Верификация программ на моделях

Лекция №7

Практические приёмы абстракции. Спецификация и верификация свойств при помощи автоматов Бюхи.

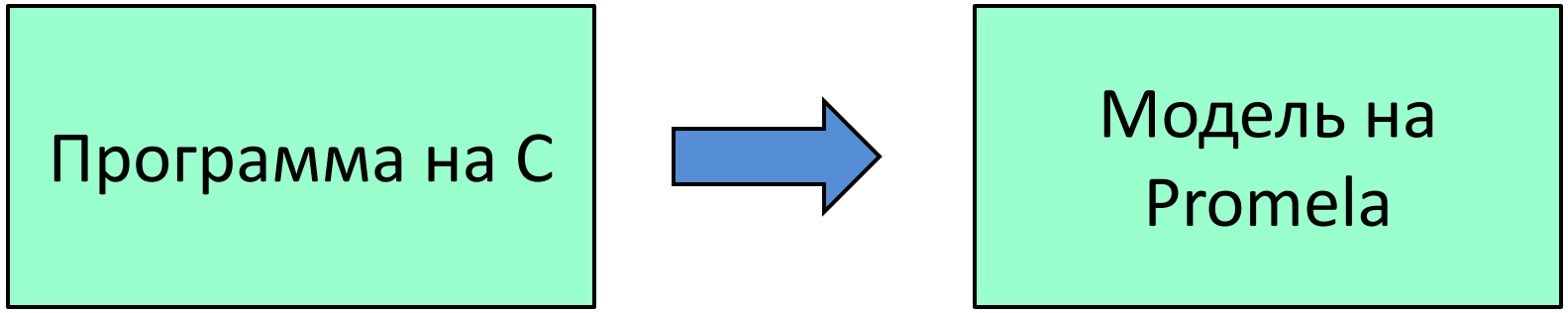
Константин Савенков (лектор)

План лекции

- Практические приёмы абстракции
- Проверка свойств правильности
- Автоматы Бюхи
- Проверка свойств при помощи автоматов Бюхи

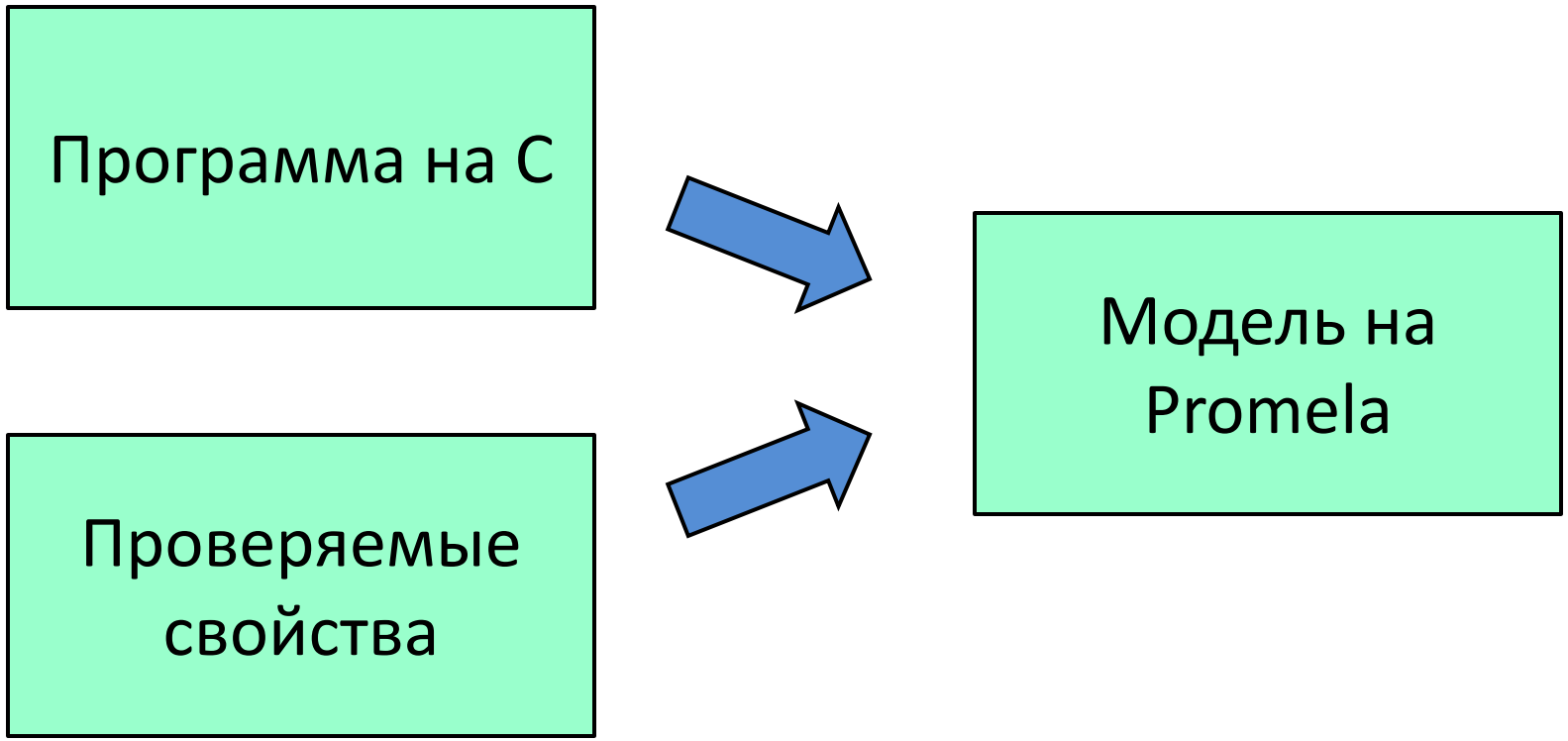
Практические приёмы абстракции

Общая схема



- **Неправильный подход!**
  - Если строить модель без оглядки на проверяемые свойства, то она получится слишком детальной.
  - Скорее всего, не хватит ресурсов на верификацию.

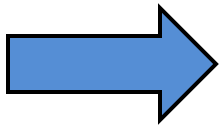
Общая схема



- **Модель должна моделировать только то, что влияет на выполнимость проверяемых свойств.**

Пример

```
void f(int x)
{
  while(x < 10){
    if(x < 3){
      printf("Case 1\n");
    }
    else{
      printf("Case 2\n");
    }
    x++;
  }
}
```

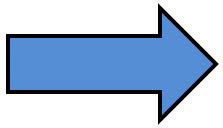


```
proctype f(int x){
  do
  :: x < 3 ->
       printf("Case 1\n");
       x = x + 1
  :: x >= 10 ->
       break;
  :: else ->
       printf("Case 2\n");
       x = x + 1
  od
}

init {
  int x;
  do
  :: x++;
  :: x--;
  :: break;
  od;
  run f(x)
}
```

- Модель точная, но у неё слишком много достижимых состояний.
- **Давайте посмотрим на заданные свойства.**

Пример

```
void f(int x)
{
  while(x < 10){
    if(x < 3){
      printf("Case 1\n");
    }
    else{
      printf("Case 2\n");
    }
    x++;
  }
}
```



```
proctype f(int x){
  do
  :: x < 3 ->
       printf("Case 1\n");
       x = x + 1
  :: x >= 10 ->
       break;
  :: else ->
       printf("Case 2\n");
       x = x + 1
  od
}

init {
  int x;
  do
  :: x++;
  :: x--;
  :: break;
  od;
  run f(x)
}
```

«Проверить, что перед печатью "Case 2" всегда была выполнена печать "Case 1".»

