

Система верификации Spin. Язык Promela

Данилов И. Г.
к.т.н., ассистент каф. МОП ЭВМ

Институт компьютерных технологий и информационной безопасности ЮФУ

23 октября 2015 г.

Начнем с упражнений

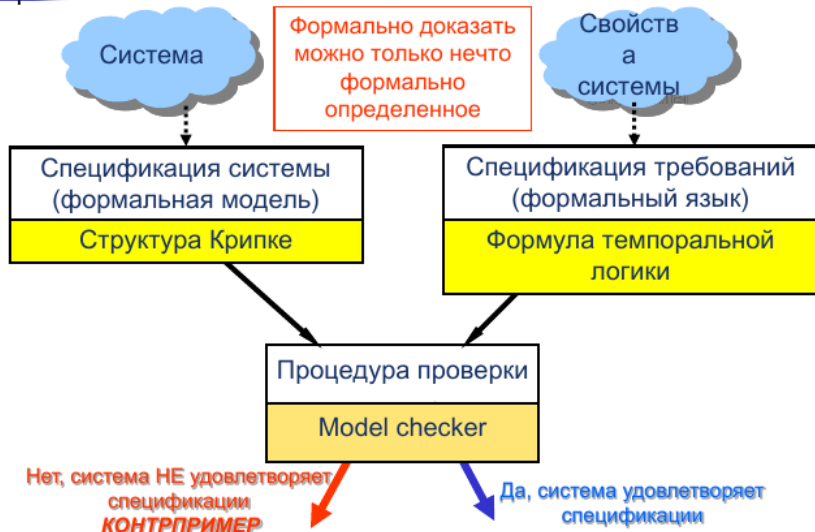
Известно, что если в формуле Φ LTL каждый темпоральный оператор будет предварен квантором пути A , то такая формула CTL не всегда будет эквивалентна исходной формуле Φ .

- 1 Проверьте, выполняются ли LTL-формула FGp и CTL-формула $AFAGp$ на структуре Крипке $M1 = (S, S_0, R, AP, L)$, где:
 $S = \{s_0, s_1, s_2\}$, $S_0 = \{s_0\}$, $R = \{(s_0, s_0), (s_0, s_1), (s_1, s_2), (s_2, s_2)\}$,
 $AP = \{p\}$, $L(s_0) = L(s_2) = \{p\}$, $L(s_1) = \emptyset$.
- 2 Проверьте, выполняются ли LTL-формула GFp и CTL-формула $AGAFp$ на структуре Крипке $M2 = (S, S_0, R, AP, L)$, где:
 $S = \{s_0, s_1\}$, $S_0 = \{s_0\}$, $R = \{(s_0, s_0), (s_0, s_1), (s_1, s_1)\}$, $AP = \{p\}$,
 $L(s_0) = \emptyset$, $L(s_1) = \{p\}$.

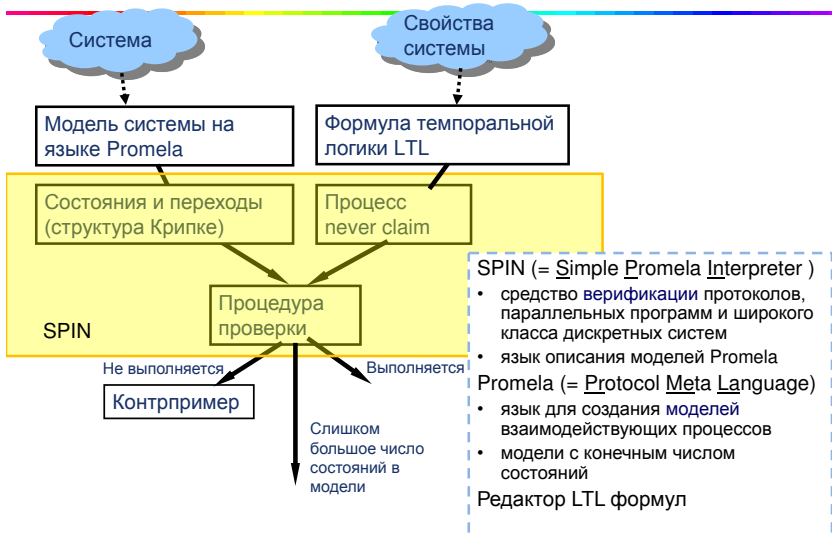
Используемые материалы:

- Материалы лекций Константина Савенкова, ВМиК МГУ, курс “Верификация программ на моделях”;
- Верификация параллельных и распределенных программных систем, лекция в Comp. Sci. Club 18.03.2012, Ю.Г. Карпов, И.В. Шошмина, А. Б. Беляев;
- Моделирование и верификация распределенных систем в среде SPIN, лекция в Comp. Sci. Club 18.03.2012, И.В. Шошмина, Ю.Г. Карпов, А. Б. Беляев;
- Model Checking. Верификация параллельных и распределенных программных систем, Ю.Г. Карпов.

Общая схема верификации Model checking



Средство верификации SPIN



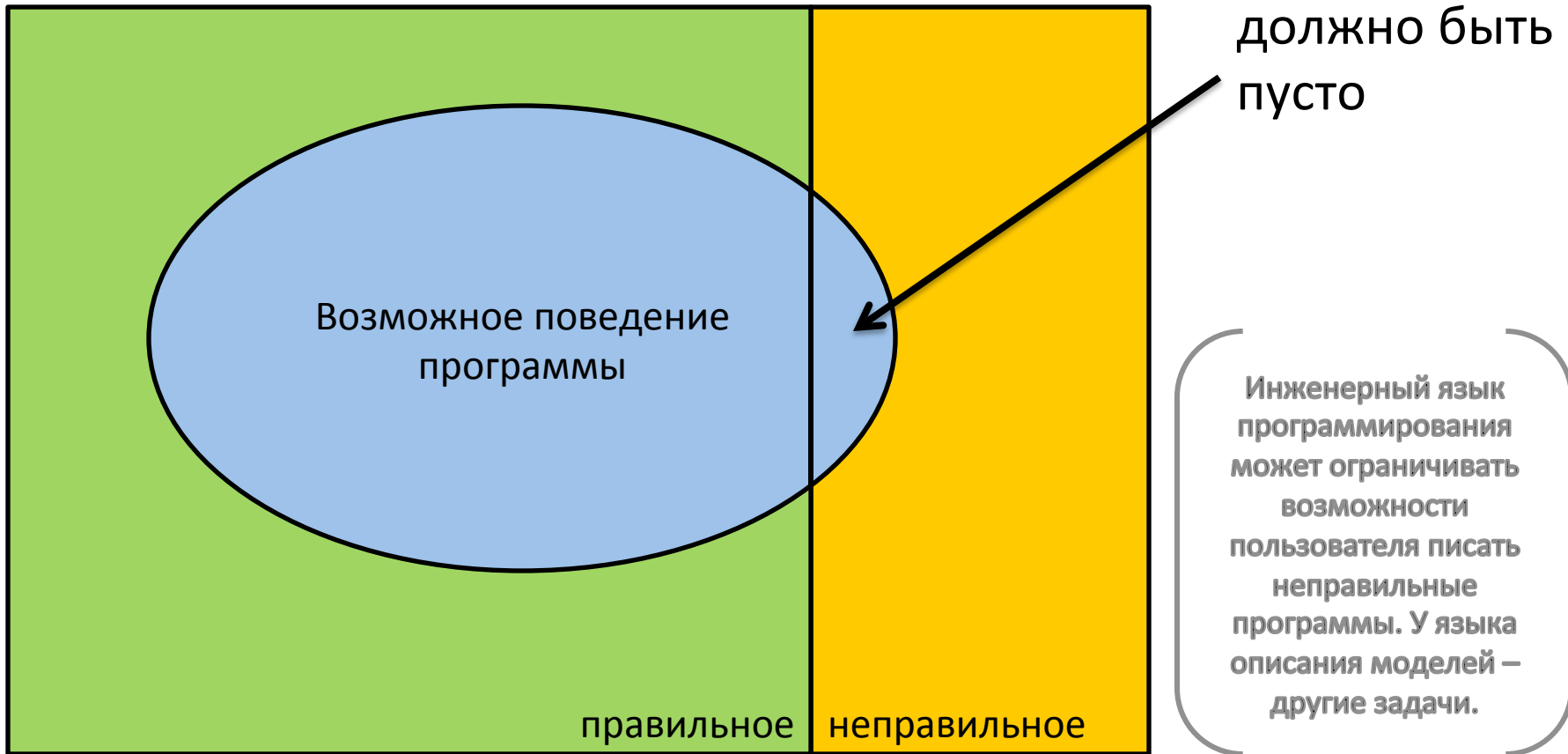
Верификация программ на моделях

Лекция №4

Среда верификации SPIN. Описание
моделей на языке Promela.

