

Linguagens e Métodos Formais

José Carlos Bins Filho

28 de Fevereiro de 2014

Modelos de Computação

- Hierarquia de Chomsky

Chomsky Hierarchy	Grammar	Language	Machine
Type-0	Unrestricted	Recursively enumerable	Turing Machine
-	-	Recursive	-
Type-1	Context Sensitive	Context-Sensitive	Linear Bounded Automaton
-	Indexed	Indexed	Nested Stack Automaton
-	Tree-adjoining	Mildly Context-Sensitive	Embedded Pushdown Automaton
Type-2	Context-Free	Context-Free	Pushdown Automaton
-	Deterministic Context-free	Deterministic Context-Free	Determ. Pushdown Automaton
-	Visibly Pushdown	Visibly Pushdown	Visibly Pushdown Automaton
Type-3	Regular	Regular	Finite Automaton
-	-	Star-free Language	Aperiodic Finite State Automaton

Modelos de Computação

- Modelos de computação são modelos que definem conjuntos de operações permitidas numa computação e os custos correspondente. São usados para provar teoremas sobre computabilidade e para medir a complexidade de algoritmos.
- Existem modelos computacionais simples e genéricos.
- Os modelos simples são normalmente usados em aplicações restritas. Exemplo:
 - Expressões Regulares: Especificam padrões de strings, usados principalmente em Linguagens de programação.
 - Autômatos Finitos: Usados especialmente em desenvolvimento de circuitos.
 - Linguagens Livres de contexto: Especificam a sintaxe de linguagens de programação.
 - Existem muitos outros modelos como: Autômato de pilha, diversos tipos de gramáticas, etc...

Modelos de Computação

- Os modelos genéricos são capazes de representar qualquer tipo de computação (Tese de Church). O mais conhecido é a Máquina de Turing, mas existem outros. Os principais são:
 - Funções recursivas parciais (μ -functions). As funções primitivas recursivas estudadas são uma subclasse das μ -functions.
 - Cálculo Lambda
 - Lógica combinatória: similar ao cálculo lambda
 - Algoritmo de Markov: usa regras similares a gramáticas para operar sobre um string de símbolos
 - Máquina de Post: usa um conjunto de células e instruções para se deslocar e marcar/desmarcar as células
 - Máquina de Registradores: Idealização de um computador que usa registradores de tamanho ilimitado.
 - Sistemas de Produção: Basicamente Gramáticas de tipo-0.

Máquinas de Turing Universal

- Objetivo
 - Descrever uma máquina de Turing U, capaz de simular qualquer outra máquina de Turing M. Para isto a máquina deve conter na fita:
 - o conjunto de instruções sobre o comportamento da máquina a ser simulada;
 - o conteúdo da fita da máquina a ser simulada.
- Codificação: Para simplificar o processo se codifica uma máquina de Turing usando um alfabeto restrito = $\{0, 1\}$.
 $M = \{\{0, 1\}, \text{Fita}, \{0, 1, \epsilon\}, \text{Cabeçote}, \{q_1(\text{estado inicial}), q_2(\text{estado final}), \dots\},$
 - Codificando uma transição: $\delta(q_i, X_j) = (q_k, X_l, D_m)$
 Se assumirmos que:
 - $\{X_1, X_2, X_3\} = \{0, 1, \epsilon\}$ e
 - $\{D_1, D_2\} = \{\text{Esquerda}, \text{Direita}\}$
 então podemos codificar uma transição como: $0^i 10^j 10^k 10^l 10^m$
 e o conjunto P como: $111t_1 11t_2 11t_3 11\dots 11t_n 111$

Máquinas de Turing Universal

- Exemplo (cont):
 - Se aplicarmos a sentença 1011 à máquina teremos o 1011 ao fim do string e portanto
 $\langle M, 1011 \rangle = 111010010001010011000101010010011$
 $000100100101001100010001000100101111011$
- Linguagem Universal: Defina L_u como a linguagem $\{\langle M, w \rangle \mid M \text{ aceita } w\}$.
 L_u é chamada de universal porque o problema de um string $w \in \{0, 1\}^*$ ser ou não aceito por uma máquina de Turing específica M é equivalente ao problema de w ser aceito por uma máquina M' que contém apenas símbolos $\{0, 1, \epsilon\}$ e construída usando a codificação vista anteriormente.

Máquinas de Turing Universal

- Exemplo:
 - Dados $M = \{\{0, 1\}, \text{Fita}, \{0, 1, \epsilon\}, \text{Cabeçote}, \{q_1(\text{estado inicial}), q_2(\text{estado final}), q_3, P\}$ onde P é:

$$\begin{aligned} \delta(q_1, 1) &= (q_3, 0, D) \\ \delta(q_3, 0) &= (q_1, 1, D) \\ \delta(q_3, 1) &= (q_2, 0, D) \\ \delta(q_3, \epsilon) &= (q_3, 1, E) \end{aligned}$$
 - Máquina que inverte os bits de um string começando por 1.
 - O string que representa a máquina é dado por:

$$\begin{aligned} &111010010001010011000101010010011 \\ &00010010010100110001000100010010111 \end{aligned}$$
 - Note que este não é o único string que satisfaz a codificação da máquina dada.

Máquinas de Turing Universal

- Simulador de MT: JFlap
 - O Jflap será o simulador usado para fazer o trabalho.

Problemas

- Uma Gramática Livre de Contexto dada é ambígua?
- A questão acima é um problema?
- Aqui nós estamos interessados em questões que envolvam apenas as resposta sim ou não.
- Informalmente, um problema é uma questão contendo um ou mais parâmetros e cuja resposta é sim ou não.
- Uma lista de valores (argumentos) para os parâmetros é chamado de instância do problema.
- No caso de ambiguidade acima a instância é a CFG específica.
- Restringindo apenas a problemas com respostas sim ou não, e codificando as instâncias dos problemas como strings sobre um alfabeto finito nós podemos transformar a resolução do problema na busca de um algoritmo.

Definição

- Exemplo:
 - Problema: Não existe inteiro positivo i , com $i \geq 3$, para o qual a equação $x^i + y^i = z^i$ é válida. (Conjectura de Fermat)
 - A conjectura de Fermat resistiu mais de 300 anos e finalmente foi provada em 1994 por Andrew Wiles, portanto agora é chamada de o Último Teorema de Fermat ou Teorema de Fermat-Wiles.
 - Portanto a resposta para o problema é um algoritmo que dê como resposta sim.
 - Note:
 - Este problema tem só uma instância, $\{x, y, z, i\}$ não são parâmetros do algoritmo mas variáveis.
 - Mesmo antes da conjectura ter sido provada o problema já era decidível, pois se a conjectura fosse falsa bastava um algoritmo que respondesse não.
 - De fato não importa se a conjectura é verdadeira ou não, para ele ser decidível ou não, porque existem algoritmos que decidem o problema em ambos os casos (verdadeira - responde sempre sim; falsa - responde sempre não)

Definição

- **Definição:** Um problema é decidível se existe um algoritmo que receba como entrada uma instância do problema e determine se a resposta para aquela instancia é sim ou não.
- Neste caso a linguagem que representa o problema é recursiva.
- Caso este algoritmo não exista o problema é dito indecidível.
- Alguns autores usam a noção de semi-decidibilidade. Um problema é semi-decidível se existe um algoritmo que pára quando a resposta do problema for sim, mas pode não parar caso a resposta seja não. Neste caso a linguagem que representa o problema é recursivamente enumerável.
- Uma consequência nada intuitiva é que se o problema só possui uma instância ele é trivialmente decidível.

