ANALISANDO PROGRAMAS CONCORRENTES DE MEMÓRIA COMPARTILHADA

Alexandre Mota & Augusto Sampaio

Programação concorrente

- A atomicidade de uma operação é crucial para evitar problemas com concorrência
- Em instruções de baixíssimo nível, elas são garantidas por hardware
- Mas em mais alto nível, elas precisam de algoritmos que garantam isto
  - Exclusão mútua ou sincronização condicional

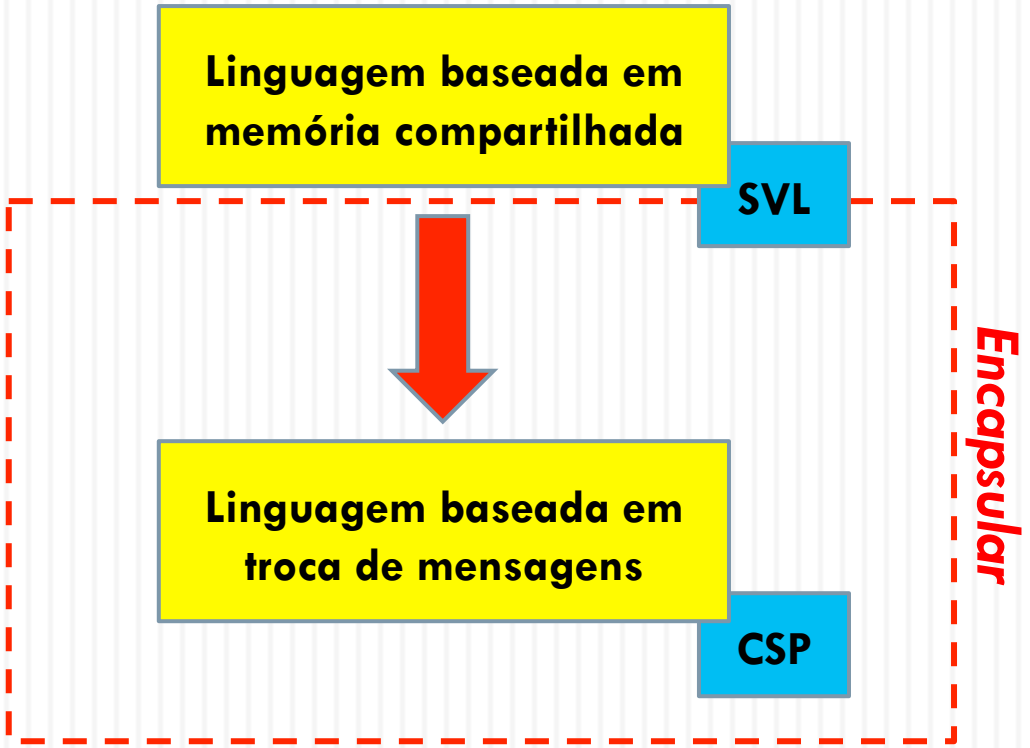
Troca de Mensagens vs memória compartilhada

- Troca de mensagens é mais simples de usar e criar aplicações modulares
  - Mas pouco disponíveis em linguagens de programação
- Maioria das linguagens de programação suporta concorrência usando memória compartilhada
- Testar programas concorrentes ainda é desafiador

Problema

- Como modelar e verificar programas concorrentes de memória compartilhada?

Uma alternativa...



Um exemplo de SVL

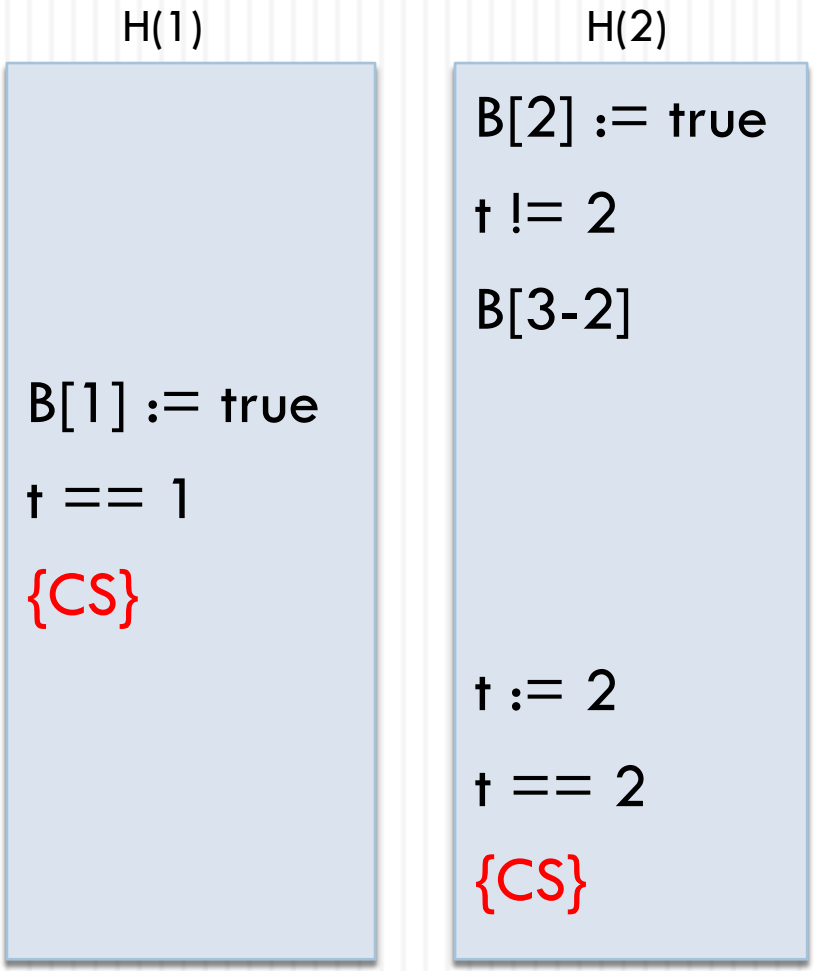
- Linguagem alto nível para SVA
- Exemplo:
  - proposta de exclusão mútua (Hyman)
  - Estado inicial: t=1 e i = {1,2} (N = 2)

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

- Hyman funciona? Analisando com SVA temos...