Пример

```
void f(int x)
{
  while(x < 10){
    if(x < 3){
      printf("Case 1\n");
    }
    else{
      printf("Case 2\n");
    }
    x++;
  }
}
```

«Проверить, что перед печатью “Case 2” всегда была выполнена печать “Case 1”.»

```
mtype {X1, X2, X3};
proctype f(mtype x){
  do
  :: x == X1 ->
       printf("Case 1\n");
       incr(x);
  :: x == X3 ->
       break;
  :: x == X2 ->
       printf("Case 2\n");
       incr(x);
  od
}
init{
  mtype x;
  if
  :: x = X1
  :: x = X2
  :: x = X3
  fi;
  run f(x)
}
```

```
inline incr(x){
  if
  :: x += 1;
  :: skip
  fi
}
```

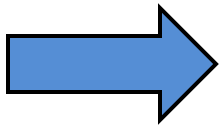
Не загромождаем модель

Приёмы абстракции

- Абстракция предикатов
- Абстракция типов данных
- Абстракция наблюдаемых переменных
- Абстракция от «лишнего» кода
- Абстракция от функций и системных вызовов

Абстракция предикатов

```
if(x < 0){
  printf (“Case 1”);
}
else{
  printf (“Case 2”);
}
…
```



```
if
:: printf (“Case 1”);
:: printf (“Case 2”);
fi
```

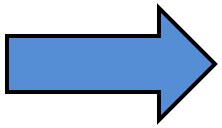
- В модели увеличивается количество возможных вычислений
  - если формула не нарушается ни на одном вычислении модели, то на исходной программе она также будет выполнима,

    следует из корректности модели
  - если формула нарушается на одном из «добавленных» вычислений, необходимо увеличить уровень детализации модели

    модель неадекватна

Абстракция типов данных

```
int x;
…
if(x < 0) {
…
}
else {
  if(x < 3) {
  …
  }
  else {
  …
  }
…
}
```



```
mtype { X1, /*x < 0*/
        X2, /*0 <= x < 3*/
        X3  /*x >= 3*/
      };

if
:: x == X1 -> …
:: x == X2 -> …
:: x == X3 -> …
fi
```

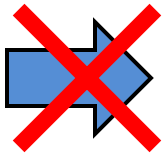
- Диапазон типа данных разбивается на классы эквивалентности относительно конструкций организации потока управления и проверяемых свойств.
- Необходимо корректно моделировать:
  - предикаты в операторах ветвления,
  - операторы, изменяющие значения переменных этого типа.

Абстракция наблюдаемых переменных

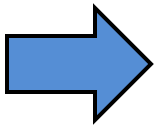
- Из модели удаляется переменные, значения которых не влияют на выполнимость заданных свойств:
  - не используются при формулировке свойства,
  - не влияют на поток управления модели.
- Операторы модели могут быть связаны достаточно сложными зависимостями по данным и управлению, поэтому в ходе удаления переменных допускается **только расширение** множества возможных вычислений модели

Абстракция наблюдаемых переменных

```
int x;
…
while (x != 0){
 x = get_value();
}
…
printf (“Something\n”);
```



```
printf (“Something\n”);
```

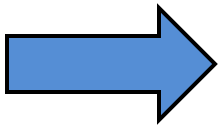


```
do
:: skip
:: break
od;
printf(“Something\n”)
```

- Свойство – операция печати будет выполнена один раз,
- Если абстрагируемся от цикла – теряем возможность бесконечного зацикливания (сужаем множество возможных вычислений).

Абстракция «лишнего» кода

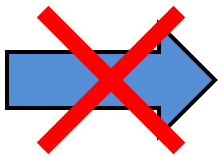
```
…
some_func();
printf(“Func called!\n”)
…
```



```
…
some_func: skip;
…
```

- Свойство – some_func() будет вызвана.
- Можно абстрагироваться от:
  - кода, не влияющего на проверяемое свойство (printf),
  - реального выполнения операторов, если нас интересует лишь факт его выполнения («функция была вызвана»).
- Здесь также необходимо следить за допустимыми вычислениями.

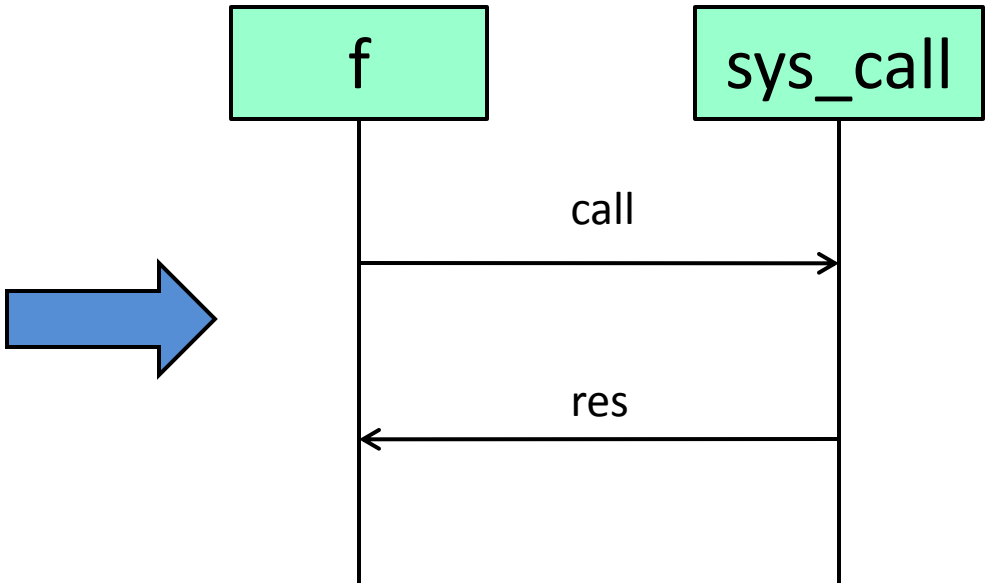
```
…
int x;
x = some_func();
if (x)
…
```



```
…
int x;
some_func: skip;
if
:: x -> …
…
```

Абстракция от функций и системных вызовов

```
f()
{
   sys_call();
}
```



- Выполнение реальной функции заменяется на обмен сообщениями с процессом-«заглушкой»,
- Обмен сообщениями должен корректно моделировать вызов функции:
  - синхронное взаимодействие, гарантированный отклик.

От чего не стоит абстрагироваться

- Поток управления:
  - ветвление, циклы;
- Можно добавлять новые ветви и пути выполнения:
  - см. «абстракция предикатов»,
  - если на добавленных путях будет нарушаться свойство, то эта ошибка будет обнаружена при анализе контрпримера.
- Нельзя убирать пути выполнения на основе «метода пристального взгляда»
  - можно случайно убрать путь выполнения программы, на котором происходит нарушение свойства правильности программы.

Пример поэтапной абстракции

```
int x = 0;
…
while (x < 5){
 x++;
}
…
printf (“Something\n”);
```

1

```
cycle: do
       :: inc: skip
       :: break
       od;
printf(“Something\n”)
```

2

```
//finite cycle removed
printf (“Something\n”);
```

- Свойство – «Рано или поздно “Something” будет напечатано».
- При полной предикатной абстракции модель корректна, но не адекватна свойству.
- Если мы докажем, что в цикле – конечное число итераций, от него можно абстрагироваться.
- Для этого нужно показать, что между двумя проверками условия цикла всегда происходит инкремент x.

