Caderno de Estudo

Afonso Almeida

25 de Novembro de 2020

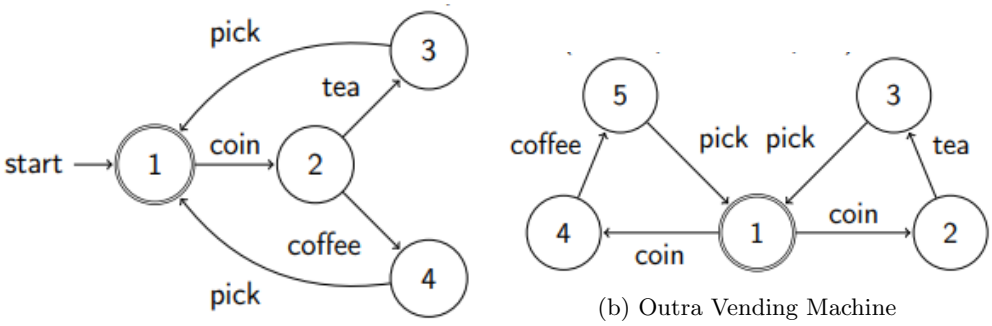
1 Aula 1

1.1 Autómatos e Expressões regulares

O desenho e modelação de programas concorrentes difere da de programas sequenciais devido aos seus diferentes propósitos. Em geral, programas sequenciais costumam executar com o objetivo de retornar ou fazer algum cálculo, enquanto que programas concorrentes são mais reativos e focam-se na interação entre vários componentes, muitas vezes correndo em loop infinito. Os programas sequenciais focam-se em **funcionalidade**, enquanto que programas concorrentes se focam no **comportamento**. É possível representar o comportamento de programas sequenciais através de expressões regulares e autómatos.

Será que o mesmo acontece para programas reativos?

Um exemplo de um sistema reativo é a ***vending machine***, que representa uma máquina de venda de bebidas que vende café ou chá (consoante o escolhido pelo utilizador).



(a) Vending Machine

(b) Outra Vending Machine

Figure 1: Duas vending machines para a mesma expressão

O utilizador começa por inserir uma moeda, escolher entre chá ou café, e apanhar a bebida. Isto também pode ser representado pela expressão regular

$$coin.(tea+coffee).pick = coin.(tea.pick + coffee.pick)$$

A *vending machine* está correta porque a linguagem do autómato é a mesma da expressão que denota o comportamento da máquina.

No entanto, é possível obter autómatos "equivalentes" a partir da mesma expressão, como é o caso da *vending machine* em (b). Neste caso, quando o utilizador insere uma moeda, o autómato tem de escolher (não-deterministicamente) se vai para o estado 2 ou 4, o que leva a que o utlizador já não consiga escolher entre café e chá, sendo a escolha feita por ele. É assim evidente que é necessária uma linguagem que suporte esta noção de equivalência, porque neste caso, dois autómatos equivalentes têm comportamento diferente. Adicionalmente, existem vários exemplos de sistemas reativos cujo comportamento os autómatos não conseguem representar corretamente:

- Serviço de streaming: é algo que nunca deve terminar, porque tem de continuamente servir os seus clientes. No entanto, um autómato tem de ter um estado final.

- VoIP: todos os participantes falam em tempo real, no entanto um autómato não consegue representar comunicação síncrona.
- Máquina de multibanco: na fase de autenticação, o utilizador tem de inserir o seu cartão e o seu pin, que são ações de input e que têm de ser respondidas com ações de output pela máquina. No entanto, usando autómatos não há distinção entre ações de input e output.

1.2 CCS - Calculus of Communicating Systems

1.2.1 Ações

Seja $\mathcal{N}$ um conjunto contável de nomes de ação. Ações em CCS podem ser definidas através de

α ::=		Ações
	a	Ação de input
	$\mid \overline{a}$	Ação de output
	$\mid \tau$	Ação interna

1.2.2 Processos

Considerando para cada variável de processo A a definição A(x_1, ..., x_n) = P em que o nome das variáveis x_1, ..., x_n ocorrem no escopo de P,

P, Q, R ::=		Processos
	0	Processo vazio
	$\mid$ A$\langle a_1, ..., a_n \rangle$	Definição de processo
	$\mid \alpha$.P	Prefixo de ação
	$\mid$ (new a)P	Esconder ação
	$\mid$ P|Q	Composição paralela
	$\mid$ P + Q	Escolha (não-determinista)

É de notar que

1. Esconder uma ação **(new a)P** faz com que a mesma fique reduzida ao escopo de P, isto é, apenas é visível em P. Para além disso, renomear uma ação interna resulta numa ação interna.
2. O prefixo de ação significa que o processo P=$\alpha.P_1$ pode realizar uma ação α e continuar como o processo P_1.
3. Em composição paralela, $P_1|P_2$, os processos P_1 e P_2 existem em simultâneo.

Usando CCS para reescrever a *vending machine*, obtém-se:

- System = VM | Client
- VM = coin.(tea.$\overline{pick}$.VM + coffee.$\overline{pick}$.VM)
- Client = $\overline{coin}$.($\overline{tea}$.pick.0 + $\overline{coffee}$.pick.0)

1.2.3 Conjunto de ações

As ações de um processo é um conjunto indutivamente definido pelas seguintes regras:

$$\begin{gathered}\mathrm{Act}(\mathrm{A}\langle a_1, ..., a_n\rangle) = a_1, ..., a_n \\ \mathrm{Act}(\alpha.\mathrm{P}) = \mathrm{a} \cup \mathrm{Act}(\mathrm{P}), \text{ se } \alpha{=}\mathrm{a} \text{ ou } \alpha{=}\overline{a} \\ \mathrm{Act}((\text{new a})\mathrm{P}) = \mathrm{a} \cup \mathrm{Act}(\mathrm{P}) \\ \mathrm{Act}(\mathrm{P} \mid \mathrm{Q}) = \mathrm{Act}(\mathrm{P}) \cup \mathrm{Act}(\mathrm{Q}) \\ \mathrm{Act}(\mathrm{P} + \mathrm{Q}) = \mathrm{Act}(\mathrm{P}) \cup \mathrm{Act}(\mathrm{Q})\end{gathered}$$

