Métodos Formais em Engenharia de Software

Ana Paiva

apaiva@fe.up.pt

Universidade do Porto
Faculdade de Engenharia
FEUP

Agenda

- VDMTools
- Características da linguagem VDM++
  - Classes; Variáveis de instância; Operações; Funções (polimórficas, de ordem superior, lambda, …); Tipos; Operadores; Expressões
  - Design-by-contact:
    - Definição de invariantes; pré e pós-condições
    - Ligação do VDM++ ao UML
- Exemplo da *Vending Machine*
- Consistência da especificação: obrigações de prova e teste
- Concorrência em VDM++

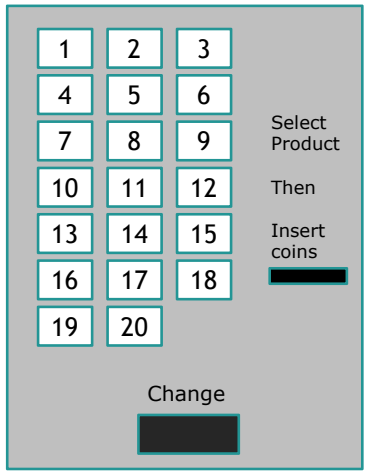
Vending Machine

- Build a VDM++ model of a vending machine for products like drinks, snacks, cookies, etc. This machine accepts coins of 0.05; 0.1; 0.2; 0.5 and 1 euro.
- There is a stock of coins inside the vending machine which is used to give the due change to clients who may need it.
- The vending machine has also a stock of products and each one has its price.
- The Product Vending Machine provides different services in two different possible states: in configuration or waiting for user interaction;

While in configuration: it is possible to update the stock of products and coins;
While waiting:

- The vending machine should show the products available to the clients.
- The clients should select the product they want and then insert euro coins to pay for it;
- The vending machine should give change every time it is possible (there are coins in stock for that purpose); otherwise it should give back all the money introduced by the client without selling the product.

Vending machine



a) Accepts coins of 0.05; 0.1; 0.2; 0.5 and 1 euro

b) Each of the 20 boxes can have at most 15 units.

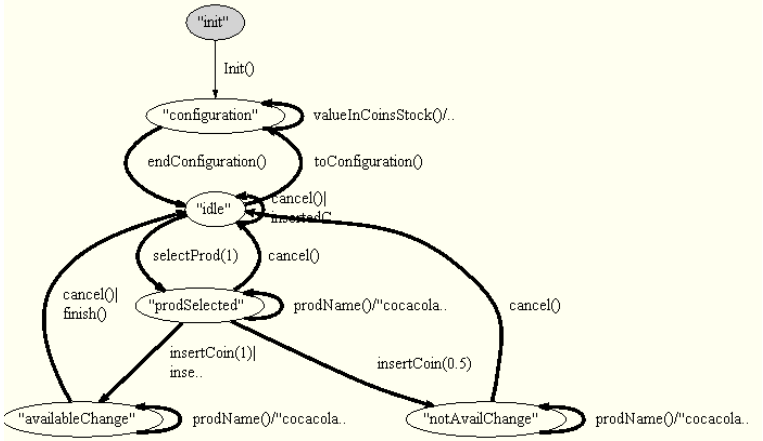
c) Each product has a price

d) different possible states: configuration, idle, init, prodSelected, availableChange, notAvailChange

e) Different operations available depending on the current state.

f) If there is no change, it is possible to cancel

Vending Machine



Vending Machine – constantes e tipos

```
values
  public Capacity: real = 15;

types
  public String = seq of char;
  public Coins = nat
    inv c == c in set {100, 50, 20, 10, 5};
  public Boxes = nat
    inv b == b in set {1,...,20};

  public Products :: name: String
                     quantity : nat1
                     price : nat1;

  public ProductsInBoxes = map Boxes to [Products]
    inv pb ==
      forall b in set dom pb & pb(b) = nil or
        pb(b)<>nil => pb(b).quantity <= Capacity;

  public State = <init> | <configuration> | <idle> | <prodSelected> |
                 <availableChange> |  <notAvailChange>;

instance variables
  public stockProd: ProductsInBoxes;
  public stockCoins: map Coins to nat;
  public stateMachine: State := <init>;
  public prodSelected: [Boxes] := nil;
  public insertedCoins: seq of Coins := [];
  public coinsTroco: seq of Coins := [];
```


Vending Machine – operações/funções

```
operations
Construtor
    public VendingMachine : () ==> VendingMachine
    VendingMachine () ==
    (
       stateMachine := <configuration>;
       stockProd := { |-> };
       stockCoins := { |-> };
    )
    pre stateMachine = <init>;
```

