes: set of Station, stationOrigin : Station,
transportMap: TransportGraph, meansOfTransportation : set of Type, weightFactor:
nat) res: bool ==
      (
            distances(stationOrigin) = [0,0,0] and
            forall sta in set settledNodes\{stationOrigin} & (exists v in set
settledNodes & (
                  prev(sta)=v and neighbour(transportMap, sta, v,
meansOfTransportation) and
                  let tup in set transportMap.connections be
                  st (tup.source = v and tup.destination = sta and tup.type =
sta.meanOfTransportationUsed)
                  in (distances(sta)(weightFactor) = distances(v)(weightFactor) +
getConnectionWeight(tup, weightFactor))
            ))
            and
            forall u1,v in set settledNodes & (neighbour(transportMap, u1, v,
meansOfTransportation) =>
                  let tup in set transportMap.connections
                  be st (tup.source = v and tup.destination = u1 and tup.type in
set meansOfTransportation)
                  in (distances(u1)(weightFactor) <= distances(v)(weightFactor) +
getConnectionWeight(tup, weightFactor)))
      );

      -- returns a connection weightFactor
      getConnectionWeight(connection: Connection, weightFactor: nat) res: real ==
getConnectionVars(connection)(weightFactor);
      -- returns the calculated variables (distance, price and duration) of a
connection
```

```
	getConnectionVars(connection: Connection) res: seq of real ==
connection.calculatedVariables;

	-- verifies that all nodes are linked to another node (at least have on
connection related to them)
	setOfLinkedVertices(settledNodes: set of Station, stationOrigin : Station,
transportMap: TransportGraph, meansOfTransportation : set of Type) res: bool ==
	(
		forall u1 in set settledNodes\{stationOrigin} & (exists v in set
settledNodes & neighbour(transportMap, u1, v, meansOfTransportation))
	);

	-- checks if the node (station) u is neighbour of node v (has a connection
between them in which the u is destination and v is source). It takes into
	-- account the means of transporation selected
	neighbour(transportMap: TransportGraph, u: Station, v: Station,
meansOfTransportation : set of Type) res: bool ==
		exists tup in set transportMap.connections & (tup.source = v and
tup.destination = u and getConnectionVars(tup) <> [0,0,0] and tup.type in set
meansOfTransportation);

	-- checks that the node chosen is indeed the node with the shortests
weightFactor
	isMinimumNode(weightFactor : nat, distances : map Station to seq of real,
minimumNode : Station, transportMap : TransportGraph, settledNodes : set of
Station, unsettledNodes : set of Station) res: bool ==
	(
		dom distances = {minimumNode}
		or
		let sta in set dom distances
		be st (sta in set settledNodes inter unsettledNodes)
		in (distances(sta)(weightFactor) >= distances(minimumNode)
(weightFactor))
	);

	-- returns finalResults of a trip
	getFinalResults(trip : Trip) res: seq of real == trip.finalResults;

	-- checks if the trips don't exceed the maximum duration
	checkMaximumDuration(trips : seq of Trip, maxDuration : real) res: bool ==
	(
		trips  = [] or
		let trip in seq trips
		in (getFinalResults(trip)(3) < maxDuration)
	);

	-- INVARIANTS
	-- If node (station) has been visited (not in the unsettledNodes) it means
that is in the settledNodes
	visitedImpliesConnected(settledNodes : set of Station, unsettledNodes : set
of Station, transportMap : TransportGraph) res: bool ==
	(
		let v = transportMap.stations\unsettledNodes in (forall u in set v & u
in set settledNodes)
	);
```

```
traces

end SearchEngine
```

3.3. Station

```
class Station
types
values

instance variables

	public name: Utilities`String;
	public arrivalTime: real := 0;
	private calculatedVariables: seq of real := [0,0,0]; -- seq of reals --> 1: distance; 2: price; 3: duration
	public meanOfTransportationUsed: Connection`Type := <NONE>;
	private seatsAvailable: nat := 0;

	inv arrivalTime >= 0;
  inv len calculatedVariables = 3;
  inv meanOfTransportationUsed = <Bus> or meanOfTransportationUsed = <Plane> or
	meanOfTransportationUsed = <Train> or meanOfTransportationUsed = <Walk> or
meanOfTransportationUsed = <NONE>;
	inv seatsAvailable >= 0;


operations

	-- constructor
	public Station: Utilities`String ==> Station
	Station(n) ==
	(
		name := n;
		return self;
	)
	post name = n;

	-- sets calculated variables
	public setCalculatedVariables: seq of real ==> ()
	setCalculatedVariables(vars) ==
	(
		calculatedVariables := vars;
	)
	post calculatedVariables = vars;

	-- set arrival Time
	public setArrivalTime: real ==> ()
	setArrivalTime(time) ==
	(
		arrivalTime := time;

	);
```

```
	-- get calculated variables
	public getCalculatedVariables: () ==> seq of real
	getCalculatedVariables() ==
	(
		return calculatedVariables;
	);

