APRESENTAÇÃO DA DISCIPLINA

Alexandre Mota & Augusto Sampaio

Motivação

- Computadores multicore e arquiteraturas paralelas estão cada vez mais presentes em nosso dia-a-dia
- Software está cada vez mais complexo (“pesado”)
- Explorar paralelismo induz em ganho de performance
- Linguagens de programação mais difundidas (Java, C++) são difíceis para explorar paralelismo
- Testar aplicações concorrentes/paralelas é muito mais complicado

Motivação

- Comunicação entre aplicações (processos) concorrentes se dá por troca de mensagens ou compartilhamento de memória
  - Compartilhamento de memória é leve mas baixo nível
  - Troca de mensagens é alto nível mas pesado
- CSP é uma linguagem de especificação formal projetada para modelar aplicações concorrentes, paralelas e distribuídas

Motivação

- Com CSP é possível ir além de testes (verificação de modelos-model checking)
- CSP é baseada em troca de mensagens, mas é possível simular compartilhamento de memória
- É possível evoluir de troca de mensagens para compartilhamento de memória
- Obtém-se aplicação leve (compartilhamento de memória) a partir de modelo seguro (troca de mensagens)

Problema

- Dado o seguinte programa:

```
int x, y=0, z=0;
co
    x=y+z;
  //
    y=1; z=2;
oc;
```

- Quais valores x pode assumir? Por que isto ocorre?

Problema

- Dado o seguinte cenário:

```
co
   1A. Solicita acesso exclusivo ao disco 1 (bloqueia);
   2A. Solicita acesso exclusivo ao disco 2 (bloqueia);
   3A. Libera disco 2 e depois disco 1;
   Volta a 1A;
//
   1B. Solicita acesso exclusivo ao disco 2 (bloqueia);
   2B. Solicita acesso exclusivo ao disco 1 (bloqueia);
   3B. Libera disco 1 e depois disco 2;
   Volta a 1B;
oc;
```

- Que problema pode ocorrer?

Solução

- Para troca de mensagens
  - Garantir ausência de seqüência indesejável
- Para memória compartilhada
  - Garantir exclusão mútua na seção crítica

TROCA DE MENSAGENS

CSP

- Notação conveniente para modelar sistemas concorrentes
- Alto nível de abstração
- Ferramentas para analisar propriedades automaticamente
- Possui biblioteca para Java (JCSP)

Jantar dos Filósofos em CSP

$$N = 5$$
$$PHILNAMES = \{0..N-1\}$$
$$FORKNAMES = \{0..N-1\}$$

$$channel\ sits, eats, thinks, getsup : PHILNAMES$$
$$channel\ picks, putsdown : PHILNAMES.FORKNAMES$$

