

LinkedIn Network

Formal Modeling | VDM++

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

*Something something, a name here?*

Contents

[1. Informal system description and list of requirements 3](#_Toc435377798)

[1.1 Informal system description 3](#_Toc435377799)

[1.2 List of requirements 3](#_Toc435377800)

[2. Visual UML model 4](#_Toc435377801)

[2.1 Use case model 4](#_Toc435377802)

[2.2 Class model 6](#_Toc435377803)

[3. Formal VDM++ model 7](#_Toc435377804)

[3.1 Class MoneyUtils 7](#_Toc435377805)

[3.2 Class Product 8](#_Toc435377806)

[3.3 Class VendingMachine 9](#_Toc435377807)

[4. Model validation 12](#_Toc435377808)

[4.1 Class MyTestCase 12](#_Toc435377809)

[4.2 Class TestVendingMachine 12](#_Toc435377810)

[5. Model verification 16](#_Toc435377811)

[5.1 Example of domain verification 16](#_Toc435377812)

[5.2 Example of invariant verification 16](#_Toc435377813)

[6. Conclusions 17](#_Toc435377814)

[7. References 17](#_Toc435377815)

# 1. Informal system description and list of requirements

## 1.1 Informal system description

**Add**

**Upload Curriculum**

Displays common connections with queried user

**Query Person: <Name\_Here>**

**Common Connections**

John, April

**Search**

**Skills**

C++; Java; …

Search for a user

**Carl Rogers**

**Connections: 42**

John, Johanna, April, …

Shows each of the user connections and total nr. of connections.



**Add**

## 1.2 List of requirements

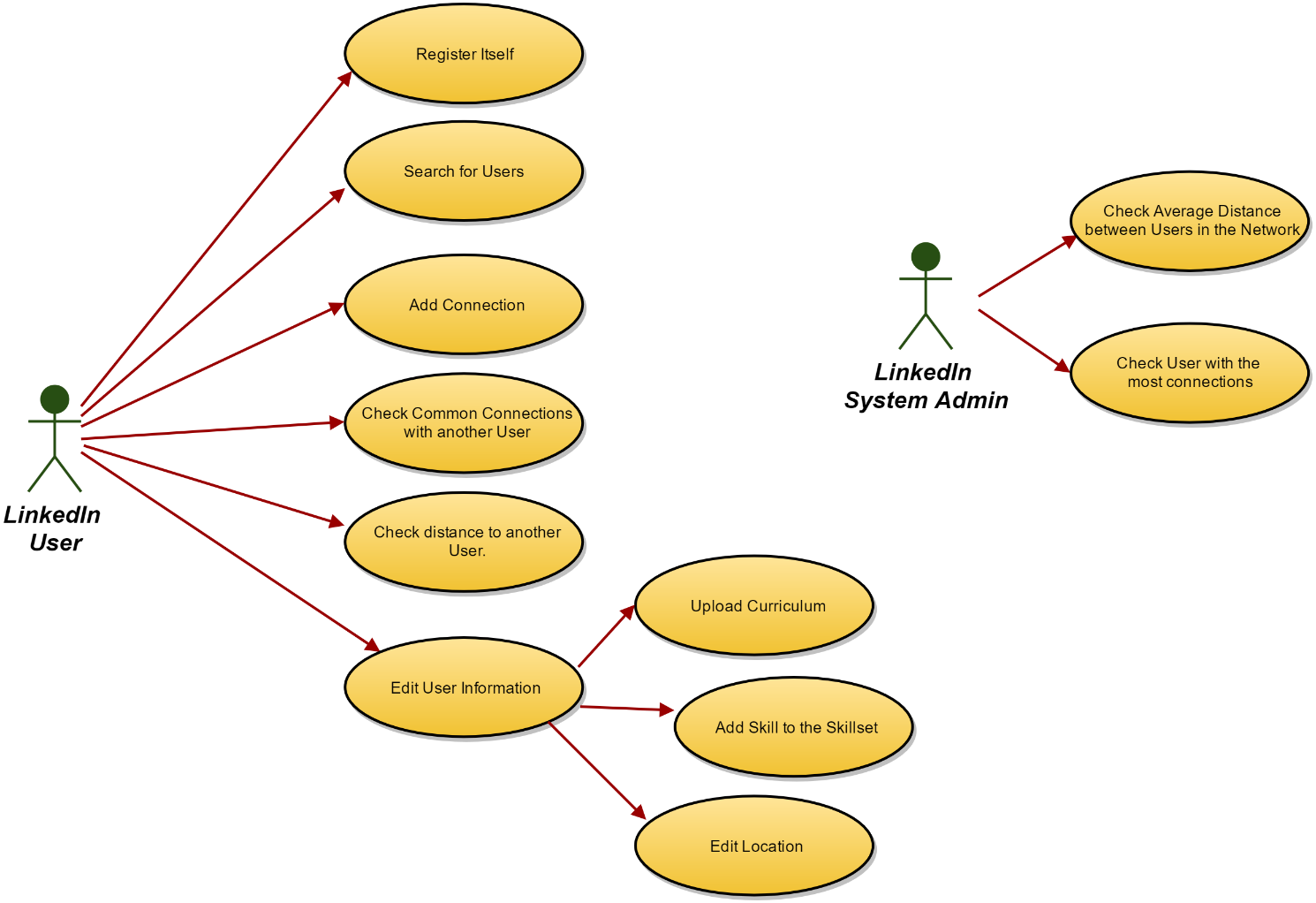
The actors in this system are the **LinkedIn Users**.

