Верификация программ на моделях

Лекция №6

Свойства живучести в SPIN.

Спецификация и верификация свойств при помощи автоматов Бюхи.

Константин Савенков (лектор)

План лекции

- Проверка свойств живучести в Spin. Конструкции never
- Проверка свойств правильности
- Автоматы Бюхи
- Проверка свойств при помощи автоматов Бюхи

Способы описания свойств правильности (напоминание)

- 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

Конструкции **never** (отрицание свойств)

Never say never

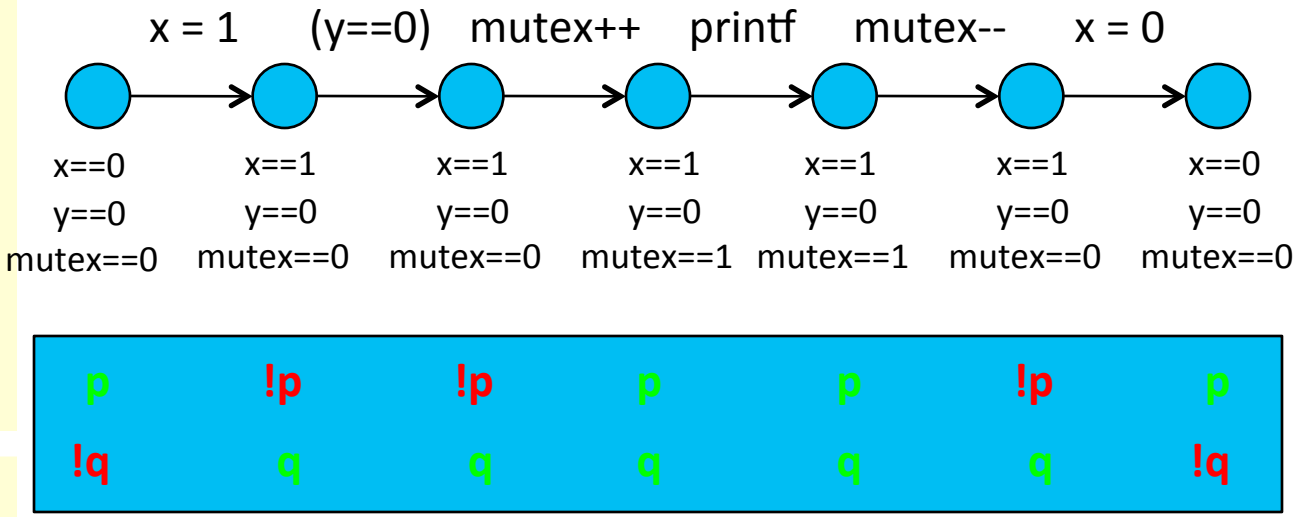
(народная пословица)

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

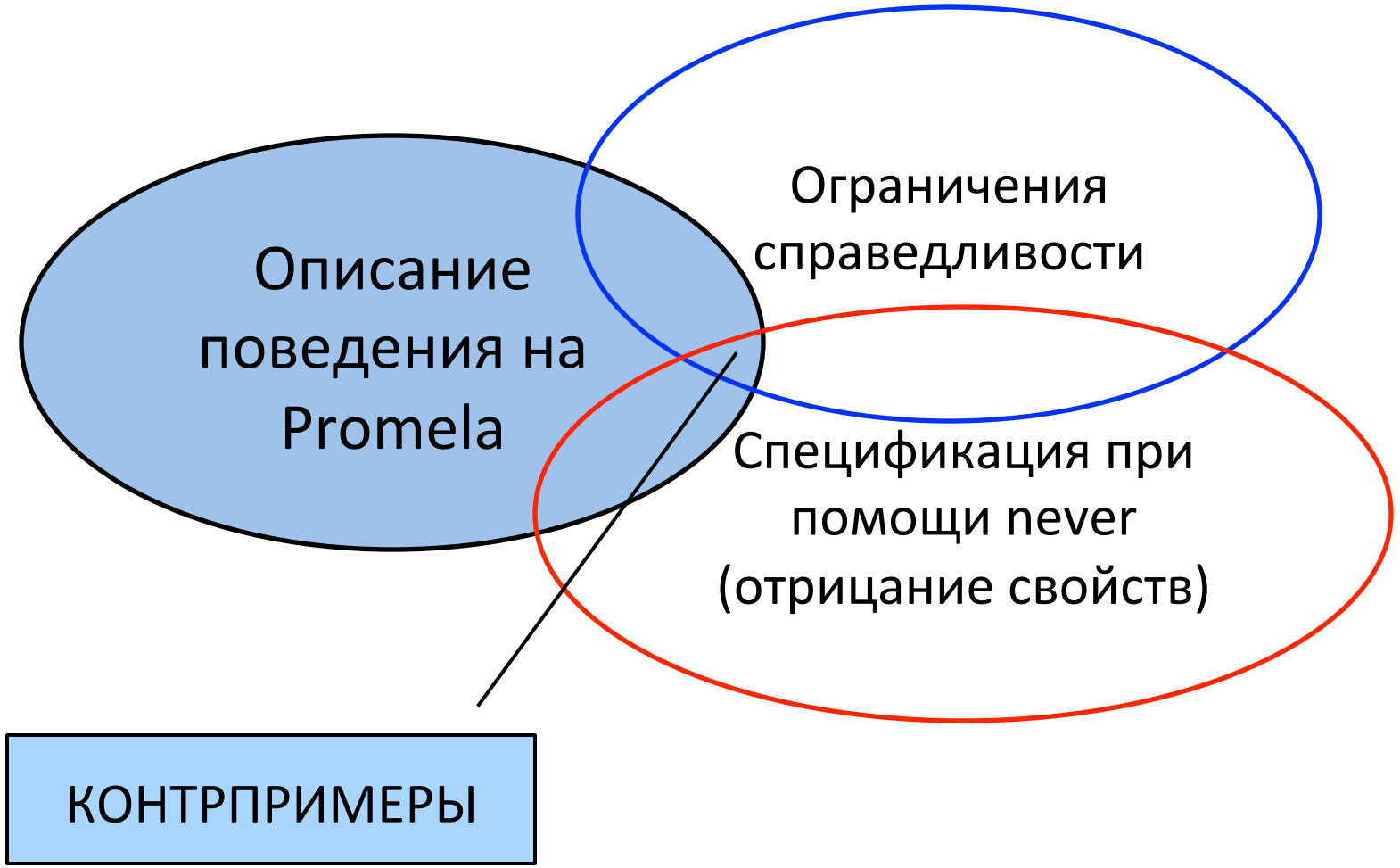
```
never
{
  do
  :: p -> break
  od
  do
  :: q -> break
  od
}
```

ПРАВИЛЬНО!