Типичные ошибки и «странности»

```
bit b;
...
if
:: b == 1 -> A
:: b == 1 -> A
:: b == 1 -> A
:: b == 0 -> B
fi
```

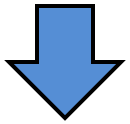
«Я хочу повысить вероятность b == 1»

Недетерминизм связан не с понятием **вероятности**, а с понятием **возможности**

Типичные ошибки и «странности»

```
bit b;

active proctype A()
{
  do
  :: b == 1 -> A
  :: else -> skip
  od
}
```

«Я жду, пока не выполнится условие!»



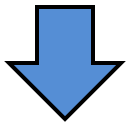
```
bit b;

active proctype A()
{
  (b == 1) -> A
}
```

Используйте нативные конструкции Promela!

Типичные ошибки и «странности»

```
bit b;
...
if
:: b = 1
:: b = 0
fi;
if
:: b == 1 -> A
:: b == 0 -> B
fi
```

«Переменная может принимать разные значения»



```
if
:: A
:: B
fi
```

От неё вообще нужно абстрагироваться!

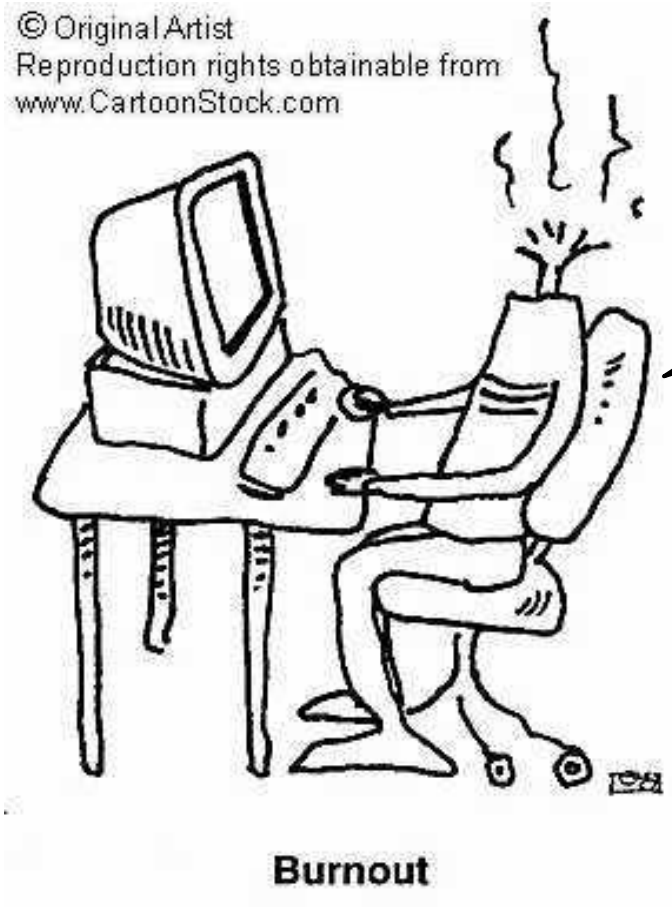
Типичные ошибки и «странности»

```
chan f1 = [0] of bit;
chan f2 = [0] of bit;

active proctype caller()
{
  do
  :: if
     ::f1!m;
     :: else -> break
     fi;
     ...
  od
}

active proctype callee()
{
  msg m;
  do
  :: f1?m -> f2!m
  od
}
```

«Но я же должен учесть, что канал может быть занят!»

Некорректное моделирование вызова функции: вызов «сбрасывается», если процесс-«заглушка» занят

Типичные ошибки и «странности»



«Я сначала построю модель, а потом уберу всё лишнее»

Сначала прочитайте спецификацию!!!

