Verificação de Protocolos

Promela, Spin

FEUP/MRSC/AMSR
MPR

Bibliografia

» Aula preparada com base nos seguintes documentos

– Gerard J. Holzmann, “Tutorial: Design and Validation of Protocols”, Computing Science Technical Report, AT&T Bell Laboratories, May 1991

– Gerard J. Holzmann, “ The Model Checker SPIN”, IEEE Transactions on Software Engineering, Vol. 23, N 5, May 1997.

– Telelogic, “Tutorial: The SDL Validator”.

– Zhoar Manna and Amir Pnueli, “The Temporal Logic of Reactive and Concurrent Systems: Specification”, Springer-Verlag, 1992.

– Joseph Sifakis. Tutorial B, “The automatic verication of protocols”, In A. Dantine, G. Leduc, and P. Wolper, editors, 13th IFIP Symposium on Protocol Specification, Testing and Verification, May 1993, Liege Belgium, 1993.

Verificação de Protocolos

- Detecção de situações patológicas
  - » *Deadlocks*
  - » *Livelocks*
  - » Recepção imprevista de sinais
- Validação
  - » Fornecimento dos serviços desejados
  - » Sequência correcta de serviços
- Executada sobre modelos
- Automatizada

Especificação Informal – Protocolo com Bit Alternado (./..)

```
1   The protocol defines a simplex data-transfer channel, i.e., it
2   will pass data in only one direction, from sender to receiver.
3   Control information, however, flows in both directions. It is
4   assumed that the system has perfect error detection.
5   To each message sent from A to B we attach an extra bit called
6   the alternation bit. After B receives the message it decides if
7   the message is error-free. It then sends back to A a
8   verification message, consisting of a single verify bit,
9   indicating whether or not the immediately preceding A to B
10  message was error-free. After A receives this verification, one
11  of three possibilities hold:
12  1. The A to B message was good
13  2. The A to B message was bad
14  3. A cannot tell if the A to B message was good or bad
15  because the verification message (sent from B to A) was in error
16  In cases 2 and 3 A resends the same A to B message as before. In
17  case 1 A fetches the next message to be sent, and sends it,
18  inverting the setting of the alternation bit with respect to the
19  previous A to B message.
```

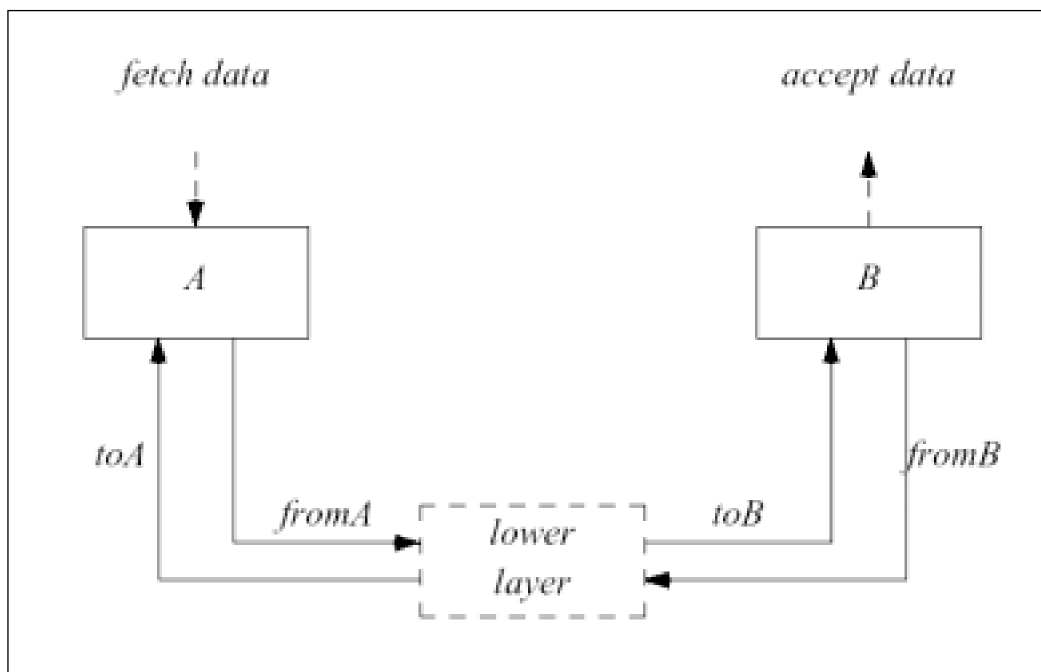
Especificação Informal Protocolo com Bit Alternado (../..)


```
20  Whenever B receives a message that is not in error it compares
21  the alternation bit of this new message to the alternation bit of
22  the most recent error-free reception. If the alternation bits
23   are equal the new message is not accepted. The new message is
24  accepted only if the two alternation bits differ. The
25  verification messages from B to A indicate error-free reception
26  independently of the acceptance of the messages.
27  Initialization of this scheme depends upon A and B agreeing on an
28  initial setting of the alternation bit. This is accomplished by
29  an A to B message whose error-free reception (but not necessarily
30  acceptance) forces B's setting of the alternation bit. Multiple
31  receptions of such a message cannot do harm.
32  This protocol has the property that every message fetched by A is
33  received error-free at least once and accepted at most once by B.
```

