

Formal Modeling of the ‘Via Verde’ system in VDM++

Diogo Vaz Nunes

Gabriel Cardoso Candal

Miguel Jorge Gonçalves Pereira

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

17/12/2014

Contents

[1. Informal system description and list of requirements 3](#_Toc406519310)

[1.1 Informal system description 3](#_Toc406519311)

[1.2 List of requirements 3](#_Toc406519312)

[2. Visual UML model 4](#_Toc406519313)

[2.1 Use case model 4](#_Toc406519314)

[2.3 Class model 4](#_Toc406519315)

[3. Formal VDM++ model 5](#_Toc406519316)

[3.1 Class ViaVerde 5](#_Toc406519317)

[4. Model validation 5](#_Toc406519318)

[4.1 Class MyTestCase 5](#_Toc406519319)

[4.2 Class TestViaVerde 6](#_Toc406519320)

[5. Model verification 6](#_Toc406519321)

[5.1 Example of domain verification 6](#_Toc406519322)

[5.2 Example of invariant verification 6](#_Toc406519323)

[6. Conclusions 6](#_Toc406519324)

[7. References 6](#_Toc406519325)

# 1. Informal system description and list of requirements

## 1.1 Informal system description

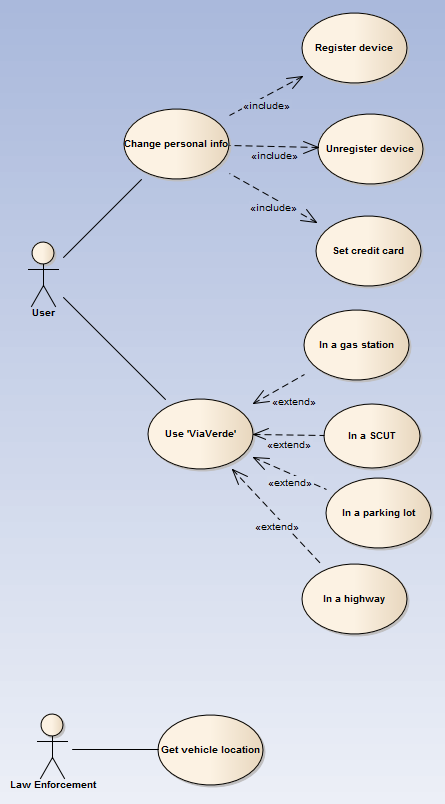
Via Verde facilitates the usage and payment of certain services through the attachment of a device to the front of a vehicle. Those services include highways, parking lots, gas stations and SCUT. For those vehicles not identified by a device, the system will record its usage anyway, so that the user can associate a device to that vehicle when he wishes.

