


Моделирование и верификация распределенных систем в среде SPIN



Ирина Владимировна Шошмина
ИКНК, СПбПУ

Цель лекции –

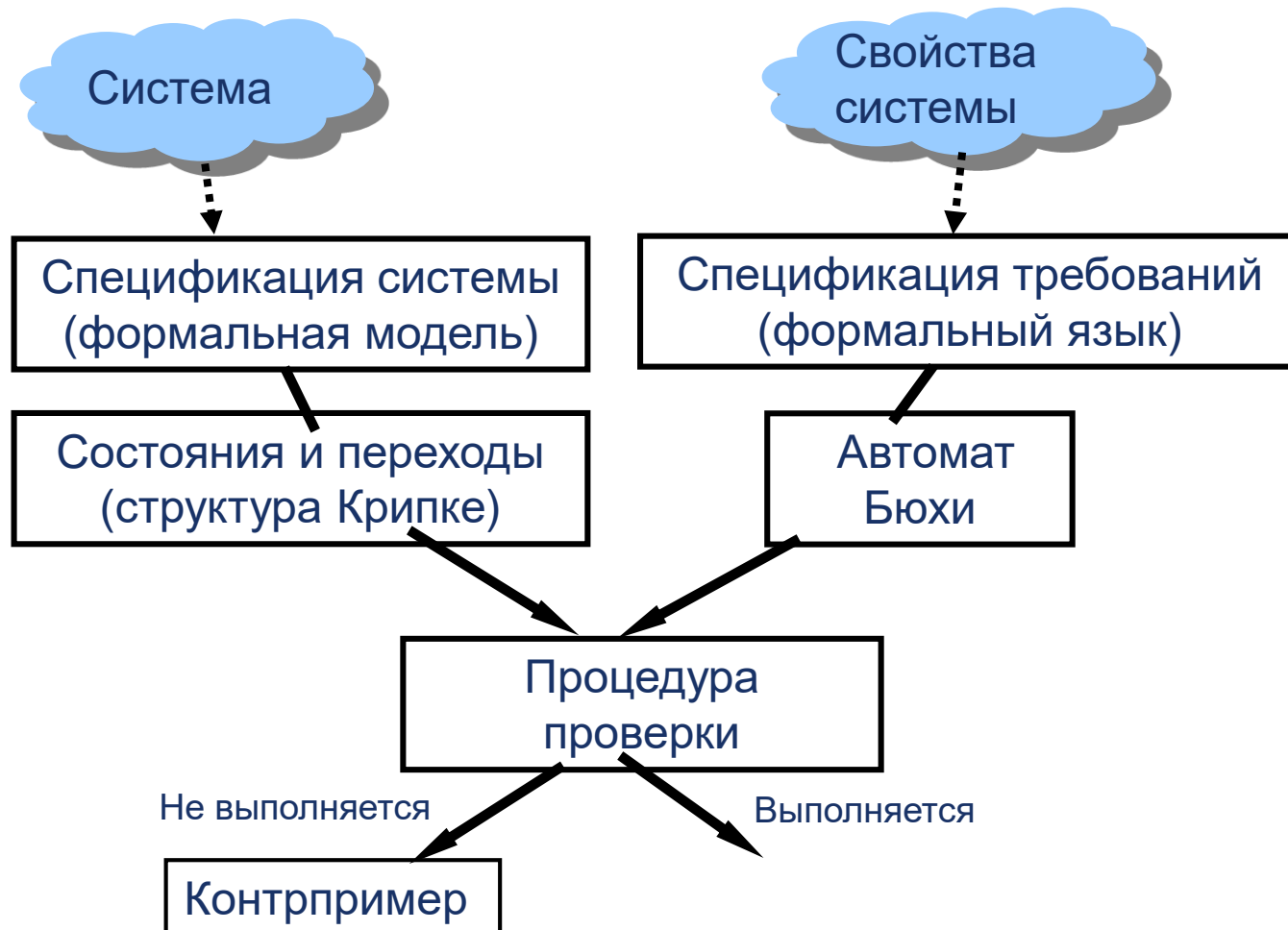
- познакомиться с практическими аспектами моделирования и формальной верификации на примере программного средства SPIN

Задачи лекции:

- освоить основные конструкции входного языка SPIN, Promela
- построить элементарные модели в SPIN
- получить опыт верификации моделей в SPIN

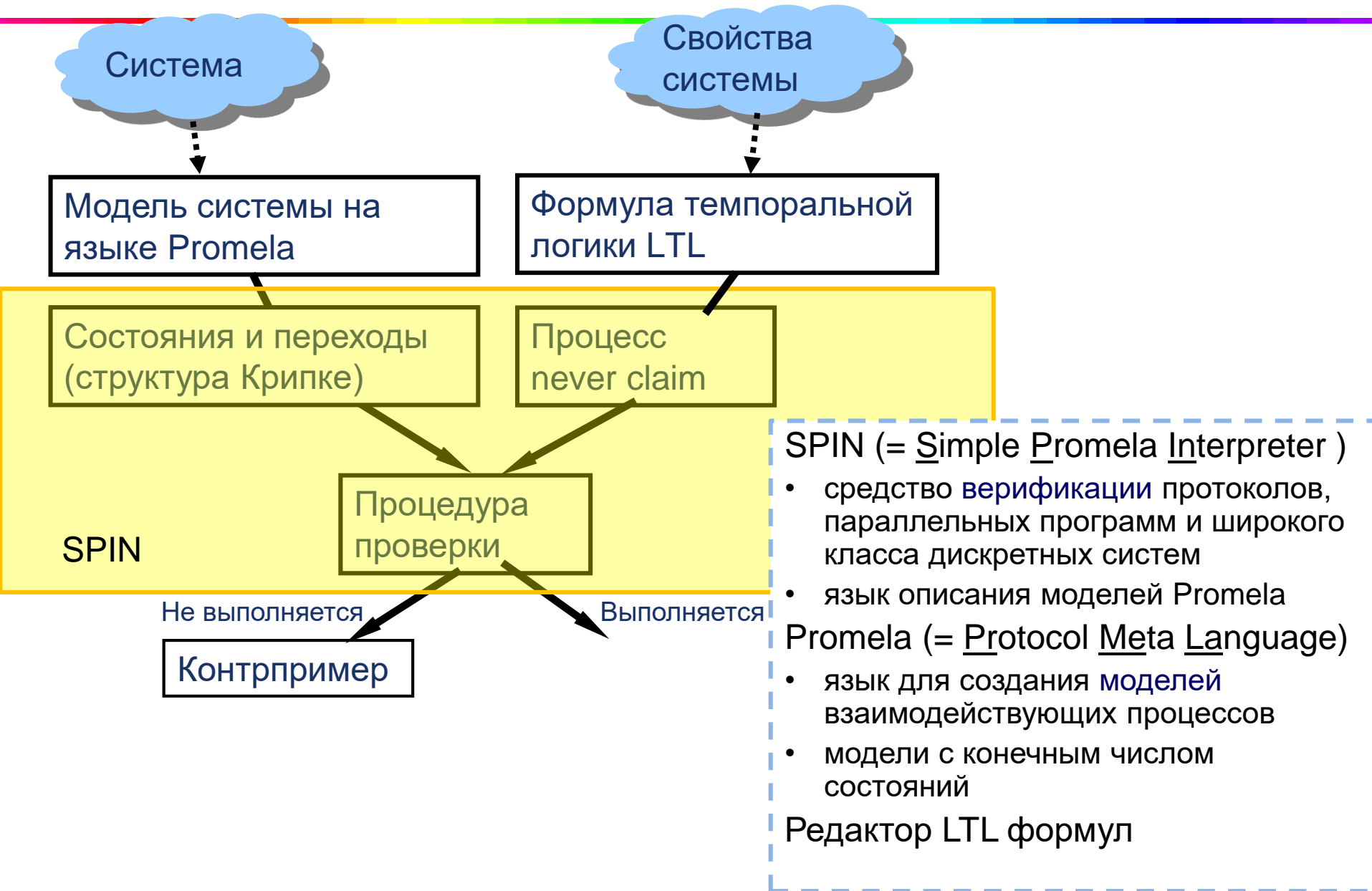
Общая структура метода проверки модели для LTL

теоретико-автоматный подход



*Известный вам
из предыдущих
лекций курса*

Что такое SPIN?

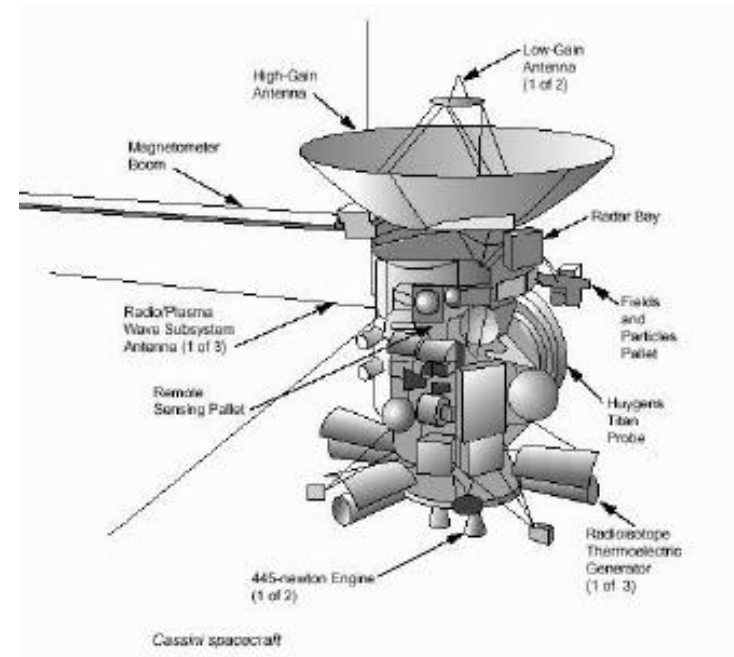


SPIN – широко распространенный пакет верификации

- разрабатывается Bell Labs с 1996
- премия ACM System Award в 2001

Функции:

- строить параллельные модели систем
- выражать требования на языке LTL
- автоматически верифицировать выполнение требования на модели (метод Model Checking)



Использовался при верификации:

- ATC PathStar (Lucent)
- системы управления шлюзами в Роттердаме
- аэрокосмические системы Mars Exploration Rovers (NASA), включая последний марсоход
- и в других проектах

Promela – входной язык SPIN

Promela - язык моделирования взаимодействующих процессов

Каждая программа на Promela – модель, абстракция реальной системы

- включает конструкции для создания процессов и описания межпроцессного взаимодействия
- включает конструкции для верификации моделей

Синтаксис Promela. Пример 1

Последовательный алгоритм на
псевдо-коде

```
int x=0; /*Глобальная переменная*/
```

```
repeat 10 times {  
    x = x+1;  
}
```

Алгоритм на Promela?

Синтаксис Promela

Алгоритм на псевдо-коде

```
int x=0; /*Глобальная переменная*/  
  
repeat 10 times {  
    x = x+1;  
}
```

- **Процесс** – основная структурная единица языка Promela
 - Процессы запускаются операторами запуска
 - В процессе операторы выполняются последовательно

Алгоритм на Promela

```
#define N 10  
int x = 0;
```

объявление констант и
глобальных
переменных

```
init{
```

объявление процесса

тело процесса

```
}
```

- В программе на Promela должен быть **хотя бы один** процесс
 - в Promela нет функций
- Процесс **init** - **особый процесс в Promela**
 - Он не требует отдельного оператора запуска
 - Процесса init может и не быть в программе на Promela
 - **init** – зарезервированное слово языка Promela

Оператор цикла

- Синтаксис Promela похож на синтаксис языка C

Алгоритм на псевдо-коде

```
int x=0; /*Глобальная переменная*/  
  
repeat 10 times {  
    x = x+1;  
}
```

```
do  
:: условие -> список команд  
:: условие -> список команд  
...  
:: условие -> список команд  
od
```

условие -> список команд

Защищенный список команд (защищен условием). Синтаксис предложен Дейкстрой

Алгоритм на Promela

```
#define N 10  
int x = 0;  
  
init{  
    int i = 0;  
    do  
    :: i < N ->  
        x = x + 1;  
        i ++;  
    :: else -> break  
    od  
}
```

ЦИКЛ

else – иначе,
break – выход из цикла,
->, ; - разделение операторов

Пример 2. Модель с 2-мя процессами

Promela позволяет легко моделировать многопроцессные алгоритмы
Изменим предыдущий алгоритм, запустим его для двух процессов

```
int x=0;
```

```
P1::
```

```
int t = 0, i = 0;
```

```
repeat 10 times {
```

```
    t = x;
```

```
    x = t+1;
```

```
    i = i+1
```

```
}
```

```
P2::
```

```
int t = 0, i = 0;
```

```
repeat 10 times {
```

```
    t = x;
```

```
    x = t+1;
```

```
    i = i+1
```

```
}
```

P1, P2 - процессы

x - разделяемая (глобальная) переменная

i, t - локальные переменные

Код процессов P1 и P2 совпадает

Пример 2. Модель с 2-мя процессами

```
#define N 10  
int x = 0;
```

`proctype` <имя процесса> -
объявление процесса

```
active[2] proctype P() {  
  int t = 0, i = 0;  
  do  
    :: i < N ->  
      t = x; printf("MSC: t=%d", t);  
      x = t + 1; printf("MSC: x=%d", x);  
      i ++;  
    :: else -> break  
  od  
}
```

`active` - непосредственный
запуск процесса

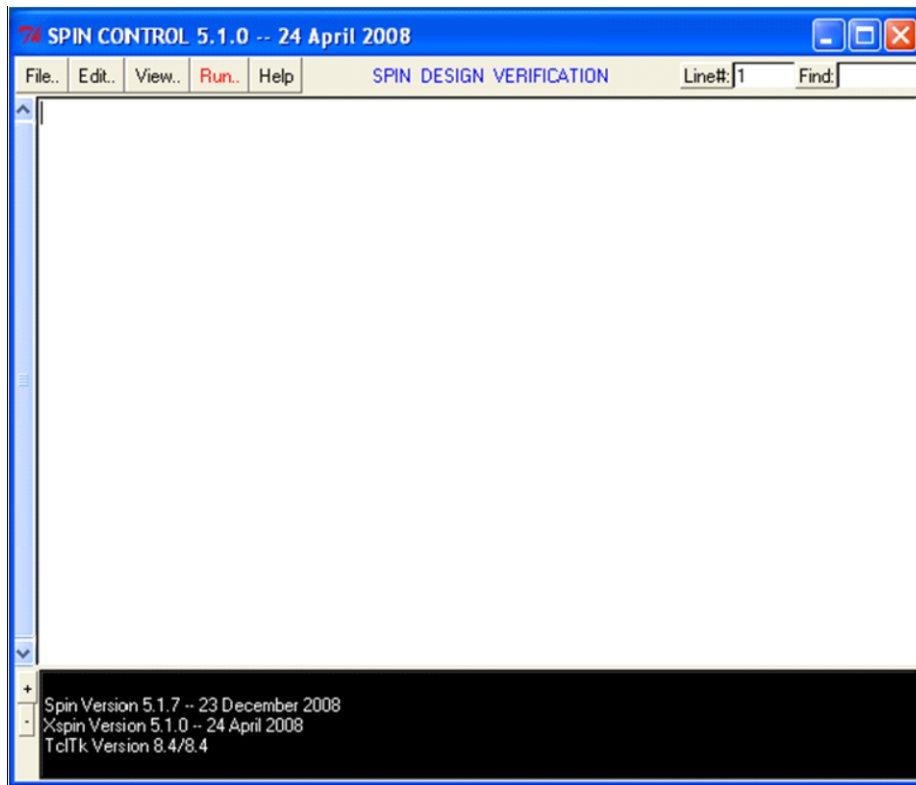
`active[2]` - запустить 2
копии процесса

В Promela существует несколько способов запуска процессов:

- непосредственный запуск из объявления с помощью служебного слова `active`,
- запуск оператором `run`,
- запуск основного (особого) процесса `init`

Запуск на выполнение моделей в Spin

Xspin - графическая оболочка Spin



Существует несколько возможностей запустить SPIN:

- С помощью командной строки,
- С помощью графической оболочки Xspin
- С помощью графической оболочки iSpin, и другие

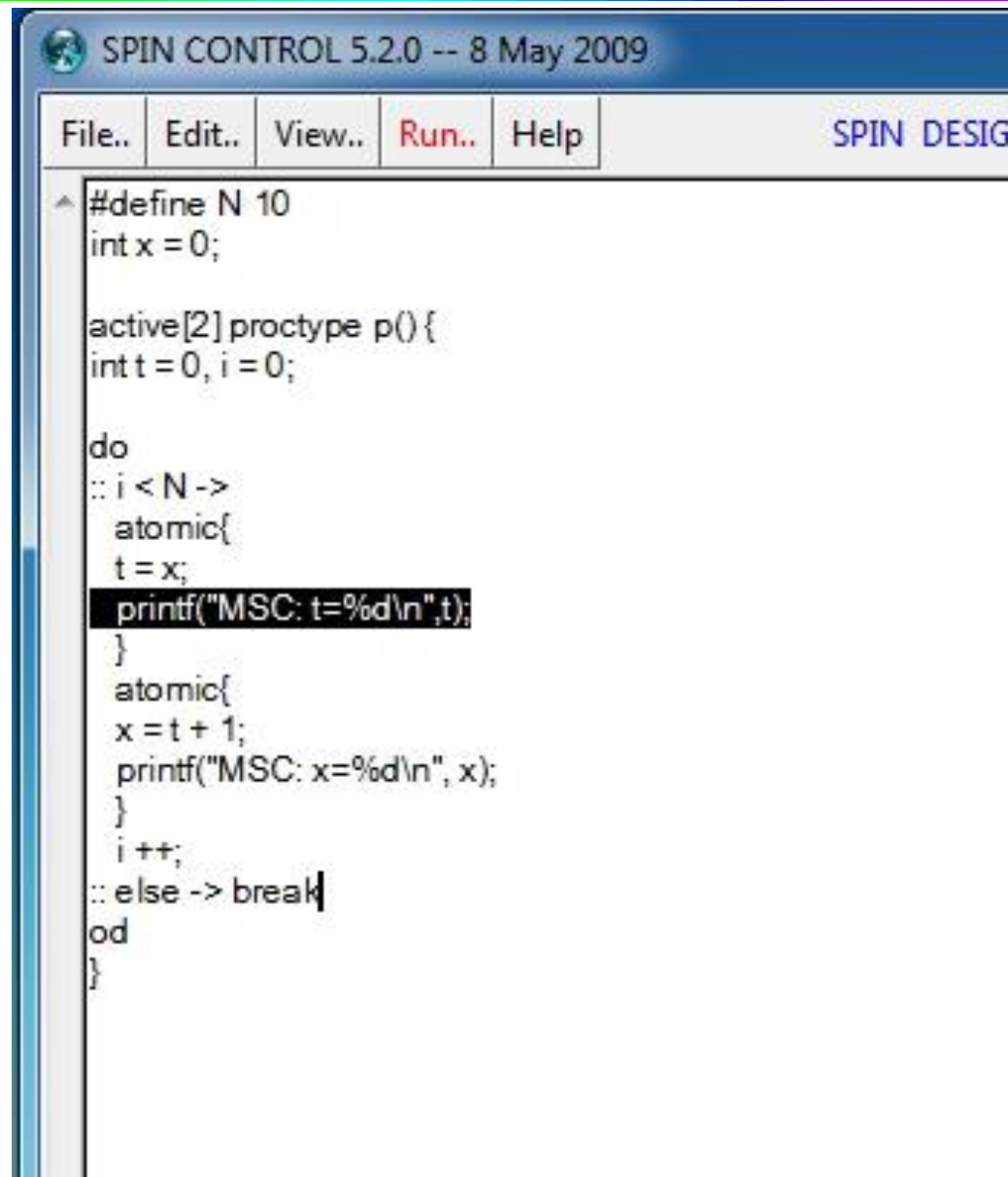
Основное окно редактора XSpin

```
$ /bin/spin/xspin.tcl
```