Um exemplo de SVL

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

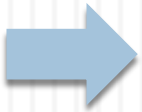


Entrada para SVA

- Linguagem imperativa restrita para escrever threads
  - formato SVL -> CSP intermediário -> CSP resultante
- Variáveis compartilhadas são inteiros e booleanos
  - inclui array unidimensionais
- Programa são threads
  - Comandos executados em ordem arbitrária
  - Expressões não são atômicas
- Construtor para comandos atômicos

CSP intermediário

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



```
P(i) = Iter.Sq.(
 Bassign.(BA.1.i,True),
 Sq.(While.(Not.Eq.IV.2.Const.i,
      Sq.(While.(BVar.BA.1.(3-i),Skip),
          Iassign.(IV.I.2,Const.i))
      ),
        {-- CRITICAL SECTION --}
          Bassign.(BA.1.i,False)
)

Prog = Compile((<<P(1),P(2)>>,
                (<(IVar.I.2,1)>,<>)))
```

Criando um compilador em CSP

- Na verdade o termo mais apropriado seja um simulador
  - O CSP resultante captura o comportamento do programa
- Para criar a infra-estrutura que simula o programa de entrada, basicamente necessita-se de:
  - Uso de datatype
  - Funções recursivas (casamento de padrões)
- Definições estão em svacomp.csp

Tipos de dados

- Para inteiros, temos:

Operações binárias e unárias

```
datatype BinIOps = Plus | Times | Minus | Div | Mod | Max | Min
datatype UIOps = Uminus

datatype IExpr =  IVar.ivnames | IArc.(ianames,IExpr) |
                  Const.{MinI..MaxI} | BIOp.BinIOps.IExpr.IExpr |
                  UIOp.UIOps.IExpr | ErrorI
```

Variáveis inteiras, componentes de array, constantes e expressões unárias e binárias e erro

Tipos de dados

- E para booleanos, temos semelhantemente:

```
datatype BinBOps = And | Or | Xor
datatype CompOps = Eq | Neq | Gt | Ge | Lt | Le

datatype BExpr = BVar.bvnames | BArc.(banames,IExpr) |
                 True | False | Not.BExpr |
                 BBOp.BinBOps.BExpr.BExpr |
                 CompOp.CompOps.IExpr.IExpr | ErrorB
```

Comandos

Faz nada, seqüência de dois comandos, seqüência comandos (lista), loop infinito, loop condicional, if/then/else, atribuição inteira, atribuição booleana, sinal, sinal inteiro, comando atômico e erro

```
datatype Cmd = Skip |
               Sq.(Cmd,Cmd) | SQ.Seq(Cmd) |
               Iter.Cmd | While.(BExpr,Cmd) |
               Cond.(BExpr,Cmd,Cmd) |
               Iassign.(IExpr,IExpr) | Bassign.(BExpr,BExpr)|
               Sig.Signals | ISig.(ISignals,IExpr) |
               Atomic.Cmd  | ErrorC
```

Local da variável

Sinais não sincronizam

Nomes de variáveis

- MinI e MaxI definem a faixa de valores inteiros do programa
- Nomes de variáveis e arrays é definido como segue:

```
datatype namestype = IV.Int | IA.Int.Int
                   | BV.Int | BA.Int.Int | NonVar
```

Id do array e id do componente

Nomes de variáveis

- Para cada programa definir índices de arrays e variáveis utilizados
  - ianums, banums, ivnums e bvnums
- Conjuntos de arrays, variáveis e componentes dos arrays definidos como segue:

```
ianames = {IA.j | j <- ianums}
banames = {BA.j | j <- banums}

ivnames = union({IV.j | j <- ivnums},
                {IA.j.k | j <- ianums, k <- itype(IA.j)})
bvnames = union({BV.j | j <- bvnums},
                {BA.j.k | j <- banums, k <- itype(BA.j)})
```

ltype(a) retorna o tipo do índice do array

Estratégia de compilação

- Programa consiste de
  - declarações e initializações das variáveis e arrays
  - Threads que compartilham os elementos anteriores
- Constrói-se rede que consiste de um processo para cada thread
  - e um processo para cada localização
- Processos-thread não se comunicam diretamente
  - Se comunicam com processos de variáveis
  - Comunicam externamente através de sinais

Comportamento da compilação

- De forma simplificada

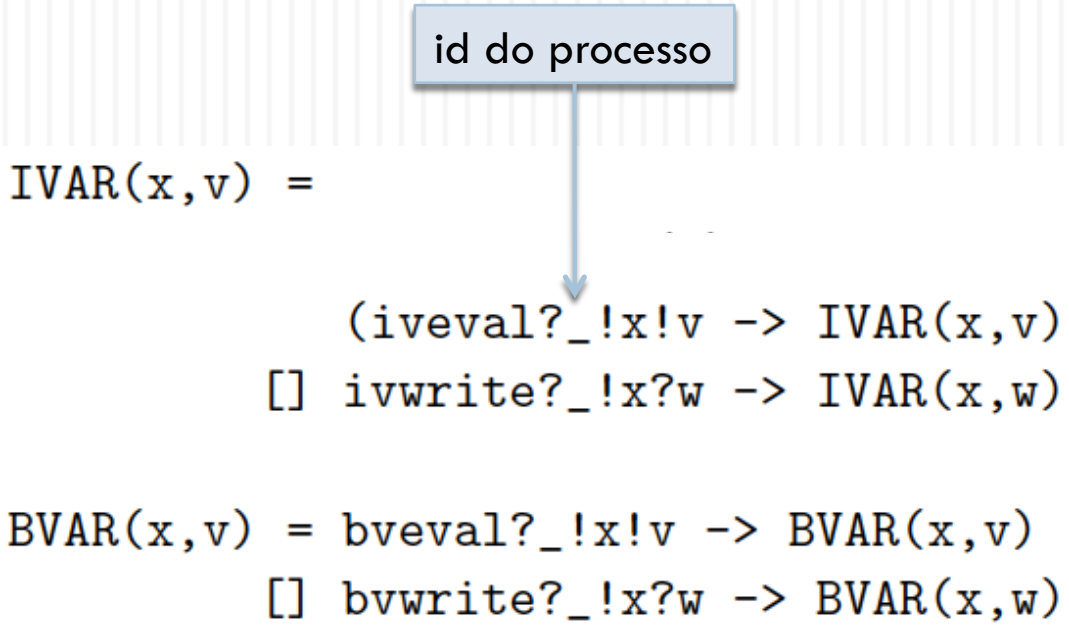
```
Threads
    [| {|iveval, ivwrite,
          bveval, bvwrite|} |] Vars
```

- Onde

```
Threads = T1 ||| ... ||| Tj
Vars = V1 ||| ... ||| Vw
```

Estratégia de compilação

- Localização individual (variável ou componente de array) é um processo simples
  - particularmente na ausência de construções atômicas