» Estará este protocolo clara e correctamente especificado?

» Conseguiremos demonstrá-lo antes da implementação?

PROMELA – Protocol Meta-Language



PROMELA – Protocol Meta-Language

- 2 mensagens

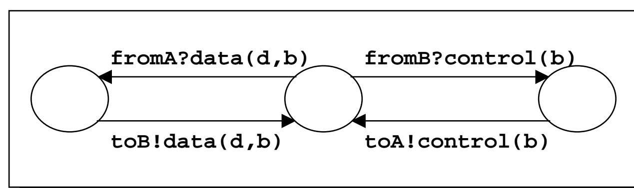
```
mtype = { data, control }      /* data and acks */
```

- Canais de comunicação entre A e B

```
chan fromA =    [N] of { byte, byte, bit };       /* data, udata, seqno */
chan toB =      [N] of { byte, byte, bit };       /* data, udata, seqno */
chan fromB =    [N] of { byte, bit };             /* control, seqno */
chan toA =      [N] of { byte, bit };             /* control, seqno */
```

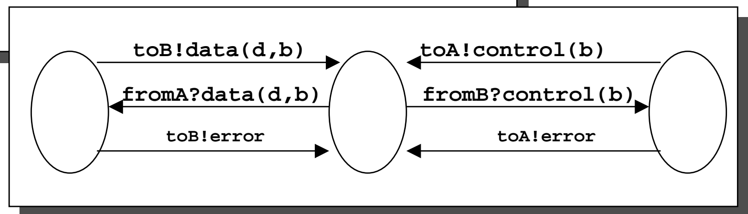

Canal Ideal

```
proctype lower(chan fromA, toA, fromB, toB)
{   byte d; bit b;

    do
    :: fromA?data(d,b)  -> toB!data(d,b)
    :: fromB?control(b);   toA!control(b)
    od
}
```



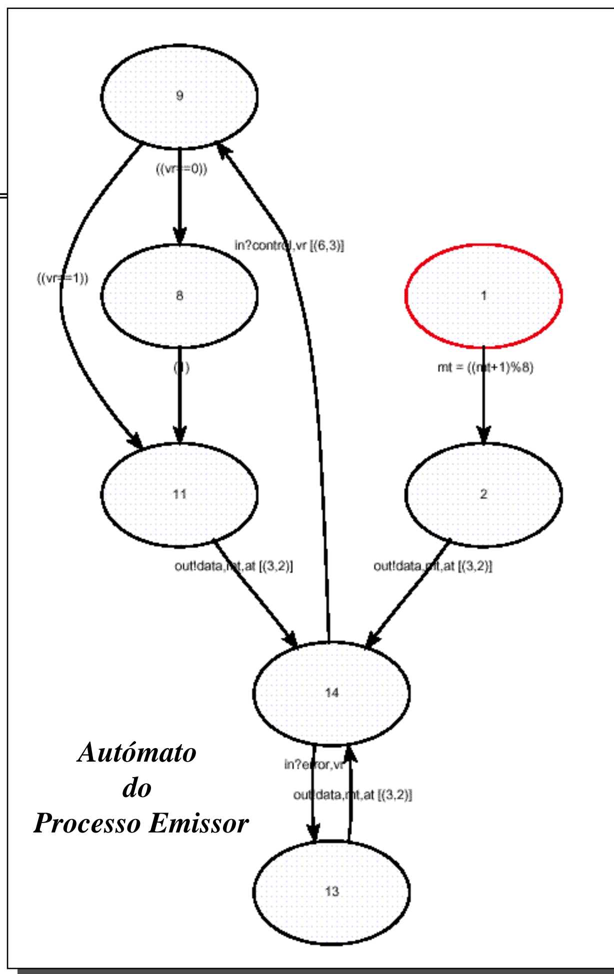
```
mtype = { data, control, error }

proctype lower(chan fromA, toA, fromB, toB)
{   byte d; bit b;

    do
    :: fromA?data(d,b) ->
            if
            :: toB!data(d,b)    /* correct */
            :: toB!error        /* distorted */
            fi
    :: fromB?control(b) ->
            if
            :: toA!control(b)
            :: toA!error
            fi
    od
}
```


