Верификация программ на моделях

Лекция №6

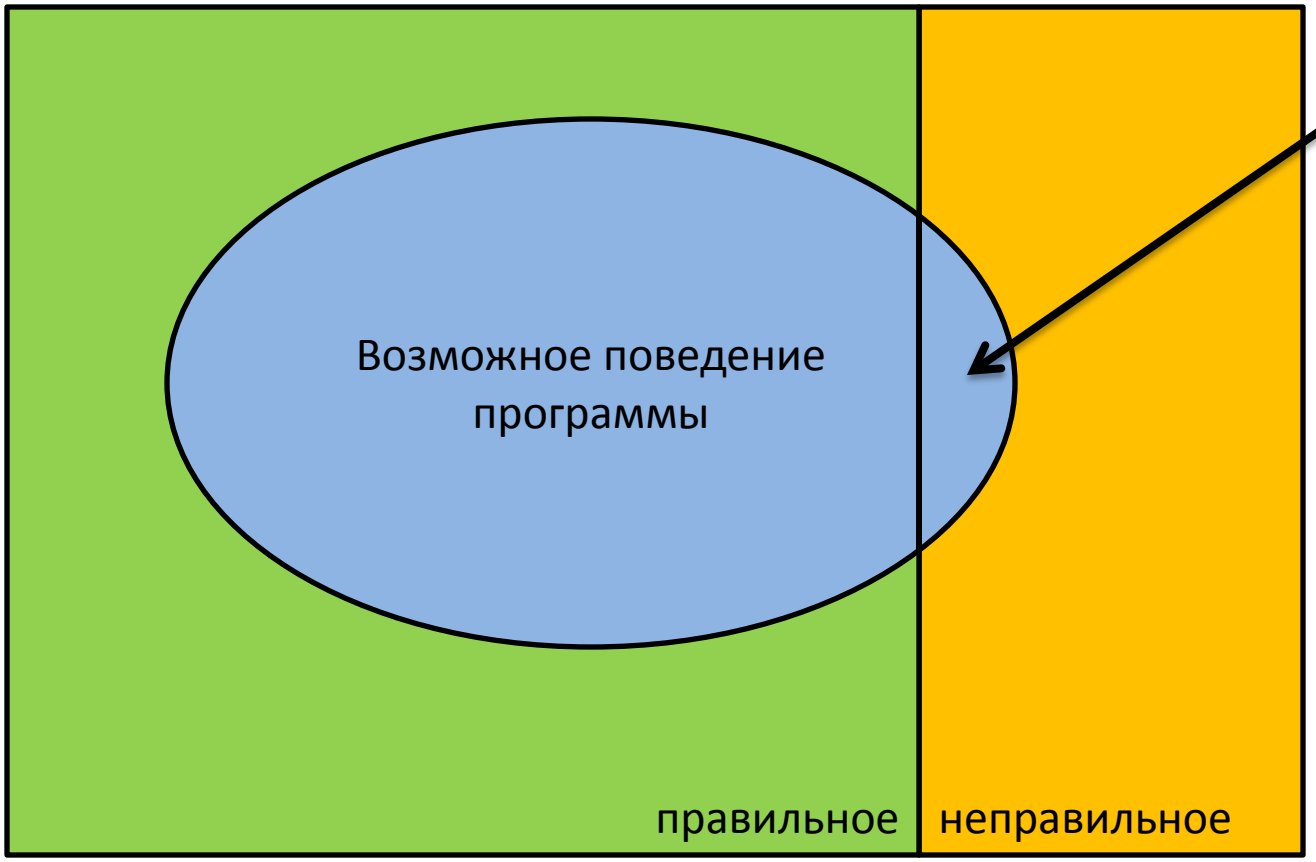
Свойства линейного времени. Спецификация и верификация свойств при помощи Spin.

Константин Савенков (лектор)

План лекции

- Рассуждения о правильности программы,
- Свойства безопасности и живучести,
- Инструменты SPIN/Promela для спецификации свойств.

Верификация программы при помощи модели



Инженерный язык программирования может ограничивать возможности пользователя писать неправильные программы. У языка описания моделей – другие задачи.

Основные компоненты модели в SPIN

- Спецификация поведения *(возможное поведение)*
  - асинхронное выполнение процессов,
  - переменные, типы данных,
  - каналы сообщений;
- Свойства правильности *(правильное поведение)*
  - ассерты,
  - метки завершения и прогресса,
  - утверждения о невозможности (never claims),
  - трассовые ассерты,
  - свойства по умолчанию:
    - отсутствие тупика,
    - отсутствие недостижимого кода,
  - формулы темпоральной логики.

Рассуждения о правильности программы

- Требования правильности задаются как утверждения о возможных или невозможных вариантах выполнения модели,
  - рассуждения о вероятности не допускаются из соображений строгости доказательства;

- Мы утверждает, что некоторые варианты выполнения модели
  - либо **невозможны**,
  - либо **неизбежны**;

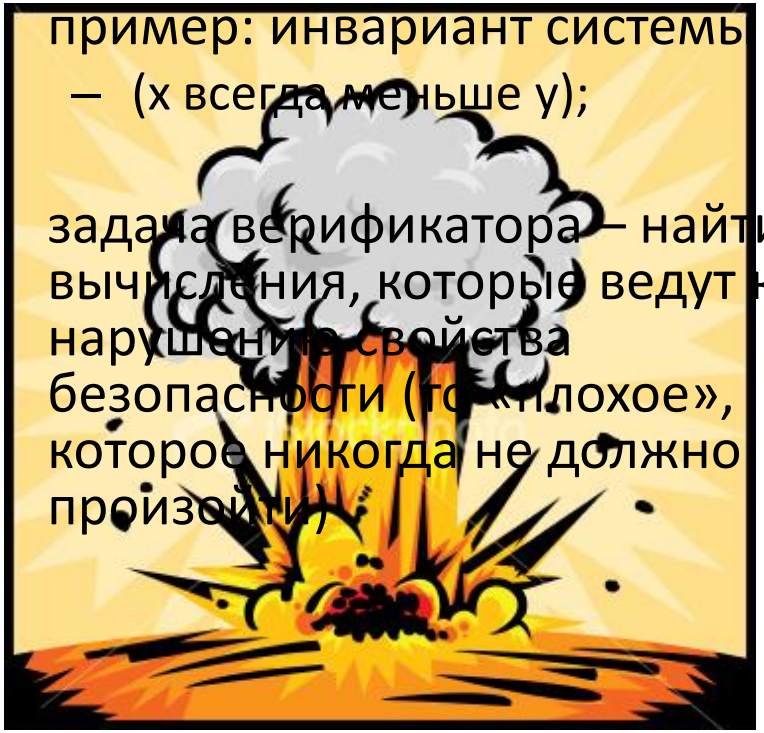
- Двойственность утверждений:
  - если что-то неизбежно, то противоположное – невозможно,
  - если что-то невозможно, то противоположное – неизбежно,
  - используя логику, можно переходить от одного к другому при помощи логического отрицания.