## 1.2 List of requirements

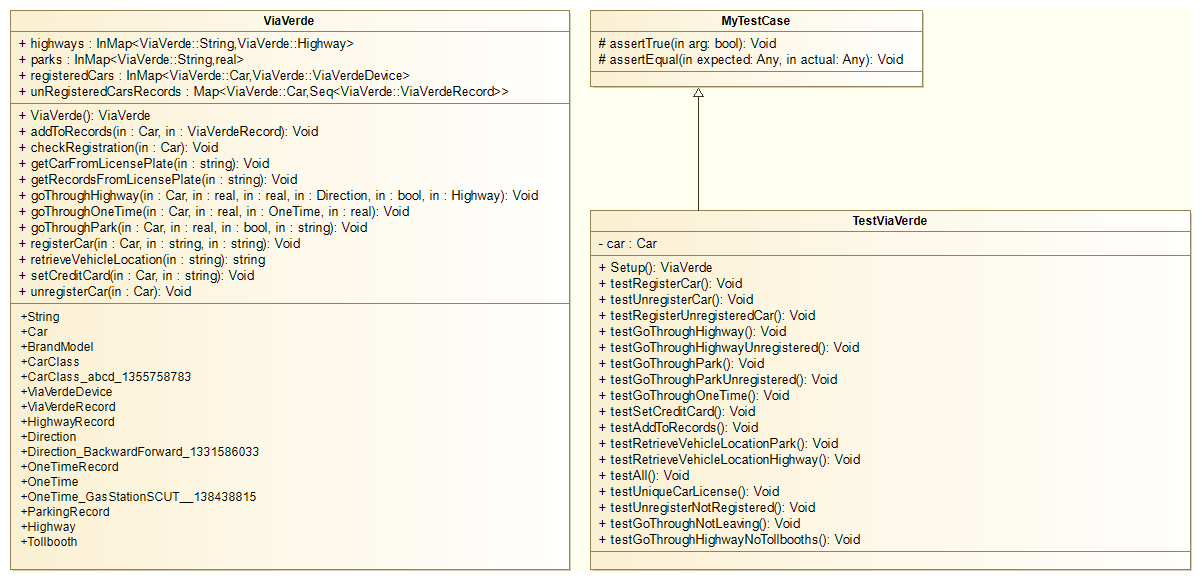
|  |  |  |
| --- | --- | --- |
| **Id** | **Priority** | **Description** |
| R1 | Mandatory | Clients may register a Via Verde device for their vehicles. |
| R2 | Mandatory | Clients may unregister a Via Verde device from their previously registered vehicles. |
| R3 | Mandatory | Clients may change the credit card number associated to one of his Via Verde devices. |
| R4 | Mandatory | Clients may use the services provided by Via Verde, having a device or not. |
| R5 | Mandatory | Clients must be able to check the records for a given car. |
| R6 | Mandatory | The police must be able to retrieve a vehicle’s current location, if that vehicle is still in a park or in a highway. |

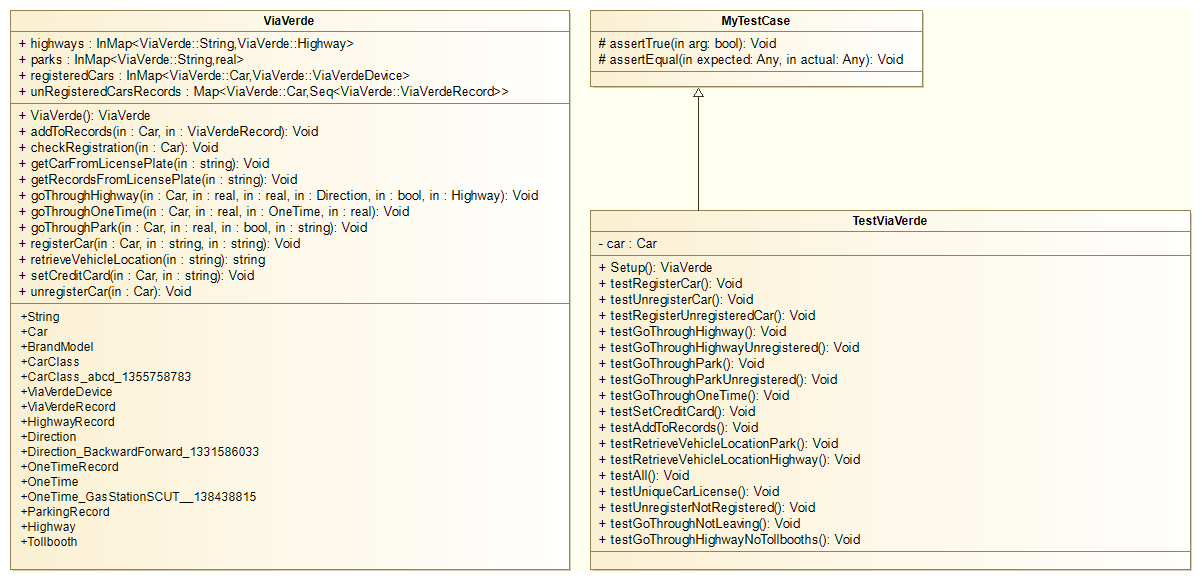
# 2. Visual UML model

## 2.1 Use case model



## 2.3 Class model





# 3. Formal VDM++ model

## 3.1 Class ViaVerde

**class** ViaVerde

**types**

**public** String = **seq1** **of** **char**;

**public** Car::licensePlate:String

brandModel:BrandModel;