Canal Não Ideal



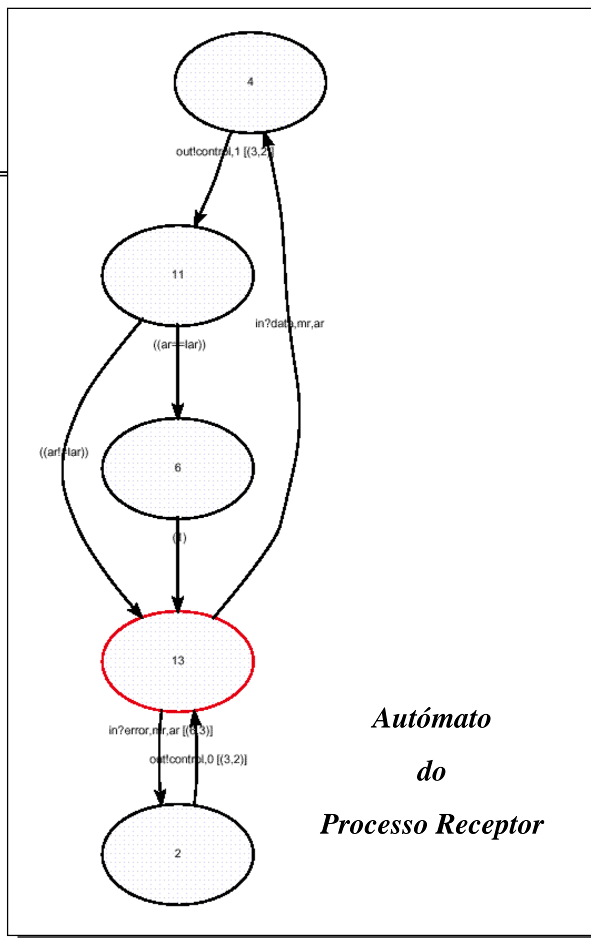
Processo Emissor

```
proctype A(chan in, out)
 {   byte mt;                       /* message data */
     bit at;                        /* alternation bit */
     bit vr;                        /* verify bit */

     FETCH;                         /* get a new mesg */
     out!data(mt,at);               /* send it */
     do
     :: in?control(vr) ->           /* line 11, await response */
          if
          :: (vr == 1) ->           /* line 12, correct send */
               FETCH;               /* line 17, new message */
               at = 1-at            /* line 18, toggle bit */
          :: (vr == 0) ->           /* line 13, send error */
               skip                 /* line 16, don't fetch */
          fi;
          out!data(mt,at)           /* line 16 */
     :: in?error(vr) ->             /* line 14-15, recv error */
          out!data(mt,at)           /* line 16 */
     od
}
```



Processo Receptor

```
proctype B(chan in, out)
{     byte mr;                     /* message data */
      byte last_mr;                /* mr of last error-free msg */
      bit ar;                      /* alternation bit */
      bit lar;                     /* ar of last error-free msg */

      do
      :: in?error(mr,ar) ->       /* lines 7-10 */
               out!control(0)     /* lines 8,25,26 */
      :: in?data(mr,ar) ->        /* lines 7-10 */
               out!control(1);    /* lines 8,25,26 */
           if                     /* line 20 */
           :: (ar == lar) ->      /* line 21 */
                    skip          /* line 23 */
           :: (ar != lar) ->      /* line 24 */
                    ACCEPT;       /* line 24 */
                    lar = ar;     /* line 22 */
                    last_mr = mr
           fi
      od
}
```



