Universidade do Minho

Licenciatura em Ciências da Computação

Interação e Concorrência

Problem Set - 1

Carlos Beiramar a84628
17 de abril de 2021

Conteúdo

1 Exercícios

1.1 Exercício 1

Enunciado: Make yourself familiar with *mCRL2* (read the documentation and try examples). Complete the tutorial available from this course webpage.

Resolução:

```
sort Position = struct start | finish;

act
    forward_flashlight,forward_adventurer,forward,forward_referee: Int # Int;

    back_flashlight,back,report,back_referee,back_adventurer: Int;

proc Flashlight(pos:Position) =
       (pos == start) ->
         % Case 1.
         sum s,s':Int . forward_flashlight(s,s') . Flashlight(finish)
       <>
         % Case 2.
         sum s:Int . back_flashlight(s) . Flashlight(start);

proc Adventurer(speed:Int, pos:Position)=
        (pos == start) ->
        sum s:Int .
        (s > speed) -> forward_adventurer (speed,s) . Adventurer(speed, finish)
                        <>  forward_adventurer(s,speed). Adventurer(speed,finish)
        <>
        back_adventurer(speed) . Adventurer(speed, start);


proc Referee (minutes:Int, num_finished:Int)=
    sum s,s': Int . forward_referee(s,s').Referee(minutes + max(s,s'), num_finished +2)
    +
    (num_finished < 4) -> sum s:Int . back_referee(s) . Referee(minutes + s, num_finished - 1)
                          <>
                            report(minutes). Referee(minutes,num_finished);

init
  allow({forward, back,report},
   comm({forward_adventurer | forward_adventurer | forward_flashlight | forward_referee
            -> forward,
         back_adventurer | back_flashlight | back_referee -> back},
       Adventurer(1,start) || Adventurer(2,start)  ||
       Adventurer(5,start) || Adventurer(10,start) ||
       Flashlight(start) || Referee(0,0)
     ));
```

Figura 1: Código do tutorial

1.2 Exercício 2

Enunciado: Write a short note (maximum 3 pages) explaining carefully, for the layman, with a running example, how *mCRL2* can be used. Extra bonus to answers not relying in (non modified) examples available from the tool documentation.

Resolução: *mCRL2* ou, por outras palavras, **micro Common Representation Language 2** é uma linguagem de especificação que poderá ser utilizada para analisar o comportamento dos sistemas distribuídos.

Esta linguagem é apoiada num conjunto de ferramentas (toolset) que permite fazer simulações, observações e verificar o comportamento dos requisitos do software. Esta linguagem é baseada numa Álgebra de Comunicação de Processos (ACP), que permite a construção de processos deveras complexos e cujo conceito fundamental é o "processo".
Estes processos podem realizar ações e, podem também, ser compostos por outros processos. Todos estes processos têm um estado que pode influenciar todas as ações que o processo vai executar, ações essas que têm a capacidade de alterar esses mesmos estados.

O exemplo que decidi abordar foi o famoso problema: *jantar de filósofos*. Este problema consiste num grupo de três filósofos que se senta numa mesa para jantar na qual cada um tem o seu respetivo prato e, entre cada par de pratos existe um e um só garfo. Para comer as refeições que vão ser servidas, cada um dos filósofos necessita de exatamente 2 garfos. A ilustração deste problema está presente na figura abaixo.