Свойства безопасности и живучести

(автор – Leslie Lamport)

безопасность

- «ничего плохого никогда не произойдёт»;
- пример: инвариант системы
  - (x всегда меньше y);
- задача верификатора – найти вычисления, которые ведут к нарушению свойства безопасности (то «плохое», которое никогда не должно произойти)



живучесть

- «рано или поздно произойдёт что-то хорошее»;
- пример: «отзывчивость»
  - если отправлен запрос, то рано или поздно будет сгенерирован ответ;
- задача верификатора – найти вычисления, в которых это «хорошее» может откладываться до бесконечности

Немного подробнее…

- Свойства безопасности – самые простые свойства («состояние, где истинно φ, недостижимо»).

Достаточно исследовать всё (конечное) множество достижимых состояний

- Можно ли в рамках свойства безопасности сформулировать, что ***φ* неизбежно**?
- **«состояние, где истинно !φ, недостижимо»** – не то, это слишком сильное свойство;
- **«состояние, где истинно φ, достижимо»**

  – а это, наоборот, слишком слабое.

Немного подробнее…

- **«φ неизбежно»** – означает, что **φ обязательно достижима**;
- для спецификации такой модальности и нужны свойства живучести.

Необходимо показать, что свойство достижимо в любом начальном вычислении системы. В частности, нужно исследовать циклы в системе переходов.

Способы описания свойств правильности

- Свойства правильности могут задаваться как:
  - свойства **достижимых состояний** (свойства безопасности),
  - свойства **последовательностей состояний** (свойства живучести);
- В языке Promela
  - ассерты:
    - локальные ассерты процессов,
    - инварианты системы процессов;
  - метки терминальных состояний:
    - задаём допустимые точки останова процессов;

свойства состояний

  - метки прогресса (поиск циклов бездействия);
  - утверждения о невозможности (never claims)
    - например, определяются LTL-формулами;
  - трассовые ассерты.

свойства последова-тельностей состояний

Ассерты – самый древний способ проверить правильность программы

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

```
>spin -a simple.pml
>gcc -o pan pan.c
>./pan -E
pan: assertion violated (state==2) (at depth 6)
pan: wrote simple.pml.trail
...
```

Игнорируем ошибки некорректного останова процесса

```
>spin -t -p simple.pml
Starting A with pid 0
Starting B with pid 1
  1:    proc  1 (B) line   7 "simple.pml" (state 1)     [((state==1))]
  2:    proc  0 (A) line   3 "simple.pml" (state 1)     [((state==1))]
  3:    proc  1 (B) line   7 "simple.pml" (state 2)     [state = (state-1)]
  4:    proc  1 (B) line   8 "simple.pml" (state 3)     [assert((state==0))]
  5: proc 1 terminates
  6:    proc  0 (A) line   3 "simple.pml" (state 2)     [state = (state+1)]
spin: line   4 "simple.pml", Error: assertion violated
...
```

Предотвращение гонки

```
byte state = 1;
active proctype A()
{       atomic((state == 1) -> state++);
        assert(state == 2)
}
active proctype B()
{       atomic((state =
        assert(state ==
```

добавляем две атомарные последовательности

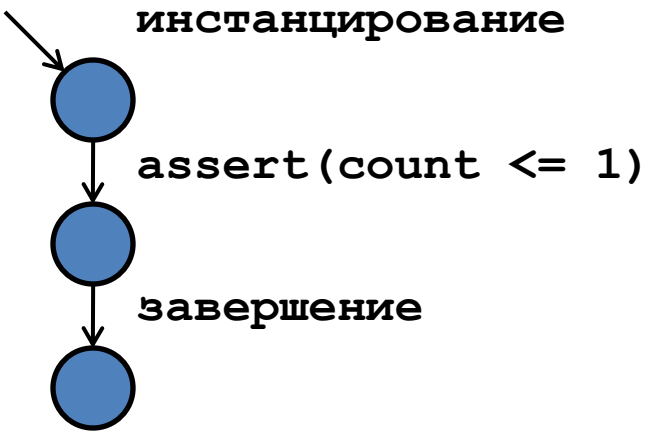
```
>spin -a simple.pml
>gcc -o pan pan.c
>./pan -E
(Spin Version 5.1.4 -- 27 January 2008)
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      - (disabled by -E flag)
State-vector 20 byte, depth reached 3, errors: 0
        6 states, stored
        0 states, matched
        6 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
    2.501       memory usage (Mbyte)
unreached in proctype A
        (0 of 5 states)
unreached in proctype B
        (0 of 5 states)
```

Задание инвариантов системы

```
mtype = {p, v};

chan sem = [0] of {mtype};

byte count;

active proctype semaphore()
{
        do
        :: sem!p ->
           sem?v
        od
}

active proctype user()
{
        do
        :: sem?p;
           count++;
           /*critical section*/
           count--;
           sem!v
        od
}
```

```
active proctype invariant()
{
        assert(count <= 1)
}
```

Сколько «стоит» такая проверка?



Добавление такого инварианта увеличивает пространство поиска

в 3 раза…

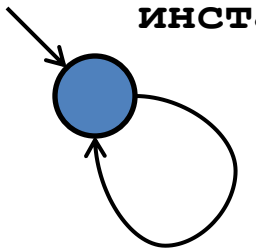
(с 16 достижимых состояний до 48)

Проявляем интуицию

```
mtype = {p, v};

chan sem = [0] of {mtype};

byte count;

active proctype semaphore()
{
        do
        :: sem!p ->
           sem?v
        od
}

active proctype user()
{
        do
        :: sem?p;
           count++;
           /*critical section*/
           count--;
           sem!v
        od
}
```

```
active proctype invariant()
{
    do ::assert(count <= 1) od
}
```

Сколько «стоит» такая проверка?




```
assert(count <= 1)
```

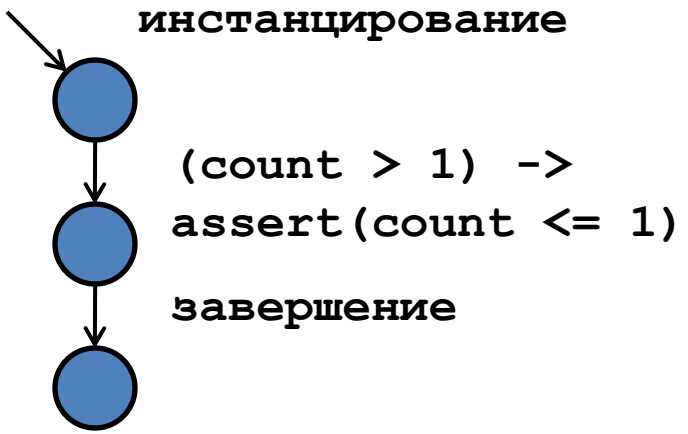
Пространство достижимых состояний не увеличивается, однако добавляется много новых переходов

Ещё лучше…

```
mtype = {p, v};

chan sem = [0] of {mtype};

byte count;

active proctype semaphore()
{
        do
        :: sem!p ->
           sem?v
        od
}

active proctype user()
{
        do
        :: sem?p;
           count++;
           /*critical section*/
           count--;
           sem!v
        od
}
```

```
active proctype invariant()
{
    d_step { !(count <= 1) ->
             assert(count <= 1)}
}
```