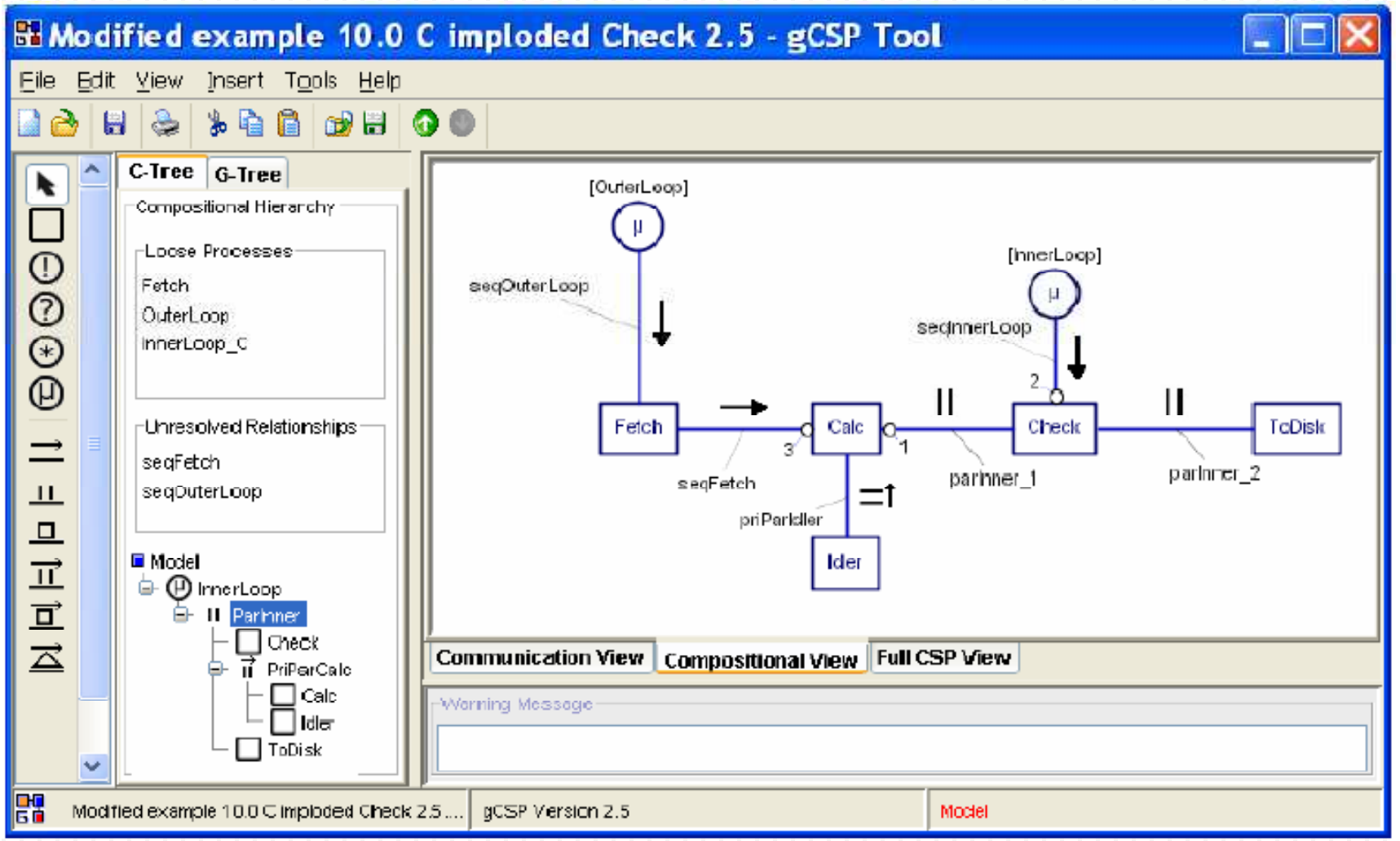
$$PHIL(i) = thinks!i \rightarrow sits!i \rightarrow picks!i!i \rightarrow picks!i!((i+1)\%N) \rightarrow$$
$$eats!i \rightarrow putsdown!i!((i+1)\%N) \rightarrow$$
$$putsdown!i!i \rightarrow getsup!i \rightarrow PHIL(i)$$

$$FORK(i) = picks!i!i \rightarrow putsdown!i!i \rightarrow FORK(i)$$
$$\Box\ picks!((i-1)\%N)!i \rightarrow putsdown!((i-1)\%N)!i \rightarrow FORK(i)$$