Конструкция never

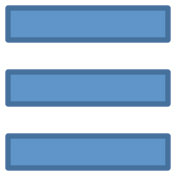
- может быть как детерминированной, так и нет;
- содержит **только** выражения без побочных эффектов (соотв. булевым высказываниям на состояниях);
- используются для описания **неправильного** поведения системы;
- прерывается при блокировании:
  - блокируется => наблюдаемое поведение не соответствует 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
}
```



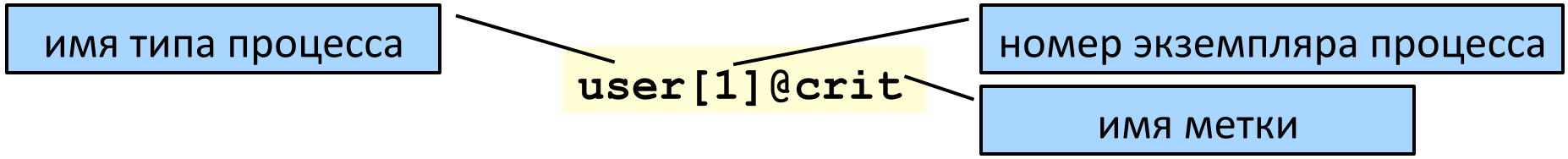
```
never
{
   do
   :: assert(invariant)
   od
}
```

```
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)

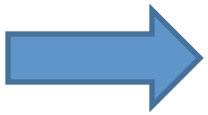
Используем метки управления вместо счётчика процессов

```
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 выполняются p или 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 приведёт 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)
```

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
```

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.

Автоматы Бюхи. Проверка свойств линейного времени.

Проверяемые свойства
(напоминание)

- Свойства моделей
  - Tr(M) – множество всех трасс модели,
  - φ – свойство правильности.
- Свойство **выполняется** на модели:

$$M \models \varphi \Leftrightarrow \forall \delta, \left(\left(\delta \in Tr(M) \right) \rightarrow \delta \models \varphi \right)$$

- Свойство нарушается на модели, если нарушается хотя бы на одной из трасс:

$$\neg \left(M \models \varphi \right) \Leftrightarrow \exists \delta, \left(\left(\delta \in Tr(M) \right) \wedge \neg \left(\delta \models \varphi \right) \right)$$

Отрицание свойств

(вспоминаем про двойственность)

- Доказательство нарушения свойства φ

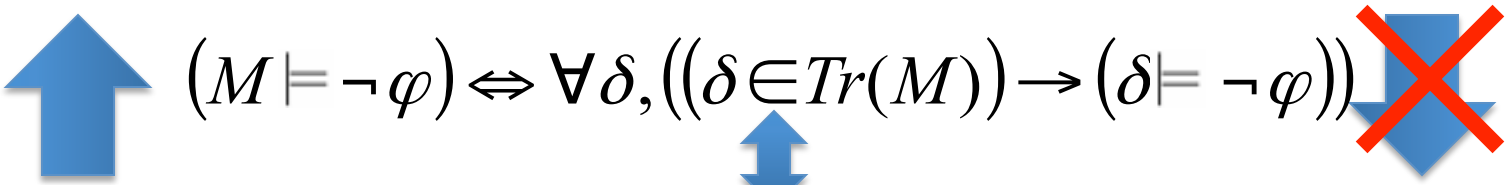
$$\neg\left(M \models \varphi\right) \Leftrightarrow \exists \delta, \left(\left(\delta \in Tr(M)\right) \wedge \neg\left(\delta \models \varphi\right)\right)$$

- Отличается от доказательства выполнения $\neg\phi$

$$\left(M \models \neg\varphi\right) \Leftrightarrow \forall \delta, \left(\left(\delta \in Tr(M)\right) \rightarrow \left(\delta \models \neg\varphi\right)\right)$$

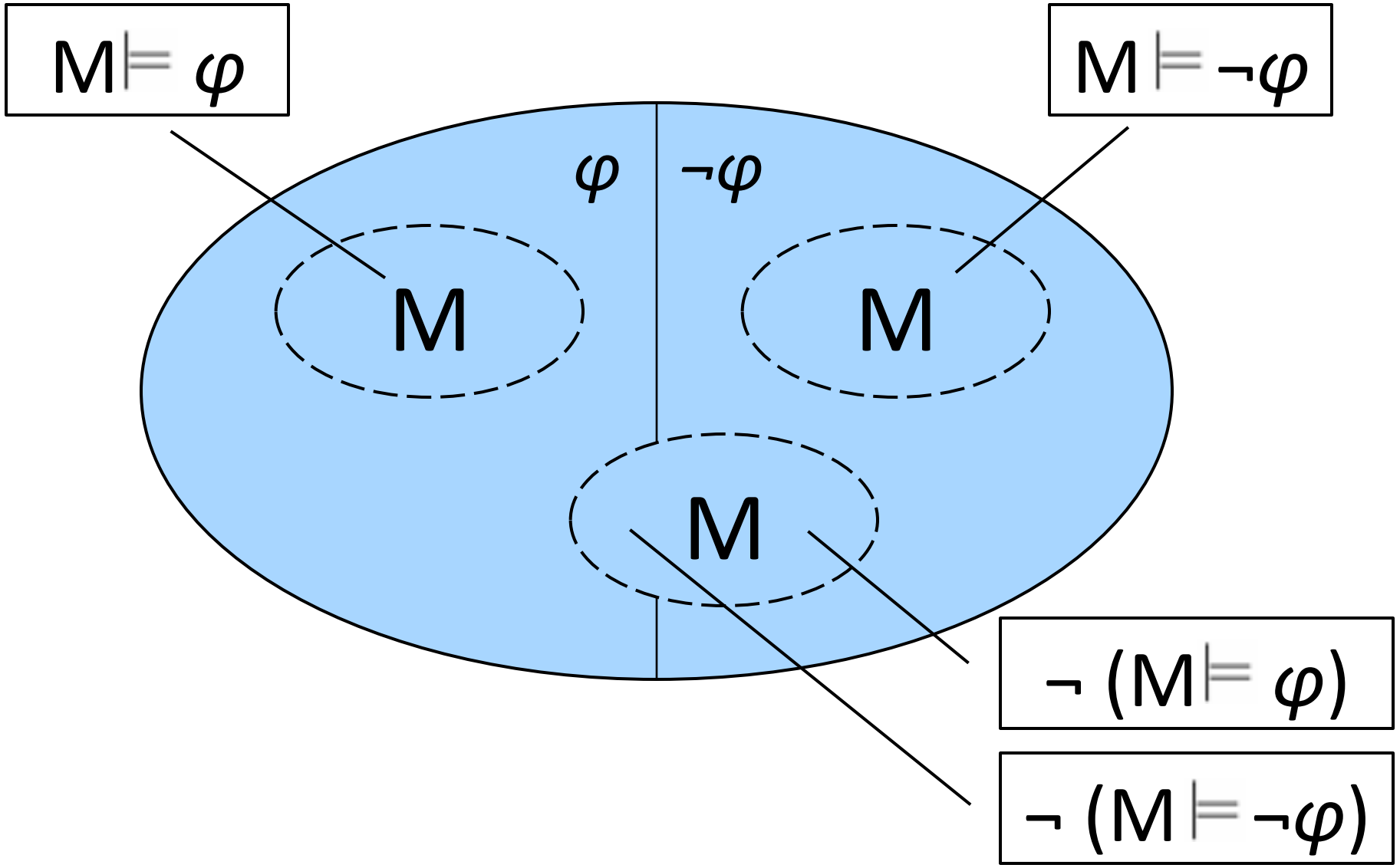
- **ПОЧЕМУ?**

$$\neg\left(M \models \varphi\right) \Leftrightarrow \exists \delta, \left(\left(\delta \in Tr(M)\right) \wedge \neg\left(\delta \models \varphi\right)\right)$$



$$\left(M \models \neg\varphi\right) \Leftrightarrow \forall \delta, \left(\left(\delta \in Tr(M)\right) \rightarrow \neg\left(\delta \models \varphi\right)\right)$$

Более наглядно



Пример

- Одновременное выполнение $\neg(M \models \varphi)$ и $\neg(M \models \neg\varphi)$