1.2.4 Ações livres e bounded

O conjunto de **ações livres** de um processo, i.e., visíveis por todos os processos, é o conjunto

$$\mathrm{fn}(\mathrm{P}) = \mathrm{Act}(\mathrm{P}) \backslash \mathrm{bn}(\mathrm{P})$$

O conjunto de **ações bounded** de um processo, i.e., apenas visíveis pelo mesmo, é o conjunto $\mathrm{bn}(\mathrm{P}) \subseteq \mathrm{Act}$ indutivamente definido pelas regras:

$$\begin{gathered}\mathrm{bn}(\mathrm{A}\langle a_1, ..., a_n\rangle) = \emptyset \\ \mathrm{bn}(\alpha.\mathrm{P}) = \mathrm{bn}(\mathrm{P}) \\ \mathrm{bn}((\text{new a})\mathrm{P}) = \mathrm{a} \cup \mathrm{bn}(\mathrm{P}) \\ \mathrm{bn}(\mathrm{P} \mid \mathrm{Q}) = \mathrm{bn}(\mathrm{P}) \cup \mathrm{bn}(\mathrm{Q}) \\ \mathrm{bn}(\mathrm{P} + \mathrm{Q}) = \mathrm{bn}(\mathrm{P}) \cup \mathrm{bn}(\mathrm{Q})\end{gathered}$$

1.2.5 Substituição de ações

Seja $\mathrm{P}\{\overrightarrow{a} \leftarrow \overrightarrow{b}\}$ a substituição das ocorrências das ações livres de $\overrightarrow{b}$ em P por $\overrightarrow{a}$.
Exemplo: Substituir *water* por *coffee*, e *cola* por *tea*.

$$\begin{gathered}\{\text{water} \leftarrow \text{cofee}\}\{\text{cola} \leftarrow \text{tea}\}(\text{tea}.\overline{pick}.\text{VM} + \text{coffee}.\overline{pick}.\text{VM}) \\ \{\text{water} \leftarrow \text{coffee}\}(\text{cola}.\overline{pick}.\text{VM} + \text{coffee}.\overline{pick}.\text{VM}) \\ (\text{cola}.\overline{pick}.\text{VM} + \text{water}.\overline{pick}.\text{VM})\end{gathered}$$

Por vezes é necessário renomear ações bounded para evitar conflitos

$$((\text{new a})\text{a.b.0})\{\text{a} \leftarrow \text{b}\} = (\text{new a})\text{a.a.0}$$

Isto leva a que a ação livre b seja renomeada para a, que está bound.
Para evitar isto, deve-se seguir a regra:

$$(\text{new a})\mathrm{P} = (\text{new b})\mathrm{P}\{\text{a} \leftarrow \text{b}\} \text{ se } \text{b} \notin \mathrm{bn}(\mathrm{P})$$

1.2.6 Relação de Transição

Dado um conjunto P de equações CCS que especificam um sistema, a relação de transição do sistema é definido pelo conjunto de triplos

$$\{\xrightarrow{a} \in \mathrm{P} \times \mathrm{P} \mid a \in Act\}$$

No entanto usa-se a notação $\mathrm{s} \xrightarrow{a} \mathrm{s'}$ por ser mais fácil de ler.

1.2.7 Regras SOS de CCS

Prefixação $\dfrac{}{\alpha.A \xrightarrow{\alpha} P}$ **Definição** $\dfrac{P_A\{\overrightarrow{a} \leftarrow \overrightarrow{b}\} \xrightarrow{a} P'}{A(\overrightarrow{b}) \xrightarrow{a} P'} A(\overrightarrow{a}) \stackrel{def}{=} P_A$

$$\textbf{L-Par}\ \frac{Q \xrightarrow{\alpha} Q'}{(Q|P) \xrightarrow{\alpha} (Q'|P)} \qquad \textbf{R-Par}\ \frac{P \xrightarrow{\alpha} P'}{(Q|P) \xrightarrow{\alpha} (Q|P')}$$

$$\textbf{L-Sum}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \textbf{R-Sum}\ \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\textbf{Sincronização}\ \frac{Q \xrightarrow{\alpha} Q' \quad P \xrightarrow{\overline{\alpha}} P'}{(Q|P) \xrightarrow{\tau} (Q'|P')} \qquad \textbf{Restrição}\ \frac{P \xrightarrow{a} P'}{(new\ a)P \xrightarrow{\alpha} (new\ a)P'}\ \alpha \notin \{a,\overline{a}\}$$

1.3 Aula 2

1.3.1 LTS - Labelled Transition Systems

Um LTS é um triplo (Proc, Act, $\xrightarrow{a}$) em que

- Proc é o conjunto de estados
- Act é o conjunto de ações
- $\xrightarrow{a} \subseteq$ Proc x Act x Proc é a relação de transição.

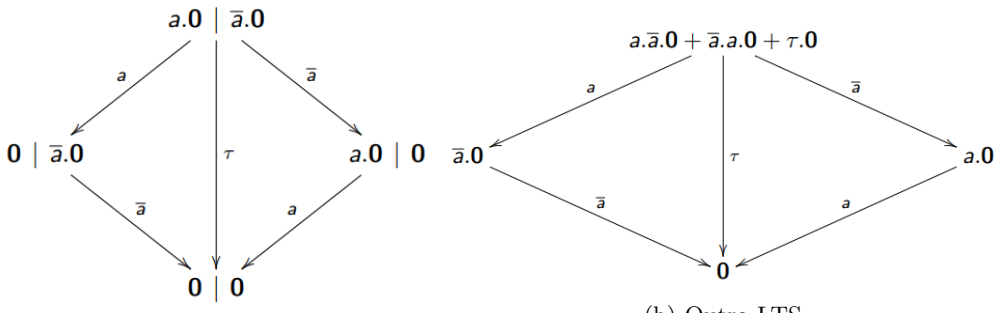
LTS são particularmente úteis porque nos dão uma ideia mais visual da progressão do sistema.

Um **pointed LTS** é um quádruplo (Proc, Act, $\xrightarrow{a}$, s). A única diferença da definição anterior é a distinção do estado inicial s, que pertence a Proc.