Константин Савенков (лектор)

Верификация программы при помощи модели



SPIN, Promela, LTL

- SPIN:

- моделирование,
- верификация;

[реализация недетерминизма]

- Promela:

- недетерминированный язык с охраняемыми командами,

[расширяет возможности абстракции]

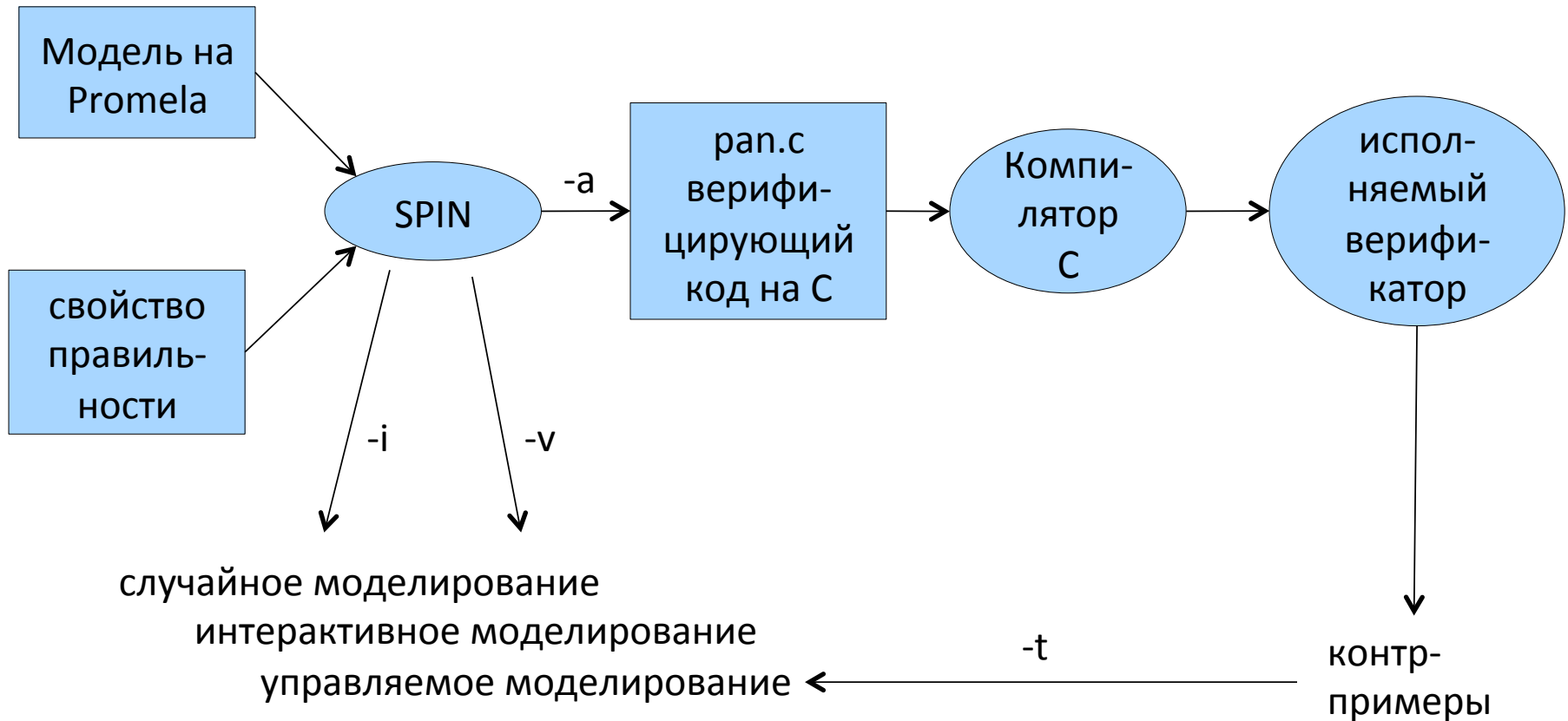
- задача языка – **не** предотвратить описание моделей плохих программ,

[goto поддерживаются]

- задача языка – разрешить описывать такие модели, которые могут быть верифицированы

[конечное число состояний]

Процесс верификации



Ключевые моменты

- У моделей – конечное число состояний (потенциально бесконечные элементы моделей в Promela ограничены)
 - гарантирует разрешимость верификации,
 - тем не менее, у модели может быть бесконечное число вычислений;
- асинхронное выполнение процессов
 - нет глобальных часов,
 - по умолчанию синхронизация разных процессов отсутствует;

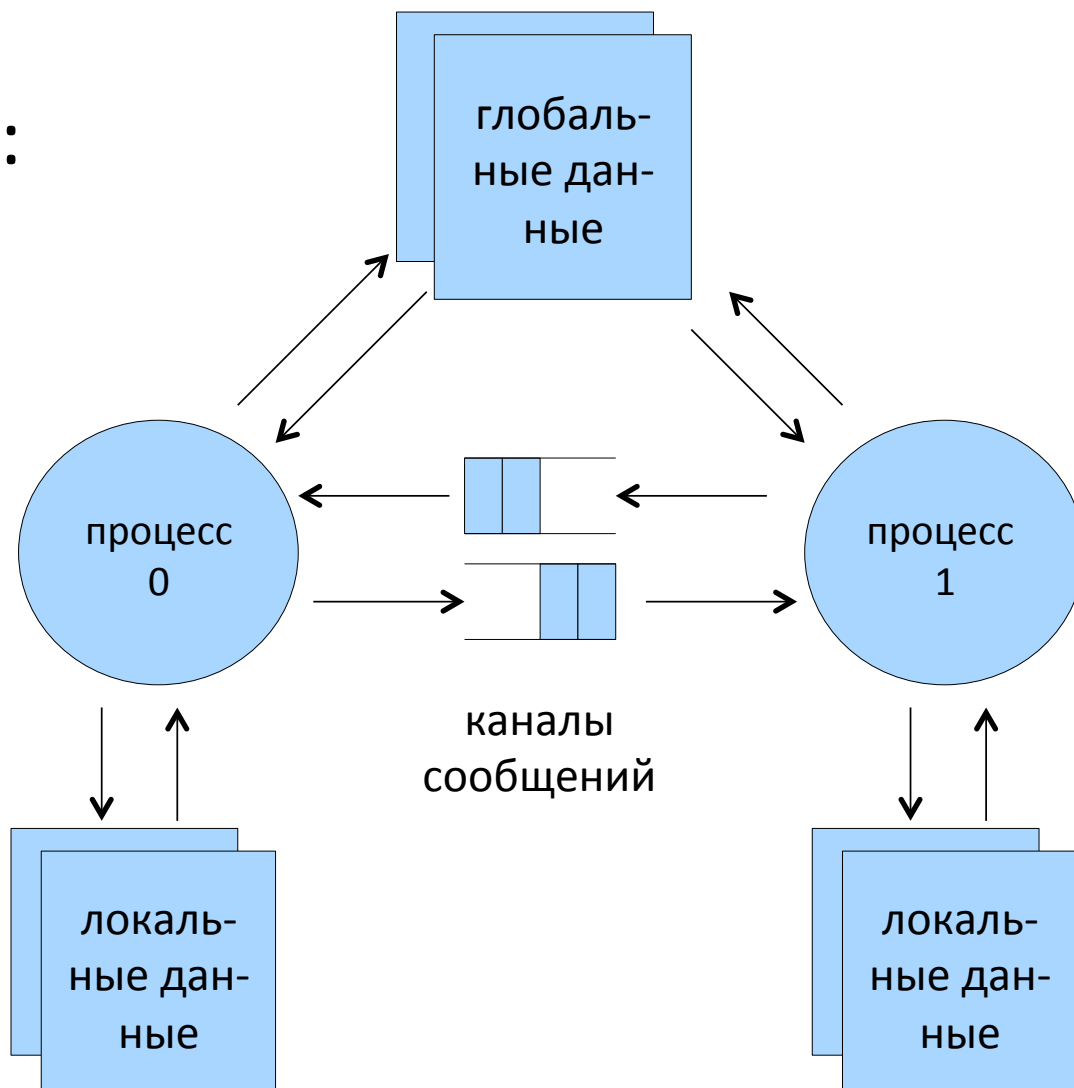
Ключевые моменты

- Недетерминированный поток управления
 - абстракция от деталей реализации;
- Понятие выполнимости оператора
 - с любым оператором связаны понятия предусловия и эффекта,
 - оператор выполняется (производя эффект), только если предусловие истинно, в противном случае он заблокирован,
 - Пример: $q ? m$ – если канал q не пуст, читаем из него сообщение, иначе ждём.

