Formal Modeling of "Rome2Rio" in VDM++

Mestrado Integrado em Engenharia Informática e Computação, MIEIC

Métodos Formais em Engenharia de Software, MFES

Nádia de Sousa Varela de Carvalho, up201208223, ei12047@fe.up.pt

Tiago Lascasas dos Santos, up201503616, up201503616@fe.up.pt

7th January 2019

Contents

1. Informal system description and list of requirements

1.1 Informal system description

The system we pretend to model is based on Rome2rio, a Web Application that allows people to specify an origin and a destination on a map and then get the most efficient routes between those two points using different criteria.

The routes can take into consideration the type of transportation preferred and an optimization criterium, such as the overall price, duration or distance.

Finally, the application also has the need for administrators who help maintain and update the set of available locations and the connections between them.

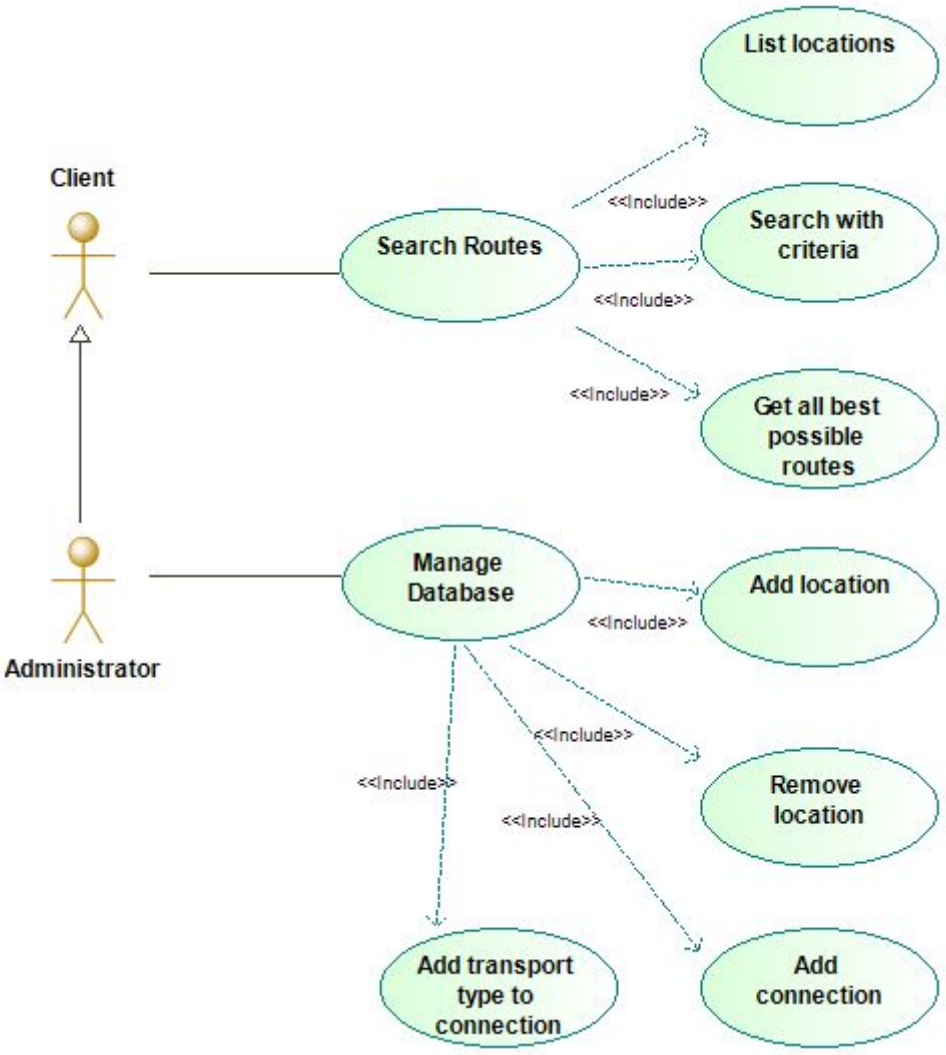
1.2 List of requirements

Id	Priority	Description
R1	Mandatory	The Administrator can add locations. Each location should have a unique name and geographic coordinate, and also have a country.
R2	Mandatory	The Administrator can add connections between locations, often with more than one type of transportation. For each connection, there can only be one of each transportation type, and they should all have information about the price, estimated time and distance.
R3	Mandatory	There can only be a single straight connection between A and B, and that does not imply the existence of a connection from B to A - if the connection is intended to be bidirectional, an additional connection from B to A should be created as well.
R4	Mandatory	The available transportation types should be by car, plane, train, bus or ferry.
R5	Mandatory	The Administrator should be able to authenticate himself using an editable password.
R6	Mandatory	The Client can list the possible locations available.
R7	Mandatory	The Client can fetch the most efficient route between two locations, by entering source and destination cities and by specifying the optimization criterium (time, distance or price) and, optionally, the transportation type.
R8	Mandatory	The Client can obtain a list with the most efficient route using all combinations of transportation type and optimization criteria.

These requirements are directly translated onto use cases as shown next.

2. Visual UML model

2.1 Use case model



The major use case scenarios (to be used later as test scenarios) are described next. They have been devised from the requirements and refined into nearly atomic operations, each of which directly translatable into a high-level operation provided by the model.

Scenario	**UC1 - Login as Administrator**
Description	An administrator should be able to enter the system in order to have privileged access
Pre-conditions	1. The current user status should be Client
Post-conditions	1. The current user status should be changed to Administrator
Steps	1. The admin should enter the password 2. If the password matches the saved one, access is granted

Scenario	UC2 - Change the Administrator password
Description	The Administrator can change its access password
Pre-conditions	1. The current user status is Administrator
Post-conditions	1. The password is changed to a new one
Steps	1. The admin enters the new password 2. If the new password is not an empty string, it is altered

Scenario	UC3 - Add a location
Description	The Administrator can add a new location to the system, providing an unique identifier and geographical coordinate, as well as the country
Pre-conditions	1. The current user status is Administrator 2. There exists no location with the same name or coordinate 3. The coordinates are within boundaries
Post-conditions	1. The system has one more location in record
Steps	1. The admin enters the new location 2. If either the coordinate or the location is a duplicate, it fails and nothing is changed

Scenario	UC4 - Add a connection between two existing locations
Description	Normal scenario for adding routes between two locations and respective information for one transportation type (duration, distance, price).
Pre-conditions	1. The current user status is Administrator 2. There is yet no route between the two locations
Post-conditions	3. There is now a directed edge on the graph connecting the first location to the second
Steps	1. The admin enters the new connection and its travel type 2. If a connection already existed, it fails and nothing is changed

Scenario	UC5 - Add a new transportation type to an existing connection
Description	The Administrator can add another transportation type to an existing connection
Pre-conditions	1. The current user status is Administrator 2. The edge to alter already exists 3. The edge still doesn't have this transportation type set
Post-conditions	1. The edge has one more distinct transportation type
Steps	1. The admin enters the new transportation type 2. If the new type is not valid, it fails and nothing is altered

Scenario	UC6 - Remove a location
Description	The Administrator can delete an existing location
Pre-conditions	1. The current user status is Administrator 2. The location to remove exists
Post-conditions	1. There is one less location in the system 2. All connections to and from the location were eliminated
Steps	1. The admin enters the location to remove 2. If the provided location does not exist, it fails and nothing is altered