```
byte x = 0;

init {
  do
  :: x = 0
  :: x = 2
  od
}
```

```
never {
  do
  :: assert(x == 0)
  od
}
```

Нарушается выполнением x = 2

```
never {
  do
  :: assert(x != 0)
  od
}
```

Нарушается выполнением x = 0

Автоматы и логика

- Проще проверять нарушение свойства, чем его выполнение

  (достаточно найти один контрпример)

- Нарушение свойства описывается при помощи конструкции never – автомата, распознающего неправильное поведение

  (автоматы Бюхи)

- Свойства на последовательностях состояний удобно описывать при помощи темпоральной логики

  (Логика LTL)

Конечные автоматы

- Конечный автомат A задаётся сигнатурой

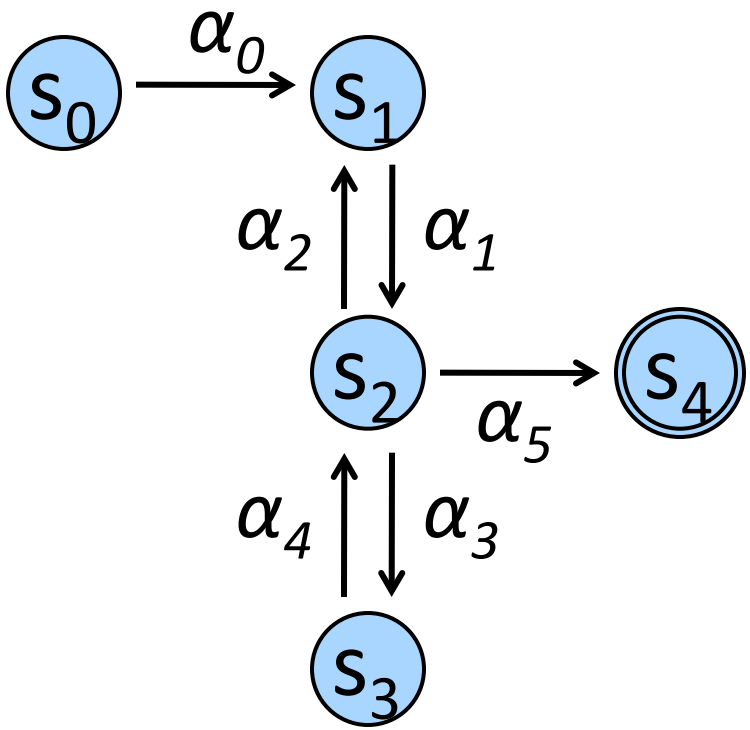
$$\langle S, s_0, L, F, T \rangle$$

где

- S – множество состояний,
- $s_0 \in S$ – начальное состояние,
- L – конечное множество меток (символов),
- $F \subseteq S$ – множество терминальных символов,
- $T \subseteq S \times L \times S$ – отношение перехода на состояниях.

Пример конечного автомата

$$A = \langle S, s_0, L, F, T \rangle$$



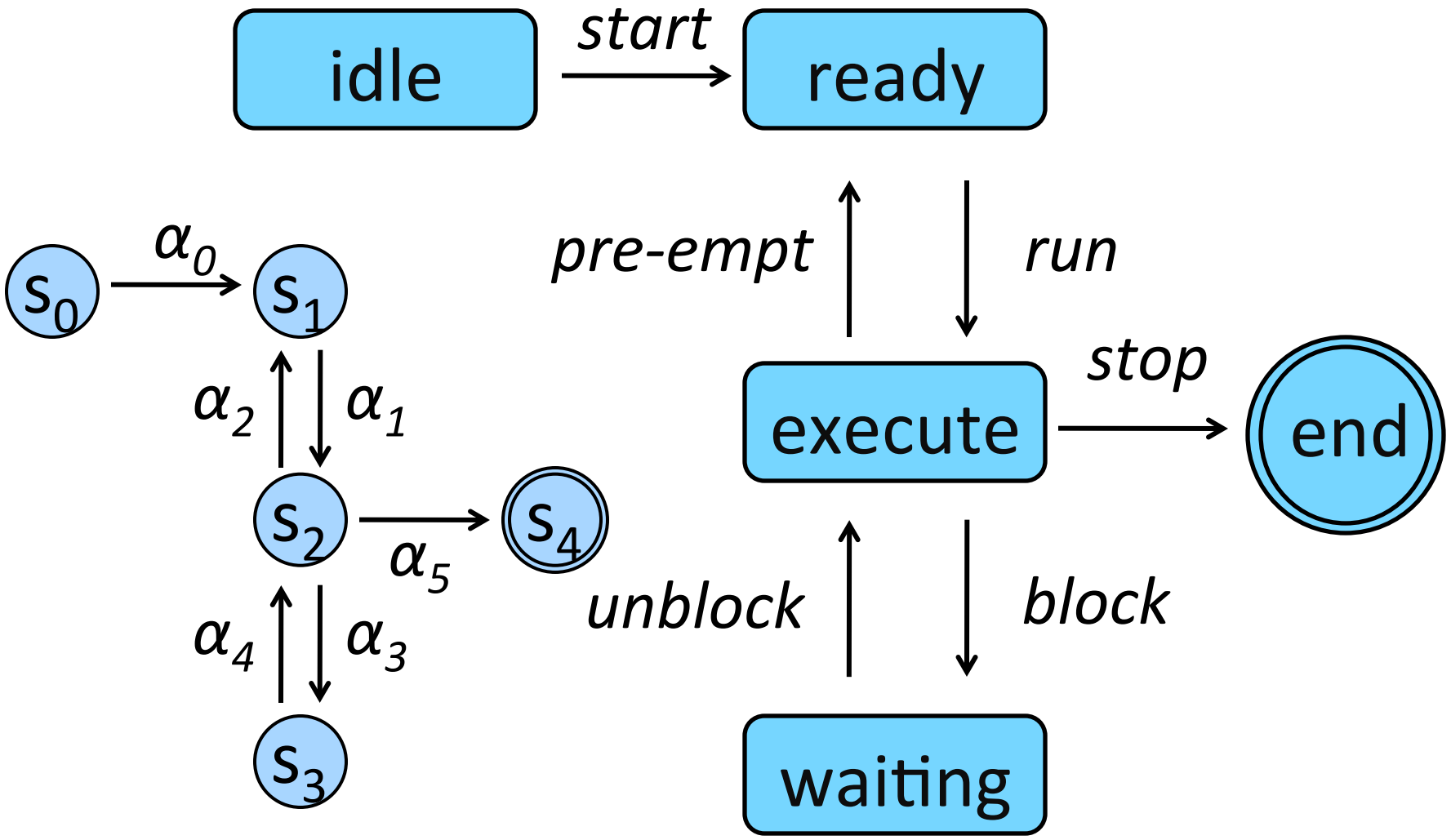
$$S = \{s_0, s_1, s_2, s_3, s_4\}$$

$$L = \{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$$

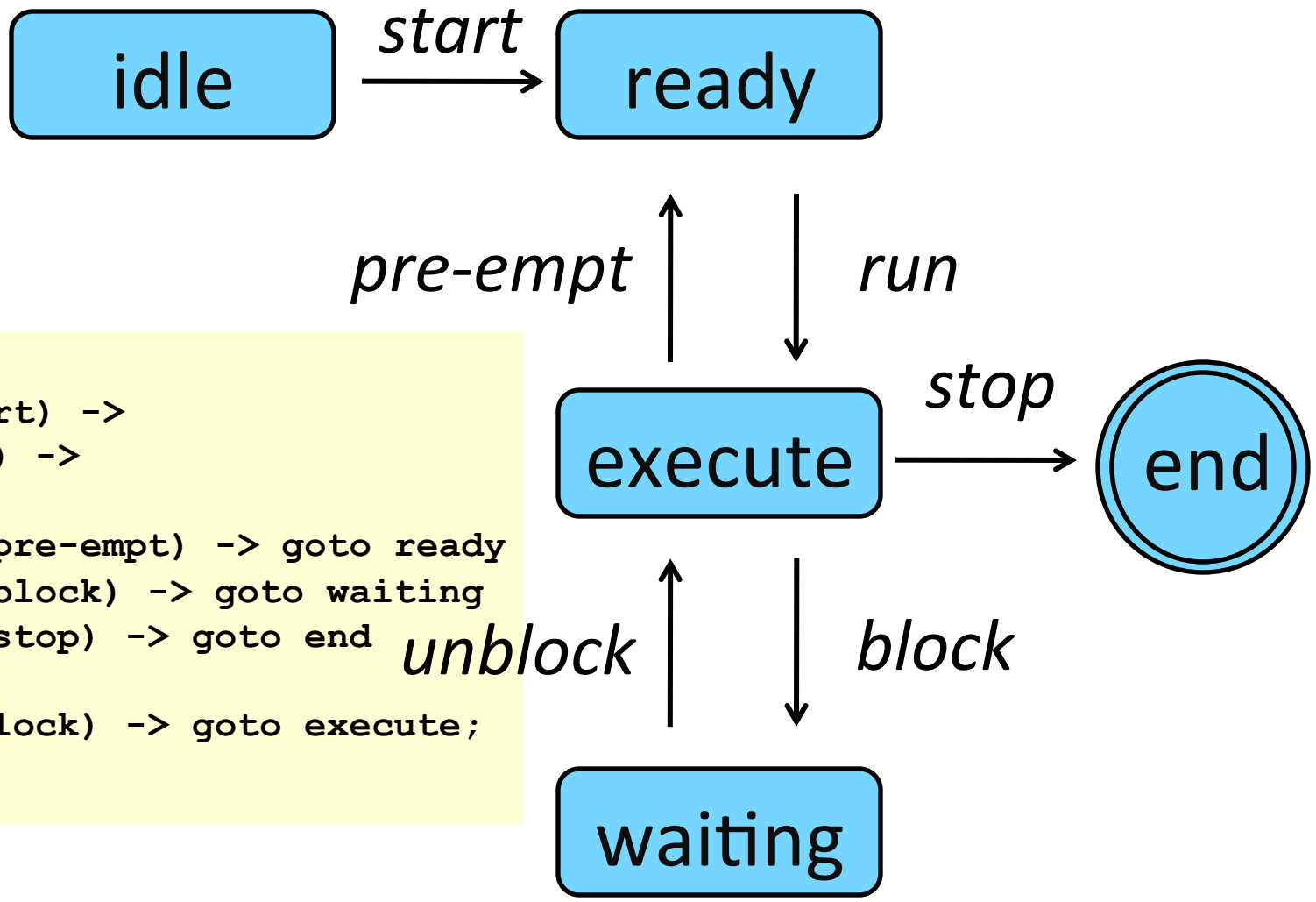
$$F = \{s_4\}$$

$$T = \{(s_0, \alpha_0, s_1), (s_1, \alpha_1, s_2), \ldots\}$$

Вариант интерпретации
(планировщик процессов)



Записываем в виде never