Estratégia de compilação

- Quando uma thread executa atomicamente, nenhuma outra thread acessa variáveis, e também nenhum outro processo entra na seção atômica até que seja finalizada

```
BVAR_at(x,v) = bveval?_!x!v -> BVAR_at(x,v)
             [] bvwrite?_!x?w -> BVAR_at(x,w)
             [] start_at?j -> BVAR_inat(j,x,v)

BVAR_inat(j,x,v) = bveval.j!x!v -> BVAR_inat(j,x,v)
                 [] bvwrite.j!x?w -> BVAR_inat(j,x,w)
                 [] end_at.j -> BVAR_at(x,v)
```

Compilando uma thread

```
MainProc(Skip,j) = SKIP

MainProc(Sig.x,j) = x -> SKIP

MainProc(Sq.(p,q),j) = MainProc(p,j);MainProc(q,j)
MainProc(SQ.<>,j) = SKIP
MainProc(SQ.<p>^Ps,j) = MainProc(p,j);MainProc(SQ.Ps,j)

MainProc(Iter.p,j) = MainProc(p,j);MainProc(Iter.p,j)
```

Expressões e continuação

- i e b são expressões (Iexpr e BExprs) a serem avaliadas
- Funções IExpEval(e,P,j) e BExpVal(b,P,j) retornam:
  - a função P aplicada ao resultado da avaliação (x)
  - Ou um erro, caso a expressão passe dos limites para um valor inteiro
- Exemplo: BExpVal(True,P,1) = P(true)
- P é a continuação da thread, definida a partir do valor de x

Compilando uma thread

```
MainProc(While.(b,p),j) =
        let P(x) = if x then MainProc(p,j);MainProc(While.(b,p),j)
                        else SKIP
        within BExpEval(b,P,j)
```

P(true) ou P(false)

```
MainProc(Cond.(b,p,q),j) =
        let P(x) = if x then MainProc(p,j) else MainProc(q,j)
        within BExpEval(b,P,j)
```

Compilando uma atribuição

- Bassign(el,e) significa que uma localização el é atualizada com um valor e . É representado pelo evento
  - `bvwrite.j.lv.rv`
  - onde el e e são mapeados para lv e rv

- Funções BLvEval(el,Q,j) e ILvEval(el,Q,j) retornam:
  - Q(lv) para uma expressão el (BExpr e IExpr) do tipo variável ou componente de array
  - Ou erro, se o tipo for inadequado ou localização inexistente

- Exemplo: BlvEval(Bvar.BV.1) = Q(BV.1)

Compilando uma thread

```
MainProc(Bassign.(el,e),j) =
                          let Q(lv) =
                          let P(rv) = bvwrite.j.lv.rv -> SKIP
                              within BExpEval(e,P,j)
                          within BLvEval(el,Q,j)

MainProc(Iassign.(el,e),j) =
                          let Q(lv) =
                          let P(rv) = if member(rv,ctype(lv)) then
                                                ivwrite.j.lv.rv -> SKIP
                                      else error.j -> STOP
                              within IExpEval(e,P,j)
                          within ILvEval(el,Q,j)
```

ctype(v) retorna
o tipo da variável v

Compilando uma thread

```
MainProc(ISig.(c,e),j) = let P(x) = c!x  -> SKIP
                         within IExpEval(e,P,j)

MainProc(Atomic.p,j) = start_at.j -> MainProc(p,j);end_at.j -> SKIP

MainProc(ErrorC,j) = error.j -> STOP
```

Avaliando expressões

- Acontece em dois estágios:
  - Localizações são instanciadas
  - É calculado o valor da expressão

- Da esquerda para a direita, cada localização v da expressão:
  - é sinalizada para instanciação
    - `ISV.v`
  - Em seguida instanciada
    - `iveval.j.v.1?x`

- Isto é realizado pelas funções ffetchi(e) em conjunto com IExpEval e BExpEval

Avaliando expressões

```
datatype fetch =  NoF | ISV.ivnames | BSV.bvnames | ErrorX

ffetchi(IVar.n) = ISV.n
ffetchi(IArc.(v,e)) = let f = ffetchi(e)
                  within
                  (if f == NoF then
                     (let k=evaluatei(e) within
                        if ok(k) then
                          let n=num(k) within
                            (if member(n,itype(v)) then ISV.v.n
                                                   else ErrorX)
                        else ErrorX)
                   else f)

ffetchi(Const.k) = NoF

ffetchi(BIOp._.ea.eb) = let ffe = ffetchi(ea) within
                      if ffe==NoF then ffetchi(eb)
                                  else ffe

ffetchi(UIOp._.e) = ffetchi(e)

ffetchi(ErrorI) = NoF
```

Fetch de variável inteira

Retorna Ok.result

Verifica result

Num(Ok.result) = result

Não é necessário fetch

Primeira ocorrência é sinalizada

Avaliando expressões

- Juntando tudo temos

```
IExpEval(e,P,j) =
          let IXEF(NoF) =
                 let k=evaluatei(e) within
                    if ok(k) and num(k)>=MinI and num(k) <=MaxI then
                                P(num(k))
                       else error.j -> STOP
              IXEF(ISV.v) = iveval.j.v?x -> IExpEval(subsi(v,x,e),P,j)
              IXEF(_) = error.j -> STOP
          within IXEF(ffetchi(e))
```

Substitui na expressão a localização pelo seu valor

Verificando programas

- Validade de expressões booleanas (BEXPR)
  - `assert always/never BEXPR in PROGRAM`
- Presença de sinais durante a execução
  - `assert nosignal { SIGNALS } in PROGRAM`
- Refinamentos

Exclusão Mútua de Hyman

```
H(i) = {iter {b[i] := true;
              while !(t = i) do
                {while b[3-i] do skip;
                 t := i
                };
              count := count + 1;
              if count > 1 then sig(mutexerror);
              count := count -1;
              b[i] := false;
              }
        }

Prog = <H(1),H(2)>
```

Exclusão Mútua de Hyman

- Propriedade em SVL

```
assert

nosignal {outofrange, mutexerror} in Prog
```

- Propriedade em CSP

```
assert

CHAOS(diff(Events,{|outofrangeT,
  mutexerrorT|})) [T= Prog
```