запуск оболочки XSpin из Cygwin

Окно редактора в XSPIN

Код на Promela



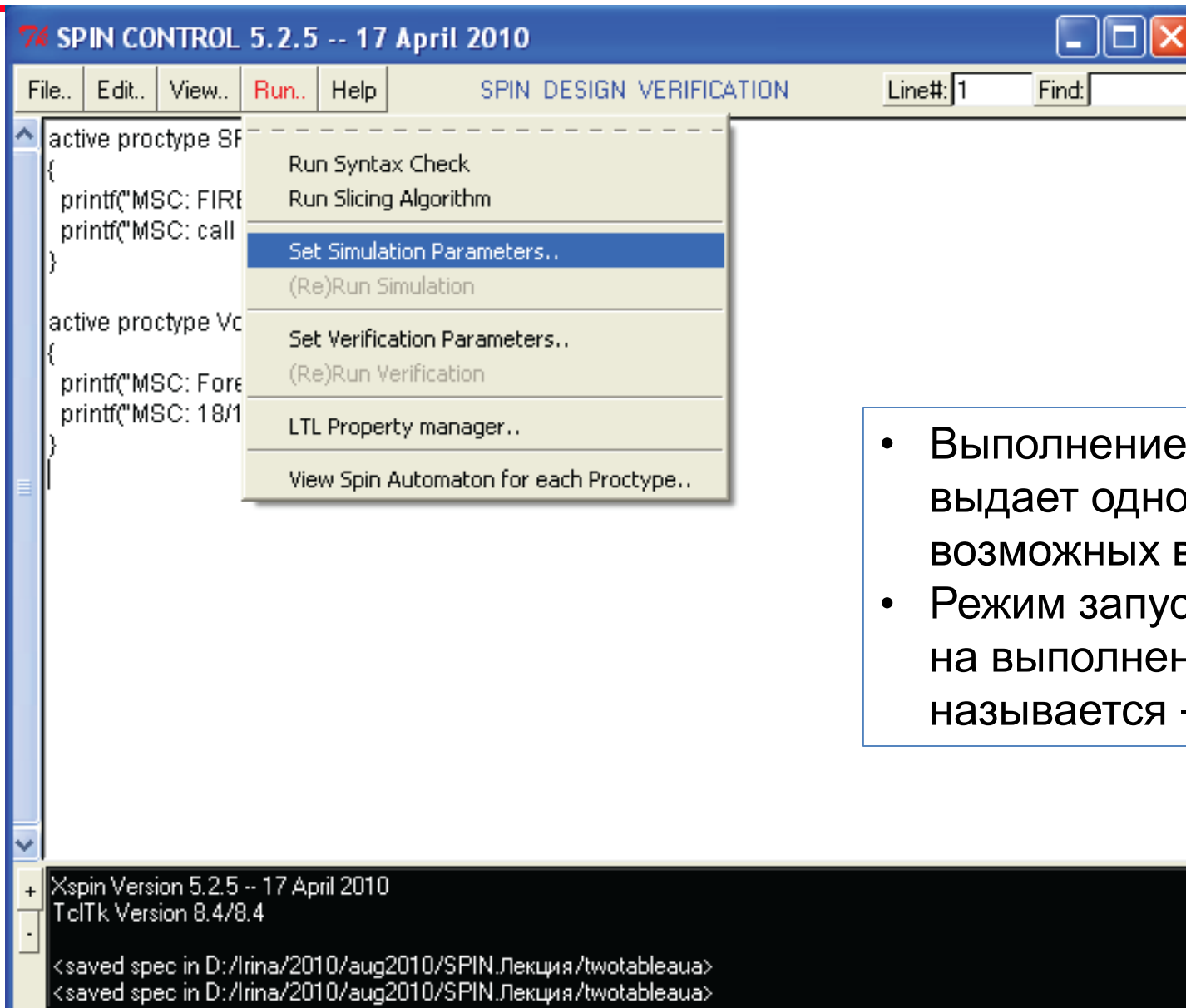
The screenshot shows the SPIN CONTROL 5.2.0 editor window. The title bar reads "SPIN CONTROL 5.2.0 -- 8 May 2009". The menu bar includes "File..", "Edit..", "View..", "Run..", and "Help". The text "SPIN DESIGN" is visible in the top right corner. The main text area contains the following Promela code:

```
^ #define N 10
  int x = 0;

  active[2] proctype p() {
    int t = 0, i = 0;

    do
      :: i < N ->
        atomic{
          t = x;
          printf("MSC: t=%d\n", t);
        }
        atomic{
          x = t + 1;
          printf("MSC: x=%d\n", x);
        }
        i++;
      :: else -> break
    od
  }
```

Запустим модель на выполнение



- Выполнение модели выдает одно из возможных вычислений
- Режим запуска модели на выполнение называется - **симуляция**

Параметры симуляции (вид в Xspin)

Simulation Options

Display Mode

- ☒ MSC Panel - with:
 - ☒ Step Number Labels
 - ☐ Source Text Labels
 - ☒ Normal Spacing
 - ☐ Condensed Spacing
- ☐ Time Sequence Panel - with:
 - ☒ Interleaved Steps
 - ☐ One Window per Process
 - ☐ One Trace per Process
- ☒ Data Values Panel
 - ☒ Track Buffered Channels
 - ☒ Track Global Variables
 - ☐ Track Local Variables
 - ☒ Display vars marked 'show' in MSC
- ☐ Execution Bar Panel

Simulation Style

- ☒ Random (using seed)
Seed Value
- ☐ Guided
 - ☒ Using pan_in.trail
 - ☐ Use
- Steps Skipped
- ☐ Interactive

A Full Queue

- ☒ Blocks New Msgs
- ☐ Loses New Msgs

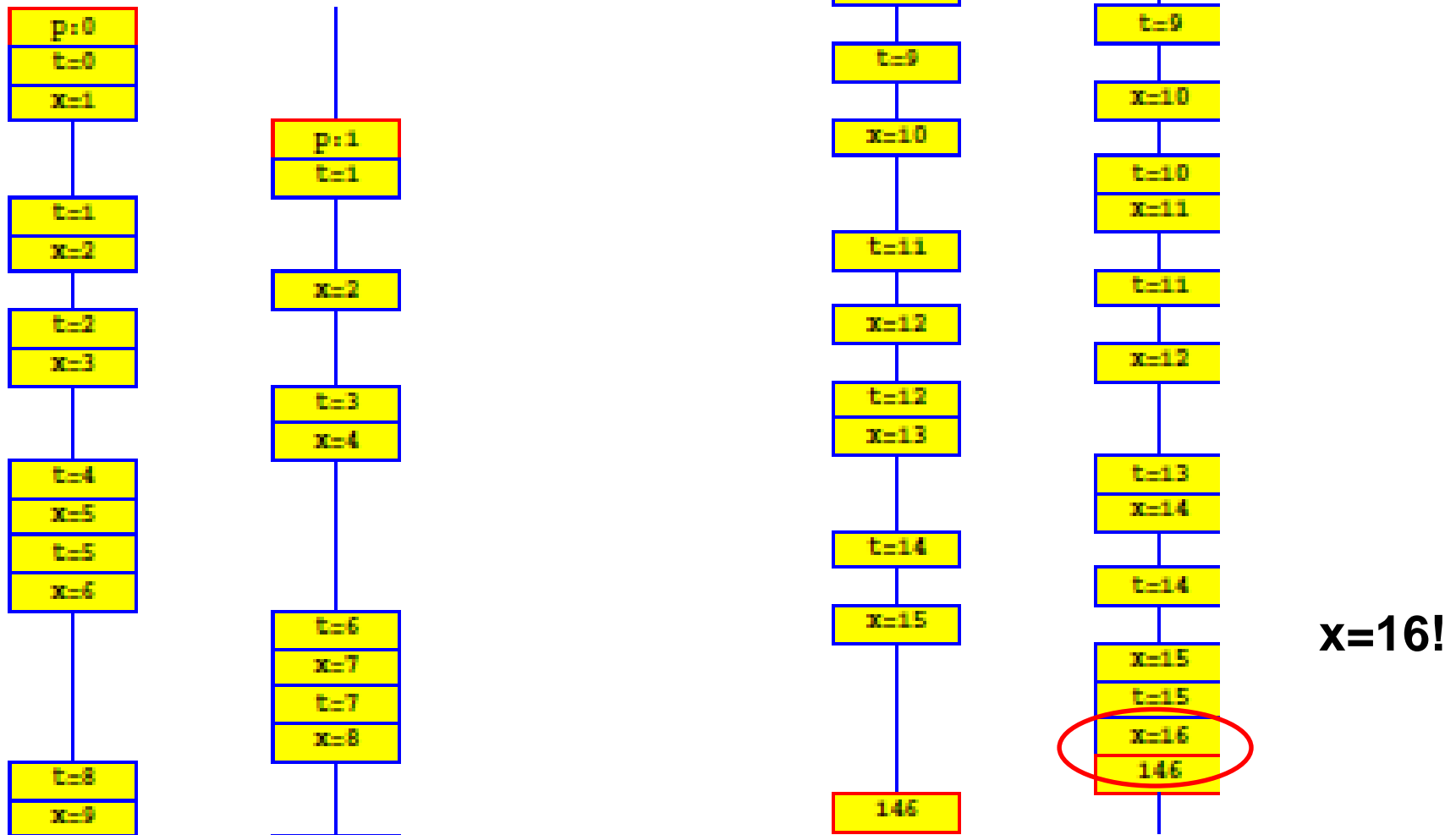
Hide Queues in MSC

Queue nr:

Queue nr:

Queue nr:

Вычисление, сгенерированное SPIN



- Вывод в окне Message Sequence Chart (MSC) XSPIN
- Наглядно видим явление произвольного чередования (interliving) процессов

Формальная семантика языка Promela

Алгоритм на псевдо-коде

```
int x=0; /*Глобальная переменная*/  
  
repeat 10 times {  
    x = x+1;  
}
```

- Вернемся к примеру 1
- Семантика языка Promela
формальная
- Каждая модель на Promela –
помеченная система переходов,
т.е. **структура Крипке**

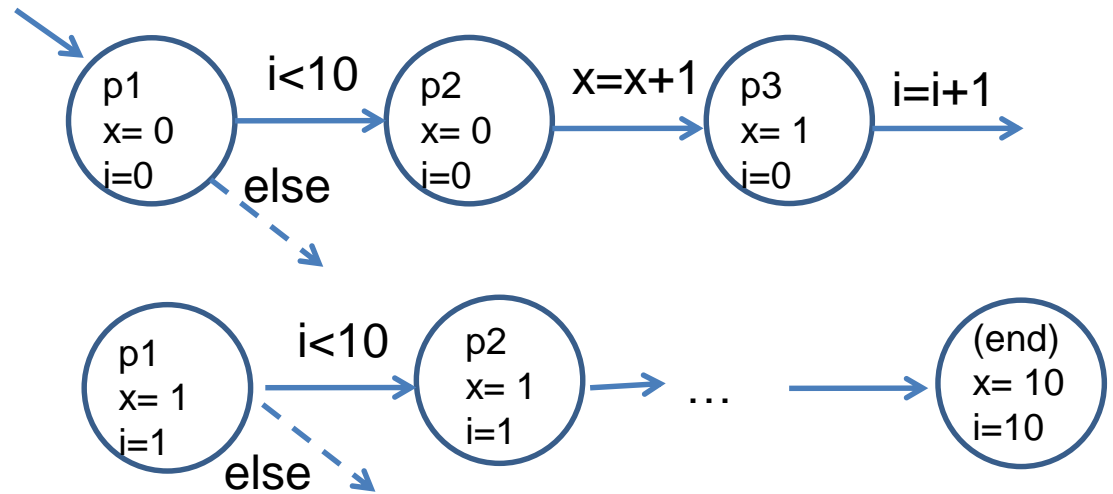
Алгоритм на Promela

```
#define N 10  
int x = 0;  
  
init{  
    int i = 0;  
    do  
        :: i < N ->  
            x = x + 1;  
            i ++;  
        :: else -> break  
    od  
}
```

Построение структуры Крипке по модели на Promela

```
#define N 10
int x = 0;

init{
    int i = 0;
p1:    do
    :: i < N ->
p2:        x = x + 1;
p3:        i ++;
    :: else ->
p4:        break
    od
}
```



- Состояние системы строятся по значения глобальных и локальных переменных, меткам операторов
- Переходы происходят по операторам программы
- Из состояния есть переход, когда в этом состоянии оператор может быть выполнен
- При симуляции выполняется **одно** из возможных вычислений структуры Крипке
- При верификации проверяются **все возможные** вычисления структуры Крипке

Типы данных в Promela

- Модели на Promela **конечны**
 - Количество состояний в системе переходов конечно
 - Все переменные имеют конечный диапазон

Тип переменных	Диапазон
bool	0,1
byte	0..255
int	$-2^{31} .. 2^{31}-1$
chan	1..255
mtype	1..255
pid	0..255

Пример 3.

Что произойдет при моделировании процесса Q после того, когда i достигнет значения 255?

```
byte i = 0;  
active proctype Q()  
{  
  do  
    :: i ++;  
  od  
}
```

Значение i не может быть больше 255!

Выполнимость и невыполнимость операторов в SPIN

Пример 4.

```
byte i = 0;  
active proctype Q() {  
  i ++;  
}
```

`i++` - `i` было 0, `i` станет равным 1, остались в пределах диапазона этого типа переменной - **ВЫПОЛНИТСЯ!**

Пример 5.

```
byte i = 0;  
active proctype P() {  
  (i == 2);  
  printf("Hello i=%d", i);  
}
```

`i==2` – это проверка, что `i` равно 2, но `i` равно 0 – **не ВЫПОЛНИТСЯ!**

- Все операторы в SPIN **проверяются на выполнимость**
 - Кроме `skip`, `break`, `printf` – они всегда выполнимы
- Если оператор выполнить **нельзя**, то процесс **заблокируется**

Синхронизация по данным

Пример 6.

```
byte i = 0;
active proctype Q() {
    i ++;
    i ++
}
active proctype P() {
    (i == 2);
    printf("Hello i=%d", i);
}
```

Объединим два
предыдущих примера

i – глобальная переменная

Процесс *P* при проверке условия заблокируется и будет ждать,...
пока процесс *Q* не поменяет переменную *i*

Оба процесса успешно завершат работу

Невыполнимость и предел диапазона типа переменной

Пример 3.

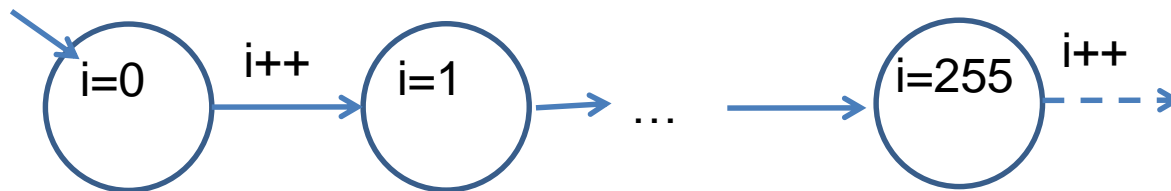
```
byte i = 0;
```

$i = 255$

```
active proctype Q() {  
do  
:: i ++;  
od  
}
```

Что будет, если модель достигает
предела диапазона типа переменной?

- SPIN **проверит** можно ли **выполнить** оператор `i ++`
- У `i++` нет защитного условия. Это синтаксический сахар. Здесь опущено условие `true`
- `i++` будет **нельзя** выполнить, когда `i` достигнет предел диапазона
- Процесс Q **заблокируется**
 - Процесс Q не выйдет из цикла! Он остановится



- Из состояния `i=255` нет перехода!

Оператор цикла и выполнимость условий

Пример 7. Как выполняется такая модель в SPIN?

```
bool i = 0;  
active proctype Q() {  
do  
:: i -> break  
od  
}
```

- Если бы $i=1$, то процесс выполнил бы break и вышел из цикла
- Но $i=0$

Оператор цикла и выполнимость условий

Пример 8. Как выполняется такая модель в SPIN?


```
bool i = 0;
active proctype Q() {
do
:: i -> break
od
}
```

- Если бы $i=1$, то процесс выполнил бы break и вышел из цикла
- Но $i=0$

Семантика оператора цикла:

- Если не выполняется **ни одно из условий**, процесс блокируется
- Если выполняется **несколько условий**, то одно из них выбирается недетерминированно (при моделировании)
- В **режиме верификации** рассматриваются все возможные варианты условий

```
do
:: условие -> список команд
:: условие -> список команд
...
:: условие -> список команд
od
```



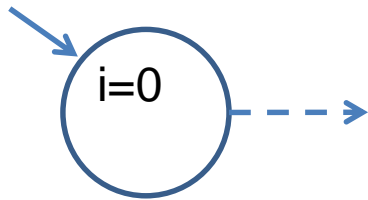
guard – защита, условие, охрана

Оператор цикла и выполнимость условий

Пример 8. Как выполняется такая модель в SPIN?

```
active proctype Q() {  
  bool i = 0;  
  do  
    :: i -> break  
  od  
}
```

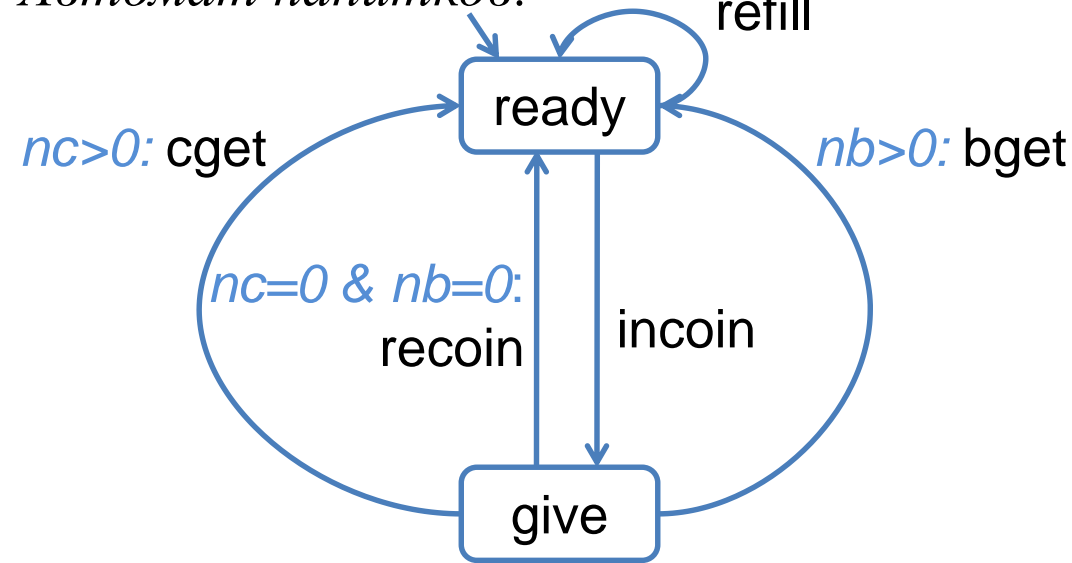
- Процесс Q не выйдет из цикла!
Он остановится



