Laboratório 1: Animação de Processos no CWB-NC

Luís Soares Barbosa

Sumário

O objectivo deste trabalho é consolidar a capacidade de especificar *sistemas reactivos na linguagem de processos que estudou e* analisar *essas especificações por recurso ao* CONCURRENCY WORKBENCH *(*CWB-NC*). 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. Apenas complementarmente deverá anexar algum* printout *do* CWB-NC.

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

---

1 Pimeiros Passos

1.1 O CWB-NC

O CWB-NC (acrónimo do *Concurrency Workbench of the New Century*) é uma ferramenta de domínio público, que incorpora um conjunto de técnicas para modelação e verificação de sistemas concorrentes. Permite, em particular, recorrer à linguagem de modelação estudada neste curso para especificar processos e testar as noções de equivalência e refinamento estudadas. O CWB-NC permite, além disso, simular o comportamento de um processo passo-a-passo, detectar estados em *deadlock* ou determinar se um processo satisfaz determinadas propriedades (quer de segurança quer de animação) formuladas numa lógica adequada.

O CWB-NC, e documentação associada, pode ser obtido para diversas plataformas computacionais no endereço

`www.cs.sunysb.edu/ cwb/`

1.2 Definição de Ambientes

Uma das primeiras coisas que pode fazer no CWB-NC é definir processos e associa-los a identificadores (designados por *agent identifiers* ou *variables* no Manual). Para isso recorre-se ao comando `agent`. Por exemplo,

```
Command: agent P = a.0;
```

O processo `P` que acabou de definir realiza a acção `a` e fica inactivo (`0` é a representação do processo **0**). Note que os identificadores de processos iniciam-se sempre por uma letra maiúscula, enquanto os

símbolos de acções se iniciam com minúsculas. As associações entre definições de processos e identificadores constituem o *ambiente de trabalho*, cujo o estado corrente pode ser inpeccionado através do comando `pe`. Assim,

```
Command: pe;

 ** Agents **

agent P = a.0
```

Note que é sempre possível re-utilizar um identificador já envolvido numa associação. Nesse caso, a associação prévia é substituida pela nova. Tente, por exemplo, redefinir `agent P = a.a.0;`, repare no aviso que obteve e execute de novo `pe`. O comando `clear;` remove todas as associações no ambiente de trabalho.

1.3 Aspectos Sintáticos

O comando `agent` pode ajuda-lo a explorar a sintaxe correcta para a especificação de processos no CWB-NC. Recorra ao comando `help syntax;` para a recapitular sempre que necessário. Repare que os processos são construidos a partir de acções (sempre iniciadas com minúsculas) e acções complementares iniciadas por `'`. Por exemplo, `'a` ou `'b`. A acção interna τ é representada por `tau`. Os operadores escolha (`+`), composição paralela (`|`), restricção (`\`) têm a sintaxe esperada. A substituição de variáveis tem aqui o estatuto de um operador próprio, dito de renomeação e representado por `[a / b]`. Nesta fase tente definir alguns processos e assegurar-se que domina a sua sintaxe.

Exercício 1

Verifique o seguinte conjunto de definições incorrectas, típicas de utilizadores pouco experimentados. Forneça-as ao sistema e analise as mensagens de erro que obteve. Será capaz de corrigir os erros?

```
a.b.0 | 'a.'b0                    (c.b.0 | 'c.'b.0)\a,b
(a.0 + b.0).c.0                  a (b.0 + c.0)
(a. (b.0 | c.0)                  B[a\b]
(c.0 + tau.0)\tau                a.b.
```

Repare, por fim, que pode sempre usar um identificador de processo mesmo que não tenha sido previamente definido, por exemplo `P = a.Q`. O CWB-NC apenas acusará o facto quando tentar analisar o processo `P`. Pode, também, usar identificadores recursivamente: por exemplo, `P = a.P` corresponde ao processo $P \triangleq a.P$.

2 Simulação e Análise de Processos

2.1 Análise de Transições

A forma mais básica de análise de um processo consiste em estudar as suas transições, o que pode ser feito com o comando `transitions` ou, abreviadamente, `tr`. O comando aceita como parâmetro um processo e fornece como resultado a lista de transições disponíveis a partir dele. Por exemplo,

```
Command: tr c.0;
```

```
--- c ---> 0

Command: tr c.C|d.D;
--- c ---> C|d.D
--- d ---> c.C|D
```

Exercício 2

Verifique as transições iniciais dos seguintes processos. Tente primeiro calcula-las mentalmente e use depois o CWB-NC para conferir os seus cálculos.