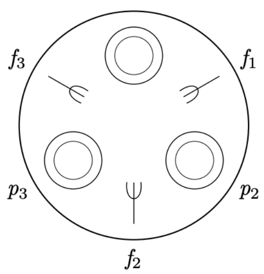


Figura 2: Jantar de filósofos

O código usado está representado na figura que se segue:

```
sort Plate_id = struct p1 | p2 | p3;
     Fork_id = struct f1 | f2 | f3;


%define functions

map left_fork, right_fork: Plate_id -> Fork_id;

eqn
    left_fork(p1) = f1;
    left_fork(p2) = f2;
    left_fork(p3) = f3;

    right_fork(p1) = f3;
    right_fork(p2) = f1;
    right_fork(p3) = f2;


act pick_fork,drop_fork: Plate_id # Fork_id;

    up_fork, down_fork, lock,free: Plate_id # Fork_id;

    eat: Plate_id;

proc
    Plate (p: Plate_id) = (pick_fork(p,left_fork(p)) . pick_fork(p,right_fork(p))) + (pick_fork(p,right_fork(p)) . pick_fork(p,left_fork(p))) . eat(p).
                          (drop_fork(p,left_fork(p)) . drop_fork(p,right_fork(p))) + (drop_fork(p,right_fork(p)) . drop_fork(p,left_fork(p))) . Plate(p);


    Fork (f: Fork_id) = sum p: Plate_id . up_fork(p,f) . down_fork(p,f) . Fork(f);

init allow({lock,free,eat},comm({pick_fork|up_fork -> lock, drop_fork|down_fork -> free},
                                Plate(p1) || Plate(p2) || Plate(p3) || Fork(f1) || Fork(f2) || Fork(f3)));
```

Figura 3: Código do jantar

Explicação do código:

Foram definidas duas *structs*, `Plate_id` que representa os 3 pratos presentes na mesa atribuindo a cada um deles um id e uma *struct* `Fork_id` que é aplicada aos garfos de forma análoga.
A *função* `left_fork` designa qual é o garfo que está à esquerda de cada um dos filósofos e a *função* `right_fork` designa qual é o garfo que está à direita de cada um dos filósofos. As *ações*:

- *pick_fork(p_i,left_fork(p_i))* e *drop_fork(p_i,left_fork(p_i))*, indicam que o filósofo p_i faz pick e drop,respetivamente, aos garfos que estão ao seu lado. As ações correspondentes feitas pelos *Fork* são *up_fork(p_i,f)* e *down_fork(p_i,f)*, que significam que o *fork* f é levantado ou pousado pelo filósofo p_i.
- a ação *eat(p)* significa que o filósofo p está a comer.
- as ações *lock* e *free* foram feitas para indicar quando um *Fork* é levantado ou pousado por um filósofo, respetivamente.

O processo *Plate(p_i)* planeia o comportamento de cada Filósofo. Primeiramente pega em 2 *Forks* independetemente da ordem, depois come e pousa os *Forks*. O processo *Fork(f_i)* planeia o comportamente de cada garfo. Pode ser usado por um filósofo p e depois pousado pelo mesmo, podendo voltar a repetir este comportamento. Este sistema consiste em três *Plate* e três *Fork* a ocorrerem em paralelo.

#	Action	State Change
0		s1_Plate1 := 1,p_Plate1 := p1,s2...
1	lock(p2,f1)	s2_Plate1 := 5,s4_Fork := 2,p_...
2	lock(p2,f2)	s2_Plate1 := 7,s5_Fork := 2,p_...
3	lock(p1,f3)	s1_Plate1 := 5,s6_Fork := 2
4	eat(p2)	s2_Plate1 := 8
5	free(p2,f2)	s2_Plate1 := 9,s5_Fork := 1,p_...
6	free(p2,f1)	s2_Plate1 := 2,p_Plate := p1,s4...
7	lock(p3,f2)	s3_Plate1 := 5,s5_Fork := 2,p_...
8	lock(p1,f1)	s1_Plate1 := 7,s4_Fork := 2
9	eat(p1)	s1_Plate1 := 8
	free(p1,f1)	s1_Plate1 := 9,s4_Fork := 1
	free(p1,f3)	s1_Plate1 := 2,s6_Fork := 1
	lock(p3,f3)	s3_Plate1 := 7,s6_Fork := 2,p_...
	eat(p3)	s3_Plate1 := 8
	free(p3,f3)	s3_Plate1 := 9,s6_Fork := 1,p_...
	free(p3,f2)	s3_Plate1 := 2,p_Plate2 := p1,s...

Figura 4: Exemplo de execução do códgigo

1.3 Exercício 3

Enunciado: Consider the following junction controlled by three traffic lights (processes A_1, A_2 and A_3 whose individual behaviour is specified by the following labelled transition system:

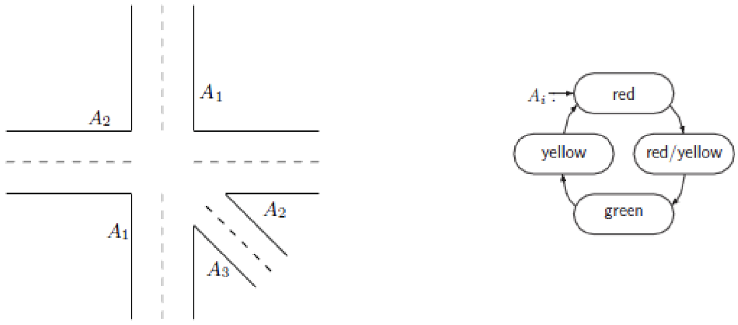


Figura 5: Exemplo do sistema

1.3.1 Exercício 3.1

Enunciado: Specify in *MCRL*2 the process traffic light. Please note you need to choose identifiers for all the relevant actions which are omitted in the graphical sketch.