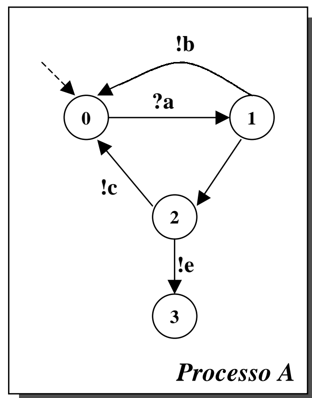
Processo Inicial

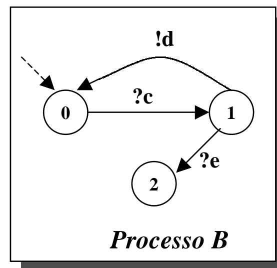
```
#define N      2
#define MAX    8
#define FETCH  mt = (mt+1) % MAX
#define ACCEPT assert(mr == (last_mr+1)%MAX)

mtype = { data, control, error };

#include "lynch0.A"
#include "lynch0.B"
#include "lynch0.C"

  init {
       chan fromA =   [N] of { byte, byte, bit };
       chan toB =     [N] of { byte, byte, bit };
       chan fromB =   [N] of { byte, bit };
       chan toA =     [N] of { byte, bit };
       atomic {
              run A(toA, fromA);
              run B(toB, fromB);
              run lower(fromA, toA, fromB, toB)
              }
        }
```

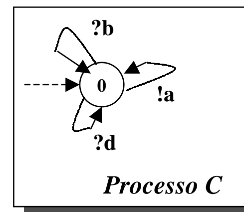
Formalização do Funcionamento - Estado do Sistema

- ***Estado do Sistema***
  - » Valores das variáveis locais e globais
  - » Ponto de execução (estado de controlo) de cada processo
  - » Conteúdo de cada canal de mensagens

Ex. S3= < 0, 1, 0 >. Ex. S8=< P.s=2, P.sender=Q, P.q=<>, P.n=0, Q.s=1, Q.sender=P, Q.q=<c> >

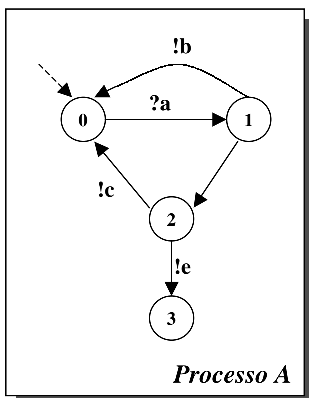


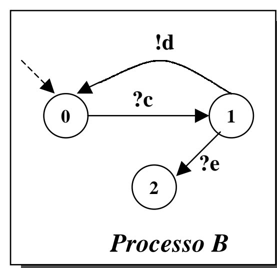


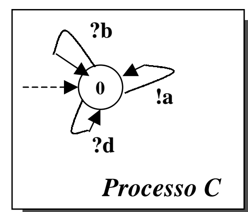


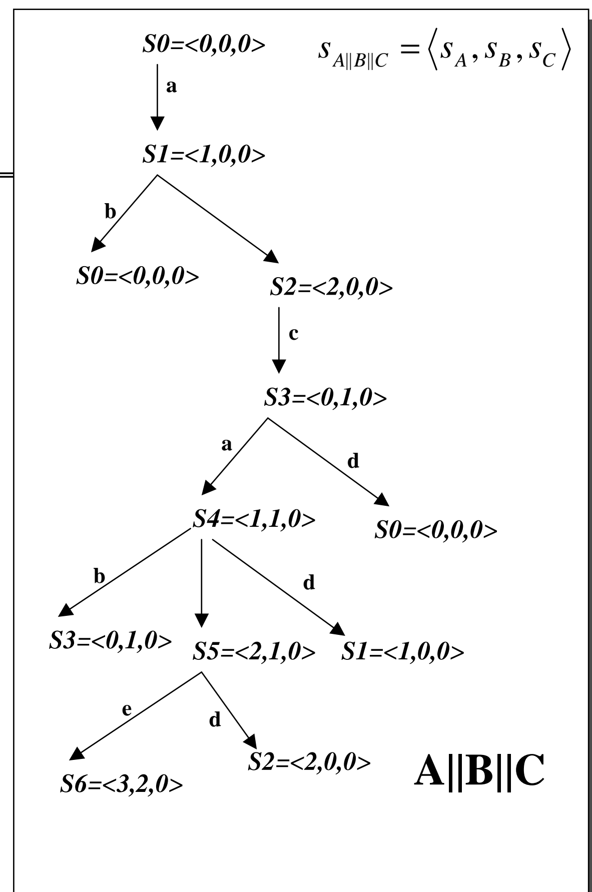
$$s_{A\|B\|C} = \langle s_A, s_B, s_C \rangle$$