Exeção levantada quando expressão inteira extrapola Min..Max

Exclusão Mútua de Hyman

Pelo log vemos: (1) expressões não são atômicas, (2) leitura e escrita de variáveis são atômicas

```
Checking CHAOS(diff(Events,{|
   outofrangeT,mutexerrorT|}))
   [T= Prog
xfalse
BEGIN TRACE example=0
   process=1
bvwriteT.BA.1.2.true.0
breq.BE.1
ivevalT.I.2.1.0
beval.BE.1.true
bvevalT.BA.1.1.false.0
...
```

Log de FDR (results)

Result: false

P(1)	P(2)
	b[2] assigned true
	Evaluation of !(t = i(2)) starts
	!(t(1) = i(2)) is true
	b[1] is false
b[1] assigned true	
Evaluation of !(t = i(1)) starts	
!(t(1) = i(1)) is false	
Evaluation of count+1 starts	
	t assigned 2
	Evaluation of !(t = i(2)) starts
count(0)+1 is 1	
count assigned 1	
	!(t(2) = i(2)) is false
	Evaluation of count+1 starts
	count(1)+1 is 2
	count assigned 2
Evaluation of count > 1 starts	
count(2) > 1 is true	
mutexerror	

Debug do log

Exclusão Mútua de Hyman

- Propriedade em SVL

```
assert never (count > 1) in Prog
```

- Propriedade em CSP

```
assert

CHAOS(diff(Events, {|assertionfailedT|}))
[T=
Compile((<<H(1), H(2), (<(IVar.I.2,1)>,<>)),
            Atomic.Cond.(Not.Gt.IVar.I.1.Const.1,
                         Skip,Sig.assertionfailed)>>)
```

```
Assertion = atomic Iter {
                if count > 1 then
                Sig(assertionfailed)  }
```

Exclusão Mútua de Hyman

Result: false

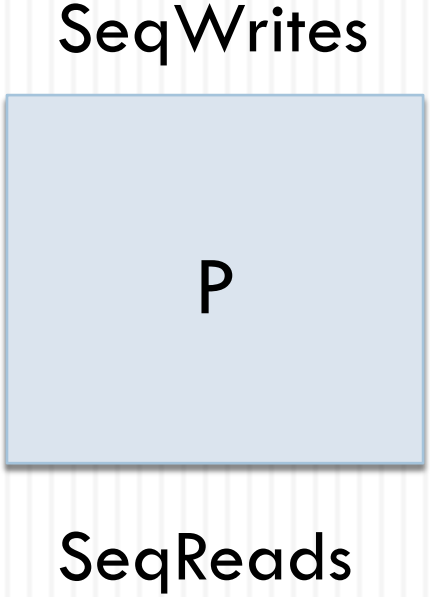
P(1)	P(2)	assertion
	b[2] assigned true	
	Evaluation of !(t = i(2)) starts	
	!(t(1) = i(2)) is true	
	b[1] is false	
b[1] assigned true		
Evaluation of !(t = i(1)) starts		
!(t(1) = i(1)) is false		
Evaluation of count+1 starts		
	t assigned 2	
count(0)+1 is 1		
	Evaluation of !(t = i(2)) starts	
	!(t(2) = i(2)) is false	
count assigned 1		
	Evaluation of count+1 starts	
	count(1)+1 is 2	
	count assigned 2	
		Atomic section entered
		Evaluation of !(count > 1) starts
		!(count(2) > 1) is false
		assertionfailed

Refinamento de programas

- Relação de traces entre modelos CSP
- Eventos são leituras e escritas do ambiente em variáveis compartilhadas
- Programas parciais
  - interagem com o resto do programa através de variáveis compartilhadas
  - variáveis compartilhadas definem o contexto

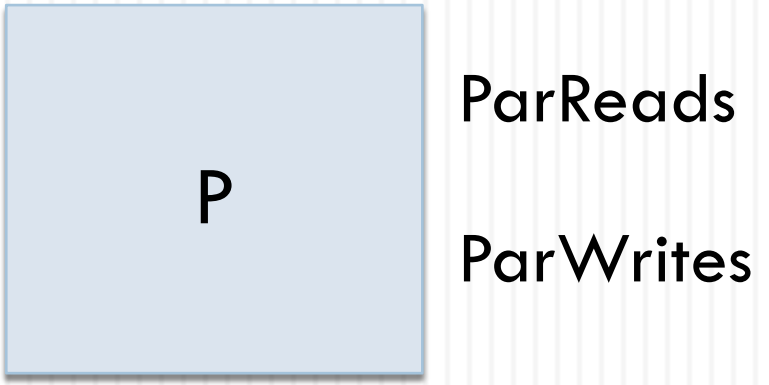
Refinamento de programas

- Contexto sequencial
  - Processo visto como uma função
    - Prog = P
    - Ou, Prog = atomic {P} ||| Q
- Localizações modificadas antes da execução e lidas depois
  - SeqWrites
  - SeqReads



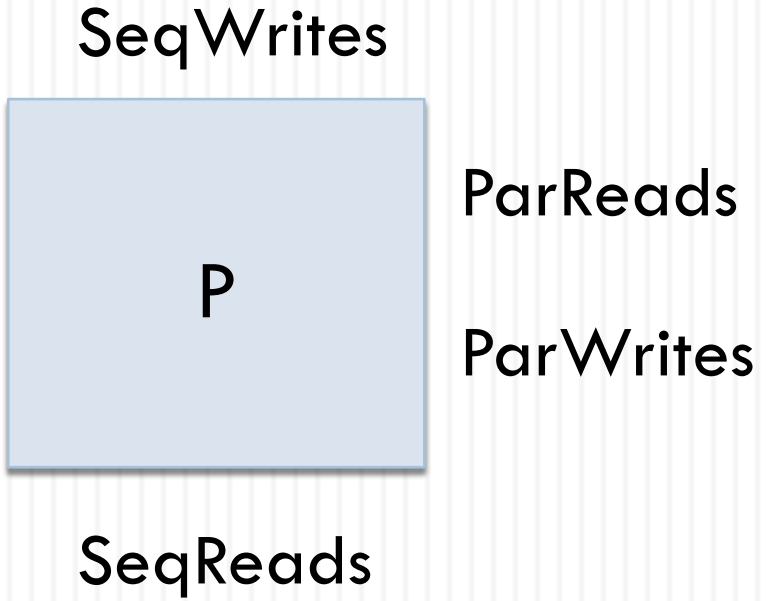
Refinamento de programas

- Contexto paralelo
  - Inicia ao mesmo tempo com outras threads
    - Prog = P1 ||| ... ||| PN
- Localizações modificadas e lidas durante execução
  - ParWrites
  - ParReads



Refinamento de programas

- Contexto geral
  - Parte de uma thread (não atômico)
    - P1 = { ... P ... }
    - Prog = P1 ||| ... ||| PN
- Localizações modificadas e lidas antes, durante e depois



Refinamento de programas

- Considerar contexto geral
- Q [G= P iff
  - Sinais/erros de P são subconjunto dos sinais/erros de Q
  - Eventos de leitura/escrita do ambiente interagindo com P são subconjunto dos eventos de leitura/escrita do ambiente interagindo com Q, nas mesmas localizações
- Ações do processo não são observáveis (internas)

Programas PP e QQ

- Exemplo:
  - Inicialmente x,y = 0,0
  - ParReads = {x,y} ParWrites = SeqWrites = SeqReads = {}

```
PP(i) = iter{
 x := 1;
 x := 0;
 y := 1;
 y := 0 }
```

```
QQ(i) = iter{
 x := 1;
 y := 1;
 x := 0;
 y := 0 }
```

PP e QQ são diferentes no contexto geral?