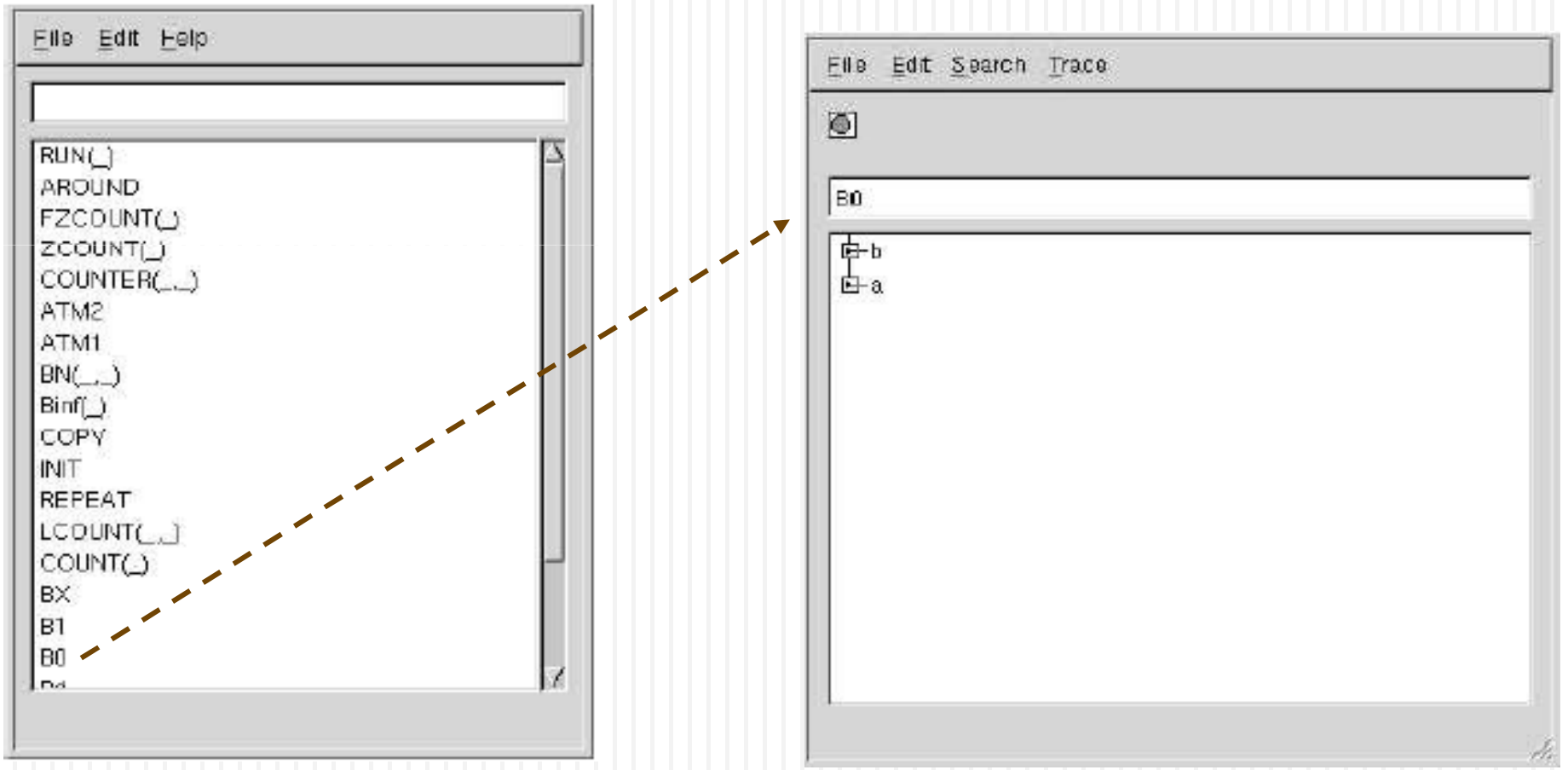
$$PHILS = ||| i : PHILNAMES \bullet PHIL(i)$$
$$FORKS = ||| i : FORKNAMES \bullet FORK(i)$$