Устройство модели на Promela

Три типа объектов:

- процессы,
- глобальные и локальные объекты данных,
- каналы сообщений.



Hello, world!

немедленное инстанцирование

описываем тип процесса

```
active proctype main()  
{  
    printf("Hello, world!\n")  
}
```

Имя процесса
(**не** ключевое слово)

“;” -- разделитель команд,
а не терминальный символ

Да, это очень
похоже на C

расширение – опционально

```
>spin hello.pml  
      Hello, world!  
1 process created
```

Hello, world!

- Основная структурная единица языка Promela – не функция, а процесс.
- Если процесс один, то можно описать его проще:

```
init {  
    printf("Hello, world!\n")  
}
```

〔 не выделяем определение
типа процесса 〕

- Полная форма – с явным инстанцированием:

```
init {  
    run main()  
}
```

〔 создаём процесс в ходе
выполнения модели 〕

Процессы в Promela

- Поведение процесса задаётся в объявлении типа процесса (**proctype**),
- Экземпляр процесса – инстанцирование **proctype**,
- Два вида инстанцирования процессов:
 - в начальном состоянии системы,

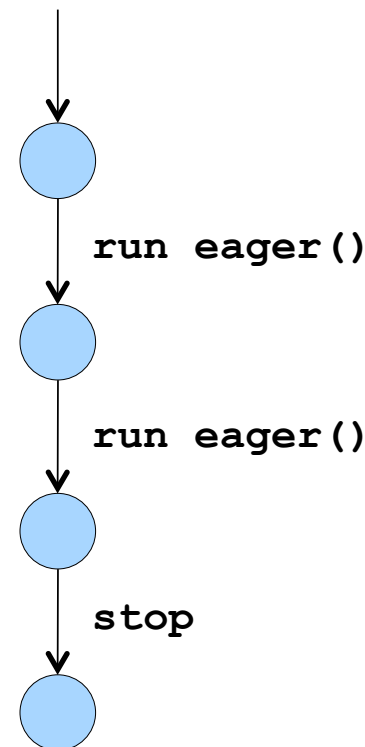
[префикс **active**]
 - в произвольном достижимом состоянии системы.

[оператор **run**]

Процессы в Promela

```
active [2] proctype eager()  
{  
    run eager();  
    run eager();  
}
```

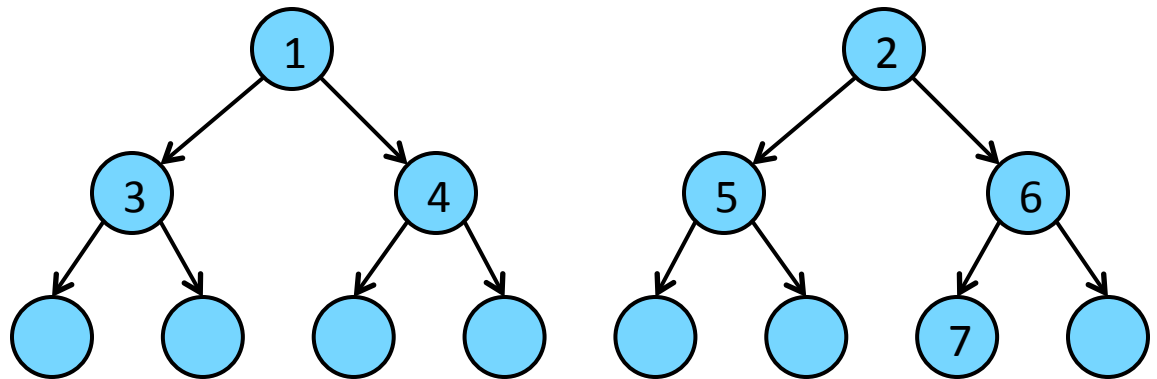
Два процесса инстанцированы
в начальном состоянии
системы



Каждый процесс инстанцирует
две своих копии, а затем
завершает выполнение

Процессы в Promela

```
active [2] proctype eager()  
{  
  run eager();  
  run eager();  
}
```



- Почему такая модель остаётся конечной?
 - run – оператор-выражение,
 - run либо возвращает pid нового процесса, либо 0, если инстанцирование не удалось,
 - выражение выполняется, только если возвращает не 0,
 - максимальное число процессов ограничено 255.

Оператор run

```
proctype irun(byte x)
{
    printf("it is me %d, %d\n",x,_pid)
}

init{
    pid a,b;
    a = run irun(1);
    b = run irun(2);
    printf("I created %d and %d", a, b)
}
```

предопределённая переменная `_pid`

Присваивания и `printf` выполняются всегда. Выражения – только если их значение не равно 0.

```
>spin irun.pml
    it is me 1, 1
    I created 1 and 2
        it is me 2, 2
3 processes created
>
```

Отступ по умолчанию `pid+1` позиция табуляции

1 из 6 возможных чередований

Взаимодействие процессов

- Два способа синхронизации процессов:
 - глобальные (разделяемые) переменные,
 - обмен сообщениями (буферизованные или синхронные каналы),
 - глобальных часов нет;
- У каждого процесса есть локальное состояние:
 - «счётчик команд» (состояние потока управления),
 - значения локальных переменных;
- У модели в целом – глобальное состояние:
 - значение глобальных переменных,
 - содержимое каналов сообщений,
 - множество активных процессов.

Почему число состояний модели конечно?

- Число активных процессов конечно,
- У каждого процесса – ограниченное число операторов,
- Диапазоны типов данных ограничены,
- Размер всех каналов сообщений ограничен.

Явная синхронизация процессов

```
bool toggle = true;

short cnt;

active proctype A() provided (toggle == true)
{
L:      cnt++;

        printf("A: cnt=%d\n", cnt);

        toggle = false;

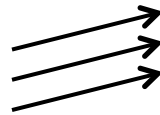
        goto L
}

active proctype B() provided (toggle == false)
{
L:      cnt--;

        printf("B: cnt=%d\n", cnt);

        toggle = true;

        goto L
}
```



```
./spin provided.pml | more
      A: cnt=1
      B: cnt=0
      A: cnt=1
      B: cnt=0
...
```

Процесс выполняется,
только если значение
provided clause равно
true.

По умолчанию
значение равно **true**.

Основные операторы Promela

- Задают элементарные преобразования состояний,
- Размечают дуги в системе переходов соответствующего процесса,
- Их немного – всего 6 типов,
- Оператор может быть:
 - выполнимым: **может** быть выполнен,
 - заблокированным: (пока что) **не может** быть выполнен.

выполнимость может зависеть
от глобального состояния

Основные операторы Promela

- 3 типа операторов уже встречались:

- оператор печати (**printf**),

всегда безусловно выполним, на состояние не влияет

- оператор присваивания,

всегда безусловно выполним, меняет значение только одной переменной, расположенной слева от «=»

- оператор-выражение.

выполним, только если выражение не равно 0 (истинно)

2 < 3 – выполним всегда,

x < 27 – выполним, только если значение $x < 27$,

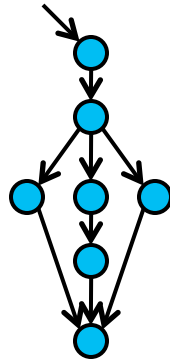
3 + x – выполним, только если $x \neq -3$.