1. `a.P | 'a.Q`
2. `(a.P + b.0)|c.Q`
3. `(a.P + b.0)|'a.Q`
4. `(a.P | 'a.Q)\a`
5. `((a.0 + b.0) | (c.0 + d.0))`
6. `((a.P + b.0) | 'a.Q)\a`

Verifique, agora, as transições iniciais de `a.0 + P`, com `P` não definido. Qual a mensagem de erro que obteve? Porque razão o mesmo não terá sucedido com os outros exemplos (em que também `P` e `Q` não estavam definidos)?

Concerteza já concluiu que o comando `tr` apenas calcula as transições iniciais de um processo. Por exemplo, em

```
Command: tr c.P;
--- c ---> P
```

as derivações de `P` não são analisadas por este comando.

Exercício 3

Considere, agora, os processos

```
Command: agent P = a.P;
Command: agent Q = a.P + b.Q;
```

Determine as transições a partir de `P`, `Q` e `P|Q`.

Há um ponto que requer alguma atenção quando se calculam as transições de processos definidos recursivamente. A recursão deve ser guardada por uma prefixação. Caso contrário o CWB-NC não irá cooperar:

```
Command: agent P = a.0 + P;
Command: tr P;

Resetting tables ...

*** Non-well-founded recursion in P ***
```

Neste exemplo `P` ocorre guardado na primeira parcela mas não guardado na segunda. Como regra de algibeira considere que sempre que, partindo de um identificador, fôr capaz de prosseguir através de definições até ao mesmo identificador, sem passar por uma prefixação, então o idenficador está definido por uma recursão não guardada.

Muitas vezes há interesse em explorar o comportamento de um processo para além das transições iniciais. Naturalmente tal pode ser feito por aplicação de `tr` repetidamente a todas as derivações do processo. O procedimento é certamente moroso e susceptível de erros. Uma solução melhor consiste em usar o comando `sim` que activa um pequeno simulador que permite explorar todo o universo de transições de um processo. Tente, por exemplo, simular o processo seguinte

```
(a.b.0 | a.c.0 | 'a.0)\a
```

Este processo tem três componentes em paralelo: duas começam por `a` e continuam por `b` e `c`, respectivamente; a terceira inicia-se por `'a` e pode, consequentemente, sincronizar com qualquer das duas primeiras. Assim,

```
Command: sim (a.b.0 | a.c.0 | 'a.0)\a;

Simulated agent: (a.b.0 | a.c.0 | 'a.0)\a
Transitions:
    1: --- tau<a> ---> (a.b.0 | c.0 | 0)\a
    2: --- tau<a> ---> (b.0 | a.c.0 | 0)\a
[0]Sim:
```

O resultado significa que existem duas transições iniciais não observáveis, ambas resultantes de sincronizações através de `a` (repare na forma como isso é indicado). O simulador solicita, então, a escolha de uma transição. Vamos escolher a primeira digitando `1;`

```
[0]Sim: 1;

  --- tau<a> --->

Simulated agent: (a.b.0 | c.0 | 0)\a
Transitions:
    1: --- c ---> (a.b.0 | 0 | 0)\a

[1]Sim:
```

Vemos, assim, que a primeira transição τ pode continuar com uma transição por `c`. De novo podemos continuar a explorar as transições seguintes ou voltar atrás digitando `return` seguido de um número que indica a posição na simulação à qual se pretende regressar. Note que esta posição é sempre indicada entre parentesis rectos antecedendo o *prompt* `Sim:`. Por exemplo, digitando `return 0` a simulação regressa ao agente inicial:

```
[1]Sim: return 0;

Simulated agent: (a.b.0 | a.c.0 | 'a.0)\a
Transitions:
    1: --- tau<a> ---> (a.b.0 | c.0 | 0)\a
    2: --- tau<a> ---> (b.0 | a.c.0 | 0)\a

[0]Sim:
```

Desta forma pode explorar o comportamento do processo, evoluindo para trás e para diante ao longo das suas transições. O simulador tem, ainda, alguns outros comandos, tal como `history`, que retorna uma lista dos processos encontrados até ao momento juntamente com os números de posição

na simulação em que ocorreram (confira no Manual as outras opções disponíveis). Digite `quit;` para terminar a simulação.

Exercício 4

Continue exaustivamente a simulação deste processo. Quantos processos distintos encontra? Quantas transições existem? Como reage o simulador quando pretende prosseguir a partir de um processo sem transições disponíveis?