Abastece máquina com um produto numa certa posição. Substitui o que se encontrava antes na mesma posição. Pressupõe a máquina em configuração.

```
    public SetStockProducts (num: Boxes, name: String, price: nat1, quant: nat1) ==
       stockProd := stockProd ++ {num |-> mk_Products(name, quant, price)}
    pre stateMachine = <configuration> and
                       quant <= Capacity and price mod 5 = 0;
```


Vending Machine – operações/funções

Altera o *stock* de moedas. Pressupõe a máquina em configuração.

```
    public SetStockCoins(novoStockCoins: map Coins to nat) ==
       stockCoins := stockCoins ++ novoStockCoins
    pre stateMachine = <configuration>;
```

Consulta a quantidade em *stock* de um produto pelo número.

```
    public GetStockProducts(number: Boxes) res : nat1 ==
       return stockProd(number).quantity
    pre stateMachine = <configuration> and number in set dom stockProd;
```

Consulta o preço de um produto pelo número.

```
    public GetPriceProduct(number: Boxes) res : nat1 ==
       return stockProd(number).price
    pre stateMachine = <configuration> and number in set dom stockProd;
```

Fim de configuração

```
    public EndConfiguration() ==
      stateMachine := <idle>
    pre stateMachine = <configuration>;
```

Vending Machine – operações/funções

Selecciona um produto pelo número. Depois de seleccionar um produto pode inserir moedas.

```
public SelectProduct(number : Boxes) ==
  prodSelected := number
pre stateMachine = <idle> and number in set dom stockProd and
     stockProd(number) <> nil;
```

Para saber se ainda não inseriu o valor suficiente de moedas.

```
public InsertedCoinsValue() res : nat ==
(
   dcl sum:nat := 0;
   for all e in set inds insertedCoins do sum:=sum+insertedCoins(e); return sum;
)
pre prodSelected <> nil;
```

Insere uma moeda de um determinado valor. Deve estar um produto seleccionado, e não devem ter sido inseridas moedas de valor suficiente.

```
public InsertCoin(c: Coins) ==
  insertedCoins := insertedCoins ^ [c]
pre prodSelected <> nil and InsertedCoinsValue() < stockProd(prodSelected).price;
```


Vending Machine – operações/funções

Pergunta/observa o nome do produto seleccionado.

```
public GetNomeProdSel() res : String ==  return stockProd(prodSelected).name
pre prodSelected <> nil;
```

Calcula o valor das moedas que fazem parte do troco

```
public Sum(troco: seq of Coins) res : nat ==
 (     dcl sum:nat := 0;
       for all e in set inds troco do sum := sum + troco(e);
       return sum;
 );
```

Cancela a compra em curso

```
public Cancelar() ==
 ( prodSelected := nil;  insertedCoins := []; coinsTroco := []  )
pre prodSelected <> nil;
```

Simula utilizador a recolher o produto seleccionado

```
public RecolheProduto() == (
      if (stockProd(proSelected).quantity<>1) then
         stockProd(prodSelected).quantity := stockProd(prodSelected).quantity - 1 else
         stockProd := {prodSelected} <-: stockProd ;
      coinsTroco := [];
)
pre prodSelected in set dom stockProd;
```


Vending Machine – operações/funções

Qual o valor total das moedas em stock?

```
public SumMap(x:map Coins to nat) res:nat
      (dcl sum: nat := 0;
      for all e in set dom x do sum:=sum+e*x(e);
      return sum;)
```

Calcular o troco (moedas com valor igual a troco)

```
public calcularTroco () res: map Coins to nat
(  …
  exists m:map Coins to nat &
      SumMap(m) = InsertedCoinsValue()- stockProd(prodSelected).price
      and forall e in set dom m & e in set dom stockCoins and
                  stockCoins(e) >= m(e);
  …
);
```

Não executável!

Vending Machine – operações/funções

Se possível, dá troco ao cliente

```
public GiveChange() res: seq of Coins ==
(
   dcl sortedCoins: seq of Coins := [100,50,20,10,5];
   dcl troco : nat := InsertedCoinsValue() - stockProd(prodSelected).price;
   coinsTroco := [];
   while (sortedCoins <> []) do  (
     if (hd sortedCoins in set dom stockCoins and stockCoins(hd sortedCoins) > 0 and
        hd sortedCoins <= troco)
     then  (
        coinsTroco := coinsTroco ^ [hd sortedCoins];
        if (stockCoins(hd sortedCoins)=1) then stockCoins := {hd sortedCoins} <-: stockCoins
        else stockCoins(hd sortedCoins) := stockCoins(hd sortedCoins)-1;
        troco := troco - hd sortedCoins;
     )
     else sortedCoins := tl sortedCoins
   );
   if (troco <> 0) then   (
      for all e in set elems coinsTroco do stockCoins(e) := stockCoins(e)+1;
      coinsTroco := [];
   );
   return coinsTroco;
 )
 pre self.prodSelected <> nil and self.InsertedCoinsValue() > self.stockProd(self.prodSelected).price;
```