$$SYSTEM = PHILS[|\ \{|\ picks, putsdown\ |\}\ |]FORKS$$

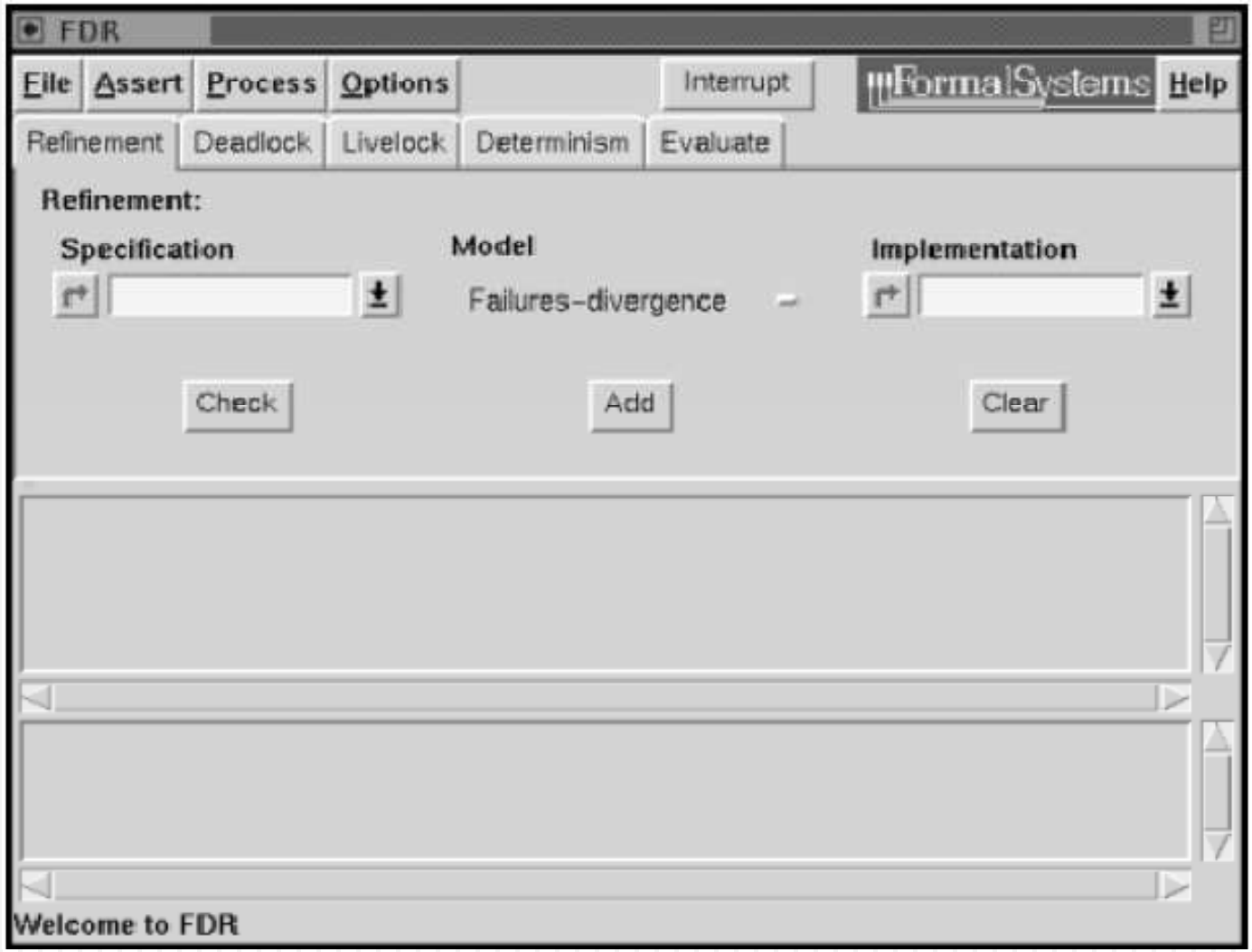
gCSP


Modified example 10.0 C imploded Check 2.5 - gCSP Tool
File Edit View Insert Tools Help
C-Tree
G-Tree
Compositional Hierarchy
Loose Processes
Fetch
OuterLoop
InnerLoop_C
Unresolved Relationships
seqFetch
seqOuterLoop
Model
InnerLoop
Parinner
Check
PriParCalc
Calc
Idler
ToDisk
[OuterLoop]
seqOuterLoop
[InnerLoop]
seqInnerLoop
Fetch
Calc
Check
ToDisk
seqFetch
priParIdler
Idler
parInner_1
parInner_2
Communication View
Compositional View
Full CSP View
Warning Message
Modified example 10.0 C imploded Check 2.5 ...
gCSP Version 2.5
Model

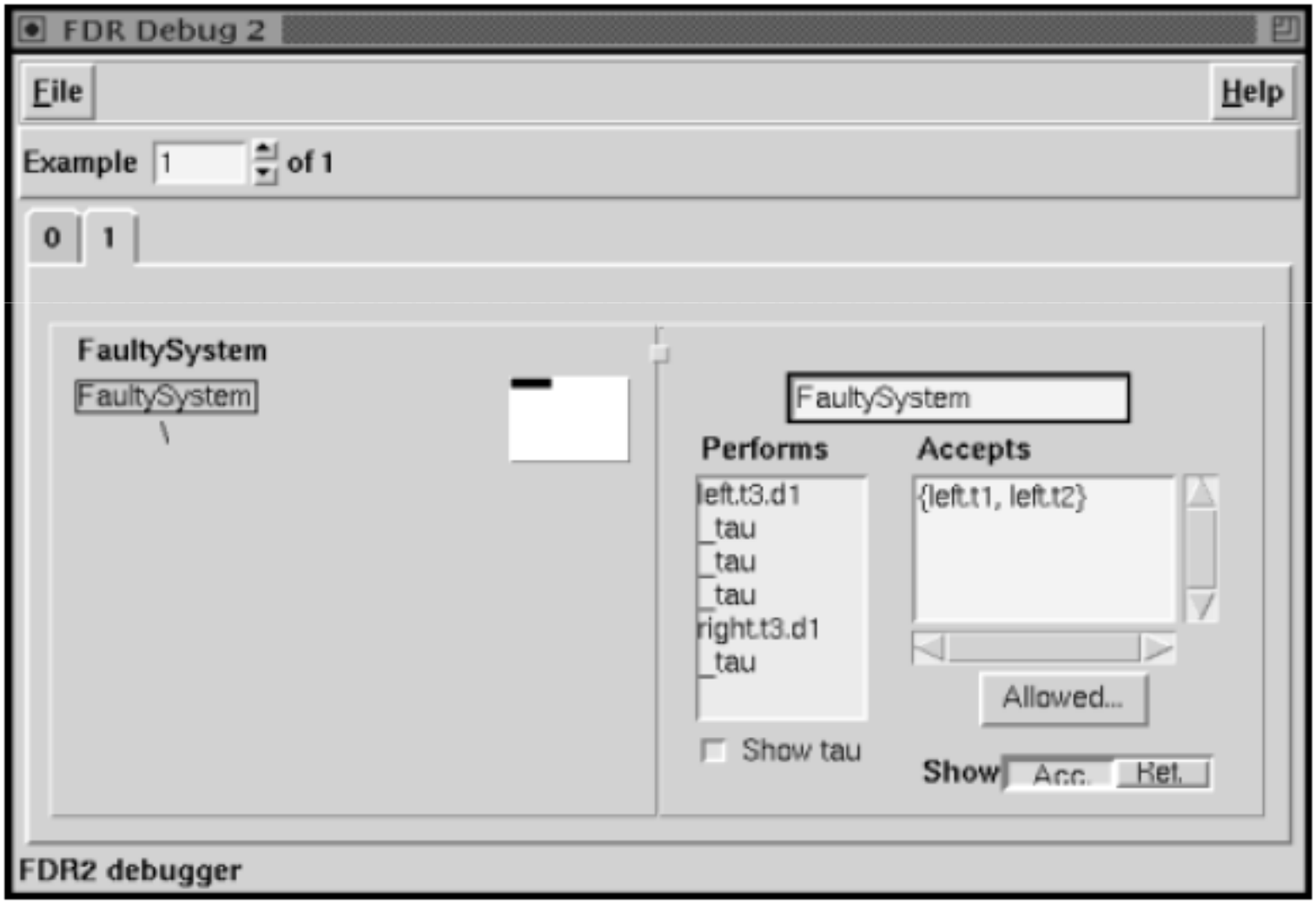
ProBE (Animador de CSP)



