

Formal Modeling of a Vending Machine in VDM++

João Pascoal Faria

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

1/12/2015

Contents

[1. Informal system description and list of requirements 3](#_Toc405150495)

[1.1 Informal system description 3](#_Toc405150496)

[1.2 List of requirements 3](#_Toc405150497)

[2. Visual UML model 4](#_Toc405150498)

[2.1 Use case diagram 4](#_Toc405150499)

[2.2 State machine diagram 5](#_Toc405150500)

[2.3 Class diagram 6](#_Toc405150501)

[3. Formal VDM++ model 7](#_Toc405150502)

[3.1 Class MoneyUtils 7](#_Toc405150503)

[3.2 Class Product 9](#_Toc405150504)

[3.3 Class VendingMachine 9](#_Toc405150505)

[4. Model validation 12](#_Toc405150506)

[4.1 Class MyTestCase 12](#_Toc405150507)

[4.2 Class TestVendingMachine 12](#_Toc405150508)

[5. Model Verfication 16](#_Toc405150509)

[6. Conclusions 16](#_Toc405150510)

[7. References 16](#_Toc405150511)

# 1. Informal system description and list of requirements

## 1.1 Informal system description

Slot for inserting coins

Slot to pick change

Credit: 1.20

Displays status and amount inserted

Buttons to select products (showing price and name)

Slot to pick product

Vending Machine

Cancel button

Keypad to enter key to open machine (not visible on the front)

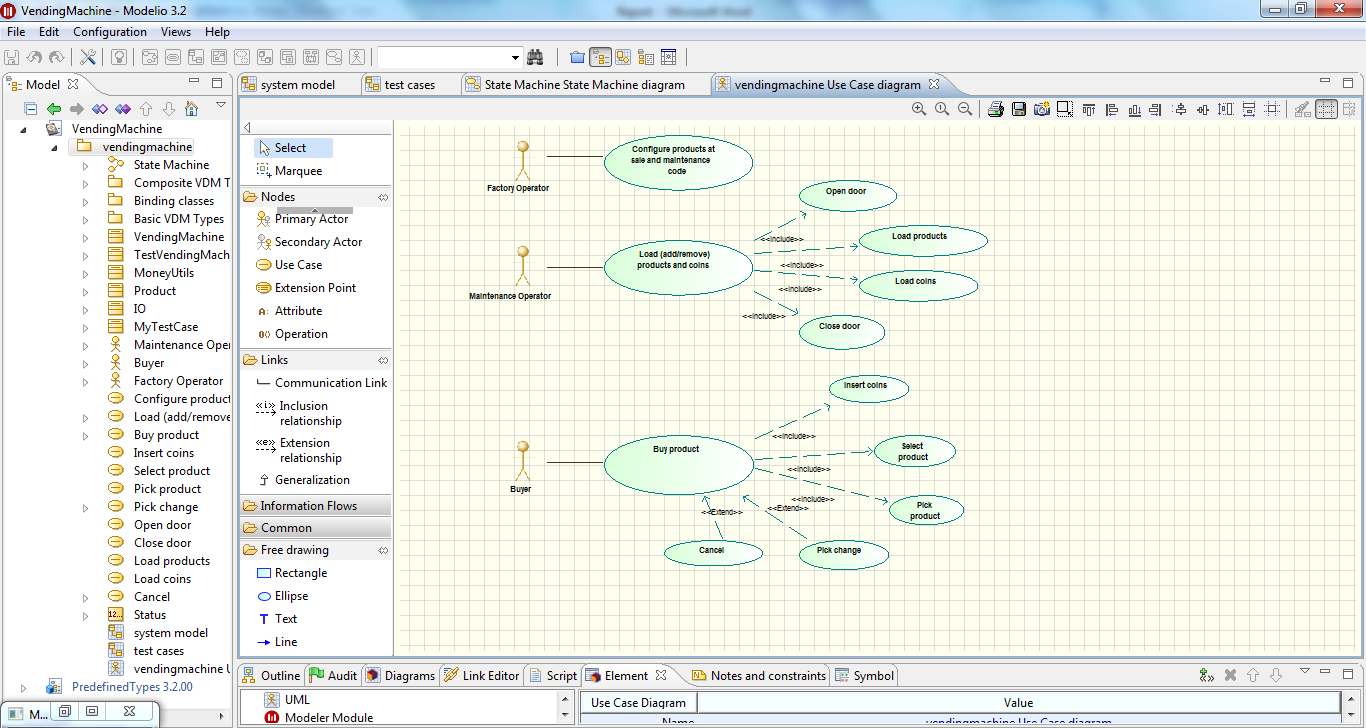


## 1.2 List of requirements

|  |  |  |
| --- | --- | --- |
| **Id** | **Priority** | **Description** |
|  |  |  |
|  |  |  |
|  |  |  |