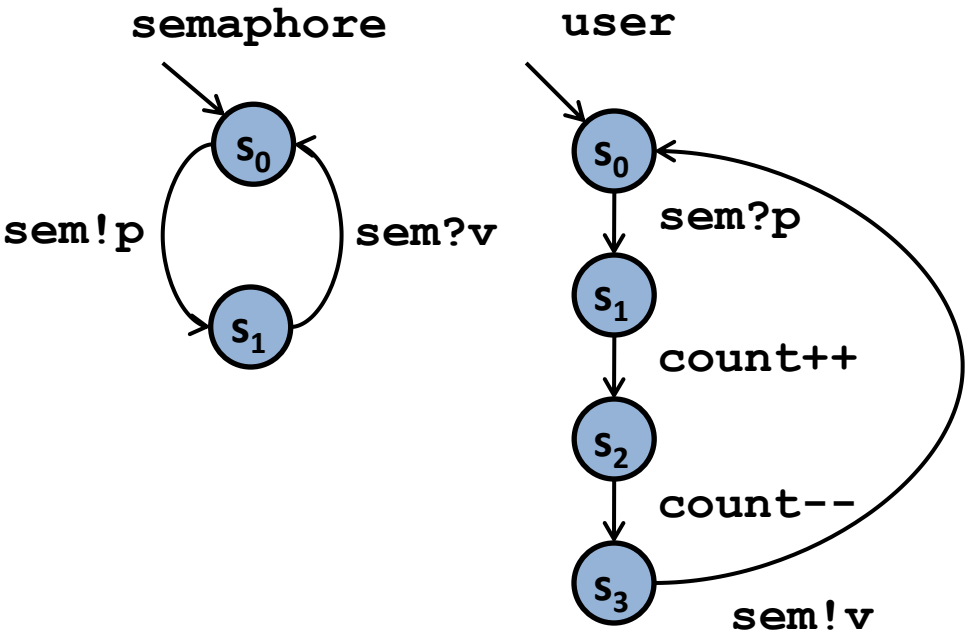
Сколько «стоит» такая проверка?



- Не увеличивает пространство достижимых состояний,
- Не увеличивает количество переходов.

Допустимые состояния останова

```
mtype = {p, v};

chan sem = [0] of {mtype};

byte count;

active proctype semaphore()
{
end:    do
        :: sem!p ->
           sem?v
        od
}

active proctype user()
{
end:    do
        :: sem?p;
           count++;
           /*critical section*/
           count--;
           sem!v
        od
}
```

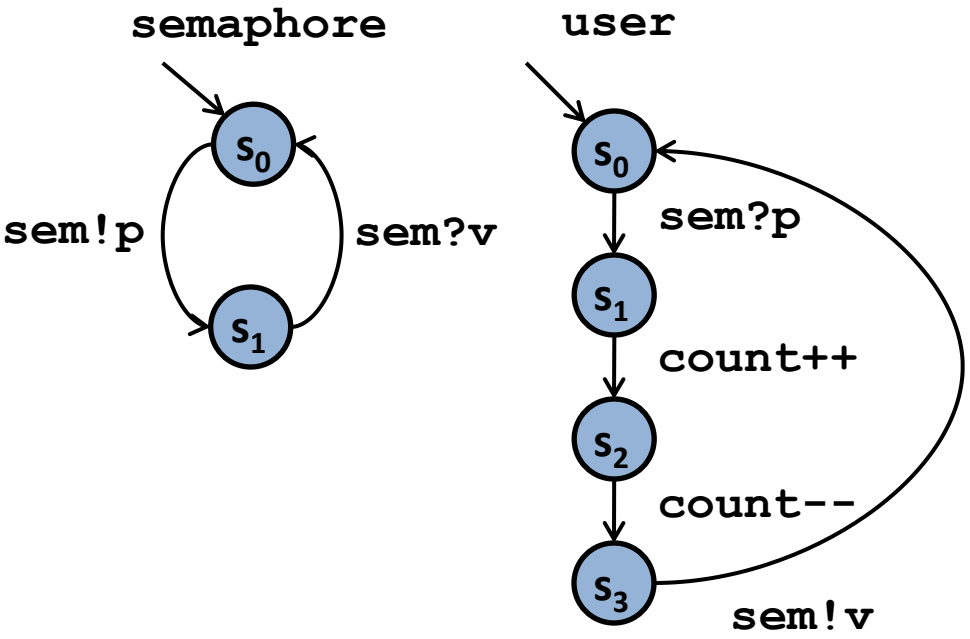


Ни один из процессов не должен завершаться; допустимое состояние останова для обоих процессов – s_0.

Верификатор должен искать недопустимые состояния останова, не отвлекаясь на допустимые.

Состояния прогресса

```
mtype = {p, v};

chan sem = [0] of {mtype};

byte count;

active proctype semaphore()
{
        do
        :: sem!p ->
progress: sem?v
        od
}

active proctype user()
{
        do
        :: sem?p;
           count++;
           /*critical section*/
           count--;
           sem!v
        od
}
```



Система демонстрирует прогресс всякий раз, когда user получает доступ к критической секции, т.е. когда процесс semaphore достигает состояния s_1.

Теперь верификатор будет искать достижимые *циклы бездействия*.

Пример

```
byte x = 2;

active proctype A()
{
    do
    :: x = 3 - x
    od
}

active proctype B()
{
    do
    :: x = 3 - x
    od
}
```

- x бесконечно варьирует между 2 и 1,
- у каждого процесса – одно состояние,
- метки progress не используются => по умолчанию всякий цикл рассматривается как потенциально бездейственный

```
>spin -a fair.pml
>gcc -DNP -o pan pan.c
>./pan -l
pan: non-progress cycle (at depth 2)
pan: wrote fair.pml.trail
(Spin Version 5.1.4 -- 27 January 2008)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim             +
        assertion violations    + (if
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)
-vector 24 byte, depth reached 5, errors: 1
        3 states, stored
        0 states, matched
        3 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
    2.501       memory usage (Mbyte)
```

Обнаружение циклов бездействия

Запуск алгоритма проверки

Q1: Что будет, если отметить один из do-od циклов меткой progress?

Q2: А если разметить оба цикла?

Какой цикл мы обнаружили?

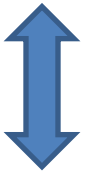
```
>spin -t -p fair.pml
Starting A with pid 0
Starting B with pid 1
spin: couldn't find claim (ignored)
  2:    proc  1 (B) line  13 "fair.pml" (state 1)       [x = (3-x)]
  <<<<<START OF CYCLE>>>>>
  4:    proc  1 (B) line  13 "fair.pml" (state 1)       [x = (3-x)]
  6:    proc  1 (B) line  13 "fair.pml" (state 1)       [x = (3-x)]
spin: trail ends after 6 steps
#processes: 2
                x = 1
  6:    proc  1 (B) line  12 "fair.pml" (state 2)
  6:    proc  0 (A) line   5 "fair.pml" (state 2)
2 processes created
```

Мы не знаем относительно скорости выполнения процессов:

- возможно, что процесс B выполнит бесконечно больше шагов, чем процесс A;
- цикл бездействия, обнаруженный Spin – это не обязательно *справедливый* цикл.