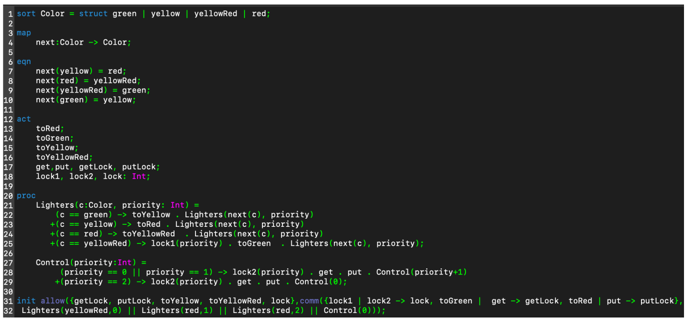
Resolução:

```
sort Color = struct green | yellow | yellowRed | red;

map
    next:Color -> Color;

eqn
    next(yellow) = red;
    next(red) = yellowRed;
    next(yellowRed) = green;
    next(green) = yellow;

act
    toRed;
    toGreen;
    toYellow;
    toYellowRed;

    get,put, getLock, putLock;


proc
    Lighters(c:Color)=
        (c == green) -> toYellow . Lighters(next(c))
       +(c == yellow) -> toRed . Lighters(next(c))
       +(c == red) -> toYellowRed  . Lighters(next(c))
       +(c == yellowRed) -> toGreen  . Lighters(next(c));


init allow({getLock, putLock, toRed, toYellowRed},comm({toGreen |  get -> getLock, toYellow | put -> putLock},
 Lighters(yellowRed) || Lighters(red) || Lighters(red));
```

Figura 6: Sistema de transições

1.3.2 Exercício 3.2

Enunciado: Specify in $MCRL2$ a control process X to ensure that the green light is activates in sequence, and in infinite cycle respecting the following order: A_1, A_2 and A_3.



```
sort Color = struct green | yellow | yellowRed | red;

map
    next:Color -> Color;

eqn
    next(yellow) = red;
    next(red) = yellowRed;
    next(yellowRed) = green;
    next(green) = yellow;

act
    toRed;
    toGreen;
    toYellow;
    toYellowRed;
    get,put, getLock, putLock;
    lock1, lock2, lock: Int;

proc
    Lighters(c:Color, priority: Int) =
        (c == green) -> toYellow . Lighters(next(c), priority)
       +(c == yellow) -> toRed . Lighters(next(c), priority)
       +(c == red) -> toYellowRed  . Lighters(next(c), priority)
       +(c == yellowRed) -> lock1(priority) . toGreen  . Lighters(next(c), priority);

    Control(priority:Int) =
         (priority == 0 || priority == 1) -> lock2(priority) . get . put . Control(priority+1)
        +(priority == 2) -> lock2(priority) . get . put . Control(0);

init allow({getLock, putLock, toYellow, toYellowRed, lock},comm({lock1 | lock2 -> lock, toGreen |  get -> getLock, toRed | put -> putLock},
 Lighters(yellowRed,0) || Lighters(red,1) || Lighters(red,2) || Control(0)));
```

Figura 7: Processo controlo

Explicação do código: Na primeira linha está definida a estrutura das cores para os semáforos. A função *next* foi definida com o objetivo de indicar ao semáforo qual será a sua próxima cor. As ações:

- $toRed$, $toGreen$, $toYellow$, $toYellowRed$, são ações que indicam ao semáforo que vai mudar para essa cor.
- get, put, $getLock$, $putLock$, ações para realizar as sincronizações.
- $lock1$, $lock2$ e $lock$, recebem como parâmetro um inteiro que vai ser utilizado para indicar qual o semáforo a que vamos dar $lock$.