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 ¬ф

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

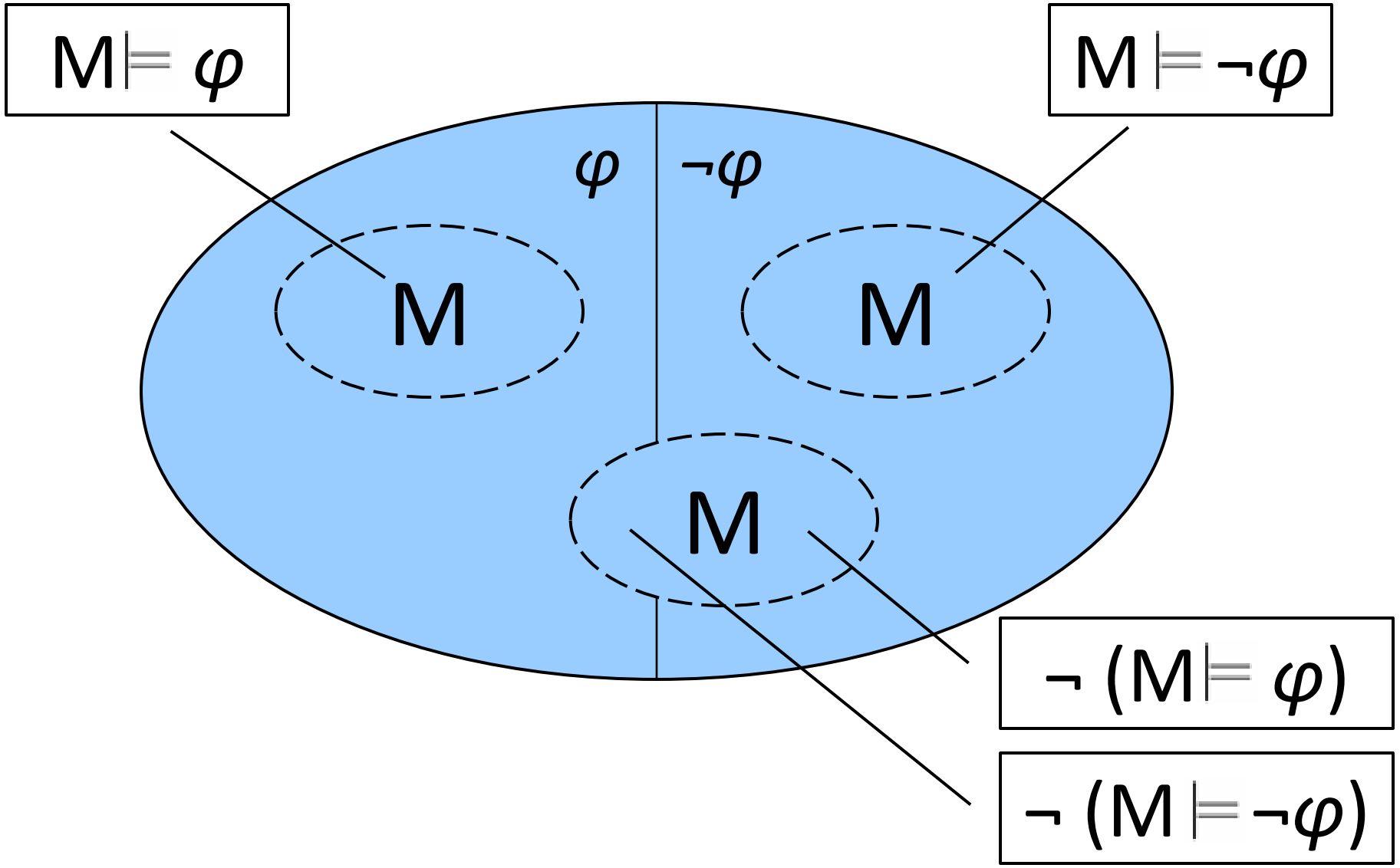
- **ПОЧЕМУ?**

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

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

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

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

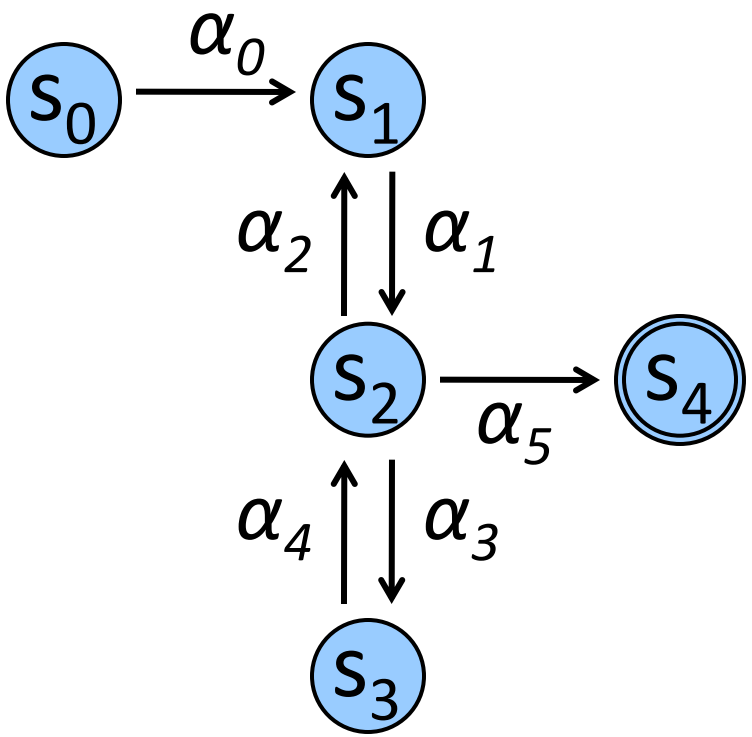
$$\langle S, s_0, L, F, T \rangle$$

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

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



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

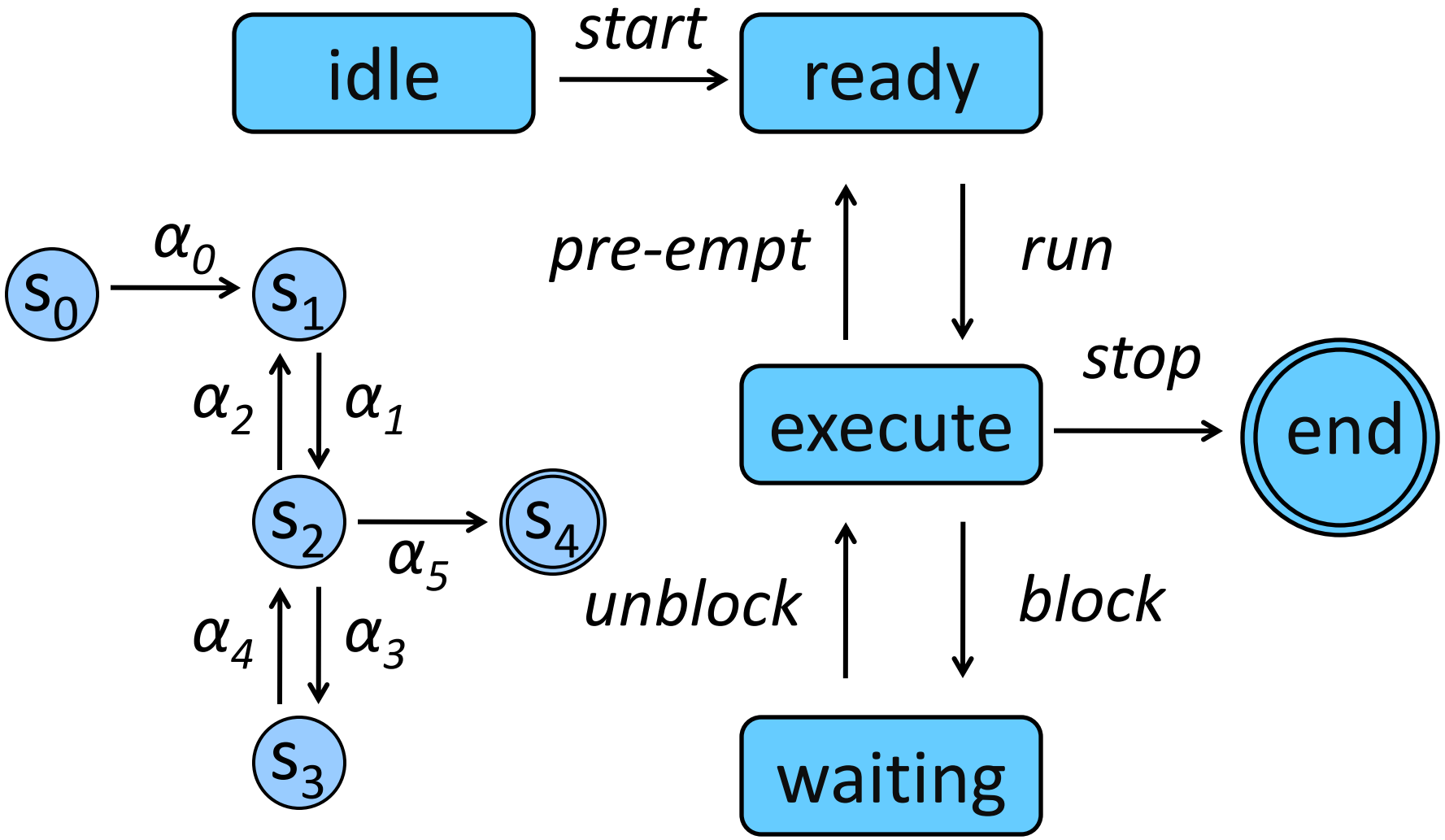
$$L = \{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$$