Справедливые циклы

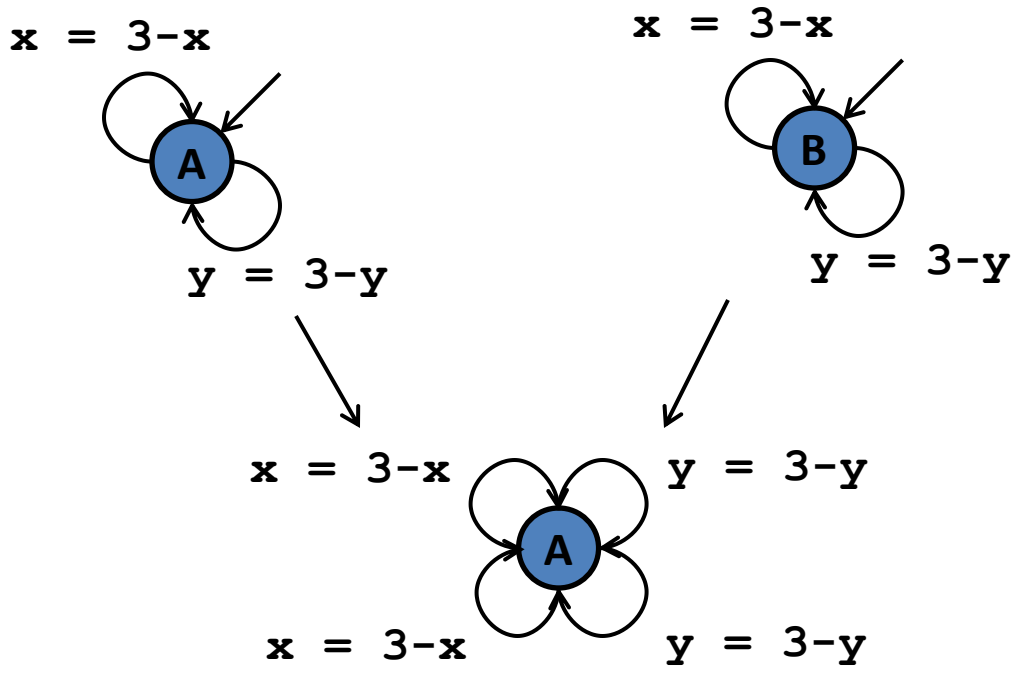
- Часто мы ожидаем *конечного* прогресса:
  - если процесс может сделать шаг, он в конце концов его сделает.
- Существует два основных варианта справедливости:
  1) **слабая** справедливость:
  - если оператор **выполним** бесконечно **долго**, то он в конце концов будет выполнен;
  2) **сильная** справедливость:
  - если оператор **выполним** бесконечно **часто**, то он в конце концов будет выполнен.



- Справедливость применима как к внешнему, так и к внутреннему недетерминизму.

выбор оператора vs. выбор процесса

```
byte x = 2, y = 2;

active proctype A()
{
    do
    :: x = 3 - x;
    :: y = 3 - y
    od
}

active proctype B()
{
    do
    :: x = 3 - x;
    :: y = 3 - y
    od
}
```



SPIN поддерживает один из вариантов слабой справедливости (**>pan -f**):

> «если процесс содержит хотя бы один оператор, который бесконечно долго остаётся выполнимым, то этот процесс рано или поздно сделает шаг».

Это касается только потенциально бесконечных вычислений (циклов).

Поиск слабо справедливых циклов бездействия

```
>./pan -l -f
pan: non-progress cycle (at depth 6)
pan: wrote fair.pml.trail
```

```
>./spin -t -p fair.pml
Starting A with pid 0
Starting B with pid 1
spin: couldn't find claim (ignored)
  2:    proc  1 (B) line  13 "fair.pml" (state 1)       [x = (3-x)]
  4:    proc  1 (B) line  13 "fair.pml" (state 1)       [x = (3-x)]
  6:    proc  0 (A) line   6 "fair.pml" (state 1)       [x = (3-x)]
  <<<<<START OF CYCLE>>>>>
  8:    proc  1 (B) line  13 "fair.pml" (state 1)       [x = (3-x)]
 10:    proc  0 (A) line   6 "fair.pml" (state 1)       [x = (3-x)]
spin: trail ends after 10 steps
#processes: 2
                x = 1
 10:    proc  1 (B) line  12 "fair.pml" (state 2)
 10:    proc  0 (A) line   5 "fair.pml" (state 2)
```

Теперь в цикле есть шаги обоих процессов

Указываем полезные действия

```
byte x = 2, y = 2;

active proctype A()
{
    do
    :: x = 3 - x;
    :: y = 3 - y; progress: skip
    od
}

active proctype B()
{
    do
    :: x = 3 - x;
    :: y = 3 - y; progress: skip
    od
}
```

ВОПРОСЫ:

- Будут ли циклы бездействия с меткой progress в одном процессе?
- А справедливые циклы?
- А если метки есть в обоих процессах?

Использование условий справедливости

- Любой тип справедливости можно описать при помощи формулы LTL (позже)
- добавление предположения о справедливости всегда увеличивает сложность верификации
- использование *сильной* справедливости существенно дороже слабой:
  - слабая: сложность (по времени и памяти) растёт **линейно** от числа активных процессов,
  - сильная: сложность растёт **квадратично**.

Пример: протокол голосования

```
#define N   5  /* nr of processes (use 5 for demos)*/
#define I   3  /* node given the smallest number   */
#define L   10 /* size of buffer  (>= 2*N) */
               /* file ex.8 */
mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte};

byte nr_leaders = 0;

proctype node (chan in, out; byte mynumber)
{   bit Active = 1, know_winner = 0;
    byte nr, maximum = mynumber, neighbourR;
...
}

init {
    byte proc;
    atomic {
            proc = 1;
            do
            :: proc <= N ->
                    run node (q[proc-1], q[proc%N],
                              (N+I-proc)%N+1);
                    proc++
            :: proc > N ->
                    break
            od
    }
}
```

```
out!one(mynumber);
do
:: in?one(nr) ->
   if
   :: Active ->
       if
       :: nr != maximum ->
            out!two(nr);
            neighbourR = nr
       :: else ->
            know_winner = 1;
            out!winner,nr;
       fi
   :: else ->
       out!one(nr)
   fi
:: in?two(nr) ->
   if
   :: Active ->
        if
        :: neighbourR > nr && neighbourR > maximum ->
             maximum = neighbourR;
             out!one(neighbourR)
        :: else ->
             Active = 0
        fi
   :: else ->
        out!two(nr)
   fi
:: in?winner,nr;
   if
   :: nr != mynumber
   :: else -> nr_leaders++;
   fi;
   if
   :: know_winner
   :: else -> out!winner,nr
   fi;
   break
od
```