Чередование операторов (интерливинг)

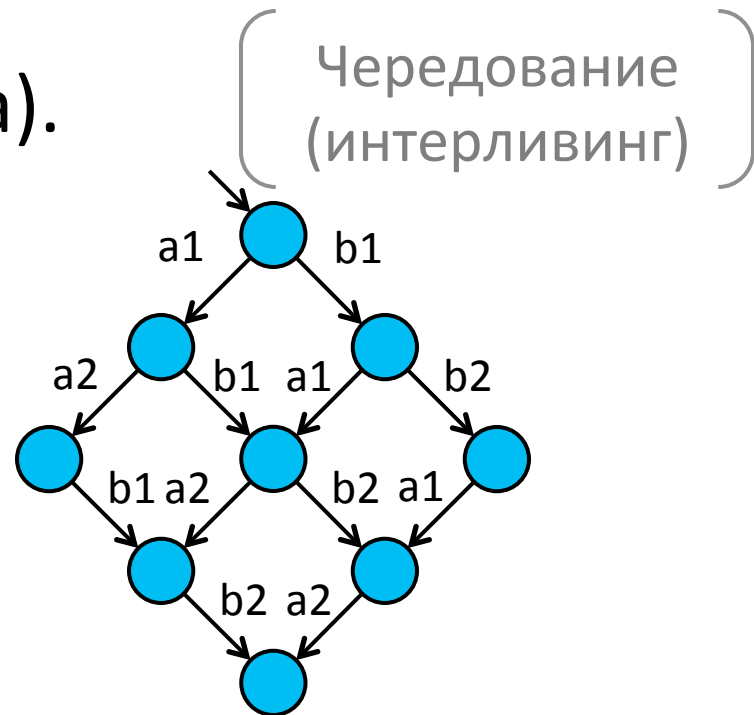
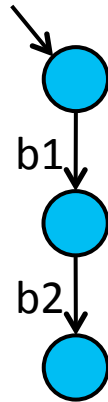
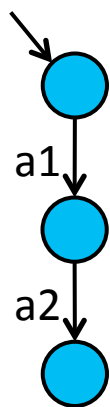
- процессы выполняются параллельно и асинхронно,
 (между выполнением двух последовательных операторов
 одного процесса может быть сколь угодно длинная пауза)
- произвольная диспетчеризация процессов,
- выполнение операторов разных процессов происходит в произвольном порядке,
 (основные операторы выполняются атомарно)
- в теле одного процесса допускается недетерминированное ветвление.

Два уровня недетерминизма

- **Внутренний** (выбор действия в процессе),



- **Внешний** (выбор процесса).



Выполнимость операторов

- Основной инструмент управления выполнимостью операторов в Promela – выражения (expressions)

C

```
while (x <= y)
    /* wait */;
y++;
```

Promela

```
(x > y) -> y = y + 1;
```

Псевдо-операторы

- Несколько специфических операторов:
 - **skip**: всегда выполним, без эффекта, эквивалент выражения **(1)**,
 - **true**: всегда выполним, без эффекта, эквивалент выражения **(1)**,
 - **run**: **0**, если при создании процесса превышен лимит, **_pid** в противном случае.

И ещё один тип оператора

- `assert` (выражение)
 - всегда выполним, не влияет на состояние системы,
 - SPIN сообщает об ошибке, если значение выражения равно 0 (`false`),
 - используется для проверки свойств *безопасности* (состояний).

```
int n;  
  
active proctype invariant() {  
    assert(n <= 3)  
}
```

В силу асинхронности выполнения процессов, данный оператор может быть выполнен в любой момент.

Пример

```
int x;  
proctype A()  
{  
    int y = 1;  
    skip;  
    run B();  
    x = 2;  
    (x > 2 && y == 1);  
    printf("x %d, y %d\n", x, y)  
}
```

значение по умолчанию – 0

выполнимо, если получится
создать процесс B

будет выполнено, только
если **другой** процесс
изменит значение x

Оператор run

- Все выражения без run в Promela не приводят к побочному эффекту;
- в отличие от C, в выражении Promela нельзя изменить значение переменной;
- в выражении может быть только один оператор run:
 - **run B() && run A()** – может быть заблокирован с частичным эффектом,
 - **! (run B())** – эквивалент $(_nr_pr \geq 255)$,
 - **run B() && (a > b)** – создаёт процессы до тех пор, пока $(a \leq b)$;
- возврат 0 – как правило, ошибка при разработке модели.

Пример – два процесса

```
mtype = { P, C };  
mtype turn = P;  
active proctype producer()  
{  
  do  
    :: (turn == P) ->  
      printf("Produce\n");  
      turn = C  
  od  
}  
active proctype consumer()  
{  
  do  
    :: (turn == C) ->  
      printf("Consume\n");  
      turn = P  
  od  
}
```

перечислимый тип (нумерация с 1)

глобальная переменная

по умолчанию – 0, поэтому инициализируем

бесконечный цикл

последовательность вариантов
(option sequence)

индикатор начала последовательности

страж (guard)

Пример – два процесса

```
mtype = { P, C };  
mtype turn = P;  
active proctype producer()  
{  
    do  
        :: (turn == P) ->  
            printf("Produce\n");  
            turn = C  
    od  
}  
active proctype consumer()  
{  
    do  
        :: (turn == C) ->  
            printf("Consume\n");  
            turn = P  
    od  
}
```

-uN – ограничение количества шагов

```
./spin -u14 pc.pml  
Produce  
Consume  
Produce  
Consume  
-----  
depth-limit (-u14 steps) reached  
#processes: 2  
turn = C  
14: proc 1 (consumer) line 17  
"pc.pml" (state 3)  
14: proc 0 (producer) line 6  
"pc.pml" (state 4)  
2 processes created
```


if

```
active proctype consumer()
{
    do
        :: (turn == C) ->
            printf("Consume\n");
            turn = P
    od
}
```

```
active proctype consumer()
{
    again: if
        :: (turn == C) ->
            printf("Consume\n");
            turn = P
        fi;
        goto again
}
```

- break – выход из тела do,
- :: else – если ни одна из альтернатив не выполняется,
- если выполняется несколько альтернатив – внутренний недетерминизм

```
wait: if
    :: (turn == P) -> ...
    :: else -> goto wait
fi
```



```
(turn == P) ->...
```

ИЛИ

```
(turn == P);...
```

Пример: взаимное исключение

```
bool busy;
byte mutex;
proctype P(bit i)
{ (!busy) -> busy = true;
  mutex++;
  printf(P%d in critical section\n", i);
  mutex--;
  busy = false
}
active proctype invariant()
{ assert(mutex <= 1)
}
init {
  atomic { run P(0); run P(1) }
}
```

Показывает, что критическая секция занята

Количество процессов в критической секции

Ждём, пока критическая секция не освободится, и занимаем её

потенциальная **гонка**:
оба процесса могут вычислить (!busy) до выполнения оператора busy = true

Цикл не нужен

Атомарный запуск двух экземпляров P

Запуск верификатора

```
> ./spin -a mutex.pml
> gcc -DSAFETY -o pan pan.c
> ./pan
pan: assertion violated (mutex<=1) (at depth 10)
pan: wrote mutex.pml.trail
(Spin Version 5.1.4 -- 27 January 2008)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim                - (none specified)
        assertion violations        +
        cycle checks                - (disabled by -DSAFETY)
        invalid end states          +
State-vector 24 byte, depth reached 19, errors: 1
        73 states, stored
        32 states, matched
        105 transitions (= stored+matched)
        1 atomic steps
hash conflicts:                0 (resolved)
        2.501                memory usage (Mbyte)
pan: elapsed time 0 seconds
```

Моделирование контрпримера

```
./spin -t -p mutex.pml
Starting invariant with pid 0
Starting :init: with pid 1
Starting P with pid 2
  1:   proc 1 (:init:) line 14 "mutex.pml" (state 1) [(run P(0))]
Starting P with pid 3
  2:   proc 1 (:init:) line 14 "mutex.pml" (state 2) [(run P(1))]
  3:   proc 3 (P) line 4 "mutex.pml" (state 1) [(!(busy))]
  4:   proc 2 (P) line 4 "mutex.pml" (state 1) [(!(busy))]
  5:   proc 3 (P) line 4 "mutex.pml" (state 2) [busy = 1]
  6:   proc 3 (P) line 5 "mutex.pml" (state 3) [mutex = (mutex+1)]
      P1 in critical section
  7:   proc 3 (P) line 6 "mutex.pml" (state 4) [printf('P%d in critical section\\n',i)]
  8:   proc 2 (P) line 4 "mutex.pml" (state 2) [busy = 1]
  9:   proc 2 (P) line 5 "mutex.pml" (state 3) [mutex = (mutex+1)]
      P0 in critical section
 10:   proc 2 (P) line 6 "mutex.pml" (state 4) [printf('P%d in critical section\\n',i)]
spin: line 11 "mutex.pml", Error: assertion violated
spin: text of failed assertion: assert((mutex<=1))
 11:   proc 0 (invariant) line 11 "mutex.pml" (state 1) [assert((mutex<=1))]
spin: trail ends after 11 steps
#processes: 4
      busy = 1
      mutex = 2
 11:   proc 3 (P) line 7 "mutex.pml" (state 5)
 11:   proc 2 (P) line 7 "mutex.pml" (state 5)
 11:   proc 1 (:init:) line 15 "mutex.pml" (state 4) <valid end state>
 11:   proc 0 (invariant) line 12 "mutex.pml" (state 2) <valid end state>
4 processes created
```