**public** CarClass = **<a>** | **<b>** | **<c>** | **<d>**;

-- Brand \* Model \* Class

**public** BrandModel = **<Mercedes>** \* **<Benz>** \* **<a>** | **<Audi>** \* **<A4>** \* **<a>** | **<KIA>** \* **<Leo>** \* **<a>** | **<Hyundai>** \* **<Coupe>** \* **<a>**

| **<BMW>** \* **<X3>** \* **<a>** | **<Mitsubishi>** \* **<Lancer>** \* **<a>**;

**public** ViaVerdeDevice::id:String

creditCard:String

records:**seq** **of** ViaVerdeRecord;

**public** ViaVerdeRecord::record:OneTimeRecord | ParkingRecord | HighwayRecord

cost: **real**;

**public** HighwayRecord::entryKm: **real**

exitKm: **real**

entryTime: **real**

exitTime: **real**

direction: Direction

highway: String;

**public** ParkingRecord::entryTime: **real**

exitTime: **real**

park: String;

**public** OneTimeRecord::oneTime: OneTime

time: **real**;

**public** OneTime = **<GasStation>** | **<SCUT>**;

**public** Highway::costPerClass: **inmap** CarClass **to** **real**

tollbooths: **set** **of** Tollbooth

name: String;

**public** Direction = **<Forward>** | **<Backward>**;

**public** Tollbooth::km: **real**

direction: Direction;

**instance variables**

**public** highways: **inmap** String **to** Highway := { |-> };

--public highways: inmap String to Highway := { "jo" |-> mk\_Highway({<a> |-> 10, <b> |-> 11, <c> |-> 12, <d> |-> 13}, {mk\_Tollbooth(0, <Forward>) |-> 10, mk\_Tollbooth(10, <Forward>) |-> 12}, "jo")};

--Park name -> Cost per hour

**public** parks: **inmap** String **to** **real** := { |-> };

--public parks: inmap String to real := { "jo2" |-> 10};

**public** registeredCars: **inmap** Car **to** ViaVerdeDevice := { |-> };

**public** unRegisteredCarsRecords: **map** Car **to** **seq** **of** ViaVerdeRecord := { |-> };

--All cost, distance and time units >= 0

--Entry time/KM < Exit time/KM

--From here

**inv** forall highway **in set** rng highways &

forall cpc **in set** rng highway.costPerClass &

cpc >= 0;

**inv** forall highway **in set** rng highways &

forall tollbooth **in set** highway.tollbooths &

tollbooth.km >= 0;

**inv** forall costPerHour **in set** rng parks & costPerHour >= 0;

**inv** forall vv\_records **in set** rng unRegisteredCarsRecords &

forall vv\_record **in set** elems vv\_records &

validRecordKm(vv\_record) and

consistentRecordKm(vv\_record, rng highways) and

consistentNames(vv\_record, rng highways, dom parks);

**inv** forall device **in set** rng registeredCars &

forall vv\_record **in set** elems device.records &

validRecordKm(vv\_record) and

consistentRecordKm(vv\_record, rng highways) and

consistentNames(vv\_record, rng highways, dom parks);

--Until here

--A car is either registered or unregistered

**inv** dom registeredCars inter dom unRegisteredCarsRecords = {};

--Unique license plate in the whole system

**inv** not exists a, b **in set** dom unRegisteredCarsRecords union dom registeredCars &

a.licensePlate = b.licensePlate and

(a.brandModel <> b.brandModel or

(a in **set** dom registeredCars and b in **set** dom unRegisteredCarsRecords) or

(a in **set** dom unRegisteredCarsRecords and b in **set** dom registeredCars));

--An highway must have prices set for every class

**inv** not exists h **in set** rng highways & dom h.costPerClass inter {<a>, <b>, <c>, <d>} <> {<a>, <b>, <c>, <d>};

--A car can't be in more then a place at a time (being it a park, an highway, or a mix)

**inv** forall vv\_records **in set** rng unRegisteredCarsRecords &

noTwoRecordsWithExitZero(vv\_records, false);

**inv** forall device **in set** rng registeredCars &

noTwoRecordsWithExitZero(device.records, false);