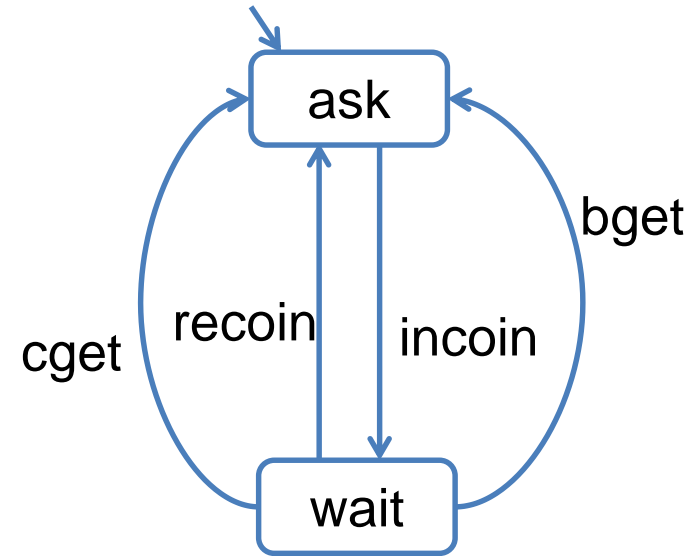
- Если бы `i=1`, то процесс выполнил бы `break` и вышел из цикла
- Но `i=0`
- Иными словами, SPIN проверяет, выполняется ли хотя бы одно условие в цикле, если нет, то процесс блокируется
- На самом деле: **все операторы** в Promela проверяются на **выполнимость**
- *Если оператор не выполним, то процесс блокируется*

Пример 9. Автомат напитков и студент

Автомат напитков:



Студент:



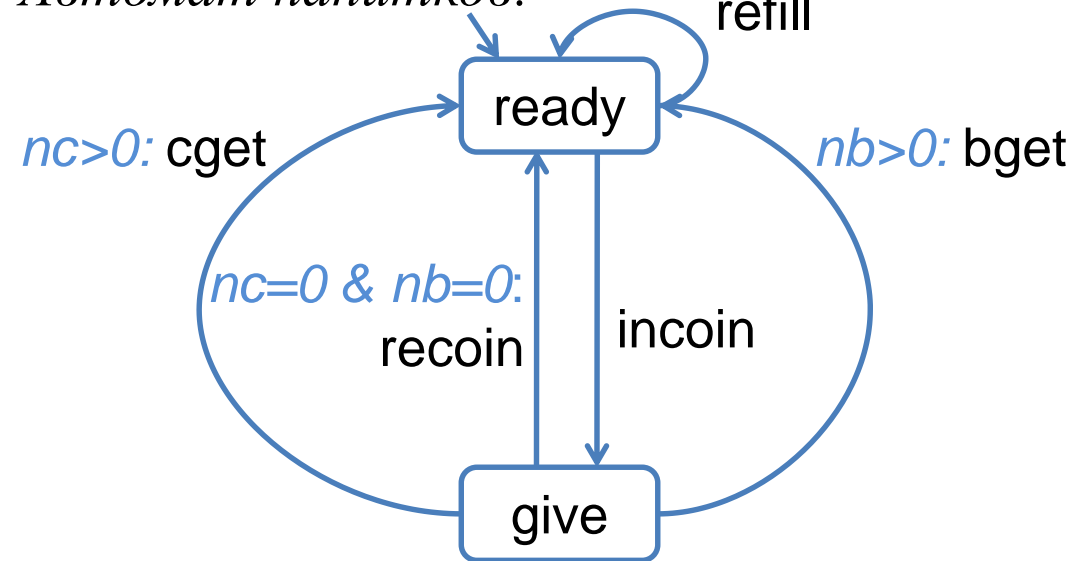
- Поведение автомата и студента независимы -> моделируем отдельными процессами

- *nc* – количество банок колы
- *nb* – количество банок пива
- *incoin* – в автомат бросили монету

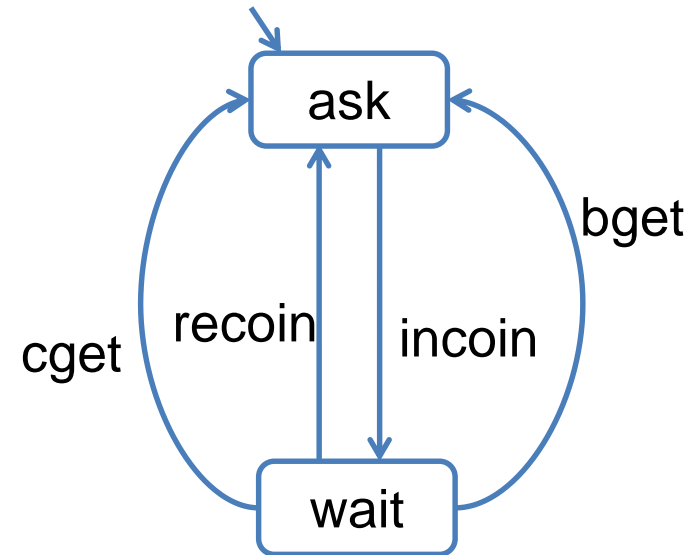
- *recoin* – автомат вернул монету
- *cget* – автомат выдал колу
- *bget* – автомат выдал пива

Взаимодействие процессов: каналы

Автомат напитков:



Студент:

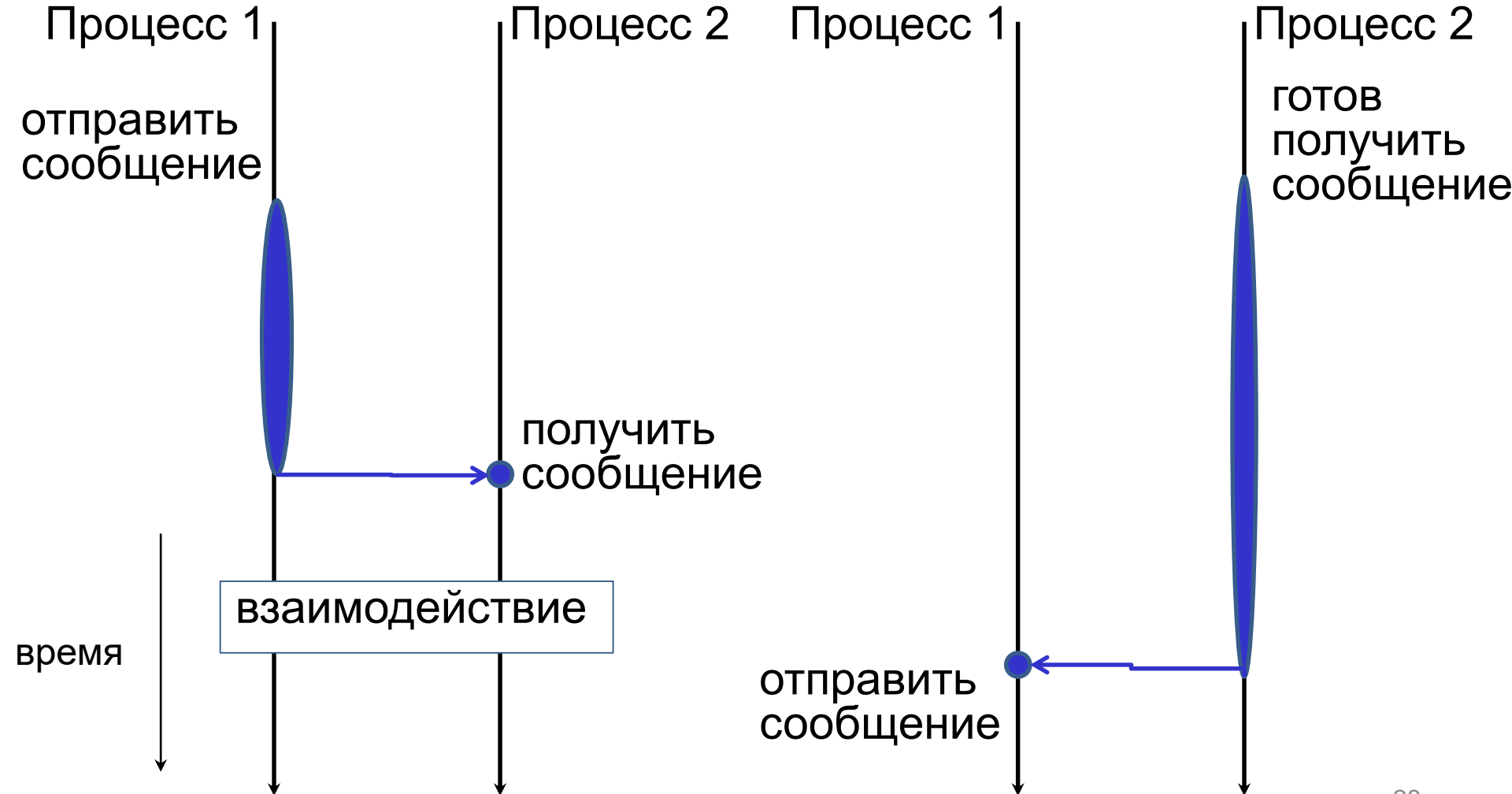


Как только студент бросил монетку, автомат **сразу** её получил!

- Такое взаимодействие называется **синхронным**
- Моделируется с помощью **каналов**
- У нас здесь будет два канала: один от студента к автомату **stch**, другой от автомата студенту **mcch**
- Это каналы без памяти (канала с нулевым объемом, синхронные каналы) – **рандеву-канал**

Рандеву-каналы (handshaking)

Каналы с синхронным взаимодействием



Предложено Э. Дейкстрой

Объявление каналов в Promela

```
chan mcch = [0] of {mtype};
```

канал сообщений, исходящих от автомата напитков

переменная
типа канал

емкость канала 0
свидетельствует
о рандеву-канале

формат сообщения mtype

```
chan stch = [0] of {mtype};
```

канал сообщений, исходящих от студента

- Типы сообщений системы

```
mtype = {incoin, cget, bget, recoin};
```

типы сообщений
системы

mtype – специальный перечислимый тип в Promela,
удобен для задания сообщений

Операции с каналами

mcch ! incoin ←

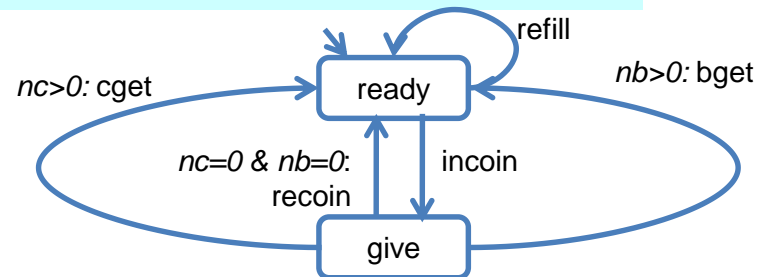
отправить
сообщение

mcch ? incoin ←

получить
сообщение

Модель автомата напитков

```
active proctype Machine() {  
    int nc = NMAX, nb = NMAX, state = ready;  
    do  
        :: (state == ready) ->  
            nc = NMAX; nb = NMAX; printf("MSC: refilled\n")  
        :: (state == ready) ->  
            mcch ? incoin; state = give  
        :: (state == give) && (nc == 0) && (nb == 0) ->  
            stch ! recoin; state = ready  
        :: (state == give) && (nc > 0) ->  
            stch ! cget; state = ready  
        :: (state == give) && (nb > 0)  
            stch ! bget ; state = ready  
    od  
}
```



Недетерминизм выбора условий в операторе цикла

guard – защита, условие, охрана

ЦИКЛ

```
do
  :: (state == ready) -> /*...*/
  :: (state == ready) -> /*...*/
  :: (state == give) && (nc == 0) && (nb == 0) -> /*...*/
  :: (state == give) && (nc > 0) -> /*...*/
  :: (state == give) && (nb > 0) -> /*...*/
od
```

Семантика цикла (повтор):

- Если выполняется **несколько условий**, то одно из них выбирается недетерминированно (при моделировании)
- В **режиме верификации** рассматриваются все возможные варианты условий

Оператор выбора также выполняется недетерминированно

Модель на Promela поведения студента

```
active proctype Student() {  
  mtype msg;  
  int state = ask;  
  do  
    :: (state == ask) -> mcch ! incoin; state = wait  
    :: (state == wait) -> stch ? msg; guard – защита, условие  
    if  
      :: (msg == recoin) -> printf("MSC: try again\n")  
      :: (msg == cget) -> printf("MSC: get cola\n")  
      :: (msg == bget) -> printf("MSC: get beer\n");  
    fi;  
    state = ask  
  od  
}
```

Оператор выбора

```
if  
  :: условие -> список команд  
  :: условие -> список команд  
  ...  
  :: условие -> список команд  
fi
```

- Структура оператора выбора похожа на структуру оператора цикла

Верификация LTL формул

Сформулируем требование

Когда-нибудь в будущем студент получит банку пива?

getbeer - атомарный предикат

F getbeer – на всех путях когда-нибудь в будущем студент получит банку пива

Зададим требование в редакторе требований SPIN

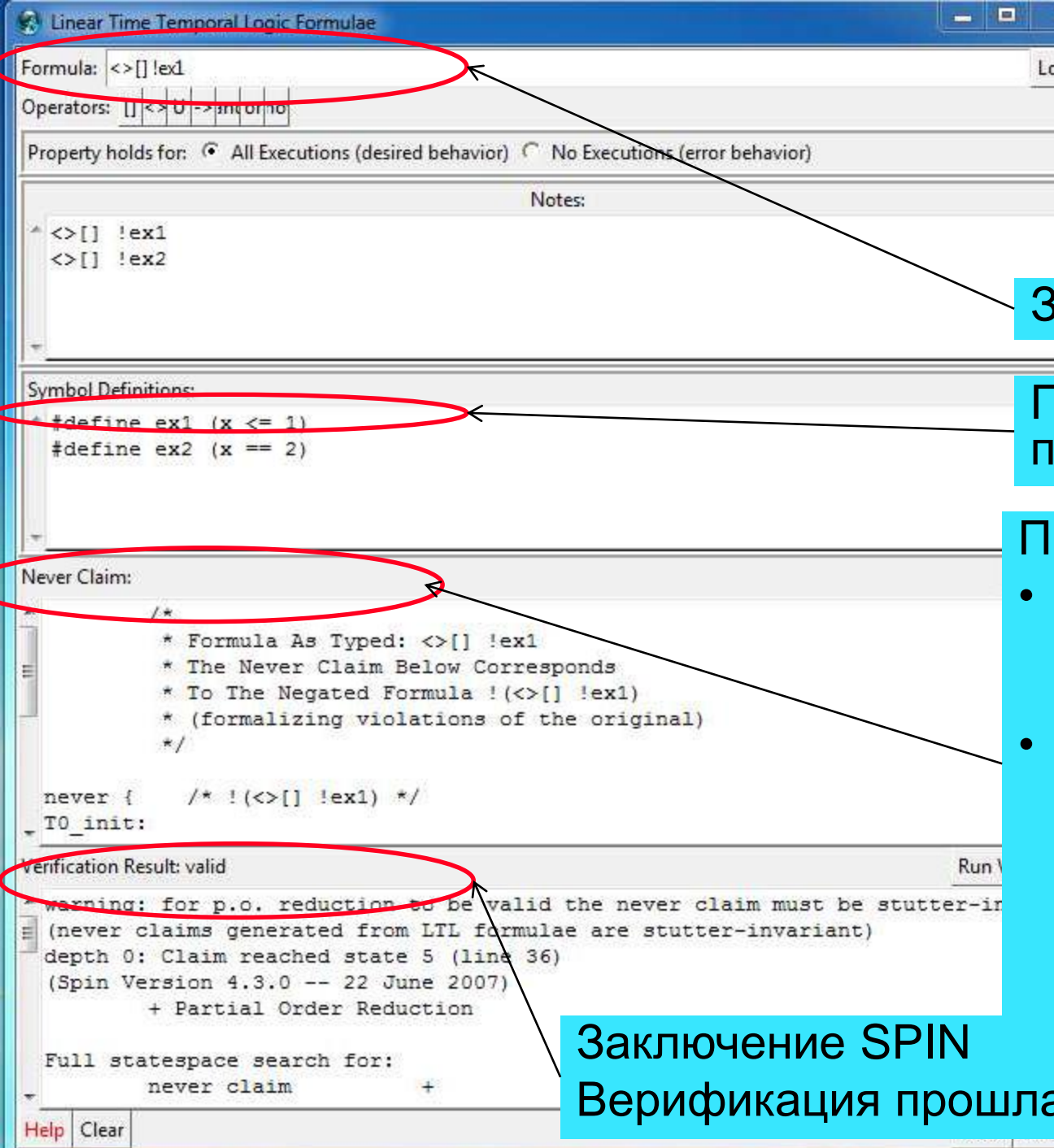
Синтаксис LTL формул в SPIN

<>	F	когда-нибудь в будущем на каком-нибудь пути будет выполняться свойство
[]	G	всегда в будущем на всех путях будет выполняться заданное свойство
U	U	Until
!	\neg	отрицание
&&	\wedge	конъюнкция
	\vee	дизъюнкция
->	\rightarrow	импликация

LTL-формула выполняется для любого пути, стартовавшего в допустимом начальном состоянии

- LTL-формула

F getbeer отображается как **<> getbeer**



Окно редактора LTL формул

Запись формулы

Пропозициональная переменная

Процесс never claim

- Отрицание LTL формулы на языке Promela
- При верификации строится синхронная композиция процесса never claim и модели

Заключение SPIN
Верификация прошла успешно!

Formula: <> getbeer

Operators: ☐ [] ☐ <> ☐ U ☐ -> ☐ and ☐ or ☐ no

Property holds for: ☒ All Executions (desired behavior) ☐ No Executions (error behavior)

Notes:

Symbol Definitions:

Never Claim:

```
/*
 * Formula As Typed: <> getbeer
 * The Never Claim Below Corresponds
 * To The Negated Formula !(<> getbeer)
 * (formalizing violations of the original)
 */

never { /* !(<> getbeer) */
accept_init:
T0_init;
```

Verification Result: not valid

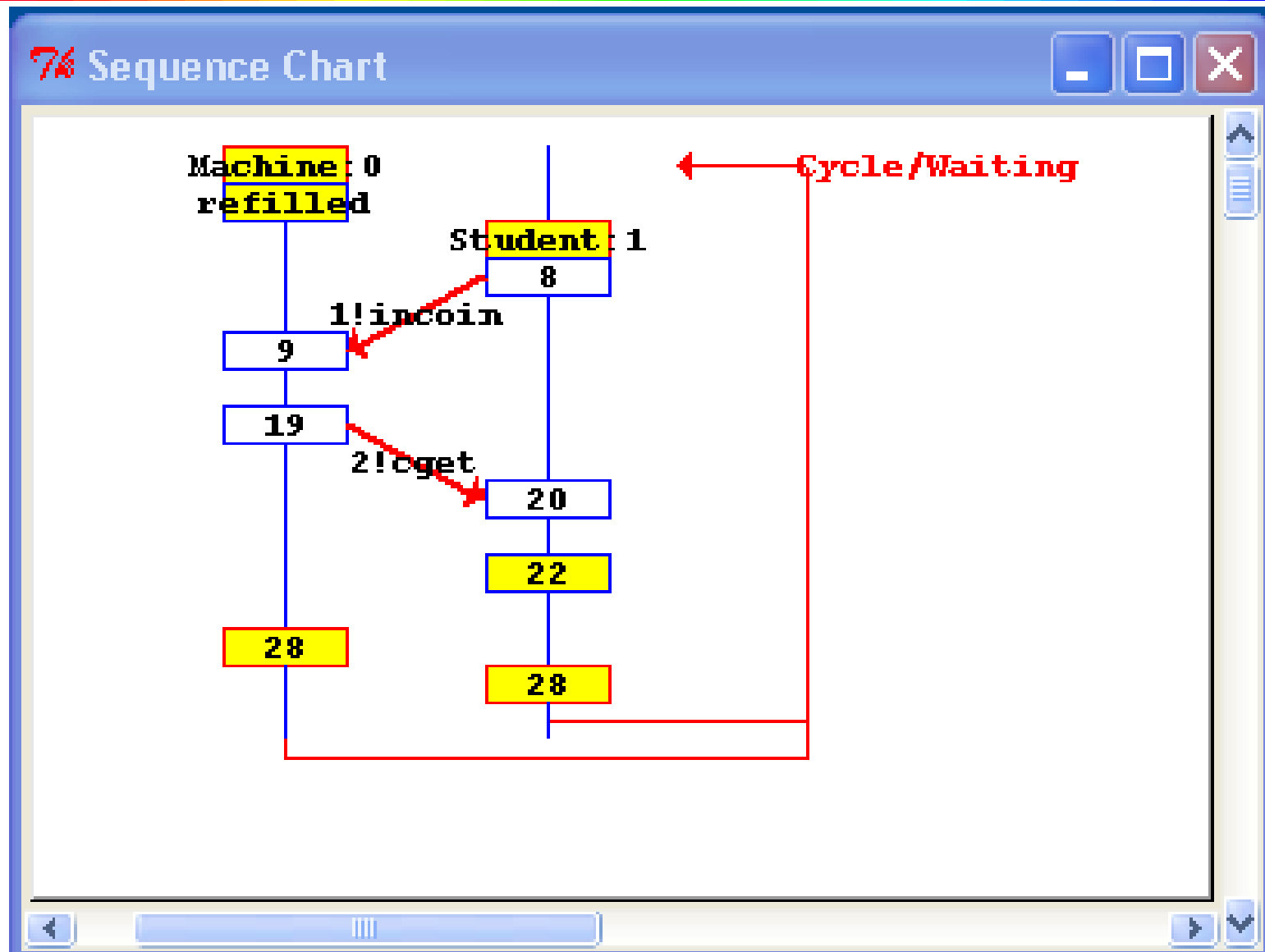
depth 0: Claim reached state 3 (line 51)
pan: 1: acceptance cycle (at depth 2)
pan: wrote pan_in.trail