Взаимное исключение (другой вариант)

```
bit x,y;  
byte mutex;
```

Сигнал о входе/выходе из критической
секции

```
active proctype A()  
{
```

Число процессов в критической секции

```
    x = 1;
```

```
    (y == 0) ->
```

```
    mutex++;
```

```
    printf("%d\n", _pid);
```

```
    mutex--;
```

```
    x = 0;
```

```
}
```

```
active proctype invariant()  
{ assert(mutex != 2)  
}
```

```
active proctype B()  
{
```

```
    y = 1;
```

```
    (x == 0) ->
```

```
    mutex++;
```

```
    printf("%d\n", _pid);
```

```
    mutex--;
```

```
    y = 0;
```

```
}
```

!(y == 0) означает, что
B – в критической секции

Верификация

```
> ./spin -a mutex2.pml
> gcc -DSAFETY -o pan pan.c
> ./pan
pan: invalid end state (at depth 3)
pan: wrote mutex2.pml.trail

(Spin Version 5.1.4 -- 27 January 2008)
Warning: Search not completed
        + Partial Order Reduction

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

State-vector 20 byte, depth reached 16, errors: 1
        23 states, stored
        3 states, matched
        26 transitions (= stored+matched)
        0 atomic steps
hash conflicts:                0 (resolved)
        2.501          memory usage (Mbyte)
```

Взаимное исключение (другой вариант)

```
bit x,y;  
byte mutex;
```

```
active proctype A()  
{
```

```
  x = 1;
```

```
  (y == 0) ->
```

```
  mutex++;
```

```
  printf("%d\n", _pid);
```

```
  mutex--;
```

```
  x = 0;
```

```
}
```

```
active proctype invariant()  
{ assert(mutex != 2)
```

```
}
```

```
active proctype B()  
{
```

```
  y = 1;
```

```
  (x == 0) ->
```

```
  mutex++;
```

```
  printf("%d\n", _pid);
```

```
  mutex--;
```

```
  y = 0
```

```
}
```

гонка

если эти операторы будут выполнены
последовательно, процессы заблокируют
друг друга (deadlock, invalid endstate)

Моделирование контрпримера

```
> ./spin -t -p mutex2.pml
Starting A with pid 0
Starting B with pid 1
Starting invariant with pid 2
  1:   proc 2 (invariant) line 25 "mutex2.pml" (state 1)      [assert((mutex!
=2))]
  2:   proc 2 terminates
  3:   proc 1 (B) line 16 "mutex2.pml" (state 1)             [y = 1]
  4:   proc 0 (A) line 6 "mutex2.pml" (state 1)               [x = 1]
spin: trail ends after 4 steps
#processes: 2
      x = 1
      y = 1
      mutex = 0
  4:   proc 1 (B) line 17 "mutex2.pml" (state 2)
  4:   proc 0 (A) line 7 "mutex2.pml" (state 2)
3 processes created
>
```


Алгоритм Петерсона (1981)

```
mtype = {A_Turn, B_Turn};  
bool x,y;  
byte mutex;  
mtype turn = A_Turn;  
  
active proctype A()  
{ x = true;  
  turn = B_Turn;  
  (!y || turn == A_Turn) ->  
  mutex++;  
  /*critical section*/  
  mutex--;  
  x = false;  
}  
  
active proctype invariant()  
{ assert(mutex <= 1)  
}
```

Сигнал о входе/выходе из критической секции

Число процессов в критической секции

Чей ход?

```
active proctype B()  
{ y = true;  
  turn = A_Turn;  
  (!x || turn == B_Turn) ->  
  mutex++;  
  /*critical section*/  
  mutex--;  
  y = false;  
}
```

Вариант алгоритма Лампорта (1981)

```
byte turn[2];
```

```
byte mutex;
```

```
active [2] proctype P()
```

```
{ bit i = _pid;
```

```
L:
```

```
    turn[i] = 1;
```

```
    turn[i] = turn[i+1];
```

```
    (turn[1-i] == 0) ||  
(turn[i] < turn[1-i]) ->
```

```
    mutex++;
```

```
    assert(mutex == 1);
```

```
    mutex--;
```

```
    turn[i] = 0;
```

```
    goto L;
```

```
}
```

Чей ход?

Число процессов в критической секции

Может ли алгоритм достичь
максимального $i = 255$?

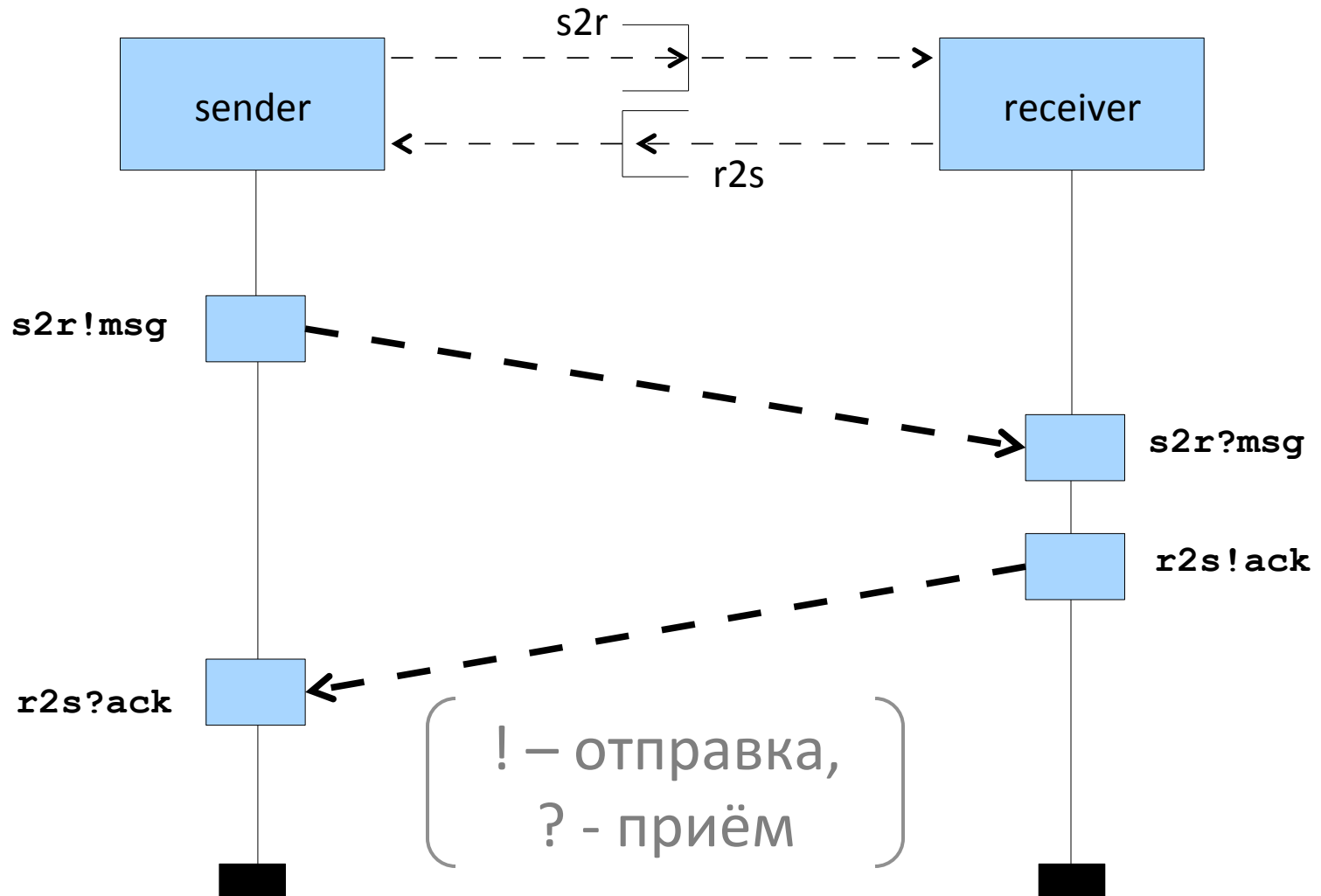
Будет ли алгоритм корректен, если при
достижении 255 обнулять i ?