	-- sets means of transportation used to get to this station
	public setMeanOfTransportationUsed: Connection`Type ==> ()
	setMeanOfTransportationUsed(type) ==
	(
		meanOfTransportationUsed := type;
	);

	-- gets means of transportation used to get to this station
	public getMeanOfTransportationUsed: () ==> Connection`Type
	getMeanOfTransportationUsed() ==
	(
		return meanOfTransportationUsed;
	);

	-- returns available seats for the connection that leads prevStation to this 
station
	public getAvailableSeats: set of Connection * Station ==> nat
	getAvailableSeats(connections, prevStation) ==
	(
		dcl seats : nat := 0;
		for all con in set connections do (
			if(stringEqual(prevStation.name, con.source.name) and 
stringEqual(name, con.destination.name) and con.type = meanOfTransportationUsed) 
then (
				seats := con.getAvailableSeats();
			);
		);

		return seats;
	);

	-- decreases available seats in the connection used to get to this station
	public decreaseAvailableSeats: set of Connection * Station * nat ==> ()
	decreaseAvailableSeats(connections, prevStation, seatsToBuy) ==
	(
		dcl seats : nat := 0;
		for all con in set connections do (
			if(stringEqual(prevStation.name, con.source.name) and 
stringEqual(name, con.destination.name) and con.type = meanOfTransportationUsed) 
then (
				con.decreaseNumberOfSeats(seatsToBuy);
			);
		);
	);

	-- checks if a string is equal
	private stringEqual: Utilities`String * Utilities`String ==> bool
	stringEqual(s1, s2) ==
```

```
	(
	 if len s1 <> len s2 then
		return false;
	 for idx = 1 to len s1 do
		if s1(idx) <> s2(idx) then return false;

	 return true;
	);


functions

traces

end Station
```

3.4. TransportGraph

```
class TransportGraph

types
	public Type = Connection`Type;

instance variables
	public stations : set of Station;
	public connections: set of Connection;

	inv checkValidConnections(connections, stations);

operations
	-- Constructor
	public TransportGraph:() ==> TransportGraph
	TransportGraph() ==
	(
		stations := {};
		connections := {};
		createDatabase();
	);

	-- adds a new connection and possibly new stations
	private addConnection: Type * Utilities`String * Utilities`String * real *
seq of real * nat ==> ()
	addConnection(t, s, d, dist, ttbl, seats) ==
	(
		dcl tempConnection: Connection;
		dcl originStation : Station := getStationWithCreation(s);
		dcl destinationStation : Station := getStationWithCreation(d);

		tempConnection := new Connection(t, originStation, destinationStation,
dist, ttbl, seats);
		connections := connections union {tempConnection};

		stations := stations union {originStation, destinationStation};
	)
	pre dist > 0 and ttbl <> [];
```

```
-- creates new connections and stations
public createDatabase: () ==> ()
      createDatabase() ==
      (
            addConnection(<Bus>,   "Porto", "Lisbon", 300, [6, 12],12);
            addConnection(<Walk>,   "Porto", "Gaia", 10, [1, 2, 3, 4, 5, 6,
7, 8, 9, 10, 11, 12],10);
            addConnection(<Plane>, "Porto", "Lisbon", 300, [8, 22],5);
            addConnection(<Train>, "Porto", "Lisbon", 350, [10, 12, 16, 20,
22],10);
            addConnection(<Bus>,   "Porto", "Madrid", 1500, [6, 12],10);
            addConnection(<Plane>,   "Porto", "Paris", 1300, [8, 20],15);
            addConnection(<Plane>, "Amsterdam", "Bologna", 300, [6, 12],5);
            addConnection(<Plane>, "Bologna", "Paris", 2900, [9, 21],6);
            addConnection(<Bus>,   "Macedo de Cavaleiros", "Porto", 350,
[10, 12, 20],4);
            addConnection(<Train>, "Porto", "Madrid", 100, [10, 12, 16, 20,
22],19);
            addConnection(<Train>, "Lisbon", "Faro", 280, [9, 11, 15,
19],20);
            addConnection(<Plane>, "Lisbon", "Faro", 250, [8, 12, 20],90);
            addConnection(<Bus>,   "Lisbon", "Faro", 285, [8, 12],54);
            addConnection(<Bus>, "Lisbon", "Madrid", 750, [8, 12, 15,
17],12);
            addConnection(<Plane>, "Lisbon", "Madrid", 650, [8, 10, 12, 15,
19, 22],3);
            addConnection(<Train>, "Lisbon", "Madrid", 680, [8, 10, 12, 15,
17, 19, 22],5);
            addConnection(<Plane>, "Lisbon", "Barcelona", 1347, [8, 12, 15,
19, 22],2);
            addConnection(<Train>, "Madrid", "Barcelona", 625, [8, 10, 12,
15, 17, 19, 22],10);
            addConnection(<Plane>, "Madrid", "Barcelona", 625, [8, 12, 15,
19, 22],5);
            addConnection(<Train>, "Madrid", "Krakow", 1342, [8],10);
            addConnection(<Plane>, "Krakow", "Moscow", 2789, [9],5);
      );

