Interação e Concorrência

2ª e 3ª Fase

Universidade do Minho

Grupo 5

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

Introdução

Nestas duas ultimas fases do trabalho proposto pretende-se desenvolver uma *Vending Machine* que seja capacitada de simular alguns processos concorrentes à nossa máquina que foi ainda enriquecida com algumas funcionalidades, como por exemplo, um gestor inteligente de trocos, gestor de Stock, um fornecedor dinâmico de Produtos e ainda um Utilizador considerado como agente reativo. Por fim, criámos 7 propriedades válidas no nosso trabalho.

Grupo 5

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

2ª Fase

Nesta segunda fase do nosso trabalho, nós pegamos na variação dois que tínhamos desenvolvido na 1ª fase e acrescentamos o seguinte:

- O utilizador como agente reativo;
- Controlador responsável pela gestão de pagamentos;
- Carregamento dinâmico do Stock;

Para lidar com a gestão de trocos e de produtos foi necessário definir na parte das estruturas (sort) de lados os seguintes parâmetros:

Vários tipos de moedas que o nosso utilizador tem ao seu dispor para introduzir na máquina.

- Coin = struct five | ten | twenty | fifty

Os vários produtos que a nossa máquina pode retornar, ou seja, os produtos que o utilizador pode escolher.

- Product = struct Apple | Chocolate | Coffee;

Utilizámos uma estrutura de dados já definidos no mcrl2, o “Bag” que basicamente a cada elemento tem um número associado, neste caso temos um “Bag” de Produtos pois vai ser importante na parte de gestão de Stock, e o mesmo acontece para a parte da gestão de trocos em que temos um “Bag “, sendo que cada elemento (five,tem,twenty,fifty) tem um número associado.

- Stock = Bag(Product);
- Bank = Bag(Coin);

Usámos um par em que o primeiro elemento é um Produto e o segundo é um inteiro, ou seja, o número que existe desse dado produto no nosso “Bag”.

- PairProduct = struct pair(fst:Product, snd:Int);

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

Ações

```
act
        ins, acc, coin, ret :Coin;
        opt, pay, request, ready, ok, product_ready, deny, product_denied
:Product;
        supply, demand, refill:PairProduct;
        change, changeS;
```

Relativas ao tipo Coin temos quatro ações:

- **ins-** O utilizador insere uma moeda;
- **acc-** A máquina reconhece uma determinada moeda;
- **coin-** Ocorre através do sincronismo entre a ação de o utilizador inserir uma moeda e a máquina aceitar uma meda.
- **ret-** A máquina retorna uma determinada moeda.

Relativas ao tipo Product temos oito ações:

- **opt**- O utilizador opta por um dado produto.
- **pay-** A máquina verifica se existe dinheiro suficiente para pagar o produto.
- **request-** Ocorre através do sincronismo entre a ação de o utilizador optar por m dado produto e a máquina verificar se existe dinheiro para pagar esse produto, em caso afirmativo a máquina vai estar pronta retornar o produto.
- **ready-** A máquina diz que está pronta para disponibilizar o produto.
- **ok-** o nosso gestor de pagamentos confirma que existe dinheiro necessário para pagar o produto.
- **product_ready-** Ocorre através do sincronismo entre a ação de a máquina informar que está pronta a dar o produto e o gestor de pagamentos dizer que tem dinheiro suficiente, é disparada a ação de product_ready, ou seja, o produto está pronto a ser colocado no tabuleiro.
- **deny**- O nosso gestor de pagamentos ao verificar se existe dinheiro para pagar o produto, diz que não tem dinheiro suficiente.
- **deny**- A máquina não tem produtos para serem dados.
- **product_denied**- Ocorre através do sincronismo entre entre o deny do gestor de pagamentos e do deny da máquina já definidos anteriormente, basicamente esta ação serve para dizer que não é possível dar o produto em questão.