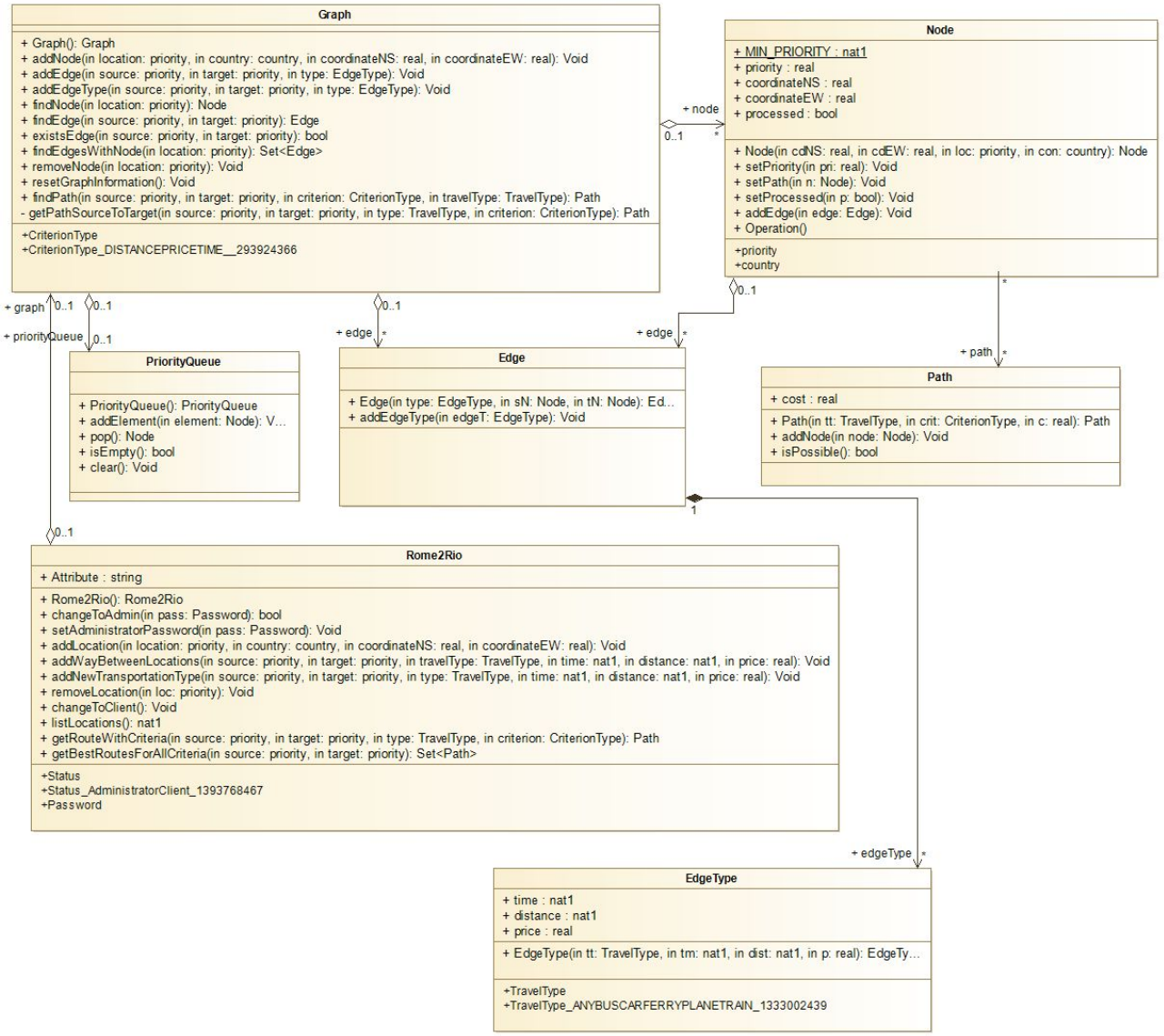
Scenario	**UC7 - Logout as Administrator**
Description	The Administrator should logout in order to change the user type to Client
Pre-conditions	1. The current user status is Administrator
Post-conditions	1. The current user status is Client
Steps	1. The admin logs out of the system 2. If the user status wasn't Administrator, it fails and nothing is altered

Scenario	**UC8 - List all locations**
Description	An User (Client or Administrator) can list all existing locations
Pre-conditions	1. There must be at least one location in the system
Post-conditions	(none)
Steps	1. The User lists the locations 2. If there are no locations, nothing is done

Scenario	**UC9 - Search for the best route given a criteria and a transportation preference**
Description	Search scenario, in which the User searches for the most efficient route between two locations using an optimization criterium (time, price or distance) and, optionally, a single transport type.
Pre-conditions	1. The source and target destinations exist 2. The graph has all information related to shortest path algorithms fully reset before starting
Post-conditions	1. The graph has information about the last execution of a shortest path algorithm on it
Steps	1. The User specifies the source, target and criterium 2. Optionally, it can specify a transportation type. If it doesn't, the type <ANY> is picked 3. If it was possible to reach the target from the source, the shortest path and total cost is presented. Otherwise, it is conveyed the information that it is impossible to find a path

Scenario	**UC10 - Get the set of all optimal routes using all available criteria and transportation types**
Description	Alternative search scenario, in which the user searches for routes between two locations specifying dates and all optimal paths, using different criteria and transportation types, are shown
Pre-conditions	1. The source and target destinations exist 2. The graph has all information related to shortest path algorithms fully reset before starting
Post-conditions	1. The graph has information about the last execution of a shortest path algorithm on it
Steps	1. The User specifies the source and target 2. All shortest paths using the combination of all criteria are shown. These can go from 0 (impossible to reach the target from the source) to 18 (it was possible to achieve the target from the source on all combinations)

2.2 Class model [1] [2]



1

2

Class	Description
`Edge`	Defines a directed edge and the functions to work with all the routes between two locations.
`EdgeType`	Defines a specific type of transportation inside of an edge, including all its parameters, such as estimated time, distance and price.
`Node`	Defines a node (location), together with its outward edges and location information (geographic coordinates and country).
`Path`	Defines a Path in between locations using a specific criterium.
`PriorityQueue`	Defines a support priority queue for the graph to use when calculating routes using Dijkstra’s Shortest Path algorithm (queue ordered by the element with least priority, rather than by the one with highest).
`Graph`	Defines the system graph, in which all locations and routes are defined and includes functions to deal with calculating all and also shortest paths (implementation of Dijkstra’s Shortest Path algorithm) between two locations.
`Rome2Rio`	Core model; defines the state variables and operations available to the users (both normal end users and administrators).
`MyTestCase`	Superclass for test classes; defines `assertEquals` and `assertTrue`. It was taken from the provided example report.
`TestRome2Rio`	Defines the test/usage scenarios and test cases for the Rome2Rio system.

3. Formal VDM++ model

3.1 Class Edge

```
-- Class to represent a directed edge in the travel graph
-- On this context, an edge is a connection between two locations
class Edge
instance variables
  public edgeType:set of (EdgeType);     -- a set for all the types of the edge
  public sourceNode:Node;               -- the source node
  public targetNode:Node;               -- the target node

  -- invariant: there can not be two of the same EdgeTypes for a single edge
  inv not exists e1, e2 in set edgeType & e1 <> e2 and e1.travelType = e2.travelType;

operations
 -- Constructor
    public Edge : EdgeType * Node * Node ==> Edge
    Edge(type,sN,tN) ==
    (
       edgeType := {type};
       sourceNode := sN;
       targetNode := tN;
       return self;
    )
    pre sN <> tN;

    -- Adds a type to this edge
    public addEdgeType : EdgeType ==> ()
    addEdgeType(edgeT) ==
      edgeType := edgeType union {edgeT}

end Edge
```

3.2 Class EdgeType

```
-- Class to represent an edge type
-- On this context, an edge type is a means of transport that exists on a certain
-- connection (edge), and it holds info about the distance, duration and price
class EdgeType

types
    -- All the transportation types, along with a generic type <ANY>
  public TravelType = <ANY> | <CAR> | <BUS> | <PLANE> | <TRAIN> | <FERRY>;

instance variables
  public travelType:TravelType;                    -- the means of transport
  public time:nat1;                                -- the time in hours
  public distance:nat1;                            -- the distance in km
  public price:real;                               -- the price in euro

operations

 -- Constructor
```

```
    public EdgeType : TravelType * nat1 * nat1 * real ==> EdgeType
    EdgeType(tt,tm,dist,p) ==
    (
       travelType := tt;
       time := tm;
       distance := dist;
       price := p;
       return self;
    )
    pre tt <> <ANY>;

end EdgeType
```

3.3 Class Node

```
-- Class to represent a node in the travel graph
-- On this context, a node is a geographical location
class Node

types
    public Location = seq1 of (char);    -- type for a location name
    public Country = seq1 of (char)      -- type for a country name

values
    public
      MIN_PRIORITY = 9999999999          -- minimal priority a node can have
                                         -- (the lesser, the more prioritary)

instance variables
  public priority:real;                  -- priority of the node
  public coordinateNS:real;              -- normalized North-South coordinate
  public coordinateEW:real;              -- normalized East-West coordinate
  public location:Location;              -- location name
  public country:Country;                -- country it belongs to
  public outwardEdges:set of (Edge);     -- edges that have this node as its source
  public path:Node;                      -- node that precedes this one
  public processed:bool;                 -- flag to tell whether the node was processed
                                         -- during Dijkstra's algorithm
operations
  -- Constructor
    public Node : real * real * Location * Country ==> Node
    Node(cdNS, cdEW, loc, con) ==
    (
       priority := MIN_PRIORITY;
       coordinateNS := cdNS;
       coordinateEW := cdEW;
       location := loc;
       country := con;
       outwardEdges := {};
       processed := false;
       path := self;
       return self;
    )
    pre cdNS >= -90.0 and
          cdNS <= 90.0 and
          cdEW >= -180.0 and
```

```
        cdEW <= 180.0;

    -- Sets the priority of the node
    public setPriority : real ==> ()
    setPriority(pri) ==
      priority := pri
    pre pri >= 0.0 and pri <= MIN_PRIORITY;

    -- Sets the path of the node
    public setPath : Node ==> ()
    setPath(n) ==
      path := n;

    -- Changes the processed flag
    public setProcessed : bool ==> ()
    setProcessed(p) ==
      processed := p;

    -- Adds an edge with this node as source
    public addEdge : Edge ==> ()
    addEdge(edge) ==
      outwardEdges := {edge} union outwardEdges;

end Node
```