Refinamento de programas

- Ext_atomic = true
  - ambiente pode entrar em seções atômicas
  - Impede atividade do programa
  - Contexto paralelo e geral

Programas PP e QQ

- PP

x,y = 0,0
x,y = 1,0
x,y = 0,0
x,y = 0,1
x,y = 0,0
...

- QQ

x,y = 0,0
x,y = 1,0
x,y = 1,1
x,y = 0,1
x,y = 0,0
...

Programas PP e QQ

- ext_atomic = false
  - O ambiente em qualquer ponto lê x e y como 0 e 1
  - PP [G= QQ
  - QQ [G= PP

- ext_atomic = true
  - Em seção atômica ambiente detecta x,y = 1,1 em QQ e não em PP
  - Estados de PP subconjunto de QQ, não o contrário
  - QQ [G= PP

Mecanizando refinamento

- Analisando contexto geral
- Programa é P é representado
  - PStart; CSP(P); PEnd
- O ambiente modelado com uma thread (j = -1) em paralelo com o programa no alphabeto X igual

```
{{|ivwrite.-1,bvwrite.-1,iveval.-1,bveval.-1|},
  Signals', ISignals, Errors,
  {|start_at.-1,end_at.-1 | ext_atomic |}}
```

Mecanizando refinamento

- Ambiente até PStart

GRefReg0 = PStart -> GRefReg1(false)

[] ivwrite.-1?x:inter(SeqWrites,ivnames)?_ -> GRefReg0

[] bvwrite.-1?x:inter(SeqWrites,bvnames)?_ -> GRefReg0

Mecanizando refinamento

- Ambiente entre PStart e PEnd

```
GRefReg1(ineat) = PEnd -> GRefReg2
  [] ivwrite.-1?x:inter(ParWrites,ivnames)?_ -> GRefReg1(ineat)
  [] bvwrite.-1?x:inter(ParWrites,bvnames)?_ -> GRefReg1(ineat)
  [] iveval.-1?x:inter(ParReads,ivnames)?_ -> GRefReg1(ineat)
  [] bveval.-1?x:inter(ParReads,bvnames)?_ -> GRefReg1(ineat)
  [] ext_atomic and (not ineat)&start_at.-1 -> GRefReg1(true)
  [] ext_atomic and  ineat&end_at.-1 -> GRefReg1(false)
  [] ([] x:Union({Signals,ISignals}) @ x -> GRefReg1(ineat))
  [] ([] x:Errors @ x -> STOP)
```

Mecanizando refinamento

- Ambiente depois de PEnd

```
GRefReg2 =
   iveval.-1?x:inter(SeqReads,ivnames)?_ -> GRefReg2
   [] bveval.-1?x:inter(SeqReads,bvnames)?_ -> GRefReg2
```