Relativo ao tipo Pair-Product temos 3 ações:

- **supply-** O nosso gestor de Stock pode adicionar produtos ao nosso Stock gerar, que é o Bag de Produtos, por exemplo podemos querer adicionar ao nosso Stock, Pair(Apple,5), ou seja, adicionámos 5 unidades de maçãs ao nosso Bag.
- **demand**- É pedido para repor produtos no Stock.
- **refill**- Ocorre através do sincronismo entre a ação do gestor do Stock adicionar produtos ao Stock, e a ação de ser pedido para se meter produtos no Stock, o refill coloca uma determinada quantidade de um produto no nosso Stock geral com todos os Produtos.
- **change e changeS**-– Ação de pedido de troco por parte do utilizador e de autorização de disponibilização de troco por parte do gestor de pagamentos.

Funções utilizadas:

map

troco: Nat # Bank -> List(Coin);

A função *troco* tem como parâmetros de entrada um número natural e um *Bank* (que contém todas as moedas presentes na máquina) e retorna uma lista de moedas.

Esta função tem como finalidade fazer o cálculo dos trocos dado o valor do saldo do Utilizador e o *Bag* (*bank*) que tem todas as moedas existentes na máquina.

sub: Nat # Coin -> Nat;

A função *sub* tem como parâmetros de entrada, um número natural e uma *coin* e retorna um número natural.

Pretende-se com esta função subtrair ao saldo do Utilizador o valor de uma *coin.*

remove: List(Coin) # Coin -> List(Coin);

A função *remove* recebe uma lista de moedas e uma *coin* e devolve uma lista de moedas.

Esta função remove uma determinada moeda a uma lista de moedas (servirá para devolver o troco ao utilizador. Será explicada mais à frente).

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

```
var
        total:Nat;
        bank:Bank;
        coin:Coin;
        lista:List(Coin);
```

As variáveis que nós utilizamos foram:

- total- É um natural que nos diz o dinheiro total que está na máquina.
- bank- É um "Bag" que contém as moedas existentes na máquina
- coin- Representa apenas uma moeda.
- lista- Representa uma lista de moedas.

Visto que temos um gestor de Trocos, é necessário ter atenção ao valor que temos na nossa máquina e a cada vez que o nosso gestor de Pagamentos faz uma ação em sincronismo com a máquina e dá sucesso é necessário retirar a moeda ou moedas do nosso bank de moedas e subtrair ao valor total que a máquina tinha, para isso é utilizada a função definida anteriormente (sub) que subtrai ao nosso total cada uma das moedas que são aceitáveis pela nossa máquina.

```
eqn
        (coin==fifty)  -> sub(total,coin) = Int2Nat(total-50);
        (coin==twenty) -> sub(total,coin) = Int2Nat(total-20);
        (coin==ten)    -> sub(total,coin) = Int2Nat(total-10);
        (coin==five)   -> sub(total,coin) = Int2Nat(total-5);
```

Caso a nossa lista seja vazia não é possível estar a retirar nenhuma moeda da lista.

lista == []) -> remove(lista,coin) = [];

Caso se remova a cabeça da nossa lista então o nosso resultado final vai ser a nossa cauda.

(coin == head(lista)) -> remove(lista,coin) = tail(lista);

Caso a nossa moeda a remover se encontre na cauda então é retornada uma nova lista com a cabeça e com o resto da cauda sem a moeda que se removeu.

(coin != head(lista)) -> remove(lista,coin) = [coin] ++ remove(tail(lista),coin);

Com a nossa função *troco* pretende-se permitir fazer uma gestão inteligente dos trocos. Por exemplo, para um valor de troco superior a 20 (*total*) é verificado se existe no banco (*bank*) a moeda de 50 (*twenty*) e se tal existir coloca-a na lista que servirá para retornar o troco e retira-a do banco. Ao saldo é descontado o valor da moeda escolhida. Em seguida repete o processo até o saldo chegar a zero. Não existindo no banco uma moeda de 20 o processo é iniciado com a maior moeda existente no banco e inferior ao total.

