Laboratório 2: Análise de Processos no CWB-NC

Luís Soares Barbosa

Sumário

Este trabalho laboratorial em torno da utilização do CWB-NC, *dá seguimento ao anterior mas incidindo, agora, das facilidades disponíveis para análise de processos e técnicas associadas. Tal como na sessão anterior, o texto que se segue deverá ser lido com o* CWB-NC *activado de forma a poder responder às questões e exercícios propostos. As suas respostas deverão ser dadas de forma clara, sucinta e manuscrita. Complementarmente poderá anexar os* printouts *relevantes.*

Disciplina: Métodos de Programação IV (2005-06)
Docente *Luís Soares Barbosa,* Departamento de Informática, Universidade do Minho

---

1 Sequências Observáveis

Para além da simulação passo-a-passo, o CWB-NC oferece diversas facilidades que permitem explorar o comportamento de um processo sem ter necessidade de parar em todas as transições. Uma delas é activada pelo o comando `vs` que calcula as sequências visíveis de acções (*i.e.*, que não incluem o τ) de um comprimento especificado. Experimente o exemplo seguinte de um processo que recebe sinais na porta `in` e os re-envia na porta `out`, até que a recepção de um `abort` o desactiva. Para tornar o exemplo mais interessante, vamos permitir que o processo possa escolher internamente entre a realização de um `abort` ou a continuação com a sequência de `in`s e `out`s.

```
Command: agent A = tau.in.'out.A + tau.abort.0;

Command: vs(1,A);
=== abort ===>
=== in ===>

Command: vs(2,A);
=== in 'out ===>

Command: vs(3,A);
=== in 'out abort ===>
=== in 'out in ===>
```

Como é evidente neste exemplo, o comando `vs` não regista as acções não observáveis. De acordo com as convenções de CCS, a dupla seta (`==>`) é usada para referir transições observáveis.

Exercício 1

Neste exemplo, o processo `A` nunca pode realizar `abort` entre `in` e `out`. Modifique a definição de `A` de modo a tornear esta situação e confirme a correcção da sua proposta usando o comando `vs`.

O comando `vs` não fornece qualquer informação sobre as derivações que resultam das transições calculadas. Para isso poderá recorrer, alternativamente, ao comando comando `obs` (experimente-o neste exemplo!).

2 Análise de Alguns Exemplos

2.1 Acesso a Recursos

Vamos agora considerar uma família de problemas relacionados com a gestão de recursos partilhados por vários utilizadores. Os recursos podem ser memória, processadores, largura de banda, etc. Os utilizadores serão programas, máquinas, protocolos, etc. O importante é definir correctamente protocolos de acesso.

A situação mais simples é aquela em que o gestor `M` dá acesso ao recurso através da acção `'g`, e existe apenas um utilizador no sistema (que, consequentemente, obtém acesso sempre que deseja). Não existe qualquer competição: o sistema funciona como um bar com um único cliente e um empregado que lhe fornece cervejas sempre que solicitado!

Exercício 2

Verifique esse comportamento definindo

```
agent M   = 'g.M;
agent U   = g.t.U;
agent S   = (M|U)\g;
```

e usando o comando `vs`.

Levanta-se, agora, um problema interesante: qual o comprimento das sequências observáveis que temos de inpeccionar com o comando `vs` para ter a certeza de que o processo `S` se comporta da forma esperada? A questão pode parecer trivial dada a simplicidade do processo em causa. Mas toca um problema fundamental. Suponha que `S` era qualquer coisa mais complicada. Como poderíamos saber que ele permitia sempre a realização de acções `t`? Podemos verificar o que sucede definindo um sistema similar mas no qual o funcionário do bar apresenta um comportamento estranho: a qualquer altura pode fechar o bar e ir embora:

```
agent M2  = 'g.M2 + tau.0;
agent U   = g.t.U;
agent S2  = (M2|U)\g;
```

Experimente determinar as sequências de acções observáveis para este novo processo. Verificará que são idênticas às calculadas para `S`! Quer isto dizer que por mais longas que sejam as sequências observáveis consideradas, o comportamento dos dois processos é indistinguível. Mas, certamente, o leitor não desejaria ser cliente deste segundo bar...