O processo *Lighters* recebe como argumento uma cor e um inteiro que define a prioridade para cada semáforo e executa uma ação que vai depender dessa mesma cor e, após essa ação, há uma chamada recursiva com a próxima cor e com a mesma prioridade.

O processo *Control* recebe como argumento um inteiro que nos indica a prioridade do semáforo e contém as ações $lock2$, get, put e a chamada recursiva do mesmo. O objetivo deste processo é garantir que apenas um semáforo contém a cor verde e que essa cor ocorre por ordem de prioridades dos semáforos.

Como se pode verificar na função *Lighters* quando a cor é $YellowRed$ é executado um $lock1$ que está sincronizado, na **init**, com a *ação* $lock2$ presente no Control. Depois de executado este lock em simultâneo, o semáforo muda para *green* e, o *lock* só é libertado quando este mesmo semáforo mudar para *red*, porque a ação *put* está sincronizada com a ação *toRed*.

#	Action	State Change
0		s1_Lighters := 1,c_Lighters := yellowRed,s2_Lighters := 1,c_Li...
1	lock(0)	s1_Lighters := 2,s4_Control := 2
2	getLock	s1_Lighters := 1,c_Lighters := green,s4_Control := 5
3	toYellowRed	c_Lighters1 := yellowRed
4	toYellowRed	c_Lighters2 := yellowRed
5	toYellow	c_Lighters := yellow
6	putLock	c_Lighters := red,s4_Control := 1,priority_Control := 1
7	lock(1)	s2_Lighters := 2,s4_Control := 2
8	getLock	s2_Lighters := 1,c_Lighters1 := green,s4_Control := 5
9	toYellow	c_Lighters1 := yellow
	toYellowRed	c_Lighters := yellowRed
	putLock	c_Lighters1 := red,s4_Control := 1,priority_Control := 2
	lock(2)	s3_Lighters := 2,s4_Control := 3,priority_Control := 0
	getLock	s3_Lighters := 1,c_Lighters2 := green,s4_Control := 4
	toYellow	c_Lighters2 := yellow
	putLock	c_Lighters2 := red,s4_Control := 1

Figura 8: Exemplo de execução do código

Como é possível verificar neste exemplo de execução, é feito um $lock(0)$ para dar *lock* ao semáforo com prioridade a 0, de seguida faz-se um $getLock$ para alterar a cor do semáforo para *green*, após isso é mudada a cor dos outros 2 semáforos para $yellowRed$ e a cor que se sucede a esta é o *green*, no entanto não é possível ter 2 semáforos a *green* por isso é obrigatório mudar a cor do semáforo com prioridade 0 para *yellow* e fazer um $putLock$ para que seja possível meter a *green* o semáforo com prioridade 1.

1.3.3 Exercício 3.3

Enunciado: Draw the transition system of the following process:

$$S = X \mid A_1 \mid A_2 \mid A_3$$

Resolução:

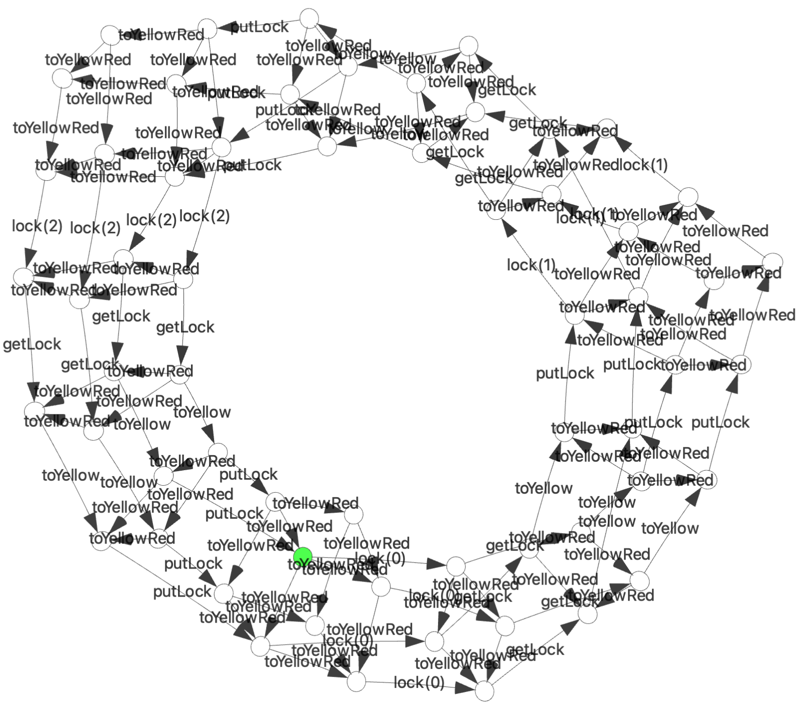


Figura 9: Sistema de transições

1.3.4 Exercício 3.4

Enunciado: Apply once the expansion theorem to process S, Comment the result.