Esta função é chamada recursivamente até ser preenchida a lista com todas as moedas que servirão para devolver o troco ou até a máquina ficar sem moedas. Neste caso o Utilizador não recebe a totalidade do troco.

```
((total>=50) && ((bank * {fifty:1}) != {:})) -> troco(total, bank) = [fifty] ++
troco(Int2Nat(total-50),bank - {fifty:1});

      ((total>=50) && ((bank * {twenty:1}) != {:})) -> troco(total, bank) = [twenty] ++
troco(Int2Nat(total-20),bank - {twenty:1});

      ((total>=50) && ((bank * {ten:1}) != {:})) -> troco(total, bank) = [ten] ++
troco(Int2Nat(total-10),bank - {ten:1});

      ((total>=50) && ((bank * {five:1}) != {:})) -> troco(total, bank) = [five] ++
troco(Int2Nat(total-5),bank - {five:1});

      ((total>=50) && ((bank * {five:1,ten:1,twenty:1,fifty:1}) == {:})) -> troco(total, bank)
= [];


      ((total>=20) && (total<50) && ((bank * {twenty:1}) != {:})) -> troco(total, bank) =
[twenty] ++ troco(Int2Nat(total-20),bank - {twenty:1});

      ((total>=20) && (total<50) && ((bank * {ten:1}) != {:})) -> troco(total, bank) = [ten] ++
troco(Int2Nat(total-10),bank - {ten:1});

      ((total>=20) && (total<50) && ((bank * {five:1}) != {:})) -> troco(total, bank) = [five]
++ troco(Int2Nat(total-5),bank - {five:1});

      ((total>=20) && (total<50) && ((bank * {five:1,ten:1,twenty:1}) == {:})) -> troco(total,
bank) = [];

      ((total>=10) && (total<20) && ((bank * {ten:1}) != {:})) -> troco(total, bank) = [ten] ++ troco(Int2Nat(total-
10),bank - {ten:1});

      ((total>=10) && (total<20) && ((bank * {five:1}) != {:})) -> troco(total, bank) = [five] ++ troco(Int2Nat(total-
5),bank - {five:1});

      ((total>=10) && (total<20) && ((bank * {five:1,ten:1}) == {:})) -> troco(total, bank) = [];


      ((total>=5) && (total<10) && ((bank * {five:1}) != {:})) -> troco(total, bank) = [five] ++ troco(Int2Nat(total-
5),bank - {five:1});

      ((total>=5) && (total<10) && ((bank * {five:1}) == {:})) -> troco(total, bank) = [];


      (total==0) -> troco(total,bank) = [];
```


Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

Vending Machine

Para o nosso trabalho estar a fazer tudo aquilo que é pedido, foi necessário definir cinco Processos:

O processo User aceita várias ações que podem ser feitas pelo Uilizador, sendo que este pode introduzir moedas, escolher produtos e ainda pode pedir troco.

```
User =
                    sum c:Coin.(ins(c).User) +
                    sum p:Product.(opt(p).User) +
                    changeS.User;
```

O processo Supplier é responsável por adicionar produtos ao nosso Stock, isto é feito através da nossa ação definida anteriormente supply em que indica qual o produto que se quer introduzir e a respetiva quantidade.

```
Supplier =
                    supply(pair(Apple,5)).Supplier +
                    supply(pair(Chocolate,5)).Supplier +
                    supply(pair(Coffee,5)).Supplier;
```

O processo Returner está diretamente relacionado com o processo Payment, em que dada uma lista de moedas que formam o possível troco que vai ser dado ao utilizador, visto que uma lista é composta por várias moedas estas vão ser enviadas para o utilizador através da ação ret, sendo enviada uma a uma, até a nossa lista ficar vazia, tal como nos parâmetros do Returner temos uma lista de moedas o nosso banco de moedas e o valor total na máquina, em que vai ser retornada uma lista do possível troco.

```
Returner(total:Nat, bank:Bank, lista:List(Coin)) =
                    (lista == []) -> Payment(0,bank) +
                    (lista != []) -> sum c:Coin.((c in lista) ->
ret(c).Returner(sub(total,c),bank - {c:1},remove(lista,c)));
```