(Spin Version 5.2.5 -- 17 April 2010)
Warning: Search not completed

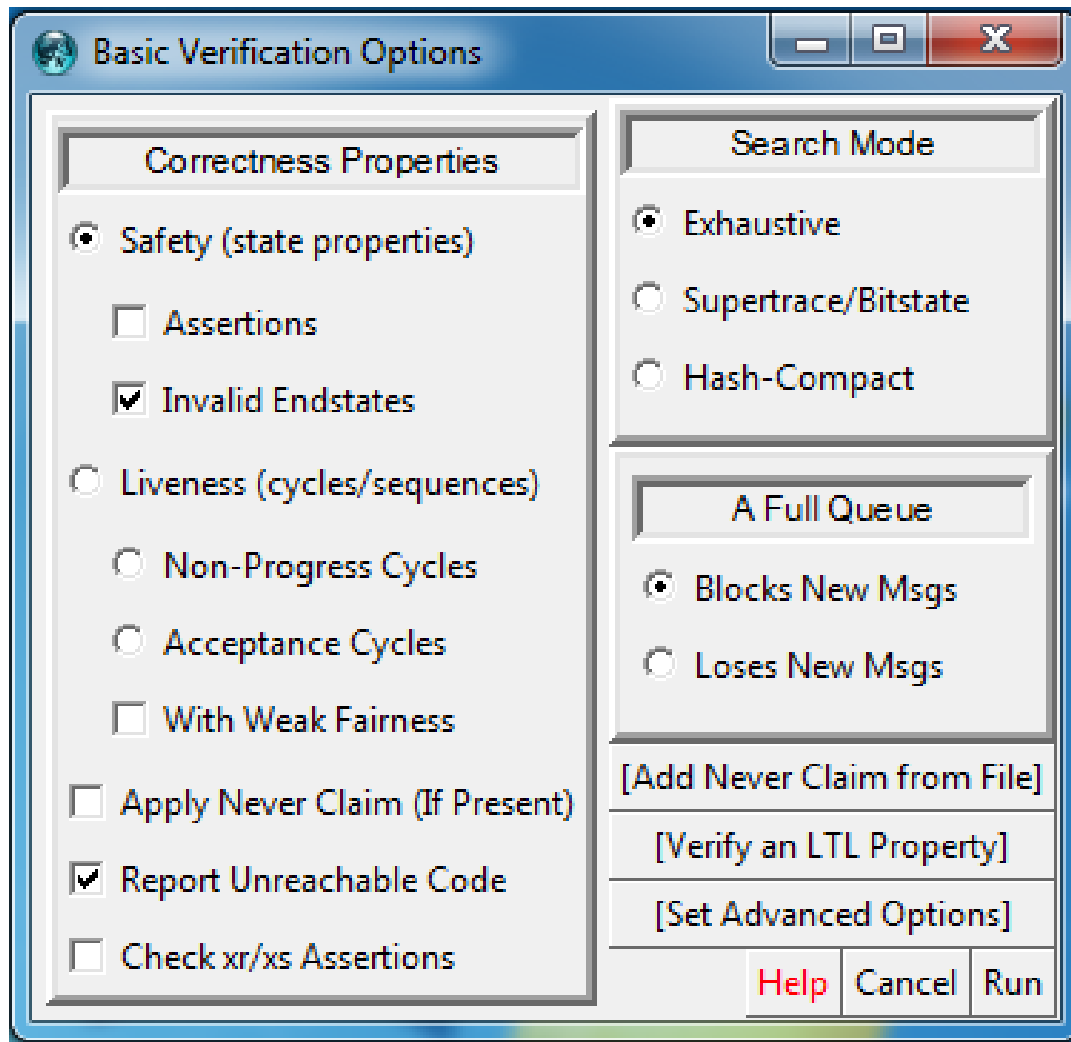
Full statespace search for:
☒ never claim +

Верификация
прошла неуспешно
Требование
нарушено

Контрпример в окне диаграммы взаимодействия



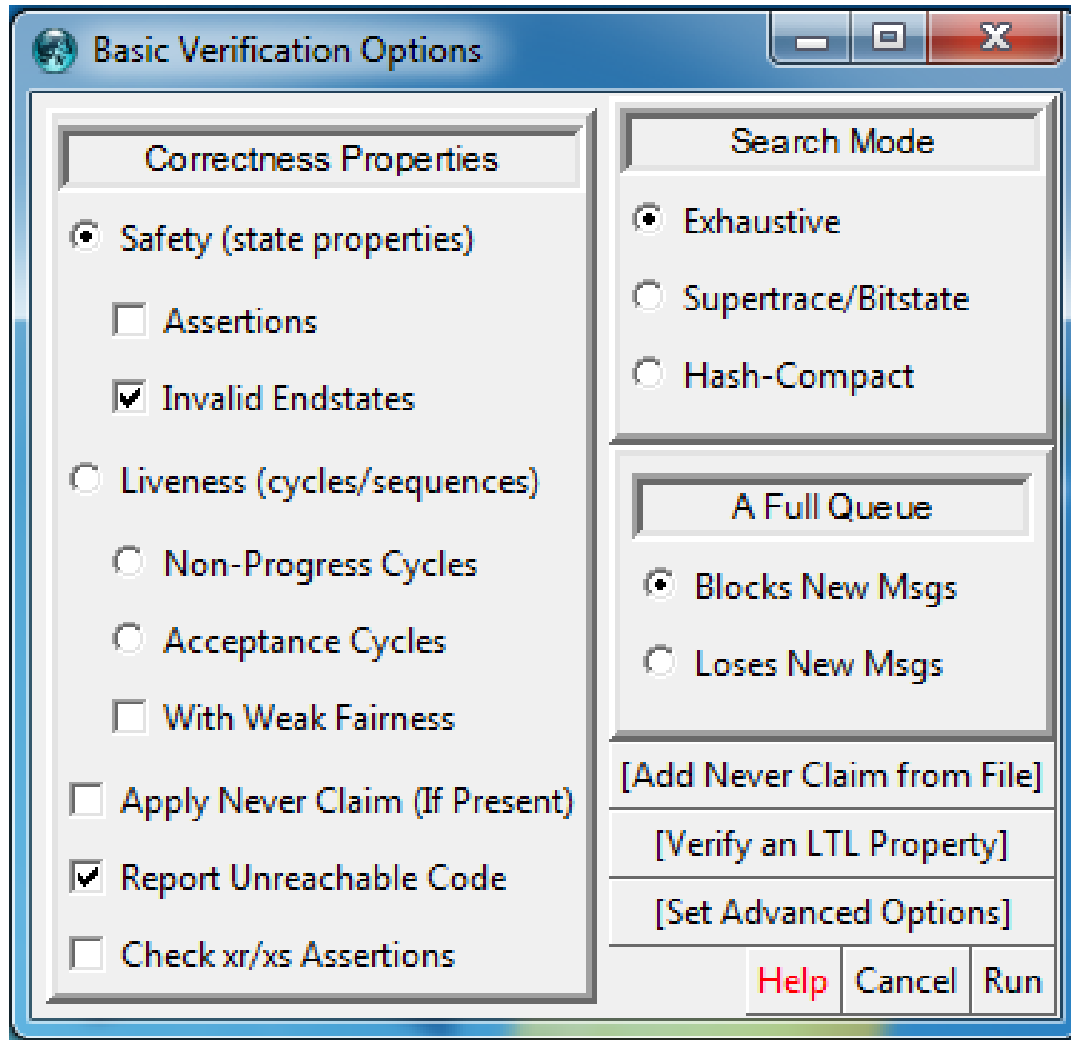
Параметры верификации SPIN



- Точный (учитывается 100% состояний)
 - **Exhaustive**
- Аппроксимационные (учитывается 90% состояний)
 - **Supertrace/Bitstate** – супертрассы (с потерей состояний)
 - **Hash-Compact** – КОМПАКТНОЕ хэширование

Сдача курсовой производится в **ТОЧНОМ** режиме верификации

Параметры верификации SPIN



- Safety – проверка встроенных свойств (безопасности)
- Liveness – проверка свойств пользователя, заданных LTL - формулой

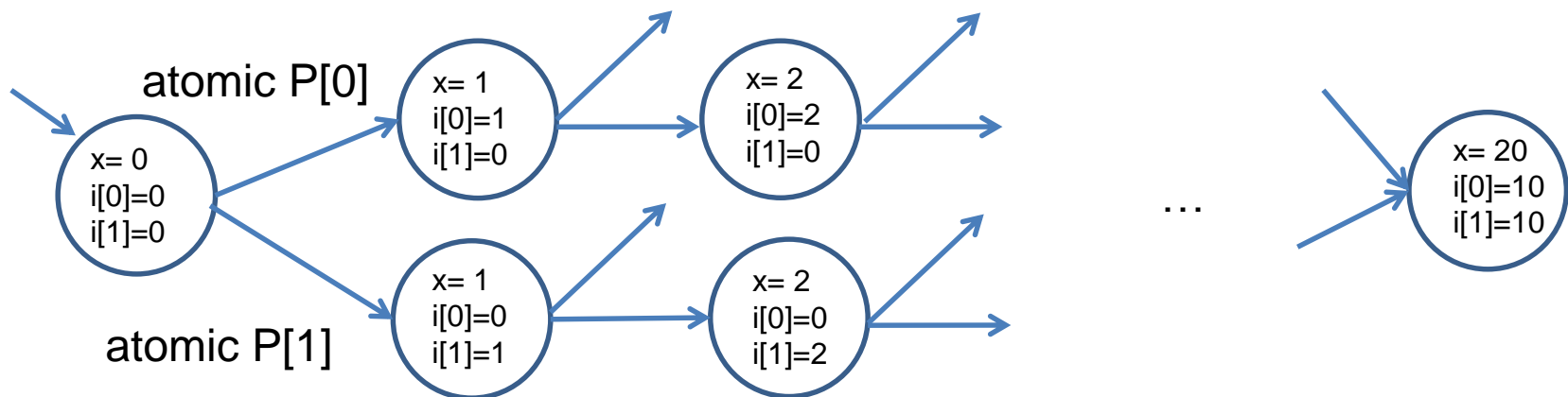
Атомарность операций играет большую роль в распределенных алгоритмах

```
#define N 10
int x = 0;
active[2] proctype P() {
  int t = 0, i = 0;
  do
    :: i < N ->
      atomic{
        t = x; printf("MSC: t=%d", t);
        x = t + 1; printf("MSC: x=%d", x);
        i ++;
      }
    :: else -> break
  od
}
```

Пример 2. Использование оператора `atomic`

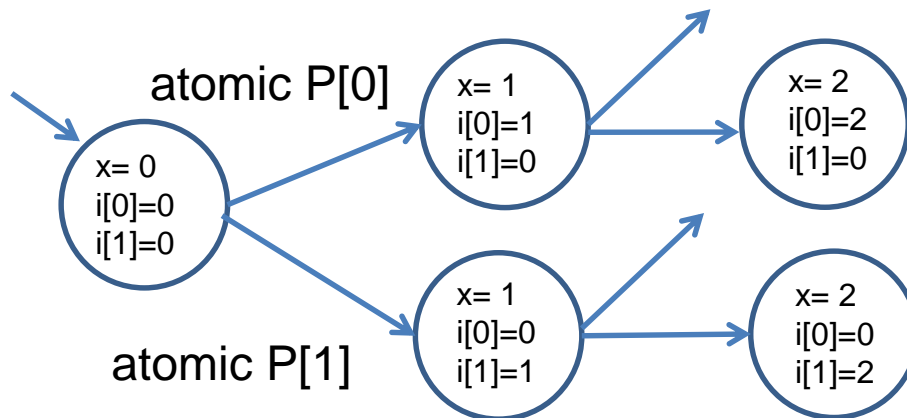
Если нужно объединить, несколько операций в одну – неделимую – атомарную, то в SPIN это сделать просто с **atomic**

На любом вычислении в конце будет $x=20$!



Оператор atomic

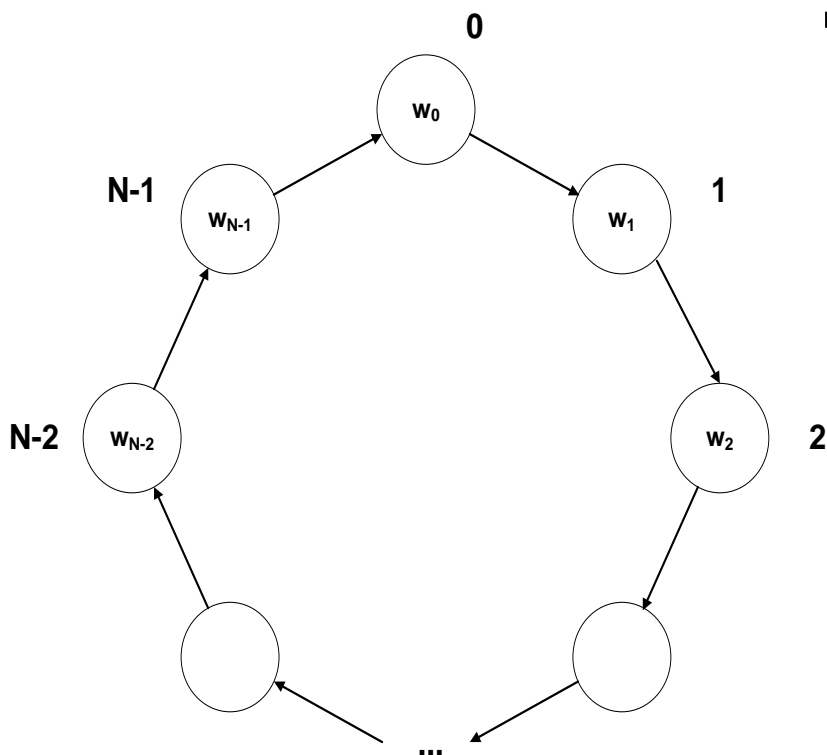
```
#define N 10
int x = 0;
active[2] proctype P() {
  int t = 0, i = 0;
  do
    :: i < N ->
      atomic{
        t = x; printf("MSC: t=%d", t);
        x = t + 1; printf("MSC: x=%d", x);
        i ++;
      }
    :: else -> break
  od
}
```



Пример 2. Использование оператора atomic

- Необходимо чрезвычайно аккуратно пользоваться конструкцией **atomic**,
 - чтобы **не уничтожить необходимое чередование**
 - При моделировании стремятся **отразить реальные механизмы, на практике атомарность требует специальных механизмов синхронизации**
- Пользуйтесь **atomic**, только есть необходимость и понимание

Пример 6. Задача выбора лидера



Дано: однонаправленное кольцо

- количество узлов N
- веса узлов w_i ($i=0..N-1$) уникальны
- узлы взаимодействуют только с соседями
- количество узлов фиксировано
- узлы взаимодействуют с помощью асинхронных каналов

Требуется построить протокол:
набор **ЛОКАЛЬНЫХ** правил для
каждого узла, которые позволят
получить **ГЛОБАЛЬНЫЙ** результат -
каждому узлу определить лидера

- например, узел с наибольшим весом

Есть **эффективный** алгоритм выбора лидера (Dolev-Klawe-Rodeh, Peterson)

количество сообщений – $2N\log_2 N + O(N)$

Алгоритм выбора лидера Петерсона

Цель: каждый узел должен определить максимальный вес во всем кольце

- В начале узел знает только свой вес

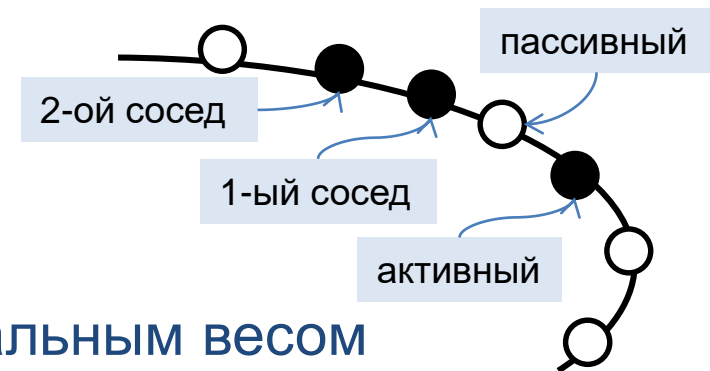
Каждый узел или *активный*,
или *пассивный*

Активный узел

- характеризуется **локальным максимальным весом**
- и **весом ближайшего соседа слева** (из числа активных)
- обрабатывает информацию от **двух** ближайших активных соседей слева

Пассивный узел

- **не имеет** текущего веса
- **пропускает** через себя сообщения, не обрабатывая



Узлы взаимодействуют друг с другом с помощью **асинхронных** каналов

Набор локальных правил алгоритма

активного узла

- **max** – локальный максимум – свой текущий вес
- **left** – текущий вес активного соседа слева

A0. $\text{max} = w_i$

послать сообщение **one(max)**

A1. обработка сообщения с текущим весом активного соседа слева (обработка сообщения **one**), на этой фазе *может определиться лидер*

A2. обработка сообщения с текущим весом «левого» соседа своего активного соседа (обработка сообщения **two**); на этой фазе происходит *переход в состояние пассивности*

A3. фаза сообщения о лидере

Фазы **A1**, **A2** сменяют друг друга, пока не будет найден лидер

Набор локальных правил алгоритма

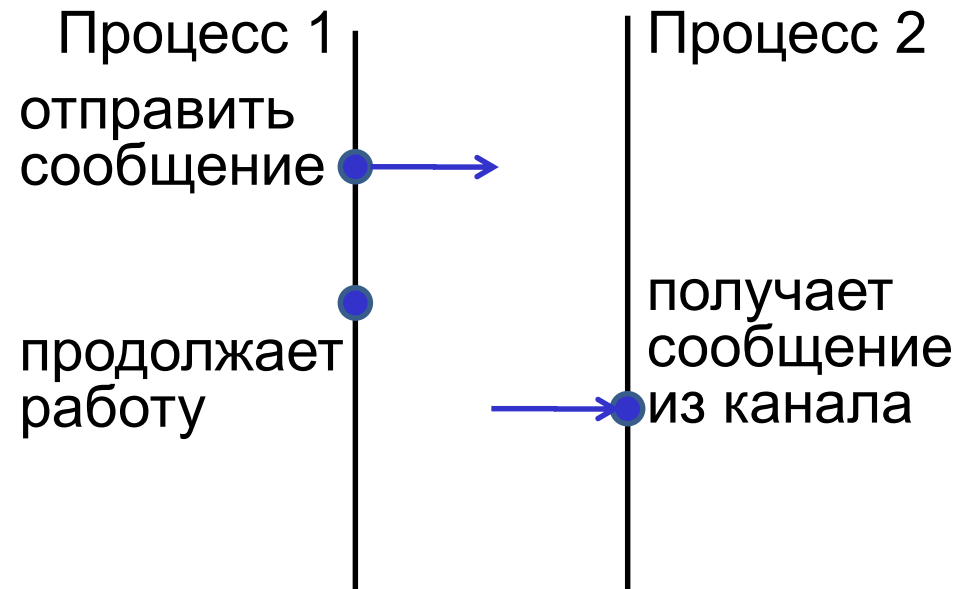
активного узла

- **max** – локальный максимум – свой текущий вес
- **left** – текущий вес активного соседа слева

```
A0. max =  $w_i$ 
    послать сообщение one(max)
A1. получить сообщение one(q)
    если ( $q \neq \text{max}$ ) то
        left := q
        послать сообщение two(left)
    иначе послать сообщение winner(max)
    max является глобальным максимумом
A2. получить сообщение two(q)
    если ( $\text{left} > q$ ) и ( $\text{left} > \text{max}$ ) то
        max := left
        послать сообщение one(max)
    иначе узел становится пассивным
A3. фаза сообщения о лидере
```