2.2 Outras Técnicas de Análise

Dois outros comandos revelam-se muito úteis na análise dos diversos estados (processos) acessíveis por realização de transições — *i.e.*, os nodos do respectivo grafo de transições. São eles o comando `state`, que lista todos os estados acessíveis, e `size` que apenas retorna o seu número. Por exemplo,

```
Command: states A;
1: 0
2: abort.0
3: 'out.A
4: in.'out.A
5: A

Command: size A;

A has 5 states
```

Exercício 5

Tente `states` e `size` no processo que definiu no exercício anterior.

Identificadores livres e recursões não guardadas causam sempre problemas quando tenta aplicar este tipo de comandos. Outra fonte de erros, geralmente mais difícil de detectar, é a infinitude do espaço de estados.

Exercício 6

Defina `agent S = push.(S | pop.0)` e inspeccione as sequências observáveis de comprimento de 1 a 4. Que padrão observa? Aplique o simulador a este processo até se certificar que entendeu o seu comportamento.

Note que se o espaço de estados é infinito, qualquer tentativa de executar `size` ou `states` desencadeia uma computação que não termina (pelo menos até esgotar a memória disponível!). O comando pode ser abortado com control-c em qualquer ponto, mas, infelizmente, não irá obter qualquer mensagem de erro interessante: de facto, o CWB-NC não pode saber antecipadamente que o espaço de estados em análise era realmente infinito [1].

Se o espaço de estados é finito, mas muito grande, o sistema pode demorar algum tempo a calcular uma resposta. Um teste que pode realizar consiste em definir um processo `T` como `a.0`. Verifique que `T` tem dois estados, `T|T` tem quatro, `T|T|T` tem oito e assim sucessivamente. Cada vez que agrega mais

[1] É útil recordar que a segunda razão mais comum para um comando parecer não terminar é a presença de um espaço de estados infinito (ou muito grande). A primeira é obviamente o esquecimento do ;.

uma componente em paralelo o espaço de estados é duplicado, assim como o é o tempo necessário para execução de um comando como `size` ou `states`.

Exercício 7

Determine, experimentalmente, o número máximo de estados que pode considerar de forma que o comando `size` demore menos de 30 segundos a ser executado no seu computador.

Exercício 8

Defina os processos

```
agent B = in.'out.B ;
agent S = (B[m/out]|B[m/in])\m ;
```

Note que, em alternativa ao operador de renomeação, pode usar uma notação de definição mais próxima daquela a que recorremos nas aulas:

```
agent B(in,out) = in.'out.B(in,out) ;
agent S = (B(in,m)|B(m,out))\m ;
```

Desenhe o respectivo grafo de transições, usando, primeiro o comando `states` para determinar os estados acessíveis. Use o simulador para explorar as transições disponíveis. A partir do grafo de transições que desenhou deduza as sequências observáveis de comprimento 4. Verifique o resultado com o comando `vs`.

Outra ferramenta de análise que o CWB-NC disponibiliza é o comando `sort` que calcula a espécie sintática de um processo, sendo muito útil para detectar gralhas na escrita das especificações. Tente, por exemplo,

```
agent X = (a.b.X | a'.c.0 | 'a.'b.X)\{a,b};

Command:  sort X;
```

O resultado é `{a',c}`. Mas, atendendo a que `a` e `b` foram tornados internos, o resultado esperado deveria ser apenas `{c}`. Reparando com mais atenção, verificamos que a plica no `a` está mal colocada. Corrija e calcule de novo:

```
agent Y = (a.b.X | 'a.c.0 | 'a.'b.X)\{a,b};

Command:  sort Y;
```

E voltou a obter o mesmo resultado `{a',c}`!

Exercício 9

Porquê?

Quando trabalhar com exemplos maiores será aconselhável recorrer a ficheiros externos para guardar e editar o seu trabalho. O ficheiro poderá conter comentários, que, no CWB-NC, começam sempre por * e se estendem até ao fim da linha. O ficheiro é carregado via

```
Command: input "nome_do_ficheiro";
```

No caso de usar comandos em situações em que a informação produzida pelo sistema é muito grande, poderá sempre redireccionar a saída para um outro ficheiro, que, mais tarde poderá imprimir ou editar. Basta fazer

```
Command: output "nome_do_ficheiro";
```

O comando `output;` sem mais argumentos volta a redireccionar a saída para o terminal.