|  |  |  |
| --- | --- | --- |
| **Id** | **Priority** | **Description** |
| R1 | Mandatory | A user can register itself on the network, specifying their name. |
| R2 | Mandatory | A user can search for other users by their name. |
| R3 | Mandatory | A user has a list of connections and he can add another user to this list. |
| R4 | Mandatory | A user can upload his CV information. |
| R5 | Mandatory | The system should be able to check the average distance between users in the network. |
| R6 | Mandatory | The system should be able to check the user with the most connections. |
| R7 | Mandatory | A user can query the system for the connections he has in common with another user. |
| R8 | Mandatory | The system should be able to check the distance between two users. |
| R9 | Mandatory | A user can add a skill to his list of skills. |
| R10 | Optional | A user can add and also edit his current location. |
|  |  |  |

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

# 2. Visual UML model

## 2.1 Use case model [[1]](#footnote-1)



The major use case scenarios (to be used later as test scenarios) are described next.

|  |  |
| --- | --- |
| **Scenario** | **Register** |
| **Description** | Normal scenario for a user to create his LinkedIn account. |
| **Pre-conditions** | 1. LinkedIn network has a non-null set of users. *(initial system state)*  2. The user isn’t registered yet on this network. *(initial system state) HOW WILL WE CHECK THIS?*  3. The user provides a valid name, age and gender. *(input)*  *TODO: CONSTRUTORES COM ISTO?* |
| **Post-conditions** | 1. A person instance is created with the provided info. *(final system state)*  2. The created person instance is added to the LinkedIn users. *(final system state)* |
| **Steps** | ---------------------------------------- |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Search for Users** |
| **Description** | Normal scenario for a User search. |
| **Pre-conditions** | 1. LinkedIn network has a non-null set of users. *(initial system state)*  2. The users all have a non-null name. *(initial system state)*  3. The name input for the search is a valid string. *(input)* |
| **Post-conditions** | 1. If a match is made, the user’s info (name, age, gender) is shown, otherwise the system emits a warning. *(output)* |
| **Steps** | ----------------------------------------- |
| **Exceptions** | 1. No match is made. IS THIS SUPPOSED TO BE AN EXCEPTION? |

|  |  |
| --- | --- |
| **Scenario** | **Add Connections** |
| **Description** | Scenario where the user adds a person (another user) as a connection. |
| **Pre-conditions** | 1. The other user is registered in the network. *(initial system state)* |
| **Post-conditions** | 1. The added person is now part of the set of connections of the user. *(final system state)* |
| **Steps** | ------------------------------------------- |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Check Common Connections with another User** |
| **Description** | Scenario where the user checks which people both he and another user are connected to. |
| **Pre-conditions** | 1. The user to check is registered in the network. *(initial system state)* |
| **Post-conditions** | 1. The number of common connections with the other user is bigger or equal to zero, which the system will display. *(output)* |
| **Steps** | ------------------------------------------- |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Check Distance to Another User** |
| **Description** | Normal scenario where a user checks how many connections away from him another user is. If a user has no connections, his distance is infinite. |
| **Pre-conditions** | 1. The user to check is registered in the network. *(initial system state)* |
| **Post-conditions** | 1. The minimum number of connections to reach the other user (distance) is bigger or equal to zero (in case it’s himself), which the system will display. *(output)* |
| **Steps** | ------------------------------------------- |
| **Exceptions** | 1. If either user has no connections – the distance to one another is infinite; |

|  |  |
| --- | --- |
| **Scenario** | **Upload Curriculum** |
| **Description** | Normal scenario in which a user uploads his whole CV into the system, taking the form of a “string” that represents the PDF or other type of file. |
| **Pre-conditions** | 1. Is there any initial state to respect here? *(initial system state)*  2. The CV is a valid string type. *(input)* |
| **Post-conditions** | 1. The CV field is updated with the input. *(final system state)* |
| **Steps** | --------------------------------------------- |
| **Exceptions** | (unspecified) |