Um pointed LTS é **acessível** se qualquer processo do LTS for alcançável através de uma transição com uma dada ação, ou

$$\forall p \in Proc. \exists t \in Act* .s \xrightarrow{t} p$$

Daqui adiante, apenas se consideram LTS acessíveis para representar sistemas concorrentes. Em relação com CCS, qualquer termo CCS pode ser escrito sob a forma de um LTS acessível e vice-versa.



(a) Um LTS

(b) Outro LTS

Figure 2: Dois LTS equivalentes

Duas expressões CCS sintaticamente diferentes podem representar o mesmo LTS, como é o caso na Figura 2. Os LTS **(a)** e **(b)** são equivalentes porque para quaisquer dois estados (1 de cada LTS), pela mesma ação chega-se outros dois estados equivalentes.

Em rigor:

- **0** é equivalente a **0** | **0** porque no primeiro caso pode executar o processo vazio, e no segundo pode executar o processo vazio concorrentemente, isto é, não podem realizar mais nenhuma ação.
- $\overline{a}.0$ é equivalente a $0 \mid \overline{a}.0$ porque a primeira expressão há uma ação a realizar e na segunda expressão, um dos processos a correr concorrentemente chegou ao estado final, pelo que o outro processo só pode realizar $\overline{a}$ (o mesmo acontece entre a.0 e $a.0 \mid 0$)
- $a.0 \mid \overline{a}.0$ e $a.\overline{a}.0 + \overline{a}.a.0 + \tau.0$ são equivalentes porque ambos podem executar a ou $\overline{a}$ e ir para a situação acima.

A ideia principal a reter é que um observador externo a interagir com sistemas equivalentes não deve conseguir distingui-los. Para isto acontecer, todas as sequências de ações num sistemas têm de ser possíveis no outro e vice-versa.

1.3.2 Simulação

Uma relação binária $\mathcal{R} \subseteq$ Proc x Proc é uma simulação se sempre que

$$(p, q) \in \mathcal{R} \text{ então, para cada a} \in \text{Act, se p} \xrightarrow{a} \text{p' então q} \xrightarrow{a} \text{q' e (p', q')} \in \mathcal{R}$$

Exemplo: Considerando os processos P = coin.tea.pck.0 e Q = coin.(tea.pick.0 + coffee.pick.0), podemos observar que P é simulado por Q:

$$\text{P} \xrightarrow{coin} \text{tea.pick.0} \quad \text{e} \quad \text{Q} \xrightarrow{coin} \text{(tea.pick.0 + coffee.pick.0)}$$
$$\text{tea.pick.0} \xrightarrow{tea} \text{pick.0} \quad \text{e} \quad \text{(tea.pick.0 + coffee.pick.0)} \xrightarrow{tea} \text{pick.0}$$
$$\text{pick.0} \xrightarrow{pick} 0 \quad \text{e} \quad \text{pick.0} \xrightarrow{pick} 0$$

No entanto, P não simula Q:

$$\text{Q} \xrightarrow{coin} \text{(tea.pick.0 + coffee.pick.0)} \quad \text{e} \quad \text{P} \xrightarrow{coin} \text{tea.pick.0}$$
$$\text{(tea.pick.0 + coffee.pick.0)} \xrightarrow{coffee} \text{pick.0} \quad \text{mas} \quad \text{tea.pick.0} \xrightarrow{\not\ coffee}$$

Assim, Simulação não é suficiente para definir uma relação de equivalência entre dois processos, porque apenas existe a ideia que um processo tem de simular o outro, e não que se tenham de simular mutuamente. No entanto, serve como base para a Bisimulação.

1.3.3 Bisimulação

Uma relação binária $\mathcal{R} \subseteq$ Proc x Proc é uma **bisimulação forte** se sempre que (p, q) $\in \mathcal{R}$ então, para cada a $\in$ Act,

1. Se p $\xrightarrow{a}$ p' então q $\xrightarrow{a}$ q' e (p', q') $\in \mathcal{R}$
2. Se q $\xrightarrow{a}$ q' então p $\xrightarrow{a}$ p' e (p', q') $\in \mathcal{R}$

Bisimulação forte: dois processos p e q são fortemente bisimilares ($p \sim q$) se existir uma bisimulação forte $\mathcal{R}$ tal que (p,q) $\in \mathcal{R}$

Como se pode provar que dois processos são bisimilares? Basta apresentar uma bisimulação, como por exemplo:

$$\text{P} = \text{coin.P'} \quad \text{e} \quad \text{P'} = \text{tea.P"} \text{ e P"} = \text{pick.0}$$
$$\text{Q} = \text{coin.Q'} \quad \text{e} \quad \text{Q'} = \text{(tea.Q" + tea.Q"')} \text{ em que}$$
$$\text{Q"} = \text{pick.0} \quad \text{e} \quad \text{Q"'} = \text{pick.(0} \mid \text{0)}$$

A relação binária {(P, Q), (P', Q'), (P", Q"), (P", Q"'), (0, 0), (0, 0 | 0)} é uma bisimulação.

Por outro lado, como se pode provar que dois processos não são bisimilares? Uma forma simples é através do **jogo de bisimulação**.

A ideia é que a bisimulação pode ser encarada como um jogo entre dois jogados, um atacante e um defesa. O atacante joga primeiro e pode escolher qualquer transição válida, α, a partir de (p,q):

$$(\text{p, q}) \xrightarrow{\alpha} (\text{p', q}) \text{ ou } (\text{p, q}) \xrightarrow{\alpha} (\text{p, q'})$$

O defesa tem de tentar transitar pela mesma ação α a partir de (p', q) ou (p, q'), dependendo da jogada do atacante:

$$(\text{p', q}) \xrightarrow{\alpha} (\text{p', q'}) \text{ ou } (\text{p, q'}) \xrightarrow{\alpha} (\text{p', q'})$$

O atacante e o defesa continuam a jogar até:

- O defesa não conseguir fazer nenhuma transição que corresponda à transição do atacante. Neste caso o atacante ganha.
- O jogo chega a estados (p, q) vazios (em que não há transições). Neste caso o defesa ganha.
- O jogo continua para sempre. Neste caso o defesa ganha.
- O jogo chega a estados (p, q) que já foram visitados. Neste caso, o defesa ganha.

Assim, o sistema é uma bisimulação se e só se houver uma estratégia para o defesa.