Estados do Sistema – Comunicação por Rendez-Vous

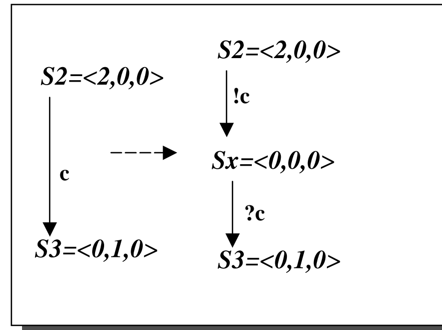




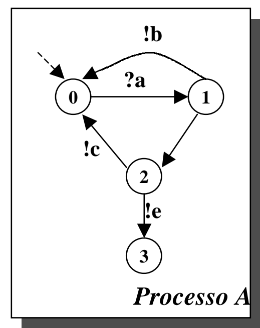




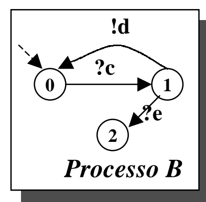
Alternativamente ...

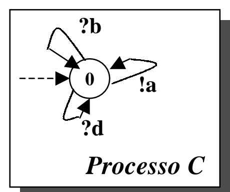



```
mtype = { a,b,c,d,e};
chan ca = [0] of { mtype};
chan ac= [0] of { mtype};
chan cb = [0] of { mtype};
chan bc = [0] of { mtype};
chan ab = [0] of { mtype};
chan ba = [0] of { mtype};

proctype A()
{
    do
    ::ca?a;
        if
        :: skip;
            if
            ::ab!c;
            ::ab!e; break
            fi
        :: ac!b;
        fi
    od
}
```



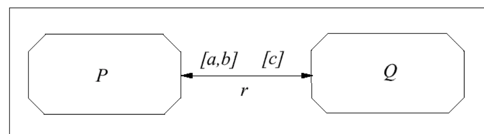

```
proctype B()
{
    do
    ::ab?c;
        if
        :: bc!d;
        :: ab?e -->break
        fi
    od
}
proctype C()
{
end:
    do
    ::bc?d;
    ::ca!a;
    ::ac?b;
    od
}
init {
    atomic {run A(); run B(); run C();}
     }
```

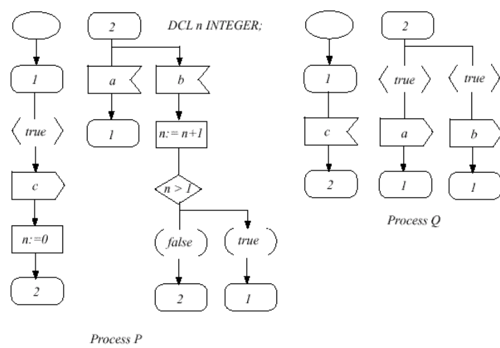


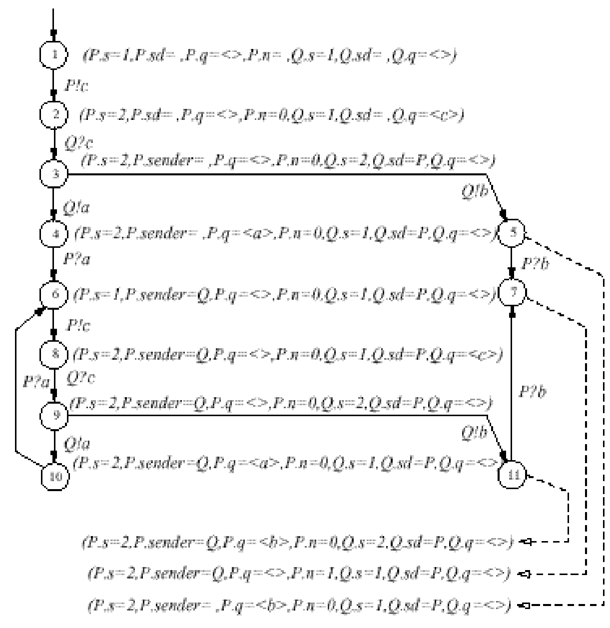


Estados do Sistema – SDL, Comunicação com Filas

state= (P.state, P.sender, P.queue, P.n, Q.state, Q.sender, Q.queue)