FDR (Analisador de CSP)



FDR (Analisador de CSP)



Projeto 1

- Desenvolver modelo em CSP para um sistema usando comunicação através de troca de mensagem, onde propriedades básicas (deadlock, livelock e determinismo) devem ser avaliadas
- Apresentar refinamento deste modelo

MEMÓRIA COMPARTILHADA

Atomicidade Simples

- Ação Atômica
  - Transformação de estado indivisível (Estado intermediário não é visível por outros processos)
- Ação Atômica de Granularidade Simples
  - Implementada diretamente em hardware
- Em geral operações não são atômicas e executá-las concorrentemente pode gerar inconsistências
  - Exemplo ilustrado anteriormente
- Solução trivial: disjunção das variáveis...

Sincronização

- Para obter atomicidade, usamos sincronização para prevenir interleavings indesejáveis
- Através de
  - Combinação de ações atômicas simples em ações compostas: **exclusão mútua**
  - Retardo na execução de processo até estado do programa satisfazer um predicado: **sincronização condicional**

await

- Notação: ⟨ ⟩
- Comando ⟨await (B) S;⟩, onde
  - B (Condição de atraso)
  - S (Seqüência de comandos seqüenciais)
- B é garantido ser true quando a execução de S começa
- Nenhum estado interno em S é visível para outros processos
- Pode ser usado para exclusão mútua quando B não for usado e sincronização condicional quando B é usado

Como implementar await?

- Problema clássico
- N processos (template abaixo) repetidamente executam seção crítica/não-crítica de código
- **process** CS[i=1 to n]{

```
while (true) {
  entry protocol;
  critical section;
  exit protocol;
  noncritical section;
}}
```

SVA

- Codificação de programas que usam memória compartilhada em termos de CSP
  - Uma linguagem concorrente imperativa
- Estudo de algoritmos clássicos e melhorias dos mesmos através de refinamento de CSP
- Modelo mais próximo de implementação Java, por exemplo

Um Exemplo

- Algoritmo de Hyman (problema exclusão mútua)

Abreviação de while true do C

```
H(i) = {iter {b[i] := true;
              while !(t = i) do
                {while b[3-i] do skip;
                 t := i};
              {CRITICAL SECTION}
              b[i] := false;}}
```

- i varia de 1 a 2 (2 processos)

Projeto 2

- Desenvolver modelo em CSP do mesmo sistema do Projeto 1, mas desta vez usando comunicação através de memória compartilhada (SVA), onde propriedades básicas (deadlock, livelock e determinismo) devem ser avaliadas
- Apresentar refinamento deste modelo

Avaliação

- Presença em sala de aula
- Exercícios
- 2 provas
- 2 projetos
- A nota será dada em termos de 25% de cada prova e projeto, mas os exercícios serão levados em conta no empenho (visão global)
  - Pode exercer peso sobre a nota

Referências

- Roscoe, A.W. The Theory and Practice of Concurrency. Prentice-Hall, 1998.
- Andrews, G.R. Multithreaded, Parallel, and Distributed Programming. Addison-Wesley, 2000.
- PDF's no site da disciplina