Resolução:

$$S = Control(0) \mid Lighters(yellowRed, 0) \mid Lighters(red, 1) \mid Lighters(red, 2)$$

$$S = (Control(0)|Lighters(yellowRed, 0)|Lighters(red, 1)|Lighters(red, 2))\backslash_{\{lock,getLock,putLock\}}$$

a $= \tau \cdot \Big(\tau \cdot \big(\tau \cdot Control(1)\big)\Big)\backslash_{\{getLock,putLock,lock\}}$

Explicação: O primeiro τ ocorre porque a ação $lock2$ está sincronizada com a ação $lock1$, originando a ação $lock$. O segundo τ ocorre devido à ação get que está sincronizada com a ação $toGreen$ originando a ação $getLock$. O terceiro τ ocorre devido à ação put que está sincronizada com a ação $toYellow$ originando a ação $putLock$.

b $= \tau \cdot \Big(\tau \cdot \big(Lighters(green, 0)\big)\Big)\backslash_{\{getLock,putLock,lock\}}$

Explicação: O primeiro tau vem da relação de sincronização entre as ações $lock1$ e $lock2$.O segundo τ é resultante da ação $toGreen$ pois, esta está em sincronização com a ação get.

c $= toYellowRed \cdot \big(Lighters(yellowRed, 1)\big)\backslash_{\{getLock,putLock,lock\}}$

d $= toYellowRed \cdot \big(Lighters(yellowRed, 2)\big)\backslash_{\{getLock,putLock,lock\}}$

Explicação: Estas últimas duas expansões vão de acordo com a definição que está no processo $Lighters$. Obtendo-se assim, pelo teorema da expanção:

$$S = a + b + c + d$$

1.4 Exercício 4

Enunciado: Consider the following specification of a *pipe*, as supported e.g. in UNIX:

$$\mathcal{U} \rhd \mathcal{V} \stackrel{\text{def}}{=} (\{c/out\}\mathcal{U}|\{c/in\}\mathcal{V})\backslash_{\{c\}}$$

under the assumption that, in both processes, actions $\overline{out}$ and in stand for, respectively, the output and input ports.

1.4.1 Alínea a

Enunciado: Consider now the following processes only partially defined:

$$\mathcal{U}_1 \mathrel{\hat{=}} \overline{out} \cdot \mathcal{T}$$

$$\mathcal{V}_1 \mathrel{\hat{=}} in \cdot \mathcal{R}$$

$$\mathcal{U}_2 \mathrel{\hat{=}} \overline{out} \cdot \overline{out} \cdot \overline{out} \cdot \mathcal{T}$$

$$\mathcal{V}_2 \mathrel{\hat{=}} in \cdot in \cdot in \cdot \mathcal{R}$$

Prove, by equational reasoning, or refute the following properties:

i. $\mathcal{U}_1 \rhd \mathcal{V}_1 \sim \mathcal{T} \rhd \mathcal{R}$

ii. $\mathcal{U}_2 \rhd \mathcal{V}_2 = \mathcal{U}_1 \rhd \mathcal{V}_1$

Resolução:

i. Tem-se que:

$$\mathcal{T} \rhd \mathcal{R} = (\{c/out\} \cdot \mathcal{T} \mid \{c/in\} \cdot \mathcal{R})\backslash_{\{c\}} \tag{1}$$

$$\begin{aligned} \mathcal{U}_1 \rhd \mathcal{V}_1 &= (\{c/out\} \cdot \mathcal{U}_1|\{c/in\} \cdot \mathcal{V}_1)\backslash_{\{c\}} \\ &= (\bar{c} \cdot \{c/out\}|c \cdot \{c/in\}\mathcal{R})\backslash_{\{c\}} \\ &= \tau \cdot (\{c/out\} \cdot \mathcal{T}|\{c/in\} \cdot \mathcal{R})\backslash_{\{c\}} \end{aligned} \tag{2}$$