Запускаем моделирование

```
>spin -c leader.pml
proc 0 = :init:
proc 1 = node
proc 2 = node
proc 3 = node
proc 4 = node
proc 5 = node
q\p   0   1   2   3   4   5
  5   .   .   .   .   .   out!one,5
  3   .   .   .   out!one,2
  2   .   .   out!one,3
  4   .   .   .   .   out!one,1
  2   .   .   in?one,3
  4   .   .   .   .   in?one,1
  1   .   .   .   .   .   out!one,4
  3   .   .   .   in?one,2
  1   .   in?one,4
  5   .   .   .   .   .   in?one,5
  2   .   out!two,4
  1   .   .   .   .   .   out!two,5
  5   .   .   .   .   out!two,1
  3   .   .   out!two,3
  5   .   .   .   .   .   in?two,1
  4   .   .   .   out!two,2
  1   .   in?two,5
  4   .   .   .   .   in?two,2
```

```
  1   .   .   .   .   .   out!one,5
  3   .   .   .   in?two,3
  2   .   .   in?two,4
  1   .   in?one,5
  2   .   out!one,5
  2   .   .   in?one,5
  3   .   .   out!one,5
  3   .   .   .   in?one,5
  4   .   .   .   out!one,5
  4   .   .   .   .   in?one,5
  5   .   .   .   .   out!one,5
  5   .   .   .   .   .   in?one,5
  1   .   .   .   .   .   out!winner,5
  1   .   in?winner,5
  2   .   out!winner,5
  2   .   .   in?winner,5
  3   .   .   out!winner,5
  3   .   .   .   in?winner,5
  4   .   .   .   out!winner,5
  4   .   .   .   .   in?winner,5
  5   .   .   .   .   out!winner,5
  5   .   .   .   .   .   in?winner,5
-------------
final state:
-------------
6 processes created
```

Верификация по умолчанию

```
>./spin -a leader.pml
>./gcc -o pan pan.c
>./pan
(Spin Version 5.1.4 -- 27 January 2008)
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +
State-vector 200 byte, depth reached 106, errors: 0
     4813 states, stored
     1824 states, matched
     6637 transitions (= stored+matched)
       12 atomic steps
hash conflicts:         13 (resolved)
Stats on memory usage (in Megabytes):
    0.991       equivalent memory usage for states (stored*(State-vector + overhead))
    1.070       actual memory usage for states (unsuccessful compression: 107.92%)
                state-vector as stored = 217 byte + 16 byte overhead
    2.000       memory used for hash table (-w19)
    0.305       memory used for DFS stack (-m10000)
    3.282       total actual memory usage
unreached in proctype node
        line 40, state 26, "out!two,nr"
        (1 of 44 states)
unreached in proctype :init:
        (0 of 11 states)
pan: elapsed time 0.03 seconds
pan: rate 160433.33 states/second
```

Ошибок нет, но мы ещё не сказали, что должен делать алгоритм

Похоже, в модели есть недостижимый код

Свойства правильности

- При помощи ассертов можно проверить два простых факта:
  - должен выиграть процесс с максимальным номером,
  - должен быть только один победитель.
- Правда ли оператор отправки сообщения недостижим?

```
out!one(mynumber);
do
:: in?one(nr) ->
   if
   :: Active ->
       if
       :: nr != maximum ->
            out!two(nr);
            neighbourR = nr
       :: else ->
            know_winner = 1;
            out!winner,nr;
       fi
   :: else ->
       out!one(nr)
   fi
:: in?two(nr) ->
   if
   :: Active ->
        if
        :: neighbourR > nr && neighbourR > maximum ->
             maximum = neighbourR;
             out!one(neighbourR)
        :: else ->
             Active = 0
        fi
   :: else ->
        out!two(nr); assert(false)
   fi
:: in?winner,nr -> assert(nr == N) ;
   if
   :: nr != mynumber
   :: else -> nr_leaders++; assert(nr_leaders == 1)
   fi;
   if
   :: know_winner
   :: else -> out!winner,nr
   fi;
   break
od
```

Более осмысленная верификация

```
>./spin -a leader.pml
>./gcc -DSAFETY -o pan pan.c
>./pan
(Spin Version 5.1.4 -- 27 January 2008)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -
DSAFETY)
        invalid end states      +

State-vector 196 byte, depth reached 112, errors: 0
     4819 states, stored
     1824 states, matched
     6643 transitions (= stored+matched)
       12 atomic steps
hash conflicts:         5 (resolved)

    3.379       memory usage (Mbyte)

unreached in proctype node
        line 40, state 26, "out!two,nr"
        line 40, state 27, "assert(0)"
        (2 of 47 states)
unreached in proctype :init:
        (0 of 11 states)

pan: elapsed time 0.01 seconds
```

Проверяем простейшие свойства безопасности

Появились ассерты

Ошибок не найдено!

Число состояний увеличилось: ассерты

Действительно, недостижимое состояние

Alternating Bit Protocol

(с потерей сообщения и таймаутом)

```
mtype = {msg, ack};
chan to_s = [1] of {mtype, bit};
chan to_r = [1] of {mtype, bit};
chan from_s = [1] of {mtype, bit};
chan from_r = [1] of {mtype, bit};

active proctype sender()
{ bit a;
  do
  :: from_s!msg,a ->
      if
      :: to_s?ack,eval(a) ->
          a = 1 - a
      :: timeout
      fi
  od
}

active proctype receiver()
{ bit a;
  do
  :: to_r?msg,eval(a) ->
      from_r!ack,a;
progress:
      a = 1 - a
  od
}
```

```
active proctype channel()
{ mtype m; bit a;
  do
  :: from_s?m,a ->
      if
      :: to_r!m,a
      :: skip
      fi
  :: from_r?m,a ->
      to_s!m,a
  od
}
```

Теряем сообщение

Повторная отправка

Будет ли алгоритм эффективно работать при потере сообщений?

Проверяем!

```
> ./spin -a abp_lossy.pml
> gcc -DNP -o pan pan.c
> ./pan -l
pan: non-progress cycle (at depth 4)
pan: wrote abp_lossy.pml.trail
```

В найденном сценарии мы бесконечно часто теряем сообщение

Хорошо бы рассмотреть и другие сценарии!

```
./spin -t -c abp_lossy.pml
proc 0 = sender
proc 1 = receiver
proc 2 = channel
spin: couldn't find claim (ignored)
q\p   0   1
  1   from_s!msg,0
  1   .   .   from_s?msg,0
  <<<<<START OF CYCLE>>>>>
  1   from_s!msg,0
  1   .   .   from_s?msg,0
spin: trail ends after 12 steps
-------------
final state:
-------------
#processes: 3
                queue 1 (from_s):
 12:    proc  2 (channel) line  33 "abp_lossy.pml" (state 4)
 12:    proc  1 (receiver) line  21 "abp_lossy.pml" (state 4)
 12:    proc  0 (sender) line  11 "abp_lossy.pml" (state 5)
3 processes created
```

Уточняем свойство

```
active proctype channel()
{ mtype m; bit a;
  do
  :: from_s?m,a ->
      if
      :: to_r!m,a
      :: skip; progress: skip
      fi
  :: from_r?m,a ->
      to_s!m,a
  od
}

active proctype receiver()
{ bit a;
  do
  :: to_r?msg,eval(a) ->
      from_r!ack,a;
progress:
      a = 1 - a
  od
}
```

Размечаем цикл потери сообщений меткой прогресса, исключая из поиска

Уточнённая проверка

```
> ./spin -a abp_lossy2.pml
> gcc -DNP -o pan pan.c
> ./pan -l
(Spin Version 5.1.4 -- 27 January 2008)
        + Partial Order Reduction
Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)
State-vector 60 byte, depth reached 53, errors: 0
       73 states, stored (98 visited)
       64 states, matched
      162 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
    2.501       memory usage (Mbyte)
unreached in proctype sender
        line 17, state 10, "-end-"
        (1 of 10 states)
unreached in proctype receiver
        line 27, state 7, "-end-"
        (1 of 7 states)
unreached in proctype channel
        line 40, state 12, "-end-"
        (1 of 12 states)
```