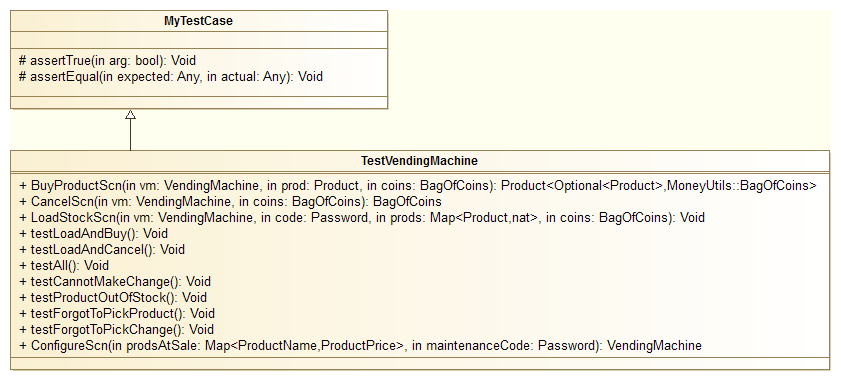
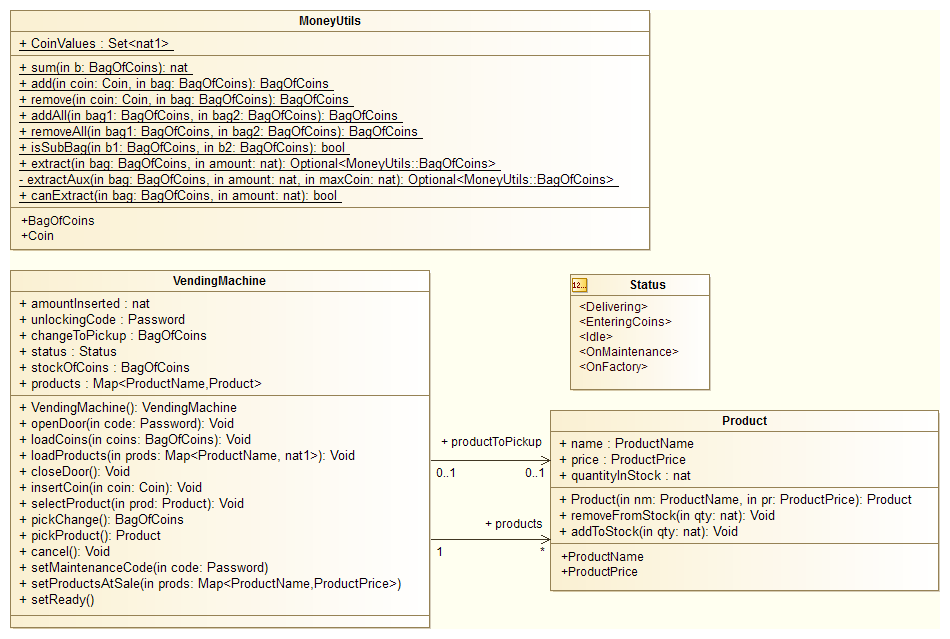
|  |  |
| --- | --- |
| **Scenario** | **Add Skill to the Skillset** |
| **Description** | Normal scenario in which a user adds a new skill to his set of existing skills. |
| **Pre-conditions** | 1. The skillset is a non-null set of skills, which it means it is initialized and has 0 or more items. *(initial system state)*  2. The skill to insert is a valid String that doesn’t exist in the user’s skillset yet. *(input)* |
| **Post-conditions** | 1. The skill is added to the user’s skillset. |
| **Steps** | ------------------------------------------------ |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Edit Location** |
| **Description** | Normal scenario in which a user edits the place of his current location. |
| **Pre-conditions** | 1. Is there any initial state to respect here? *(initial system state)*  2. The new location is a valid string type. *(input)* |
| **Post-conditions** | 1. The user’s location is updated with the input. *(final system state)* |
| **Steps** | ---------------------------------------------------- |
| **Exceptions** | (unspecified) |

|  |  |
| --- | --- |
| **Scenario** | **Check Average Distance between Users in the network** |
| **Description** | Scenario where the system is requested to check the average distance between all the existing users. |
| **Pre-conditions** | 1. The network’s set of users isn’t null or empty. *(initial system state)* SHOULD WE ALLOW IT TO BE EMPTY? |
| **Post-conditions** | 1. The average distance between users is bigger or equal to zero (in there’s only 1 user), which the system will display. *(output)* |
| **Steps** | -------------------------------------------------- |
| **Exceptions** | 1. If no user has connections – the average distance is infinite; DO WE LET IT BE LIKE THIS? How do we calculate in case there are only some users without any connections? |

|  |  |
| --- | --- |
| **Scenario** | **Check User with the most connections** |
| **Description** | Scenario where the system is requested to check which existing user has the most connections. In case of draw, returns all the ones with the most connections. |
| **Pre-conditions** | 1. The network’s set of users isn’t null or empty. *(initial system state)* SHOULD WE ALLOW IT TO BE EMPTY? |
| **Post-conditions** | 1. The system displays the Person or set of People that have the most connections in the system. *(output)* |
| **Steps** | ------------------------------------------------- |
| **Exceptions** | (unspecified) |