O gestor de pagamentos *Payment*, recebe o saldo do utilizador e a lista de moedas existentes no banco. Sempre que o utilizador introduz uma moeda, no processo *Payment*, através da ação *acc* é verificado se o saldo do utilizador ultrapassa o limite de 60 cêntimos e

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

nesse caso a máquina devolve a moeda (*ret*). Caso contrário é acrescentado ao saldo do utilizador o valor dessa moeda e à lista das moedas da máquina é acrescentada a moeda.

No caso da escolha de um produto pelo utilizador, o gestor de pagamentos verifica se o saldo é suficiente para disponibilizar esse produto. Em caso afirmativo é dado o *ok* para a disponibilização do produto e retirado o valor correspondente ao preço do produto ao saldo do utilizador. Caso contrário, a máquina não disponibiliza o produto. Finalmente, se for pedido o troco, é evocado o processo *Returner,* que irá devolver o troco.

```
Payment(total:Nat, bank:Bank) =

                acc(five).  ((total+5 > 60)  -> ret(five).Payment(total,bank) <>
Payment(Int2Nat(total+5),bank + {five:1})) +

                acc(ten).   ((total+10 > 60) -> ret(ten).Payment(total,bank) <>
Payment(Int2Nat(total+10),bank + {ten:1})) +

                acc(twenty).((total+20 > 60) -> ret(twenty).Payment(total,bank)
<> Payment(Int2Nat(total+20),bank + {twenty:1})) +

                acc(fifty). ((total+50 > 60) -> ret(fifty).Payment(total,bank) <>
Payment(Int2Nat(total+50),bank + {fifty:1})) +

                pay(Chocolate).((total >= 20) ->
ok(Chocolate).Payment(Int2Nat(total-20),bank) <>
deny(Chocolate).Payment(total,bank)) +

                pay(Coffee).   ((total >= 45) -> ok(Coffee).
Payment(Int2Nat(total-45),bank) <> deny(Coffee).Payment(total,bank)) +

                pay(Apple).    ((total >= 10) -> ok(Apple).
Payment(Int2Nat(total-10),bank) <> deny(Apple).Payment(total,bank)) +

                changeS.Returner(total,bank,troco(total,bank));
```

O processo Mach recebe o stock da nossa máquina e para e para cada produto verifica se ele consta no nosso Stock, no caso de existir e o utilizador desejar esse produto, através de ações de sincronização com o Processo Payment é verificado se o saldo que existe na máquina é suficiente para pagar esse produto, em caso afirmativo então o produto está pronto a ser colocado no tabuleiro e é retirado do nosso Stock geral, em caso negativo, é recusado usa-se a ação denied, no caso de um dado Produto já não existir no nosso Stock o nosso gestor de Stock faz um Supplier desse determinado produto através da ação demand.

Vending Machine

```
Mach(stock:Stock) =
                ( (stock * {Chocolate:1}) != {:} ) ->
                        pay(Chocolate).(
                                ready(Chocolate).Mach(stock - {Chocolate:1})
                                        +
                                deny(Chocolate).Mach(stock)) +
                ( (stock * {Coffee:1}) != {:} ) ->
                        pay(Coffee).(ready(Coffee).Mach(stock - {Coffee:1})
                                +
                        deny(Coffee).Mach(stock)) +


                ( (stock * {Apple:1}) != {:} ) ->
                        pay(Apple).
                                (ready(Apple).Mach(stock - {Apple:1})
                                        +
                                deny(Apple).Mach(stock)) +
                ( (stock * {Apple:1}) == {:} ) ->
demand(pair(Apple,5)).Mach(stock + {Apple:5}) +
                ( (stock * {Coffee:1}) == {:} ) ->
demand(pair(Chocolate,5)).Mach(stock + {Chocolate:5}) +
                ( (stock * {Chocolate:1}) == {:} ) ->
demand(pair(Coffee,5)).Mach(stock + {Coffee:5});
```