Mecanizando refinamento

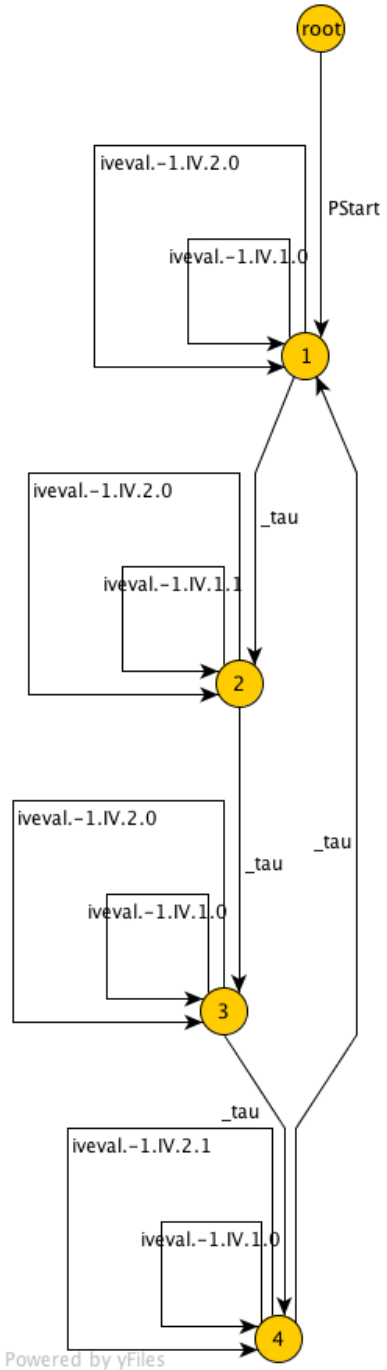
- Q [G= P iff

  PStart; CSP(Q); PEnd |[X]| Ambiente

  [T=

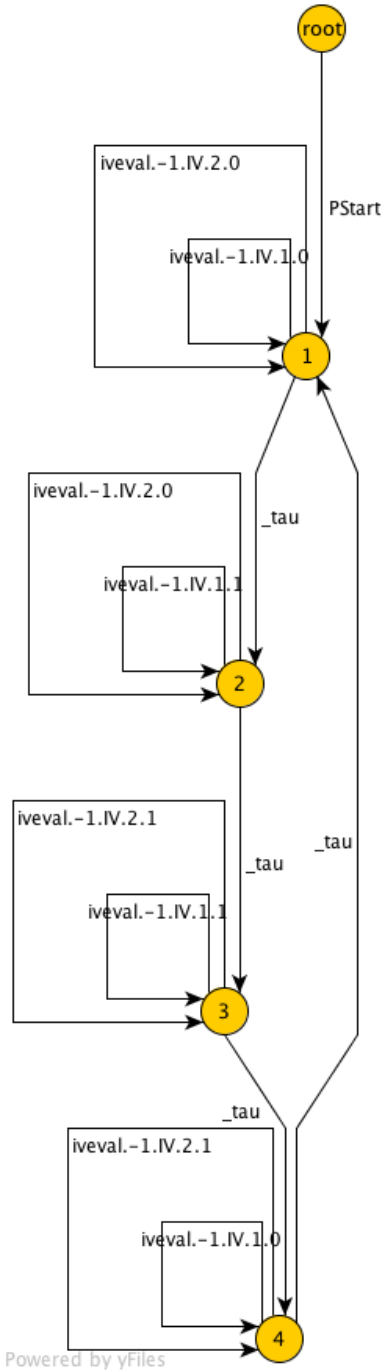
  PStart; CSP(P); PEnd |[X]| Ambiente

Prog

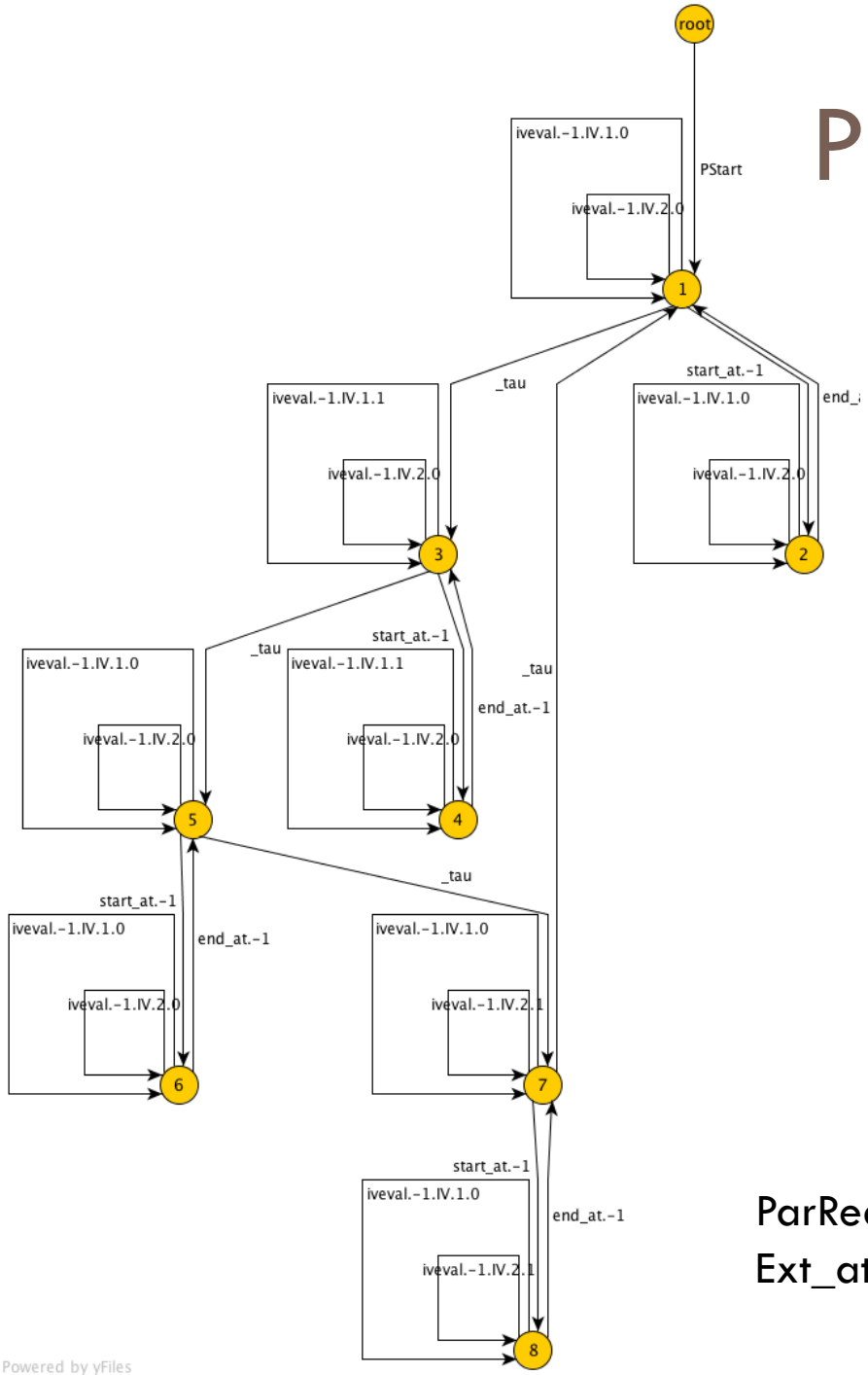


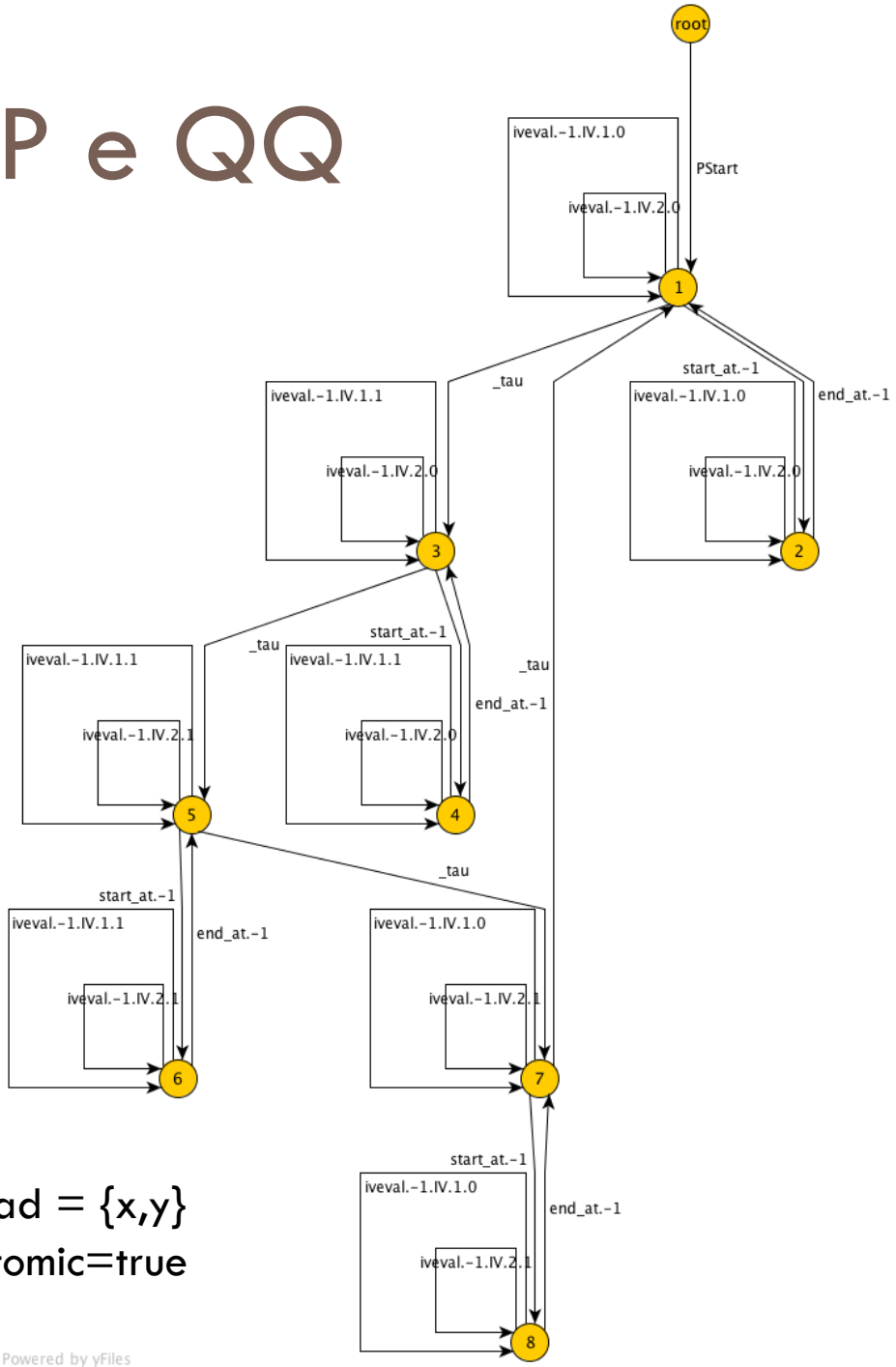
PP e QQ

ParRead = {x,y}
Ext_atomic=false



PP e QQ





ParRead = {x,y}

Ext_atomic=true

Algoritmo Bakery: N Procs


```
int turn[1:n]=([n] 0)
process CS[i=1 to n]  {
    while (true) {
        turn[i]=1; turn[i]=max(turn[1:n]) + 1;
        for [j=1 to n st j != i]
          while (turn[j]!=0 and
                    (turn[i],i) > (turn[j],j)) skip;
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

Bakery em SVL

```
B(i) = iter {
 int j,k;

 turn[i] := 1;

 turn[i] :=
   MAX(turn[1],..,
       turn[N];
```

```
k := 1;
while (k <= N) do {
  if !(k = i) then
   while turn[k]>0 &&
     (turn[i] > turn[k]||
     (turn[i] = turn[k]&&
      i > k)) do skip;
   k := k + 1;
 };
 sig(css.i);
 sig(cse.i);
 turn[i] := 0
}
```

Calculando o máximo

```
MaxD(i) = {
  j := 1;
  while j <= N do {
    if turn[i] < turn[j] then
      turn[i] := turn[j];
    j := j + 1 };
  turn[i] := turn[i] + 1
}
```

- Pode usar dois valores de turn[j]
- Possivelmente
  - turn[i]' < turn[i]

Bakery em SVL

- Considerando N = 3 e MaxD
- Compilação
  - WideStruct = hierarchCompress < B(3),B(2),B(1)>
- Propriedade
  - %%SPEC = css?i -> cse!i -> SPEC
- Verificar que não viola seção crítica (true)
  - assert %- SPEC [T= WideStruct \{|error,verror|} -% in WideStruct

Bakery em SVL

- Violação da seção crítica