Concluimos, pois, que testar apenas as sequências de acções observáveis não nos garante a ausência de comportamentos indesejáveis. Alternativamente podemos simular o comportamento em causa. Ao fazê-lo para o processo `S2` rapidamente o CWB-NC nos informa

```
**Deadlock**
```

querendo assim dizer que foi alcançado um estado a partir do qual não existem mais transições disponíveis. Uma vez que o objectivo inicial era construir um processo perpétuo, um estado de *deadlock* constitui um erro que distingue `S2` de `S`. Nem sempre, porém, descobrimos estas situações tão rapidamente. Em geral, o espaço de estados do processo em análise é tão grande que a hipótese de usar simulação interactiva nem se coloca como uma alternativa séria.

Por esta razão o CWB-NC fornece comandos de análise mais poderosos. Um deles é o comando `deadlocks` (ou, abreviadamente, `fd`) que determina todas as situações em que não existem evoluções possíveis. Experimente-o sobre os dois processos em mãos:

```
Command:  fd S;

No such agents.

Command:  fd S2;
--- tau ---> (0|U)\g
```

Desta forma distinguimos, com um simples teste, os dois processos. Mais, ainda, são-nos indicadas as transições que conduziram a situações de *deadlock* — uma informação que pode ser muito valiosa no projecto de sistemas concorrentes.

Outro tipo de análise que o CWB-NC disponibiliza é o comando `eq` que testa a equivalência observacional entre dois processos. Recorde que dois processos *observacionalmente equivalentes* apresentam as mesmas capacidades de realizar acções observáveis. Assim, sempre que um deles pode realizar uma determinada acção, o outro pode fazer o mesmo e atingir um estado observacionalmente equivalente ao estado atingido pelo primeiro processo. Para testar esta possibilidade de análise faça

```
Command:  agent Sp = t.Sp;

Command:  eq(S,Sp);
true
Command:  eq(S2,Sp);
false
```

A equivalência encontrada entre `S` e `Sp` não deixa de ser um pouco inesperada. De facto, o processo `Sp` tem apenas um estado (ele próprio), pelo que a equivalência significa que todos os estados em `S` têm o mesmo comportamento observável que `Sp`, *i.e.*, a partir de qualquer deles podemos observar a realização uma acção `t`.

2.2 Controlo de Recursos Partilhados

Vamos, agora, voltar ao problema dos gestores de recursos, supondo que estes são partilhados por mais do que um processo utilizador. Sejam, pois, os processos seguintes,

```
agent M3   = 'f.M3 + 'g.M3;
agent U1  = f.t.U1;
agent U2  = g.u.U2;
agent S3  = (U1|U2|M3)\{f,g};
```

Exercício 3

Utilizando os comandos `vs`, `fd`, `sim` e `eq`, analise o comportamento de `S3` e compare-o com o de `S` e `S2`. Será capaz de encontrar uma especificação mais simples que seja equivalente a `S3`?

Uma facilidade do CWB-NC é a possibilidade de parametrizar processos por outros processos [1]. Por exemplo, podemos definir o processo `PS` que modela um sistema de controlador e utilizadores de um recurso, parametrizado relativamente à especificação do controlador.

```
agent PS(X)  = (U1|U2|X)\{f,g};
```

Podemos, agora, utilizar `PS(M3)`, em vez de `S3`, ou instancia-lo com outras especificações.

Exercício 4

Defina os seguintes controladores:

```
agent N1   = 'f.N1;
agent N2   = tau.'f.N2 + tau.'g.N2;
agent N3   = 'f.'g.N3;
```

Explore o comportamento de `PS(N1)`, `PS(N2)` e `PS(N3)`. É capaz de conjecturar quais as sequências observáveis em cada um dos casos? Pode algum deles entrar em *deadlock*? Quantos estados possuem? É algum deles equivalente a `PS(M3)`?