**operations**

**public** ViaVerde: () ==> ViaVerde

ViaVerde() ==

return self;

**public** registerCar(car: Car, creditCard: String, id: String) == (

if(car in **set** dom unRegisteredCarsRecords) **then** (

atomic(

registeredCars := registeredCars munion { car |-> mk\_ViaVerdeDevice(id, creditCard, unRegisteredCarsRecords(car))};

unRegisteredCarsRecords := {car} <-: unRegisteredCarsRecords;)

) **else**

registeredCars := registeredCars munion { car |-> mk\_ViaVerdeDevice(id, creditCard, [])};

);

**public** unregisterCar(car: Car) == (

atomic(

unRegisteredCarsRecords := unRegisteredCarsRecords munion { car |-> registeredCars(car).records};

registeredCars := {car} <-: registeredCars;)

) **pre** car in **set** dom registeredCars;

**public** goThroughHighway(car: Car, km: **real**, time: **real**, direction: Direction, entrance:**bool**, highway: Highway) == (

**dcl** cost: **real** := 0;

checkRegistration(car);

if(not entrance) **then** (

if(car in **set** dom registeredCars) **then** (

**dcl** kmCost: **real** := (abs (registeredCars(car).records(len registeredCars(car).records).record.entryKm-km));

cost := kmCost \* highway.costPerClass(car.brandModel.#3);

registeredCars(car).records(len registeredCars(car).records).cost := cost;

registeredCars(car).records(len registeredCars(car).records).record.exitKm := km;

registeredCars(car).records(len registeredCars(car).records).record.exitTime := time

) **else** (

**dcl** kmCost: **real** := (abs (unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).record.entryKm-km));

cost := kmCost \* highway.costPerClass(car.brandModel.#3);

unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).cost := cost;

unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).record.exitKm := km;

unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).record.exitTime := time

))

**else**

addToRecords(car, mk\_ViaVerdeRecord(mk\_HighwayRecord(km, 0, time, 0, direction, highway.name), cost))

)

**pre** highway.name in **set** dom highways and

if(not entrance) **then** (

if(car in **set** dom registeredCars) **then** (

registeredCars(car).records(len registeredCars(car).records).record.exitKm = 0

) **else** if(car in **set** dom unRegisteredCarsRecords) **then** (

unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).record.exitKm = 0

) **else** true

) **else** true

and if(entrance) **then** (

km <> 0 and

time <> 0

) **else** true;

**public** goThroughPark(car: Car, time: **real**, entrance: **bool**, park: String) == (

**dcl** cost: **real** := 0;

checkRegistration(car);

if(not entrance) **then** (

if(car in **set** dom registeredCars) **then** (

cost := (abs (registeredCars(car).records(len registeredCars(car).records).record.entryTime-time)) \*parks(park);

registeredCars(car).records(len registeredCars(car).records).cost := cost;

registeredCars(car).records(len registeredCars(car).records).record.exitTime := time

) **else** (

cost := (abs (unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).record.entryTime-time)) \*parks(park);

unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).cost := cost;

unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).record.exitTime := time;

))

**else**

addToRecords(car, mk\_ViaVerdeRecord(mk\_ParkingRecord(time, 0, park), cost));

)

**pre** park in **set** dom parks and

if(not entrance) **then** (

if(car in **set** dom registeredCars) **then** (

registeredCars(car).records(len registeredCars(car).records).record.exitTime = 0

) **else** if(car in **set** dom unRegisteredCarsRecords) **then** (

unRegisteredCarsRecords(car)(len unRegisteredCarsRecords(car)).record.exitTime = 0

) **else** true

) **else** true

and if(entrance) **then**

time <> 0

**else** true;

**public** goThroughOneTime(car: Car, cost: **real**, oneTime: OneTime, time: **real**) == (

checkRegistration(car);

addToRecords(car, mk\_ViaVerdeRecord(mk\_OneTimeRecord(oneTime, time), cost));

);

**public** setCreditCard(car: Car, creditCard: String) == (

registeredCars(car).creditCard := creditCard;

)