$$F = \{s_4\}$$

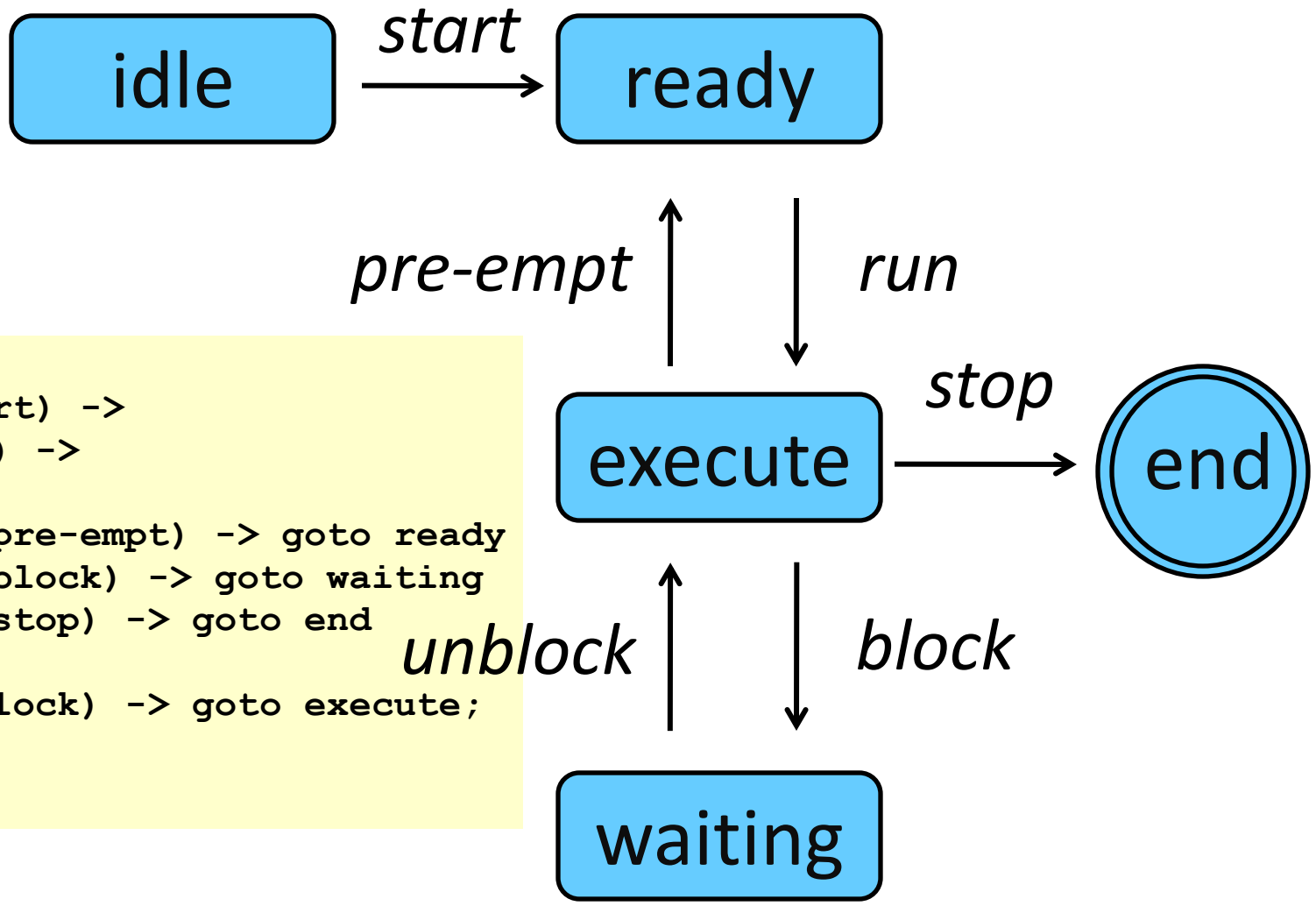
$$T = \{(s_0, \alpha_0, s_1), (s_1, \alpha_1, s_2), \ldots\}$$

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

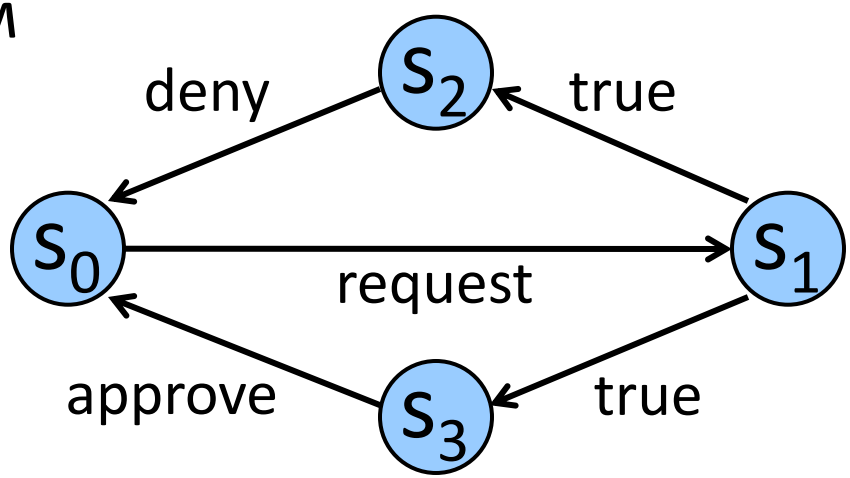
- Конечный автомат $A = \langle S, s_0, L, F, T \rangle$

называется детерминированным, только если

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

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

Модель сервера запросов, работающего в бесконечном цикле



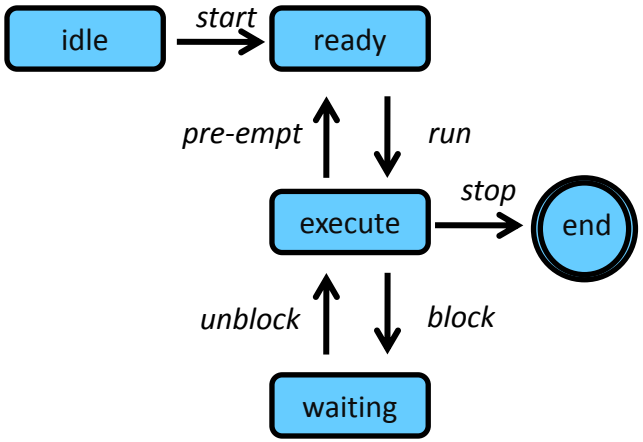
Определение *прохода*

- 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.

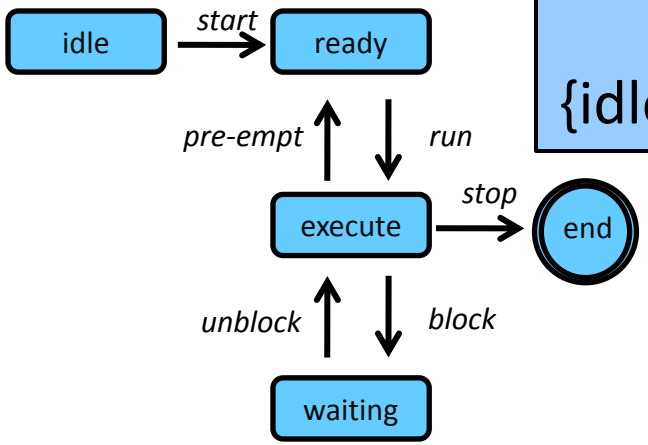


Последовательность состояний:
{idle, ready, {execute, waiting}*}

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

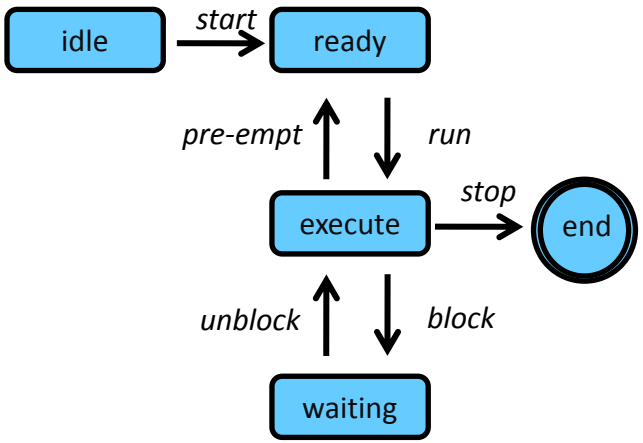


Последовательность состояний
для допускающего прохода:
{idle, ready, execute, waiting, execute, end}