# 2. Visual UML model

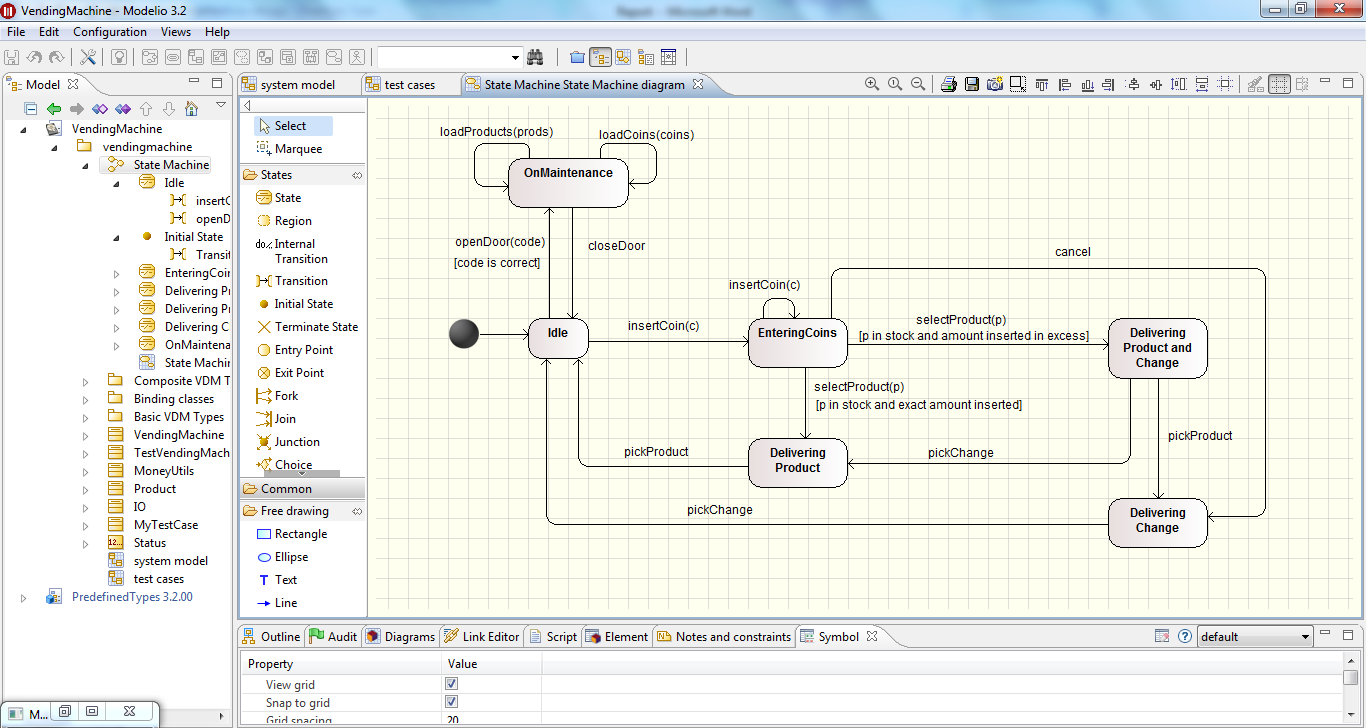
## 2.1 Use case diagram