**pre** car in **set** dom registeredCars;

**public** addToRecords(car: Car, viaVerdeRecord: ViaVerdeRecord) == (

if(car in **set** dom registeredCars) **then**

registeredCars(car).records := registeredCars(car).records ^ [viaVerdeRecord]

**else**

unRegisteredCarsRecords(car) := unRegisteredCarsRecords(car) ^ [viaVerdeRecord];

);

**public** checkRegistration(car: Car) == (

if(car not **in set** (dom registeredCars) union (dom unRegisteredCarsRecords)) **then**

unRegisteredCarsRecords := unRegisteredCarsRecords munion { car |-> []};

);

**public** retrieveVehicleLocation(licensePlate: String) res: **<NotFound>** | String ==

let vv\_records = getRecordsFromLicensePlate(licensePlate) **in**

if exists vv\_record **in set** elems vv\_records & is\_ParkingRecord(vv\_record.record) and vv\_record.record.exitTime = 0

**then** return (iota vv\_record **in set** elems vv\_records & is\_ParkingRecord(vv\_record.record) and vv\_record.record.exitTime = 0).record.park

elseif exists vv\_record **in set** elems vv\_records & is\_HighwayRecord(vv\_record.record) and vv\_record.record.exitKm = 0

**then** return (iota vv\_record **in set** elems vv\_records & is\_HighwayRecord(vv\_record.record) and vv\_record.record.exitKm = 0).record.highway

**else** return <NotFound>

**pre** exists car **in set** dom unRegisteredCarsRecords union dom registeredCars &

car.licensePlate = licensePlate;

**public** getRecordsFromLicensePlate(licensePlate: String) res: **seq** **of** ViaVerdeRecord == (

**dcl** car: Car | **<NotFound>** := getCarFromLicensePlate(licensePlate);

**dcl** records: **seq** **of** ViaVerdeRecord;

if car = <NotFound> **then** return [];

if car in **set** dom registeredCars

**then** records := registeredCars(car).records

**else** records := unRegisteredCarsRecords(car);

return records;

)

**pre** exists car **in set** dom unRegisteredCarsRecords union dom registeredCars &

car.licensePlate = licensePlate;

**public** getCarFromLicensePlate(licensePlate: String) res: Car == (

return iota car **in set** dom unRegisteredCarsRecords union dom registeredCars &

car.licensePlate = licensePlate;

)

**pre** exists car **in set** dom unRegisteredCarsRecords union dom registeredCars &

car.licensePlate = licensePlate;

**functions**

**public** remove[@T](e: @T, s: **seq** **of** @T) res: **seq** **of** @T ==

[s(i) | i **in set** inds s & s(i) <> e];

**public** removeMap[@T](e: @T, s: **map** @T **to** **seq** **of** ViaVerdeRecord) res: **map** @T **to** **seq** **of** ViaVerdeRecord ==

{ i |-> s(i) | i **in set** dom s & s(i) <> e};

--Positive distances/times, entry before exit

**private** validRecordKm(vv\_record: ViaVerdeRecord) res: **bool** ==

is\_OneTimeRecord(vv\_record.record) => vv\_record.record.time >= 0 and

( is\_HighwayRecord(vv\_record.record) =>

vv\_record.record.entryKm >= 0 and vv\_record.record.exitKm >= 0 and

greaterOrZero(vv\_record.record.entryKm, vv\_record.record.exitKm) ) and

( is\_ParkingRecord(vv\_record.record) =>

vv\_record.record.entryTime >= 0 and vv\_record.record.exitTime >= 0 and

greaterOrZero(vv\_record.record.entryTime, vv\_record.record.exitTime) );

--Entry/exit km in records match existing tollbooths

**private** consistentRecordKm(vv\_record: ViaVerdeRecord, highways: **set** **of** Highway) res: **bool** ==

is\_HighwayRecord(vv\_record.record) => (

exists highway **in set** highways &

highway.name = vv\_record.record.highway and

( exists tollbooth **in set** highway.tollbooths &

vv\_record.record.entryKm = tollbooth.km and

vv\_record.record.direction = tollbooth.direction )

and vv\_record.record.exitKm > 0 =>

( exists tollbooth **in set** highway.tollbooths &

vv\_record.record.exitKm = tollbooth.km and

vv\_record.record.direction = tollbooth.direction )