```
never {
idle:     (start) ->
ready:    (run) ->
execute:  if
          :: (pre-empt) -> goto ready
          :: (block) -> goto waiting
          :: (stop) -> goto end
          fi;
waiting:  (unblock) -> goto execute;
end:      skip
}
```

Детерминизм и недетерминизм

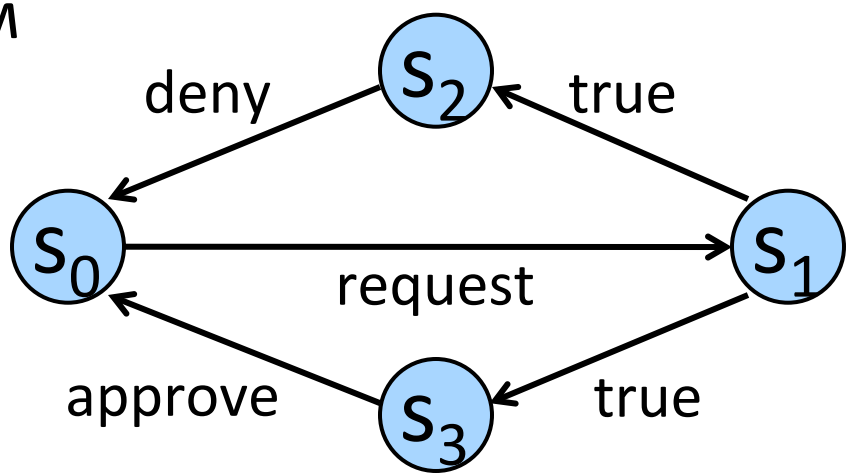
- Конечный автомат $A = \langle S, s_0, L, F, T \rangle$

называется детерминированным, только если

$$\forall s, \forall l, \left(\left((s, l, s') \in T \wedge (s, l, s'') \in T \right) \rightarrow s' \equiv s'' \right)$$

  - т.е. целевое состояние перехода однозначно определяется исходным состоянием и меткой
  - в противном случае автомат называется недетерминированным

Модель сервера запросов, работающего в бесконечном цикле



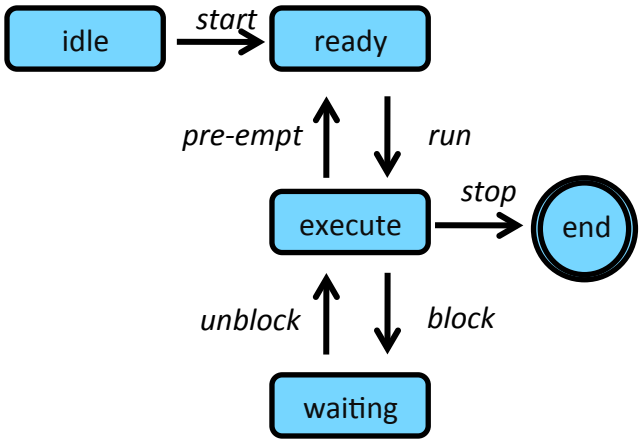
Определение *прохода*

- Проходом конечного автомата $\langle S, s_0, L, F, T \rangle$ называется такое **упорядоченное** и, возможно, бесконечное множество переходов из T:

$$\sigma = \langle (s_0, l_0, s_1), (s_1, l_1, s_2), (s_2, l_2, s_3), \ldots \rangle$$

что $\forall i, i \geq 0 : (s_i, l_i, s_{i+1}) \in T$.

- Проход соответствует последовательности состояний из S и слову в алфавите L.

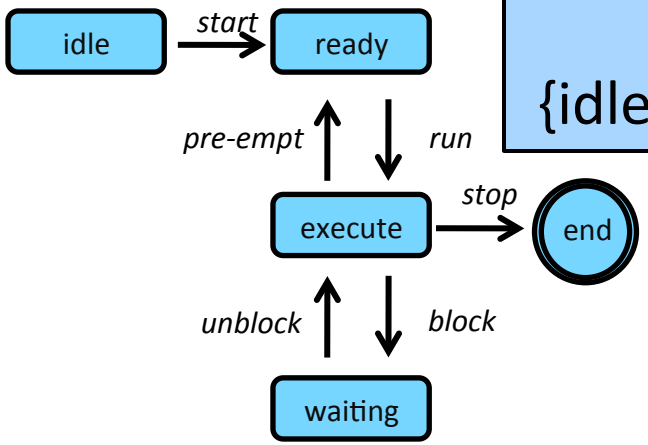


Последовательность состояний:
{idle, ready, {execute, waiting}*}

Соответствующее слово в L:
{start, run, {block, unblock}*}

Допускающий проход

- Допускающим проходом конечного автомата A называется конечный проход σ, финальный переход которого (s_{n-1}, l_{n-1}, s_n) ведёт в терминальное состояние

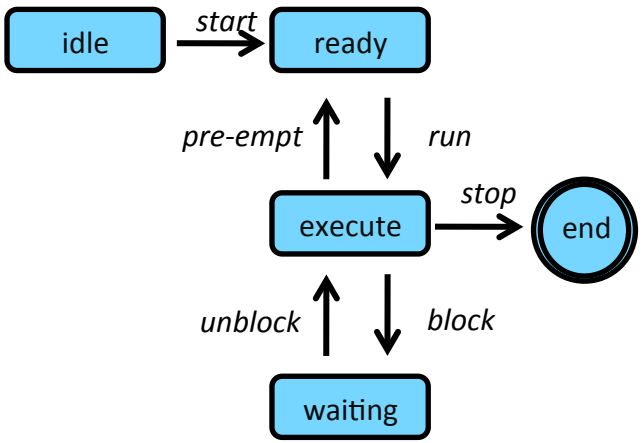


Последовательность состояний
для допускающего прохода:
{idle, ready, execute, waiting, execute, end}

Соответствующее слово в L:
{start, run, block, unblock, stop}

Язык автомата

- Языком автомата A называется множество слов в алфавите L, соответствующих допускающим проходам автомата A



Язык автомата:

```
{
   start,
   run,
   {{pre-empt,run}+
     {block,unblock}*}*,
   stop
}
```

Самое короткое слово языка:
{start, run, stop}

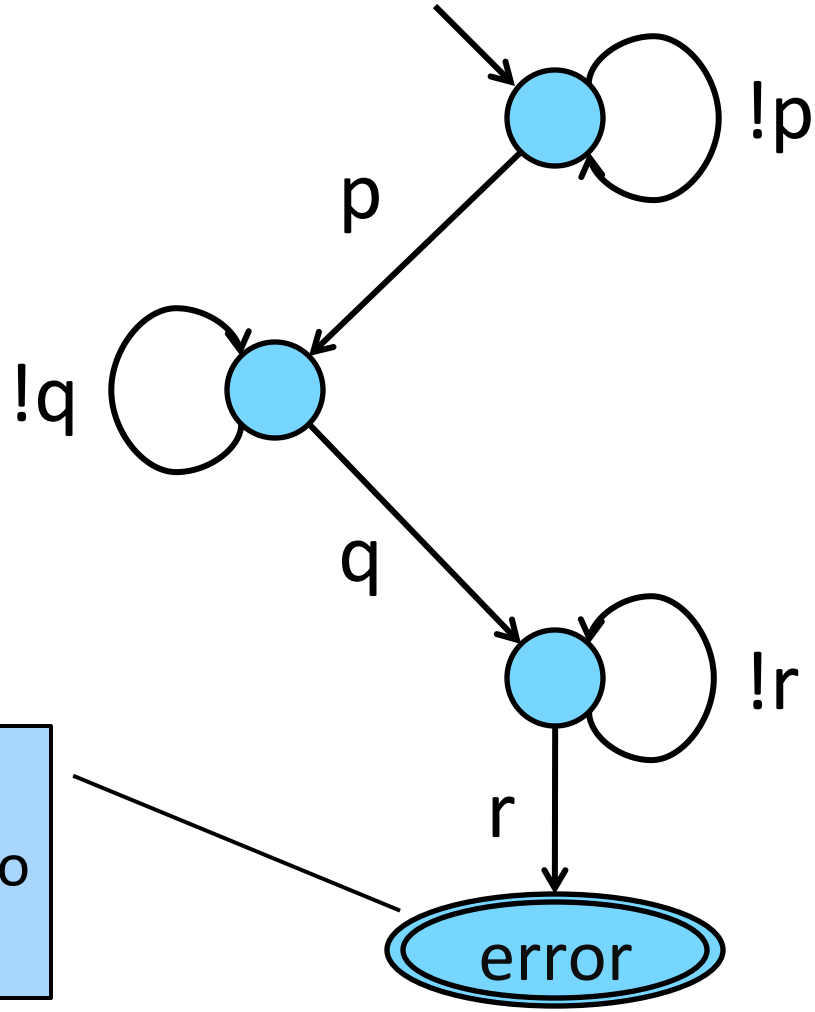
Описание свойств при помощи автомата

Пример свойства:

если сначала p=T,

а позже q=T,

то впоследствии r=F

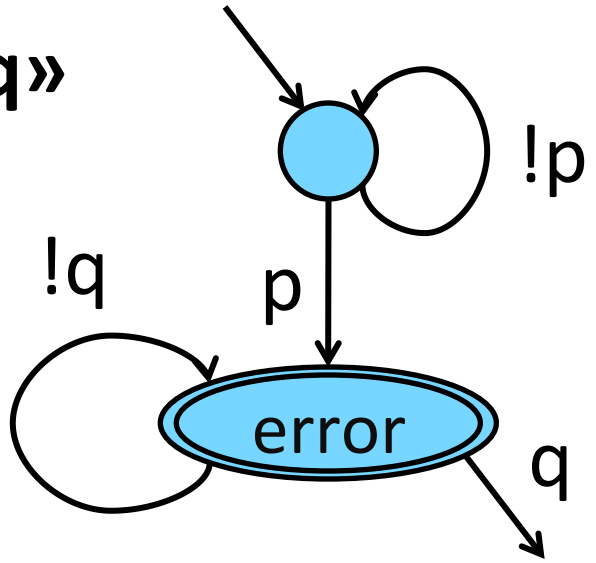


Если мы попали в терминальное состояние, то свойство нарушается

Иногда нужно рассуждать о потенциально бесконечной задержке

Классическое свойство живучести:

«если p, тогда впоследствии q»



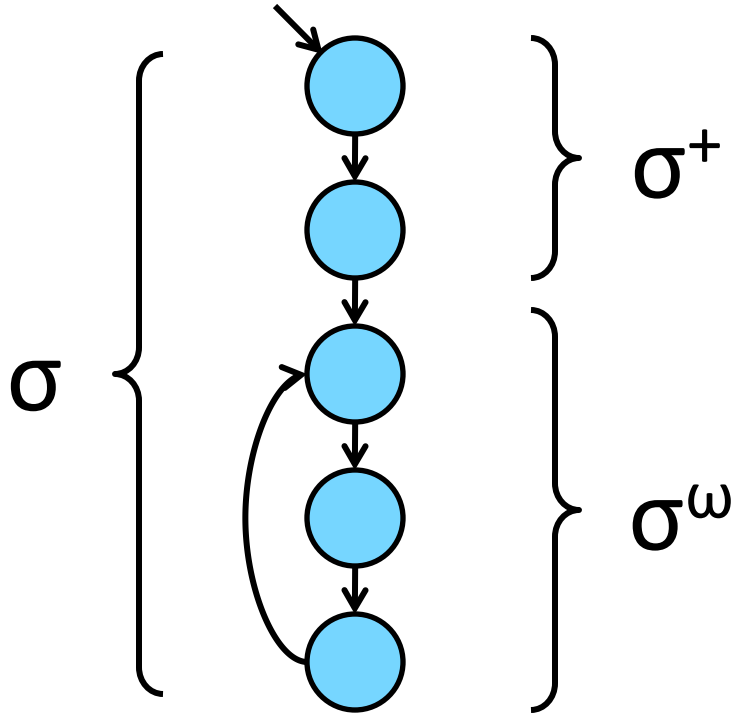
Такое свойство может быть нарушено только бесконечным проходом

Классическое определение описывает лишь конечные проходы

Нужно описать, что автомат не может находиться в терминальном состоянии бесконечно долго.

Немного обозначений

- Для любого бесконечного прохода σ конечного автомата можно выделить два множества:
  - множество σ^+ состояний, встречающихся конечное число раз,
  - множество σ^ω состояний, встречающихся бесконечное число раз.

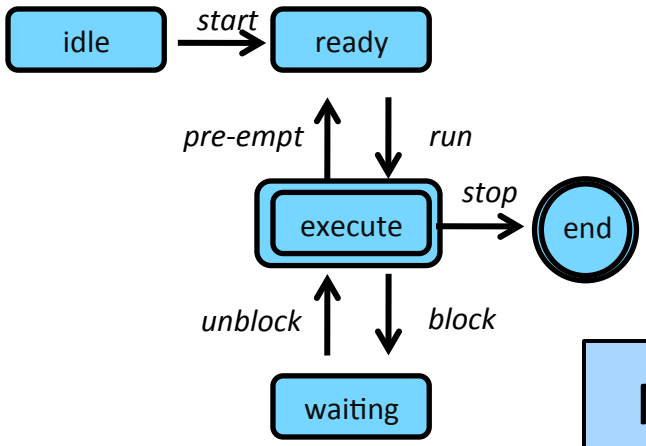


Допускающий проход по Бюхи
(ω-допускание)

- Допускающим ω-проходом конечного автомата A называется такой бесконечный проход σ, что

$$\exists i \geq 0, (s_{i-1}, l_{i-1}, s_i) \in \sigma : s_i \in F \wedge s_i \in \sigma^{\omega}$$