## 2.2 Class model [[2]](#footnote-2) [[3]](#footnote-3)



|  |  |
| --- | --- |
| **Class** | **Description** |
| MoneyUtils | Defines utility types and functions to work with bags (multisets) of coins. |
| Product | Defines a product at sale in a vending machine. |
| VendingMachine | Core model; defines the state variables and operations available to the users. |
| MyTestCase | Superclass for test classes; defines assertEquals and assertTrue. |
| TestVendingMachine | Defines the test/usage scenarios and test cases for the vending machine. |

# 3. Formal VDM++ model

## 3.1 Class MoneyUtils [[4]](#footnote-4)

**class** MoneyUtils

/\*

Contains utility types and functions to work with bags (multisets) of coins.

Illustrates the definition of auxiliary data types, as well as the definition

of a functionality (extract/makeChange) at different levels of abstraction.

JPF, FEUP, MFES, 2014/15.

\*/

**values**

-- possible coin values, in cents of euros

**public** CoinValues : **set** **of** **nat1** = {1, 2, 5, 10, 20, 50, 100, 200};

**types**

**public** Coin = **nat1**

**inv** c == c in **set** CoinValues;

**public** BagOfCoins = **map** Coin **to** **nat1**; -- maps coin values to quantities

**functions**

-- Computes the total amount in a bag of coins

**public** sum: BagOfCoins -> **nat**

sum(b) ==

if b = {|->} **then** 0

**else** let c **in set** dom b **in** b(c) \* c + sum({c} <-: b);

-- Adds a coin to a bag of coins and returns the new bag

**public** add: Coin \* BagOfCoins -> BagOfCoins

add(coin, bag) ==

if coin in **set** dom bag **then** bag ++ { coin |-> bag(coin) + 1}

**else** bag munion {coin |-> 1};

-- Removes a coin from a bag of coins and returns the new bag

**public** remove: Coin \* BagOfCoins -> BagOfCoins

remove(coin, bag) ==

if bag(coin) = 1 **then** {coin} <-: bag **else** bag ++ {coin |-> bag(coin) - 1}

**pre** coin in **set** dom bag;

-- Adds two bags of coins and returns the new bag

**public** addAll: BagOfCoins \* BagOfCoins -> BagOfCoins

addAll(bag1, bag2) ==

{ c |-> (if c in **set** dom bag1 **then** bag1(c) **else** 0)

+ (if c in **set** dom bag2 **then** bag2(c) **else** 0) |

c **in set** dom bag1 union dom bag2};

-- Subtracts the first bag of coins from the second one, and returns the result

**public** removeAll: BagOfCoins \* BagOfCoins -> BagOfCoins

removeAll(bag1, bag2) ==

{c |-> bag2(c) - (if c in **set** dom bag1 **then** bag1(c) **else** 0) |

c **in set** dom bag2 & not (c in **set** dom bag1 and bag1(c) = bag2(c))}

**pre** isSubBag(bag1, bag2);

-- Checks if the first bag of coins is a subbag of the second one

**public** isSubBag: BagOfCoins \* BagOfCoins -> **bool**

isSubBag(b1, b2) ==

dom b1 subset dom b2

and forall c **in set** dom b1 & b1(c) <= b2(c);

/\*

-- Extracts (computes) a subbag that makes up a given amount.

-- Version 1, highest possible level of abstraction, following definition (not used).

public extract1: BagOfCoins \* nat -> BagOfCoins

extract1(bag, amount) ==

let e in set allSubBags(bag) be st sum(e) = amount in e

pre canExtract1(bag, amount);

-- Checks if is is possible to make a given amount from a bag.

-- Version 1, highest possible level of abstraction, following definition (not used).

public canExtract1: BagOfCoins \* nat -> bool

canExtract1(bag, amount) ==

exists e in set allSubBags(bag) & sum(e) = amount;

-- Auxiliary function for version 1, that generates a set with all possible subbags

-- of a a bag of coins (not used).

private allSubBags: BagOfCoins -> set of BagOfCoins

allSubBags(bag) ==

if bag = {|->} then {{|->}}

else let c in set dom bag in

dunion {{s, add(c, s)} | s in set allSubBags(remove(c, bag))};

\*/

-- Extracts (computes) a subbag that makes up a given amount.

-- Version 2, less abstract, following a greedy algorithm with backtracing.

-- Returns nil if there is no solution.

**public** extract: BagOfCoins \* **nat** -> [BagOfCoins]

extract(bag, amount) ==

extractAux(bag, amount, amount);

-- Auxiliary function that does the work of 'extract'.

-- The third argument is the maximum value of coins to use.