Единственный цикл бездействия связан с бесконечной потерей сообщений

Пример: протокол голосования

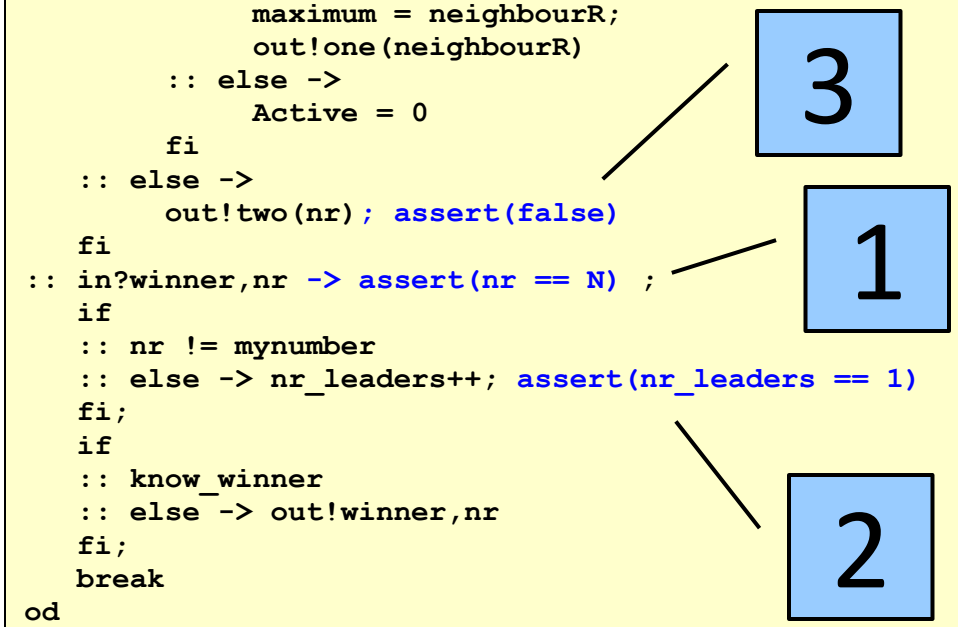
```
#define N   5  /* nr of processes (use 5 for demos)*/
#define I   3  /* node given the smallest number   */
#define L   10 /* size of buffer  (>= 2*N) */
               /* file ex.8 */
mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte};

byte nr_leaders = 0;

proctype node (chan in, out; byte mynumber)
{   bit Active = 1, know_winner = 0;
    byte nr, maximum = mynumber, neighbourR;
...
}

init {
    byte proc;
    atomic {
            proc = 1;
            do
            :: proc <= N ->
                    run node (q[proc-1], q[proc%N],
                              (N+I-proc)%N+1);
                    proc++
            :: proc > N ->
                    break
            od
    }
}
```

```
out!one(mynumber);
do
:: in?one(nr) ->
   if
   :: Active ->
       if
       :: nr != maximum ->
            out!two(nr);
            neighbourR = nr
       :: else ->
            know_winner = 1;
            out!winner,nr;
       fi
   :: else ->
       out!one(nr)
   fi
:: in?two(nr) ->
   if
   :: Active ->
        if
        :: neighbourR > nr && neighbourR > maximum ->
             maximum = neighbourR;
             out!one(neighbourR)
        :: else ->
             Active = 0
        fi
   :: else ->
        out!two(nr)
   fi
:: in?winner,nr;
   if
   :: nr != mynumber
   :: else -> nr_leaders++;
   fi;
   if
   :: know_winner
   :: else -> out!winner,nr
   fi;
   break
od
```

Свойства правильности

- Кое-что мы уже проверили при помощи ассертов:
  1. Выигрывает процесс с максимальным номером.
  2. Должен быть только один победитель.
  3. Оператор отправки сообщения не достижим.

```
out!one(mynumber);
do
:: in?one(nr) ->
   if
   :: Active ->
       if
       :: nr != maximum ->
            out!two(nr);
            neighbourR = nr
       :: else ->
            know_winner = 1;
            out!winner,nr;
       fi
   :: else ->
       out!one(nr)
   fi
:: in?two(nr) ->
   if
   :: Active ->
        if
        :: neighbourR > nr && neighbourR > maximum ->
             maximum = neighbourR;
             out!one(neighbourR)
        :: else ->
             Active = 0
        fi
   :: else ->
        out!two(nr); assert(false)
   fi
:: in?winner,nr -> assert(nr == N) ;
   if
   :: nr != mynumber
   :: else -> nr_leaders++; assert(nr_leaders == 1)
   fi;
   if
   :: know_winner
   :: else -> out!winner,nr
   fi;
   break
od
```



Циклы бездействия

- Свойство:
  - Программа эффективно работает, пока увеличивается значение переменной maximum.
- Проверяем:

```
> ./spin -a leader2.pml
> gcc -DNP -o pan pan.c
> ./pan -l
...
```

(циклов бездействия не найдено)

```
out!one(mynumber);
do
:: in?one(nr) ->
   if
   :: Active ->
       if
       :: nr != maximum ->
            out!two(nr);
            neighbourR = nr
       :: else ->
            know_winner = 1;
            out!winner,nr;
       fi
   :: else ->
       out!one(nr)
   fi
:: in?two(nr) ->
   if
   :: Active ->
        if
        :: neighbourR > nr && neighbourR > maximum ->
progress:    maximum = neighbourR;
             out!one(neighbourR)
        :: else ->
             Active = 0
        fi
   :: else ->
        out!two(nr)
   fi
:: in?winner,nr;
   if
   :: nr != mynumber
   :: else -> nr_leaders++
   fi;
   if
   :: know_winner
   :: else -> out!winner,nr
   fi;
   break
od
```

Конструкции **never** (отрицание свойств)

Never say never

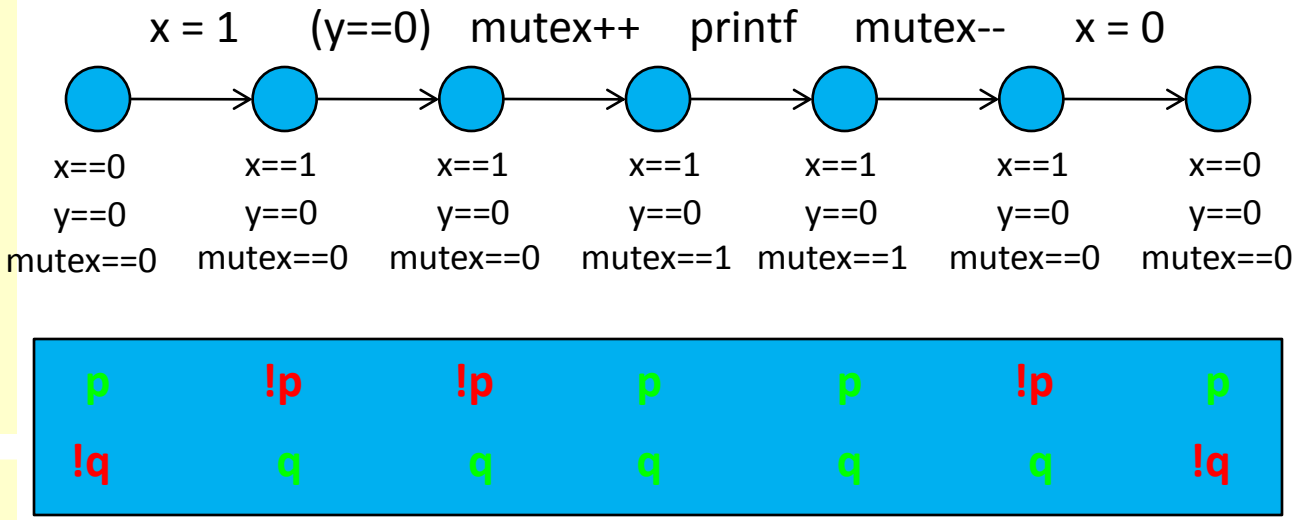
(народная пословица)