Resumo de decidibilidade

Ling.	Ger.	Rec.	$\bar{D} = D$	Fech.	Decidível	Indecidível
Reg.	ER	AF	Sim	$L_1 \cup L_2$ $L_1 \cap L_2$ $L_1 L_2$ L^* \bar{L}	$L_1 = L_2$ $AF_1 = AF_2$ $L = \emptyset$ $ L = \infty$ $w \in L$	
LC	GLC	AP	Não	$L_1 \cup L_2$ $L_1 L_2$ L^*	$L = \emptyset$ $ L = \infty$ $w \in L$	$L_1 = L_2$ $GLC_1 = GLC_2$ \bar{L} $L_1 \cap L_2$
Rec.	GT0	MT MPost	Sim	$L_1 \cup L_2$ $L_1 \cap L_2$ $L_1 L_2$ L^* \bar{L}		$L_1 = L_2$ $L = \emptyset$ $ L = \infty$ $w \in L$
RE	GT0	MT MPost	Sim	$L_1 \cup L_2$ $L_1 \cap L_2$ $L_1 L_2$ L^*		$L_1 = L_2$ $L = \emptyset$ $ L = \infty$ $w \in L$

Complexidade

- Taxa de crescimento:
 - Em análise de algoritmos é importante podermos estimar o uso de algum recurso (normalmente processamento e memória) em função do tamanho da entrada do algoritmo.
 - Um algoritmo pode ser eficiente para entradas pequenas mas ser irrealizável para entradas grandes.
 - Para descobrir se o algoritmo é factível temos que calcular a taxa de crescimento do algoritmo ou sua complexidade.
 - Isto é feito através da análise assintótica.

Análise Assintótica

- O cálculo é uma aproximação, o que significa que nem todos os recursos são computados, mas todos os recursos importantes, principalmente os que são recursivos, devem ser levados em conta.
 - Exemplo: for ($x=1$; $x < n$; $x++$) $f = f * x$
Quais eventos serão contados?
Qual a complexidade do problema?
E se o calculo a ser feito fosse $f = f * x * (x - 1) * (x - 2) * (x - 3)$?

Análise Assintótica

- Na análise assintótica o que importa não é o custo exato de uso de um recurso, mas como este curso se comporta quando o tamanho do problema aumenta.
- Notação Big Oh.
 - Diz-se $f(n) = O(g(n))$, significando, $f(n)$ é limitada por $g(n)$, se existe um número real k e uma constante positiva c tal que:

$$\forall n > k : |f(n)| \leq c|g(n)|$$

$g(n)$ é um dos possíveis limites superiores da função $f(n)$. Outras funções podem ser limites superiores de $f(n)$, mas em complexidade de algoritmos $g(n)$ deve ser a função com a menor taxa de crescimento que é limite superior de $f(n)$

- Ex: $f(x) = 3x^2 + 2x + 7 \Rightarrow f(x) = O(n^2)$

$$3x^2 + 2x + 7 \leq 3x^2 + 2x^2 + 7x^2 \leq 12x^2$$

$$\text{para } x \geq 1 : f(x) \leq 12x^2$$

$$\text{Logo: } f(x) = O(n^2) \text{ com } k = 1 \text{ e } c = 12$$

Classes de Complexidade

- **Definição:** Classe de complexidade é um conjunto de linguagens que possuem a mesma medida de complexidade para um determinado recurso.
- Note:
 - Para toda linguagem L pertencente a classe existe um algoritmo para o qual $\forall w, w \in L$ é decidível.
 - Os recursos mais usados para medida de complexidade são tempo de execução e memória.
 - O custo exato do recurso para os algoritmos L_1 e L_2 , ambos pertencentes a mesma classe de complexidade, pode ser diferente, mas a medida de complexidade deles é o que determina a participação deles à classe.

Classes de Complexidade

- Classe P:
 - A primeira classe importante é a classe P (de polinomial).
 - **Definição:** $P = \{L \subseteq \{0, 1\}^* \mid \text{existe um algoritmo que decide } L \text{ em tempo polinomial}\}$
Por exemplo: n^2, n^3, n^4, \dots ou $O(n^k)$
 - Note:
 - Existe uma grande diferença entre o tempo de execução de um algoritmo com complexidade $O(n^2)$ e $O(n^4)$.
Por exemplo de 10 para 100 entradas a diferença de crescimento de custo entre os dois algoritmos é de $\frac{O(n^4)}{O(n^2)} = 100 \rightarrow 10000$.
 - No entanto esta diferença é muito menor que a diferença entre algoritmos polinomiais e exponenciais, o que faz que todos os polinomiais sejam colocados em uma única classe.

Classes de Complexidade

- Classe NP:
 - Não ter sido descoberto um algoritmo polinomial para decidir um algoritmo não significa que o algoritmo não pertence a classe P.
 - Na realidade para muitos problemas que se achava que não eram polinomiais foram descobertos algoritmos polinomiais que os decidem.
 - Um problema recente (2002) foi o da composição de um número. Um número é dito composto se ele é o produto de dois outros números diferentes de 1.
 - **Definição:** NP é a classe das linguagens que tem verificadores de tempo polinomiais.
 - NP é uma classe importante pois possui muitos problemas de interesse prático.

Classes de Complexidade

- Classe NP:
 - Para alguns problemas não foram descobertos algoritmos polinomiais que os solucionem.
 - Duas possibilidades?
 - Não existem algoritmos para solucionar os problemas em tempo polinomial.
 - Existem algoritmos mas estes ainda não foram descobertos.
 - Verificabilidade Polinomial:
 - **Definição:** Um verificador V para uma linguagem L é um algoritmo tal que:
 $L = \{w \mid V \text{ aceita } \langle w, c \rangle \text{ para algum string } c\}$
 - Exemplo:
 - L : conjunto de todos os grafos direcionados
 - w : um grafo direcionado que se quer testar se é Hamiltoniano
 - c : um ciclo hamiltoniano

Note que caso c seja um ciclo Hamiltoniano então o grafo é hamiltoniano
 - Neste caso o tempo de execução da verificação é polinomial.

Classes de Complexidade

- $P \times NP$
 - P é a classe das linguagens que pode ser decidida em tempo polinomial
 - NP é a classe das linguagens que podem ser testadas em tempo polinomial
- Portanto ou $P \subset NP$ ou $P = NP$
- $P = NP$ é um dos maiores problemas matemáticos deste século.
- A maioria dos cientistas acredita que $P \neq NP$, mas não existe prova. Se $P \neq NP$ então não existe como substituir os algoritmos que usam força-bruta (teste de todas as possibilidades) para resolver um problema.

Classes de Complexidade

- NP-Completo
 - Em 1970 Cook & Levin descobriram problemas NP cuja complexidade está relacionada a da classe NP inteira.
 - Eles provaram que se existir uma solução polinomial para um destes problemas então todos os problemas NP tem solução polinomial.
 - Este tipo de problema é chamado de NP-completo
 - Portanto se for achado algum algoritmo que solucione um problema NP-completo em tempo polinomial isto significa que $P = NP$.
 - **Definição:** Uma linguagem B é NP-completa se satisfaz as condições abaixo:
 - $B \in NP$
 - $\forall A \in NP$ A é redutível a B em tempo polinomial
- Exemplo: Problema SAT (Satisfazibilidade Booleana)
 - Dada uma fórmula booleana usando apenas as operações E, OU e NÃO (\wedge, \vee, \neg), a fórmula é dita satisfazível se alguma atribuição de valores para as variáveis faz o resultado da fórmula ser verdadeiro (V ou 1).
 - A fórmula $f = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$ é satisfazível.
Para $x = 0, y = 1$ e $z = 0$.



Problemas Intratáveis e Algoritmos de Aproximação

- Exemplo de Problemas Intratáveis:
 - O Problema 3-SAT
 - Problema de satisfazibilidade de fórmulas booleanas na sua Forma Normal Conjuntiva (Produtório de Somatórios) onde cada somatório possui 3 variáveis.
Observação: Embora k-Sat, $k \geq 3$ é NP-Completo, e Sat também é NP-Completo, o problema 2-Sat é P. Existem algoritmos (Even, Itai & Shamir (1976) e Aspvall, Plass & Tarjan (1979)) que resolvem o problema em tempo linear. (Cuidado o livro texto está incorreto pois diz que todos os algoritmos k-sat são NP-Completo)
 - O problema do caixeiro viajante (TSP)
 - Encontre uma rota que parta da cidade 1, visite as demais cidades exatamente uma vez e retorne à cidade 1, tal que a distância percorrida seja mínima
 - Em outras palavras, queremos encontrar uma permutação, dentre as $n!$, que tenha distância mínima.
 - Problema da Mochila (Knapsack)
 - Dada uma lista de itens com os respectivos tamanhos e valores e uma mochila com uma dada capacidade, ache o conjunto de itens de maior valor que cabem na mochila.