3.4 Path

```
-- Class to represent a path made of nodes and edges
-- On this context, it is the shortest path from A to B
-- and in which the "distance" is based on specified criteria
class Path

instance variables
    public path: seq of (Node);                          -- sequence with the path nodes
    public travelType: EdgeType`TravelType;              -- method of transportation
    public criteria: Graph`CriterionType;                -- edge weight criterion
    public cost: real;                                   -- final cost of the path

operations
    -- Constructor
    public Path: EdgeType`TravelType * Graph`CriterionType * real ==> Path
    Path(tt, crit, c) ==
    (
      path := [];
      travelType := tt;
      criteria := crit;
      cost := c;
    );

    -- Adds a new node to the head of the path
    public addNode: Node ==> ()
    addNode(node) ==
      path := [node] ^ path
    post (hd path) = node;

    -- Returns whether the path is set or not
    public isPossible: () ==> bool
```

```
    isPossible() ==
      return (len path) > 1;

end Path
```

3.5 PriorityQueue

```
-- Class to represent a priority queue
-- On this context, it always pops the node with the smallest priority,
-- as Dijkstra's algorithm always picks the node with the least weight
class PriorityQueue

instance variables
  public elements: set of (Node)        -- set of nodes in the queue

operations
  -- Constructor
  public PriorityQueue: () ==> PriorityQueue
  PriorityQueue() ==
  (
      elements := {};
      return self;
  );

  -- Adds a node to the queue, checking if it already exists
  public addElement : Node ==> ()
  addElement(element) ==
   elements := elements union {element}
  pre element not in set elements
  post element in set elements;

  -- Pops the lowest priority element from the queue, and returns it
  public pop : ()  ==> Node
  pop() ==
  (
   dcl res: Node;
   dcl min: real := 9999999999;
      for all e in set elements do
      (
        if e.priority < min then
        (
        res := e;
        min := e.priority;
        );
      );
      elements := elements \ {res};
      return res;
  )
  pre card elements > 0;

  -- Checks if the queue is empty
  public isEmpty : () ==> bool
  isEmpty() ==
   return card elements = 0;

  -- Removes all elements from the queue
  public clear : () ==> ()
  clear() ==
```

```
   elements := {}
  post card elements = 0;

end PriorityQueue
```

3.6 Graph

```
/** Represents a travel network, and runs Dijkstra’s algorithm over it **/
class Graph

types
    -- variable to minimize in the shortest path algorithm
    public CriterionType = <DISTANCE> | <TIME> | <PRICE>;

instance variables
  public node:set of (Node);                    -- set with all nodes
  public priorityQueue:PriorityQueue;           -- priority queue for Dijkstra's algorithm
  public edge:set of (Edge);                    -- set with all edges

  -- invariant: there are no repeated locations
  inv not exists n1, n2 in set node &
             n1 <> n2 and n1.location = n2.location;

  -- invariant: there are no repeated coordinate pairs (NS, EW)
  inv not exists n1, n2 in set node &
      (n1 <> n2 and
             n1.coordinateNS = n2.coordinateNS and
                        n1.coordinateEW = n2.coordinateEW);

operations
  -- Constructor
    public Graph: () ==> Graph
    Graph() ==
    (
        node := {};
        priorityQueue := new PriorityQueue();
        edge := {};
        return self;
    );

  -- Adds a new node to the graph
  public addNode : Node`Location * Node`Country * real * real ==> ()
  addNode(location, country, coordinateNS, coordinateEW) ==
  (
      dcl n: Node := new Node(coordinateNS, coordinateEW, location, country);
      node := {n} union node;
  );

  -- Adds an edge to two existing unconnected nodes
  public addEdge : Node`Location * Node`Location * EdgeType ==> ()
  addEdge(source, target, type) ==
  (
```

```
        dcl s: Node := findNode(source);
        dcl t: Node := findNode(target);
        dcl e: Edge := new Edge(type, s, t);
        edge := {e} union edge;
        s.addEdge(e);
    )
    pre not exists e1 in set edge &
        source = e1.sourceNode.location and
        target = e1.targetNode.location;

    -- Adds an edge type to an existing edge
    public addEdgeType : Node`Location * Node`Location * EdgeType ==> ()
    addEdgeType(source, target, type) ==
    (
        dcl e: Edge := findEdge(source, target);
        e.addEdgeType(type);
    )
    pre not exists e in set edge
          & e.sourceNode.location = source
          and e.targetNode.location = target
          and type in set e.edgeType;

    -- Gets a node of the node set by location
    pure public findNode : Node`Location ==> Node
    findNode(location) ==
    (
        dcl res: Node;
        for all n in set node do if n.location = location then res := n;
        return res;
    )
    pre exists1 n in set node & n.location = location;

    -- Gets an edge given the source and target
    public findEdge : Node`Location * Node`Location  ==> Edge
    findEdge(source, target) ==
    (
        dcl res: Edge;
        for all e in set edge do if e.sourceNode.location = source and
e.targetNode.location = target then res := e;
        return res;
    )
    pre exists1 e in set edge & e.sourceNode.location = source and e.targetNode.location =
target;

    -- Checks if exists an edge given source and target
    pure public existsEdge : Node`Location * Node`Location  ==> bool
    existsEdge(source, target) ==
        return (exists e in set edge & e.sourceNode.location = source and
e.targetNode.location = target);

    -- Finds all edges that come to or from the given nodes
```

```
  pure public findEdgesWithNode : Node`Location ==> set of (Edge)
  findEdgesWithNode(location) ==
  (
      dcl res : set of (Edge);
      res := {};
      for all e in set edge do if e.sourceNode.location = location or
e.targetNode.location = location then res := res union {e};
      return res
  );

  -- Removes an existing node from the graph
  public removeNode : Node`Location ==> ()
  removeNode(location) ==
  (
      dcl n: Node := findNode(location);
      dcl edgesWithNode :  set of (Edge) := findEdgesWithNode(location);
      edge := edge \ edgesWithNode;
      node := node \ {n};
  )
  pre exists n1 in set node & n1.location = location;

    -- Resets the Dijkstra information on all nodes in order for the algorithm to be run
again
    public resetGraphInformation : () ==> ()
    resetGraphInformation() ==
    (
        priorityQueue.clear();
        for all n in set node do
        (
              n.setPriority(Node`MIN_PRIORITY);
              n.setProcessed(false);
              n.setPath(n);
        );
    )
    post forall n in set node &
        (n.priority = Node`MIN_PRIORITY and
              not n.processed and
              n.path.location = n.location);

  -- Finds the shortest path between two nodes using the specified criteria using
Dijkstra's Shortest Path algorithm
  public findPath : Node`Location * Node`Location * CriterionType * EdgeType`TravelType
==> Path
  findPath(source, target, criterion, travelType) ==
  (
      dcl foundTarget: bool := false;
      dcl startingNode: Node := findNode(source);

      -- reset priorities, paths and processed information
      resetGraphInformation();
```

```
	-- get the starting node and add it to the queue
	startingNode.setPriority(0);
	priorityQueue.addElement(startingNode);

	-- process nodes until the queue is empty or the target node is found
	while priorityQueue.isEmpty() <> true and foundTarget <> true do
	(
	  -- get the lowest priority node
	  dcl currentNode: Node := priorityQueue.pop();

	  -- if the node is the target, set the flag to true
	  if currentNode.location = target then
	        foundTarget := true
	  else
	  (
	        -- else, go through all the outward edges of this node
	        for all outwardEdge in set currentNode.outwardEdges do
	        (
	              for all type in set outwardEdge.edgeType do
	              (
	                dcl weight: real := 0;
	                dcl destNode: Node := outwardEdge.targetNode;

	            -- for each edge, see if it is relevant according to the travel type
	            -- if it isn't, the edge is ignored as if it didn't exist
	                        if type.travelType = travelType or travelType = <ANY> then
	                        (
	                              -- get the node weight based on the search criterium
	                              if criterion = <TIME> then weight := type.time
	                              elseif criterion = <DISTANCE>
	                                    then weight := type.distance
	                              else weight := type.price;

	              -- if the path from the current node plus weight is
-- lesser than the edge's
	              -- target weight, replace the edge target's weight by
-- this one and add the
	              -- edge's target to the priority queue
	                              if currentNode.priority + weight < destNode.priority
then
	                              (
	                                    destNode.setPriority(currentNode.priority +
	                                                                  weight);
	                                    destNode.setPath(currentNode);
	                                    if destNode.processed = false then
	                                    (
	                                          destNode.setProcessed(true);
	                                          priorityQueue.addElement(destNode);
	                                    );
	                              );
	                        );
	                  );
```

```
            );
        );
      );
      -- if the target was found, get the path from target to source
      -- returns the path if the target was found, or an empty sequence if it wasn't
      if foundTarget = true then
        return getPathSourceToTarget(source, target, travelType, criterion)
      else return new Path(travelType, criterion, -1);
  )
  pre exists n1, n2 in set node & n1.location = target and n2.location = source
  post forall n in seq RESULT.path &
                    n.priority <> Node`MIN_PRIORITY;    -- non-trivial post condition


  -- Gets a sequence holding the sequence of nodes that form the shortest path between
  -- the given source and target
  private getPathSourceToTarget : Node`Location * Node`Location * EdgeType`TravelType *
CriterionType ==> Path
  getPathSourceToTarget(source, target, type, criterion) ==
    (
        dcl currentNode: Node := findNode(target);
        dcl path: Path := new Path(type, criterion, currentNode.priority);

        while currentNode.location <> source do
        (
              path.addNode(currentNode);
              currentNode := currentNode.path;
        );
        path.addNode(currentNode);

        return path;
    );

end Graph
```