Grupo 5

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

Por fim, declarámos a inicialização do nosso programa da seguinte forma:

```
init
        allow(
                { coin, change, ret, refill, request, product_ready,
product_denied },
                comm({ ins|acc -> coin,
                    changeS|changeS -> change,
                    demand|supply -> refill,
                    opt|pay|pay -> request,
                    ok|ready -> product_ready,
                    deny|deny -> product_denied },
                User || Mach({Apple:5, Chocolate:5, Coffee:5}) || Supplier ||
Payment(0,{five:10, ten:10, twenty:10, fifty:10})
        ));
```

Declarámos que as ações permitidas são: coi, change, ret, refill, request, product_ready e product_denied.

Também definimos as ações sincronizadas que são usadas para as comunicações entre os vários processos, que são: coin, change, refill, request, product_ready, product_denied.

No final, inicializámos os processos paralelos User, Mach com um Stock inicial de produtos, Supplier, Payment com o total a 0 e um Banco com 10 moedas de cada tipo.

3ª Fase

Nesta fase, usando a lógica de *Hennessy-Milner*, criámos propriedades para a nossa vending machine.

1ª propriedade:

```
<coin(fifty).coin(fifty).ret(fifty)> true
```

Com esta propriedade, provámos que sempre que inserimos duas moedas de cêntimos haverá sempre o retorno de uma moeda de cêntimos.

2ª propriedade:

```
<true*.request(Coffee).product_ready(Coffee).request(Coffee).product_ready(Coffee)> false
```

Com esta propriedade, prova-se que para qualquer ação, nunca podemos requisitar dois cafés à máquina.

3ª propriedade:

```
<coin(fifty).coin(ten).((request(Coffee).product_ready(Coffee)) +
(request(Chocolate).product_ready(Chocolate)))>true
```

Nesta propriedade, prova-se que após inserir 60 cêntimos pode-se pedir ou um chocolate, ou um café.

4ª propriedade:

```
forall p:Product . <exists c:Coin . coin(c).request(p).product_ready(p)> val(true)
```

Aqui, verifica-se que para todo o Produto p, existe uma moeda c tal que após a inserção desta moeda pode-se sempre pedir um produto à máquina.

Grupo 5

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

5ª propriedade:

```
exists c1,c2:Coin .
<coin(c1).coin(c2).request(Coffee).product_ready(Coffee).request(Apple).product_ready(Apple)> val(true)
```

Aqui, existe duas moedas c1 e c2 tal que após a inserção das duas pedir um café e de seguida uma maçã é verdade.

6ª propriedade:

```
exists c1:Coin . <coin(c1).request(Apple).product_denied(Apple)> val(c1==five)
```

Existe uma moeda c1 tal que após a introduzir, pedir uma maçã e este pedido ser rejeitado, é porque c1 é uma moeda de 5 cêntimos.

7ª propriedade:

```
forall stock:Stock . val((stock * {Apple:1}) == {:}) => [request(Apple)] false
```

Para todo o stock do tipo Stock (Bag(Coin)), caso o stock não contenha maçãs, pedir uma maçã à máquina é falso.

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913

Conclusão

No fim de bastante trabalho e pesquisa realizada, pensámos que o trabalho está bem implementado e nas expectativas do grupo.

Devido à complexidade do trabalho, não é possível gerar o LTS, e também provar algumas propriedades pois existem bastantes ações parametrizadas e também vários processos em concorrências em que alguns também são parametrizado.

Grupo 5

Carlos Pereira – 71899
Carlos Leite – 70743
Carlos Martins – 68913