Основные операторы языка Promela

- присваивание: `x++, x--, x = x + 1, x = run P();`
- выражение: `(x), (1), run P(), skip, true, else, timeout;`
- печать: `printf("x = %d\n", x);`
- ассерт: `assert(1+1 == 2), assert(false);`
- отправка сообщения: `q!m;`
- приём сообщения: `q?m;`

Последние два типа операторов Promela

отправка и приём сообщений

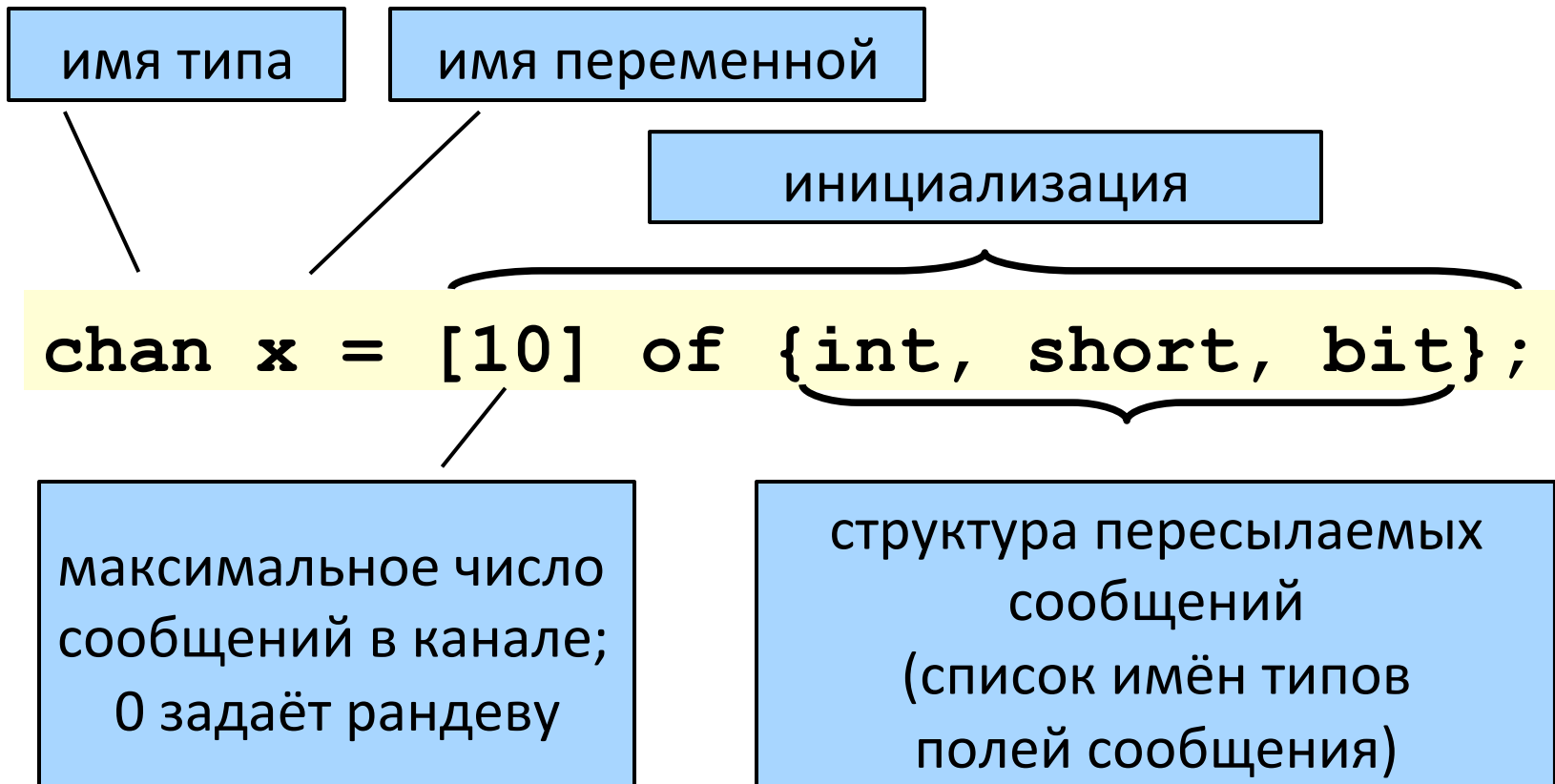


Каналы сообщений

- Сообщения передаются через **каналы** (очереди/буфера ограниченного объёма),
- каналы бывают двух типов:
 - буферизованные (асинхронные),
 - небуферизованные (синхронные, рандеву);

Каналы сообщений

- пример объявления канала:



Каналы сообщений

- пример объявления канала:

неинициализированная
канальная переменная **a**

канал с типа «рандеву»

```
chan a;  
chan c = [0] of {bit};  
chan toR = [2] of {mtype, bit, chan};  
chan line[2] = [1] of {mtype, record};
```

массив из
двух каналов

пользовательский тип

каналы можно
передавать по
каналам

Объявление mtype

(mtype = *message type*)

- способ определить СИМВОЛЬНЫЕ КОНСТАНТЫ (до 255),

- объявление mtype:

в итоге объявляется 6 констант

```
mtype = {foo, bar};  
mtype = {ack, msg, err, interrupt};
```

- объявление переменных типа mtype:

```
mtype a;  
mtype b = foo;
```

неинициализированная, значение 0

значение всегда отлично от 0

Отправка и приём сообщений

- отправка: **ch!expr₁, . . . , expr_n**
 - значения expr_i должны соответствовать типам в объявлении канала;
 - **выполнима**, если заданный канал **не полон**;
- приём: **ch?const₁ или var₁, ..., const_n или var_n**
 - значения var_i становятся равны соотв. значениям полей сообщения;
 - значения const_i ограничивают допустимые значения полей;
 - **выполним**, если заданный канал **не пуст** и первое сообщение в канале соответствует всем константным значениям в операторе приёма сообщения.

Отправка и приём сообщений

- пример:

```
#define ack 5
```

```
chan ch = [N] of {int, bit};
```

```
bit seqno;
```

```
ch!ack, 0;
```

```
ch?ack, seqno
```

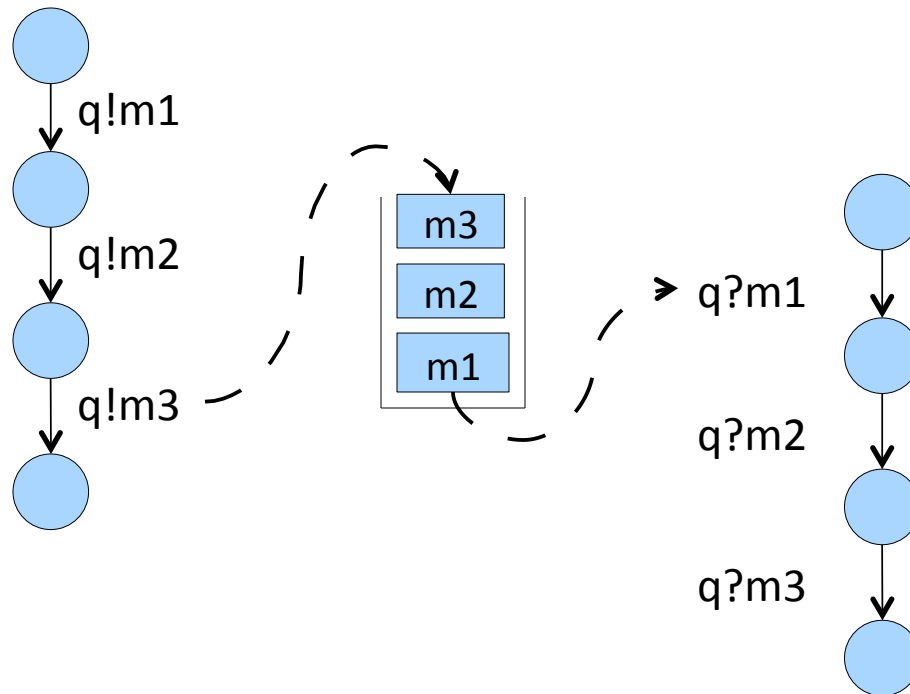
другой вариант:

```
ch!ack(0);
```

```
ch?ack(seqno)
```

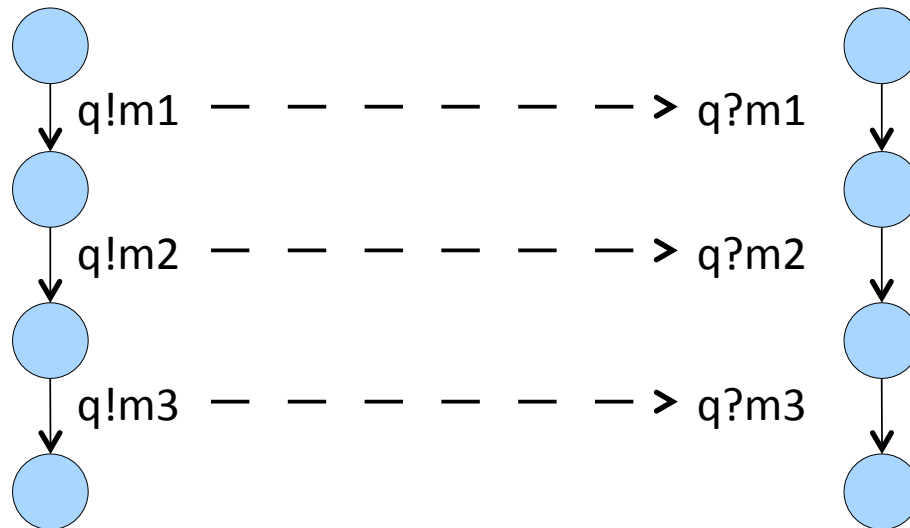
Асинхронная передача сообщений

- асинхронные сообщения буферизуются для последующего приёма, пока канал не полон,
- отправитель блокируется, когда канал полон,
- получатель блокируется, когда канал пуст.