3.7 Rome2Rio

```
class Rome2Rio
/*
  Contains the core model of the Rome2Rio System.
  Defines the state variables and all the operations
  required to execute all use cases
*/
types
    public Status= <Administrator> | <Client>;      -- types of user
  public Password = seq of char;                    -- type for the admin password

instance variables
  public graph:Graph;                               -- the travel graph
  public status : Status;                           -- the current user's type
  -- Items observable by the administrator:
  public adminCode : Password := "default";         -- the current admin password
```

```
operations
    -- Constructor
    public Rome2Rio : () ==> Rome2Rio
    Rome2Rio() ==
    (
        graph := new Graph();
        status := <Client>;
        return self;
    );

    -- UC1: Changes the type of user to admin
    public changeToAdmin : Password ==> bool
    changeToAdmin(pass) ==
    (
        if pass = adminCode then
        (
              status := <Administrator>;
              return true;
        )
        else return false;
    )
    pre status = <Client>;

    /** ADMINISTRATOR OPERATIONS **/

    -- UC2: Changes the admin password
    public setAdministratorPassword : Password ==> ()
    setAdministratorPassword(pass) ==
        adminCode := pass
    pre status = <Administrator>;

    -- UC3: Adds a new location to the platform
    public addLocation : Node`Location * Node`Country * real * real ==> ()
    addLocation(location, country, coordinateNS, coordinateEW) ==
       graph.addNode(location, country, coordinateNS, coordinateEW)
    pre status = <Administrator>;

    -- UC4: Adds a connection between two locations
    public addWayBetweenLocations : Node`Location *  Node`Location * EdgeType`TravelType
* nat1 * nat1 * real ==> ()
    addWayBetweenLocations(source, target, travelType, time, distance, price) ==
    (
      if(graph.existsEdge(source, target) = false) then
              graph.addEdge(source, target, new EdgeType(travelType, time, distance,
price))
    )
    pre status = <Administrator>;

    -- UC5: Adds a new transportation type to an existing connection
    public addNewTransportationType : Node`Location * Node`Location *
EdgeType`TravelType * nat1 * nat1 * real ==> ()
    addNewTransportationType(source, target, type, time, distance, price) ==
```

```
    (
        dcl et: EdgeType := new EdgeType(type, time, distance, price);
        graph.addEdgeType(source, target, et);
    )
    pre type <> <ANY> and status = <Administrator>;

    -- UC6: Removes a location, along with all its connections
    public removeLocation : Node`Location ==> ()
    removeLocation(loc) ==
        graph.removeNode(loc)
    pre status = <Administrator>;

    /** CLIENT OPERATIONS **/

    -- UC7: Changes the type of user to client
    public changeToClient : () ==> ()
    changeToClient() ==
        status := <Client>
    pre status = <Administrator>;

    -- UC8: Lists all available locations
  public  listLocations : ()  ==> nat1
  listLocations() ==
  (
      dcl i: nat := 0;
      for all n in set graph.node do
      (
        IO`print(n.location);
        IO`print(" at (");
        IO`print(n.coordinateNS);
        IO`print(", ");
        IO`print(n.coordinateEW);
        IO`println(")");
        i := i + 1;
      );
      return i;
  );

  -- UC9: calculates the best route given the travel type and optimization criterion
  public getRouteWithCriteria : Node`Location * Node`Location * EdgeType`TravelType *
Graph`CriterionType ==> Path
    getRouteWithCriteria(source, target, type, criterion) ==
        return graph.findPath(source, target, criterion, type);

  -- UC10: gets the best paths using each of the availabe criteria and travel types
  public getBestRoutesForAllCriteria : Node`Location *  Node`Location ==> set of (Path)
  getBestRoutesForAllCriteria(source, target) ==
  (
      dcl paths: set of (Path) := {};

      paths := paths union {graph.findPath(source, target, <PRICE>, <ANY>)};
      paths := paths union {graph.findPath(source, target, <DISTANCE>, <ANY>)};
```

```
        paths := paths union {graph.findPath(source, target, <TIME>, <ANY>)};

        paths := paths union {graph.findPath(source, target, <PRICE>, <CAR>)};
        paths := paths union {graph.findPath(source, target, <DISTANCE>, <CAR>)};
        paths := paths union {graph.findPath(source, target, <TIME>, <CAR>)};

        paths := paths union {graph.findPath(source, target, <PRICE>, <BUS>)};
        paths := paths union {graph.findPath(source, target, <DISTANCE>, <BUS>)};
        paths := paths union {graph.findPath(source, target, <TIME>, <BUS>)};

        paths := paths union {graph.findPath(source, target, <PRICE>, <PLANE>)};
        paths := paths union {graph.findPath(source, target, <DISTANCE>, <PLANE>)};
        paths := paths union {graph.findPath(source, target, <TIME>, <PLANE>)};

        paths := paths union {graph.findPath(source, target, <PRICE>, <FERRY>)};
        paths := paths union {graph.findPath(source, target, <DISTANCE>, <FERRY>)};
        paths := paths union {graph.findPath(source, target, <TIME>, <FERRY>)};

        paths := paths union {graph.findPath(source, target, <PRICE>, <TRAIN>)};
        paths := paths union {graph.findPath(source, target, <DISTANCE>, <TRAIN>)};
        paths := paths union {graph.findPath(source, target, <TIME>, <TRAIN>)};

        for all p in set paths do
          if not p.isPossible() then paths := paths \ {p};

        return paths;
    );

end Rome2Rio
```

4. Model validation

4.1 Class MyTestCase

```
class MyTestCase
/*
  Superclass for test classes, simpler but more practical than VDMUnit`TestCase.
  For proper use, you have to do: New -> Add VDM Library -> IO.
  JPF, FEUP, MFES, 2014/15.
*/

operations

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

   -- Simulates assertion checking by reducing it to post-condition checking.
   -- If values are not equal, prints a message in the console and generates
   -- a post-conditions violation.
```

```
   protected assertEqual: ? * ? ==> ()
   assertEqual(expected, actual) ==
      if expected <> actual then (
         IO`print("Actual value (");
         IO`print(actual);
         IO`print(") different from expected (");
         IO`print(expected);
         IO`println(")\n")
      )
   post expected = actual

end MyTestCase
```