**private** extractAux: BagOfCoins \* **nat** \* **nat** -> [BagOfCoins]

extractAux(bag, amount, maxCoin) ==

if amount = 0 **then** {|->}

**else** let coins = reverse [c | c **in set** dom bag & c <= maxCoin and c <= amount] **in**

if coins = [] **then** nil

**else** let c = hd coins,

remaining = extractAux(remove(c, bag), amount - c, c)

**in** if remaining <> nil **then** add(c, remaining)

**else** extractAux(bag, amount, c - 1);

-- Checks if is is possible to make a given amount from a bag.

-- Version 2, less abstract.

**public** canExtract: BagOfCoins \* **nat** -> **bool**

canExtract(bag, amount) ==

extract(bag, amount) <> nil;

**end** MoneyUtils

## 3.2 Class Product

**class** Product

/\*

Defines a product at sale in a vending machine.

JPF, FEUP, MFES, 2014/15.

\*/

**types**

**public** ProductName = **seq1** **of** **char**;

**public** ProductPrice = **nat1**; -- in cents

**instance variables**

/\* **Tip**: variables are declared public to facilitate queries \*/

**public** name: ProductName;

**public** price: ProductPrice;

**public** quantityInStock : **nat** := 0;

**operations**

**public** Product : ProductName \* ProductPrice ==> Product

Product(nm, pr) == (

name := nm;

price := pr;

return self

);

**public** removeFromStock: **nat** ==> ()

removeFromStock(qty) ==

quantityInStock := quantityInStock - qty

**pre** qty <= quantityInStock;

**public** addToStock: **nat** ==> ()

addToStock(qty) ==

quantityInStock := quantityInStock + qty;

**end** Product

## 3.3 Class VendingMachine

**class** VendingMachine

/\*

Contains the core model of the vending machine.

Defines the state variables and operations available to the users.

Among other features, illustrates the usage of 'atomic'.

JPF, FEUP, MFES, 2014/15.

\*/

**types**

**public** BagOfCoins = MoneyUtils`BagOfCoins;

**public** ProductName = Product`ProductName;

**public** ProductPrice = Product`ProductPrice;

**public** Status=**<OnFactory>** | **<OnMaintenance>** | **<Idle>** | **<EnteringCoins>** | **<Delivering>**;

**public** Password = **seq** **of** **char**;

**instance variables**

/\* Tip: variables declared public to facilitate queries \*/

-- Items observable by buyer (in display, selection buttons, and pickup slots):

**public** products: **map** ProductName **to** Product := { |-> };

**public** status : Status := <OnFactory>;

**public** amountInserted: **nat** := 0;

**public** changeToPickup : BagOfCoins := {|->};

**public** productToPickup : [Product] := nil;

-- Items observable by maintenance operator:

**public** stockOfCoins: BagOfCoins := {|->};

-- Items observable by factory operator:

**public** maintenanceCode : Password := [];

**inv** amountInserted <> 0 <=> status = <EnteringCoins>;

**inv** changeToPickup <> {|->} or productToPickup <> nil <=> status = <Delivering>;

**inv** productToPickup <> nil => productToPickup in **set** rng products;

**operations**

/\*\* FACTORY OPERATIONS \*\*/

**public** VendingMachine: () ==> VendingMachine

VendingMachine() ==

return self;

**public** setMaintenanceCode: Password ==> ()

setMaintenanceCode(code) ==

maintenanceCode := code

**pre** status = <OnFactory>;

**public** setProductsAtSale: **map** ProductName **to** ProductPrice ==> ()

setProductsAtSale(prods) ==

products := {name |-> new Product(name, prods(name)) | name **in set** dom prods}

**pre** status = <OnFactory>;

**public** setReady: () ==> ()

setReady() ==

status := <Idle>

**pre** status = <OnFactory>;

/\*\* MAINTENANCE OPERATIONS \*\*/

**public** openDoor: Password ==> ()

openDoor(code) ==

if code = maintenanceCode **then**

status := <OnMaintenance>

**pre** status = <Idle>;

**public** loadCoins: BagOfCoins ==> ()

loadCoins(coins) ==

stockOfCoins := coins

**pre** status = <OnMaintenance>;

-- 'prods' is a mapping from product name to number of items

**public** loadProducts: **map** ProductName **to** **nat1** ==> ()

loadProducts(prods) ==

for **all** nm **in set** dom prods **do**

products(nm).addToStock(prods(nm))

**pre** status = <OnMaintenance>

and dom prods subset dom products;

**public** closeDoor: () ==> ()

closeDoor() ==

status := <Idle>

**pre** status = <OnMaintenance>;

/\*\* BUYER OPERATIONS \*\*/

**public** insertCoin: MoneyUtils`Coin ==> ()

insertCoin(coin) ==

atomic (

stockOfCoins := MoneyUtils`add(coin, stockOfCoins);

amountInserted := amountInserted + coin;

status := <EnteringCoins>

)