-- lists all connections
public listConnections: () ==> set of Connection
      listConnections() == return connections;

-- lists all stations
public listStations: () ==> set of Station
      listStations() == return stations;

-- checks if 2 strings are equal
public stringEqual: Utilities`String * Utilities`String ==> bool
stringEqual(s1, s2) ==
(
 if len s1 <> len s2 then
      return false;
 for idx = 1 to len s1 do
      if s1(idx) <> s2(idx) then return false;
```

```
        return true;
    );

    -- returns connections with the station provided as origin
    public getConnectionsWithSource: Utilities`String ==> set of Connection
    getConnectionsWithSource(s) ==
    (dcl result: set of Connection;
     result := {};
        for all e in set connections do
        (
            if stringEqual(e.source.name, s) then result := result union
{e};
        );
        return result;
    );

TransportGraph    -- returns all stations that have connections with the origin
name equal to the one provided
    public getNeighborsOfNode: Utilities`String ==> set of Station
    getNeighborsOfNode(name) ==
    (
        dcl result: set of Station := {};
        for all e in set connections do
        (
            if stringEqual(e.source.name, name) then result := result union
{e.destination};
        );
        return result;
    );

    -- returns all stations that have connections with the destination name
equal to the one provided
    public getConnectionsWithDestination: Utilities`String ==> set of Connection
    getConnectionsWithDestination(d) ==
    (
        dcl result: set of Connection;
        result := {};
        for all e in set connections do
        (
            if stringEqual(e.destination.name, d) then result := result
union {e};
        );
        return result;
    );

    -- returns station with the name provided
    public getStation: Utilities`String ==> Station
    getStation(stationName) ==
    (
        for all station in set stations do (
            if stringEqual(station.name, stationName) then
                    return station;
        );
        return new Station("Error");
```

```
	);


	-- Searches for a station by name and, if it does not exist, the station is created with the specified name
	public getStationWithCreation: Utilities`String ==> Station
	getStationWithCreation(stationName) ==
	(
		dcl stationRes : Station := new Station("");
		for all station in set stations do (
			if stringEqual(station.name, stationName) then stationRes := station;
		);

		if(stringEqual(stationRes.name, "")) then (
			stationRes := new Station(stationName);
		);

		return stationRes;
	);

functions

	-- checks that all sources and destination in all connections exist in the set of stations
	public checkValidConnections(connections : set of Connection, stations : set of Station) res: bool ==
	(
		let c in set connections
		in (c.source in set stations and c.destination in set stations)
	);

traces

end TransportGraph
```

3.5. Trip

```
class Trip
types
	public Segment :: startCity : Utilities`String
			timeDuration: real
			distance: real
			price: real
			meanOfTransport: Connection`Type
			seatsAvailable: nat;

instance variables
	protected segments : seq of Segment;
	public finalResults : seq of real;
	protected availableSeatsForTrip : real := Utilities`MAX_INT;

operations

	-- constructor
```

```
	public Trip: seq of Segment ==> Trip
	Trip(segs) ==
	(
		segments := segs;
		finalResults := [];
		return self;
	)
	post segments = segs and finalResults = [];

	-- add new segment
	public addSegment: Utilities`String * seq of real * Connection`Type * nat
==> ()
	addSegment(origin, distValues, meanType, seatsAvailable) ==
	(
		dcl segment : Segment := mk_Segment(origin, distValues(3),
distValues(1), distValues(2), meanType, seatsAvailable);
		segments := segments ^[segment];
	);

	-- add new segment withouy mean of transportation info
	public addSegmentFirst: Utilities`String * seq of real  * nat ==> ()
	addSegmentFirst(origin, distValues, seatsAvailable) ==
	(
		dcl segment : Segment := mk_Segment(origin, distValues(3),
distValues(1), distValues(2), <NONE>, seatsAvailable);
		segments := segments ^[segment];
	);

	-- get segments
	public getSegments: () ==> seq of Segment
	getSegments() ==
	(
		return segments;
	);

	-- set final results
	public setFinalResults : seq of real * real ==> ()
	setFinalResults(results, arrivalTime) ==
	(
		finalResults := [results(1), results(2), arrivalTime];
	)
	post finalResults = [results(1), results(2), arrivalTime];

	-- gets number of seats available per trip
	public getAvailableSeats: () ==> real
	getAvailableSeats() ==
	(
	  availableSeatsForTrip := Utilities`MAX_INT;
		for idx = 2 to len segments do (
			if (segments(idx).seatsAvailable >= 0 and
segments(idx).seatsAvailable <= availableSeatsForTrip) then
				availableSeatsForTrip := segments(idx).seatsAvailable;
		);

		return availableSeatsForTrip;
```

```
	)
	post availableSeatsForTrip = Utilities`MAX_INT or
isMinSeatAvailable(segments, availableSeatsForTrip);