т.е. по крайней мере одно терминальное состояние встречается бесконечно часто.



Допускающий ω-проход:
{idle, ready, {execute, ready}*}

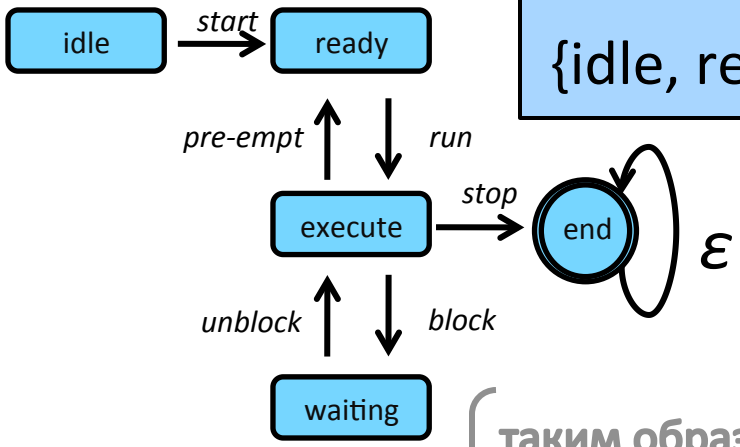
Соответствующее ω-слово:
{start, run, {pre-empt, run}*}

Множество допускаемых автоматом ω-слов называется его ω-языком

Расширение автоматов Бюхи

(конечные проходы как частный случай бесконечных)

- Расширяем алфавит автомата меткой ε (пустой переход),
- Дополняем все конечные проходы бесконечным повторением перехода по метке ε.



Допускающий ω-проход:
{idle, ready, execute, waiting, execute, [end,]*}

Соответствующее ω-слово:
{start, run, block, unblock, stop, ε*}

таким образом, правило ω-допускания становится применимо как для конечных, так и для бесконечных проходов

Проверка свойств при помощи автоматов Бюхи

- При помощи автомата Бюхи можно описать наблюдаемое поведение программы и требования к нему,
- Проход автомата соответствует наблюдаемому вычислению (трассе) программы,
- Определение допускаемости прохода позволяет рассуждать о выполнении или нарушении требований (свойств правильности).

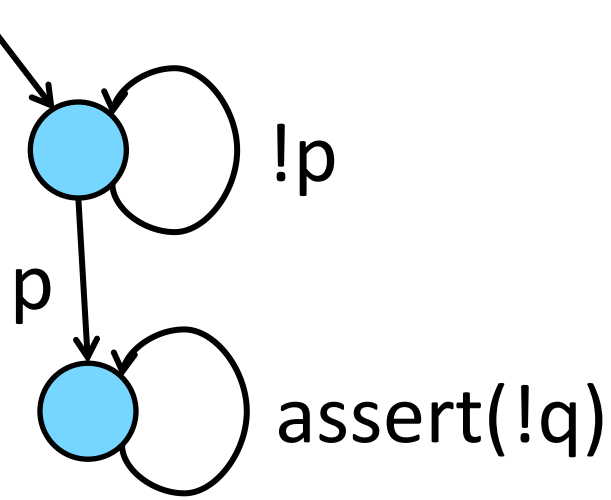
Безопасность и живучесть

- **Безопасность**