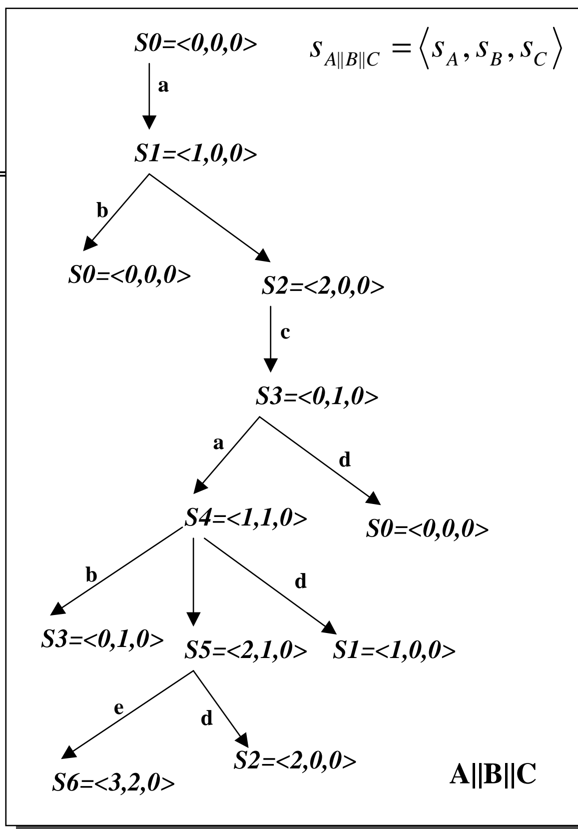






Formalização do Funcionamento – Sequência de Execução

- Sequência de Execução
  - » Caracterização
    - – Conjunto ordenado e finito de estados do sistema
      Ex. se2=<s0,s1,s2,s3,s4,s3>
    - – 1º estado da sequência de execução → estado inicial do sistema (`S0`)
      Variáveis inicializadas a zero. Canais de mensagens vazios. Processo *init* activo
    - – Para cada estado `Si` → existe instrução que coloca sistema em `Si+1`
  - » Tipos de sequências
    - – Terminantes
      - Nenhum estado ocorre mais do que uma vez
      - Não há execução possível a partir do último estado
        Ex. se3=< S0, S1, S2, S3, S4, S5, S6 >
    - – Cíclicas
      - Nenhum estado, excepto o último, ocorre mais do que uma vez
      - Último estado é igual a um dos anteriores
        Ex. **se4=< S0, S1, S2, S3, S4, S5, S2 >**
- Funcionamento de um modelo definido por
  - » Conjunto de todas as sequência de execução
    Ex. M={ se1, se2, se3, se4, se5, se6 }



se1=< S0, S1, S0 >
se2=< S0, S1, S2, S3, S4, S3 >
se3=< S0, S1, S2, S3, S4, S5, S6 >
se4=< S0, S1, S2, S3, S4, S5, S2 >
se5=< S0, S1, S2, S3, S4, S1 >
se6=< S0, S1, S2, S3, S0 >

Sequências de Execução

seqTerminantes={ se3 }
seqCíclicas={ se1, se2, se4, se5, se6 }

Verificação de Funcionamento

- Verificação → demonstração de
  - » existência de uma propriedade no modelo
  - » ***Satisfação de um requisito de funcionamento***
- Requisito de funcionamento definido
  - » Sobre estados
    - – Utilização de Proposições (expressão Booleana)
    - – Proposição (Si) → {Verdadeiro, Falso} → **`assert(condition)`**
      - Executável em todos os estados
      - `condition == True` → expressão não tem efeito
      - `condition == False` → estado violador da condição
  - » Sobre sequências de execução
    - – Etiquetas especiais → **`end, progress, accept`**

assert( ), Invariantes

- Invariante de processo
  - » Neste caso, ambos os requisitos são falsos

```
byte state = 1;
proctype A()
{ (state == 1) -> state = state + 1;
   assert(state == 2)
}
proctype B()
{ (state == 1) -> state = state - 1;
   assert(state == 0)
}
init { run A(); run B() }
```