Problemas Intratáveis e Algoritmos de Aproximação

- A Teoria da NP-Completo permite que o projetista concentre os seus esforços de uma forma mais produtiva ao revelar que a busca por um algoritmo eficiente para um certo problema é extremamente improvável, se não impossível
- Itens Importantes:
 - reduções são uma forma de mostrar que dois problemas são essencialmente idênticos. Um algoritmo rápido para um problema implica em um algoritmo rápido para o outro
 - Na prática um conjunto pequeno de problemas NP-Difíceis são suficientes para mostrar que outros problemas são difíceis (3-SAT, Vertex Cover, Hamiltonian Cycle)
 - Algoritmos de Aproximação garantem soluções que são próximas da solução ótima e provêm uma forma de lidarmos com problemas NP-Completo



Introdução

- Motivação:
 - Nossa dependência de sistemas de informação e comunicação está crescendo rapidamente.
 - Estes sistemas estão ficando mais e mais complexos.
 - Erros e falhas nestes sistemas são cada vez mais custosos e/ou perigosos
Exemplo:
 1. O erro na unidade de divisão de números em ponto flutuante do Pentium II custou US\$ 475 milhões.
 2. O erro de software do sistema de bagagens do aeroporto de Denver, Co, EUA, adiou a abertura do aeroporto por 9 meses a um custo de US\$ 300 milhões.
 3. O foguete Ariane-5, que caiu após 36 segundos de voo, foi causado pela conversão de um número em ponto flutuante de 64 bits para um inteiro de 16 bits.



Introdução

A confiabilidade dos sistemas de informação e comunicação é um item chave no processo de projeto de sistemas.

- Métodos Formais:
 - Métodos formais podem ser considerados como a aplicação de matemática à modelagem e análise de sistemas de informação e comunicação.
 - O seu objetivo é estabelecer a corretude dos sistemas com rigor matemático.

Introdução

- Tipos: Existem vários tipos de métodos formais, organizados sob a forma de famílias de métodos. Elas podem ser caracterizados por:
 - A teoria que embasa o método. **Exemplo:** Sistemas de transição, teoria dos conjuntos, algebra universal, λ -calculus
 - Um campo de aplicação específico. **Exemplo:** Processamento de dados, sistemas de tempo real, protocolos de comunicação
 - Uma comunidade de usuários.
- Métodos também podem ser específicos ou gerais.
 - Métodos específicos usam formalismos especializados que permitem uma especificação facilitada de determinadas aplicações, mas que muitas vezes forçam escolhas que podem tornar algumas descrições mais confusas mais adiante.
 - Métodos gerais são mais próximos da lógica e da matemática e permitem maior liberdade de expressão. Eles tem uma vantagem tecnológica principalmente quando os sistemas apresentam complexidades imprevistas.

Introdução

- Dificuldades:
 - A criação de boas especificações envolve o conhecimento profundo das necessidades do usuário. Conhecimento este, que normalmente nem o próprio usuário possui.
 - É impossível testar um software sobre todas as possibilidades de entrada (a não ser que seja um software extremamente simples). Mas uma especificação formal ajuda muito a desenvolver os testes e permite que se prove que o software satisfaz uma série de propriedades.
 - Um método formal necessita de um aprendizado inicial, a notação do método, e implica em um acréscimo significativo de tempo nas fases de especificação e projeto. No entanto, normalmente, diminui o tempo das fases de teste e integração.

Introdução

- Um método formal é composto por:
 - Uma linguagem de especificação
 - Um sistema de verificação

Introdução

- Objetivo: descrever uma tarefa sem entrar em detalhes de como a tarefa será resolvida.
- **Exemplo:** Tarefa: Procure por um elemento numa tabela

Introdução

- Pré e pós-condições
 1. T : conjunto
 2. P : predicado definido para todos elementos de T
 3. programa-de-procura que retorna
 4. x tal que $x \in T$ e $P(x)$

}	Pré-condições
}	Programa
}	Pós-condições
- Significado:

Se o programa for executado a partir de um estado que satisfaz as pré-condições, então, após a execução do programa, o estado alcançado satisfaz as pós-condições.
- **Observação:** As propriedades de T e P são **invariantes** do programa. Significando que x deve ser retornado sem alterar T e P .

Introdução

- Objetivo: descrever uma tarefa sem entrar em detalhes de como a tarefa será resolvida.
- **Exemplo:** Tarefa: Procure por um elemento numa tabela
- Perguntas:
 - Que tipo de tabela?
 - Como estão os elementos armazenados na tabela?
 - Qual elemento eu devo procurar?
 - O que fazer se eu não achar o elemento?
- Uma descrição semi-formal seria:
 1. T : conjunto
 2. P : predicado definido para todos elementos de T
 3. programa-de-procura que retorna
 4. x tal que $x \in T$ e $P(x)$

Introdução

- Corretude Funcional do programa.
 - O programa para e devolve x se $\exists x$ tal que x satisfaça P e entra em loop caso contrário. Isto é chamado de **Corretude parcial**
 - O programa sempre para e retorna x que satisfaça P . Isto é chamado de **Corretude total**, mas pode ser impossível de implementar, dependendo do predicado P .
- Enfraquecendo a pós-condição:
 - Retornando um par de valores:
 1. T : conjunto
 2. P : predicado definido para todos elementos de T
 3. programa-de-procura que retorna
 4. (b, x) com $b \in \{true, false\}$ e $\begin{cases} x \in T \wedge P(x) & \text{se } b = true \\ \forall x \in T \neg P(x) & \text{se } b = false \end{cases}$
 - **Observação:** Note que quando $b = false$ x não pertence obrigatoriamente a T .

Introdução

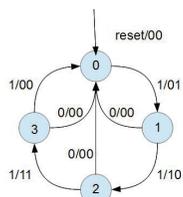
- Enfraquecendo a pós-condição (cont):
 - Usando falha:
 1. T : conjunto
 2. P : predicado definido para todos elementos de T
 3. programa-de-procura que retorna
 4. r com $r \in T + \{failure\}$ tal que $\begin{cases} P(x) & \text{se } r \in T \\ \forall x \in T \neg P(x) & \text{se } r = failure \end{cases}$
- Fortalecendo a pré-condição:
 1. T : conjunto
 2. P : predicado definido para todos elementos de T
 3. $\exists x \in T$ tal que $P(x)$
 4. programa-de-procura que retorna
 5. x tal que $x \in T$ e $P(x)$

Introdução

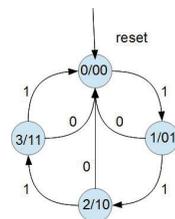
- Os métodos podem ser usados em vários níveis.
 - Nível 0: Especificação Formal: Desenvolvimento informal do programa baseado numa especificação formal.
 - Nível 1: Verificação e Desenvolvimento formais: Desenvolvimento formal e uso de verificação formal (por ferramentas ou não)
 - Nível 2: Prova de teoremas: Uso de ferramentas de verificação de forma a provar matematicamente a corretude do programa.

Usando Máquinas de Estados

- Máquinas de estados permitem modelar o comportamento de um sistema em resposta a eventos internos e externos.
- Elas mostram a resposta do sistema aos estímulos e portanto são muito usadas para modelar sistemas de tempo real.
- Máquinas de estados nada mais são que Autômatos Finitos para o qual são definitas saídas.
- Existem dois tipos:
 - Máquina de Mealy: as saídas estão associadas as transições.
 - Máquina de Moore: as saídas estão associadas aos estados.