Se resolveu correctamente o exercício anterior, poderá ter ficado surpreendido com o facto de `PS(N2)` e `PS(M3)`, apesar de exibirem as mesmas sequências observáveis e não apresentarem *deadlocks*, não serem equivalentes. Uma pequena experiência com o simulador revelará uma diferenga subtil entre ambos. De facto, em `PS(N2)` é possível, através de uma sequência de τ, chegar a um estado onde a única acção possível é `t`. Por seu lado, em `PS(M3)` sempre que `t` é oferecida, existe igualmente a possibilidade de realizar `u` (eventualmente precedido por `tau`). Dito de outro modo, `PS(N2)` pode *recusar* `u`, enquanto `PS(M3)` nunca recusa nem `u` nem `t`.

Exercício 5

Simule estes dois processos até encontrar o estado de recusa no primeiro e se convencer que tal não existe no segundo.

Este tipo de análise das "recusas" pode ser importante num ambiente que nunca permita, por exemplo, a realização de `t`. O modo mais imediato de representar esse ambiente é através de uma restrição. Analise o que se passa com `PS(N2)\t` e `PS(M3)\t`.

Exercício 6

Um sistema contém dois utilizadores e dois recursos (a que chamaremos café e açúcar), cujas portas de acesso são, respectivamente, `c` e `a`. Cada utilizador tem necessidade de aceder a ambos os recursos, primeiro ao café e depois ao açúcar. Posteriormente, cada um realiza uma acção distinta, `u` e `v`, conforme os casos, completando o seu ciclo de operação. Formalmente, $U \triangleq c.a.u.U$ e $V \triangleq c.a.v.V$. O controlador dá sempre permissão de acesso a qualquer recurso que a solicitar. Note-se, por fim, que, do ponto de vista do ambiente do sistema, apenas as acções u e v são observáveis.

Um dia pensou-se em substituir o controlador por um outro mais barato, com a particularidade de garantir acesso aos recursos de forma ordenada, *i.e.*, após um acesso ao café, insiste em oferecer um acesso ao açúcar, antes de voltar a possibilitar o acesso ao

[1] Repare que o sentido em que aqui é usada a expressão *processo parametrizado* é distinto do significado que à mesma expressão temos associado nas aulas teóricas.

café. O senhor X argumentou a favor desta substituição: *Uma vez que o controlador oferece acesso aos dois recursos pela mesma ordem em que estes são pedidos pelos utilizadores, tudo correrá bem*. No entanto, a senhora Y duvidou: *Suponha que um utilizador obteve café. Não será o sistema conduzido a um estado em que este utilizador vai ter de terminar a sua tarefa antes do segundo utilizador poder iniciar a sua? Não será isso critíco num ambiente que insista em que o segundo utilizador avance primeiro?*. Ajude X e Y a resolver esta dúvida com o auxílio do CWB-NC.

Suponha, agora, que o comportamento do utilizador V se alterou: deseja aceder ao açúcar antes de aceder ao café. Sempre optimista o senhor X argumentou: *Não há qualquer problema. Suponha que o ambiente requer uma acção v. Então tudo o que se tem de fazer é aguardar que U aceda ao café. Quando U tem café (mas não açúcar!), V pode aceder ao açúcar, depois ao café e, finalmente, realizar v.* Será mesmo assim?

2.3 Semáforos e Regiões Críticas

Um modelo mais sofisticado de controladores de recursos são ao *regiões critícas*, *i.e.*, recursos partilhados que não podem ser acedidos por mais de utilizador ao mesmo tempo. Nas aulas teóricas já viu como a exclusão mútua podia ser garantida por recurso a semáforos. Considere a seguinte definição duma situação deste tipo, no CWB-NC. Note que a definição do processo `Sem` está parametrizada por duas acções (o tipo de um parâmetro é determinado pelo primeiro caracter do seu identificador). A região crítica de cada processo corresponde às acções realizadas entre `p` e `v`.

```
agent  Sem(p,v) = 'p . 'v . Sem(p,v);
agent  U1       = p.a.b.v.U1;
agent  U2       = p.c.d.v.U2;

agent  S        = (U1 | U2 | Sem(p,v))\{p,v};
```

Exercício 7

Verifique no CWB-NC o comportamento deste sistema e descreva-o. Será capaz de encontrar uma definição mais simples?