	-- decreases the available seats of the connections used in the trip
	public discountAvailableSeats: nat * TransportGraph ==> ()
	discountAvailableSeats(nrSeatsToBuy, transportMap) ==
	(
		dcl index : nat := 1;
		dcl newSegments : seq of Segment := [];

		for seg in segments do (
			if(index <> 1) then (

				dcl station : Station := getSegmentStation(transportMap,
seg.startCity);
				dcl prevStation : Station :=
getSegmentStation(transportMap, segments(index - 1).startCity);

	station.decreaseAvailableSeats(transportMap.listConnections(), prevStation,
nrSeatsToBuy);

				newSegments := newSegments ^
				[mk_Segment(seg.startCity, seg.timeDuration, seg.distance,
seg.price, seg.meanOfTransport, seg.seatsAvailable - nrSeatsToBuy)];

			) else (
				newSegments := newSegments ^ [seg];
			);
			index := index + 1;
		);
		segments := newSegments;
	);

	-- returns station correspondent to the segment
	private getSegmentStation: TransportGraph * Utilities`String ==> Station
	getSegmentStation(transportMap, stationName) ==
	(
		dcl stationRes : Station;
		for all station in set transportMap.listStations() do (
			if(stringEqual(station.name, stationName)) then (
				stationRes := station;
			)
		);

		return stationRes;
	);

	-- returns total price of the trip
	public totalPrice:() ==> real
	totalPrice() ==
	(
		return finalResults(2);
	);
```

```
	-- checks if 2 strings are equal
	private stringEqual: Utilities`String * Utilities`String ==> bool
	stringEqual(s1, s2) ==
	(
	 if len s1 <> len s2 then
		return false;
	 for idx = 1 to len s1 do
		if s1(idx) <> s2(idx) then return false;

	 return true;
	);

functions

	-- verifies that the available seats for a trip is the minimum number of
seats in all the segments
	public isMinSeatAvailable(segments : seq of Segment, availableSeatsForTrip :
real) res: bool ==
	(
		let i in set inds segments be st i > 1 in (segments(i).seatsAvailable
>=  availableSeatsForTrip)
	);


traces
end Trip
```

3.6. TicketingSystem

```
class TicketingSystem
types
	public User :: userID: nat
					passwd: nat
					    moneyAmount: real;

instance variables
	users : set of User := {};
	transportMap : TransportGraph;

  inv uniqueID(users);
operations

	-- constructor
	public TicketingSystem: TransportGraph ==> TicketingSystem
	TicketingSystem(t) ==
	(
		transportMap := t;
		createUserDatabase();
	);
	-- adds user to set of Users
	public addUser: nat * nat * real ==> ()
	addUser(id, passwd, amount) ==
	(
		users := users union {mk_User(id, passwd, amount)};
	)
```

```
	pre id >= 0 and passwd > 999 and passwd <= 9999 and amount >= 0;

	-- fills the set of users with users
	public createUserDatabase: () ==> ()
	createUserDatabase() ==
	(
		addUser(12, 1234, 1000);
		addUser(13, 5555, 9273);
		addUser(14, 8790, 7834);
	);

	-- returns user with the given ID
	private getUserById: nat ==> User
	getUserById(ID) ==
	(
		for all u in set users do
			if u.userID = ID then return u;
		return mk_User(0, 0, 0);
	);

	-- returns all users
	public getUsersDatabase: () ==> set of User
	getUsersDatabase() ==
	(
		return users;
	);

	-- discounts from the user with the userID's account
	-- the price for nrTickets tickets with the tripPrice
	private discountMoney: nat * real * nat ==> ()
	discountMoney(userID, tripPrice, nrTickets) ==
	(
		for all u in set users do (
			if u.userID = userID then
				updateDatabase(u, u.moneyAmount - tripPrice * nrTickets);
		);
	)
	pre tripPrice >= 0 and nrTickets >= 1 and validID(users,userID) and
possibleTransactionWithPrice(users, userID, tripPrice, nrTickets);

	-- changes amount of money of a user
	public updateDatabase: User * real ==> ()
	updateDatabase(u, newAmount) ==
	(
		dcl userID: nat := u.userID;
		dcl passwd: nat := u.passwd;

		users := users \ {u};
		users := users union {mk_User(userID, passwd, newAmount)};
	);

	-- verify if the user with the given userID has enough money to buy
	-- the desired amount of tickets and if the trip has enough available
	-- places and executes the payment and purchase of tickets
	public buyTickets: nat * nat * Trip * nat ==> bool
```