Exemplo: Seja P = a.(b.c.0 + b.d.0) e Q = a.b.c.0 + a.b.d.0

1. O atacante começa por transitar de P para b.c.0 + b.d.0 por a.
2. O defesa responde ao transitar de Q para b.c.0 por a.
3. O atacante transita de b.c.0 + b.d.0 para d.0 por b.
4. O defesa responde ao transitar de b.c.0 para c.0 por b.
5. O atacante transita de d.0 para 0 por d.
6. O defesa não tem resposta e perde o jogo.

Como o defesa perdeu, provou-se que P e Q não são fortemente bisimilares.

1.4 Aula 3

1.4.1 Value-passing CCS

Até agora, apenas se apresentou o CCS "puro" em que a comunicação é usada exclusivamente para a sincronização de processos. No entanto, em algumas aplicações os processos trocam dados quando comunicam. Para representar tal, é necessário estender o CCS de modo a que haja os meios necessários para haver troca de dados.

1. Outputs podem enviar valores, em que a expressão e corresponde ao valor v, denotado e $\Downarrow$ v:

$$\overline{a}\langle e\rangle.0 \xrightarrow{\overline{a}\langle v\rangle} 0$$

2. Os inputs correspondentes têm um parâmetro formal:

$$\text{a(x).P} \xrightarrow{a(x)} \text{P}$$

3. A comunicação acontece da seguinte forma:

$$\overline{a}\langle e\rangle.0 \mid \text{a(x).P} \xrightarrow{\tau} 0 \mid \text{P } \{\text{v/x}\}$$

4. Para representar decisões, também se pode ter um processo condicional (executa P ou Q, dependendo do valor booleano de e)

$$\text{if (e) P else Q}$$

Exemplos:

- **One place buffer:** Cell = in(x).$\overline{out}\langle x\rangle$.Cell
- **Natural addition** (Parâmetros x e y, nome da resposta r)
  Sum(x,y,r) = if(y = 0) $\overline{r}\langle x\rangle$ else Sum(x + 1, y - 1, r)

Para demonstrar o uso desta extensão, apresenta-se o **ABP (Alternating-Bit Protocol)**. Este protocolo de comunicação assumes que a rede pode perder mensagens e o seu propósito é gerir a retransmissão das mensagens perdidas. Para isso:

1. As mensagens têm conteúdo e um bit extra.
2. As mensagens são enviadas repetidamente (com o mesmo bit) até que um ackonowledgement com o mesmo bit seja recebido.
3. Na comunicação estão envolvidos um Emissor (S), um Recetor (R) e o meio de comunicação (M).

Assumindo que inicialmente não há mensagens em circulação em M, o protocolo funciona da seguinte forma:

1. O Emissor envia a mensagem repetidamente (e com o mesmo bit) até receber um acknowledgement com o mesmo bit por parte do Recetor. Quando tal acontece, o Emissor complementa o bit (se era 0 passa a 1, e vice-versa) e começa a enviar a nova mensagem com este bit.
2. Quando o Recetor recebe a mensagem, envia um acknowledgement com o bit que vinha na mensagem.

Ignorando o conteúdo das mensagens, podemos definir o protocolo como:

$$\textbf{System(b)} = (\text{new ack, rec, reply, send})(\text{Sender}\langle b\rangle \mid \text{Medium} \mid \text{Receiver}\langle b\rangle)$$

em que **b** é o bit atual a ser utilizado e

$$\textbf{Sender(b)} = \text{accept.Sending}\langle b+1\rangle + \text{ack(x).Sender}\langle b\rangle$$

$$\textbf{Sending(b)} = \overline{send}\langle b\rangle\text{.Sending}\langle b\rangle + \text{ack(x).if(x=b) Sender}\langle b\rangle \text{ else Sending}\langle b\rangle$$

$$\textbf{Receiver(b)} = \text{rec(x).if(x = b+1) Received}\langle x, b+1\rangle \text{ else Received}\langle b\rangle$$

$$\textbf{Received(x,b)} = \overline{reply}\langle b\rangle\text{.Received}\langle x, b\rangle + \text{rec(x).if(x = b) } \overline{deliver}\text{.Receiver}\langle b\rangle \text{ else Received}\langle x, b\rangle$$

O Medium liga o Emissor e o Recetor, mas pode perder algumas mensagens:

$$\textbf{Medium} = \text{StoR} \mid \text{RtoS}$$

$$\textbf{StoR} = \text{send(x).}(\overline{rec}\langle x\rangle\text{.StoR} + \tau\text{.StoR})$$

$$\textbf{RtoS} = \text{reply(x).}(\overline{ack}\langle x\rangle\text{.RtoS} + \tau\text{.RtoS})$$

O estado inicial do sistema tem o mesmo comportamento do *one place buffer*, na medida em que se "lembra" do último bit usado e está pronto a aceitar uma nova mensagem, enquanto rejeita acknowledgements antigos. O sistema ideal, não considerando o conteúdo da mensagem, devia ser

$$\textbf{Spec} = \text{accept}.\overline{deliver}.\text{Spec}$$

Para provar que a implementação do ABP está correta, System(b) e Spec deviam ser equivalentes para qualquer b, isto é, System(b) $\sim$ Spec .

Através do jogo de bisimulação anteriormente apresentado,

1. System(b) transita por *accept* para (new ack, rec, reply, send)(Sending$\langle b+1\rangle$ | Medium | Receiver$\langle b\rangle$)
2. Spec transita por *accept* para $\overline{deliver}$.Spec
3. System(b) pode agora fazer um sequência (possivelmente infinita) de transições τ, e depois fazer $\overline{deliver}$
4. Spec apenas consegue fazer $\overline{deliver}$

O atacante ganha, pelo que os processos não são bisimilares. No entanto, o problema é que a relação de bisimilaridade descarta demasiado: as ações internas de System não são relevantes porque não são observáveis por um observador externo, logo são descartadas. Por isso, é preciso outra relação de equivalência que não abstraia ações internas.