Considere, agora, o caso em que existem dois recursos, protegidos por dois semáforos, e dois utilizadores. Cada utilizador necessita de aceder a ambos os recursos para completar a sua tarefa. Assim,

```
agent  Sem(p,v) = 'p . 'v . Sem(p,v);
agent  U1       = p1.p2.a.b.v2.v1.U1;
agent  U2       = p1.p2.c.d.v2.v1.U2;

agent  T        = (U1 | U2 | Sem(p1,v1) | Sem(p2,v2))\{p1,v1,p2,v2};
```

Exercício 8

Se trocar a ordem pela qual os utilizadores tentam aceder aos recursos, fazendo, por exemplo

```
agent  U2       = p2.p1.c.d.v1.v2.U2;
```

chegará a uma situação de *deadlock*. Porquê?

Esta é a "verdadeira" situação de *deadlock*, bem conhecida nos sistemas operativos e na programação paralela [2]. Existem várias estratégias para controlar este tipo de situações indesejáveis. Por exemplo, um

[2] Repare que na terminologia do CWB-NC, *deadlock* significa qualquer estado sem acções observáveis, o que não corresponde ao significado standard do termo.

deadlock pode ser interrompido (forçando os utilizadores a abandonarem os recursos permaturamente) ou evitado (por exemplo, insistindo em que os utilizadores acedam aos recursos críticos por uma ordem determinada).

Exercício 9

Um utilizador é dito *sociável* se, sempre que pode realizar uma operação de acesso a um recurso, tem igualmente a possibilidade de libertar todos os recursos que já tinha reservado. Por exemplo, na presença de dois recursos um utilizador *sociável* será definido por

```
agent  US       = p1.(p2.a.b.v2.v1.US + v1.US);
```

Considere, de novo, o sistema do exercício anterior em que dois utilizadores acedem a recursos por uma ordem diferente. Verificar-se-ia um *deadlock* se ambos os utilizadores fossem *sociáveis*? E se um o fosse e o outro não? Considere, agora, um sistema com dois recursos e três utilizadores. Existem apenas duas maneiras de ordenar o acesso aos dois recursos. Logo, com três utilizadores, pelo menos dois deles têm de ordenar os seus acessos da mesma forma. Qual dos três utilizadores deverá ser *sociável* de forma a evitar situações de *deadlock*?

Quando apenas estamos interessados em analisar situações de *deadlock* não temos necessidade de representar explicitamente as acções `a`, `b`, `c` e `d`. De facto, como o que nos preocupa é garantir que as actividades dentro de um região crítica não podem ser interrompidas, é suficiente substitui-las por uma sinalização de que o processo em causa teve sucesso no acesso a essa região. Assim,

```
agent  U1       = p1.p2.suc1.v1.v2.U1;
agent  U2       = p1.p2.suc2.v1.v2.U2;
```

Exercício 10

Em quanto diminuiu o espaço de estados do sistema composto? Pode não ser muito neste (pequeno) exemplo, mas tornar-se-á relevante em casos reais. Que aconteceria se prescindissemos completamente das sinalizações de sucesso? Seria uma boa ideia ou não?