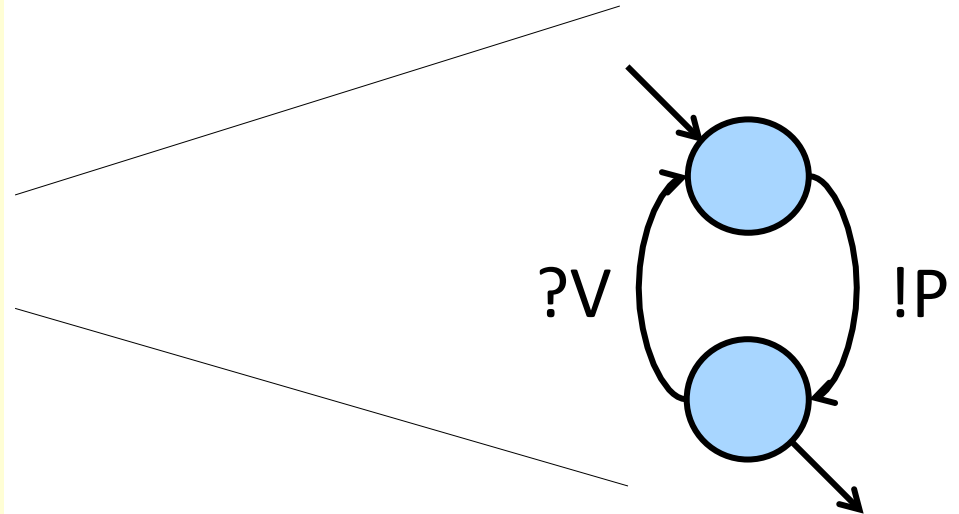
Синхронная передача сообщений

- ёмкость канала равна 0:
`chan ch = [0] of {mtype};`
- передача сообщений методом «рандеву»;
- не хранит сообщения;
- отправитель блокируется в ожидании получателя, и наоборот;
- отправка и приём выполняются *атомарно*.



Пример: моделируем семафор

```
mtype = { P, V };  
chan sema = [0] of { mtype };  
active proctype semaphore()  
{  
L:   sema!P -> sema?V; goto L  
}  
active [5] proctype user()  
{  
L:   /*non-critical*/  
      sema?P ->  
      /*critical*/  
      sema!V;  
      goto L;  
}
```



Другие операции с каналами

- `len(q)` – возвращает число сообщений в канале,
- `empty(q)` – возвращает `true`, если `q` пуст,
- `full(q)` – возвращает `true`, если `q` полон,
- `nempty(q)` – вместо `!empty(q)` (в целях оптимизации),
- `nfull(q)` – вместо `!full(q)` (в целях оптимизации).

Операции со скобками

- $q?[n, m, p]$ (есть ли в канале подходящее сообщение)
 - булево выражение без побочных эффектов,
 - равно **true**, только когда $q?m, n, p$ выполнимо, однако не влияет на значения n, m, p и не меняет содержимое канала q ;
- $q?<n, m, p>$ (широковещательный канал)
 - выполнимо тогда же, когда и $q?n, m, p$; влияет на значения n, m, p так же, как и $q?n, m, p$, однако не меняет содержимое q ;
- $q?n(m, p)$ (отделяем тип сообщения от параметров)
 - вариант записи оператора приёма сообщения (т.е. $q?n, m, p$),
 - может использоваться для отделения переменной от констант.

Область видимости объявления канала

- имя канала может быть локальным или глобальным, но канал сам по себе – всегда глобальный объект.

```
chan x = [3] of { chan };
```

глобальная переменная, видна A и B

```
active proctype A()
```

неинициализированный локальный канал

```
{ chan a;
```

```
  x?a;
```

получаем идентификатор канала от процесса B

```
  a!x
```

используем его

```
}
```

```
active proctype B()
```

инициализированный локальный канал

```
{ chan b = [2] of { chan };
```

```
  x!b;
```

отправляем процессу A идентификатор канала

```
  b?x;
```

значение x не изменилось

```
  0
```

если B умрёт, канал b исчезнет!

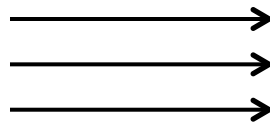
```
}
```


Особые случаи:

упорядоченная отправка, случайный приём

- **$q!!n, m, p$** – аналогично $q!n, m, p$, но сообщение n, m, p помещается в канал сразу за первым сообщением, меньшим n, m, p ;
- **$q??n, m, p$** – аналогично $q?n, m, p$, но из канала может быть выбрано **любое** сообщение (не обязательно *первое*).

```
init
{
    chan q = [3] of {int};
    int x;
    q!!5;
    q!!2;
    q?x->printf("%d\n", x);
    q?x->printf("%d\n", x)
}
```



```
> ./spin sorted.pml
      2
      5
1 process created
>
```

Основные типы данных Promela

Тип	Диапазон	Пример объявления
bit	0..1	<code>bit turn = 1;</code>
bool	false..true	<code>bool flag = true;</code>
byte	0..255	<code>byte cnt;</code>
chan	1..255	<code>chan q;</code>
mttype	1..255	<code>mttype msg;</code>
pid	1..255	<code>pid p;</code>
short	$-2^{15}..2^{15}-1$	<code>short s = 100;</code>
int	$-2^{31}..2^{31}-1$	<code>int x = 1;</code>
unsigned	$0..2^n-1$	<code>unsigned u : 3;</code>

3 бита,
0..7

- по умолчанию все объекты (и локальные, и глобальные) инициализируются нулём;
- все переменные должны быть объявлены до первого использования;
- переменная может быть объявлена где угодно.

Основные типы данных Promela

В Promela нет действительных чисел, чисел с плавающей точкой и указателей. Этот язык предназначен для описания взаимодействия объектов, а не для описания вычислений.

Массивы и пользовательские типы данных

Одномерные массивы:

```
byte a[27];  
bit flags[4] = 1;
```

- все элементы массива инициализируются одним значением,
- индексы нумеруются с 0.

Пользовательские типы данных:

```
typedef record {  
    short f1;  
    byte f2 = 4;  
}  
record rr;  
rr.f1 = 5;
```

ключевое слово

имя пользовательского типа

по умолчанию 0

объявление переменной
нового типа

ссылка на элемент структуры

Ещё один способ объявления массивов

```
typedef array {byte b[4];}  
array a[4];  
  
a[3].b[2] = 1;
```

ИЛИ

```
#define ab(x,y) a[x].b[y]  
  
ab(x,y) = ab(2,3) + ab(3,2);
```

Перед разбором все модели прогоняются через препроцессор C
(поддерживаются **#define**, **#if**, **#ifdef**, **#ifndef**, **#include**)

Вычисление выражений

- Значение всех выражений вычисляется в наиболее широком диапазоне (int);
- В присваиваниях и передаче сообщений значения приводятся к целевому типу **после** вычисления.

```
mtype = {foo, bar};  
active proctype tryme()  
{ byte x;  
  short y = 1024;  
  chan a,b;  
  mtype p;  
  
  a = a + b;  
  x = 257;  
  x = y;  
  p = y/8  
}
```

Ошибка

Предупреждение –
потеря данных

Предупреждение –
потеря данных

Сообщения об ошибке не будет

Область видимости объектов данных

- Только два уровня видимости:
 - глобальный (данные видны всем активным пользователям),
 - локальный (данные видны только одному процессу)
 - подобластей (напр. для блоков) нет,
 - локальная переменная видна везде в теле процесса.

```
active proctype main()
{ int x, y;
  {
    int y, z;
    x++; y++; z++;
  };

  printf("y=%d, z=%d\n", y, z);
}
```

Ошибка, повторное объявление y

Переменная z всё ещё видна!

Поток управления процесса

- 5 способов задать поток управления:
 - последовательная композиция (`" ; "`), метки, **goto**,
 - структуризация (макросы и **inline**),
 - атомарные последовательности (**atomic**, **d_step**),
 - недетерминированный выбор и итерации (**if . . fi**, **do . . od**),
 - escape-последовательности (**{ . . . }unless { . . . }**).