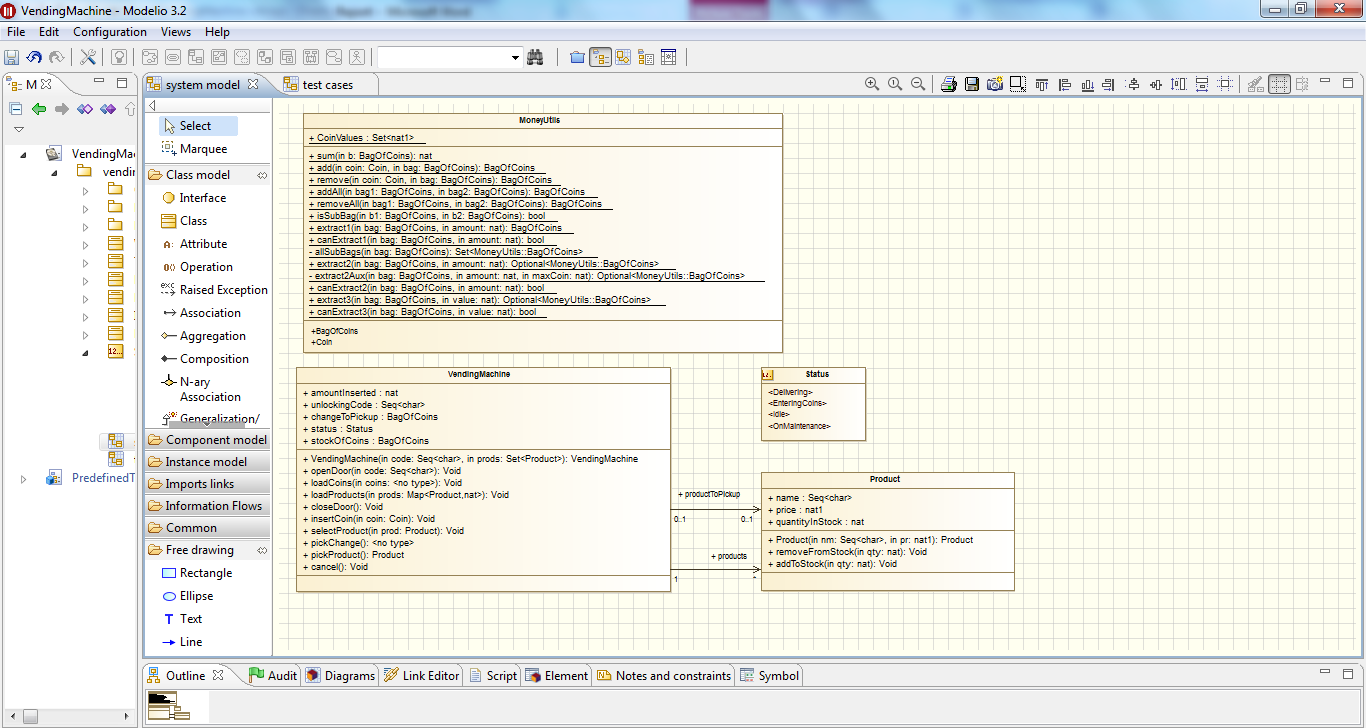
The normal scenarios corresponding to the major use cases are described in section 4.1.