Considere, agora, o sistema seguinte com três utilizadores:

```
agent  U1       = p1.p2.suc1.v2.v1.U1;
agent  U2       = p2.p3.suc2.v3.v1.U2;
agent  U3       = p3 . (p2.suc3.v2.v3.U3 + p1.suc3.v1.v3.U3);

agent  R        = (U1 | U2 | U3 | Sem(p1,v1) | Sem(p2,v2) | Sem(p3,v3))\L;
set    L        = {p1,p2,p3,v1,v2,v3};
```

Mesmo quando não existem situações de *deadlock*, podem verificar-se *deadlocks* parciais, *i.e.*, envolvendo apenas um subconjunto dos processos considerados. O CWB-NC não detecta essa situação através do comando `fd` (porquê?). Podemos, contudo, detecta-la do seguinte modo. Suponhamos que queremos verificar se `U1` pode ser bloqueado, *i.e.*, alcançar um estado do qual não possa evoluir. Para isso vamos colocar o sistema `R` num ambiente que esconda as sinalizações de sucesso `suc2` e `suc3`. A maneira de o fazer é usar o operador de renomeação para as renomear para `tau`. Assim, vamos pedir ao CWB-NC para analisar o processo

```
agent  RR = R[tau/suc2,tau/suc3]
```

Note que este tipo de renomeação não é válido em CCS. No CWB-NC, porém, é permitido, precisamente porque fornece um método interessante para analisar certo tipo de situações complexas.

Exercício 11

Usando este método verifique se algum dos três processos utilizadores pode ficar bloqueado.

Exercício 12

O comportamento resultante da renomeação para `tau` de determinadas acções num processo é, por vezes, designado por *projecção* do processo. Um efeito semelhante pode ser obtido usando o operador de restricção, como acima indicamos. Qual lhe parece ser a diferença entre estes dois modos de proceder? Em que diferentes situações serão eles úteis?

Em diversos exercícios foi-lhe pedido que encontrasse uma especificação mais simples que outra dada, *i.e.*, com um menor espaço de estados mas provadamente equivalente (via `eq`). Efectivamente o CWB-NC pode fazer esse cálculo por si. Para isso pode recorrer ao comando `min` que determina um processo equivalente ao dado com um número mínimo de estados. Surpreendentemente, este comando é relativamente eficiente. Por isso, em muitos casos, o processo mais rápido de detectar *deadlocks* consiste em minimizar o processo e verificar (com o comando `pe`) se `0` ocorre no processo minimizado. Por exemplo faça

```
Command:  min(MinR,R);

MinR has 12 states.
```

Agora use `pe` para analisar `MinR`. O resultado é, talvez, excessivamente grande para ser útil. Será, então melhor minimizar cada uma das projecções de `R`. Por exemplo,

```
Command:  min(MinR3,R[tau/suc3]);

MinR3 has 3 states.
```

Exercício 13

Examine estes 3 estados com `pe` e verifique se o comportamento exibido é ou não o esperado. Faça o mesmo para as restantes projecções de `R`.

---

3 Aplicação

Recorde o problema, estudado nas aulas, do controlador de um cruzamento entre uma estrada e uma linha férrea. O enunciado original era o seguinte:

As acções car e $train$ representam a aproximação do cruzamento por um automóvel ou um comboio, respectivamente. Por seu lado, up e dw representam a abertura e o fecho da cancela sobre a estrada,

enquanto $green$ e red modelam a recepção de um sinal de avanço ou paragem pelo comboio. Finalmente, as acções $\overline{ccross}$ e $\overline{tcross}$ traduzem, respectivamente, a travessia efectiva do cruzamento por um automóvel ou um comboio.

$$\begin{aligned} Road &\triangleq car.up.\overline{ccross}.\overline{dw}.Road \\ Rail &\triangleq train.green.\overline{tcross}.\overline{red}.Rail \\ Signal &\triangleq \overline{green}.red.Signal + \overline{up}.dw.Signal \end{aligned}$$

$$C \triangleq \mathsf{new}\ \{green, red, up, dw\}\ (Road \mid Rail \mid Signal)$$

Exercício 14

Especifique este sistema no CWB-NC e faça, com apoio dessa ferramenta, um estudo completo do problema. Utilize nessa análise os diferentes tipos de equivalência entre processos que estudou e que estão implementados no CWB-NC.