Por um lema sabe-se que, num processo E qualquer, $\tau \cdot E \neq E$. Assim é possível concluir que (1) $\neq$ (2) e como $\sim \subseteq =$ entao a afirmação $\mathcal{U}_1 \rhd \mathcal{V}_1 \sim \mathcal{T} \rhd \mathcal{R}$ é falsa.

ii. Tem-se que:

$$\begin{aligned}
\mathcal{U}_2 \rhd \mathcal{V}_2 &\,\hat{=}\, (\{c/out\}\mathcal{U} \mid \{c/in\}\mathcal{V})\backslash_{\{c\}} \\
&= (\{c/out\}(\overline{out} \cdot \overline{out} \cdot \overline{out} \cdot \mathcal{T})|\{c/in\}in \cdot in \cdot in \cdot \mathcal{R})\backslash_{\{c\}} \\
&\equiv ((\overline{c} \cdot \overline{c} \cdot \overline{c} \cdot \mathcal{T})|c \cdot c \cdot c \cdot \mathcal{R})\backslash_{\{c\}} \\
&\sim \tau \cdot \tau \cdot \tau \cdot (\mathcal{T} \mid \mathcal{R})\backslash\{c\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{U}_1 \rhd \mathcal{V}_1 &\,\hat{=}\, (\{c/out\}\overline{out} \cdot \mathcal{T} \mid \{c/out\}in \cdot \mathcal{R})\backslash_{\{c\}} \\
&\equiv (\overline{c} \cdot \mathcal{T} \mid c \cdot \mathcal{R})\backslash_{\{c\}} \\
&\sim \tau \cdot (\mathcal{T} \mid \mathcal{R})\backslash_{\{c\}}
\end{aligned}$$

Logo, depois destas substituições, foi obtida uma igualdade. Assim é possível fazer a seguinte bissimulação:

$$S = \{(\mathcal{U}_2 \rhd \mathcal{V}_2 \,,\, \mathcal{U}_1 \rhd \mathcal{V}_1)\}$$

Logo $\mathcal{U}_2 \rhd \mathcal{V}_2 \approx \mathcal{U}_1 \rhd \mathcal{V}_1$.

Para além disso, conclui-se que é possível, começar em $\mathcal{U}_2 \rhd \mathcal{V}_2$ e chegar a $(\mathcal{T} \mid \mathcal{R})\backslash\{c\}$ através de uma ação ϵ. Análogamente para $\mathcal{U}_1 \rhd \mathcal{V}_1$. Assim é concluído que a afirmação é verdadeira.

1.4.2 Alínea b

Enunciado: Show or refute the associativity of $\rhd$ wrt process equality, *i.e.*, for all, $\mathcal{P}, \mathcal{T}, \mathcal{V} \in P$,

$$\big(\mathcal{U} \rhd \mathcal{V}\big) \rhd \mathcal{T} = \mathcal{U} \rhd \big(\mathcal{V} \rhd \mathcal{T}\big)$$

Resolução:

$$(\mathcal{U} \rhd \mathcal{V}) = \big(\{c/out\} \cdot \mathcal{U} \;\mid\; \{c/in\}\mathcal{V}\big)\backslash_{\{c\}}$$

$$\big(\mathcal{V} \rhd \mathcal{T}\big) = \big(\{c/out\} \cdot \mathcal{V} \;\mid\; \{c/in\} \cdot \mathcal{T}\big)\backslash_{\{c\}}$$

$$\begin{aligned}
\big(\mathcal{U} \rhd \mathcal{V}\big) \rhd \mathcal{T} &= \Big(\{c/out\}\big(\{c/out\} \cdot \mathcal{U} \;\mid\; \{c/in\} \cdot \mathcal{V}\big) \;\mid\; \{c/in\} \cdot \mathcal{T}\Big)\backslash_{\{c\}} \\
&\equiv \big(\{c/out\}\{c/out\} \cdot \mathcal{U} \;\mid\; \{c/out\}\{c/in\} \cdot \mathcal{V} \mid \{c/in\} \cdot \mathcal{T}\big)\backslash_{\{c\}}
\end{aligned}$$

