TPC-1

Gabriel Costa, MEFIS
Universidade do Minho, Portugal
Email: pg53828@uminho.pt

Exercício 1

1.1)

a) c.(a.0||b.0)

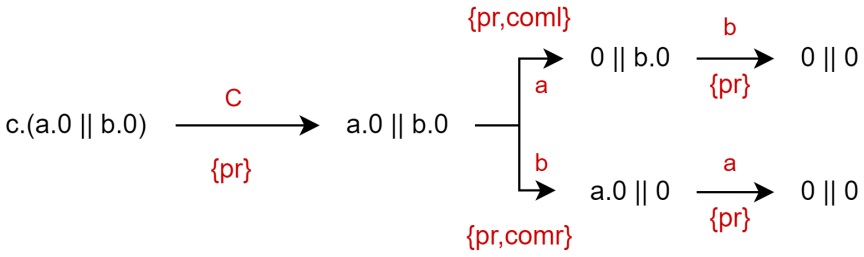
Este primeiro processo (a) comunica pelo canal c e, em seguida, realiza em paralelo os processos a.0 e b.0, onde é feita uma comunicação pelo canal 'a' no primeiro processo e pelo canal 'b' do segundo processo, terminando cada um deles depois da comunicação pelo respetivo canal.

b) rec X.(a.X + a.b.X)

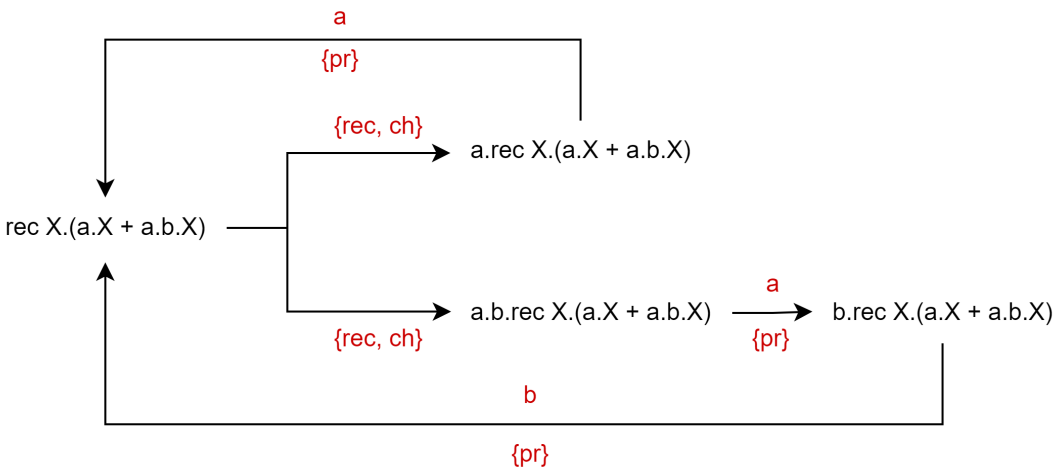
Neste segundo processo, é feita uma escolha entre o processo a.X e o processo a.b.X, sendo que cada um depois de comunicar pelos respetivos canais, volta ao estado inicial de escolher entre ambos os processos, pelo que este é interminável.

1.2)

a)



b)



Exercício 2

2.1)

O processo I está responsável pela inicialização das tarefas, começando as tarefas por ordem crescente,uma vez que só envia informação pelo canal st_i caso já tenha enviado informação pelo canal st_{i-1}. O processo S funciona de forma semelhante, com a diferença que é recursivo, e só recomeça a enviar informação pelos canais st caso todas as tarefas já tenham sido terminadas. Por fim, o processo P_i, depois de receber informação pelo canal st_i é independente de qualquer informação externa, não existindo nada que impeça que este termine a sua tarefa antes de outro processo qualquer. Quando colocados em paralelo estes processos implementam o processo P, que corresponde exatamente ao sistema acima descrito.

2.2)

$$I = \text{rec } X.\overline{st_1}.end.end.end.end.X$$
$$P_1 = \text{rec } Y_1.st_1.\overline{st_2}.a_1.b_1.\overline{end}.Y_1$$
$$P_2 = \text{rec } Y_2.st_2.\overline{st_3}.a_2.b_2.\overline{end}.Y_2$$
$$P_3 = \text{rec } Y_3.st_3.\overline{st_4}.a_3.b_3.\overline{end}.Y_3$$
$$P_4 = \text{rec } Y_4.st_4.a_4.b_4.\overline{end}.Y_4$$

Nesta nova forma, o *"schedular"* central foi removido, estando cada processo responsável por começar o processo seguinte (caso este exista). Desta forma, os processos começam sempre por ordem crescente, mantendo as caraterísticas do sistema inicial. Para permitir que este recomecem por ordem crescente ao mesmo tempo que possam acabar por qualquer ordem, o processo I for alterado, tendo agora a função de enviar o sinal de *start* ao processo P_1, esperando em seguida que todos os processos acabem antes de reenviar o sinal de *start* ao processo P_1, sendo esta ação repetida infinitamente.