Neste caso, ambos os requisitos são falsos

- Invariante de sistema

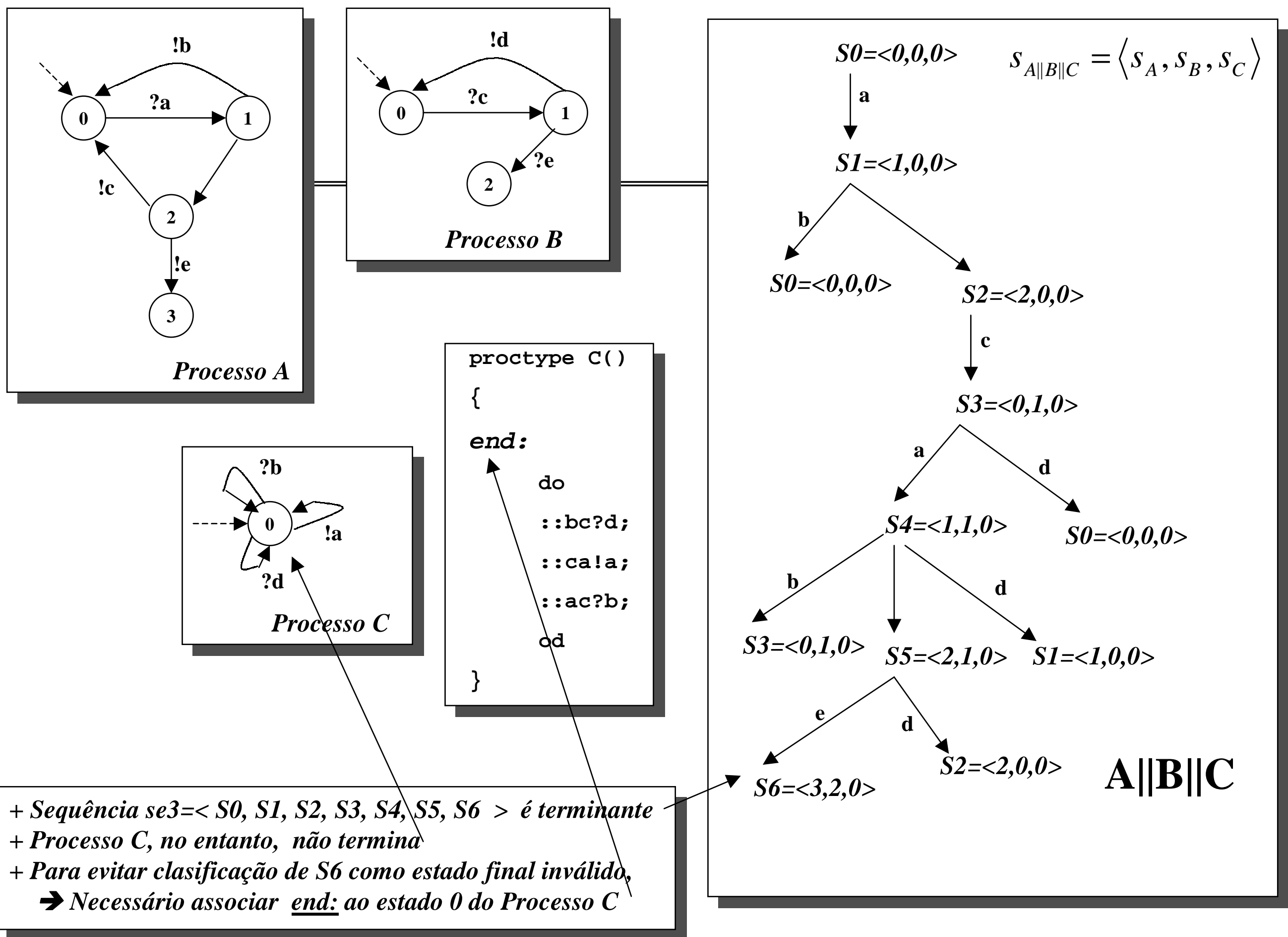
```
proctype monitor()
{
assert(invariante)
}
```

- `Processo monitor,`
  - » executa concorrentemente com outros processos

Sequências de Execução Terminantes

- Sequência de execução terminante ***EX. e3=< S0, S1, S2, S3, S4, S5, S6 >***
- Estado final de sequência de execução terminante ***(S6)***
  - » Válido, se
    - – Todos os processos *terminados*
    - – Todos os canais de mensagens vazios
  - » Inválido,
    - – terminação inesperada, *deadlock*, especificação incompleta
- Processo *terminado*
  - » Normalmente → *}*
  - » Excepções → servidores
    - – Não terminam (*}*)
    - → Introdução de etiqueta de estado final
      `end: , enda:`

```
chan sema = [0] of { byte };
proctype dijkstra()
{
  end: do
       :: sema!p -> sema?v
       od
}
```