Фазы **A1**, **A2** сменяют друг друга, пока не будет найден лидер

Каналы. Асинхронное взаимодействие процессов



- Асинхронные каналы имеют **конечную ненулевую емкость**
- Сообщение может быть записано в канал, если в канале есть место
- Процесс может прочитать сообщение из канала, если канал не пуст

- Каналы также характеризуются **дисциплиной обслуживания**
 - В Promela по умолчанию FIFO
 - Реализация в Promela другой дисциплины обслуживания требует использования специальных механизмов

Асинхронные каналы в Promela

- Типы сообщений

```
mtype = {one, two, winner};
```

Узлы обмениваются сообщениями по каналам:

```
chan p[N] = [L] of {mtype, byte};
```

массив
каналов

емкость канала > 0 ,
значит, канал
асинхронный

формат сообщения,
например, `one(q)`
то же, что и `one, q`

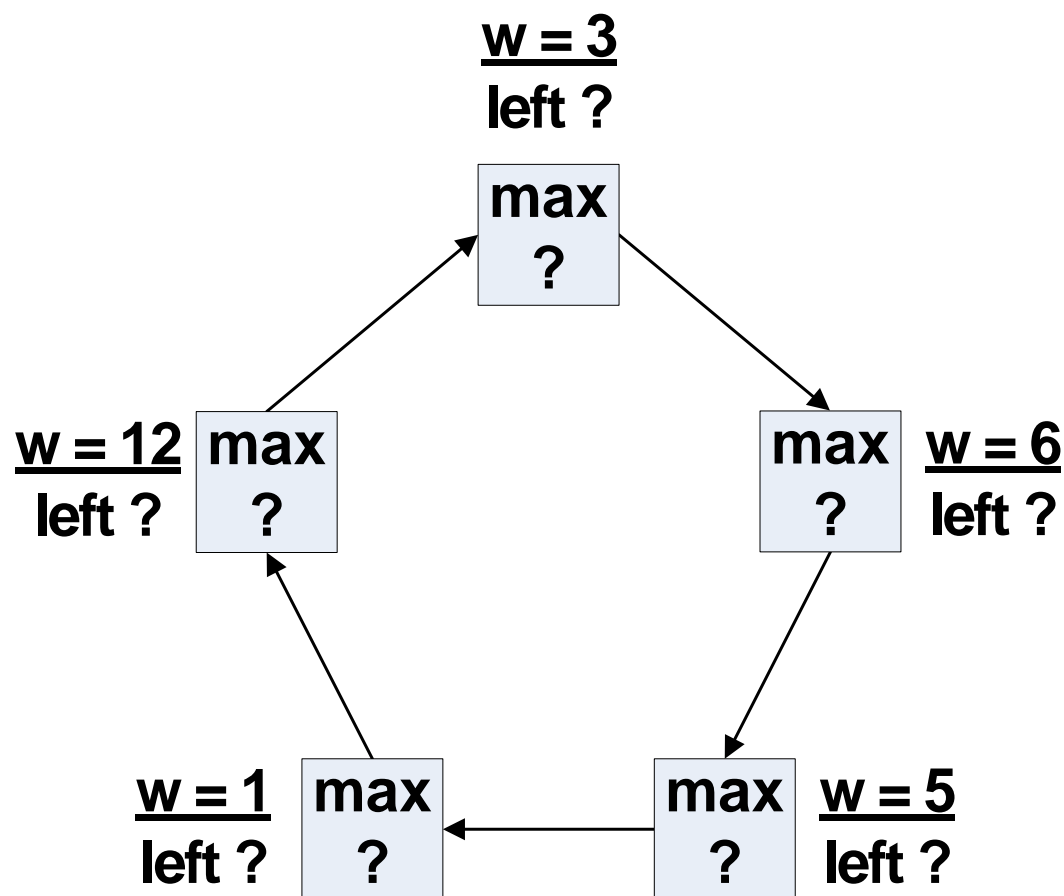
```
out ! one(q)
```

отправить
сообщение

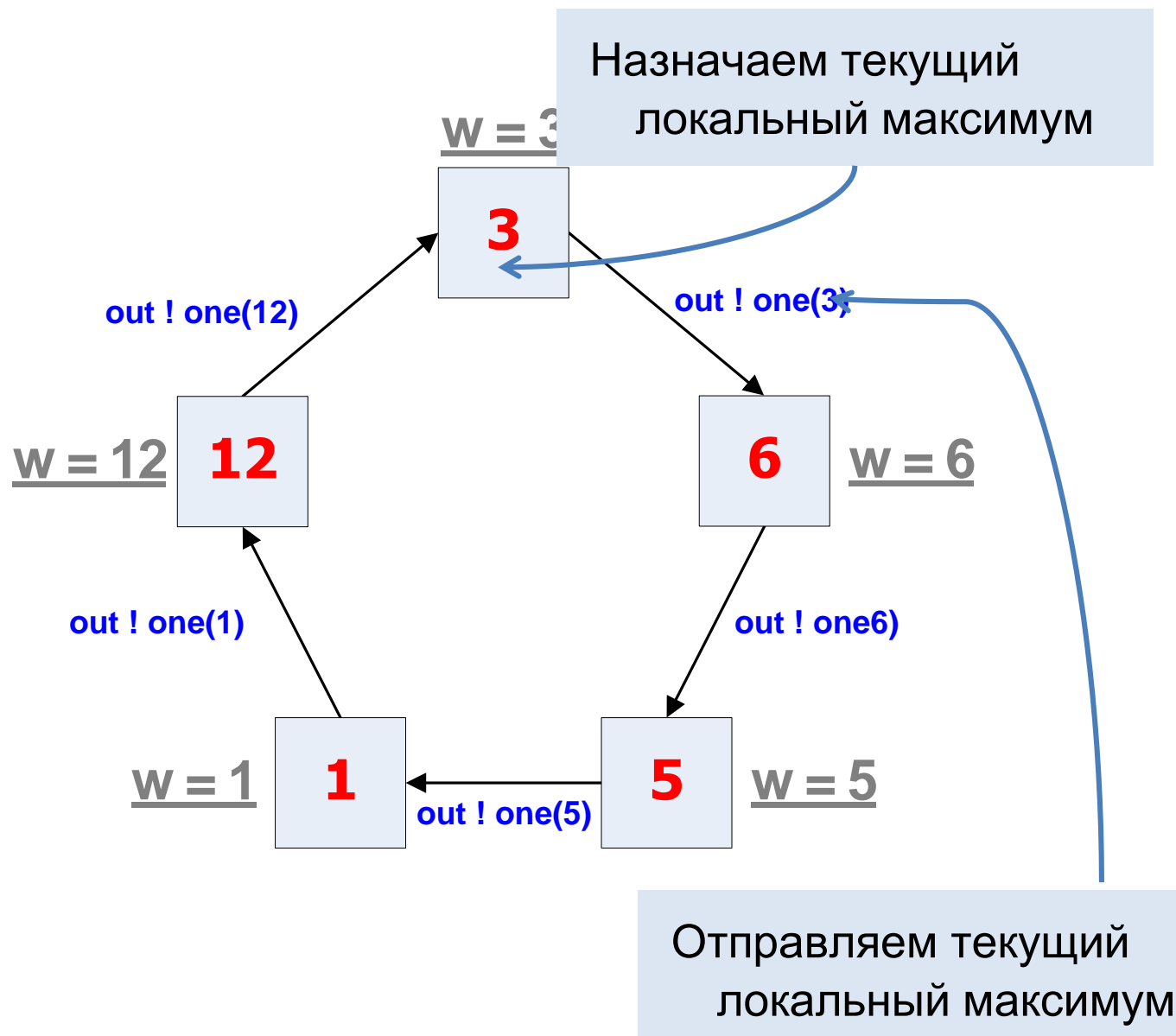
```
in ? one(q)
```

получить
сообщение

Пример. A0. Начальная фаза



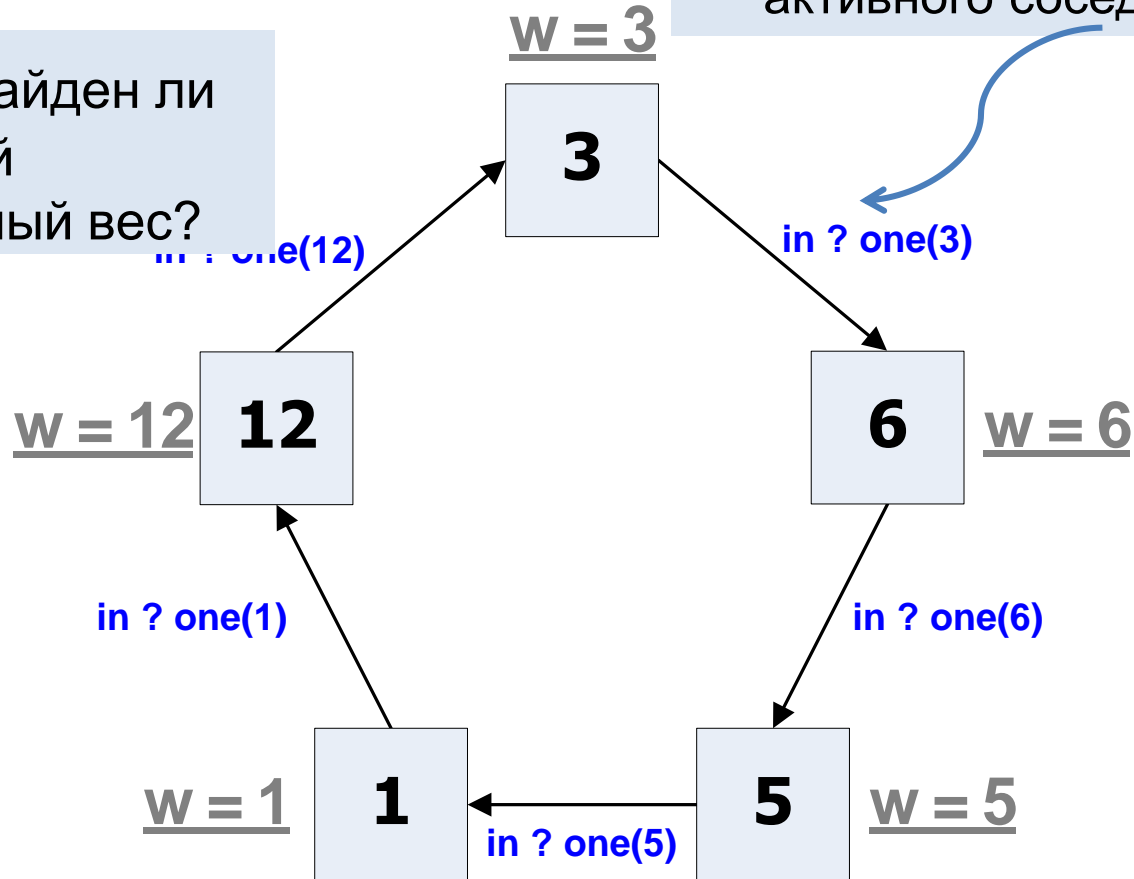
Пример. A0. Начальная фаза



Пример. Фаза A1

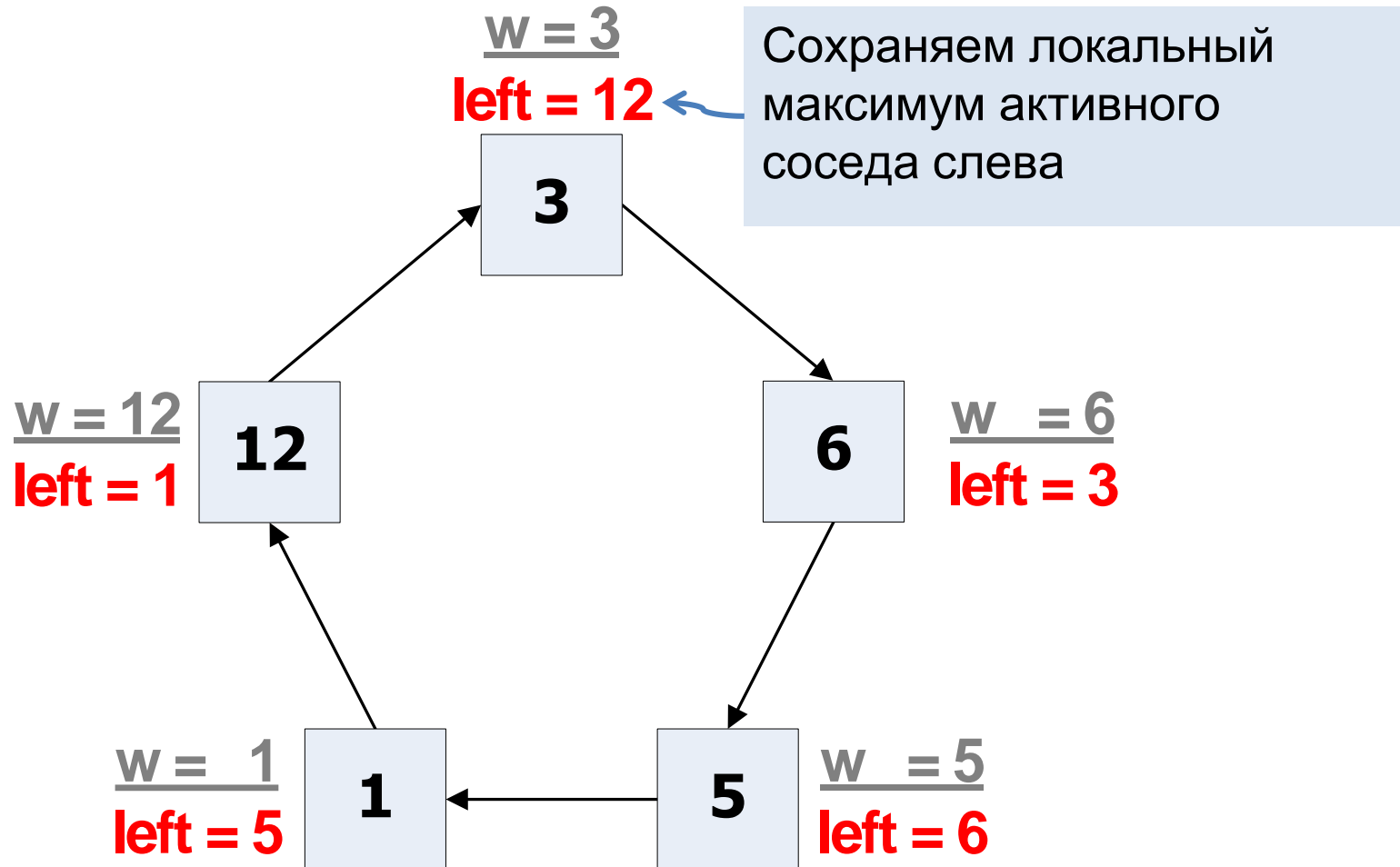
Проверяем: найден ли абсолютный максимальный вес?

Получаем локальный максимум активного соседа слева

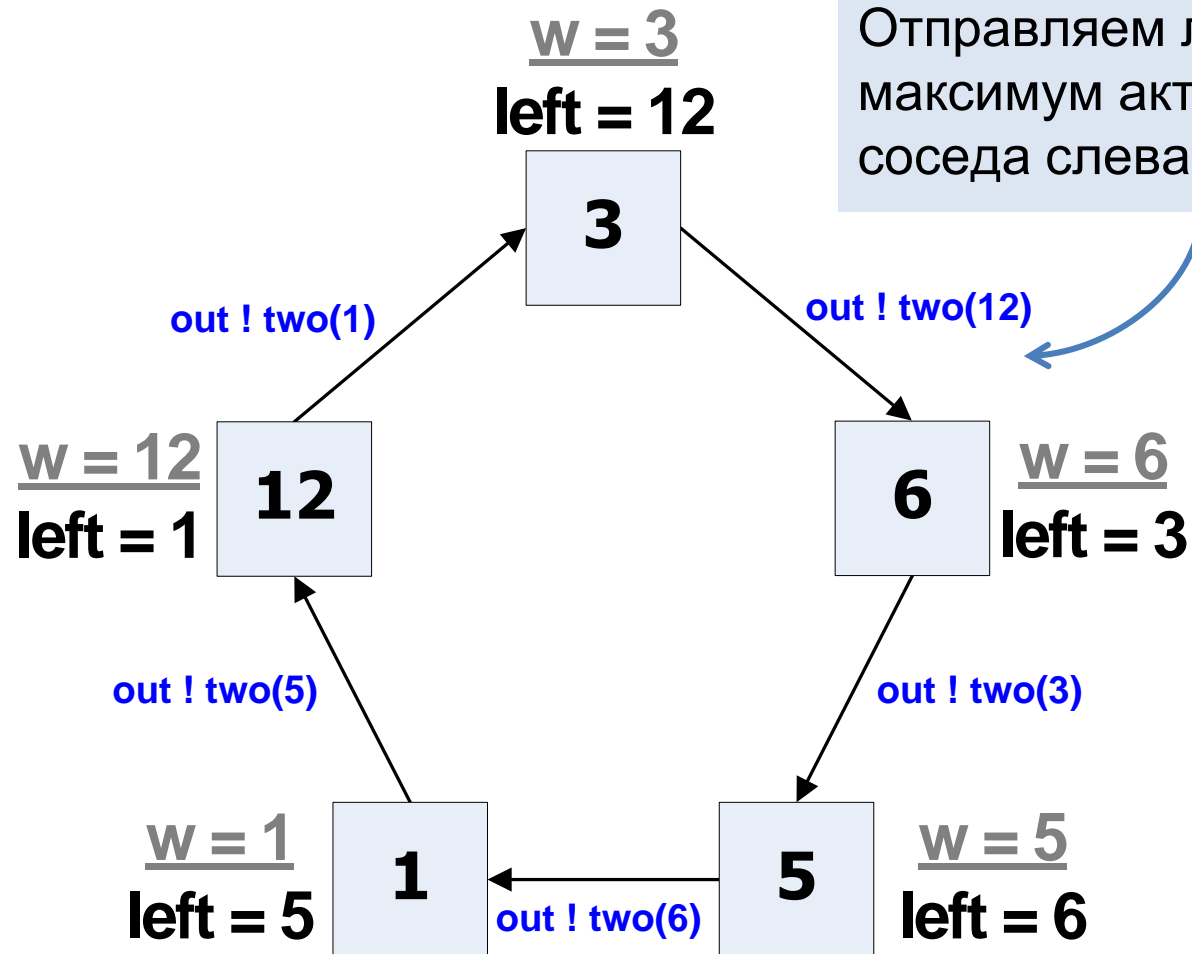


```
if (q != max) -> (left := q; send two(left))  
else победитель найден
```