Máquina de Mealy



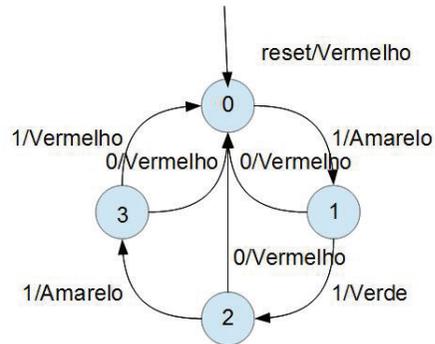
Máquina de Moore

Usando Máquinas de Estados

- **Exercícios:** 1. Modele um semáforo usando uma máquina de Mealy

Usando Máquinas de Estados

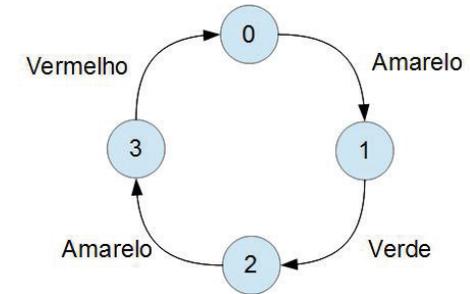
- **Exercícios:** 1. Modele um semáforo usando uma máquina de Mealy
- **Solução:** 1.



Máquina de Mealy de um farol

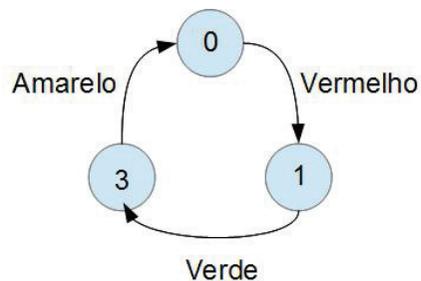
Usando Máquinas de Estados

- **Solução:** 2.

Máquina de Mealy de um farol
sem sinal de ativação

Usando Máquinas de Estados

- **Solução:** 3.

Máquina de Mealy de um farol
sem sinal de ativação
sem amarelo antes do verde

Usando Máquinas de Estados

- **Exercícios:** 2. Modele um registrador capaz de armazenar um bit usando uma máquina de Mealy

Usando Máquinas de Estados

- **Exercícios:** 2. Modele um registrador capaz de armazenar um bit usando uma máquina de Mealy
- Quais estados um registrador binário pode ter?

Usando Máquinas de Estados

- **Exercícios:** 2. Modele um registrador capaz de armazenar um bit usando uma máquina de Mealy
- Quais estados um registrador binário pode ter? $Estados = \{0, 1\}$
- Quais sinais um registrador binário pode receber?

Usando Máquinas de Estados

- **Exercícios:** 2. Modele um registrador capaz de armazenar um bit usando uma máquina de Mealy
- Quais estados um registrador binário pode ter? $Estados = \{0, 1\}$

Usando Máquinas de Estados

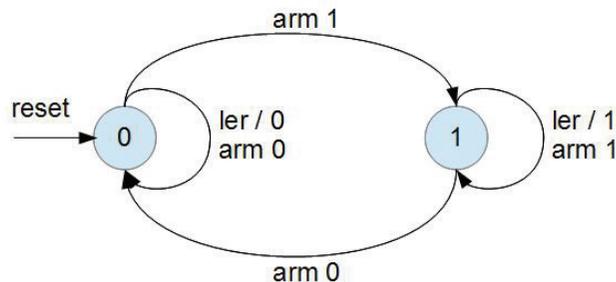
- **Exercícios:** 2. Modele um registrador capaz de armazenar um bit usando uma máquina de Mealy
- Quais estados um registrador binário pode ter? $Estados = \{0, 1\}$
- Quais sinais um registrador binário pode receber?
 $Sinais = \{reset, ler, arm\ 0, arm\ 1\}$

Usando Máquinas de Estados

- **Exercícios:** 2. Modele um registrador capaz de armazenar um bit usando uma máquina de Mealy
- Quais estados um registrador binário pode ter? $Estados = \{0, 1\}$
- Quais sinais um registrador binário pode receber?
 $Sinais = \{reset, ler, arm\ 0, arm\ 1\}$
- Quais saídas um registrador binário pode gerar?

Usando Máquinas de Estados

- **Exercícios:** 2. Modele um registrador capaz de armazenar um bit usando uma máquina de Mealy
- Quais estados um registrador binário pode ter? $Estados = \{0, 1\}$
- Quais sinais um registrador binário pode receber?
 $Sinais = \{reset, ler, arm\ 0, arm\ 1\}$
- Quais saídas um registrador binário pode gerar? $Saidas = \{0, 1\}$
- **Solução:**



Máquina de Mealy de um farol

Usando Máquinas de Estados

- **Exercícios:** 2. Modele um registrador capaz de armazenar um bit usando uma máquina de Mealy
- Quais estados um registrador binário pode ter? $Estados = \{0, 1\}$
- Quais sinais um registrador binário pode receber?
 $Sinais = \{reset, ler, arm\ 0, arm\ 1\}$
- Quais saídas um registrador binário pode gerar? $Saidas = \{0, 1\}$

Métodos Formais de Especificação de Programas

- Existem muitos métodos formais para especificação de programas.
- A lista a seguir é uma amostra do número de linguagens de especificação existentes:

Abstract State Machines (ASMs)
A Computational Logic for Applicative Common Lisp (ACL2)
ANSI/ISO C Specification Language (ACSL)
Alloy
Autonomic System Specification Language (ASSL)
B-Method
Event-B
CADP
Common Algebraic Specification Language (CASL)
Java Modeling Language (JML)
Knowledge Based Software Assistant (KBSA)

Métodos Formais de Especificação de Programas

- Continuação:

Process calculi	CSP
	LOTOS
	π -calculus
Actor model	
Esterel	
Finite Satate Process (FSP)	
Lustre	
mCRL2	
Perfect Developer	
Petri nets	
RAISE	
SPARK Ada	

FSP & LSTA

- Aqui nós vamos aprender apenas uma destas descrições: Finite State Process (FSP)
- Documentação do FSP:
<http://www.doc.ic.ac.uk/~jnm/LTSdocumentation/FSP-notation.html>
- O FSP usa uma descrição gráfica chamada LTS (Label Transition System)
- O FSP usa a ferramenta LSTA para verificar a modelagem.
 - Download do LSTA: <http://www.doc.ic.ac.uk/lsta/>

Métodos Formais de Especificação de Programas

- Continuação:

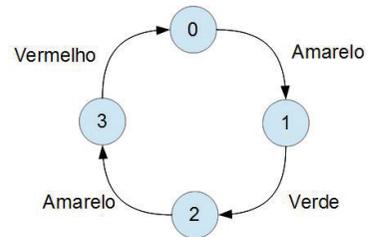
Spec sharp (Spec#)	
Specification and Description Language	
Temporal logic of actions (TLA)	
USL	
VDM	VDM-SL
	VDM++
Z notation	
Rebeca Modeling Language	

LTS

- O LTS é uma máquina de estados mas onde a transição é uma ação. Esta ação pode ser de entrada ou de saída.
- **Exemplo:** 1. Modelagem de um semáforo usando LTS

LTS

- O LTS é uma máquina de estados mas onde a transição é uma ação. Esta ação pode ser de entrada ou de saída.
- **Exemplo:** 1. Modelagem de um semáforo usando LTS



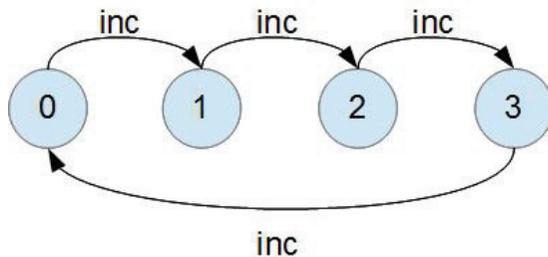
Farol usando LTS

- **Observação:** Embora o desenho seja igual a modelagem usando uma Máq. de Mealy sem ativação, a interpretação aqui é diferente. Aqui é apenas mostrada a ordem das ações e o fato do ciclo recomeçar após o vermelho.