);

--Records reference existing infrastructures

**private** consistentNames(vv\_record: ViaVerdeRecord, highways: **set** **of** Highway, parks: **set** **of** String) res: **bool** ==

( is\_HighwayRecord(vv\_record.record) =>

exists highway **in set** highways & highway.name = vv\_record.record.highway ) and

( is\_ParkingRecord(vv\_record.record) =>

exists park **in set** parks & park = vv\_record.record.park );

**private** hasExitZero(vv\_record: ViaVerdeRecord) res: **bool** ==

if is\_HighwayRecord(vv\_record.record)

**then** vv\_record.record.exitKm = 0

**elseif** is\_ParkingRecord(vv\_record.record)

**then** vv\_record.record.exitTime = 0

**else** false;

--Since 0 in an exit means "didn't exit", it can't happen twice for a vehicle

**private** noTwoRecordsWithExitZero(records: **seq** **of** ViaVerdeRecord, alreadyFound: **bool**) res: **bool** ==

if len records = 0 **then** true

**else** let hasZero = hasExitZero(hd records) **in**

if hasZero and alreadyFound

**then** false

**elseif** hasZero

**then** noTwoRecordsWithExitZero(tl records, true)

**else** noTwoRecordsWithExitZero(tl records, alreadyFound);

**private** greaterOrZero(first: **real**, second: **real**) res: **bool** ==

--Can't exit before entrance

(second >= first and first <> 0)

--May not have already exited

or second = 0;

**end** ViaVerde

# 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 TestViaVerde

**class** TestViaVerde **is subclass of** MyTestCase

**values**

car: ViaVerde`Car = mk\_ViaVerde`Car("12345", mk\_(<Mercedes>,<Benz>,<a>));

**operations**

-- Setup a default environment for the ViaVerde system

**public** Setup : () ==> ViaVerde

Setup () == (

**dcl** vv : ViaVerde := new ViaVerde();

||(

vv.highways := {"A1" |->

mk\_ViaVerde`Highway({<a> |-> 0.1, <b> |-> 0.2, <c> |-> 0.3, <d> |-> 0.4},

{mk\_ViaVerde`Tollbooth(5,<Forward>), mk\_ViaVerde`Tollbooth(105,<Forward>)},

"A1")},

vv.parks := {"Parque Gaia" |-> 0.5},

vv.registeredCars := {car |-> mk\_ViaVerde`ViaVerdeDevice("0","101010",[])});

return vv;

);

-- Tests with valid inputs

**public** testRegisterCar: () ==> ()

testRegisterCar() == (

let vv = Setup(),

newCar = mk\_ViaVerde`Car("23456", mk\_(<Audi>,<A4>,<a>)),

vvDevice = mk\_ViaVerde`ViaVerdeDevice("1","202020",[])

**in** (

vv.registerCar(newCar,"202020","1");

assertEqual(vv.registeredCars(newCar), vvDevice);

)

);

**public** testUnregisterCar: () ==> ()

testUnregisterCar() == (

let vv = Setup()

**in** (

vv.unregisterCar(car);

assertEqual(vv.registeredCars, {|->});

assertEqual(vv.unRegisteredCarsRecords, {car |-> []});

)

);

**public** testRegisterUnregisteredCar: () ==> ()

testRegisterUnregisteredCar() == (

let vv = Setup(),

vvDevice = mk\_ViaVerde`ViaVerdeDevice("1","202020",[])

**in** (

vv.unregisterCar(car);

vv.registerCar(car,"202020","1");

assertEqual(vv.registeredCars(car), vvDevice);

)

);

**public** testGoThroughHighway: () ==> ()

testGoThroughHighway() == (

let vv = Setup(),

entrykm = 5,

exitkm = 105,

entryTime = 1500,

exitTime = 1630,

direction = <Forward>,

entrance = true,

highway = vv.highways("A1"),

cost = (exitkm - entrykm) \* highway.costPerClass(car.brandModel.#3)