Пример. Фаза A1

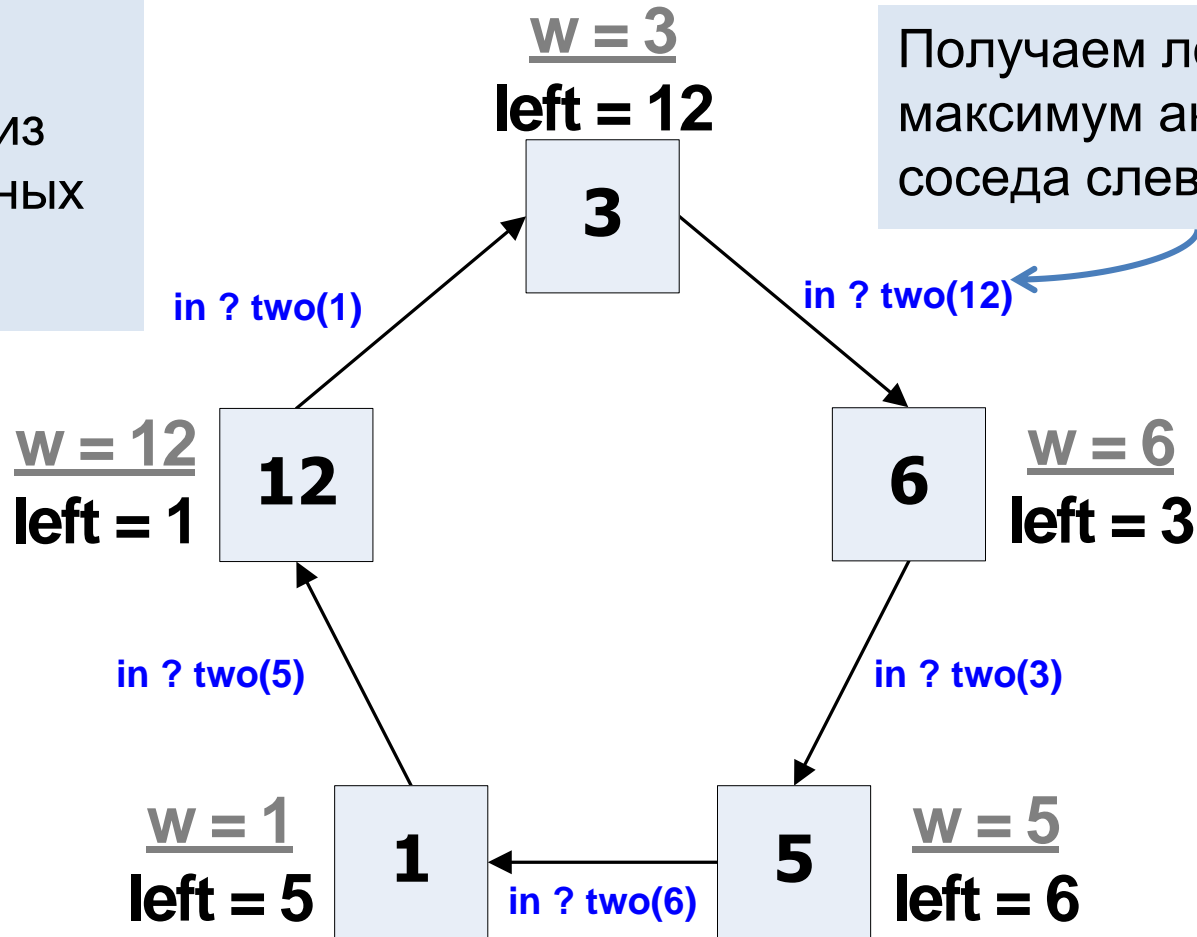


Пример. Фаза A1



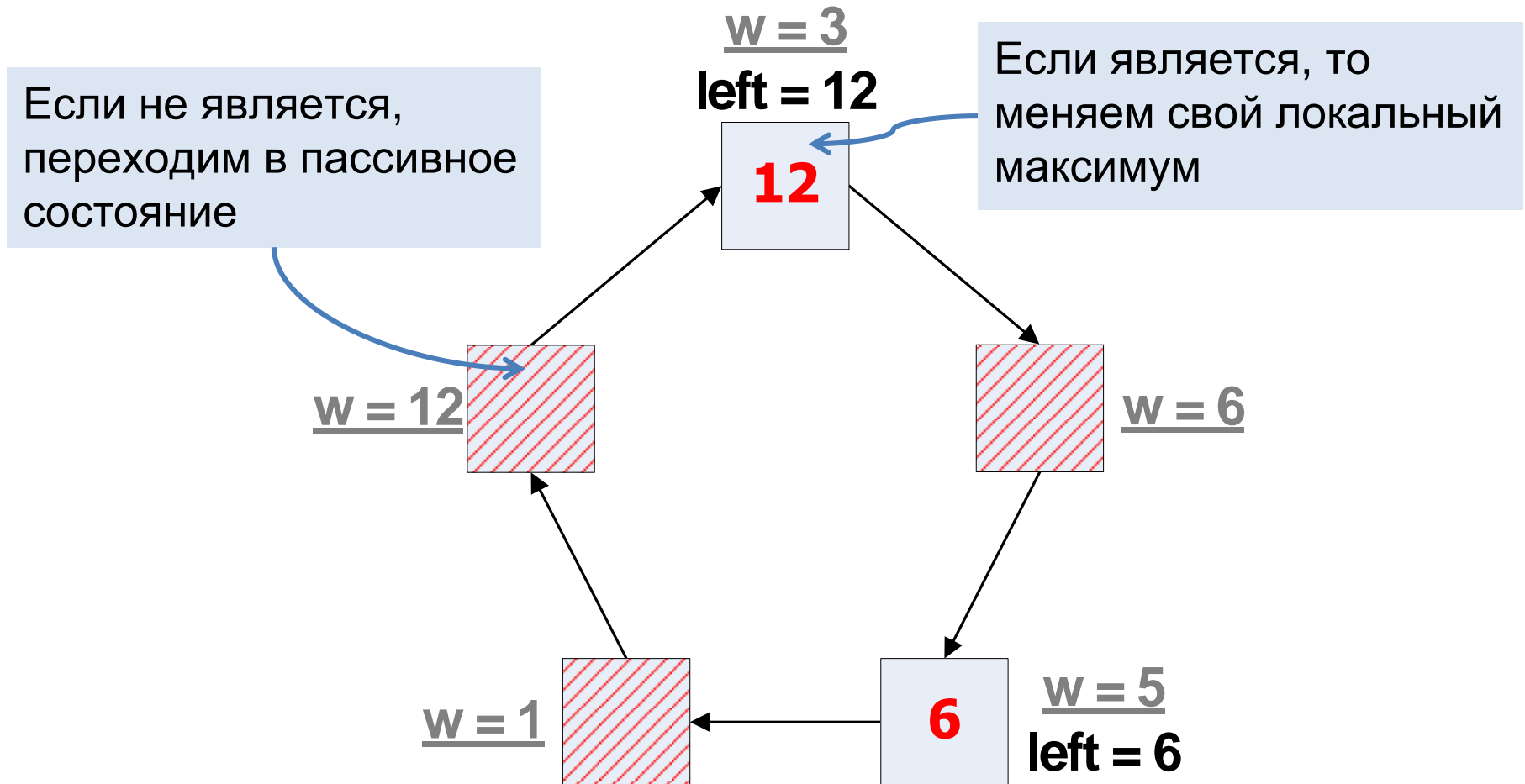
Проверяем: не является ли ближайший сосед слева носителем локального максимума из трех известных нам весов?

Пример. Фаза A2

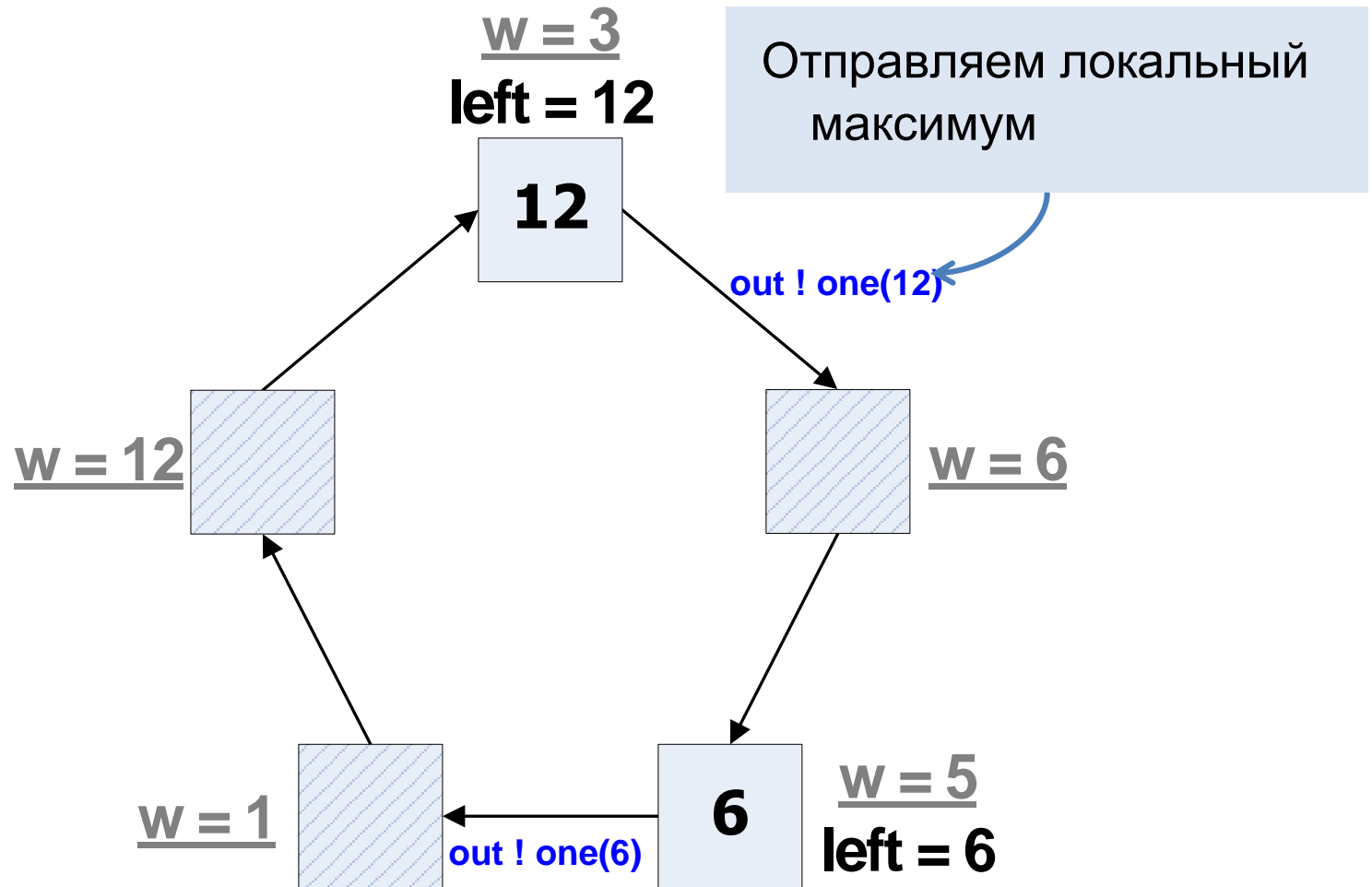


```
if((left>max) && (left>q)) -> (max:=left; send one(max))  
else узел стал пассивным
```