4.2 Class TestRome2Rio

```
class TestRome2Rio is subclass of MyTestCase

instance variables
  public r2r: Rome2Rio;

operations
    /** UNIT TESTS **/

    -- Tests the insertion of nodes, edges and edge types on the graph
    public testAddNodesAndEdges: () ==> ()
    testAddNodesAndEdges() ==
    (
      dcl graph: Graph := new Graph();

      graph.addNode("L1", "C1", 1.0, 1.0);
      graph.addNode("L2", "C2", 1.0, 2.0);
      graph.addNode("L3", "C2", 5.0, 2.0);
      assertEqual(3, card graph.node);

      graph.addEdge("L1", "L2", new EdgeType(<CAR>, 1, 1, 1.0));
      graph.addEdge("L1", "L3", new EdgeType(<FERRY>, 10, 2, 2.0));
      assertEqual(2, card graph.findNode("L1").outwardEdges);
      graph.addEdge("L3", "L2", new EdgeType(<PLANE>, 100, 3, 3.0));
      graph.addEdge("L2", "L1", new EdgeType(<BUS>, 1000, 4, 4.0));
      graph.addEdge("L2", "L3", new EdgeType(<TRAIN>, 10000, 5, 5.0));
      assertEqual(5, card graph.edge);

      assertEqual("L1", graph.findEdge("L1", "L2").sourceNode.location);
      assertEqual("L2", graph.findEdge("L1", "L2").targetNode.location);

      graph.addEdgeType("L1", "L2", new EdgeType(<FERRY>, 1, 1, 1.0));
      assertEqual(2, card graph.findEdge("L1", "L2").edgeType);
      graph.addEdgeType("L1", "L2", new EdgeType(<PLANE>, 1, 1, 1.0));
      assertEqual(3, card graph.findEdge("L1", "L2").edgeType);
      graph.addEdgeType("L1", "L2", new EdgeType(<BUS>, 1, 1, 1.0));
      assertEqual(4, card graph.findEdge("L1", "L2").edgeType);
      graph.addEdgeType("L1", "L2", new EdgeType(<TRAIN>, 1, 1, 1.0));
      assertEqual(5, card graph.findEdge("L1", "L2").edgeType);
    );

    -- Tests the devised priority queue
    public testPriorityQueue: () ==> ()
```

```
testPriorityQueue() ==
(
  dcl pq: PriorityQueue := new PriorityQueue();
  dcl node1: Node := new Node(5.1, 3.1, "L1", "C1");
  dcl node2: Node := new Node(5.2, 3.2, "L2", "C1");
  dcl node3: Node := new Node(5.3, 3.3, "L3", "C2");
  dcl node4: Node := new Node(5.4, 3.4, "L4", "C2");
  node1.setPriority(50);
  node2.setPriority(100);
  node3.setPriority(3);
  node4.setPriority(20);
  pq.addElement(node1);
  pq.addElement(node2);
  pq.addElement(node3);
  pq.addElement(node4);
  assertEqual(4, card pq.elements);
  assertEqual(node3, pq.pop());
  assertEqual(node4, pq.pop());
  assertEqual(node1, pq.pop());
  assertEqual(node2, pq.pop());
  assertEqual(true, pq.isEmpty());
  pq.addElement(node1);
  pq.addElement(node2);
  assertEqual(false, pq.isEmpty());
  pq.clear();
  assertEqual(true, pq.isEmpty());
);

-- (Not a test) builds a graph with enough information to run the
-- shortest path algorithm using multiple criteria and travel types
public makeGraph: () ==> Graph
makeGraph() ==
(
  dcl graph: Graph := new Graph();

  graph.addNode("L1", "C1", 1.0, 1.0);
  graph.addNode("L2", "C1", 2.0, 2.0);
  graph.addNode("L3", "C1", 3.0, 3.0);
  graph.addNode("L4", "C2", 4.0, 4.0);
  graph.addNode("L5", "C2", 5.0, 5.0);
  graph.addNode("L6", "C2", 6.0, 6.0);
  graph.addNode("L7", "C3", 7.0, 7.0);
  graph.addNode("L8", "C3", 8.0, 8.0);
  graph.addNode("L9", "C3", 9.0, 9.0);
  graph.addNode("L10", "C4", 10.0, 10.0);
  graph.addNode("L11", "C5", 11.0, 11.0);
  graph.addNode("L12", "C6", 12.0, 12.0);
  graph.addNode("L13", "C6", 13.0, 13.0);
  graph.addNode("L14", "C6", 14.0, 14.0);
  graph.addNode("L15", "C7", 15.0, 15.0);
  graph.addNode("L16", "C7", 16.0, 16.0);

  graph.addEdge("L1", "L2", new EdgeType(<CAR>, 10, 123, 23.0));
  graph.addEdge("L2", "L1", new EdgeType(<CAR>, 10, 234, 12.0));
  graph.addEdge("L2", "L5", new EdgeType(<CAR>, 20, 134, 55.0));
  graph.addEdge("L5", "L2", new EdgeType(<CAR>, 20, 215, 123.0));
  graph.addEdge("L2", "L4", new EdgeType(<CAR>, 15, 234, 67.0));
  graph.addEdge("L4", "L2", new EdgeType(<CAR>, 15, 642, 88.0));
```

```
graph.addEdge("L2", "L3", new EdgeType(<CAR>, 14, 2134, 234.0));
graph.addEdge("L3", "L2", new EdgeType(<CAR>, 13, 1531, 1003.0));
graph.addEdge("L5", "L6", new EdgeType(<CAR>, 60, 451, 114.0));
graph.addEdge("L6", "L7", new EdgeType(<CAR>, 55, 462, 665.0));
graph.addEdge("L6", "L8", new EdgeType(<CAR>, 22, 134, 54.0));
graph.addEdge("L8", "L6", new EdgeType(<CAR>, 51, 231, 245.5));
graph.addEdge("L7", "L3", new EdgeType(<CAR>, 100, 141, 123.0));
graph.addEdge("L5", "L9", new EdgeType(<CAR>, 4, 512, 134.0));
graph.addEdge("L9", "L5", new EdgeType(<CAR>, 90, 141, 131.0));
graph.addEdge("L9", "L10", new EdgeType(<CAR>, 30, 352, 7000.0));
graph.addEdge("L10", "L9", new EdgeType(<CAR>, 120, 42, 64.0));
graph.addEdge("L10", "L12", new EdgeType(<CAR>, 62, 875, 5000.0));
graph.addEdge("L12", "L10", new EdgeType(<CAR>, 78, 232, 56.0));
graph.addEdge("L12", "L11", new EdgeType(<CAR>, 788, 162, 6888));
graph.addEdge("L11", "L12", new EdgeType(<CAR>, 700, 124, 6444));
graph.addEdge("L11", "L9", new EdgeType(<CAR>, 33, 456, 95.0));
graph.addEdge("L9", "L11", new EdgeType(<CAR>, 33, 567, 74.0));
graph.addEdge("L11", "L13", new EdgeType(<CAR>, 9, 891, 41.0));
graph.addEdge("L13", "L11", new EdgeType(<CAR>, 10, 921, 92.0));
graph.addEdge("L13", "L16", new EdgeType(<CAR>, 87, 120, 22.0));
graph.addEdge("L16", "L13", new EdgeType(<CAR>, 86, 519, 15.0));
graph.addEdge("L16", "L15", new EdgeType(<CAR>, 50, 140, 12.0));
graph.addEdge("L15", "L16", new EdgeType(<CAR>, 55, 501, 93.0));
graph.addEdge("L15", "L13", new EdgeType(<CAR>, 23, 124, 54.0));
graph.addEdge("L13", "L15", new EdgeType(<CAR>, 22, 901, 246.0));
graph.addEdge("L13", "L14", new EdgeType(<CAR>, 17, 120, 234.0));
graph.addEdge("L14", "L13", new EdgeType(<CAR>, 15, 190, 421.0));

graph.addEdgeType("L1", "L2", new EdgeType(<PLANE>, 15, 190, 421.0));
graph.addEdgeType("L2", "L5", new EdgeType(<PLANE>, 15, 190, 421.0));
graph.addEdgeType("L5", "L9", new EdgeType(<PLANE>, 15, 190, 421.0));
graph.addEdgeType("L9", "L10", new EdgeType(<PLANE>, 12, 190, 421.0));
graph.addEdgeType("L10", "L12", new EdgeType(<PLANE>, 13, 190, 421.0));
graph.addEdgeType("L12", "L11", new EdgeType(<PLANE>, 14, 190, 421.0));
graph.addEdgeType("L9", "L11", new EdgeType(<PLANE>, 78, 190, 56.0));
graph.addEdgeType("L11", "L13", new EdgeType(<PLANE>, 15, 190, 421.0));

graph.addEdgeType("L1", "L2", new EdgeType(<TRAIN>, 1, 851, 45.0));
graph.addEdgeType("L2", "L5", new EdgeType(<TRAIN>, 2, 2614, 76.0));
graph.addEdgeType("L5", "L9", new EdgeType(<TRAIN>, 2, 1901, 82.0));
graph.addEdgeType("L9", "L10", new EdgeType(<TRAIN>, 5, 1901, 12.0));
graph.addEdgeType("L10", "L12", new EdgeType(<TRAIN>, 1, 1234, 56.0));
graph.addEdgeType("L12", "L11", new EdgeType(<TRAIN>, 8, 112, 64.0));
graph.addEdgeType("L9", "L11", new EdgeType(<TRAIN>, 8, 5413, 12.0));
graph.addEdgeType("L11", "L13", new EdgeType(<TRAIN>, 5, 4312, 23.0));

graph.addEdgeType("L1", "L2", new EdgeType(<FERRY>, 12, 81, 405.0));
graph.addEdgeType("L2", "L5", new EdgeType(<FERRY>, 29, 24, 706.0));
graph.addEdgeType("L5", "L9", new EdgeType(<FERRY>, 25, 11, 802.0));
graph.addEdgeType("L9", "L10", new EdgeType(<FERRY>, 51, 101, 102.0));
graph.addEdgeType("L10", "L12", new EdgeType(<FERRY>, 15, 14, 560.0));
graph.addEdgeType("L12", "L11", new EdgeType(<FERRY>, 83, 12, 640.0));
graph.addEdgeType("L9", "L11", new EdgeType(<FERRY>, 81, 53, 102.0));
graph.addEdgeType("L11", "L13", new EdgeType(<FERRY>, 56, 42, 203.0));

graph.addEdgeType("L1", "L2", new EdgeType(<BUS>, 14, 51, 4.0));
graph.addEdgeType("L2", "L5", new EdgeType(<BUS>, 12, 264, 7.0));
```

```
  graph.addEdgeType("L9", "L10", new EdgeType(<BUS>, 51, 191, 1.0));
  graph.addEdgeType("L10", "L12", new EdgeType(<BUS>, 166, 134, 5.0));
  graph.addEdgeType("L12", "L11", new EdgeType(<BUS>, 81, 12, 6.0));
  graph.addEdgeType("L9", "L11", new EdgeType(<BUS>, 87, 543, 2.0));
  graph.addEdgeType("L11", "L13", new EdgeType(<BUS>, 511, 4312, 3.0));

  return graph;
);

-- Tests the path building and validation
public testPath : () ==> ()
testPath() ==
(
  dcl path: Path := new Path(<CAR>, <PRICE>, 500);
  dcl n1: Node := new Node(18, 18, "X1", "C1");
  dcl n2: Node := new Node(45, 45, "X2", "C2");
  dcl n3: Node := new Node(55, 55, "X3", "C2");
  n2.path := n1;
  n3.path := n2;
  assertEqual(false, path.isPossible());
  path.addNode(n1);
  path.addNode(n2);
  path.addNode(n3);
  assertEqual(path.path(1).location, n3.location);
  assertEqual(path.path(2).location, n2.location);
  assertEqual(path.path(3).location, n1.location);
  assertEqual(true, path.isPossible());
);

-- Tests Dijkstra's algorithm for car only trips using all criteria
public testDijkstraCar: () ==> ()
testDijkstraCar() ==
(
  dcl graph: Graph := makeGraph();
  dcl path: Path;

  path := graph.findPath("L1", "L15", <TIME>, <CAR>);
  assertEqual(98, path.cost);
  assertEqual(7, len path.path);
  assertEqual("L1", path.path(1).location);
  assertEqual("L2", path.path(2).location);
  assertEqual("L5", path.path(3).location);
  assertEqual("L9", path.path(4).location);
  assertEqual("L11", path.path(5).location);
  assertEqual("L13", path.path(6).location);
  assertEqual("L15", path.path(7).location);

  path := graph.findPath("L9", "L3", <TIME>, <CAR>);
  assertEqual(124, path.cost);
  assertEqual(4, len path.path);
  assertEqual("L9", path.path(1).location);
  assertEqual("L5", path.path(2).location);
  assertEqual("L2", path.path(3).location);
  assertEqual("L3", path.path(4).location);

  path := graph.findPath("L1", "L1", <TIME>, <CAR>);
  assertEqual(0, path.cost);
  assertEqual(1, len path.path);
```

```
  assertEqual("L1", path.path(1).location);

  path := graph.findPath("L1", "L15", <PRICE>, <CAR>);
  assertEqual(361, path.cost);
  assertEqual(8, len path.path);
  assertEqual("L1", path.path(1).location);
  assertEqual("L2", path.path(2).location);
  assertEqual("L5", path.path(3).location);
  assertEqual("L9", path.path(4).location);
  assertEqual("L11", path.path(5).location);
  assertEqual("L13", path.path(6).location);
  assertEqual("L16", path.path(7).location);