1.4.2 Transição fraca

Seja (Proc, Act, $\{\xrightarrow{a} \mid a \in \text{Act}\}$) um LTS tal que $\tau \in$ Act.

$$\overset{a}{\Longrightarrow} = \begin{cases} (\xrightarrow{\tau}) * \circ \xrightarrow{a} \circ (\xrightarrow{\tau}) *, \text{ se a} \neq \tau \\ (\xrightarrow{\tau}) *, \text{ caso contrário} \end{cases}$$

- p $\overset{\tau}{\Longrightarrow}$ q denota a transição de p para q por zero ou mais ações internas.
- Se a $\neq \tau$ então p $\overset{a}{\Longrightarrow}$ q denota a transição de p para q por:
  1. Zero ou mais ações internas, seguidas de
  2. Uma transição (forte) a, seguida de
  3. Zero ou mais ações internas

1.4.3 Simulação fraca

Uma relação binária $\mathcal{R} \subseteq$ Proc x Proc é uma **simulação fraca** se sempre que (p, q) $\in \mathcal{R}$ então, para cada a $\in$ Act,

$$\text{se p} \xrightarrow{a} \text{p' então q} \overset{a}{\Longrightarrow} \text{q' e (p', q')} \in \mathcal{R}$$

1.4.4 Bisimulação fraca

Dois processos p e q são fracamente bisimilares (p $\approx$ q) se existir uma bisimulação fraca $\mathcal{R}$ tal que (p,q) $\in \mathcal{R}$

A bisimulação fraca é preservada pela prefixação, composição paralela e restrição, mas não pela escolha. Considerando os processos τ.a.0 e a.0

- τ.a.0 $\approx$ a.0 porque a.0 pode corresponder a transição τ de τ.a.0 com outro τ (i.e. nenhuma transição, fica no mesmo estado)
- No entanto, τ.a.0 + b.0 $\not\approx$ a.0 + b.0 porque quando τ.a.0 + b.0 transita por τ está a comprometer-se a futuramente realizar a ação a, enquanto que a.0 + b.0 pode corresponder com nenhuma ação e ainda escolher entre transitar por a ou b.

De forma a provar que dois processos são fracamente bisimilares, pode-se usar o **jogo de bisimulação** anteriormente referido, mas com algumas diferenças: o defesa pode adicionalmente responder usando transições fracas $\overset{\alpha}{\Longrightarrow}$ em vez de apenas transições fortes $\xrightarrow{\alpha}$ como no jogo de bisimulação forte. No entanto, o atacante ainda só pode usar transições fortes.

1.5 Aula 4

Até agora, propôs-se uma linguagem para definir sistemas concorrentes e uma noção de equivalência de forma a comparar sistemas com o mesmo comportamento. No entanto, ainda é necessário uma lógica para especificar propriedades comportamentais. Por exemplo, considerando o **one-place buffer** apresentado anteriormente:

- Como se pode garantir que depois de um *in* há sempre um *out*?
- Como se pode garantir que depois não pode haver um *in* (ou *out*) depois de um *in* (ou *out*)?

Para tal, é preciso definir uma linguagem com uma sintaxe formal e semântica, de forma a que computadores e algoritmos consigam avaliar se um processo satisfaz uma dada propriedade. Para além disso, permite definir as propriedades de uma precisão que as linguagens naturais não conseguem. Um exemplo é, ao dizer que tem de haver um *out* depois do *in*, isso aplica-se em todos os casos? Será que pode haver passos internos intermédios?

O que se pretende expressar é que, num dado momento, um sistema:

- ***pode*** fazer algo (e depois continuar com outro comportamento)
- ***tem de*** fazer algo (e depois continuar com outro comportamento)

Portanto, pretende-se que haja sequências de ações ***possíveis*** e de ações ***necessárias***.

1.5.1 Hennessy-Milner Logic

Sintaxe
Seja a $\in$ Act. O conjunto $\mathcal{F}$ de fórmulas modais é indutivamente definida pela seguinte gramática:

$$\varphi ::= \bot \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \langle a\rangle\varphi$$

Significado:

- $\bot$ denota absurdidade (impossibilidade)
- $\langle a\rangle$ denota a possibilidade de executar a ação a

Semântica
Seja $\models \subseteq Proc$ x $\mathcal{F}$ uma relação de satisfação, indutivamente definida pelas regras abaixo:

- P $\not\models \bot$
- P $\models \neg\varphi$ se P $\not\models \varphi$
- P $\models \varphi \wedge \psi$ se P $\models \varphi$ e P $\models \psi$
- P $\models \langle a\rangle\varphi$ se $\exists$ Q.P $\xrightarrow{a}$ Q e Q $\models \varphi$

Abreviações:

- $\top$ abrevia $\neg\bot$
- $\varphi \vee \psi$ abrevia $\neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \rightarrow \psi$ abrevia $\neg\varphi \vee \psi$
- [a]φ abrevia $\neg\langle a\rangle\neg\varphi$

Considere-se um Semáforo a controlar o acesso em exclusão mútua a um recurso *crit*, e três processos a tentar ganhar acesso ao recurso.

$$\text{Sem} = \text{get.crit.put.Sem}$$
$$\text{Prc}_i = \overline{get}.\overline{crit}.\overline{put}$$
$$\text{System} = (\text{new get, put})(\text{Sem} \mid \text{Prc}_1 \mid \text{Prc}_2 \mid \text{Prc}_3)$$

- Sem $\models \langle \text{get} \rangle \top$ significa que Sem **pode** realizar *get*
- Sem $\models [\text{put}] \bot$ significa que Sem **não pode** realizar *put*
- System $\models \langle \text{crit} \rangle \top$ significa que System tem de realizar uma ação interna para libertar *crit*

1.5.2 Abreviações e padrões

Seja a, b $\in$ Act e $\mathcal{A} \subseteq$ Act.

- $\langle \text{a,b} \rangle \varphi$ abrevia $\langle \text{a} \rangle \varphi \wedge \langle \text{b} \rangle \varphi$
- $\langle \mathcal{A} \rangle \varphi$ abrevia $\langle a \rangle \varphi$, para todo o a $\in \mathcal{A}$
- $\langle -a \rangle$ abrevia $\langle \text{c} \rangle \varphi$, para todo o c $\in$ Act\{a}
- $\langle -\mathcal{A} \rangle$ abrevia $\langle \text{a} \rangle \varphi$, para todo o a $\in$ Act\ $\mathcal{A}$
- $\langle - \rangle$ abrevia $\langle \text{a} \rangle \varphi$, para todo o a $\in$ Act\ $\emptyset$, pelo que qualquer ação pode ocorrer.
- $\langle - \rangle \top$ significa que uma ação qualquer **pode** acontecer.
- $[-] \bot$ significa que **nenhuma** ação pode acontecer.
- $\langle - \rangle \top \wedge [-a] \bot$ significa que apenas a ação a pode acontecer.
- $\langle - \rangle \top \wedge [-] \varphi$ significa que φ se mantém após um passo.