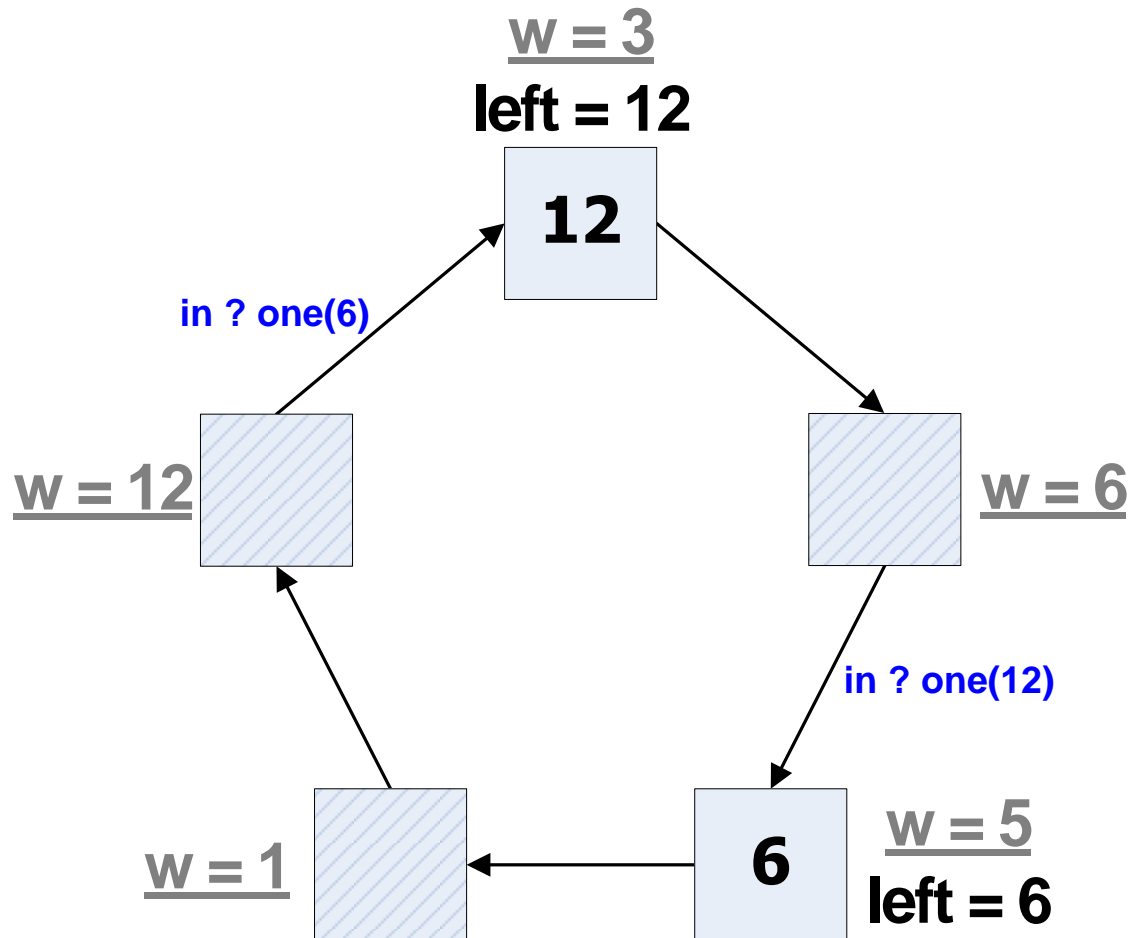
Пример. Фаза A2



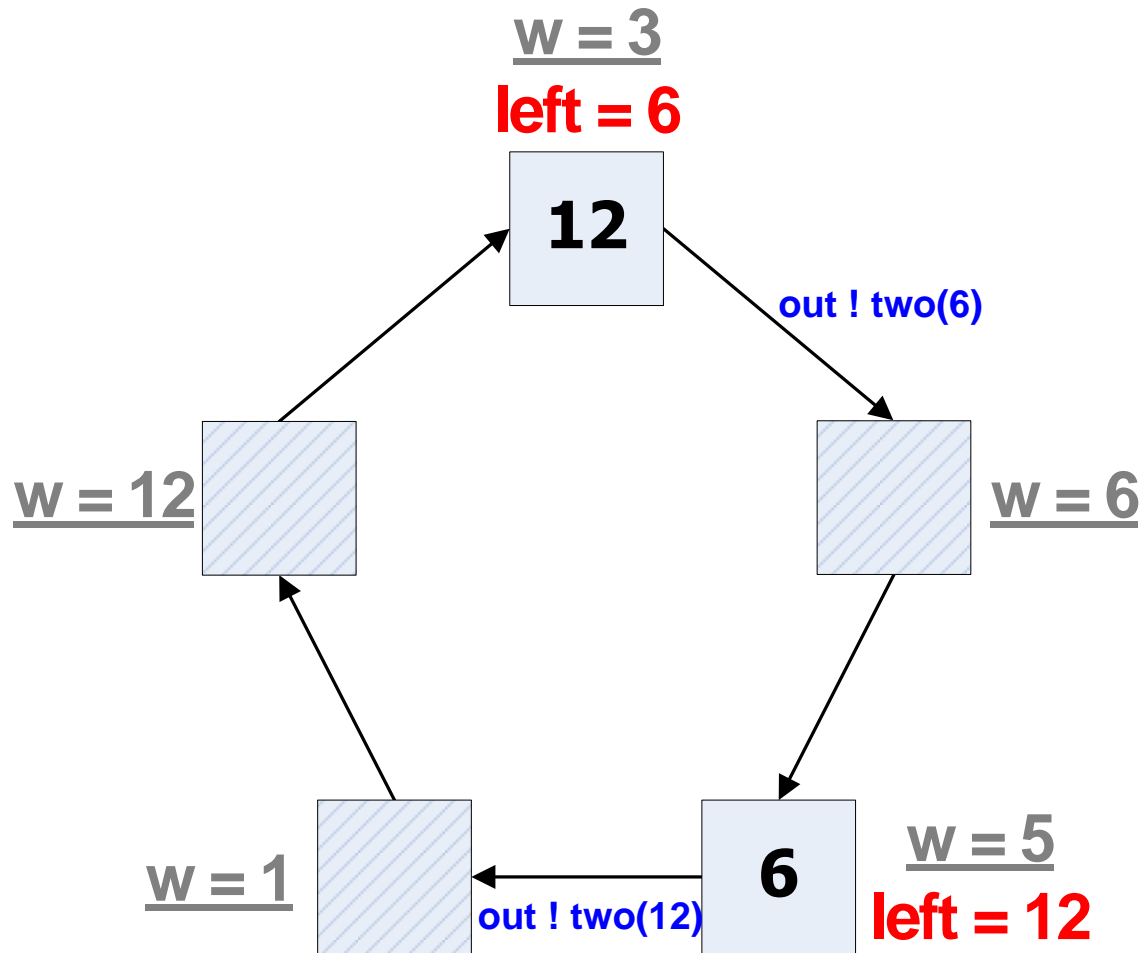
Пример. Фаза A2



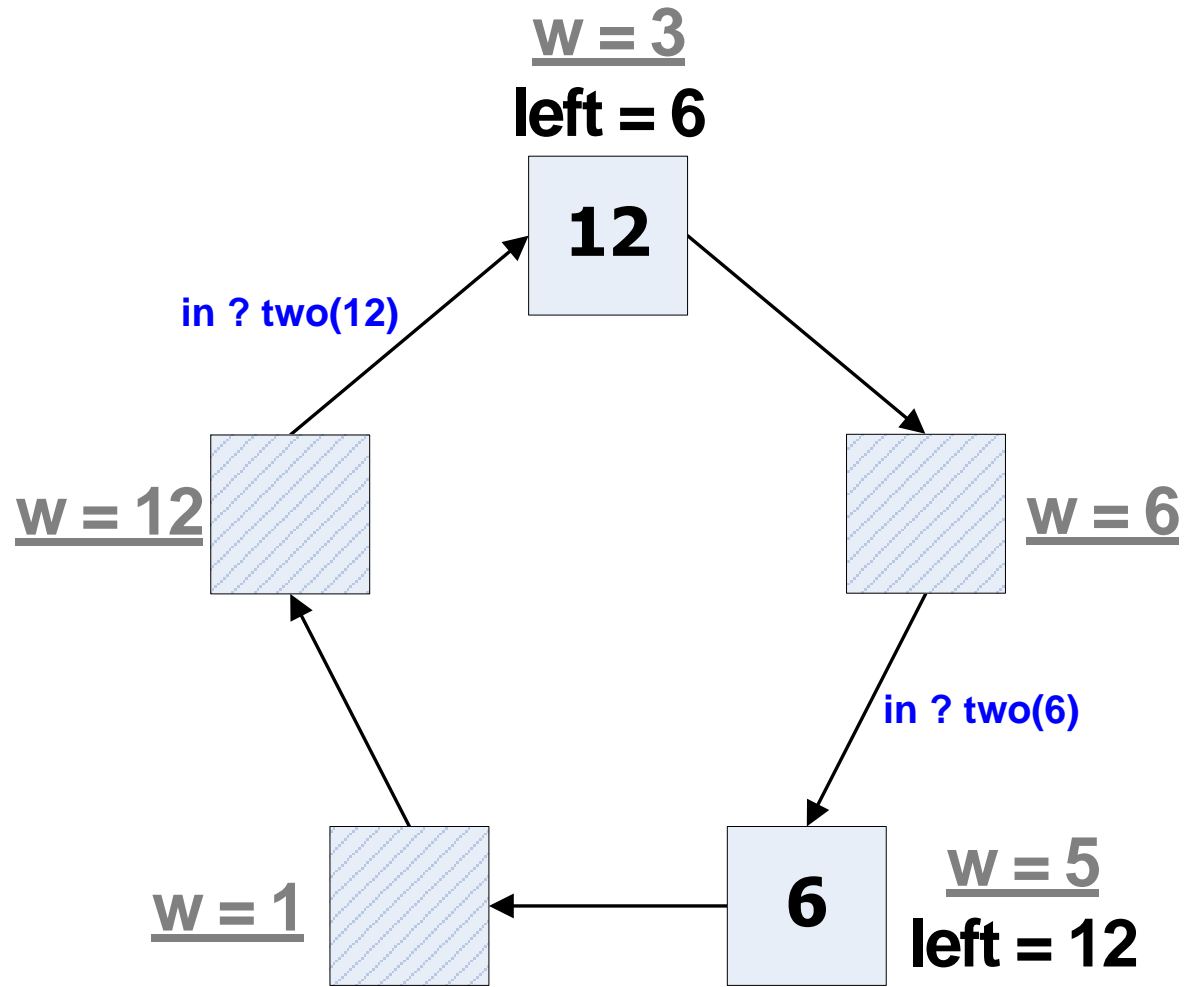
Пример. Фаза A1



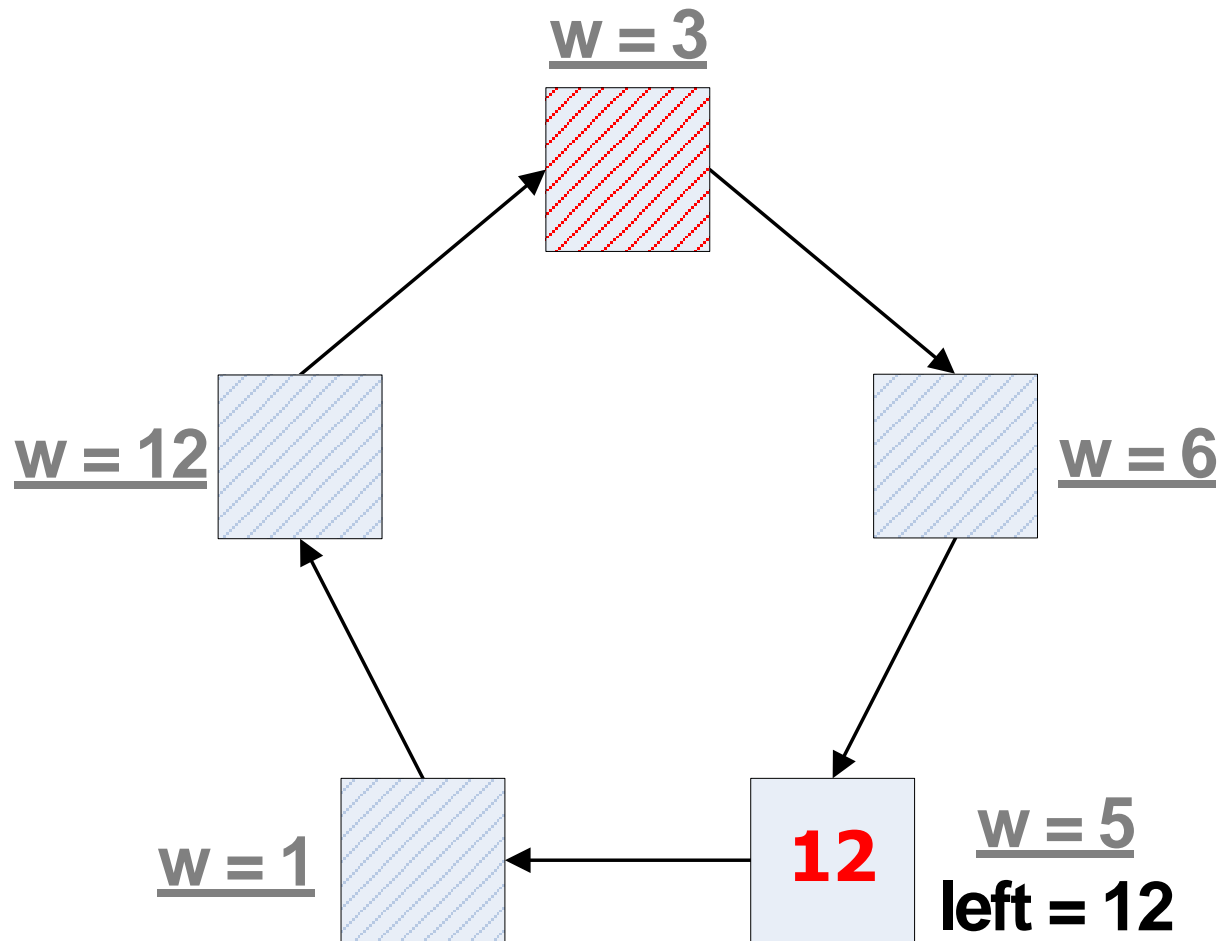
Пример. Фаза A1



Пример. Фаза A2

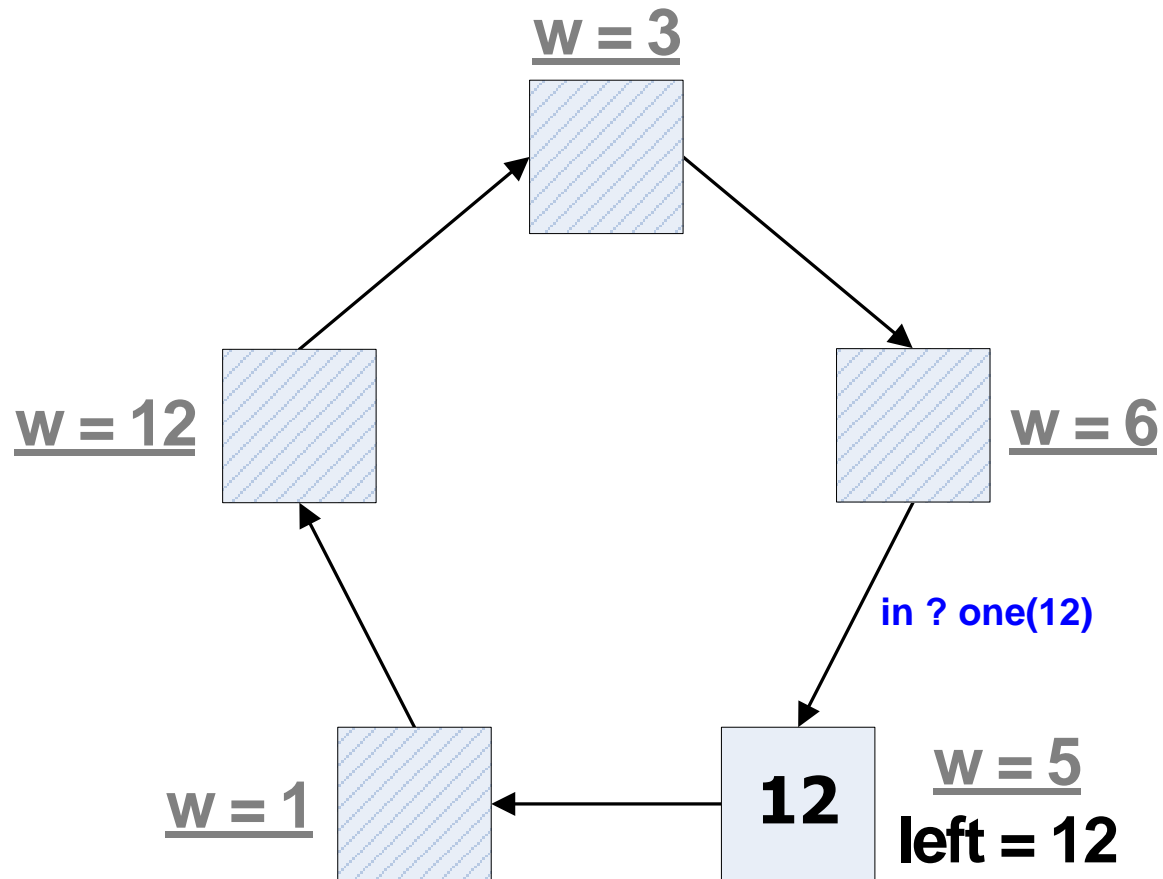


Пример. Фаза A2

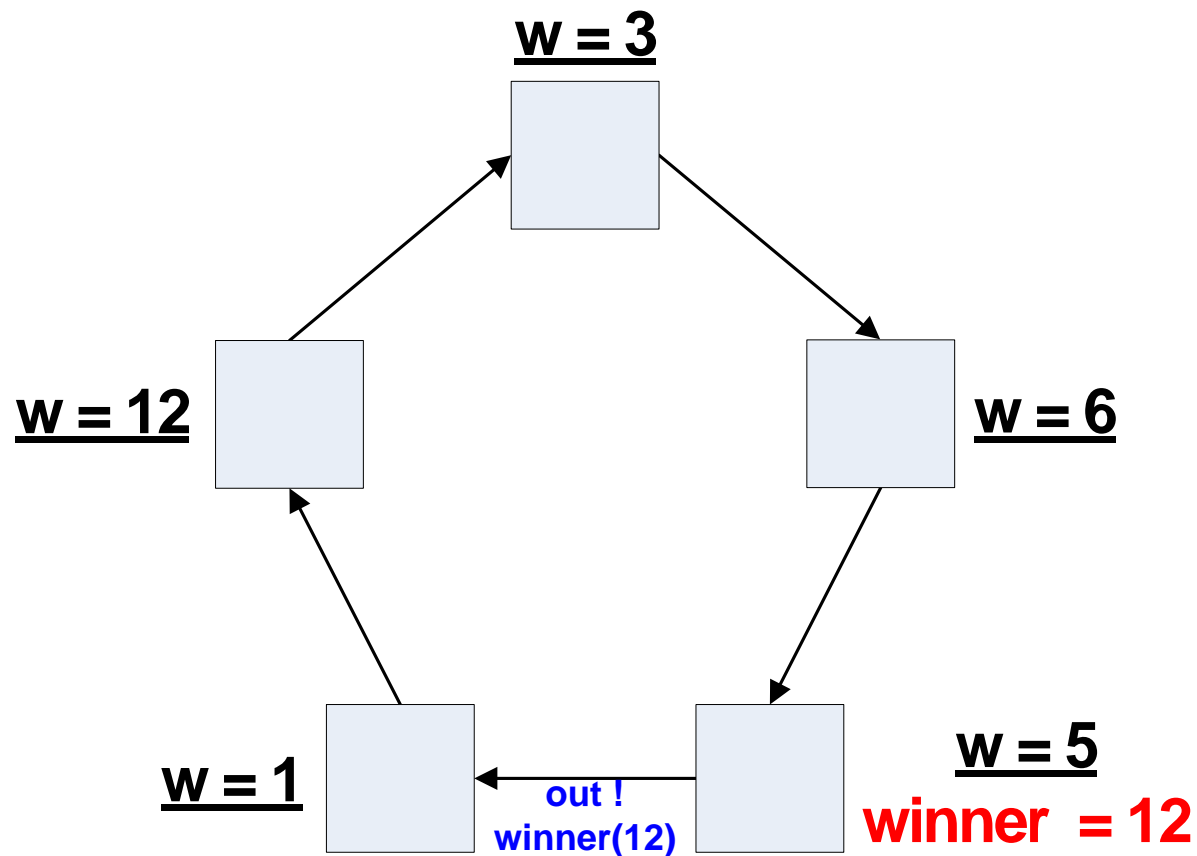




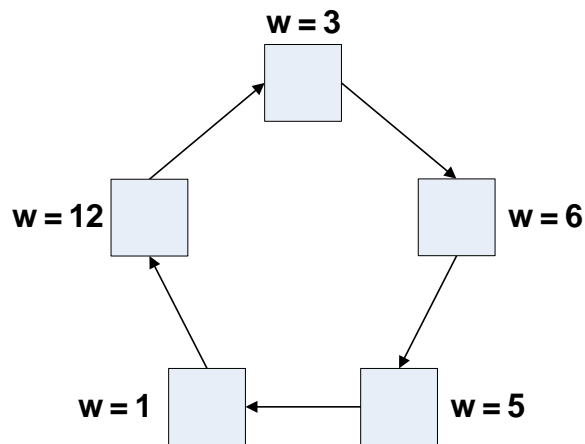
Пример. Фаза A1



Пример. Фаза A3. Лидер найден



Модель задачи о выборе лидера на языке Promela



Определим количество узлов в алгоритме для модели и не будем их менять во время его работы

```
# define N      5
```

количество узлов

Каждый узел – независимый процесс:

```
proctype node (...)
```

объявление процесса

Узлы обмениваются сообщениями трех типов:

```
mtype = {one, two, winner};
```

Узлы обмениваются сообщениями по каналам:

```
# define L      2
```

```
chan p[N] = [L] of {mtype, byte};
```

формат сообщения, например, `one(q)`
то же, что и `one, q`

массив каналов

емкость канала > 0 , значит, канал асинхронный

Модель задачи о выборе лидера на языке Promela

// Число процессов

define N 5

КОЛИЧЕСТВО УЗЛОВ

// Ограничение глубины канала

define L 2

// Типы сообщений

mtype = {one, two, winner};

// Объявление N каналов глубиной L

chan p[N] = [L] of {mtype, byte};

// Объявления процессов-узлов

proctype node (chan in, out; byte my_number) {

/* . . . */

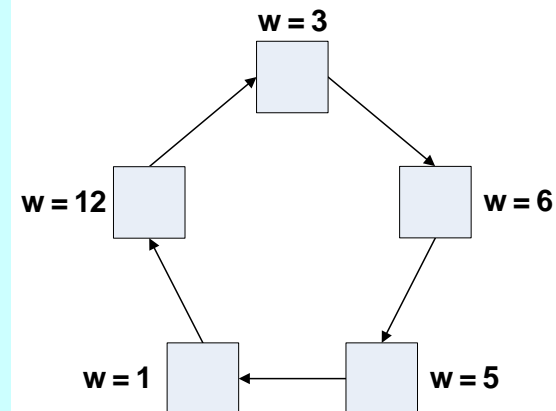
}

// Главный процесс, запуск всех процессов узлов

init {

/* . . . */

}



Каждый узел – независимый процесс

Описание алгоритма работы узла

```
proctype node (chan in, out; byte my_number)
{
    bit Active = 1,
        know_winner = 0; // флаг знаю-лидера
    byte q,
        max = my_number,
        left;
    out ! one(my_number); // A0. отправить свой параметр
    do
        :: in ? one(q)      -> /*...*/ // A1. получено сообщение one
        :: in ? two(q)      -> /*...*/ // A2. получено сообщение two
        :: in ? winner(q) -> /
            break;
    od
}
```

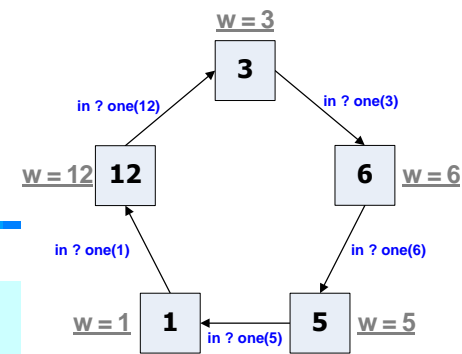
A0. Начальная фаза

A1. Получить сообщение one(q).
Обработать сообщение. Отправить сообщение two(q)

A2. Получить сообщение two(q).
Обработать сообщения. Отправить сообщения one(q)

A3. Обработка сообщения о лидере winner(q)

Фаза A1



```
:: in ? one(q) ->
```

```
if
```

```
:: Active ->
```

```
// узел в активном состоянии
```

```
if
```

```
:: q != max ->
```

```
// проверяем полученное значение
```

```
// если не равно лок. максимуму
```

```
left = q;
```

```
// то меняем параметр соседа
```

```
out ! two(q)
```

```
// передаем параметр далее
```

```
:: else ->
```

```
// иначе - нашли глоб. максимум
```

```
know_winner = 1;
```

```
// лидер этому узлу известен
```

```
out ! winner(q);
```

```
// сообщаем о выборе лидера
```

```
fi
```

```
:: else ->
```

```
//
```

```
out ! one(q)
```

```
//
```

```
fi
```

A1. Получить сообщение one(q):

1. Если $q \neq \max$, то $\text{left} := q$ и послать сообщение $\text{two}(\text{left})$

2. Иначе, \max является глобальным максимумом

Фаза A2

```
:: in ? two(q) ->

if
:: Active ->          // в активном состоянии

    if                // находимся за локальным максимумом
    :: left > q && left > max ->
        max = left;    // меняем информацию о локальном
                        // максимуме
        out ! one(max) // передаем дальше

    :: else ->         // переход в пассивное состояние
        Active = 0
fi

:: else -> // пассивны - передаем параметр без обработки
    out ! two(q)

fi
```

A2. Пришло сообщение two(q):

1. Если left больше и q, и max, то
max:=left и
послать сообщение one(max)
2. Иначе, узел становится пассивным.