LTS

- **Exemplo:** 1. Modelagem de um contador de dois bit usando LTS



Contador binário de 2 bits usando LTS



LTS

- **Exemplo:** 1. Modelagem de um contador de dois bit usando LTS



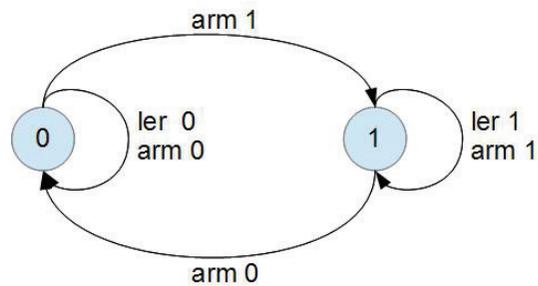
LTS

- **Exemplo:** 1. Modelagem de um registrador de um bit usando LTS



LTS

- **Exemplo:** 1. Modelagem de um registrador de um bit usando LTS



Registrador binário de 1 bit usando LTS

FSP

- Prefixo de ação ($->$):
 - $(a -> P)$: Descreve um processo que executa a ação 'a' e depois funciona conforme descrito pelo processo 'P'.
- **Exemplo:** $CLOCK = (tick -> CLOCK)$.
 - O processo CLOCK consiste em executar a ação 'tick' e depois voltar para o processo 'CLOCK'.
 - Ou seja, o processo é um processo recursivo que possui um ciclo infinito onde em cada ciclo uma ação do tipo 'tick' é executada.

FSP

- FSP é uma linguagem de descrição para especificação de sistemas concorrentes.
- A FSP gera Sistemas de Transição Marcados (LTS) que são finitos.
- Existem 2 formas de processos:
 - Processos Primitivos: que são processos sequenciais básicos (**Observação:** Sequencial não significa determinístico)
 - Processos Compostos: que são processos que chamam outros processos paralelamente, e ou redefinem processos.

FSP

- Escolha ($()$):
 - $(a -> P \mid b -> Q)$: Descreve um processo que executa uma das ações 'a' ou 'b', e caso tenha executado 'a' funciona conforme descrito pelo processo 'P' e caso tenha executado 'b' funciona conforme descrito pelo processo 'Q'. **Observação:** Aqui nada é dito sobre como será feita a escolha de qual ação executar.

Exemplo:

$$DRINKS = (black -> coffee -> DRINKS \mid white -> tea -> DRINKS)$$

- O processo DRINKS (bebidas) consiste em executar uma das ações 'black' (preto) ou 'white' (branco). Se 'black' foi a ação executada então o processo executa 'coffee' (café) e retorna para DRINKS, ou se a ação executada foi 'white' então o processo executa 'tea' (chá) e também retorna para DRINKS.
- Ou seja, o processo modela uma máquina de bebidas com dois botões, um preto, que libera café e um branco que libera chá. Quando um dos botões é apertado a ação de liberar café ou chá acontece e a máquina fica esperando um novo comando (pressão de um botão).

FSP

- Escolha (|):
 - Caso após a escolha as ações subsequentes sejam as mesmas, então pode-se juntar todas as escolhas num único conjunto, entre chaves.

Exemplo:

$$PORTA = (puxa- > solta- > PORTA | empurra- > solta- > PORTA).$$

fica:

$$PORTA = (\{puxa, empurra\}- > solta- > PORTA).$$

- O processo $PORTA^1$ permite que se abra uma porta puxando ou empurrando a mesma, mas as ações após a abertura da porta são exatamente as mesmas, portanto podem ser unidas num conjunto.

¹Este processo é um pouco diferente do processo mostrado na apostila de FSP.

FSP

- Extensão do Alfabeto:
 - Cada processo primitivo tem um alfabeto próprio que consiste em todas as ações contidas na declaração do processo.
- Exemplo:**
- $$DRINKS = (black- > coffee- > DRINKS | white- > tea- > DRINKS).$$
- O processo DRINKS tem o alfabeto {black, white, coffee, tea}
 - O alfabeto é definido implicitamente pela declaração do processo, mas em alguns casos pode ser útil aumentar explicitamente o alfabeto para impedir que outros processos utilizem as mesmas ações.

Exemplo:

$$NODRINKS = STOP + \{coffee, tea\}.$$

- O processo NODRINKS junto com o processo DRINKS gera um impasse (dead-lock), visto que o processo DRINKS não poderá executar as ações 'coffee' e 'tea' enquanto elas não estiverem liberadas no processo NODRINKS. No entanto, elas nunca estarão liberadas no processo NODRINKS, visto que o processo parou.

FSP

- Parada:
 - STOP: O processo STOP para o processo primitivo no qual ele é declarado.
 - **Observação:** Note que outros processos podem continuar executando.

Exemplo: $START = (init - > STOP).$

- O processo START executa a ação 'init' e depois para.

FSP

- Indexação:
 - A indexação permite eliminar o uso de múltiplos processos e ações assim como parametriza-los.
 - Isto pode ser feito de 2 maneiras:
 - Intervalo (range): assinala que a ação/processo recebe um dos valores do intervalo

Exemplo: $inic[x:0..10]$

 Indica que a ação 'inic' recebe um índice que varia entre os valores 0 e 10, e que o valor escolhido será colocado na variável 'x'.
 - **Observação:** Note que nada é dito como o valor é escolhido, mas apenas que um valor será escolhido.
 - Variável: quando apenas uma variável é usada como índice isto significa que anteriormente esta variável já recebeu um valor e que este valor vai ser usado como índice da ação/processo.

Exemplo: $(inic[x:0..10] - > FIM[x])$

 Neste caso o processo FIM vai receber o mesmo índice que tiver sido escolhido para a ação 'inic'.

FSP

- Declaração de intervalo e constante:
 - Para simplificar o processo pode-se declarar um intervalo e/ou constante para usar na indexação.
 - **Exemplo:** Completo de indexação. Usando constantes e intervalo.


```
const N = 10
range D = 0..10
SUM = (a[x : D] -> b[y : D] -> c[x + y] -> SUM).
```

 - O processo SUM executa a ação 'a' que recebe um índice 'x' entre 0 e 10, depois executa a ação 'b' que também recebe um índice 'y' entre 0 e 10 e depois executa a ação 'c' que terá o índice definido pela soma dos valores de 'a' e 'b'. Neste caso 'a', 'b', e 'c' funcionam como se fossem variáveis do processo. Portanto o processo recebe 'a' e 'b' e retorna a soma dos valores em 'c'.

FSP

- Guardas: Uma guarda tem a forma:
- SINTAXE: (when exp a -> P)
 - A expressão 'exp' é uma expressão similar ao C entre parênteses.
 - **Exemplo:**

```
UPDOWN = CONT[0],
CONT[x : 0..3] = (when(x < 3) inc -> CONT[x + 1] |
when(x == 3) inc -> CONT[0] |
when(x == 0) dec -> CONT[3] |
when(x > 0) dec -> CONT[x - 1]).
```

 - O Processo UPDOWN implementa um contador crescente (up) e decrescente (down).
 - A cada iteração todas as ações que passarem pelo teste da guarda podem ser executadas e uma delas será escolhida.

FSP

- Condicional: Uma condicional tem a forma:
- SINTAXE: if exp then processo-local else processo-local.
 - A expressão 'exp' é uma expressão similar ao C sem parênteses.
 - Os processos locais são processos declarados entre parênteses.
 - **Exemplo:**

```
TEST = (read[x : 0..10] ->
if x % 2 == 0
then (even -> LEVEL)
else (odds -> LEVEL)).
```

 - **Observação:** É permitido expressões lógicas com mais de uma expressão de comparação. Neste caso pode-se usar && (e) e || (ou).