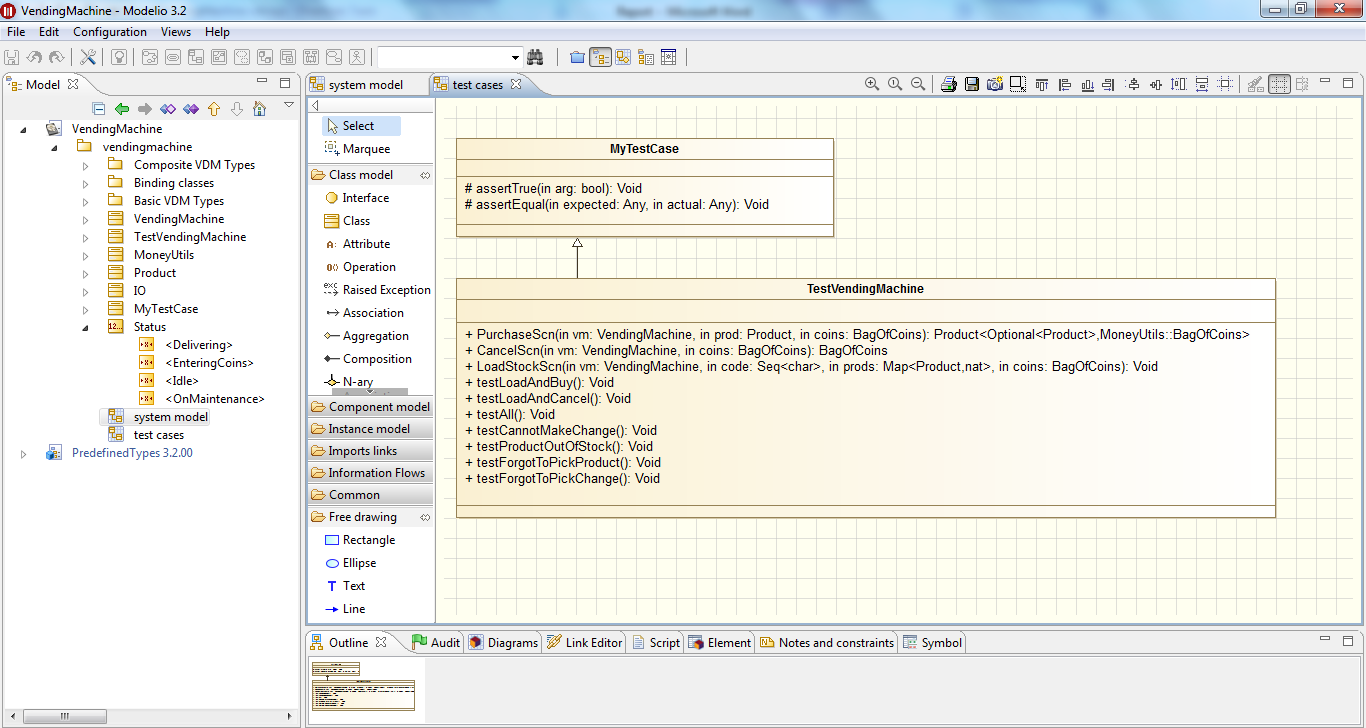
## 2.2 State machine diagram

The next figure shows the VendingMachine lifecycle.



## 2.3 Class diagram





|  |  |
| --- | --- |
| **Class** | **Description** |
| MoneyUtils |  |
| Product |  |
| VendingMachine |  |
| MyTestCase |  |
| TestVendingMachine |  |

# 3. Formal VDM++ model

## 3.1 Class MoneyUtils

**class** MoneyUtils

/\*

Contains utility definitions to work with bags (multisets) of coins.

Illustrates the definition of auxliary 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**;

**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 tow 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 another 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 the definition.

**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 the definition.

**public** canExtract1: BagOfCoins \* **nat** -> **bool**

canExtract1(bag, amount) ==

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

-- Auxiliary function that generates a set with all possible subbags

-- of a a bag of coins

**private** allSubBags: BagOfCoins -> **set** **of** BagOfCoins

allSubBags(bag) ==

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

**else** let c **in set** dom bag **in**

let subs = allSubBags(remove(c, bag)) **in**

subs union { add(c, s) | s **in set** subs};

-- 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** extract2: BagOfCoins \* **nat** -> [BagOfCoins]

extract2(bag, amount) ==

extract2Aux(bag, amount, amount);

-- Auxiliary function that does the work of 'extract2'.

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

**private** extract2Aux: BagOfCoins \* **nat** \* **nat** -> [BagOfCoins]

extract2Aux(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 **in**

let remaining = extract2Aux(remove(c, bag), amount - c, c) **in**

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

**else** extract2Aux(bag, amount, c - 1);

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

-- Version 1, less abstract.

**public** canExtract2: BagOfCoins \* **nat** -> **bool**

canExtract2(bag, amount) ==

extract2(bag, amount) <> nil;

**operations**

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

-- Version 3, similar to version 2, but following an imperative style.

-- Follows a greedy algorithm with backtracing.

-- Returns nil if there is no solution.

**public** **static** extract3: BagOfCoins \* **nat** ==> [BagOfCoins]

extract3(bag, value) ==

(

if value = 0 **then**

return {|->};

for c **in** reverse [c | c **in set** dom bag & c <= value] **do**

let remaining = extract3(remove(c, bag), value - c) **in**

if remaining <> nil **then**

return add(c, remaining);

return nil

);

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

-- Version 3, similar to version 2.

**public** **static** canExtract3: BagOfCoins \* **nat** ==> **bool**

canExtract3(bag, value) ==

return extract3(bag, value) <> nil

**end** MoneyUtils

## 3.2 Class Product

**class** Product

/\*

Defines (immutable) products at sale in a vending machine.

JPF, FEUP, MFES, 2014/15.

\*/

**instance variables**

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

**public** name: **seq** **of** **char**;

**public** price: **nat1**;

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

**operations**

**public** Product : **seq** **of** **char** \* **nat1** ==> 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.

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

JPF, FEUP, MFES, 2014/15.