Фаза A3. Обработка сообщения о лидере

```
:: in ? winner(q) -> // получено сообщение «лидер выбран»

if // проверка: совпадает ли номером узла
:: q != my_number ->
    printf("MSC: LOST\n"); // узел проиграл выборы

:: else -> // узел выиграл выборы
    printf("MSC: LEADER\n");

fi;

if // проверка: был ли найден лидер?
:: know_winner // знал и уже посылал сообщение –
               // «лидер выбран»
:: else -> out ! winner(q) // посылает сообщение
fi;

break
```

Некоторые операторы языка Promela всегда выполнимы.
Например, `break`, `skip`, `printf`

Основная функция, запускающая процессы

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

- Оператор run запускает процесс
- При помощи run можно передать процессу переменные
- Переменными могут быть каналы

По условию алгоритма требовался
одновременный запуск всех процессов

- ни active, ни run этого не обеспечивают

Конструкция atomic - предотвращение чередования

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

```
atomic {  
    оператор;  
    ...  
    оператор;  
}
```

Конструкции и переменные Promela только для верификации

```
// Число процессов
# define N      5

// Ограничение глубины канала
# define L      2

// Типы сообщений
mtype = {one, two, winner};

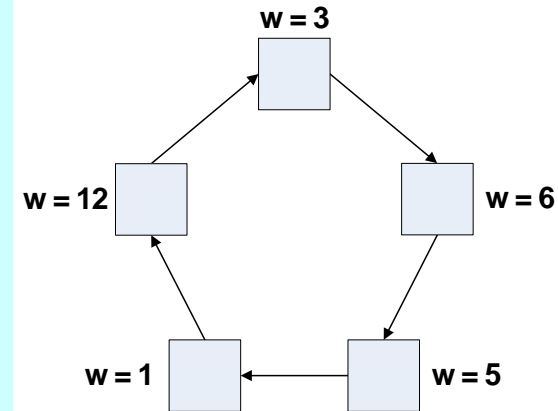
// Объявление N каналов глубиной L
chan p[N] = [L] of {mtype, byte};

// Количество лидеров
byte nr_leaders = 0;

// Объявления процессов-узлов
proctype node (chan in, out; byte my_number) {
  /* . . . */
}

// Главный процесс, запуск всех процессов узлов
init {
  /* . . . */
}
```

Глобальные переменные



Конструкции языка Promela, связанные с верификацией

- Утверждение **assert** (любое_булево_условие)
 - Если условие не всегда соблюдается, то оператор вызовет сообщение об ошибке в процессе симуляции и верификации с помощью Spin
- Метка конечного состояния (**end**)
 - указывает верификатору, чтобы тот не считал определенные операторы некорректным завершением программы
- Метка активного состояния (**progress**)
 - помечает операторы, которые для корректного продолжения работы протокола должны встретиться хотя бы раз на любой бесконечной трассе
 - не рекомендуется использовать

Фаза А3. Обработка сообщения о лидере

```
:: in ? winner(q) -> // получено сообщение «лидер выбран»

if // проверка: совпадает ли номером узла
:: q != my_number ->
    printf("MSC: LOST\n"); // узел проиграл выборы

:: else -> // узел выиграл выборы
    printf("MSC: LEADER\n");
    nr_leaders++;
    assert(nr_leaders == 1)
fi;

if // проверка: был ли найден лидер?
:: know_winner // знал и уже посылал сообщение –
                // «лидер выбран»
:: else -> out ! winner(q) // посылает сообщение
fi;

break
```

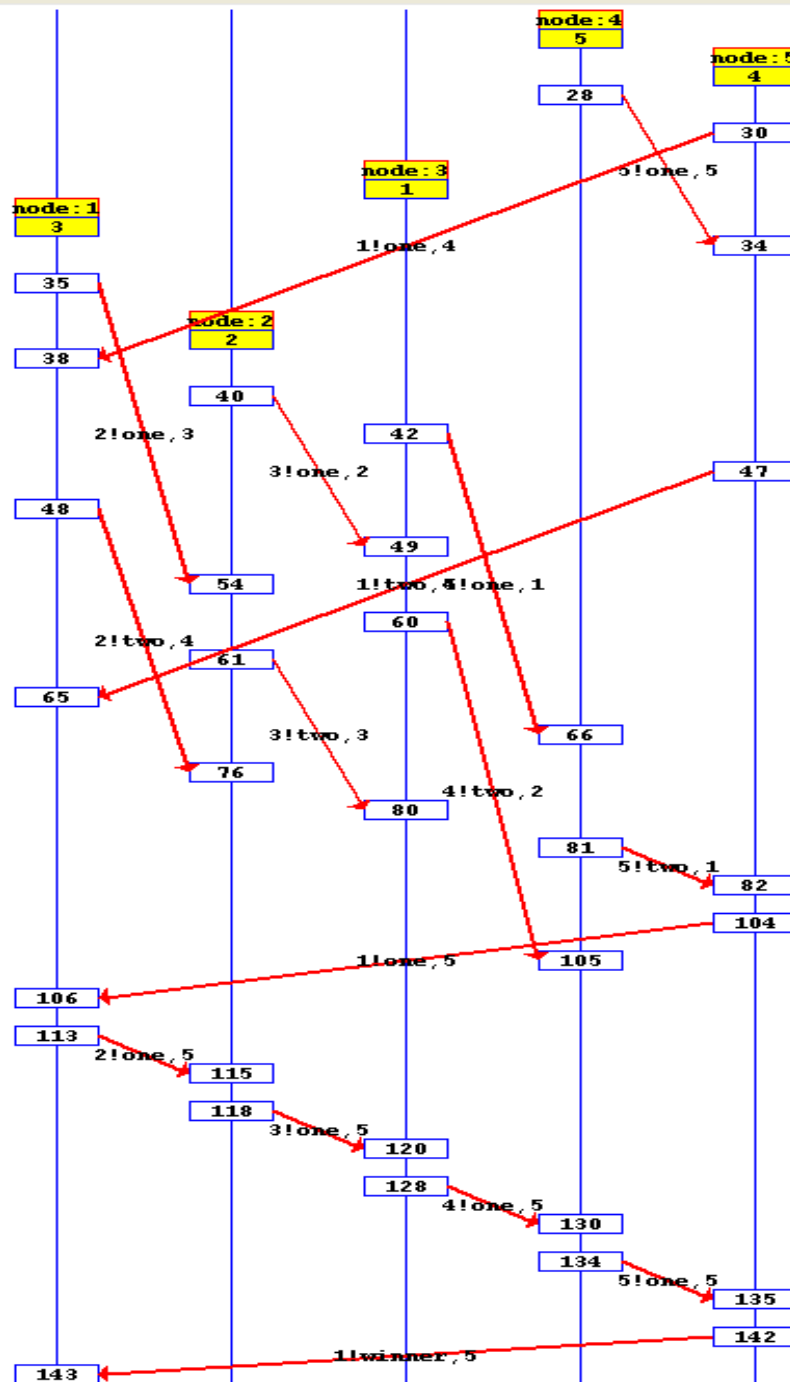
Описание алгоритма работы узла

```
proctype node (chan in, out; byte my_number)
{
    bit Active = 1,
        know_winner = 0; // флаг знаю-лидера
    byte q,
        max = my_number,
        left;
    out ! one(my_number); // A0. отправить свой параметр
end: do
    :: in ? one(q)      -> /*...*/ // A1. получено сообщение one

    :: in ? two(q)      -> /*...*/ // A2. получено сообщение two

    :: in ? winner(q) -> /*...*/ // A3. получено сообщение winner
        break;
    od
}
```

Конструкция end говорит, что
отсутствие выхода из цикла не
является ошибкой



Симуляция
программы
выбора лидера
в XSPIN

Глобальные свойства, обеспечивающие корректность алгоритма

- На любой фазе в кольце есть только один процесс с максимальным весом
- На любой фазе активный процесс характеризуется текущим весом, который является наибольшим из двух активных соседей слева

Эти два правила являются *глобальными инвариантами*

Формулировка глобальных инвариант и формальное доказательство требуют **понимания алгоритма**

Мы проверим корректность с помощью метода проверки модели, реализованного в SPIN

Требования к алгоритму выбора лидера

- лидер должен быть только один
noMore: $\text{nr_leaders} \leq 1$ //атомарное утверждение
G noMore
- лидер в конце концов будет выбран
elected: $\text{nr_leaders} == 1$ //атомарное утверждение
FG elected
- номер выбранного лидера всегда будет максимальным
nr == N //атомарное утверждение
FG nr

Linear Time Temporal Logic Formulae

Formula:

Load...

Operators:

Property holds for: ☒ All Executions (desired behavior) ☐ No Executions (error behavior)

Notes [file C:/cygwin/bin/spin/leader.ltl]:

Some other properties:
 ![] noLeader
 <> elected
 [] (noLeader U oneLeader)

Symbol Definitions:

```
#define elected      (nr_leaders > 0)
#define noLeader    (nr_leaders == 0)
#define oneLeader   (nr_leaders == 1)
```

Never Claim:

Generate

```
/*
 * Formula As Typed: <>[]oneLeader
 * The Never Claim Below Corresponds
 * To The Negated Formula !(<>[]oneLeader)
 * (formalizing violations of the original)
 */

never { /* !(<>[]oneLeader) */
  T0_init:
    if
      :: (! [(oneLeader)]) -> goto accept_S9
      :: (!1) -> goto T0_init
```

Verification Result: valid

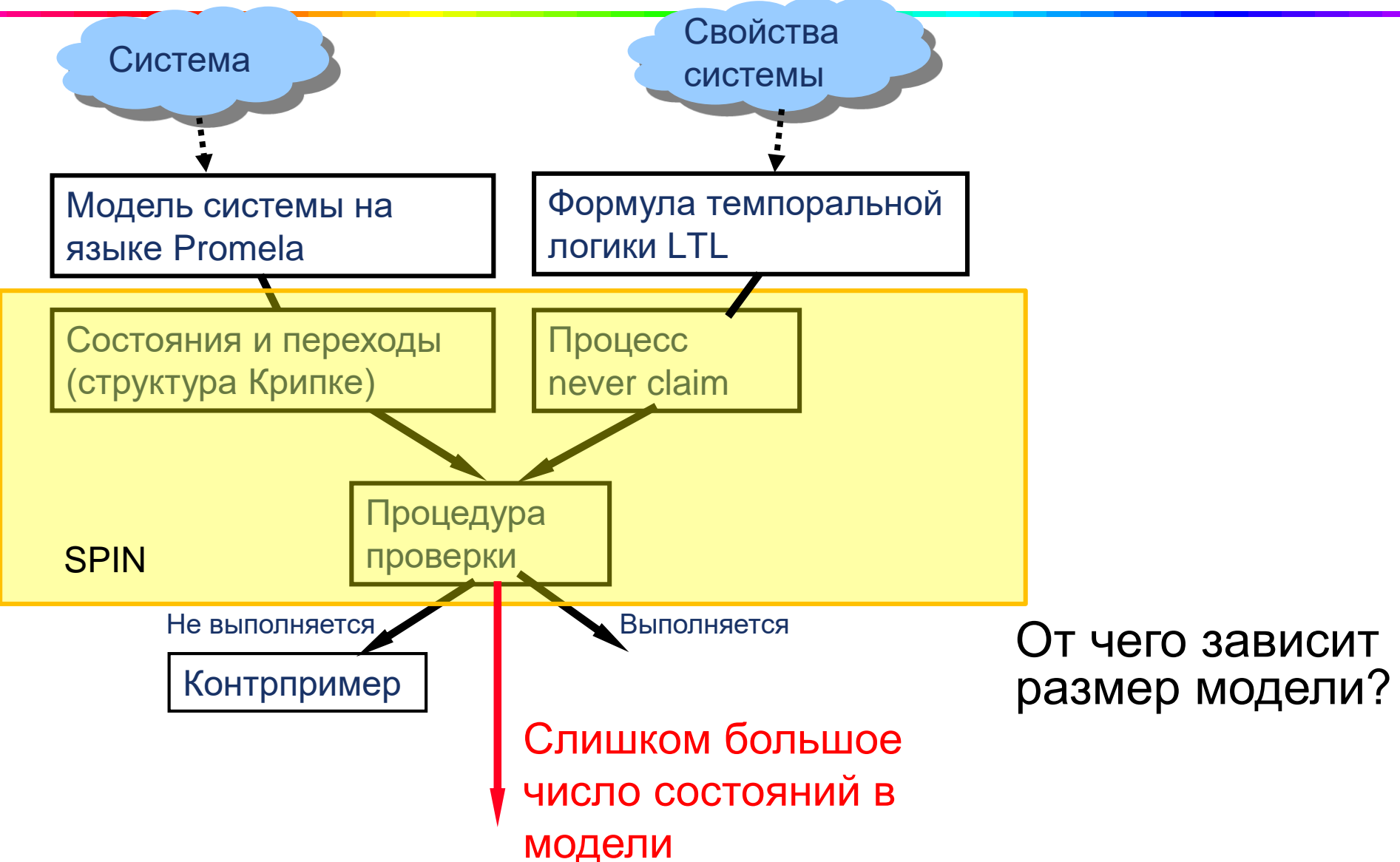
Run Verification

unreached in proctype node
 line 53, "pan.____", state 28, "out!two,nr"
 (1 of 49 states)
 unreached in proctype :init:
 (0 of 11 states)
 unreached in proctype :never:
 line 108, "pan.____", state 11, "-end-"
 (1 of 11 states)
 pan: elapsed time 0.006 seconds

Help Clear

Close Save As..

Практика: дополнительные исходы метода проверки модели



Контроль за числом состояний модели

Введем метки операторов в процессах

```
int x=0;
```

P1::

```
int t = 0, i = 0;
```

```
repeat 10 times {
```

```
p11: t = x;
```

```
p12: x = t+1;
```

```
p13: i = i+1
```

```
}
```

P2::

```
int t = 0, i = 0;
```

```
repeat 10 times {
```

```
p21: t = x;
```

```
p22: x = t+1;
```

```
p23: i = i+1
```

```
}
```

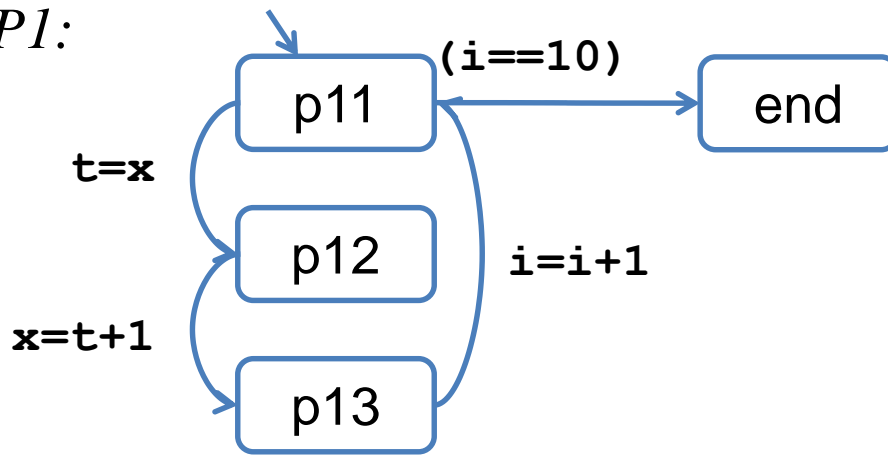
p11, p12, ..., p23 – метки операторов в процессах

Построим систему переходов, соответствующую одному из процессов

Оценка количества состояний всей модели

Композиция систем переходов каждого процесса

P1:



В каждом процессе:

- 3 метки
- Значение переменной t от 0 до 19
- Значение переменной i от 0 до 10

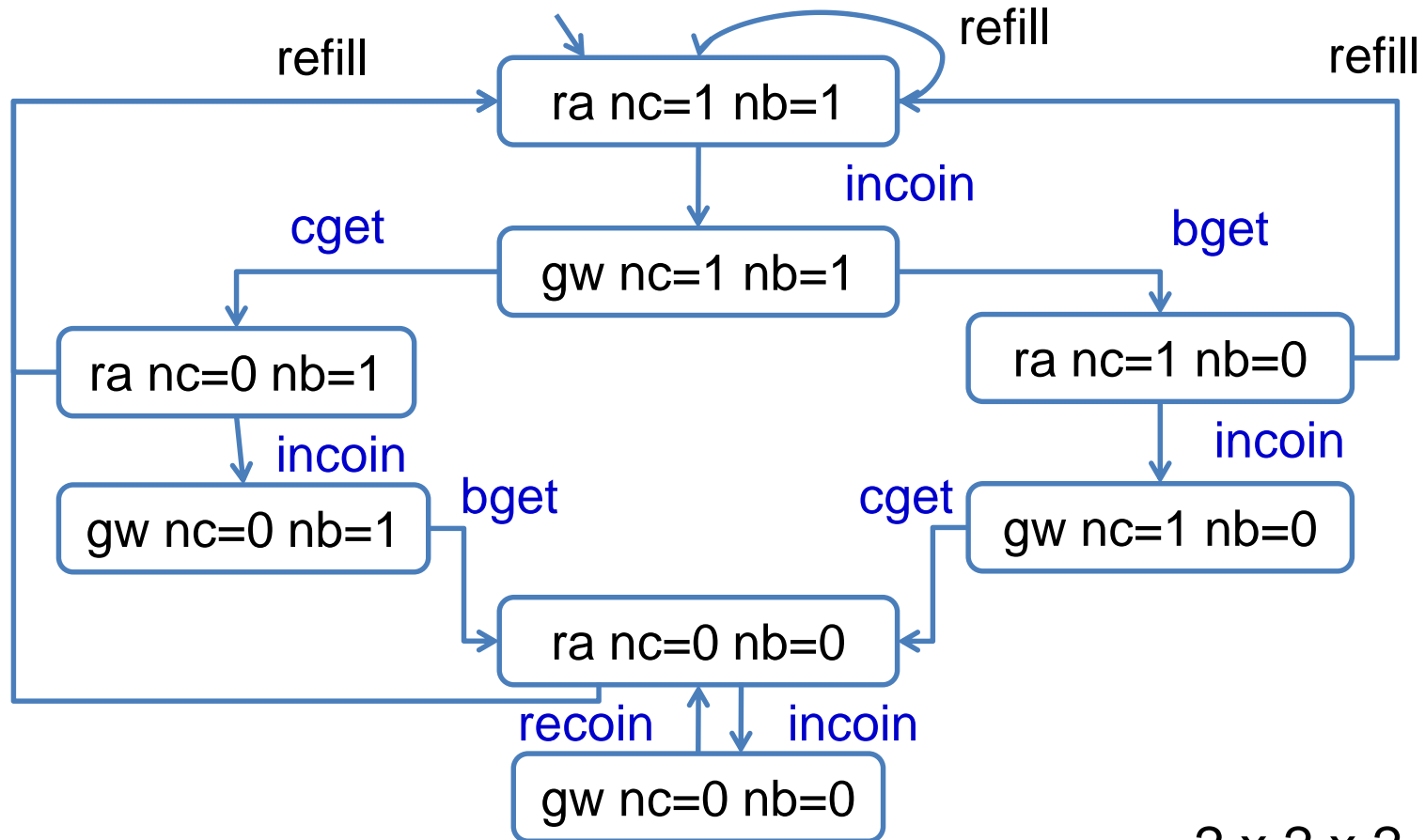
Глобальная переменная x принимает 21 значение

$$|P_1| \times |P_2| = (3 \times 20 \times 10)^2 \times 20 = 72 \times 10^5$$

Введение счетчиков увеличивает размер модели! Их нужно вводить, только если без них нельзя

Размер системы переходов с синхронным взаимодействием

Ограничим: $0 \leq nb \leq 1$ $0 \leq nc \leq 1$



$$2 \times 2 \times 2 = 8$$

- Синхронные каналы не увеличивают размер модели

Общая формула вычисления числа состояний асинхронной модели

Глобальное состояние системы:

- Значения текущих меток всех ее процессов
- Значения всех переменных каждого процесса
- Значения во всех асинхронных каналах

n • количество процессов

$Chan$ • множество асинхронных каналов

$Label_i$ • множество меток процесса i

Var_i • множество переменных процесса i

$dom(x)$ • область определения переменной x

$dom(c)$ • область определения канала c

$cap(c)$ • емкость канала c

$$\prod_{i=1}^n \left(|Label_i| \cdot \prod_{x \in Var_i} |dom(x)| \right) \cdot \prod_{c \in Chan} |dom(c)|^{cap(c)}$$

Promela – входной язык SPIN

Promela - язык моделирования взаимодействующих процессов

Каждая программа на Promela – модель, абстракция реальной системы

- включает конструкции для создания процессов и описания межпроцессного взаимодействия
- включает конструкции для верификации моделей

Promela не является языком реализации!

- в Promela отсутствует ряд средств, которые есть в языках программирования высокого уровня, например, таких:
 - указатели на данные
 - функции
 - не включено понятие времени или часов
 - отсутствуют операции с плавающей точкой

Общая характеристика Promela

- Модели в Promela *конечны*
- язык Promela имеет *формальную семантику*
- Основная единица структуры модели – *процесс*
- Процессы в Promela выполняются *асинхронно* (т.е. независимо друг от друга)
- выбор по условию, в операторе цикла происходит недетерминированно
- любая конструкция языка или **выполнимая**, или **блокирующая**
- Есть специальные конструкции для поддержки верификации
- Есть возможность доступа к скрытым переменным
- Promela – язык моделирования, а не реализации

Заключение по SPIN

- SPIN успешно используется для построения моделей распределенных алгоритмов и систем
- SPIN работает в режиме симуляции и верификации
- SPIN позволяет верифицировать свойства линейной темпоральной логики
- SPIN строит контрпример нарушения свойства, анализ которого позволяет выявить ошибку
- SPIN строит синхронную композицию модели и отрицания заданной формулы
 - проводит верификацию методами с сжатием без потери состояний и с потерей

СПАСИБО

Шошмина Ирина Владимировна
shoshmina_iv@spbstu.ru