  assertEqual("L15", path.path(8).location);

  path := graph.findPath("L15", "L7", <DISTANCE>, <CAR>);
  assertEqual(2497, path.cost);
  assertEqual(9, len path.path);
  assertEqual("L15", path.path(1).location);
  assertEqual("L13", path.path(2).location);
  assertEqual("L11", path.path(3).location);
  assertEqual("L12", path.path(4).location);
  assertEqual("L10", path.path(5).location);
  assertEqual("L9", path.path(6).location);
  assertEqual("L5", path.path(7).location);
  assertEqual("L6", path.path(8).location);
  assertEqual("L7", path.path(9).location);
);

-- Tests Dijkstra's algorithm for plane only trips using all criteria
public testDijkstraPlane: () ==> ()
testDijkstraPlane() ==
(
  dcl graph: Graph := makeGraph();
  dcl path: Path;

  path := graph.findPath("L1", "L13", <TIME>, <PLANE>);
  assertEqual(99, path.cost);
  assertEqual(8, len path.path);
  assertEqual("L1", path.path(1).location);
  assertEqual("L2", path.path(2).location);
  assertEqual("L5", path.path(3).location);
  assertEqual("L9", path.path(4).location);
  assertEqual("L10", path.path(5).location);
  assertEqual("L12", path.path(6).location);
  assertEqual("L11", path.path(7).location);
  assertEqual("L13", path.path(8).location);

  path := graph.findPath("L1", "L15", <TIME>, <PLANE>);
  assertEqual(-1, path.cost);

  path := graph.findPath("L1", "L13", <PRICE>, <PLANE>);
  assertEqual(1740, path.cost);
  assertEqual(6, len path.path);
  assertEqual("L1", path.path(1).location);
  assertEqual("L2", path.path(2).location);
  assertEqual("L5", path.path(3).location);
  assertEqual("L9", path.path(4).location);
  assertEqual("L11", path.path(5).location);
```

```
  assertEqual("L13", path.path(6).location);

  path := graph.findPath("L5", "L13", <DISTANCE>, <PLANE>);
  assertEqual(570, path.cost);
  assertEqual(4, len path.path);
  assertEqual("L5", path.path(1).location);
  assertEqual("L9", path.path(2).location);
  assertEqual("L11", path.path(3).location);
  assertEqual("L13", path.path(4).location);
);

-- Tests Dijkstra's algorithm for other travel types trips using the time criterion
public testDijkstraOthers: () ==> ()
testDijkstraOthers() ==
(
  dcl graph: Graph := makeGraph();
  dcl path: Path;

  path := graph.findPath("L1", "L13", <TIME>, <ANY>);
  assertEqual(true, path.isPossible());
  assertEqual(18, path.cost);
  assertEqual(6, len path.path);
  assertEqual("L1", path.path(1).location);
  assertEqual("L2", path.path(2).location);
  assertEqual("L5", path.path(3).location);
  assertEqual("L9", path.path(4).location);
  assertEqual("L11", path.path(5).location);
  assertEqual("L13", path.path(6).location);

  path := graph.findPath("L1", "L13", <TIME>, <FERRY>);
  assertEqual(true, path.isPossible());
  assertEqual(203, path.cost);
  assertEqual(6, len path.path);
  assertEqual("L1", path.path(1).location);
  assertEqual("L2", path.path(2).location);
  assertEqual("L5", path.path(3).location);
  assertEqual("L9", path.path(4).location);
  assertEqual("L11", path.path(5).location);
  assertEqual("L13", path.path(6).location);

  path := graph.findPath("L1", "L13", <TIME>, <TRAIN>);
  assertEqual(true, path.isPossible());
  assertEqual(18, path.cost);
  assertEqual(6, len path.path);

  path := graph.findPath("L1", "L13", <TIME>, <BUS>);
  assertEqual(false, path.isPossible());
);

/** USE CASE SCENARIOS**/
/** Only pre-conditions relating to authentication are stated here,
as all others are already built into the model itself **/

-- Tests use case UC1
public testAdminLogin: () ==> ()
testAdminLogin() ==
(
  r2r := new Rome2Rio();
```

```
  assertEqual(false, r2r.changeToAdmin("foo"));
  assertEqual(true, r2r.changeToAdmin("default"));
  assertEqual(r2r.status, <Administrator>);
);

-- Tests use cases UC2 and UC7
public testAdminPasswordChange: () ==> ()
testAdminPasswordChange() ==
(
  r2r.setAdministratorPassword("new_password");
  assertEqual(r2r.adminCode, "new_password");
  r2r.changeToClient();
  assertEqual(r2r.status, <Client>);
  assertEqual(false, r2r.changeToAdmin("default"));
  assertEqual(true, r2r.changeToAdmin("new_password"));
  assertEqual(r2r.status, <Administrator>);
);

-- Tests use case UC3
public testAddLocations : () ==> ()
testAddLocations() ==
(
  r2r.addLocation("L1", "C1", 1.0, 1.0);
  assertEqual(1, card r2r.graph.node);
  r2r.addLocation("L2", "C1", 2.0, 2.0);
  assertEqual(2, card r2r.graph.node);
  r2r.addLocation("L3", "C1", 3.0, 3.0);
  assertEqual(3, card r2r.graph.node);
  r2r.addLocation("L4", "C1", 4.0, 3.0);
  assertEqual(4, card r2r.graph.node);
);

-- Tests use case UC4
public testAddConnections: () ==> ()
testAddConnections() ==
(
  r2r.addWayBetweenLocations("L1", "L2", <CAR>, 1, 1, 1.0);
  assertEqual(1, card r2r.graph.edge);
  r2r.addWayBetweenLocations("L1", "L3", <CAR>, 2, 2, 2.0);
  assertEqual(2, card r2r.graph.edge);
  r2r.addWayBetweenLocations("L2", "L3", <CAR>, 3, 3, 3.0);
  assertEqual(3, card r2r.graph.edge);
);

-- Tests use case UC5
public testAddTravelTypes: () ==> ()
testAddTravelTypes() ==
(
  r2r.addNewTransportationType("L1", "L2", <PLANE>, 50, 50, 50.5);
  assertEqual(2, card r2r.graph.findEdge("L1", "L2").edgeType);
  r2r.addNewTransportationType("L1", "L2", <FERRY>, 30, 30, 30.5);
  assertEqual(3, card r2r.graph.findEdge("L1", "L2").edgeType);
  r2r.addNewTransportationType("L2", "L3", <PLANE>, 50, 50, 50.5);
  assertEqual(2, card r2r.graph.findEdge("L2", "L3").edgeType);
);

-- Tests use case UC6
public testRemoveLocations: () ==> ()
```

```
testRemoveLocations() ==
(
  dcl edgeCount: nat := card r2r.graph.edge;
  dcl nodeCount: nat := card r2r.graph.node;
  r2r.removeLocation("L1");
  assertEqual(edgeCount - 2, card r2r.graph.edge);
  assertEqual(nodeCount - 1, card r2r.graph.node);
  r2r.graph := makeGraph();
);

-- Tests use case UC8
public testListLocations: () ==> ()
testListLocations() ==
  assertEqual(card r2r.graph.node, r2r.listLocations());

-- Tests use case UC9
public testGetRouteWithCriteria: () ==> ()
testGetRouteWithCriteria() ==
(
  dcl path: Path;

  -- since we need a few more nodes and edges to make interesting stuff,
  -- we'll use the graph from the unit tests
  r2r.graph := makeGraph();

  -- since we already proved that Dijkstra's algorithm worked with
  -- different criteria using the unit tests, we'll just test an
  -- instance with a possible route and another one with an impossible one
  path := r2r.getRouteWithCriteria("L1", "L15", <CAR>, <TIME>);
  assertEqual(true, path.isPossible());
  assertEqual(98, path.cost);
  assertEqual(7, len path.path);
  assertEqual("L1", path.path(1).location);
  assertEqual("L2", path.path(2).location);
  assertEqual("L5", path.path(3).location);
  assertEqual("L9", path.path(4).location);
  assertEqual("L11", path.path(5).location);
  assertEqual("L13", path.path(6).location);
  assertEqual("L15", path.path(7).location);

  path := r2r.getRouteWithCriteria("L1", "L15", <PLANE>, <TIME>);
  assertEqual(false, path.isPossible());
  assertEqual(-1, path.cost);
);

-- Tests use case UC10
public testGetAllRoutes : () ==> ()
testGetAllRoutes() ==
(
  dcl paths: set of (Path) := {};
  r2r := new Rome2Rio();
  r2r.graph := makeGraph();
  paths := r2r.getBestRoutesForAllCriteria("L1", "L13");
  assertEqual(card paths, 15);

  for all p in set paths do
    assertEqual(true, p.isPossible());
);
```

```
/** UNIT TEST FOR ERRORS (FAILING INVARIANTS, PRE AND POST CONDITIONS)**/

-- uncomment each line and run to see the respective error
-- (only one at a time)
public testErrors: () ==> ()
testErrors() ==
(
  dcl graph: Graph := new Graph();
  dcl node: Node;
  dcl edge: Edge;
  dcl et: EdgeType;
  dcl pq: PriorityQueue := new PriorityQueue();
  dcl r2r: Rome2Rio := new Rome2Rio();
  graph.addNode("L1", "C1", 1, 1);

  -- out-of-bounds coordinate (pre-condition)
  --graph.addNode("L2", "C2", 200, 1);
  --graph.addNode("L2", "C2", -200, 1);
  --graph.addNode("L2", "C2", 1, 200);
  --graph.addNode("L2", "C2", 1, -200);

  -- duplicate location (invariant)
  -- graph.addNode("L1", "C1", 5, 5);

  -- duplicate coordinate (invariant)
  --graph.addNode("L2", "C1", 1, 1);

  graph.addNode("L2", "C1", 5, 5);
  graph.addEdge("L1", "L2", new EdgeType(<CAR>, 1, 1, 1));

  -- duplicate edge (pre-condition)
  --graph.addEdge("L1", "L2", new EdgeType(<TRAIN>, 1, 1, 1));

  -- duplicate edge type (invariant)
  --graph.addEdgeType("L1", "L2", new EdgeType(<CAR>, 1, 1, 1));

  -- find non-existing node (pre-condition)
  --node := graph.findNode("L55");

  -- find non-existing edge (pre-condition)
  --edge := graph.findEdge("L1", "L55");
  --edge := graph.findEdge("L55", "L1");
  --edge := graph.findEdge("L55", "L88");

  node := new Node(1, 1, "A", "B");
  pq.addElement(node);
  -- duplicate node in priority queue (pre-condition)
  --pq.addElement(node);

  -- token <ANY> on EdgeType (pre-condition)
  --et := new EdgeType(<ANY>, 1, 1, 1);

  -- remove non-existing node (pre-condition)
  --graph.removeNode("L13");

  -- admin operations without being an admin (pre-condition)
  --r2r.setAdministratorPassword("foo");
```

```
    --r2r.addLocation("foo", "bar", 1, 1);
    --r2r.addWayBetweenLocations("foo", "bar", <CAR>, 1, 1, 1);
    --r2r.addNewTransportationType("foo", "bar", <BUS>, 1, 1, 1);
    --r2r.removeLocation("foo");
    --r2r.changeToClient();
  );

  /** ENTRY POINT FOR THE TEST SUITE**/

  public testAll: () ==> ()
  testAll() ==
  (
    /*UNIT TESTS*/
    testAddNodesAndEdges();
    testPriorityQueue();
    testPath();
    testDijkstraCar();
    testDijkstraPlane();
    testDijkstraOthers();

    /*USE CASE SCENARIOS*/
    testAdminLogin();
    testAdminPasswordChange();
    testAddLocations();
    testAddConnections();
    testAddTravelTypes();
    testRemoveLocations();
    testListLocations();
    testGetRouteWithCriteria();
    testGetAllRoutes();

    /*ERROR CHECKING*/
    testErrors();
  );

end TestRome2Rio
```