  - Любое свойство безопасности можно проверить, исследуя свойства отдельных состояний модели;
  - если свойство безопасности нарушено, всегда можно определить достижимое состояние системы, в котором оно нарушается;
  - для проверки свойств безопасности требуется генерировать состояния системы и для каждого из них проверять свойство;
  - при проверке таких свойств можно обойтись без темпоральных логик и автоматов Бюхи.
- **Живучесть**
  - Для проверки свойств живучести необходимо рассматривать последовательности состояний (конечные и бесконечные проходы соотв. автомата Бюхи);
  - для проверки свойств используются другие, более сложные алгоритмы;
  - свойства удобно описывать при помощи формул темпоральной логики, а проверять – при помощи автоматов Бюхи.

Пример свойства безопасности

Как только p впервые стало истинно,
q больше не может быть истинно.

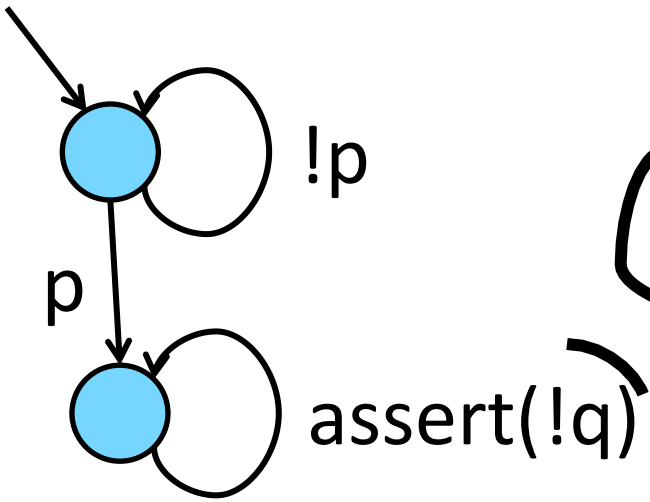
```
never
{
   do
   :: !p
   :: p -> break
   od
   do
   :: assert(!q)
   od
}
```

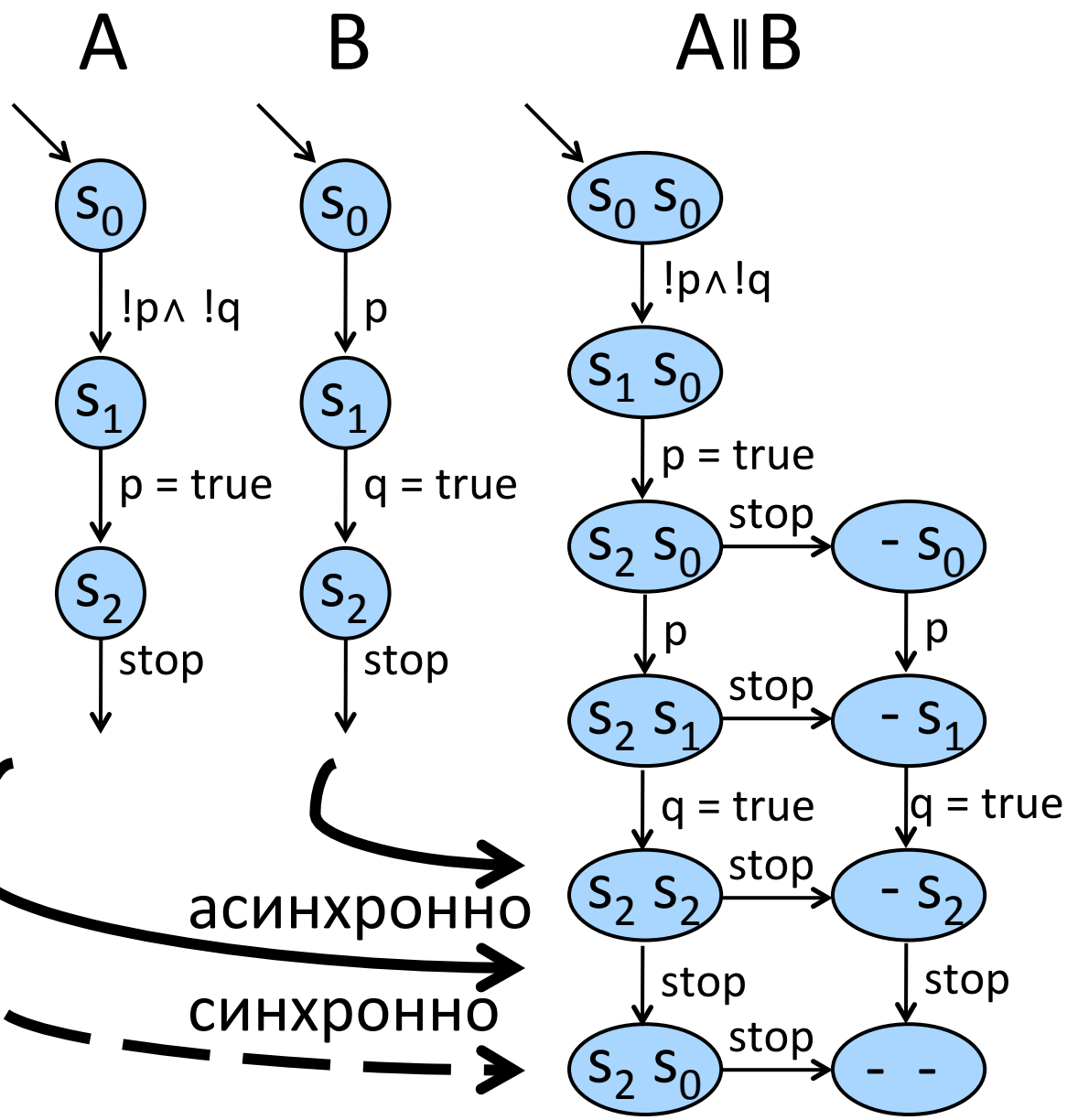


Как только достигнуто состояние, удовлетворяющее условию, будет зафиксировано нарушение свойства. Рассуждать о бесконечных вычислениях здесь не требуется.

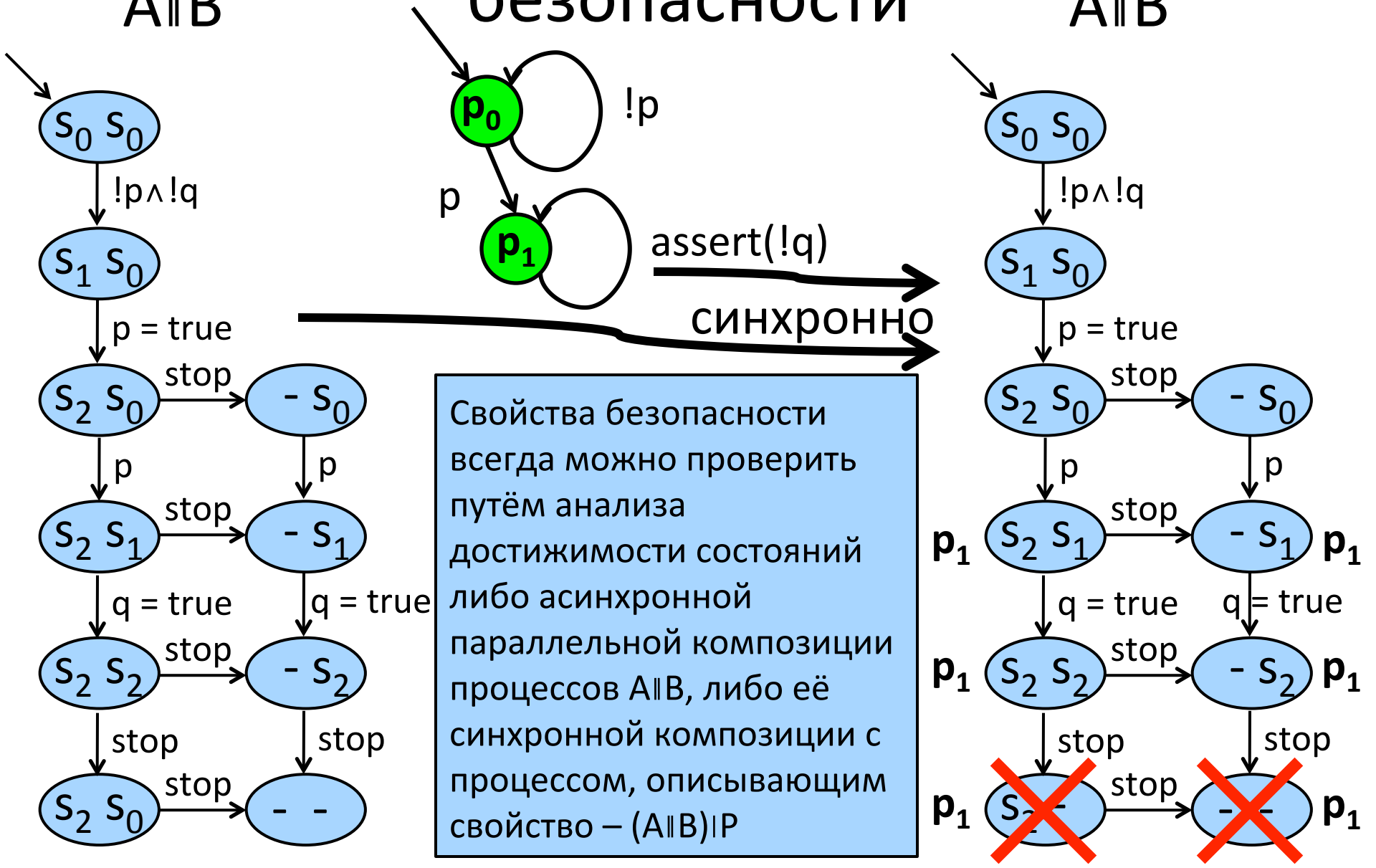
Пример поведения системы