FSP

- Parametrização: Processos podem ser parametrizados.
- **Exemplo:** BUF(M = 3) = (in[x:0..M] -> out[x] -> BUF).
 - O processo BUF recebe um parâmetro 'M' e usa este parâmetro no range de 'x' e como índice de 'out'.

FSP

- Composição: a composição permite criar processos usando outros processos primitivos.
- Composição paralela (\parallel):
- **Exemplo:**

$$A = (a \rightarrow x \rightarrow A).$$

$$B = (b \rightarrow x \rightarrow B).$$

$$\parallel \text{SYS} = (A \parallel B).$$

- Aqui os dois processos serão iniciados e ambas as ações iniciais serão executadas (talvez uma de cada vez).
- **Observação:** No exemplo acima, a ação 'x' é comum aos dois processos, portanto ambas as ações 'a' e 'b' devem ter sido executadas para que 'x' seja liberada para execução.

FSP

- Sincronização entre processos:
 - Quando processos compostos compartilham ações então estas ações são ditas compartilhadas.
 - Ações compartilhadas sincronizam processos porque elas são executadas ao mesmo tempo em todos os processos.
 - **Exemplo:**

$$\text{MAKER} = (make \rightarrow ready \rightarrow \text{MAKER}).$$

$$\text{USER} = (ready \rightarrow use \rightarrow \text{USER}).$$

$$\parallel \text{INTERACTION} = (\text{MAKER} \parallel \text{USER}).$$

- Aperto de mãos (hand-shake)
 - Um aperto de mão é quando um processo é reconhecido por outro. Isto pode ser implementado da seguinte maneira:
 - **Exemplo:**

$$\text{MAKER} = (make \rightarrow ready \rightarrow used \rightarrow \text{MAKER}).$$

$$\text{USER} = (ready \rightarrow use \rightarrow used \rightarrow \text{USER}).$$

$$\parallel \text{INTERACTION} = (\text{MAKER} \parallel \text{USER}).$$

FSP

- Sincronização entre processos:
 - Quando processos compostos compartilham ações então estas ações são ditas compartilhadas.
 - Ações compartilhadas sincronizam processos porque elas são executadas ao mesmo tempo em todos os processos.
 - **Exemplo:**

$$\text{MAKER} = (make \rightarrow ready \rightarrow \text{MAKER}).$$

$$\text{USER} = (ready \rightarrow use \rightarrow \text{USER}).$$

$$\parallel \text{INTERACTION} = (\text{MAKER} \parallel \text{USER}).$$

FSP

- Sincronização entre processos (cont):
 - Vários processos:
 - **Exemplo:**

$$\text{MAKER}_A = (makeA \rightarrow ready \rightarrow used \rightarrow \text{MAKER}_A).$$

$$\text{MAKER}_B = (makeB \rightarrow ready \rightarrow used \rightarrow \text{MAKER}_B).$$

$$\text{ASSEMBLER} = (ready \rightarrow assemble \rightarrow used \rightarrow \text{ASSEMBLER}).$$

$$\parallel \text{FACTORY} = (\text{MAKER}_A \parallel (\text{MAKER}_B \parallel \text{ASSEMBLER})).$$

FSP

- **Exemplo:** Modelo de carro e garagem.

$$\begin{aligned} \text{CAR}(I = 1) &= (\text{car}[I].\text{out} - > \text{car}[I].\text{enter} - > \\ &\quad \text{car}[I].\text{in} - > \text{car}[I].\text{exit} - > \text{CAR}). \\ \text{GARAGE}(N = 2) &= (\text{car}[x : 1..N].\text{enter} - > \text{car}[x].\text{exit} - > \\ &\quad \text{GARAGE}). \\ \parallel \text{SYS} &= (\text{CAR}(1) \parallel \text{CAR}(2) \parallel \text{GARAGE}). \end{aligned}$$

FSP

- Ou pode-se usar como se os processos fossem classes e a etiqueta fosse um objeto daquela classe.
- **Exemplo:** Modelo de carro e garagem usando etiquetagem (outro).

$$\begin{aligned} \text{CAR} &= (\text{out} - > \text{enter} - > \\ &\quad \text{in} - > \text{exit} - > \text{CAR}). \\ \text{GARAGE}(N = 2) &= (\text{car}[x : 1..N].\text{enter} - > \text{car}[x].\text{exit} - > \\ &\quad \text{GARAGE}). \\ \parallel \text{SYS} &= (\text{car}[1] : \text{CAR} \parallel \text{car}[2] : \text{CAR} \parallel \text{GARAGE}). \end{aligned}$$

FSP

- Etiquetagem (labeling): A FSP permite que se use etiquetas (labels) para processos.
- **Exemplo:** Modelo de carro e garagem usando etiquetagem.

$$\begin{aligned} \text{CAR}(I = 1) &= (\text{car}[I].\text{out} - > \text{car}[I].\text{enter} - > \\ &\quad \text{car}[I].\text{in} - > \text{car}[I].\text{exit} - > \text{CAR}). \\ \text{GARAGE}(N = 2) &= (\text{car}[x : 1..N].\text{enter} - > \text{car}[x].\text{exit} - > \\ &\quad \text{GARAGE}). \\ \parallel \text{CARS} &= (\text{CAR}(1) \parallel \text{CAR}(2)). \\ \parallel \text{SYS} &= (\text{CARS} \parallel \text{GARAGE}). \end{aligned}$$

FSP

- **Exemplo:** Modelo de carro e garagem usando etiquetagem e intervalo.

$$\begin{aligned} \text{CAR} &= (\text{out} - > \text{enter} - > \\ &\quad \text{in} - > \text{exit} - > \text{CAR}). \\ \text{GARAGE}(N = 2) &= (\text{car}[x : 1..N].\text{enter} - > \text{car}[x].\text{exit} - > \\ &\quad \text{GARAGE}). \\ \parallel \text{SYS}(N = 2) &= (\text{car}[1..N] : \text{CAR} \parallel \text{GARAGE}). \end{aligned}$$

FSP

- **Exercícios:** Use FSP para modelar um estacionamento com N lugares.
- Dica: Não precisa levar em conta que carro está entrando.

FSP

- **Exercícios:** 1. Use FSP para modelar um estacionamento com N lugares levando em conta que carro está entrando.

FSP

- **Exercícios:** Use FSP para modelar um estacionamento com N lugares.
- Dica: Não precisa levar em conta que carro está entrando.

$$\begin{aligned}
 INIC(N = 10) &= VAGAS[N], \\
 VAGAS[X : 0..N] &= (when(x > 0) chegada -> VAGAS[x - 1]) \\
 &\quad (when(x < N) partida -> VAGAS[x + 1]). \\
 ENT &= (chegada -> ENT). \\
 SAI &= (partida -> SAI). \\
 ||ESTACIONAMENTO &= (ENT || SAI || INIC(5)).
 \end{aligned}$$

FSP

- **Exercícios:** 1. Use FSP para modelar um estacionamento com N lugares levando em conta que carro está entrando.

$$\begin{aligned}
 const\ C &= 10 \\
 INIC(N = 10) &= VAGAS(N), \\
 VAGAS[X : 0..N] &= (when(x > 0) chegada -> VAGAS[x - 1]) \\
 &\quad (when(x < N) partida -> VAGAS[x + 1]). \\
 CARRO &= (fora -> entra -> dentro -> sai -> \\
 &\quad CARRO). \\
 ENT &= (carro[x : 1..C].entra -> chegada -> ENT). \\
 SAI &= (carro[x : 1..C].sai -> partida -> SAI). \\
 ||ESTAC &= (ENT || SAI || INIC(5)).
 \end{aligned}$$

FSP

- **Exercícios:** 2. Use FSP para modelar um jardim com capacidade de 100 pessoas onde existe uma entrada por onde as pessoas podem entrar a qualquer momento e para o qual um controlador pode a qualquer momento apertar um botão que imprime o total de pessoas presentes no jardim no momento.
- **Exercícios:** 3. Modifique o modelo do exercício anterior para:
 - 3.1 Colocar duas entradas, uma norte uma sul.
 - 3.2 Fazer com que a entrada e a escrita do total de pessoas não aconteçam ao mesmo tempo.
 - 3.3 Fazer com que a pessoa só possa entrar se houver vaga.
 - 3.4 Dar o o número de pessoas que entraram pela entrada sul separado da entrada norte (ou seja serão 3 contadores: entrada sul, entrada norte e total).