Nota: Processos terminados comportam-se como deadlocks: P $\models [-]$se P $\equiv 0$

1.5.3 Propriedades

Equivalência Lógica de Processos: processos que satisfazem as mesmas fórmulas são equivalentes.

$$\text{P} \sim_I \text{Q}, \text{ se } \forall \varphi \in \mathcal{F}.\mathcal{P} \models \varphi \text{ se e só se Q} \models \varphi$$

Equivalência Lógica de Fórmulas: fórmulas que satisfazem os mesmos processos são equivalentes.

$$\varphi \sim_I \psi, \text{ se } \forall \text{ P. P} \models \varphi \text{ se e só se P} \models \psi$$

Com isto, será que que dois processos que satisfazem as mesmas fórmulas em HML são fortemente bisimilares? O teorema de Hennessy-Milner responde a esta questão ao estabelecer uma relação entre equivalência lógica e bisimulação:

Processo Image-Finite
Um processo P é *image-finite* se e só se a coleção $\{\text{P'} \mid \text{P} \xrightarrow{a} \text{P'}\}$ for finito para cada ação *a*. Um LTS é *image-finite* se cada um dos seus estados também o for.

- **Teorema:** Seja (Proc, Act, $\{\xrightarrow{a} \mid \text{a} \in \text{Act}\}$) um LTS *image-finite*, e P e Q estados em Proc (e por isso também *image-finite*). Então P $\sim$ Q se e só se P e Q satisfizerem exatamente as mesmas fórmulas em HML.

- **Proposição:** Se P $\sim$ Q então P $\sim_I$ Q

1.5.4 Equivalência observacional em HML

Possibilidade eventual e necessidade eventual:

- P $\models$ $\langle\langle\rangle\rangle\varphi$, se $\exists$ Q.P $\stackrel{\tau}{\Rightarrow}$ Q e Q $\models \varphi$
- P $\models$ $[[]]\varphi$, se $\forall$ Q $\in$ {P'.P $\stackrel{\tau}{\Rightarrow}$ P'} . Q $\models \varphi$

Considerando $\mathcal{A} \in$ Act,

- $\langle\langle\mathcal{A}\rangle\rangle\varphi$ abrevia $\langle\langle\rangle\rangle\langle\mathcal{A}\rangle\langle\langle\rangle\rangle\varphi$
- $[[\mathcal{A}]]$ abrevia $[[]][\mathcal{A}][[]]\varphi$

Exemplos:

- $\langle\langle a_1\rangle\rangle$ ... $\langle\langle a_n\rangle\rangle\top$ representa a possibilidade de executar a sequência de ações observáveis a_1 ... a_n.
- $[[-\ ]]\bot$ representa a ausência de comportamento observável.

1.6 Aula 5

Assumindo que GOOD é uma propriedade desejada de um sistema, e BAD uma propriedade indesejada:

Lógica Temporal

- Nunca algo de mau acontece → (Strong) Safety, uma vez que para **todos** os estados alcançáveis, ¬BAD mantém-se.
- Eventualmente, algo de bom irá acontecer → (Weak) Liveness, uma vez que para **alguns** estados alcancáveis, GOOD mantém-se.

Não é possível definir estas propriedades porque as fórmulas em HML consideram um número finito de passos, sendo o número específico dado pela profundidade da fórmula. Assim, é necessário uma outra lógica para definir propriedades temporais dos processos.

1.6.1 CTL - Computation Tree Logic

Esta lógica considera o tempo como sendo em estrutura de uma árvore (porque há vários caminhos num LTS), em que o futuro é incerto e há vários caminhos no futuro, mas em que apenas um pode ser executado. A ideia principal é quantificar sob caminhos num LTS, em vez de apenas ações.

Tal como em HML, fórmulas em CTL podem ser interpretadas sob LTSs:

Um caminho num LTS **S** é uma sequência de estados, ligados entre si por transições. Se P é o estado inicial de S, então

$$\mathrm{P} \xrightarrow{a_1} \mathrm{P}_1 \xrightarrow{a_2} \mathrm{P}_2 \ldots$$

é um caminho de S se cada triplo $\mathrm{P}_{i-1} \xrightarrow{a_i} \mathrm{P}_i$ está na relação de transição de S. Os caminhos podem ser finitos ou infinitos.

Sintaxe

Seja $\mathcal{K} \in$ Act. O conjunto $\mathcal{F}$ de fórmulas é indutivamente definido pela gramática

$$\varphi ::= \top \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \langle\mathcal{K}\rangle\varphi \mid E(\varphi \cup \psi) \mid A(\varphi \cup \psi)$$

em que os operadores temporais:

- E denota que ***Existe*** um caminho p no LTS que satisfaz $\varphi \cup \psi$ se tiver um estado ψ e todos os estados anteriores satisfizerem φ

- A denota que em **todos** (All) os caminhos, $\varphi \cup \psi$ tem de ser satisfeito.

Existem ainda os operadores:

- F significa "*algures*" no futuro: $\quad F\ \varphi = \top \cup \varphi$
- G significa *sempre* no futuro: $\quad G\ \varphi = \neg(\top \cup \neg\varphi)$

Estes operadores são sempre prefixados por quantificadores de caminho.

Semântica
A relação de satisfação $\models \subseteq$ Proc x $\mathcal{F}$ é definido como em HML

- $P_0 \models E(\varphi \cup \psi)$ se para **algum** caminho $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2$ ... for o caso que $\exists$ i.(i $\geq 0 \wedge P_i \models \varphi \wedge \forall$ j.(0 $\leq$ j $<$ i $\wedge P_j \models \varphi$))
- $P_0 \models E(\varphi \cup \psi)$ se para **todos** os caminhos $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2$ ... for o caso que $\exists$ i.(i $\geq 0 \wedge P_i \models \varphi \wedge \forall$ j.(0 $\leq$ j $<$ i $\wedge P_j \models \varphi$))

Isto é, para alguns (ou todos) os caminhos, todos os estados satisfazem φ **até** (until) satisfazerem ψ.