Макросы – препроцессор `cpr`

- Используется для включения файлов и разворачивания макросов,
- Варианты использования:

– КОНСТАНТЫ

```
#define MAXQ 2  
chan q=[MAXQ] of {mtype,chan};
```

(альтернатива: `spin -DMAXQ=2 ...`)

– макросы

```
#define RESET(a) \  
atomic {a[0] = 0; a[1] = 0}
```

– УСЛОВНЫЙ КОД

```
#define LOSSY 1  
...  
#ifdef LOSSY  
active proctype D()  
#endif
```

```
#if 0  
COMMENTS  
#endif
```

Макросы – препроцессор `cpr`

- Минусы:
 - имена макросов не видны в парсере,
 - имена макросов не видны при моделировании,
- Варианты:
 - использовать `mtypre` для определения констант,
 - использовать другой препроцессор (ключ `-P`),
- **`inline`**-определения.

`inline`-определения

- среднее между макросом и процедурой,
- именованный фрагмент кода с параметрами,
- не функция – не возвращает значения.

```
#define swap(a,b) tmp = a;\n                a = b;\n                b = tmp\n\n#endif
```



```
inline swap(a,b) {\n    tmp = a;\n    a = b;\n    b = tmp\n}
```

Недетерминированный выбор

```
if
:: guard1 -> stmt1.1; stmt1.2; stmt1.3;
:: guard2 -> stmt2.1; stmt2.2; stmt2.3;
::
:: guardn -> stmtn.1; stmtn.2; stmtn.3;
fi
```

- оператор if выполним, если выполним хотя бы один из стражей,
- если выполнимо более одного стража, то для выполнения выбирается один из них недетерминированным образом,
- если ни один из стражей не выполним, выполнение оператора if блокируется,
- в качестве стража может быть использован любой оператор.

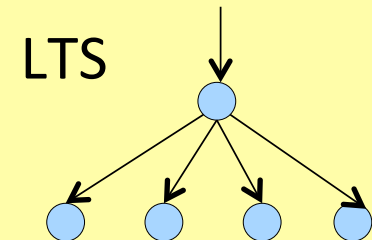
Ещё по *if*

```
/*ищем максимум среди x и y*/  
if  
:: x >= y -> m = x  
:: x <= y -> m = y  
fi
```

```
/*случайный выбор числа 0..3*/  
if  
:: n = 0  
:: n = 1  
:: n = 2  
:: n = 3  
fi
```

```
if  
:: (n%2 != 0) -> n = 0  
:: (n >= 0) -> n = n-2  
:: (n%3 == 0) -> n = 3  
:: else /* -> skip */  
fi
```

выполняется, только если не
выполняется ни один из
стражей



Выражение `else`

C

```
if (x <= y)
    x = y - x;
y++;
```

Promela

```
if
:: (x <= y) -> x = y - x
:: else
fi;
y++
```

- в отличие от C, если `else` отсутствует, то выполнение **блокируется**,
- т.е. все варианты выполнения оператора `if` должны быть явно выписаны.

Специальные выражения и переменные

- **else** – true, если ни один оператор процесса не выполним,
- **timeout** – true, если ни один оператор модели не выполним,
- **_** – переменная, доступная только по записи, значение не сохраняет,
- **_pid** – номер текущего процесса,
- **_nr_pr** – число активных процессов.

Оператор do

```
do
::guard1 -> stmt1.1;stmt1.2;stmt1.3;
::guard2 -> stmt2.1;stmt2.2;stmt2.3;
::
::guardn -> stmtn.1;stmtn.2;stmtn.3;
do
```

- в качестве стража может быть использован любой оператор,
- фактически, это оператор if, выполняемый в цикле
- из цикла можно выйти только при помощи break и goto.

Оператор do

- Ждём, пока не наступит момент (a==b)

```
do
:: (a == b) -> break;
:: else -> skip
od
```

```
L:  if
    :: ( a== b) -> skip
    else goto L
fi
```

(a == b)

- все три фрагмента эквивалентны

Alternating Bit Protocol

(Bartlett и др., 1969)

- Два процесса, отправитель и получатель;
- К каждому сообщению добавляется один *bit*;
- Получатель сообщает о доставке сообщения, возвращая бит отправителю;
- Если отправитель убедился в доставке сообщения, он отправляет новое, изменяя значение бита;
- Если значение бита не изменилось, получатель считает, что идёт повтор сообщения.

Функция eval()

Отображает текущее значение `x` на константу, которая служит ограничением для принимаемых сообщений

```
ch!msg(12)  
ch?msg(eval(x))
```

Сообщение будет принято, если значение переменной `x` равно 12

Модель на Promela

```
mtype = {msg, ack};  
chan s_r = [2] of {mtype, bit};  
chan r_s = [2] of {mtype, bit};
```

```
active proctype sender()
```

```
{ bit seqno;
```

```
  do
```

```
    :: s_r!msg,seqno ->
```

```
      if
```

```
        :: r_s?ack,eval(seqno) ->
```

```
          seqno = 1 - seqno;
```

```
        :: r_s?ack,eval(1-seqno)
```

```
      fi
```

```
  od
```

```
}
```

```
active proctype receiver()
```

```
{ bit expect, seqno;
```

```
  do
```

```
    :: s_r?msg,seqno ->
```

```
      r_s!ack,seqno;
```

```
    if
```

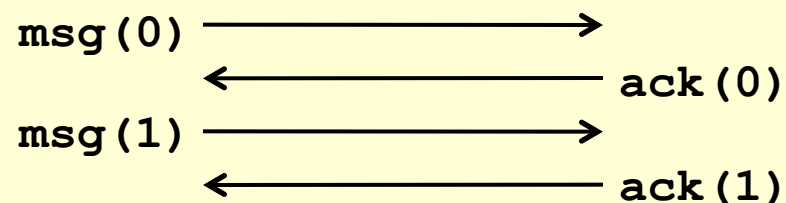
```
      :: seqno == expect;
```

```
        expect = 1 - expect
```

```
    :: else
```

```
    fi
```

```
  od
```



Считываем новое сообщение

Сохраняем сообщение

Игнорируем сообщение

Запускаем моделирование

```
>./spin -u20 -c abp.pml
proc 0 = sender
proc 1 = receiver
q\p    0    1
  1    s_r!msg,0
  1    .    s_r?msg,0
  2    .    r_s!ack,0
  2    r_s?ack,0
  1    s_r!msg,1
  1    .    s_r?msg,1
  2    .    r_s!ack,1
  2    r_s?ack,1
-----
depth-limit (-u20 steps) reached
-----
final state:
-----
#processes: 2
                queue 1 (s_r):
                queue 2 (r_s):
20:    proc 1 (receiver) line 19
"abp.pml" (state 7)
20:    proc 0 (sender) line 7
"abp.pml" (state 7)
2 processes created
```

Моделируем первые 20 шагов

Верификация по умолчанию

```
>./spin -a abp.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 44 byte, depth reached 13, errors: 0
    14 states, stored
    1 states, matched
    15 transitions (= stored+matched)
    0 atomic steps
hash conflicts:                0 (resolved)

    2.501      memory usage (Mbyte)

unreached in proctype sender
    line 15, state 10, "-end-"
    (1 of 10 states)
unreached in proctype receiver
    line 27, state 10, "-end-"
    (1 of 10 states)
```

Чем и как проверяем?

Какие свойства?

Проделанная работа

Используемая память

Обнаружен
недостижимый код
(процессы не
завершаются)

Полезные инструменты

- Просмотр пространства состояний:
 - параметры компиляции ran.c:
 - -DCHECK – выводить порядок обхода пространства состояний
 - -DVERBOSE -DSDUMP – выводить вектора состояний
 - -DBFS – обход в ширину (удобнее для анализа)
 - параметры запуска ran:
 - -d – вывод графов процессов (state – номер оператора)
- Отключение оптимизаций:
 - параметры spin:
 - -o1 – отключение оптимизации потока данных,
 - -o2 – отключение удаления мертвых переменных,
 - -o3 – отключение слияния состояний
 - параметры компиляции ran.c:
 - -DNOREDUCE – отключение редукции частичных порядков

Обратите внимание:

- инициализация переменной (`int x = 1`) не считается действием;
- порождение (`run`) и завершение процесса – действия,
 - при использовании `active` в начальном состоянии процесс уже запущен,
 - не только терминальное состояние, но и терминальное действие `-end-`;
- процессы порождаются в случайном порядке, но завершаются в только порядке, обратном порядку порождения (LIFO);
- проверка стража ветвления – действие.

Вопросы?