5. Model verification

5.1 Example of domain verification

One of the proof obligations generated by Overture is:

No.	PO Name	Type
32	TestRome2Rio`testDijkstraCar()	legal sequence application

The Proof Obligation generated by Overture is the following:

```
(2 in set (inds ((graph.findPath)("L1", "L15", <TIME>, <CAR>).path)))
```

The code under analysis is:

```
public testDijkstraCar: () ==> ()
 testDijkstraCar() ==
 (
   dcl graph: Graph := makeGraph();
   dcl path: Path;

   path := graph.findPath("L1", "L15", <TIME>, <CAR>);
   assertEqual(98, path.cost);
   assertEqual(7, len path.path);
   assertEqual("L1", path.path(1).location);
   assertEqual("L2", path.path(2).location);
   assertEqual("L5", path.path(3).location);
```

As we can see, the proof demands there to be an index 2 in the path.path sequence in order for the assert to be executed. Previously, we have seen and successfully asserted that the length of the path.path sequence was 7 (which implies that the indexes go from 1 to 7, assuredly), and thus we will not incur in a domain violation on the sequence's indexes.

5.2 Example of invariant verification

Another proof obligation generated by Overture is:

No.	PO Name	Type
6	Graph`findNode(Node`Location)	state invariant holds

The Proof Obligation generated by Overture is the following:

```
(forall location:Node`Location & ((exists1 n in set node & ((n.location) = location)) =>
(((not (exists n1, n2 in set node & ((n1 <> n2) and ((n1.location) = (n2.location)))))
and (not (exists n1, n2 in set node & ((n1 <> n2) and (((n1.coordinateNS) =
(n2.coordinateNS)) and ((n1.coordinateEW) = (n2.coordinateEW))))))) => ((not (exists n1,
n2 in set node & ((n1 <> n2) and ((n1.location) = (n2.location))))) and (not (exists n1,
```

```
n2 in set node & ((n1 <> n2) and (((n1.coordinateNS) = (n2.coordinateNS)) and
((n1.coordinateEW) = (n2.coordinateEW))))))))))
```

The code under analysis is:

```
pure public findNode : Node`Location ==> Node
findNode(location) ==
(
    dcl res: Node;
    for all n in set node do if n.location = location then res := n;
    return res;
)
pre exists1 n in set node & n.location = location;
```