Abreviações úteis:

- EF φ = E($\top \cup \varphi$) $\rightarrow$ "algures no futuro exite um caminho que satisfaz φ"
- AF φ = A($\top \cup \varphi$) $\rightarrow$ "algures no futuro todos os caminhos satisfazem φ"
- EG $\varphi = \neg$AF $\neg\varphi$ $\rightarrow$ "existe um caminho que para sempre satisfaz φ"
- AG $\varphi = \neg$EF $\neg\varphi$ $\rightarrow$ "no futuro todos os caminhos satisfazem φ"

Propriedades de sistemas distribuídos em CTL

- **Strong Safety:** AG $\neg$BAD
- **Strong Liveness:** AF GOOD
- **Weak Safety:** EG $\neg$BAD
- **Weak Liveness:** EF GOOD

Ausência de bloqueios
Todo o caminho é infinito: AG $\langle - \rangle \top$

Ausência de starvation
Em todo o caminho, um processo que queira entrar na região crítica eventualmente consegue: AG [acq1]AF $\langle$rel1$\rangle \top$

Exclusão mútua
Não exite nenhum caminho em que dois processos estão simultaneamente na região crítica: AG ([rel1]$\top \vee$ [rel2] $\bot$)

1.7 HML Recursivo

Uma fórmula HML apenas consegue descrever uma parte finita do comportamento inteiro de um processo, pelo que apenas se consegue descrever propriedades acerca de uma porção fixa das computações de um processo. No entanto, muitas vezes é necessário descrever propriedades que se querem que se verifiquem em computações arbitrariamente longas de um processo. Por exemplo, se se quiser descrever que um processo tem de ser sempre capaz de realizar uma certa ação, é necessário extender a lógica existente.

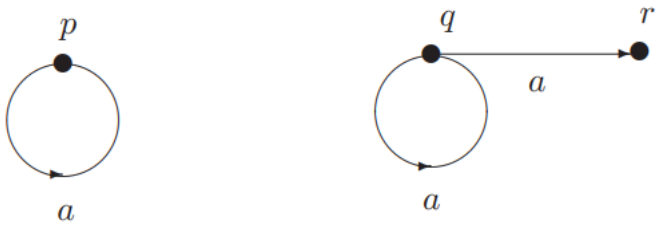


Figure 3: Processos p e q

Considerando os processos p e q, é possível observar que após realizar uma ação a, p irá sempre conseguir realizar outra ação igual, enquanto que q poderá falhar. Isto pode ser capturado formalmente em HML através de:

$$p \models [a]\langle a\rangle tt$$

Como a diferença no comportamento dos dois processos foi encontrada logo após duas transições, a fórmula que os distingue é "pequena". No entanto, assumindo que se modificava o processo q ao adicionar uma sequência de transições a r de tamanho n (com $n \geq 0$), continuava a haver uma fórmula HML que distinguia os dois processos:

$$p \models [a]^{n+1}\langle a\rangle tt$$

mas

$$q \not\models [a]^{n+1}\langle a\rangle tt$$

Não existe nenhuma fórmula em HML que funcione para todos os valores de n, o que deixa algo a desejar, porque parece existir uma razão geral pela qual os comportamentos de p e q serem diferentes. É verdade que o processo p consegue sempre (i.e. a qualquer altura em cada uma das suas computações) realizar a ação a, pelo que $\langle a\rangle tt$ é sempre verdade. Esta propriedade tem o nome de invariância $Inv(\langle a\rangle tt)$, que pode ser descrita numa extensão de HML como uma conjunção infinita:

$$Inv(\langle a\rangle tt) = \langle a\rangle tt \wedge [a]\langle a\rangle tt \wedge [a][a]\langle a\rangle tt \wedge ...$$

Esta fórmula pode ser lida como: em ordem para um processo conseguir sempre realizar uma ação a, esta ação deve ser possível **agora** (expressado por $\langle a\rangle tt$) **e, para cada inteiro positivo *i*, deve ser possível em cada estado que o processo consiga alcançar ao realizar uma sequência de ações *i*** (expressado por $[a]^i\langle a\rangle tt$, porque a é a única ação no LTS da fig.3).

No entanto, o processo q tem a opção de terminar em qualquer instante ao realizar a ação a de q para r, ou equivalentemente, é possível que q satisfaça $[a]ff$. Trata-se de uma possibilidade, à qual se chama $Pos([a]ff)$ e que pode ser expressa num extensão de HML como uma disjunção infinita:

$$Pos([a]ff) = [a]ff \vee \langle a\rangle[a]ff \vee \langle a\rangle\langle a\rangle[a]ff \vee ...$$

Esta fórmula pode ser lida como: em ordem para uma processo conseguir ter a possibilidade de recusar uma ação a num dado instante, esta ação pode ser recusada **agora** (expressado por $[a]ff$) **ou, para algum inteiro positivo *i*, deve ser possível alcançar um estado em que uma ação *a* pode ser recusada ao realizar uma sequência de ações *i*** (expressado por $\langle a\rangle^i[a]ff$, porque a é única ação no LTS da fig.3).

Extender HML com conjunções e disjunções infinitas seria algo difícil, no sentido em que seria difícil usá-las como inputs para algoritmos, por serem infinitamente compridas. Para resolver este problema, insere-se recursão na lógica. Continuando a assumir que a é a única ação, então é possível expressar $Inv(\langle a\rangle tt)$ através da equação recursiva:

$$X \equiv \langle a\rangle tt \vee [a]X$$

Esta equação captura a intuição de que um processo pode invariantemente realizar uma ação a, isto é, que um processo pode realizar uma ação a em todos os seus estados alcançáveis, pode com certeza realizar uma ação agora, e cada estado alcançado através de uma destas ações, pode invariantemente realizar uma ação a.