Рассуждения о вычислениях программы

- Существует несколько вариантов формализации вычислений распределённой системы:
  - последовательность состояний,
  - последовательность событий (переходов),
  - последовательность значений высказываний в состояниях (свойства состояний) – **трассы**.

```
bit x,y;
byte mutex;
active proctype A()
{
  x = 1;
  (y == 0) ->
  mutex++;
  printf("%d\n", _pid);
  mutex--;
  x = 0
}
```

```
p: (x == mutex)
q: (x != y)
```



Пример

- «не существует вычисления, в котором за p следует q»

```
active proctype invariant()
{
  assert(!p || !q);
}
```

НЕПРАВИЛЬНО!
Свойства только для одного состояния

```
active proctype invariant()
{
  p;
  do
  ::assert(!q);
  od
}
```

НЕПРАВИЛЬНО!
Асинхронное выполнение

never claims

(утверждения о невозможности)

- выполняются *синхронно* с моделью,
- если достигнут конец, то – ошибка,
- состоят из выражений и конструкций задания потока управления,
- фактически, описывают *распознающий автомат*.

Пример

- «не существует вычисления, в котором за p следует q»

```
never
{
  p;q
}
```

НЕПРАВИЛЬНО!
Синхронное выполнение
– будет работать только для первых двух состояний

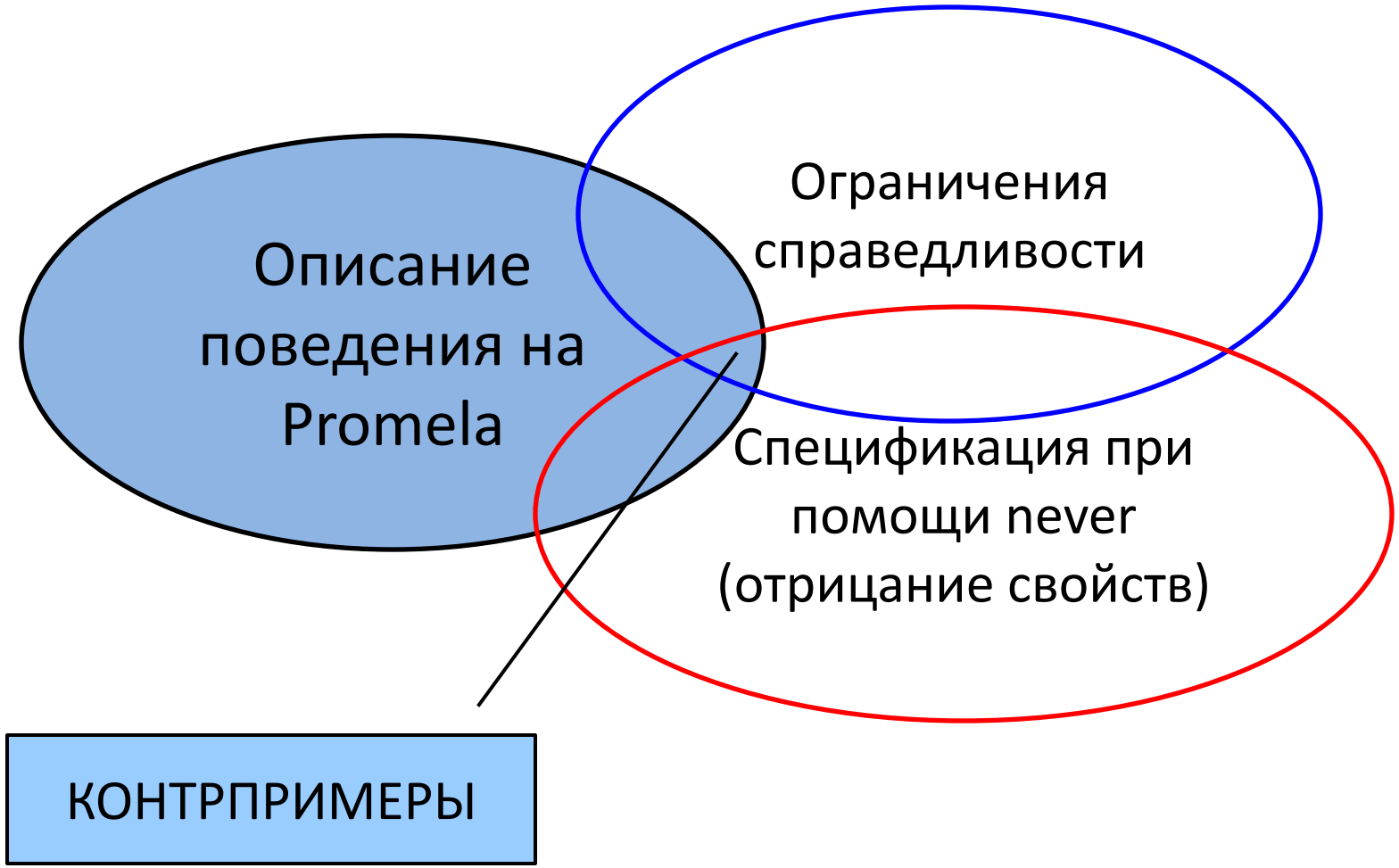
```
never
{
  do
  :: p -> break
  od
  do
  :: q -> break
  od
}
```

ПРАВИЛЬНО!

Конструкция never

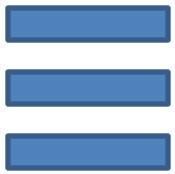
- может быть как детерминированной, так и нет;
- содержит **только** выражения без побочных эффектов (соотв. булевым высказываниям на состояниях);
- используются для описания **неправильного** поведения системы;
- прерывается при блокировании:
  - блокируется => наблюдаемое поведение не соответствует описанному,
  - паузы в выполнении тела never должны быть явно заданы как бесконечные циклы;
- never нарушается, если:
  - достигнута закрывающая скобка,
  - завершена конструкция accept (допускающий цикл);
- бездействие может быть описано как конструкция never или её часть (для обнаружения циклов бездействия есть тело never «по умолчанию»).