  1. B(2) depois do máximo tem turn[2] = 2, que é lido por B(1) na função de máximo
  2. B(2) entra e sai da seção crítica faz turn[2] := 0, que é atribuído a turn[1] na função de máximo em B(1)
  3. B(2) quer entra na secao critica, calcula turn[2] := 2 e avança até k=3
  4. B(1) depois do max faz turn[1] := 1, avança ate k=4 e entra na seção crítica
  5. B(2) também entra na seção crítica

Refinando Bakery

- Uma opção é comparar uma versão B1(i) com outra versão B2(i) do algoritmo, neste caso
  - ParRead = { turn[j] | j <- {1..N} }
  - ParWrites = { turn[j] | j <- {1..N}, j != i }
  - Ext_atomic = false
  - B1(i) [G= B2(i) ?
  - B2(i) [G= B1(i) ?

Refinando Bakery

- Outra opção é verificar refinamento apenas o trecho que muda
  - procedimento de calcular máximo
- Vantagem é isolar a mudança, o que facilita a análise
- Vamos considerar quatro implementações que cálculam o máximo

Refinando cálculo do máximo

- Considere i = 3
- No contexto sequencial todas são equivalentes
  - SeqWrites = SeqReads = {turn[3]}
- E no contexto geral (N = 3) ?
  - SeqWrites = SeqReads = {turn[3]}
  - ParReads = {turn}
  - ParWrites = {turn[1],turn[2]}
  - Ext_atomic = false

Refinando cálculo do máximo

```
MaxC(i) = {
  j := 1;
  while j <= N do {
    temp := turn[j];
    if temp >= turn[i] then
      turn[i] := temp;
    j := j + 1 };
  turn[i] := turn[i] + 1
}
```

- Copia turn[j]
- Atualiza turn[i] sempre que maior é encontrado

Refinando cálculo do máximo

- assert MaxD(3) [G= MaxC(3)
  - MaxC(3) melhor que MaxD(3)
  - não acontece turn[3]'<turn[3]

- Bakery não viola seção crítica com MaxC

Refinando cálculo do máximo

- Alternativa para MaxC