```
bool p,q;

active proctype A()
{
   (!p && !q) -> p = true
}

active proctype B()
{
   (p) -> q = true
}
```





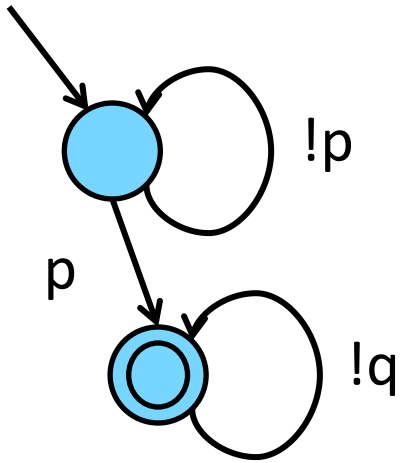
Проверка свойства безопасности



Пример свойства живучести

Как только **p** впервые стало истинно, в течение конечного числа шагов q также станет истинным.

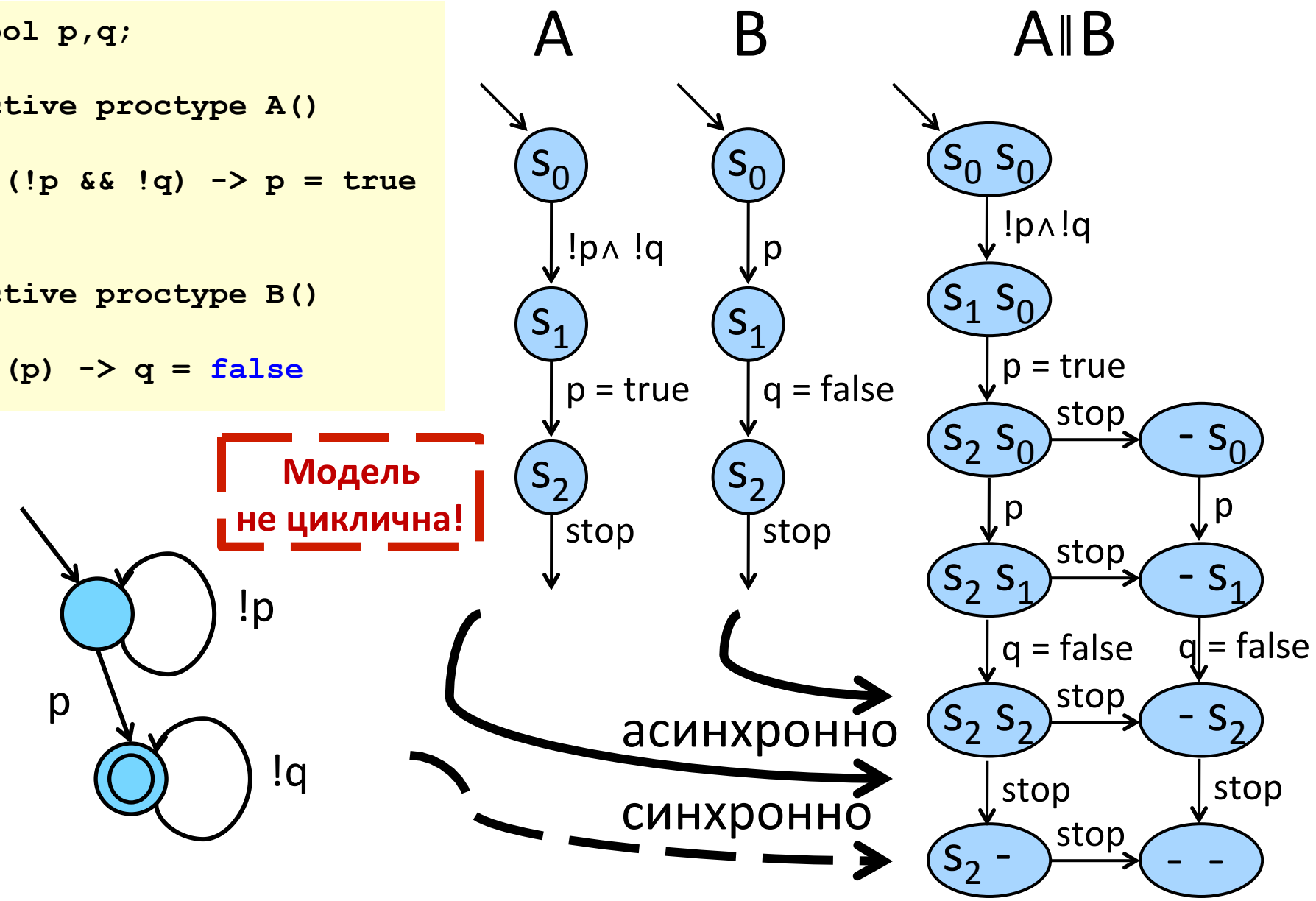
Нарушение свойства: p становится истинным, а затем **q может** навсегда остаться ложным.



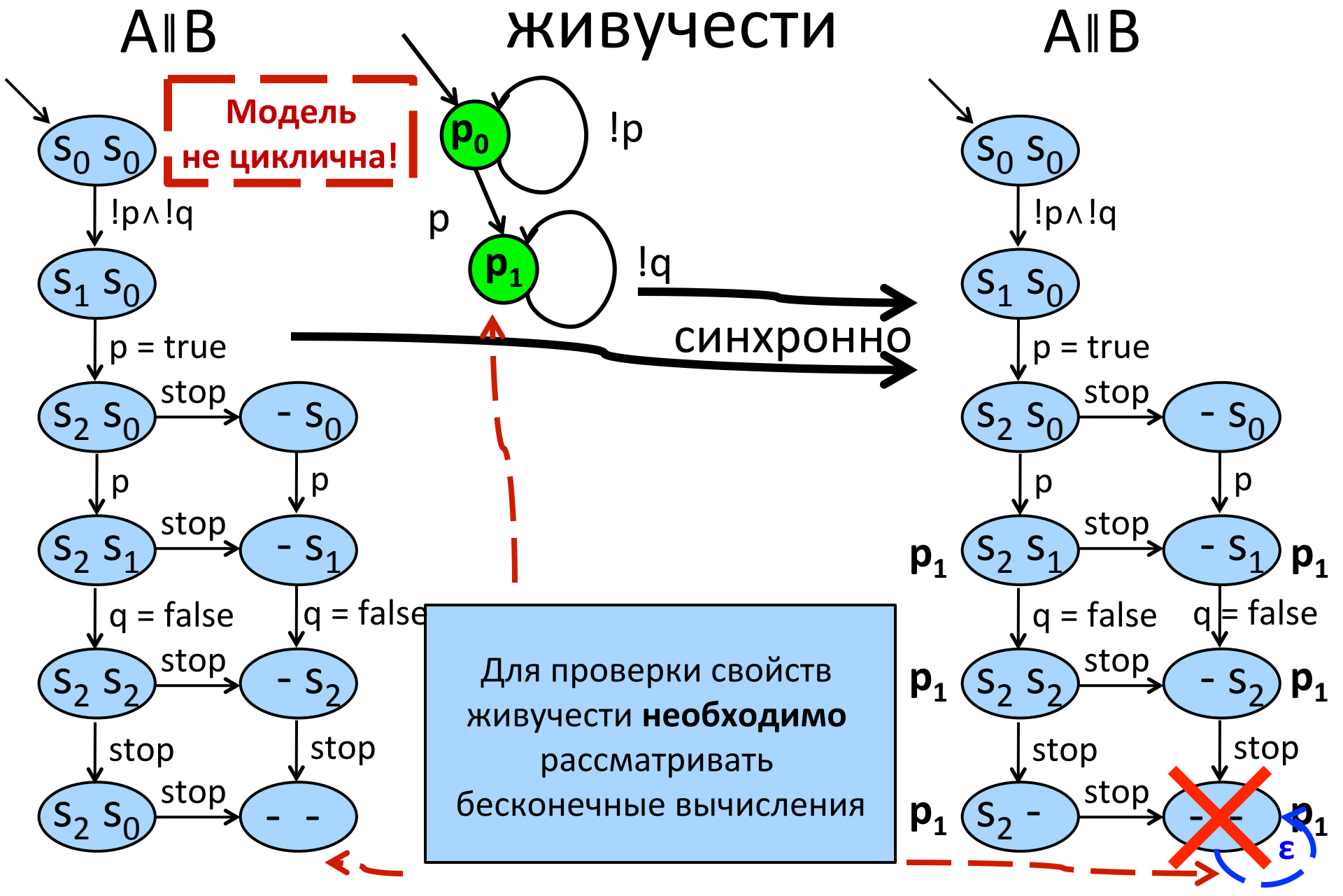
Мы можем заключить о нарушении свойства, только если обнаружим удовлетворяющую условию **потенциально бесконечную** последовательность состояний

Пример поведения системы

```
bool p,q;

active proctype A()
{
   (!p && !q) -> p = true
}

active proctype B()
{
   (p) -> q = false
}
```



Проверка свойства живучести



Спасибо за внимание!
Вопросы?