**pre** status in **set** {<Idle>, <EnteringCoins>};

**public** selectProduct: ProductName ==> ()

selectProduct(prodName) ==

let p = products(prodName),

chg = MoneyUtils`extract(stockOfCoins, amountInserted - p.price) **in** (

p.removeFromStock(1);

atomic (

stockOfCoins := MoneyUtils`removeAll(chg, stockOfCoins);

amountInserted := 0;

changeToPickup := chg;

productToPickup := p;

status := <Delivering>

)

)

**pre** status = <EnteringCoins>

and prodName in **set** dom products

and products(prodName).quantityInStock > 0

and amountInserted >= products(prodName).price

and MoneyUtils`canExtract(stockOfCoins,

amountInserted - products(prodName).price);

**public** pickChange: () ==> BagOfCoins

pickChange() ==

let r = changeToPickup **in** (

atomic(

changeToPickup := {|->};

status := if productToPickup = nil **then** <Idle> **else** <Delivering>

);

return r

)

**pre** changeToPickup <> {|->};

**public** pickProduct: () ==> ProductName

pickProduct() == (

let r = productToPickup.name **in** (

atomic (

productToPickup := nil;

status := if changeToPickup = {|->} **then** <Idle> **else** <Delivering>

);

return r

)

)

**pre** productToPickup <> nil;

**public** cancel: () ==> ()

cancel() ==

let chg = MoneyUtils`extract(stockOfCoins, amountInserted) **in**

atomic (

stockOfCoins := MoneyUtils`removeAll(chg, stockOfCoins);

amountInserted := 0;

changeToPickup := chg;

status := <Delivering>

)

**pre** status = <EnteringCoins>;

**end** VendingMachine

# 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 TestVendingMachine

**class** TestVendingMachine **is subclass of** MyTestCase

/\*

Contains the test cases for the vending machine.

Illustrates a scenario-based testing approach.

The test cases cover all usage scenarios as well as all states and transitions.

Also illustrates the usage of assertions and '||'.

JPF, FEUP, MFES, 2014/15.

\*/

**operations**

/\*\*\*\*\* USE CASE SCENARIOS \*\*\*\*\*\*/

-- Normal scenario for configuring the vending machine in the factory,

-- as described in section 2.1 of the report, covering requirement R1.

-- 'prodsAtSale' is a map from names to prices.

**public** ConfigureScn: **map** Product`ProductName **to** Product`ProductPrice \* VendingMachine`Password ==> VendingMachine

ConfigureScn(prodsAtSale, maintenanceCode) == (

**dcl** vm : VendingMachine := new VendingMachine();

|| ( vm.setProductsAtSale(prodsAtSale),

vm.setMaintenanceCode(maintenanceCode));

vm.setReady();

return vm;

)

**post** {p.name |-> p.price | p **in set** rng RESULT.products} = prodsAtSale /\*1\*/

and RESULT.maintenanceCode = maintenanceCode /\*2\*/

and RESULT.stockOfCoins = {|->} /\*3\*/

and forall p **in set** rng RESULT.products & p.quantityInStock = 0 /\*4\*/

and RESULT.status = <Idle> /\*5\*/;

-- Normal scenario for loading (adding) products and

-- setting (adding/removing) the stock of coins in a vending machine,

-- as described in section 2.1 of the report, covering requirement R2.

**public** LoadStockScn: VendingMachine \* VendingMachine`Password \*

**map** Product`ProductName **to** Product`ProductPrice \* MoneyUtils`BagOfCoins ==> ()

LoadStockScn(vm, code, prods, coins) ==

(

-- manual old values

**dcl** oldQuantityInStock : **map** Product **to** **nat** :=

{p |-> p.quantityInStock | p **in set** rng vm.products};

-- steps

vm.openDoor(code);

|| (vm.loadProducts(prods), vm.loadCoins(coins));

vm.closeDoor();

-- manual post-condition checking (not supported otherwise)

assertTrue(forall prodName **in set** dom prods &

let p = vm.products(prodName) **in**

p.quantityInStock = oldQuantityInStock(p) + prods(prodName)) /\*1\*/

)

**pre** vm.status = <Idle> /\*1\*/

and code = vm.maintenanceCode /\*2\*/

and dom prods subset dom vm.products /\*3\*/

**post** vm.stockOfCoins = coins /\*2\*/

and vm.status = <Idle>; /\*3\*/

-- Normal purchase scenario in a vending machine,

-- as described in section 2.1 of the report, covering requirements R3 and R4.