\*/

**types**

**public** BagOfCoins = MoneyUtils`BagOfCoins;

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

**instance variables**

/\* Note: variables declared public to facilitate observability by tests \*/

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

**public** products: **set** **of** Product;

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

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

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

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

-- Items observable by maintenance operator:

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

-- Items observable only by factory:

**public** unlockingCode : **seq** **of** **char**;

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

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

**inv** forall p1, p2 **in set** products & p1 <> p2 => p1.name <> p2.name;

**operations**

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

**public** VendingMachine: **seq** **of** **char** \* **set** **of** Product ==> VendingMachine

VendingMachine(code, prods) == (

unlockingCode := code;

products := prods

)

**pre** forall p **in set** prods & p.quantityInStock = 0

and forall p1, p2 **in set** prods & p1 <> p2 => p1.name <> p2.name;

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

**public** openDoor: **seq** **of** **char** ==> ()

openDoor(code) ==

if code = unlockingCode **then**

status := <OnMaintenance>

**pre** status = <Idle>;

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

loadCoins(coins) ==

stockOfCoins := coins

**pre** status = <OnMaintenance>;

**public** loadProducts: **map** Product **to** **nat** ==> ()

loadProducts(prods) ==

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

p.addToStock(prods(p))

**pre** status = <OnMaintenance>

and dom prods subset 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: Product ==> ()

selectProduct(prod) ==

let chg = MoneyUtils`extract3(stockOfCoins, amountInserted - prod.price) **in** (

prod.removeFromStock(1);

atomic (

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

amountInserted := 0;

changeToPickup := chg;

productToPickup := prod;

status := <Delivering>

)

)

**pre** status = <EnteringCoins>

and prod in **set** products

and prod.quantityInStock > 0

and amountInserted >= prod.price

and MoneyUtils`canExtract2(stockOfCoins, amountInserted - prod.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: () ==> Product

pickProduct() == (

let r = productToPickup **in** (

atomic (

productToPickup := nil;

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

);

return r

)

)

**pre** productToPickup <> nil;

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

cancel() ==

let chg = MoneyUtils`extract3(stockOfCoins, amountInserted) **in**

atomic (

amountInserted := 0;

changeToPickup := chg;

status := <Delivering>

)

**pre** status = <EnteringCoins>;

**end** VendingMachine

# 4. Model validation

## 4.1 Class MyTestCase

**class** MyTestCase

/\*

Supercalss 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 definitions for the vending machine.

Illustrates a scenario-based testing approach.

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

JPF, FEUP, MFES, 2014/15.

\*/

**operations**

/\*\*\*\*\* USAGE SCENARIOS \*\*\*\*\*\*/

-- Scenario: Normal purchase scenario in a vending machine.

-- Pre-conditions:

-- 1. The machine is initially idle. (initial system state)

-- 2. The machine has the product in stock. (initial system state)

-- 3. The buyer has enough coins. (input)

-- 4. If needed, the machine has coins in stock to give change. (initial system state)

-- Post-conditions:

-- 1. The buyer received the product. (output)

-- 2. The buyer received the change, if needed. (output)

-- 3. The stock of the product is updated in the machine. (final system state)

-- 4. The stock of coins is updated in the machine. (final system state)

-- 5. The machine is idle again. (final system state)

-- Steps:

-- 1. The buyer inserts the coins.

-- 2. The machine displays the money inserted (credit).

-- 3. The buyer selects the product.

-- 4. The machine delivers the product and the change, if needed.

-- 5. The buyer picks the product and the change, if existent

**public** PurchaseScn: VendingMachine \* Product \* MoneyUtils`BagOfCoins

==> [Product] \* MoneyUtils`BagOfCoins

PurchaseScn(vm, prod, coins) == (

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

**dcl** deliveredProd : [Product] := nil;

**dcl** change : 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) > prod.price **then** change := vm.pickChange());

return mk\_(deliveredProd, change)

)

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

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

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

and MoneyUtils`canExtract2(MoneyUtils`addAll(vm.stockOfCoins, coins),

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

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

deliveredProd = prod /\*1\*/

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

-- post-conditions not supported (old state of referenced object)

-- and vm.stockOfProducts(prod) = vm.stockOfProducts~(prod)-1 /\*3\*/

-- and vm.stockOfCoins = MoneyUtils`removeAll(change,

-- MoneyUtils`addAll(vm.stockOfCoins~, coins)) /\*4\*/

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

);

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

-- Pre-conditions:

-- 1. The machine is initially idle. (initial system state)

-- 2. The buyer has some coins to insert. (input)

-- Post-conditions:

-- 1. The buyer received back the amount inserted (same or equiv coins). (output)

-- 2. The amount of money is unchanged in the machine. (final system state)

-- 3. The machine is idle again. (final system state)

-- Steps:

-- 1. The buyer inserts the coins.