2.3)

Tanto a versão do enunciado como a versão exposta em 2.2) do sistema pode ser escrita usando a linguagem mCRL2. Desta forma, estes sistemas foram escritos nesta mesma linguagem, estando os códigos das duas versões expostas nas seguintes figuras:

```
act
  inst1, inst2, inst3, inst4, outst1, outst2,outst3,outst4, inend, outend,
a1,a2,a3,a4,b1,b2,b3,b4,act1,act2,act3,act4,finished;
proc
  I = outst1.outst2.outst3.outst4;
  S = inend.inend.inend.inend.outst1.outst2.outst3.outst4.S;
  P1 = inst1.a1.b1.outend.P1;
  P2 = inst2.a2.b2.outend.P2;
  P3 = inst3.a3.b3.outend.P3;
  P4 = inst4.a4.b4.outend.P4;
init
  allow(
    { finished, act1, act2,act3,act4,a1,a2,a3,a4,b1,b2,b3,b4},
    comm(
      { inst1|outst1 -> act1, inst2|outst2 -> act2, inst3|outst3 -> act3 ,inst4|outst4 ->
act4 ,inend|outend -> finished},
      I || S || P1 || P2 || P3 || P4
  ) ) ;
```

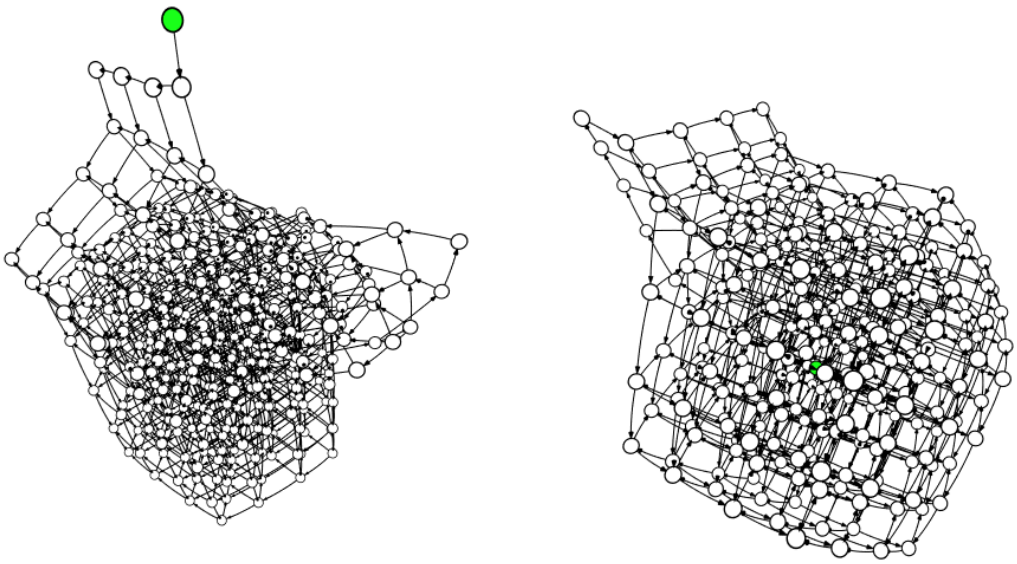
Figura 1: Sistema original

```
act
  inst1, inst2, inst3, inst4, outst1, outst2,outst3,outst4, inend, outend,
a1,a2,a3,a4,b1,b2,b3,b4,act1,act2,act3,act4,finished;
proc
  I = outst1.inend.inend.inend.inend.I;
  P1 = inst1.outst2.a1.b1.outend.P1;
  P2 = inst2.outst3.a2.b2.outend.P2;
  P3 = inst3.outst4.a3.b3.outend.P3;
  P4 = inst4.a4.b4.outend.P4;
init
  allow(
    { finished, act1, act2,act3,act4,a1,b1,a2,b2,a3,b3,a4,b4},
    comm(
      { inst1|outst1 -> act1, inst2|outst2 -> act2, inst3|outst3 -> act3 ,inst4|outst4 ->
act4 ,inend|outend -> finished},
      I || P1 || P2 || P3 || P4
  ) ) ;
```

Figura 2: Sistema com as alterações da alínea 2.2)

Recorrendo ainda a ferramentas gráficas da linguagem, podemos ainda verificar o número de estados criados para cada um dos sistemas, verificando que o primeiro apresenta 380 estados, enquanto que o sistema sem o *schedular* central diminui este número para 277, o que nos permite ter uma ideia quantitativa da simplificação do sistema inicial.

Podemos ainda representar os nossos sistemas graficamente através da IDE do mCRL2, sendo o espaço de estados dos dois sistemas os seguintes:



(a) Sistema original

(b) Sistema com as alterações da alínea 2.2)

Depois desta primeira análise, o sistema da alínea 2.2) foi analisado em maior detalhe, tendo, para esse efeito, sido realizados alguns testes sobre o sistema.