```
	buyTickets(userID, passwd, selectedTrip, nrSeatsToBuy) ==
	(
		IO`println(selectedTrip);
		if passwd = getUserById(userID).passwd then (
			if (getUserById(userID).moneyAmount >= selectedTrip.totalPrice()
* nrSeatsToBuy) then (
				-- there are enough seats
				dcl nrAvailableSeats : real :=
selectedTrip.getAvailableSeats();
				if (nrAvailableSeats <> Utilities`MAX_INT and
nrAvailableSeats >= nrSeatsToBuy) then (

					selectedTrip.discountAvailableSeats(nrSeatsToBuy,
transportMap);
					discountMoney(userID, selectedTrip.totalPrice(),
nrSeatsToBuy);

					return true;

				) else
					IO`print("Not enough seats available for purchase");
					return false;
			) else
				IO`print("User does not have enough money to make the
purchase");
				return false;
		) else
			IO`print("Password incorrect");
			return false;
	)
	pre possibleTransaction(users, userID, selectedTrip, nrSeatsToBuy);

functions

	-- there are no users with the same ID
	public uniqueID(users : set of User) res: bool ==
	(
		forall u in set users & (forall v in set users\{u} & (u.userID <>
v.userID))
	);

	-- checks that user with id userID exists
	public validID(users : set of User, userID : nat) res: bool ==
	(
		let u in set users
		in (u.userID = userID)
	);

	private getFinalResults(trip : Trip) res: seq of real == trip.finalResults;

	-- user has enough money to buy the desired amount of seats for the selected
trip
	public possibleTransaction(users : set of User, userID : nat, selectedTrip :
Trip, nrSeatsToBuy: nat) res: bool ==
	(
```

```
            let u in set users in (u.userID <> userID)
            or
            let u1 in set users
            be st (u1.userID = userID)
            in (u1.moneyAmount >= getFinalResults(selectedTrip)(2)*nrSeatsToBuy)
      );

      -- user has enough money to buy desired amount of seats
      public possibleTransactionWithPrice(users : set of User, userID : nat,
tripPrice : nat, nrSeatsToBuy: nat) res: bool ==
      (
            let u1 in set users
            be st (u1.userID = userID)
            in (u1.moneyAmount >= tripPrice*nrSeatsToBuy)
      );

traces

end TicketingSystem
```

3.7. Utilities

```
class Utilities
types
      public String = seq of char;

values
      public MAX_INT = 256;
instance variables
operations
functions
traces
end Utilities
```

4. Model Validation

4.1. Class Tests

```
class Tests
/*
            Class that creates and runs all the tests from Rome2RioTest and
TicketingSystemTest classes.
*/
instance variables
      public rome2Rio : Rome2RioTest := new Rome2RioTest();
      ticketingSystem : TicketingSystemTest := new TicketingSystemTest();
operations

      -- Calls the entry points of Rome2RioTest and TicketingSystemTest classes
      public create: () ==> ()
      create() == (
            rome2Rio.main();
            ticketingSystem.main();
      );

      -- Entry point that runs all tests
      public static main: () ==> ()
                  main() == (
                                    new Tests().create();
                  );

end Tests
```

4.2. Class Rome2RioTest

```
class Rome2RioTest

instance variables
      t : TransportGraph := new TransportGraph();
      s : SearchEngine := new SearchEngine(t);
operations
      -- Simulates assertion checking by reducing it to pre-condition checking.
      -- If 'cond' does not hold, a pre-condition violation will be signaled.
      private assertTrue: bool ==> ()
            assertTrue(cond) == return
      pre cond;

      -- Simulates the creation of connections and their insertion on the
database.
      -- Related to requirement R1
      private testGraphCreation: () ==> ()
      testGraphCreation() == (
            assertTrue(card s.getTransportGraph().listConnections() > 0);
      );

      -- Simulates the search of a path with the lowest price.
      -- Related to requirement R3
```

```
	private testPricePath: () ==> ()
	testPricePath() == (
		dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 2, -1);
		dcl segments : seq of Trip`Segment := answer(1).getSegments();
		dcl price : real := segments(3).price;

		assertTrue(price = 57);
		assertTrue(len segments = 3);
		assertTrue(segments(1).startCity = ['P','o','r','t','o']);
		assertTrue(segments(2).startCity = ['L','i','s','b','o','n']);
		assertTrue(segments(3).startCity = ['M','a','d','r','i','d']);
	);

	-- Simulates the search of a path with the shortest distance.
	-- Related to requirement R2
	private testDistancePath: () ==> ()
	testDistancePath() == (
		dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 1, -1);
		dcl segments : seq of Trip`Segment := answer(1).getSegments();
		dcl distance : real := segments(3).distance;

		assertTrue(distance = 950);
		assertTrue(len segments = 3);
		assertTrue(segments(1).startCity = ['P','o','r','t','o']);
		assertTrue(segments(2).startCity = ['L','i','s','b','o','n']);
		assertTrue(segments(3).startCity = ['M','a','d','r','i','d']);
	);