-- 2. The machine displays the money inserted (credit).

-- 3. The buyer cancels the operation.

-- 4. The machine returns back the coins inserted .

-- 5. The buyer picks the coins.

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

CancelScn(vm, coins) == (

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

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();

return vm.pickChange()

)

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

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

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

-- post-condition not supported (old state of referenced object)

-- MoneyUtils`sum(vm.stockOfCoins) = MoneyUtils`sum(vm.stockOfCoins~) /\*2\*/

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

-- Scenario: Normal scenario for loading (adding) products and

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

-- Pre-conditions:

-- 1. The machine is idle. (initial internal state)

-- 2. The operator knows the unlock code. (input)

-- 3. The machine accepts the products to load. (input)

-- Post-conditions:

-- 1. The items are added to the stock of products. (final internal state)

-- 2. The stock of coins is set to the intended one. (final internal state)

-- 3. The machine is idle again. (final internal state)

-- Steps:

-- 1. Unlock and open the machine door.

-- 2. Set the stock of coins and products, by any order.

-- 3. Close and lock the machine door.

**public** LoadStockScn: VendingMachine \* **seq** **of** **char** \* **map** Product **to** **nat** \* MoneyUtils`BagOfCoins ==> ()

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

(

vm.openDoor(code);

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

vm.closeDoor()

)

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

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

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

**post** -- not supported:

-- forall p in set dom prods &

-- p.quantityInStock = p.quantityInStock~ + prods(p) /\*1\*/

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

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** p1 : Product := new Product("Bolicao", 50);

**dcl** p2 : Product := new Product("Bongo", 70);

**dcl** code : **seq** **of** **char** := "xa1!";

**dcl** vm : VendingMachine := new VendingMachine(code, {p1, p2});

LoadStockScn(vm, code, {p1 |-> 1, p2 |-> 1}, { |-> });

let **mk\_**(-, change) = PurchaseScn(vm, p1, {20 |-> 2, 10 |-> 1})

**in** assertEqual({ |-> }, change);

let **mk\_**(-, change) = PurchaseScn(vm, p2, {20 |-> 4})

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

);

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

-- then enter coins and cancel.

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

testLoadAndCancel() == (

let p1 = new Product("Bolicao", 50),

code = "xa1!",

vm = new VendingMachine(code, {p1}),

coins = {20 |-> 2, 10 |-> 1}

**in** (

LoadStockScn(vm, code, {p1 |-> 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 p1 = new Product("Bolicao", 50),

code = "xa1!",

vm = new VendingMachine(code, {p1})

**in** (

LoadStockScn(vm, code, {p1 |-> 1}, { |-> });

vm.insertCoin(100);

vm.selectProduct(p1); -- breaks pre-condition

)

);

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

testProductOutOfStock() == (

let p1 = new Product("Bolicao", 50),

code = "xa1!",

vm = new VendingMachine(code, {p1})

**in** (

LoadStockScn(vm, code, {p1 |-> 0}, { |-> });

vm.insertCoin(50);

vm.selectProduct(p1); -- breaks pre-condition

)

);

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

testForgotToPickProduct() == (

let p1 = new Product("Bolicao", 50),

code = "xa1!",

vm = new VendingMachine(code, {p1})

**in** (

LoadStockScn(vm, code, {p1 |-> 1}, { |-> });

vm.insertCoin(50);

vm.selectProduct(p1);

-- forgot: vm.pickProduct();

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

)

);

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

testForgotToPickChange() == (

let p1 = new Product("Bolicao", 50),

code = "xa1!",

vm = new VendingMachine(code, {p1})

**in** (

LoadStockScn(vm, code, {p1 |-> 1}, { |-> });

vm.insertCoin(50);

vm.cancel();

-- forgot: vm.pickChange();

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

)

);

**end** TestVendingMachine

# 5. Model Verfication

# 6. Conclusions

# 7. References