**in** (

vv.goThroughHighway(car,entrykm,entryTime,direction,entrance,highway);

vv.goThroughHighway(car,exitkm,exitTime,direction,not entrance,highway);

assertEqual(vv.registeredCars(car).records(len vv.registeredCars(car).records).cost, cost);

)

);

**public** testGoThroughHighwayUnregistered: () ==> ()

testGoThroughHighwayUnregistered() == (

let vv = Setup(),

unRegCar = mk\_ViaVerde`Car("34567", mk\_(<KIA>,<Leo>,<a>)),

entrykm = 5,

exitkm = 105,

entryTime = 1500,

exitTime = 1630,

direction = <Forward>,

entrance = true,

highway = vv.highways("A1"),

cost = (exitkm - entrykm) \* highway.costPerClass(car.brandModel.#3)

**in** (

vv.goThroughHighway(unRegCar,entrykm,entryTime,direction,entrance,highway);

vv.goThroughHighway(unRegCar,exitkm,exitTime,direction,not entrance,highway);

assertEqual(vv.unRegisteredCarsRecords(unRegCar)(len vv.unRegisteredCarsRecords(unRegCar)).cost, cost);

)

);

**public** testGoThroughPark: () ==> ()

testGoThroughPark() == (

let vv = Setup(),

entryTime = 1933,

exitTime = 2000,

entrance = true,

cost = (exitTime - entryTime) \* vv.parks("Parque Gaia")

**in** (

vv.goThroughPark(car,entryTime,entrance,"Parque Gaia");

vv.goThroughPark(car,exitTime,not entrance,"Parque Gaia");

assertEqual(vv.registeredCars(car).records(len vv.registeredCars(car).records).cost, cost);

)

);

**public** testGoThroughParkUnregistered: () ==> ()

testGoThroughParkUnregistered() == (

let vv = Setup(),

unRegCar = mk\_ViaVerde`Car("34567", mk\_(<KIA>,<Leo>,<a>)),

entryTime = 1933,

exitTime = 2000,

entrance = true,

cost = (exitTime - entryTime) \* vv.parks("Parque Gaia")

**in** (

vv.goThroughPark(unRegCar,entryTime,entrance,"Parque Gaia");

vv.goThroughPark(unRegCar,exitTime,not entrance,"Parque Gaia");

assertEqual(vv.unRegisteredCarsRecords(unRegCar)(len vv.unRegisteredCarsRecords(unRegCar)).cost, cost);

)

);

**public** testGoThroughOneTime: () ==> ()

testGoThroughOneTime() == (

let vv = Setup(),

time = 1933,

type = <SCUT>,

cost = 2.5,

record = mk\_ViaVerde`OneTimeRecord(type, time)

**in** (

vv.goThroughOneTime(car,cost,type,time);

assertEqual(vv.registeredCars(car).records(len vv.registeredCars(car).records).record, record);

)

);

**public** testSetCreditCard: () ==> ()

testSetCreditCard() == (

let vv = Setup(),

creditCard = "5555"

**in** (

vv.setCreditCard(car,creditCard);

assertEqual(vv.registeredCars(car).creditCard, creditCard);

)

);

**public** testAddToRecords: () ==> ()

testAddToRecords() == (

let vv = Setup(),

record = mk\_ViaVerde`ViaVerdeRecord(mk\_ViaVerde`OneTimeRecord(<GasStation>, 1000), 20)

**in** (

vv.addToRecords(car,record);

assertEqual(vv.registeredCars(car).records, [record]);

)

);

**public** testRetrieveVehicleLocationPark: () ==> ()

testRetrieveVehicleLocationPark() == (

let vv = Setup()

**in** (

vv.goThroughPark(car,1000,true,"Parque Gaia");

assertEqual(vv.retrieveVehicleLocation(car.licensePlate), "Parque Gaia");

)

);

**public** testRetrieveVehicleLocationHighway: () ==> ()

testRetrieveVehicleLocationHighway() == (

let vv = Setup()