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}

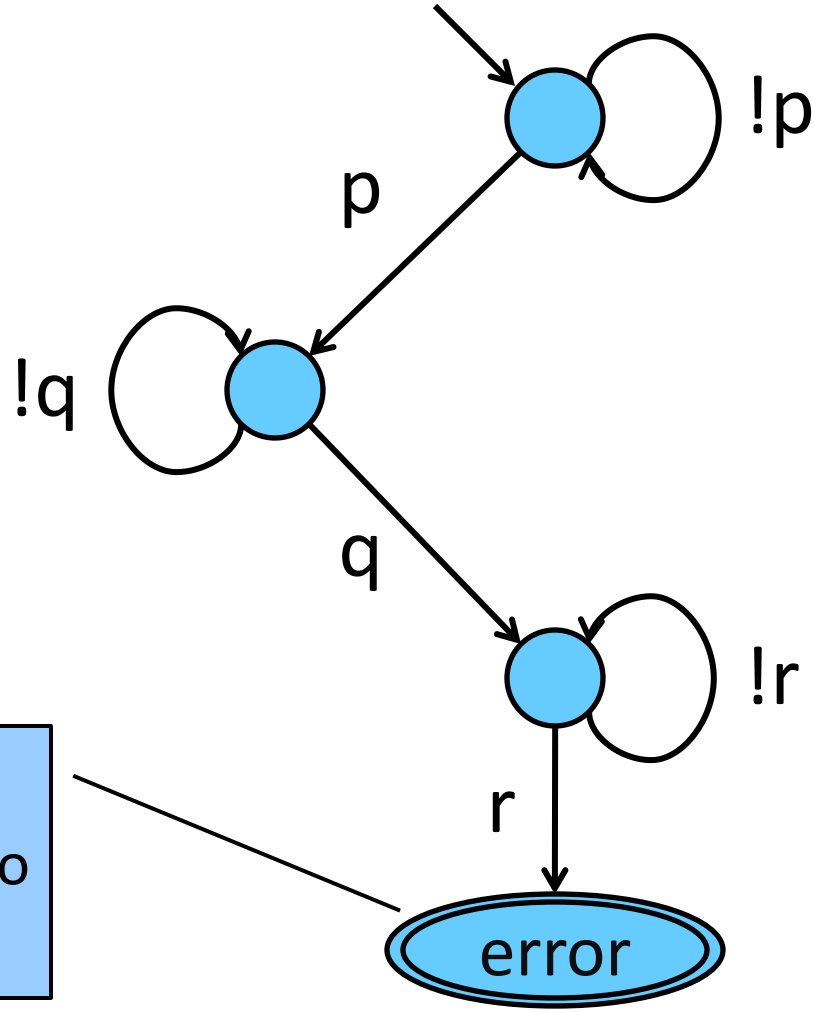
Описание свойств при помощи автомата

Пример свойства:

если сначала p=T,

а позже q=T,

то впоследствии r=F

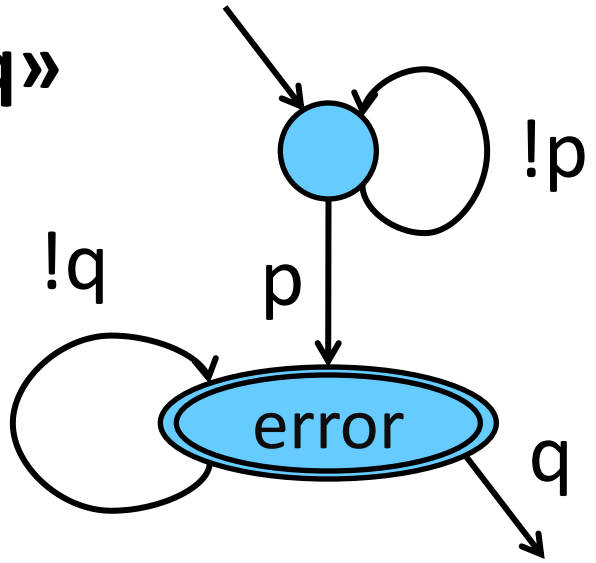


Если мы попали в терминальное состояние, то свойство нарушается

Иногда нужно рассуждать о потенциально бесконечной задержке

Классическое свойство живучести:

«если p, тогда впоследствии q»



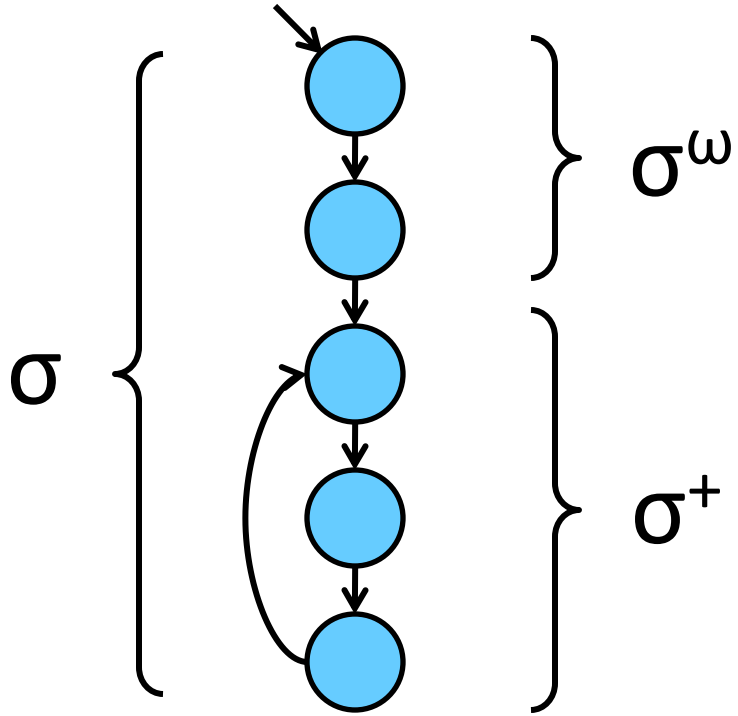
Такое свойство может быть нарушено только бесконечным проходом

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.

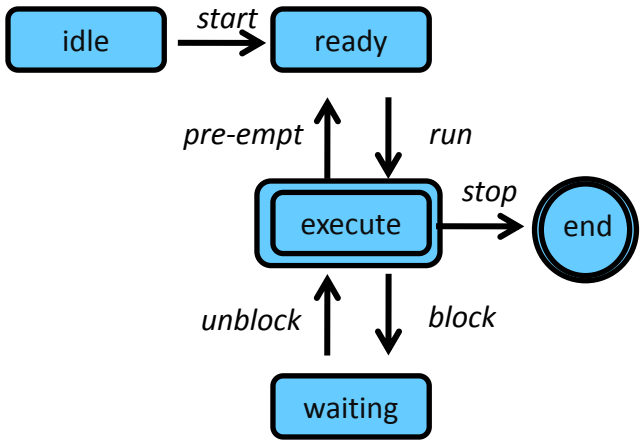


Допускающий проход по Бюхи
(ω-допускание)

- 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}*}

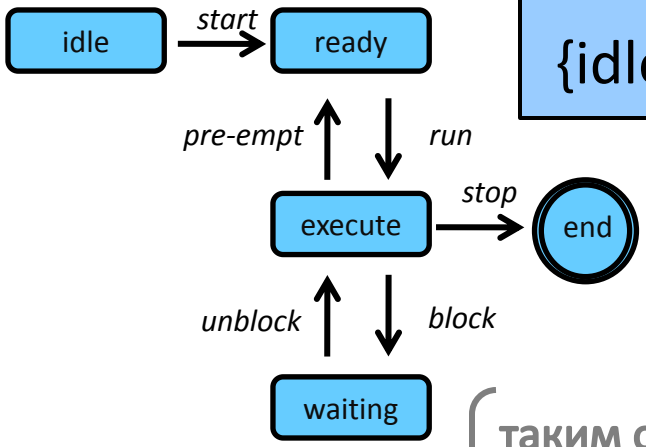
Соответствующее ω-слово:
{start, run, {pre-empt, run}*}

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).