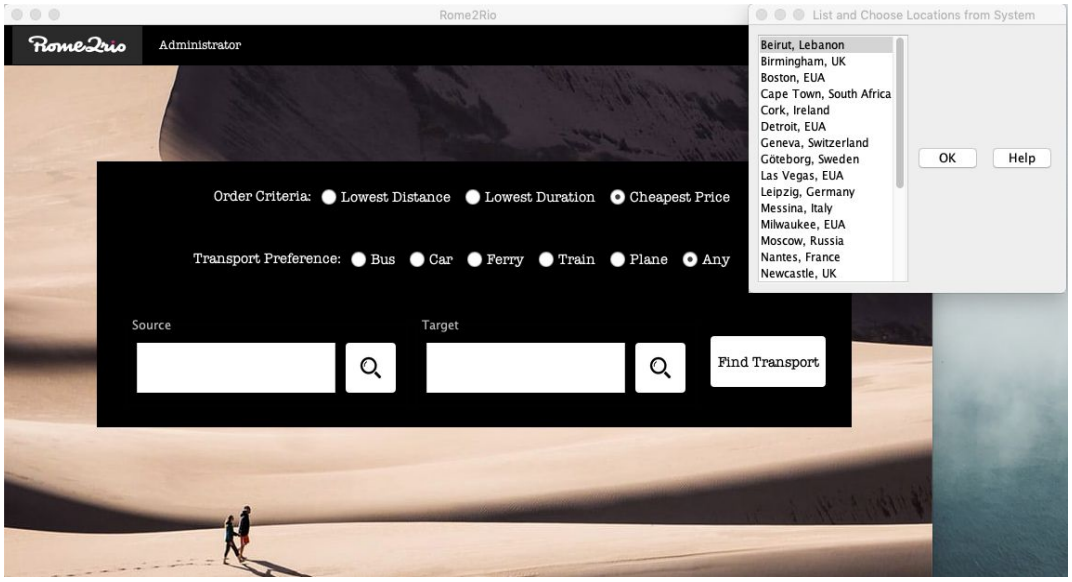
As we can see, the proof is trivial: the invariant states that there can never be two nodes with the same location or coordinates. This operation, being pure, does not change the state of the model, and thus naturally preserves the invariant.

6. Code Generation

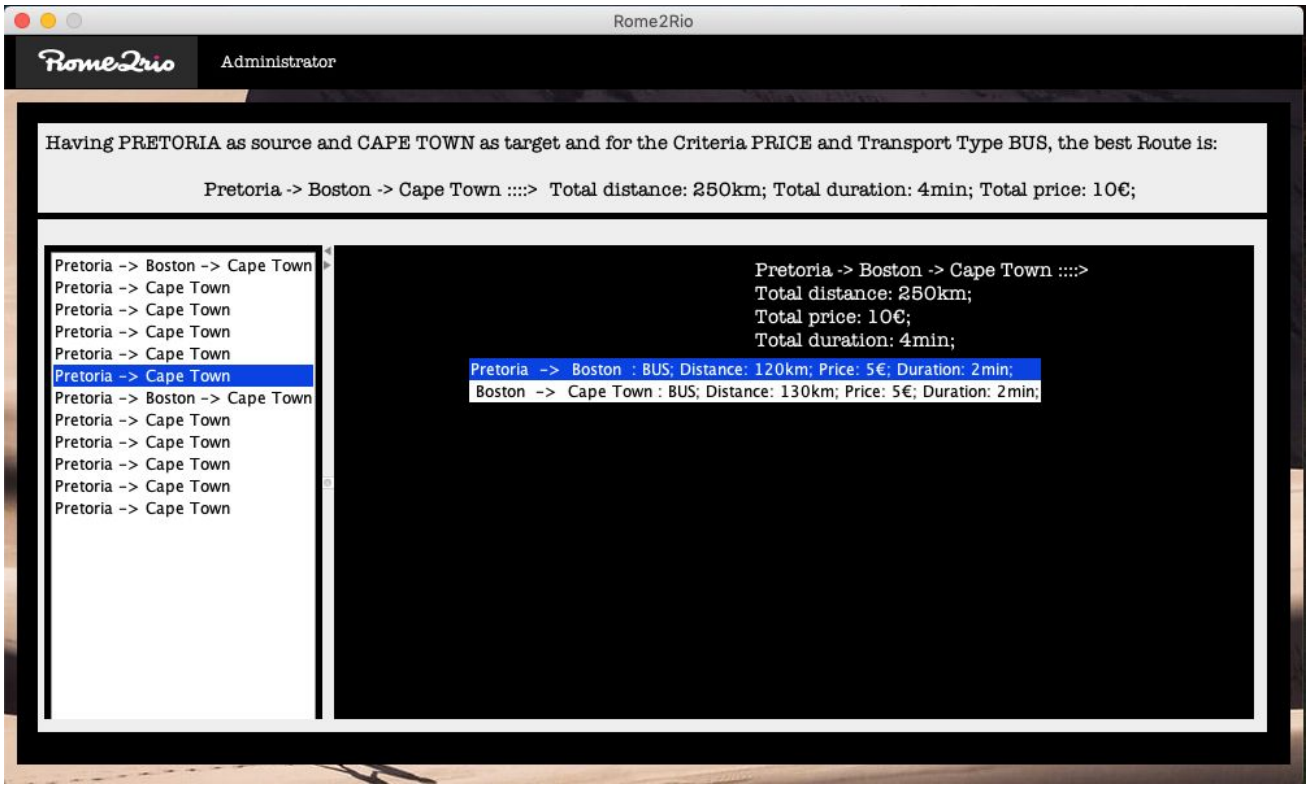
We generated the Java code from the VDM++ model for the system, as essentially the logic of it and installed in a rome2rio package inside of our Java project, and then we added an interface in java swing that “talks” with it independently and was installed in another package, the gui.

It is possible to run the generated code with the interface running the project in eclipse with Main class in the rome2rio package or with class MainMenu on the gui package. The tests of the generated code can be run from the Main class in the rome2rio package using as argument “-test”.

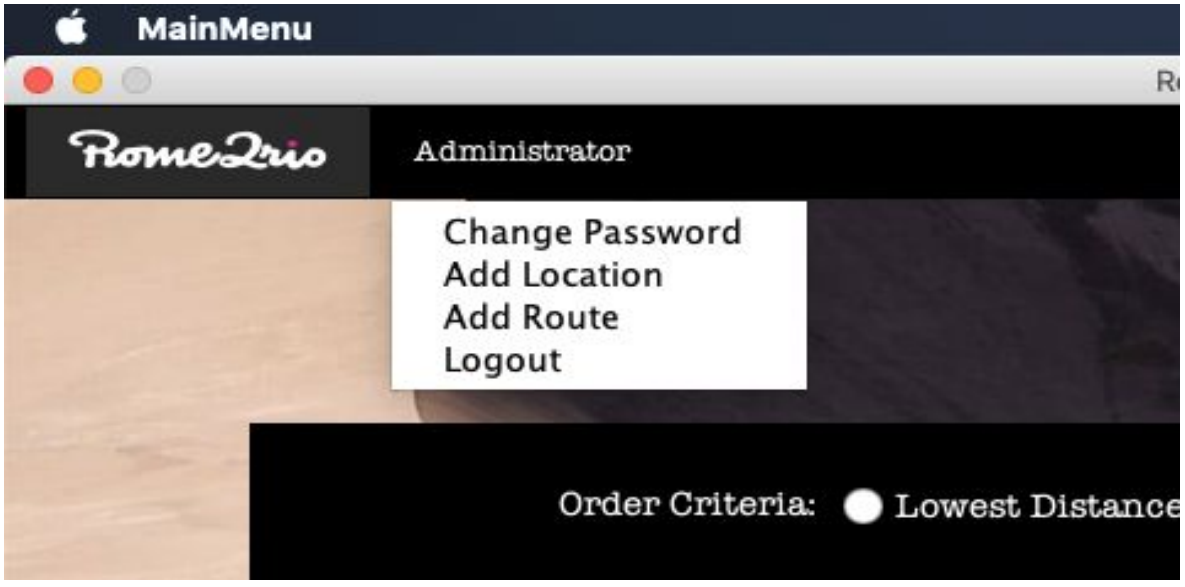
Some of the requirements mentioned above can be seen below:



1. Main Menu and Listing of Existent Locations



2. Detailed information of the existent Routes and Best Route for Criteria and Transport Choosen.



3. Administrator Options that open another frame as same as Listing Locations

7. Conclusions

The model developed covers all use case scenarios and requirements. The code coverage is 100%, and we believe that we have made a very solid model.

While the group intended to add even more features, such as dates and scheduling of trips, we decided against it since our model was already reaching the 10 pages maximum. Still, there is plenty of room for more functionalities, similar or not to the ones present on the real Rome2Rio Web Application, which could be left as future work.

The group also faced some difficulty over inconsistent behaviour of the Overture tool, from different coverage results on different OS to a NullPointerException on the debugger itself, which hindered the development significantly while we couldn’t work around it. Also, the fact that no pre- and post-conditions are generated in Java made us lose more time validating inputs on the GUI.

Finally, a huge chunk of time was also spent devising data for testing, as writing unit and feature tests for a shortest path algorithm is not trivial, as we need to use a high enough number of nodes and weights in order to test all possible scenarios.

The workload between the two members was the same.

8. References

1. Slides provided by the teaching staff
2. Tutorial for Overture/VDM++, Peter Gorm Larsen et al, Overture Technical Report Series No. TR-004, September 2015
3. Rome2rio, which this project tried to model: https://www.rome2rio.com/