Primeiramente foi realizado um teste de "*deadlock*", onde foram procuradas *deadlocks* no sistema. Para este efeito por escrita a seguinte expressão num ficheiro mcf (extensão da linguagem mCRL2):

```
[ true* ] < true > true
```

Nesta expressão, [true*] representa todas as sequência finitas de ações começadas no estado inicial, enquanto que true representa a existência de um estado qualquer. O valor lógico true no fim indica o valor lógico que dizemos que esta expressão tem. Transformando este ficheiro "mcf"num ficheiro booleano, se o valor lógico associado for igual ao que foi colocado na expressão então obteremos um "true"no

stdout, obtendo "false"no caso contrário. Assim, esta expressão afirma que todas as sequências possíveis conseguem chegar a uma outra qualquer, existindo um loop, indicando a inexistência de *deadlocks*.

Compilando esta expressão obtemos o "true"no stdout, verificando a veracidade da expressão, o que indica a inexistência de *deadlocks* no sistema.

```
C:\Users\gabri\OneDrive\Desktop\Uni\4o_ano\2o_semestre\CiberPhi\TPC1\melhoria>lps2pbes melhoria.lps -f deadlock.mcf deadlock.pbes

C:\Users\gabri\OneDrive\Desktop\Uni\4o_ano\2o_semestre\CiberPhi\TPC1\melhoria>pbes2bool deadlock.pbes
true
```

Para além da procura por *deadlocks*, foi também verificada a possibilidade das expressões tarefas serem começadas numa ordem incorreta, tendo para esse efeito sido escrita a seguinte expressão:

```
[!act1*.act2]false && [!act2*.act3]false && [!act3*.act4]false
```

Compilando esta expressão obtermos o valor lógico verdadeiro, o que indica que ações nunca ocorrem por ordem errada.

```
C:\Users\gabri\OneDrive\Desktop\Uni\4o_ano\2o_semestre\CiberPhi\TPC1\melhoria>lps2pbes melhoria.lps -f wrong_order.mcf wrong_order.pbe
s

C:\Users\gabri\OneDrive\Desktop\Uni\4o_ano\2o_semestre\CiberPhi\TPC1\melhoria>pbes2bool wrong_order.pbes
false
```

Por fim foi também verificado se a seguir a uma sequência de 4 "finished"(que indica que uma ação acabou), era realizado necessariamente o primeiro processo (P1). Para isso foi escrita esta última expressão,

```
[ true* . finished . finished . finished . finished . !act1 ] false
```

que compilando deve como resultado o valor lógico "true", indicando assim que sempre que todos os processos terminam, a próxima ação é necessariamente o começo do primeiro processo.

```
C:\Users\gabri\OneDrive\Desktop\Uni\4o_ano\2o_semestre\CiberPhi\TPC1\melhoria>lps2pbes melhoria.lps -m -f end_condition.mcf end_condit
ion.pbes

C:\Users\gabri\OneDrive\Desktop\Uni\4o_ano\2o_semestre\CiberPhi\TPC1\melhoria>pbes2bool end_condition.pbes
true
```

Limitações e melhorias

Uma das limitações que o sistema originais tinha era a escalabilidade, sendo que a introdução ou a remoção de processos iria implicar uma alteração do *schedular* central, sendo necessário parar todo o sistema para haver uma manutenção. Com o sistema descrito em 2.2) isto já não aconteceria, uma vez que adicionar um processo apenas implicaria parar o último processo, introduzindo neste a parte de inicialização do novo processo.

No entanto, ambos os sistemas apresentam na sua composição um risco de todo o sistema parar caso um dos processos falhe. No primeiro caso, se o processo S falhar, todo o sistema irá parar, uma vez que não há nenhum processo a inicializar os restantes processos P_i. Já no segundo sistema, caso o processo I falhe, o processo P_1 nunca irá iniciar, e consequentemente nenhum dos restantes processos irá também funcionar.

Da mesma maneira, ambos os sistemas apresentam o risco de ficar parados caso algum dos processos P_i falhe, uma vez que nunca irão mandar mensagem pelo canal "*end*", o irá fazer com que os processos S e I fiquem á espera durante tempo infinito por uma mensagem neste canal, não recomeçando os processos.

Uma forma de prevenir que isto aconteça é a introdução de *clocks*, que caso um dos processos não seja concluído num dado tempo esperado, este é dado como falhado, agindo o sistema em conformidade (recomeçando todos os processos por exemplo).

Concluindo, o segundo sistema oferece uma melhor escalabilidade e flexibilidade em comparação com o sistema original. No entanto ambos possuem pontos de falha que levam à falha total dos mesmos, sendo que a introdução de mecanismos como *clocks* podem levar ao aumento da capacidade de recuperação e resiliência a falhas.