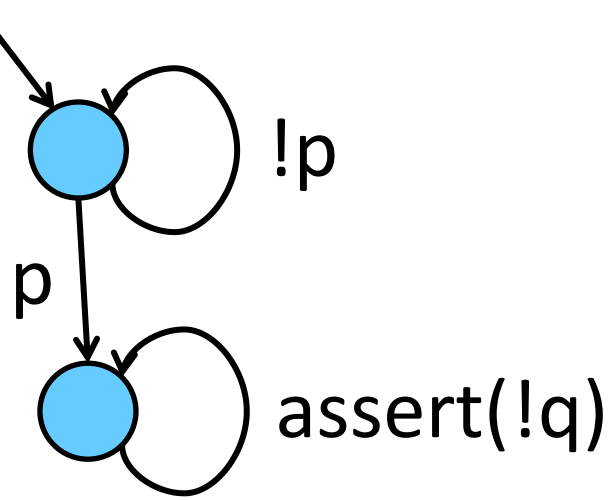
Безопасность и живучесть

- **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 проверки 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.

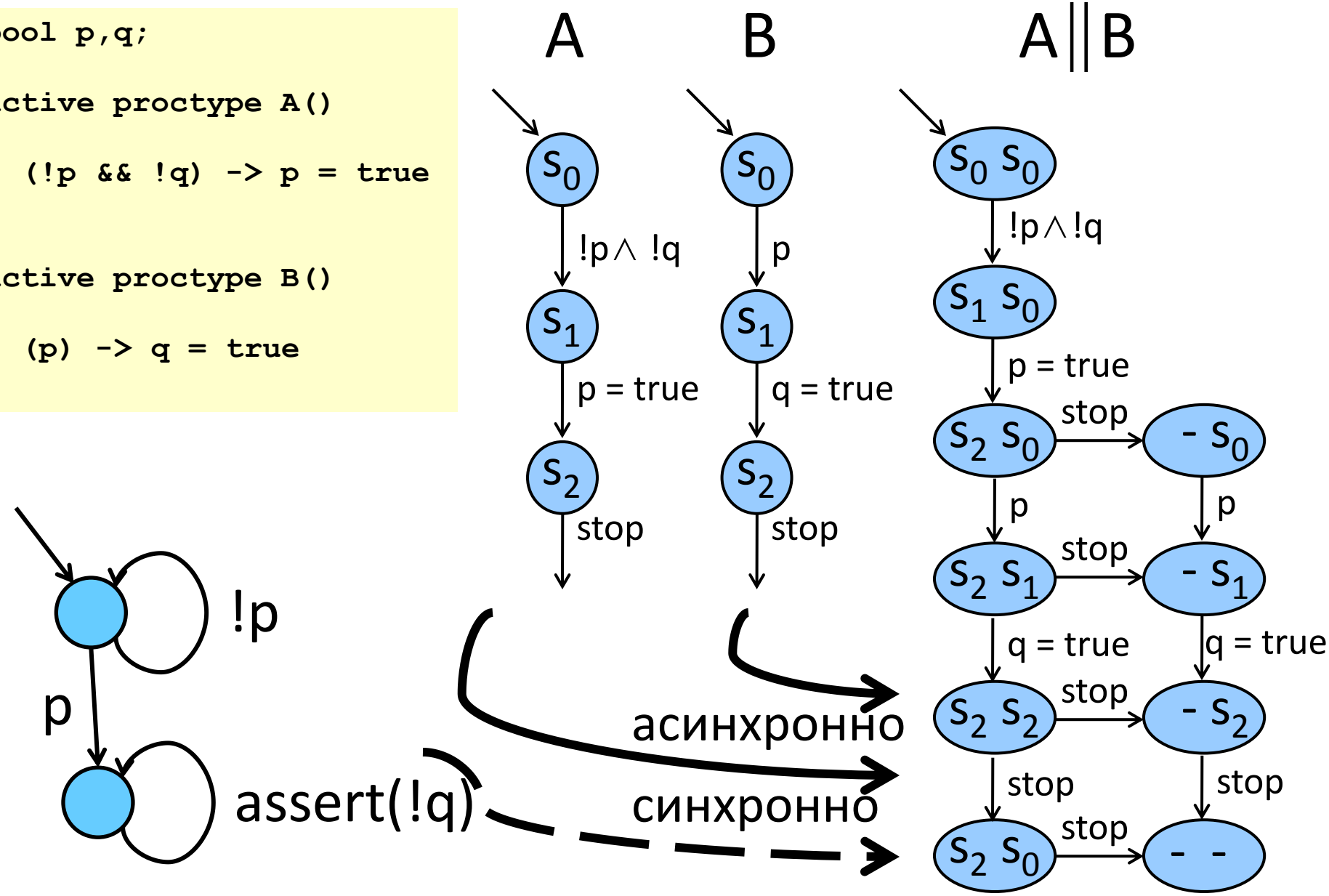
```
never
{
   do
   :: !p
   :: p -> break
   od
   do
   :: assert(!q)
   od
}
```