FSP

- **Solução:** 3.1, 3.2 e 3.3: Comentários
 - Coloquei 'vaga' para só liberar a entrada quando houver vaga no jardim (exercício 3.3).
 - Coloquei 'pessoa' para só liberar saída quando houverem pessoas dentro do jardim.
 - Coloquei as duas entradas (norte e sul (exercício 3.1)) num único processo porque, se há uma vaga, só um dos portões pode liberar a passagem. Se os dois liberarem vai dar problema (portanto não podem funcionar independentemente)
 - Coloquei o controle para liberar apenas uma operação por vez (exercício 3.2). Ou entra/sai (portao) uma pessoa ou dá o display (aperta o botao). Não permite que os dois funcionem ao mesmo tempo.

FSP

- **Solução:** 3.1, 3.2 e 3.3 (3.4 fica como exercício).

```

const C =      10
INIC(N = 10) = PESSOAS[0],
PESSOAS[x : 0..C] =
  (disp -> imprime[x] -> PESSOAS[x] |
   when (x < C) vaga -> entra -> entrou -> PESSOAS[x + 1] |
   when (x > 0) pessoa -> sai -> saiu -> PESSOAS[x - 1]).
ENT =
  (vaga -> {entnorte, entsul} -> entra -> ENT |
   pessoa -> {sainorte, saisul} -> sai -> ENT).
CONTROLE =
  (portao -> {vaga, pessoa} -> {entrou, saiu} -> CONTROLE |
   aberta -> disp -> CONTROLE).
||JARDIM = (CONTROLE || ENT || INIC).
  
```

FSP

- Re-etiquetagem (relabelling): Re-etiquetagem é aplicada para mudar nomes de ações de processos.
- Isto é normalmente feito para sincronizar processos com as ações corretas.
- **Exemplo:** Imagine duas funções iguais mas que são mutuamente exclusivas (quando uma está parada a outra esta funcionando).

Cada função pode ter a sua ativação definida como:

```

FUNCAO = (ativa -> desativa -> FUNCAO).
||JUNTAS = (FUNCAO || FUNCAO).
  
```

Mas neste caso elas vão funcionar sincronamente (quando uma funciona a outra também). Veja no LTSA.

O que se quer é exatamente ao contrário (que funcionem desincronizadamente). Como fazer?

FSP

- Fazer com que o sinal de desativar uma seja o mesmo para ativar a outra. Isto é feito com re-etiquetagem.
- SINTAXE: / {ação1/ação2}
 - Ação1 substitui ação2.
 - Isto pode estar dentro da declaração do processo (logo após o processo como no exemplo 1) ou depois da declaração (como no exemplo 2).

Exemplo: 1

$$\begin{aligned} FUNCAO &= (ativa \rightarrow desativa \rightarrow FUNCAO). \\ ||JUNTAS &= (FUNCAO/\{desativa/troca\} || FUNCAO/\{ativa/troca\}). \end{aligned}$$

- Isto é equivalente à:

$$\begin{aligned} FUNCAO1 &= (ativa \rightarrow troca \rightarrow FUNCAO1). \\ FUNCAO2 &= (troca \rightarrow desativa \rightarrow FUNCAO2). \\ ||JUNTAS &= (FUNCAO1 || FUNCAO2). \end{aligned}$$


FSP

- **Observação:** Note que mais de uma re-etiquetagem pode ser feita entre chaves se separadas por vírgula.

$$\begin{aligned} FUNCAO &= (ativa \rightarrow executa \rightarrow desativa \rightarrow FUNCAO). \\ ||JUNTAS(N = 4) &= (funcao[0..N - 1] : FUNCAO) \\ & / \{funcao[x : 0..N - 1].ativa/funcao[x - 1].desativa, \\ & ativa/funcao[0].ativa, \\ & desativa/funcao[N - 1].desativa\}. \end{aligned}$$


FSP

- Uma vantagem da re-etiquetagem é que ela pode ser usada para processos indexados
- **Exemplo:** 2. O exemplo abaixo permite que as funções sejam executadas em ordem.

$$\begin{aligned} FUNCAO &= (ativa \rightarrow executa \rightarrow desativa \rightarrow FUNCAO). \\ ||JUNTAS(N = 4) &= (funcao[0..N - 1] : FUNCAO) \\ & / \{funcao[x : 0..N - 1].ativa/funcao[x - 1].desativa\}. \end{aligned}$$

- **Exercícios:** 4 Compare a modelagem do processo FUNCAO nos exemplos 1 e 2.
 - 4.1. Elas são diferentes! Em que sentido ?
 - 4.2. Se voce usar a modelagem do exemplo 2 no exemplo 1 voce vai ver que vai dar "Dead lock". Teste e diga porque?
 - 4.3 Como voce pode modificar o exemplo 1 para aceitar a modelagem do processo FUNCAO conforme o exemplo 2?



FSP

- Escondendo ações: Algumas vezes é útil esconder nomes de ações do alfabeto. Estas ações continuam existindo e, por default, são chamadas de 'tau'. A vantagem é que outros processos, fora da composição, não verão as ações e portanto poderão usar o mesmo nome interno sem que isto signifique sincronizar os processos.
- SINTAXE 1: \ {lista de ações à serem escondidas}
- ou
- SINTAXE 1: @ {lista de ações à serem mostradas}
 - No primeiro caso as ações da lista serão escondidas e todas as outras serão visíveis.
 - No segundo caso as ações da lista serão visíveis e todas as outras serão escondidas.



FSP

- Primeira tentativa de modelagem:
- **Observação:** CUIDADO: esta modelagem dá "dead-lock". Porque?

$$\begin{aligned}
 FIL(I = 0) &= (esq.pegar \rightarrow dir.pegar \rightarrow come \rightarrow \\
 &\quad esq.larga \rightarrow dir.larga \rightarrow FIL). \\
 GARFO &= (pegar \rightarrow larga \rightarrow GARFO). \\
 \parallel JANTAR &= (fil[0] : FIL(0) \parallel fil[1] : FIL(1) \parallel fil[2] : FIL(2) \parallel \\
 &\quad fil[3] : FIL(3) \parallel fil[4] : FIL(4) \parallel \\
 &\quad \{fil[0].dir, fil[1].esq\} :: GARFO \parallel \\
 &\quad \{fil[1].dir, fil[2].esq\} :: GARFO \parallel \\
 &\quad \{fil[2].dir, fil[3].esq\} :: GARFO \parallel \\
 &\quad \{fil[3].dir, fil[4].esq\} :: GARFO \parallel \\
 &\quad \{fil[4].dir, fil[0].esq\} :: GARFO).
 \end{aligned}$$

FSP

- Recursão:
 - O FSP permite que se modele recursão através do uso de condições.
 - **Exemplo:**

$$\begin{aligned}
 FUNCAO &= (ativa \rightarrow desativa \rightarrow FUNCAO). \\
 \parallel JUNTAS(N = 4) &= (if N == 1 \\
 &\quad then \\
 &\quad \quad FUNCAO \\
 &\quad else \\
 &\quad \quad (FUNCAO / \{troca/desativa\} \parallel \\
 &\quad \quad \quad JUNTAS(N - 1) / \{troca/ativa\}) \\
 &\quad @\{ativa, desativa\}).
 \end{aligned}$$