**public** BuyProductScn: VendingMachine \* Product`ProductName \* MoneyUtils`BagOfCoins

==> [Product`ProductName] \* MoneyUtils`BagOfCoins

BuyProductScn(vm, prod, coins) == (

-- manual old values

**dcl** oldQuantityInStock : **nat** := vm.products(prod).quantityInStock;

**dcl** oldStockOfCoins : MoneyUtils`BagOfCoins := vm.stockOfCoins;

-- steps

**dcl** inserted : **nat** := 0;

**dcl** deliveredProd : [**seq** **of** **char**] := nil;

**dcl** change : MoneyUtils`BagOfCoins := {|->};

**dcl** result : [Product`ProductName] \* MoneyUtils`BagOfCoins;

for **all** c **in set** dom coins **do**

for **all** - **in set** {1 , ..., coins(c)} **do** (

vm.insertCoin(c);

inserted := inserted + c;

assertEqual(inserted, vm.amountInserted)

);

vm.selectProduct(prod);

|| (deliveredProd := vm.pickProduct(),

if MoneyUtils`sum(coins) > vm.products(prod).price **then**

change := vm.pickChange());

result := mk\_(deliveredProd, change);

-- manual post-condition checking (not supported otherwise)

assertTrue(vm.products(prod).quantityInStock = oldQuantityInStock - 1); /\*3\*/

assertTrue(MoneyUtils`addAll(vm.stockOfCoins, change) =

MoneyUtils`addAll(oldStockOfCoins, coins)); /\* 4\*/

return result;

)

**pre** vm.status = <Idle> /\*1\*/

and (prod in **set** dom vm.products and vm.products(prod).quantityInStock > 0) /\*2\*/

and MoneyUtils`sum(coins) >= vm.products(prod).price /\*3\*/

and MoneyUtils`canExtract(MoneyUtils`addAll(vm.stockOfCoins, coins),

MoneyUtils`sum(coins) - vm.products(prod).price) /\*4\*/

**post** let **mk\_**(deliveredProd, change) = RESULT **in** (

deliveredProd = prod /\*1\*/

and MoneyUtils`sum(change)=MoneyUtils`sum(coins)-vm.products(prod).price/\*2\*/

and vm.status = <Idle> /\*5\*/

);

-- Exceptional buying scenario in which the user cancels the purchase.

-- as described in section 2.1 of the report, covering requirement R5.

**public** CancelScn: VendingMachine \* MoneyUtils`BagOfCoins ==> MoneyUtils`BagOfCoins

CancelScn(vm, coins) == (

-- 'manual' old value

**dcl** oldStockOfCoins : MoneyUtils`BagOfCoins := vm.stockOfCoins;

-- steps

**dcl** inserted : **nat** := 0;

**dcl** result : MoneyUtils`BagOfCoins;

for **all** c **in set** dom coins **do**

for **all** - **in set** {1 , ..., coins(c)} **do** (

vm.insertCoin(c);

inserted := inserted + c;

assertEqual(inserted, vm.amountInserted)

);

vm.cancel();

result := vm.pickChange();

-- 'manual' post-condition checking (not supported otherwise)

assertTrue(MoneyUtils`addAll(oldStockOfCoins, coins) =

MoneyUtils`addAll(vm.stockOfCoins, result)); /\*2\*/

return result

)

**pre** vm.status = <Idle> /\*1\*/

and MoneyUtils`sum(coins) > 0 /\*2\*/

**post** MoneyUtils`sum(RESULT) = MoneyUtils`sum(coins) /\*1\*/

and vm.status = <Idle>; /\*3\*/

/\*\*\*\*\* TEST CASES WITH VALID INPUTS \*\*\*\*\*\*/

-- Test case in which we initialize a vending machine and

-- then buy two products, the first one with exact money and

-- the second one with change.

**public** testLoadAndBuy: () ==> ()

testLoadAndBuy() == (

**dcl** vm: VendingMachine := ConfigureScn({"Bolicao" |-> 50, "Bongo" |-> 70}, "xa1!");

LoadStockScn(vm, "xa1!", {"Bolicao" |-> 1, "Bongo" |-> 1}, { |-> });

let **mk\_**(-, change) = BuyProductScn(vm, "Bolicao", {20 |-> 1, 10 |-> 3}) **in**

assertEqual({ |-> }, change);

let **mk\_**(-, change) = BuyProductScn(vm, "Bongo", {20 |-> 4}) **in**

assertEqual({10 |-> 1}, change)

);

-- Test case in which we initialize a vending machine an then enter coins and cancel.

-- Also forces backtracking in the greedy algorithm for making change.

**public** testLoadAndCancel: () ==> ()

testLoadAndCancel() == (

let vm = ConfigureScn({"Bolicao" |-> 50}, "xa1!"),

coins = {20 |-> 3}

**in** (

LoadStockScn(vm, "xa1!", {"Bolicao" |-> 1}, { 50 |-> 1});

assertEqual(coins, CancelScn(vm, coins))

)

);

-- Entry point that runs all tests with valid inputs

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

testAll() == (

testLoadAndBuy();

testLoadAndCancel();

);

/\*\*\*\*\* TEST CASES WITH INVALID INPUTS (EXECUTE ONE AT A TIME) \*\*\*\*\*\*/

**public** testCannotMakeChange: () ==> ()

testCannotMakeChange() == (

let vm = ConfigureScn({"Bolicao" |-> 50}, "xa1!") **in** (

LoadStockScn(vm, "xa1!", {"Bolicao" |-> 1}, { |-> });

vm.insertCoin(100);

vm.selectProduct("Bolicao"); -- breaks pre-condition

)