	-- Simulates the search of a path with the shortest duration.
	-- Related to requirement R4
	private testDurationPath: () ==> ()
	testDurationPath() == (
		dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 3, -1);
		dcl segments : seq of Trip`Segment := answer(1).getSegments();
		dcl duration : real := segments(3).timeDuration;

		assertTrue(duration = 1.25);
		assertTrue(len segments = 3);
		assertTrue(segments(1).startCity = ['P','o','r','t','o']);
		assertTrue(segments(2).startCity = ['L','i','s','b','o','n']);
		assertTrue(segments(3).startCity = ['M','a','d','r','i','d']);
	);

	-- Simulates the search of a path with a specific means of transportation.
	-- Related to requirement R5
	private testTrainPath: () ==> ()
	testTrainPath() == (
		dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {3}, 3, -1);
		dcl segments : seq of Trip`Segment := answer(1).getSegments();
		dcl duration : real := segments(2).timeDuration;
```

```
            assertTrue(duration = 1);
            assertTrue(len segments = 2);
            assertTrue(segments(1).startCity = ['P','o','r','t','o']);
            assertTrue(segments(2).startCity = ['M','a','d','r','i','d']);
      );

      -- Simulates the search of a path with a maximum trip duration, with means
of transportation.
      -- Related to requirements R2,R3,R4,R5
      private testMaxDurationOnePath: () ==> ()
      testMaxDurationOnePath() == (
                  dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 3, 1);
                  assertTrue(answer = []);
      );

      -- Simulates the search of a path with no specific means of transportation.
      -- Related to requirement R6
      private testCombinationPath: () ==> ()
      testCombinationPath() == (
                  dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {}, 3, -1);
                  dcl segments1 : seq of Trip`Segment := answer(1).getSegments();
                  dcl segments2 : seq of Trip`Segment := answer(2).getSegments();
                  dcl segments3 : seq of Trip`Segment := answer(3).getSegments();

                  assertTrue(segments1(1).startCity = ['P','o','r','t','o']);
                  assertTrue(segments1(2).startCity = ['M','a','d','r','i','d']);
                  assertTrue(segments1(2).timeDuration = 1);

                  assertTrue(segments2(1).startCity = ['P','o','r','t','o']);
                  assertTrue(segments2(2).startCity = ['L','i','s','b','o','n']);
                  assertTrue(segments2(3).startCity = ['M','a','d','r','i','d']);
                  assertTrue(segments2(3).timeDuration = 1.25);

                  assertTrue(segments3(1).startCity = ['P','o','r','t','o']);
                  assertTrue(segments3(2).startCity = ['L','i','s','b','o','n']);
                  assertTrue(segments3(3).startCity = ['M','a','d','r','i','d']);
                  assertTrue(segments3(3).timeDuration = 13.125);
      );

      -- Simulates the search of a path with a maximum trip duration, with means
of transportation.
      -- Related to requirements R2,R3,R4,R6
      private testCombinationPathWithMaximumDuration: () ==> ()
      testCombinationPathWithMaximumDuration() == (
                  dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['B','a','r','c','e','l','o','n','a'], {}, 2, 13);
                  dcl segments1 : seq of Trip`Segment := answer(1).getSegments();
                  dcl segments2 : seq of Trip`Segment := answer(2).getSegments();

                  assertTrue(segments1(1).startCity = ['P','o','r','t','o']);
                  assertTrue(segments1(2).startCity = ['M','a','d','r','i','d']);
                  assertTrue(segments1(3).startCity =
['B','a','r','c','e','l','o','n','a']);
```

```
                assertTrue(answer(1).finalResults(3) = 10.167105263157895);

                assertTrue(segments2(1).startCity = ['P','o','r','t','o']);
                assertTrue(segments2(2).startCity = ['L','i','s','b','o','n']);
                assertTrue(segments2(3).startCity = ['M','a','d','r','i','d']);
                assertTrue(segments2(4).startCity =
['B','a','r','c','e','l','o','n','a']);
                assertTrue(answer(2).finalResults(3) = 12.822368421052632);
      );

      -- Simulates the search of a path with with no possible result, to check
error behaviour.
      -- Related to requirements R2,R3,R4,R5,R6
      private testNoPath: () ==> ()
      testNoPath() == (
                dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {4}, 3, -1);
                assertTrue(answer = []);

                answer := s.rome2Rio(['G','a','i','a'],
['K','r','a','k','o','w'], {}, 3, -1);
                assertTrue(answer = []);
      );

      -- Simulates the search of a path with wrong station names, to check error
behaviour.
      -- Related to requirements R2,R3,R4,R5,R6
      private testWrongStations: () ==> ()
      testWrongStations() == (
                dcl answer : seq of Trip := s.rome2Rio(['P','e','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 3, -1);
                assertTrue(answer = []);
                answer := s.rome2Rio(['P','o','r','t','o'],
['M','e','d','r','i','d'], {1,2}, 3, -1);
                assertTrue(answer = []);
      );