Visto que o propósito de uma fórmula é mostrar o conjunto de processos que a satisfazem, é natural esperar que um conjunto S de processos que satisfaçam a fórmula acima seja:

$$S = \langle a \rangle \text{Proc} \cap [a]S$$

S = $\emptyset$ é uma solução à equação porque nenhum processo consegue satisfazer $\langle a \rangle tt \wedge [a]ff$. No entanto, o processo p pode realizar uma ação a invariantemente e $p \notin \emptyset$, pelo que não é a solução que se pretende. De facto, pretende-se a **maior solução**, que é $S = \{p\}$. O conjunto $S = \emptyset$ é a **menor solução**.

Também pode ser o caso em que se pretenda obter a menor solução. Por exemplo, é possível expressar $Pos([a]ff)$ através da seguinte equação:

$$Y \equiv [a]ff \vee \langle a \rangle Y$$

Neste caso, a maior solução é $Y = \{p, q, r\}$, mas como o processo p não consegue terminar de todo, claramente não é a solução que se pretende. A menor solução é $Y = \{q, r\}$ e é exatamente o conjunto de processos que intuitivamente satisfazem $Pos([a]ff)$.

É possível definir numa equação recursiva se se está interessado na menor ou maior solução ao indicar por cima do sinal de igualdade. Para $Inv(\langle a \rangle tt)$ pretende-se a maior solução, pelo que se escreve

$$X \stackrel{max}{=} \langle a \rangle tt \wedge [a]X$$

e para $Pos([a]ff)$ pretende-se a menor solução, pelo que se escreve

$$Y \stackrel{min}{=} [a]ff \vee \langle a \rangle Y$$

Geralmente, podemos exprimir que a fórmula F se mantém para cada de um LTS, cujo conjunto de ações é Act. Esta propriedade é escrita $Inv(F)$ e lê-se "invariavelmente F" através da equação:

$$X \stackrel{max}{=} F \vee [Act]X$$

Se se quiser exprimir que a fórmula F possivelmente é satisfeita (escrita $Pos(F)$):

$$Y \stackrel{min}{=} F \wedge \langle Act \rangle Y$$

Intuitivamente, usam-se as maiores soluções nas as propriedades de um processo que se mantêm a não ser que este tenha uma sequência finita de computações que refute a propriedade. Por exemplo, o processo q não tem a propriedade $Inv(\langle a \rangle tt)$ porque este pode chegar a um estado em que nenhuma ação a pode ser realizada.

As menores soluções são usadas nas as propriedades de um processo se este tiver uma sequência finita de computações em que a propriedade seja verificada. Por exemplo, um processo tem a propriedade $Pos(\langle a \rangle tt)$ se tiver uma computação que leve a um estado que consiga realizar uma ação a, isto é, que o processo consiga realizar a ação a em algum ponto.

Existem mais exemplos do uso de fórmulas de HML recursivas. Considerando a fórmula $Safe(F)$, que é satisfeita por um processo p sempre que tiver uma sequência de transições completa

$$p = p_0 \xrightarrow{a1} p_1 \xrightarrow{a2} p_2 \ ...$$

em que cada processo p_i satisfaz F. (Uma sequência de transições é **completa** se for infinita ou se o último estado não tiver mais nenhuma transição). Esta invariância de F sob alguma computação pode ser descrita como:

$$X \stackrel{max}{=} F \wedge ([Act]ff \vee \langle Act \rangle X)$$

Intuitivamente, a fórmula acima diz que um processo p tem uma sequência de transição completa cujos estados satisfazem a fórmula F se, e só se:

- O próprio p satisfizer F e,
- Se p não tiver nenhuma transição (de saída) (e nesse caso, p irá satisfazer a fórmula $[Act]ff$), ou se p tiver uma transição que leve a um estado que tem uma sequência de transição completa, cujos estados satisfaçam F.

Um processo p satisfaz a propriedade $Even(F)$ (lida *eventualmente* F), se cada uma das suas sequências de transição completas contiverem pelo menos um estado que tenha a propriedade F. Isto significa que ou p satisfaz F, ou que p consegue realizar alguma transição e cada estado que este alcance ao realizar uma transição pode ele próprio chegar a um estado que tenha a propriedade F. Esta propriedade pode ser descrita como:

$$Y \stackrel{min}{=} F \vee (\langle Act \rangle tt \wedge [Act]Y)$$

Também é possível dizer que F deve ser satisfeito em cada sequência de transição até G ser verdade. Existem duas variantes desta ideia:

- $F \; \mathcal{U}^S \; G$, *forte até*, que diz que mais cedo ou mais tarde p chega a um estado em que G é verdade e em todos os estados que passa antes de tal acontecer, F terá de ser verificado. Tal pode ser descrito através de:

$$F \; \mathcal{U}^S \; G \stackrel{min}{=} G \vee (F \wedge \langle Act \rangle tt \wedge [Act](F \; \mathcal{U}^S \; G))$$

- $F \; \mathcal{U}^W \; G$, *fraco até*, que diz que F tem de ser verificado em todos os estados p que atravessar até chegar ao estado em que G se verifique (mas tal pode nunca acontecer). Esta propriedade pode ser descrita através de:

$$F \; \mathcal{U}^W \; G \stackrel{max}{=} G \vee (F \wedge [Act](F \; \mathcal{U}^W \; G))$$

A sintaxe para HML com uma variável X, é dada pela seguinte gramática:

$$F ::= X \mid tt \mid ff \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \langle a \rangle F \mid [a]F$$

Semanticamente, uma fórmula F (que pode conter uma variável X) é interpretada como uma função $\mathcal{O}_F : 2^{Proc} \rightarrow 2^{Proc}$ em que, dado um conjunto de processos que se **assume** que satisfazem X, retorna o conjunto de processos que satisfazem F.

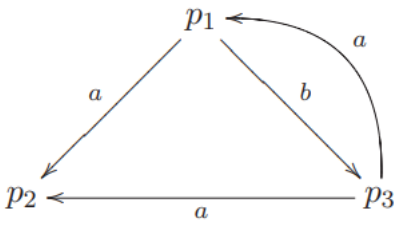


Figure 4: Exemplo de um processo

Considerando a fórmula $F = \langle a \rangle X$ e seja Proc o conjunto de estados da fig.4. Se X é satisfeito por p_1 então $\langle a \rangle X$ irá ser satisfeito por p_3, i.e., espera-se que

$$\mathcal{O}_{\langle a \rangle X}(\{p_1\}) = \{p_3\}$$