**in** (

vv.goThroughHighway(car,1,1800,<Backward>,true,vv.highways("A1"));

assertEqual(vv.retrieveVehicleLocation(car.licensePlate), "A1");

)

);

-- Entry point that runs all tests with valid inputs

**public** testAll: () ==> ()

testAll() == (

testSetCreditCard();

testAddToRecords();

testRegisterCar();

testUnregisterCar();

testRegisterUnregisteredCar();

testGoThroughHighway();

testGoThroughHighwayUnregistered();

testGoThroughPark();

testGoThroughParkUnregistered();

testGoThroughOneTime();

testRetrieveVehicleLocationPark();

testRetrieveVehicleLocationHighway();

);

-- Tests with invalid inputs

**public** testUniqueCarLicense: () ==> ()

testUniqueCarLicense() == (

let vv = Setup(),

car2 = mk\_ViaVerde`Car("12345", mk\_(<Audi>,<A4>,<a>))

**in** (

vv.registerCar(car2,"101010","1");

)

);

**public** testUnregisterNotRegistered: () ==> ()

testUnregisterNotRegistered() == (

let vv = Setup()

**in** (

vv.unregisterCar(car);

)

);

**public** testGoThroughNotLeaving: () ==> ()

testGoThroughNotLeaving() == (

let vv = Setup(),

entrykm = 2,

entryTime = 1500,

exitTime = 1630,

direction = <Forward>,

entrance = true,

highway = vv.highways("A1")

**in** (

vv.goThroughHighway(car,entrykm,entryTime,direction,entrance,highway);

vv.goThroughPark(car,exitTime, entrance,"Parque Gaia");

)

);

**public** testGoThroughHighwayNoTollbooths: () ==> ()

testGoThroughHighwayNoTollbooths() == (

let vv = Setup(),

entrykm = 2,

exitkm = 200,

entryTime = 1500,

exitTime = 1630,

direction = <Forward>,

entrance = true,

highway = vv.highways("A1")

**in** (

vv.goThroughHighway(car,entrykm,entryTime,direction,entrance,highway);

vv.goThroughHighway(car,exitkm,exitTime,direction,not entrance,highway);

)

);

**end** TestViaVerde

# 5. Model verification

## 5.1 Domain verification

## 5.2 Invariant verification

Invariant to test - **inv** **dom** registeredCars **inter** **dom** unRegisteredCarsRecords = {};

Code -

**public** unregisterCar(car: Car) == (

**atomic**(unRegisteredCarsRecords := unRegisteredCarsRecords **munion** { car |-> registeredCars(car).records};

registeredCars := {car} <-: registeredCars;)

) **pre** car **in set** **dom** registeredCars;

In the beginning the invariant must hold so:

((dom registeredCars) inter (dom unRegisteredCarsRecords)) = {}

Since car must be in registeredCars due to the pre-condition, following the execution of the code of the function we have.

((dom ({car} <-: registeredCars)) inter (dom (unRegisteredCarsRecords munion {car |-> (registeredCars(car).records)}))) = {}))

Since we are merely adding *car* to unregistedCars and restraining the domain by removing *car* from registeredCars the following is obviously true.

(((dom registeredCars) inter (dom unRegisteredCarsRecords)) = {})) => (((dom ({car} <-: registeredCars)) inter (dom (unRegisteredCarsRecords munion {car |-> (registeredCars(car).records)}))) = {}))

# 6. Conclusions

# During the development of this project we have grown more familiar with formal methods for software development and in particular the VDM++ language, we are now capable of solving similar problems and learning similar languages/tools.

# About the project we can conclude that we have successfully fulfilled all of our specifications and we are overall satisfied with the results and the development process despite some problems caused by the lack of information about VDM++.

# In spite of this we believe that we could have broadened the scope of the project by making the modeling more in depth, and regarding the development process the group had some difficulties due to not doing the tests before the code. Regarding possible future extensions, modelling payment methods would be a possibility.

# 7. References

1. VDM-10 Language Manual, Peter Gorm Larsen et al, Overture Technical Report Series No. TR-001, March 2014
2. Formal Methods in Software Engineering course’s VDM++ handouts, Ana Paiva/João Pascoal Faria