```
MaxB(i) = {
  j := 1;
  while j <= N do {
    turn[i] := max(turn[j],turn[i]);
    j := j + 1 };
  turn[i] := turn[i] + 1
}
```

- assert MaxC =G MaxB

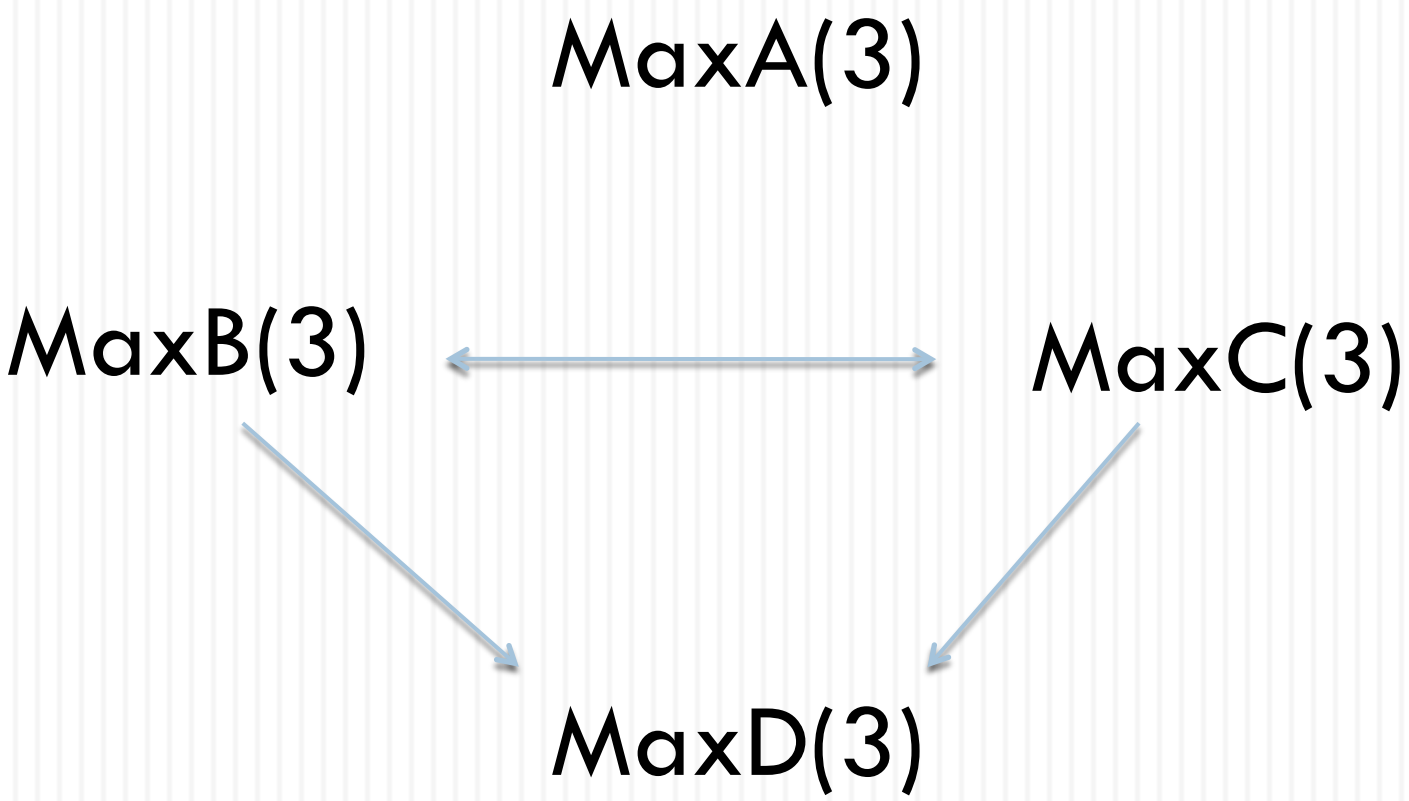
Refinando cálculo do máximo

```
MaxA(i) = {
  j := 1;
  while j <= N do {
    temp :=
       max(turn[j],turn[i]);
    j := j + 1 };
  turn[i] := temp + 1
}
```

- Atualiza turn[i] uma única vez
- Preserva seção crítica em Bakery
- Não refina, nem é refinado pelos outros

Refinando cálculo do máximo

- No contexto geral (N = 3 e i = 3)



CSP vs CSP de SVA

- Suponha um mesmo programa foi especificado em CSP (P) e em SVA (Psva)
  - O alphabeto de P é igual ao conjunto de sinais em Psva
- Em SVL é possivel comparar P com Psva
  - Assert %- P [M= Psva -% in Psva
  - Assert %- Psva [M= P -% in Psva
- Onde M = {T, F, FD}
  - Precisão em F e FD depende da estrutura de Psva

Compatibilizando sinais com canais

- Suponha em P há um canal comunica valores inteiros
  - Channel c : T1. ... .Tn
  - P = ... c.v1. ... . vn -> ...
- Mas sinais de SVA só comunicam um inteiro
  - Sig s : T
  - Psva = ... Sig(s,v) ...
- Uma forma de modelar c através de sinais é
  - Psva = ... Sig(c1, v1); ... ; Sig(cn, vn) ...

Compatibilizando sinais com canais

- Exemplo:

P(i) = send.i.1 -> receive.i.1 -> P(i)

Psva(i) = iter {
  sig(send1.i); sig(send2.1);
  sig(receive1.i); sig(receive2.1);
}
PsvaC = <Psva(1)>

Compatibilizando sinais com canais

- Definindo mapeamento de seqüência de sinais em canais
  - Map = { (s1,c1), ..., (sn, cn) }
- Linker(Map) é o processo que reconhece s e comunica c em seguida, onde (s,c)<-Map
- Na verificação Psva é susbtituído pelo mapeamento
  - MapEvs(Psva,Map) = (Psva [|Signals|] Linker(Map)) \ Signals

Compatibilizando sinais com canais

- Exemplo

MAP = { (<send1.1, send2.1>, send.1.1),
(<receive1.1, receive2.1>, receive.1.1) }

assert %- P(1) [T= MapEvs(PsvaC, MAP) -% in PsvaC

assert %- MapEvs(PsvaC, MAP) [T= P(1) -% in PsvaC

Lições associadas

- Estudo de caso sobre uso prático de CSP
- Como criar simuladores e compiladores em CSP
- Guia para uma nova ferramenta capaz de ajudar na análise de programas que usam memória compartilhada

Vantagem de SVL->CSP

- Material apresentado aqui serve para analisar programas baseados em memória compartilhada
- Mas há várias outras iniciativas, por exemplo, Promela (Spin), que já fazem isto há muito tempo
- A principal vantagem então de fazer uso do material daqui está no fato de ser possível comparar programas usando a noção de refinamento de CSP

Referências

- A. W. Roscoe. Compiling Shared Variable Programs into CSP. In Proceedings of PROGRESS workshop 2001, 2001.
- A. W. Roscoe and D. Hopkins. Sva, a tool for analysing shared-variable programms. In Proceedings of AVoCS 2007, pages 177–183, 2007.