FSP

- **Solução:**

$$\begin{aligned}
 FIL(I = 0) &= (when (I \% 2 == 0) esq.pegar \rightarrow dir.pegar \rightarrow \\
 &\quad come \rightarrow esq.larga \rightarrow dir.larga \rightarrow FIL \mid \\
 &\quad when (I \% 2 == 1) dir.pegar \rightarrow esq.pegar \rightarrow \\
 &\quad come \rightarrow dir.larga \rightarrow esq.larga \rightarrow FIL). \\
 GARFO &= (pegar \rightarrow larga \rightarrow GARFO). \\
 \parallel JANTAR &= (fil[0] : FIL(0) \parallel fil[1] : FIL(1) \parallel fil[2] : FIL(2) \parallel \\
 &\quad fil[3] : FIL(3) \parallel fil[4] : FIL(4) \parallel \\
 &\quad \{fil[0].dir, fil[1].esq\} :: GARFO \parallel \\
 &\quad \{fil[1].dir, fil[2].esq\} :: GARFO \parallel \\
 &\quad \{fil[2].dir, fil[3].esq\} :: GARFO \parallel \\
 &\quad \{fil[3].dir, fil[4].esq\} :: GARFO \parallel \\
 &\quad \{fil[4].dir, fil[0].esq\} :: GARFO).
 \end{aligned}$$

FSP

- Replicação de Processos:

- Quando um processo não possui etiquetas mesmo assim ele pode ser replicado usando o comando "forall".
- SINTAXE: $\parallel P2 = \text{forall } [i:N] P1.$
 - O processo P2 é definido como N copias do processo P1 em paralelo.
 - **Observação:** Embora "forall" permita o uso de processos não indexados isto não impede que ele seja usado para processos indexados.
 - **Exemplo:** As 3 declarações abaixo (C1, C2 e C3) são equivalentes.

$$\begin{aligned}
 C1 &= (a[0] : X \parallel a[1] : X \parallel a[2] : X \parallel a[3] : X). \\
 C2 &= (a[0..3] : X). \\
 C3 &= forall [i : 0..3] a[i] : X.
 \end{aligned}$$

FSP

- Propriedades de salvaguarda: garante que nada ruim aconteça
 - Propriedades de salvaguarda são processos determinísticos, sem ações ocultas, que especificam um conjunto de comportamentos aceitáveis para o sistema onde a propriedade está inclusa.
 - Com isto o sistema só pode gerar sequencias de ações que sejam aceitáveis para a propriedade.
 - **Exemplo:**

```
property POLIDO = (bater -> entrar -> POLIDO).
HESITANTE = (bater -> bater -> entrar -> HESITANTE).
IMPACIENTE = (entrar -> IMPACIENTE).
||JUNTAS1 = (HESITANTE || POLIDO).
||JUNTAS2 = (IMPACIENTE || POLIDO).
```

- **Observação:** Propriedades não restringem a operação do sistema, no entanto ações que não seguem a propriedade geram transições para um estado de erro.

FSP

- Semáforos e mutua-exclusão:
- Semáforos são processos largamente usados para tratar sincronização de procesos em sistemas operacionais.
- Um semáforo S é uma variável inteira que assume apenas valores não negativos.
- as únicas operações num semáforo são incrementação (up) e decretação (down).
- Os algoritmos abaixo mostram as operações destas ações:

```
down(s): if s > 0 then
           decreta s
         else
           bloqueia execução do processo
```

FSP

- Garantindo que uma ação nunca aconteça:
- **Exemplo:**

```
property NUNCA = STOP + {desastre}
```

FSP

- Semáforos e mutua-exclusão (cont):

```
up(s): if existe processo bloqueado no s then
        acorda um deles
      else
        incrementa s
```

- A definição em FSP fica:

```
const Max = 3
range Int = 0..Max
SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int] = (up -> SEMA[v + 1] |
               when (v > 0) down -> SEMA[v - 1]),
SEMA[MAX + 1] = ERROR.
```

FSP

- Semáforos e mutua-exclusão (cont):
- **Exemplo:** Três processos usando um acesso a um recurso usando mutua-exclusão.

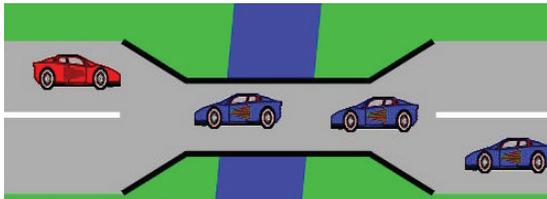
$$LOOP = (mutex.down \rightarrow pega \rightarrow devolve \rightarrow mutex.up \rightarrow LOOP).$$

$$\| DEMO = (p[1..3] : LOOP \| \{p[1..3]\} :: mutex : SEMAPHORE(1)).$$

- Porque SEMAPHORE foi inicializado com 1?
- **Observação:** 1. O ":" se usa quando em vez de uma etiqueta, se tem um conjunto de etiquetas. No caso, as etiquetas p[1], p[2] e p[3].
- **Observação:** 2. A gramática da sintaxe do FSP pode ser vista no site <http://www.doc.ic.ac.uk/~jnm/LTSdocumentation/FSP-Syntax.html>

FSP

- **Exercícios:** Modele uma ponte de uma única pista numa rodovia de 2 pistas. Ou seja, na ponte só podem passar carros de uma direção da rodovia de cada vez. Veja figura abaixo:



- Restrições:
 - Uma vez que passe um carro de um lado, os carros daquele lado continuarão a passar até que todos tenham passado.
 - Carros não podem fazer ultrapassagens na ponte.

FSP

- Semáforos e mutua-exclusão (cont):
- Como testar que o 'mutex' e SEMAFORO garantem exclusão mútua do recurso?

$$\begin{aligned} \text{property } MUTEX &= (p[j : 1..3].pega \rightarrow p[j].devolve \rightarrow MUTEX). \\ \| TESTE &= (DEMO \| MUTEX). \end{aligned}$$

- O que acontece quando SEMAPHORE é inicializado com 2?

FSP

- Dicas: Modelem os seguintes processos:
 - Carro
 - Comboio (grupo de carros de um mesmo lado)
 - Impedimento de ultrapassar
 - Ponte
 - Propriedade que uma vez que entre alguém de um comboio apenas carros daquele comboio podem entrar e/ou sair até que todos tenham passado.

FSP

- Propriedades de "vitalidade" (liveness):
 - Propriedades (de vitalidade) são garantias de que algo que se deseja que ocorra vai eventualmente acontecer.
 - Podem ser:
 - Propriedade de Progresso.
 - Prioridade das ações.
 - Lógica Temporal.

FSP

- Propriedades de progresso: garantem que um conjunto de ações aconteça infinitas vezes
 - Propriedades de progresso são um conjunto de ações onde, pelo menos, uma delas deve ser executada pelo processo infinitas vezes.
 - Ou seja, o processo não pode passar por um estado a partir do qual uma ação colocada na propriedade nunca mais ocorra.
 - **Exemplo:**

$$\begin{aligned}
 \text{DUASMOEDAS} &= (\text{escolha} \rightarrow \text{MOEDAFALSA} \mid \\
 &\quad \text{escolha} \rightarrow \text{MOEDA}), \\
 \text{MOEDA} &= (\text{joga} \rightarrow \text{cara} \rightarrow \text{MOEDA} \mid \\
 &\quad \text{joga} \rightarrow \text{coroa} \rightarrow \text{MOEDA}), \\
 \text{MOEDAFALSA} &= (\text{joga} \rightarrow \text{cara} \rightarrow \text{MOEDAFALSA}). \\
 \text{progress CARA} &= \{\text{cara}\} \\
 \text{progress COROA} &= \{\text{coroa}\}
 \end{aligned}$$

- Qual seria a propriedade de progresso que se poderia colocar no exercício do Jantar dos Filósofos?
- E no exercício da Ponte de uma só pista?