      -- Simulates the search of a path with no possible transfers between
segments, to check error behaviour.
      -- Related to requirements R2,R3,R4,R5,R6
      private testArrivalTimeProblem: () ==> ()
      testArrivalTimeProblem() == (
                dcl answer : seq of Trip :=
s.rome2Rio(['M','a','d','r','i','d'], ['M','o','s','c','o','w'], {1,2,3,4}, 2, -
1);
                assertTrue(answer = []);
      );

      -- Simulates the build of connections until the destination.
      -- Related to requirements R2,R3,R4,R5,R6
      private testGetConnectionWithDestination: () ==> ()
      testGetConnectionWithDestination() == (
                dcl answer : set of Connection :=
t.getConnectionsWithDestination("Lisbon");
                assertTrue(card answer = 3);
```

```
	);

	-- Entry point that runs all the tests.
	public static main: () ==> ()
	main() == (
			new Rome2RioTest().testGraphCreation();
			new Rome2RioTest().testPricePath();
			new Rome2RioTest().testDistancePath();
			new Rome2RioTest().testTrainPath();
			new Rome2RioTest().testDurationPath();
			new Rome2RioTest().testMaxDurationOnePath();
			new Rome2RioTest().testCombinationPath();
			new 
Rome2RioTest().testCombinationPathWithMaximumDuration();
			new Rome2RioTest().testNoPath();
			new Rome2RioTest().testWrongStations();
			new Rome2RioTest().testArrivalTimeProblem();
			new Rome2RioTest().testGetConnectionWithDestination();
	);

end Rome2RioTest
```

4.3. Class TicketingSystemTest

```
class TicketingSystemTest
/*
		Class that defines the usage scenarios and test cases for the 
ticketing system.
*/
instance variables
	t : TransportGraph := new TransportGraph();
	ticket : TicketingSystem := new TicketingSystem(t);
	s : SearchEngine := new SearchEngine(t);
operations

	-- Simulates assertion checking by reducing it to pre-condition checking.
	-- If 'cond' does not hold, a pre-condition violation will be signaled.
	private assertTrue: bool ==> ()
	assertTrue(cond) == return
	pre cond;

	-- Simulates assertion checking by reducing it to pre-condition checking.
	-- If 'cond' holds, a pre-condition violation will be signaled.
	private assertFalse: bool ==> ()
	assertFalse(cond) == return
	pre not cond;

	-- Simulates the creation of a user and its insertion on the database.
	-- Related to requirement R1
	private testUserCreation: () ==> ()
	testUserCreation() == (
		assertTrue(card ticket.getUsersDatabase() > 0);
	);
```

```
	-- Simulates the bought of tickets by a user
	-- Related to requirement R7
	private testBuyTicket: () ==> ()
	testBuyTicket() == (
		dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 3, -1);
		dcl resBuy : bool := ticket.buyTickets(12, 1234, answer(1), 1);

		assertTrue(resBuy);
	);

	-- Simulates the bought of tickets by a user, with a non existing user, to
check error behaviour
	-- Related to requirement R7
	private testBuyTicketNoValidUser: () ==> ()
	testBuyTicketNoValidUser() == (
		dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 3, -1);
		dcl resBuy : bool := ticket.buyTickets(12123123, 1234, answer(1), 1);

		assertFalse(resBuy);
	);

	-- Simulates the bought of tickets by a user without enough money, to check
error behaviour
	-- Related to requirement R7
	private testBuyTicketNotEnoughMoney: () ==> ()
	testBuyTicketNotEnoughMoney() == (
		dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 3, -1);
		dcl resBuy : bool;
		ticket.addUser(12345, 1234, 10);
		resBuy := ticket.buyTickets(12345, 1234, answer(1), 1);

		assertFalse(resBuy);
	);

	-- Simulates the bought of tickets by a user, for a trip without enough
empty seats, to check error behaviour
	-- Related to requirement R7
	private testBuyTicketNotEnoughSeats: () ==> ()
	testBuyTicketNotEnoughSeats() == (
		dcl answer : seq of Trip := s.rome2Rio(['P','o','r','t','o'],
['M','a','d','r','i','d'], {1,2}, 3, -1);
		dcl resBuy : bool;
		ticket.addUser(12345, 1234, 10000000);
		resBuy := ticket.buyTickets(12345, 1234, answer(1), 20);

		assertFalse(resBuy);
	);

	-- Entry point that runs all the tests.
	public static main: () ==> ()
	main() == (
```

```
                        new TicketingSystemTest().testUserCreation();
                        new TicketingSystemTest().testBuyTicket();
                        new TicketingSystemTest().testBuyTicketNoValidUser();
                        new TicketingSystemTest().testBuyTicketNotEnoughMoney();
                        new TicketingSystemTest().testBuyTicketNotEnoughSeats();
      );

end TicketingSystemTest
```

5. Model Verification

5.1. Domain Verification Example

One of the proof obligations generated by Overture is:

No: 308

PO Name: Trip`stringEqual(Utilities`String, Utilities`String)

PO Type: Legal sequence application

(forall s1:Utilities`String, s2:Utilities`String & (idx in set (inds s1)))

The code under analysis (with the relevant sequence application underlined) is:

```
private stringEqual: Utilities`String * Utilities`String ==> bool
  stringEqual(s1, s2) ==
  (
   if len s1 <> len s2 then
       return false;
   for idx = 1 to len s1 do
       if s1(idx) <> s2(idx) then return false;
   return true;
 );
```

In this case, s1 and s2 are strings as they are transmitted as parameters and both the sequence applications in s1(idx) and s2(idx) are correct, i.e. idx does not exceed the length of s1 or s2. Proof.

s1(idx) is always a correct application of sequence, because idx goes always from 1, when the first element of s1 is accessed, to (len s1), when the last element of s1 is accessed.

Suppose s2(idx) is not a correct sequence application, i.e. idx > len s2. But idx takes values from 1 to (len s1), so it can be maximum (len s1). Therefore, (len s1) > (len s2), but the for loop is executed just of (len s1) is equal to (len s2), because otherwishe the operation executes “return false”. In conclusion, s2(idx) is a correct sequence aplication as well.

5.2. Invariant Verification Example

Another proof obligation generated by Overture is:

No: 112

Name: SearchEngine`shortestPathAlgorithm(Station, set of Connection`Type, nat)

Type: state invariant holds

The code under analysis (with the relevant state changes underlined) is:

```
public shortestPathAlgorithm: Station * set of Connection`Type * nat ==> ()
shortestPathAlgorithm(origin, meansOfTransportation, weightFactor) ==
(
  distances := distances ++ {origin |-> [0,0,0]};
  origin.setArrivalTime(0);
  settledNodes := settledNodes union {origin};
  while(settledNodes inter unsettledNodes <> {}) do (
  dcl minimumNode : Station := getMinimumNode(weightFactor);
  settledNodes := settledNodes union {minimumNode};
  unsettledNodes := unsettledNodes \ {minimumNode};
  findMinimalDistances(minimumNode, meansOfTransportation, weightFactor);
  );
 )
pre (weightFactor = 1 or weightFactor = 2 or weightFactor = 3) and validGraph(transportMap) and validStart(stationOrigin, transportMap)
post IsShortestPath(distances, prev, settledNodes, stationOrigin, transportMap, meansOfTransportation, weightFactor);
```

The relevant invariant under analysis is, which states that if a node (station) has been visited (not in the unsettledNodes) it means that it is in the settledNodes set:

inv visitedImpliesConnected(settledNodes, unsettledNodes, transportMap);

with the implementation:

visitedImpliesConnected(settledNodes : set of Station, unsettledNodes : set of Station, transportMap : TransportGraph) res: bool ==

(

let v = transportMap.stations\unsettledNodes in (forall u in set v & u in set settledNodes)

);

After the execution of the addUser function block we have (technically, this is the post-condition of the block) that the distances from stationOrigin are calculated and that all stations (nodes) have at least on connection related to them:

IsShortestPath(distances, prev, settledNodes, stationOrigin, transportMap, meansOfTransportation, weightFactor).

When the shortestPathAlgorithm finishes its execution, it means that the while loop finished because (settledNodes inter unsettledNodes) became empty. This means that no node that is connected (so in settledNodes) can be found in the unsettledNodes set, the set with not visited nodes. Therefore, the invariant holds.

6. Code Generation

After concluding the VDM++ code, we generated Java code from the project, with Overture. After fixing some minor problems in the Java code, we tested the functionalities of the project and everything worked as supposed. Then we built a simple user interface on the terminal, so that the user can interact with the system with a user friendly menu, instead of having to call the functions with the right arguments. The interface shows the possibilities to complete all requirements, with the use cases referred in section 2.1.

7. Conclusions

After finishing the project, the group is happy with the results, since all the requirements were covered by the model developed, and we developed more than we initially predicted. Instead of using brute force to build the paths for the searches, we implemented a modified version of Dijkstra's algorithm, to have more efficiency and optimization. We also implemented an algorithm to limit the waiting time between connections (transfers), to fit all users' preferences.

If we had more time, one thing that could be improved would be the user interface, since all interactions with the user are done on a terminal. It would be better and more user friendly to have a GUI to interact with the user. We could also have tried to show more than one path for each search, for example the 5 best paths.

The work load was well split between the group members and we all completed our tasks.

8. References

- All the materials on MFES' moodle
- http://overturetool.org/documentation/tutorials.html
- https://moodle.up.pt/pluginfile.php/25564/mod_resource/content/2/VDM10_lang_man.pdf
- http://lausdahl.github.io/overturetool.github.io/files/OvertureIDEUserGuide.pdf
- http://www3.risc.jku.at/publications/download/risc_5224/Master_Thesis.pdf