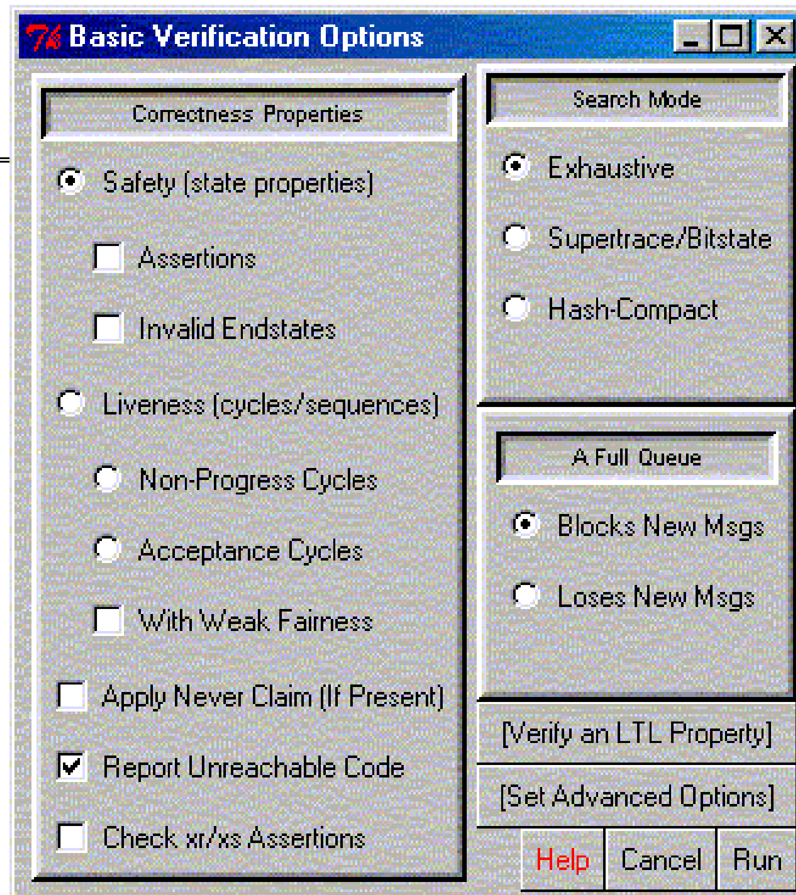
Sequências de Execução Cíclicas

- Válidas → se há progresso
  - » Ex. incremento de um contador, aceitação de uma nova mensagem
  - » Estados marcados nos processos como *progress*

```
chan sema = [0] of { byte};
proctype dijkstra()
{
  end:        do
              :: sema!p ->
  progress:          sema?v
              od
}
```

- Inválidas → se não há progresso
  - » Estados marcados nos processos como *accept*
  - » No exemplo, o requisito é violado porque a sequência é válida

```
chan sema = [0] of { byte};
proctype dijkstra()
{
  end:        do
              :: sema!p ->
  accept:            sema?v
              od
}
```



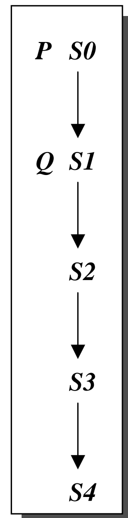
A Primitiva Never

- `never { ...body... }`
  - `never` → semelhante a `proctype`
  - `body` → `especificação de funcionamento`
  - `Usada para descrever funcionamento impossível`
- Descreve 1 requisito temporal
  - Usa proposições. Não especifica funcionamento autónomo
  - Descreve propriedades sobre funcionamento já existente
  - É ***monitor síncrono*** com o resto do sistema
- Funcionamento de *never*
  - Proposição verdadeira → continua observação
  - Proposição falsa → validador trunca execução da sequência → requisito não observado
- A expressão temporal é observada se atinge estado final normal
  - ➔ Detectado erro de funcionamento

Never – Exemplos

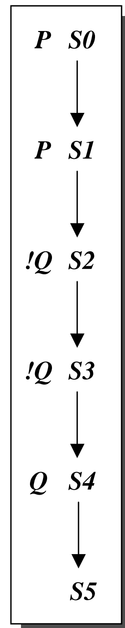
- Erro de funcionamento em que
  - » P é verd no 1º estado da seq
  - » Q é verd no 2º estado da seq

```
never {
    P → Q
}
```



- Um exemplo mais complexo

```
never {
    x:  do
        :: P || !P
        :: P -> break
        od;
    y:  do
        :: !Q
        :: Q -> break
        od;
    z:  skip
}
```



Requisitos Temporais Sobre Sequências Cíclicas

- Detecta uma *má* sequência cíclica em que

  Uma vez identificado um estado que satisfaz P,

  !Q passa a ser satisfeita nos estados seguintes da sequência

```
never {
            do
            :: skip
            :: P -> break
            od;
accept:     do
            :: !Q
            od;
}
```

Requisitos sobre Cíclicas e Terminantes

- Detecta mau comportamento em sequências cíclicas e terminantes

```
never {
            do
            :: skip
            :: P -> break
            od;
accept:     do
            :: !Q
            :: (timeout && !Q) -> break
            od;
}
```