);

**public** testProductOutOfStock: () ==> ()

testProductOutOfStock() == (

let vm = ConfigureScn({"Bolicao" |-> 50}, "xa1!") **in** (

vm.insertCoin(50);

vm.selectProduct("Bolicao"); -- breaks pre-condition

)

);

**public** testForgotToPickProduct: () ==> ()

testForgotToPickProduct() == (

let vm = ConfigureScn({"Bolicao" |-> 50}, "xa1!") **in** (

LoadStockScn(vm, "xa1!", {"Bolicao" |-> 1}, { |-> });

vm.insertCoin(50);

vm.selectProduct("Bolicao");

-- forgot: vm.pickProduct();

vm.insertCoin(50); -- breaks pre-condition

)

);

**public** testForgotToPickChange: () ==> ()

testForgotToPickChange() == (

let vm = ConfigureScn({"Bolicao" |-> 50}, "xa1!") **in** (

LoadStockScn(vm, "xa1!", {"Bolicao" |-> 1}, { |-> });

vm.insertCoin(50);

vm.cancel();

-- forgot: vm.pickChange();

vm.insertCoin(50); -- breaks pre-condition

)

);

**end** TestVendingMachine

# 5. Model verification

## 5.1 Example of domain verification

One of the proof obligations generated by Overture is:

|  |  |  |
| --- | --- | --- |
| No. | PO Name | Type |
| 61 | VendingMachine`setProductsAtSale | legal map application |

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

**public** setProductsAtSale: **map** ProductName **to** ProductPrice ==> ()

setProductsAtSale(prods) ==

products := { name |-> **new** Product(name, prods(name)) | **name in set dom prods**};

In this case the proof is trivial because the quantification '**name in set dom prods'** assures that the map is accesses only inside its domain.

## 5.2 Example of invariant verification

Another proof obligation generated by Overture is:

|  |  |  |
| --- | --- | --- |
| No. | PO Name | Type |
| 73 | VendingMachine`pickChange | state invariant holds |

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

**public** pickChange: () ==> BagOfCoins

pickChange() ==

**let** r = changeToPickup **in** (

**atomic**(

changeToPickup := {|->};

status := **if** productToPickup = **nil** **then** **<Idle>** **else** **<Delivering>**

);

**return** r

)

**pre** changeToPickup **<>** {|->};

The relevant invariant under analysis is:

**inv** changeToPickup **<>** {|->} **or** productToPickup **<>** **nil** <=> status = **<Delivering>**;

After the execution of the '**atomic**' block we have (technically, this is the post-condition of the block):

changeToPickup = {|->} **and** (status = **if** productToPickup = **nil** **then** **<Idle>** **else** **<Delivering>)**

We have to prove that this implies that the invariant holds, i.e., that the following condition holds:

(changeToPickup = {|->} **and** (status = **if** productToPickup = **nil** **then** **<Idle>** **else** **<Delivering>)) =>**

**(**changeToPickup **<>** {|->} **or** productToPickup **<>** **nil** <=> status = **<Delivering>)**

This formally implies that:

(status = **if** productToPickup = **nil** **then** **<Idle>** **else** **<Delivering>)**

**=> (**productToPickup **<>** **nil** <=> status = **<Delivering>)**

which can be rewritten as:

(**if** productToPickup = **nil** **then** status = **<Idle>** **else** status = **<Delivering>)**

**=> (if** productToPickup **=** **nil** **then** status <> **<Delivering>) else** status = **<Delivering>)**

which is obviously true.

# 6. Conclusions

The model that was developed covers all the requirements.

If space permitted, as future work, it would be useful to also specify and test the behavior in exceptional conditions.

This project took approximately 16 hours to develop.

# 7. References

1. Validated Designs for Object-oriented Systems, J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, M. Verhoef, Springer, 2005
2. VDM-10 Language Manual, Peter Gorm Larsen et al, Overture Technical Report Series No. TR-001, March 2014
3. Overture tool web site, http://overturetool.org

1. **Tip**: Some use cases are more general and get divided into their sub-components, like “Edit User Information”, which is divided into the various ways an user can edit his info. [↑](#footnote-ref-1)
2. **Tip**: Typically, you may have three kinds of classes, for modeling (i) data types , (i) the system itself and (iii) test cases. [↑](#footnote-ref-2)
3. **Tip:** It is more practical to create the class diagrams in the end of the project, using the export/import features of Overture and Modelio. In the begin of the project, you can simply sketch the class structure on paper to organize ideas. [↑](#footnote-ref-3)
4. **Tip**: Copy/paste the source code from the files generated by Overture with coverage information. [↑](#footnote-ref-4)