Pela definição da Semântica tem-se que "|"é um **monóide abeliano** logo é associativo. Assim,

$$
\begin{aligned}
&\equiv \Big(\{c/out\}\cdot\mathcal{U} \;\mid\; \{c/in\}\big(\{c/out\}\cdot\mathcal{V} \;\mid\; \{c/in\}\cdot\mathcal{T}\big)\Big)\backslash_{\{c\}}\\
&\equiv \Big(\{c/out\}\cdot\mathcal{U} \;\mid\; \{c/in\}\cdot\big(\mathcal{V}\rhd\mathcal{T}\big)\Big)\\
&\equiv \mathcal{U}\rhd\big(\mathcal{V}\rhd\mathcal{T}\big)
\end{aligned}
$$

1.4.3 Alínea c

Enunciado: Show that $\mathbf{0}\rhd\mathbf{0}=\mathbf{0}$

Resolução:

$$\mathbf{0}\rhd\mathbf{0} = (\{c/\overline{out}\}\mathbf{0}\,|\,\{c/in\}\mathbf{0})\backslash_{\{c\}}$$

$$= (\mathbf{0}\,|\,\mathbf{0})\backslash_{\{c\}}$$

Como c $\notin fn(\mathbf{0}\rhd\mathbf{0})$ tem-se que:

$$\mathbf{0}\rhd\mathbf{0} = \mathbf{0}\backslash_{\{c\}}\,|\,\mathbf{0}\backslash_{\{0\}}$$

e como $\mathbf{0}\backslash\{c\}\equiv 0$,

$$\mathbf{0}\rhd\mathbf{0} = (\mathbf{0}\,|\,\mathbf{0})$$

e como $\mathbf{0}\,|\,\mathbf{0}\equiv\mathbf{0}$,

Logo, $\mathbf{0}\rhd\mathbf{0}=\mathbf{0}$.

1.5 Exercício 5

Enunciado: Consider a combinator $\circlearrowleft_n$ whose operational semantics is given by following rule

$$\frac{E \xrightarrow{\text{a}} E'}{\circlearrowleft_0 E \xrightarrow{\text{a}} E'} \quad \frac{E \xrightarrow{\text{a}} E'}{\circlearrowleft_n E \xrightarrow{\text{a}} \circlearrowleft_{n-1} E} \text{ for } n > 0$$

1.5.1 Alínea a

Enunciado: Explain its purpose.

Resolução: Este processo funciona como um duplicador de ações, isto é, sempre que o processo E executa uma ação a e vai para E' ($E \xrightarrow{\text{a}} E'$) então, o processo $\circlearrowleft_n (E)$ pode executar a ação a n vezes.

Exemplo:

$$\circlearrowleft_2 (x \cdot y \cdot 0) \xrightarrow{\text{x}} \circlearrowleft_1 (x \cdot y \cdot 0) \xrightarrow{\text{x}} \circlearrowleft_0 (x \cdot y \cdot 0) \xrightarrow{\text{x}} (y \cdot 0)$$

1.5.2 Alínea b

Enunciado: Discuss wether, and for which values of m and n, one may have $\circlearrowleft_n (\circlearrowleft_m E) \sim \circlearrowleft_n E$.

Resolução: O único valor que m e n podem ter é 0. Veja se o seguinte exemplo. Supondo um processo $E = a \cdot b \cdot 0$

$$\begin{array}{ccc}
\circlearrowleft_0 (\circlearrowleft_0 a \cdot b \cdot 0) & \sim & \circlearrowleft_0 (a \cdot b \cdot 0) \\
\downarrow a & & \downarrow a \\
\circlearrowleft_0 (b \cdot 0) & \sim & (b \cdot 0) \\
\downarrow b & & \downarrow b \\
0 & \sim & 0
\end{array}$$

É possível verificar que, quando $m = 0$ e $n = 0$ obtemos uma bissimulação.

Explicação:

- No primeiro passo foi usada a definição da semântica operacional do combinador $\circlearrowleft_n$ dos dois lados.

- No segundo passo, do lado esquerdo foi usada a definição do combinador $\circlearrowleft_n$ mas, no lado direito, foi simplesmente aplicada uma ação que pertence ao processo E, ou seja uma ação que o processo E pode realizar.

Foi atingido assim o mesmo estado sempre pelas mesmas ações em ambos os lados, concluindo-se assim que a afirmação é verdadeira.