Пересечение множеств трасс (языков)



Проверка инварианта системы при помощи конструкции never

```
never
{
  do
  :: invariant
  :: else -> break
  od
}
```



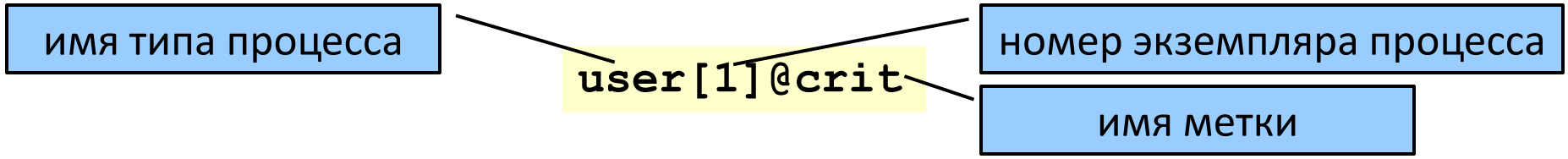
```
never
{
  do
  :: assert(invariant)
  od
}
```

```
never
{
  do
  :: atomic{ !invariant ->
        assert(invariant)
     }
  od
}
```

Ссылки на точки процессов

из тела never

- из тела never можно сослаться на точку (состояние управления) любого активного процесса;
- синтаксис такой ссылки:
  - **`proctypename[pidnr]@labelname`**
- это выражение истинно только если процесс с номером *pidnr* находится в точке описания типа процесса *proctypename*, размеченной меткой *labelname*;



- если существует только один процесс типа user, то можно опустить часть [pidnr]:

`user@crit`

Ссылки на точки процессов (пример)

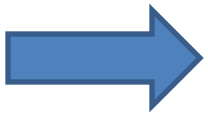
Используем метки управления вместо счётчика процессов

```
never
{
  do
  :: user[1]@crit && user[2]@crit -> break
  :: else
  od
}
```

```
mtype = {p, v};
chan sem = [0] of { mtype };

active proctype semaphore()
{
  do sem!p ; sem!v od
}

active [2] proctype user()
{  assert(_pid == 1 || _pid == 2);
   do
   :: sem?p ->
crit:   /*критическая секция*/
       sem?v
   od
}
```

Проверяем, что процесс завершился

```
active proctype runner()
{
  do
  :: ... ...
  :: else -> break
  od
}
```



```
active proctype runner()
{
  do
  :: ... ...
  :: else -> break
  od;
L:(false)
}
```

```
runner@L
```

Конструкции never:

- могут содержать любые конструкции потока управления:
  - if, do, unless, atomic, d_step, goto;
- должны содержать только выражения:
  - т.е. q?[ack] или nfull(q), но не q?ack или q!ack;
- не должны содержать меток progress и end;
- нужно аккуратно использовать never вместе с метками progress;
- могут использоваться для фильтрации интересующего нас поведения:

```
never
{
  do
  :: atomic {(p || q) -> assert(r)}
  od
}
```

Проверяем **`assert(r)`** на каждом шаге, но лишь для тех вычислений, где одновременно выполняются p и q.

Видимость

- все конструкции never – глобальны;
- тем самым, в них можно ссылаться на
  - глобальные переменные,
  - каналы сообщений,
  - точки описания процессов (метки),
  - предопределённые глобальные переменные,
  - но **не** локальные переменные процессов;
- **нельзя** ссылаться на **события** (действия), только на **состояния**. **А если очень хочется?**

Ассерты на трассы

- Используются для описания правильных и неправильных последовательностей выполнения операторов send и receive.

```
mtype = {a, b };

chan p = [2] of mtype;
chan q = [1] of mtype;

trace {
  do
  :: p!a; q?b
  od
}
```

Этот ассерт фиксирует лишь взаимный порядок выполнения операций посылки сообщений в канал p и приёма сообщений по каналу q.

Он утверждает, что каждая отправка сообщения a в канал p сопровождается получением сообщения b из канала q.

Отклонение от этой схемы приветёт к сообщению об ошибке.

Если в ассерте упоминается хотя бы одна операция отправки сообщения в канал q, ему должны соответствовать все подобные операции

В ассертах на трассы могу использоваться лишь операторы отправки и получения сообщений.

Не могут использоваться переменные, только константы, mtype или _

q?_ используется для обозначения приёма любого сообщения

Пример

Верно ли, что в протоколе голосования типы сообщений **one**, **two** и **winner** приходят в строгом порядке, так что никто не увидит сообщение **one** после сообщения **two**?

```
trace {
  do
  :: q[0]?one,_
  :: q[0]?two,_ -> break
  od;
  do
  :: q[0]?two,_
  :: q[0]?winner,_ -> break
  od
}
```

Верификация
(неправда!)

```
> ./spin -a leader_trace.pml
> gcc -o pan pan.c
> ./pan
pan: event_trace error (no matching event) (at depth 64)
pan: wrote leader_trace.pml.trail

(Spin Version 5.1.4 -- 27 January 2008)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        trace assertion         +
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +
State-vector 200 byte, depth reached 63, errors: 1
       52 states, stored
        0 states, matched
       52 transitions (= stored+matched)
       12 atomic steps
hash conflicts:         0 (resolved)
    2.501       memory usage (Mbyte)
pan: elapsed time 0 seconds
```

Как же так?

Ассерт нарушен!

```
> ./spin -t -c leader_trace.pml
proc 0 = :init:
proc 1 = node
proc 2 = node
proc 3 = node
proc 4 = node
proc 5 = node
q\p   0   1   2   3   4   5
  1   .   .   .   .   .   out!one,4
  5   .   .   .   .   out!one,5
  5   .   .   .   .   .   in?one,5
  1   .   .   .   .   .   out!two,5
  4   .   .   .   out!one,1
  4   .   .   .   .   in?one,1
  5   .   .   .   .   out!two,1
  5   .   .   .   .   .   in?two,1
  1   .   .   .   .   .   out!one,5
  3   .   .   out!one,2
  3   .   .   .   in?one,2
  4   .   .   .   out!two,2
  4   .   .   .   .   in?two,2
  2   .   out!one,3
  2   .   .   in?one,3
  3   .   .   out!two,3
  3   .   .   .   in?two,3
  1   .   in?one,4
  2   .   out!two,4
  2   .   .   in?two,4
  1   .   in?two,5
  1   .   in?one,5
spin: trail ends after 64 steps
```

Ассерты notrace

- обратное утверждение: ассерт notrace утверждает, что описанный шаблон поведения невозможен

```
mtype = {a, b };

chan p = [2] of mtype;
chan q = [1] of mtype;

notrace {
  do
  :: p!a; q?b
  :: q?b; p!a
  od
}
```

Этот ассерт утверждает, что не существует вычисления, в котором отправка сообщения a в канал p сопровождается получением сообщения b из q, и наоборот.

Сообщение об ошибке генерируется, если достигнута закрывающая фигурная скобка ассерта notrace.

О невозможном и неизбежном

- **ассерт** формализует утверждение:
  - указанное выражение **не может** принимать значение ложь, если достигнут ассерт;
- **метка end** формализует утверждение:
  - система **не может** завершить работу без того, чтобы все активные процессы либо завершились, либо остановились в точках, помеченных метками end;
- **метка progress** формализует утверждение:
  - система **не может** выполняться бесконечно без того, чтобы проходить через точку, помеченную меткой progress бесконечно часто;
- **конструкция never** формализует утверждение:
  - система **не может** демонстрировать поведение (конечное или бесконечное), полностью совпадающее с описанным в теле never;
- **ассерт на трассах** формализует утверждение:
  - система **не может** демонстрировать поведение, отличное от описанного шаблона.

Спасибо за внимание!
Вопросы?