1.5.3 Alínea c

Enunciado: Show that $E \sim F$ implies $\circlearrowleft_n E \sim \circlearrowleft_n F$.

Resolução: Supondo que $E \sim F$.

$R = \{ (\circlearrowleft_n E , \circlearrowleft_n F) \,|\, E \sim F \}$

Sabe-se que ($\circlearrowleft_n E$) tem n transições por a para E, ($\circlearrowleft_n E \xrightarrow{\text{a}} \circlearrowleft_{n-1} E$), então, ($\circlearrowleft_n F$) também vai ter n transições por a para F, ($\circlearrowleft_n F \xrightarrow{\text{a}} \circlearrowleft_{n-1} F$), precisamente porque o E é bissimilar a F,($E \sim F$), e, por definição de bissimilares se o E transita por a então o F também transita por a.

1.5.4 Alínea d

Enunciado: Show, by a counter-example, that, whenever $\sim$ is replaced by $\approx$, the implication above fails.

Resolução:

$R = \{\, (\circlearrowleft_n E \,, \circlearrowleft_n F\,) \,|\, E \;\approx\; F \,\}$

Para todos os casos apresentados de seguida, sabemos sempre que:

$$\circlearrowleft_0 E \overset{a}{\Rightarrow} E'$$

e,

$$\circlearrowleft_0 F \overset{a}{\Rightarrow} F'$$

Existem vários casos quando realizámos $\circlearrowleft_n E \overset{\mathrm{a}}{\Rightarrow}$

- 1º caso: *Transição simples*

$$\circlearrowleft_n E \overset{a}{\rightarrow} \circlearrowleft_{n-1} E$$

  então, por $E \approx F$, quando realizámos $\circlearrowleft_n F \overset{\mathrm{a}}{\Rightarrow}$ também temos uma *Transição simples*

$$\circlearrowleft_n F \overset{a}{\rightarrow} \circlearrowleft_{n-1} F$$

- 2º caso: *Transição por τ*

$$\circlearrowleft_n E \overset{\tau}{\rightarrow} \circlearrowleft_{n-1} E$$

  então, ou o processo F repetiu a transição por τ, se o F começava por uma transição por τ,

$$\circlearrowleft_n F \overset{\tau}{\rightarrow} \circlearrowleft_{n-1} F$$

  ou, supondo que,

$$E \overset{\mathrm{a}}{\Rightarrow} E' \rightsquigarrow E \overset{\mathrm{a}}{\rightarrow} E'' \overset{\mathrm{a}}{\rightarrow} E'$$

  e que,

$$F \stackrel{a}{\Rightarrow} F' \rightsquigarrow F \stackrel{a}{\rightarrow} F'$$

então temos,

$$\circlearrowleft_n E \stackrel{\tau}{\rightarrow} \circlearrowleft_{n-1} E$$

e,

$$\circlearrowleft_0 E \stackrel{\tau}{\rightarrow} E'' \stackrel{a}{\rightarrow} E'$$

temos também,

$$\circlearrowleft_n F \stackrel{\tau}{\rightarrow} \circlearrowleft_{n-1} F$$

e,

$$\circlearrowleft_0 F \stackrel{a}{\rightarrow} F'$$

Encontrámos assim um problema, neste caso, no processo E a primeira ação que ocorre é um τ e vai ser duplicada n vezes enquanto que no processo F duplica-se a ação a n vezes.

1.5.5 Alínea e

Enunciado: How could the operacional semantics of this new combinator be changed so that the implication mentiones above holds? I.e so that $E \approx F \Rightarrow \circlearrowleft_n E \approx \circlearrowleft_n F$?

Resolução:

$$\frac{E \stackrel{\epsilon}{\Rightarrow} E'' \quad E'' \stackrel{a}{\Rightarrow} E''' \quad E''' \stackrel{\epsilon}{\Rightarrow} E'}{\circlearrowleft_0 E \stackrel{a}{\Rightarrow} E'} \qquad \frac{E \stackrel{\epsilon}{\Rightarrow} E'' \quad E'' \stackrel{a}{\Rightarrow} E''' \quad E''' \stackrel{\epsilon}{\Rightarrow} E'}{\circlearrowleft_n E \stackrel{a}{\Rightarrow} \circlearrowleft_{n-1} E}$$