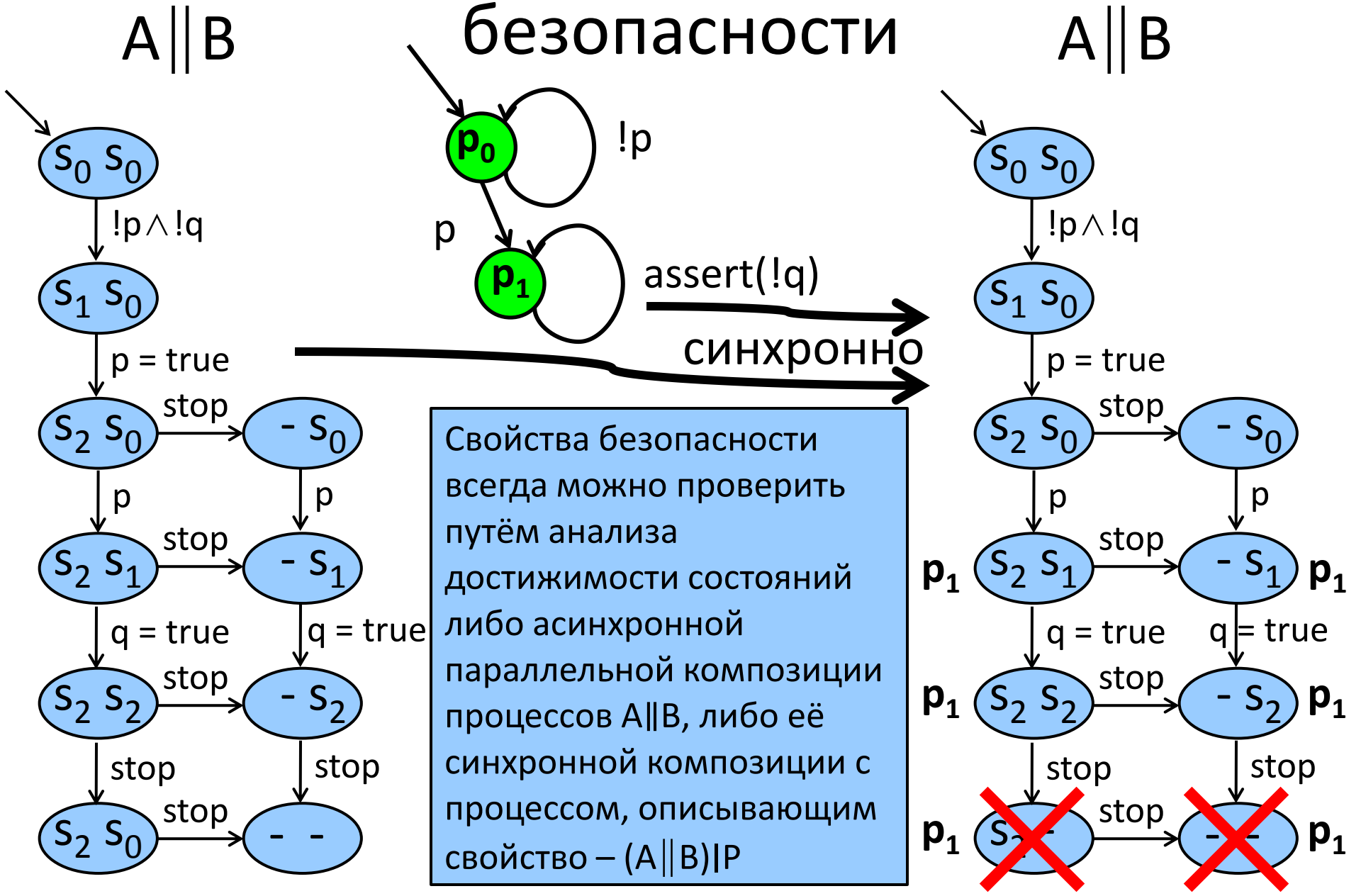
Как только достигнуто состояние, удовлетворяющее условию, будет зафиксировано нарушение свойства. Рассуждать о бесконечных вычислениях здесь не требуется.

Пример поведения системы

```
bool p,q;

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

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



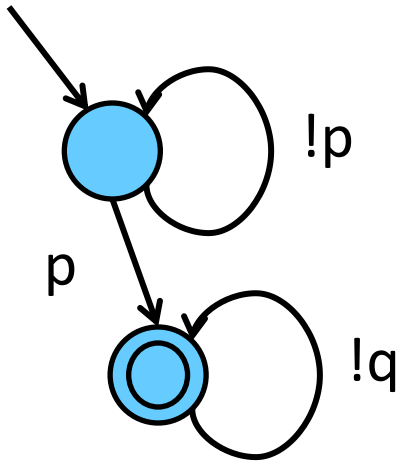
Проверка свойства безопасности



Пример свойства живучести

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

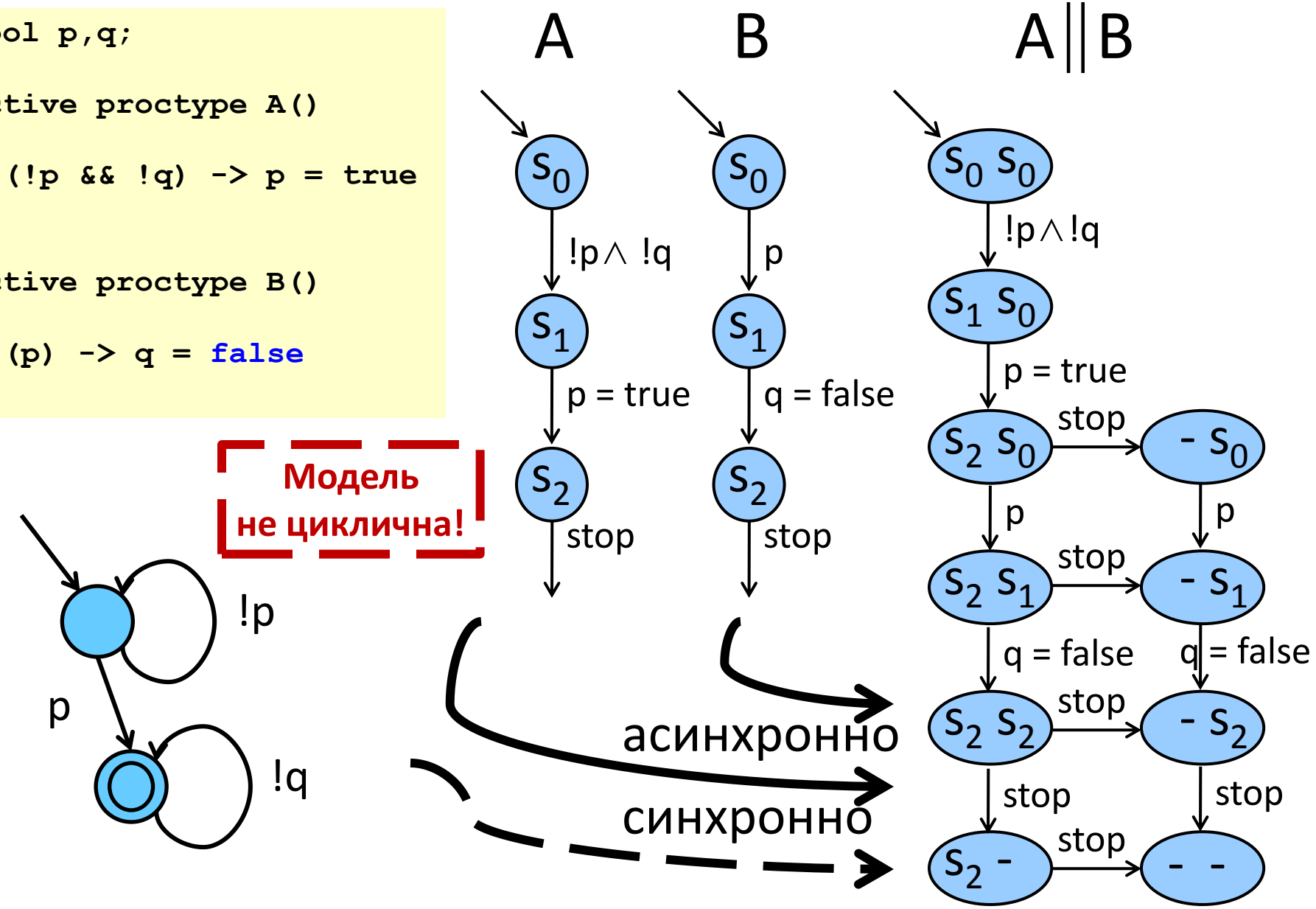
Нарушение свойства: p становится истинным, а затем **q может** навсегда остаться ложным.



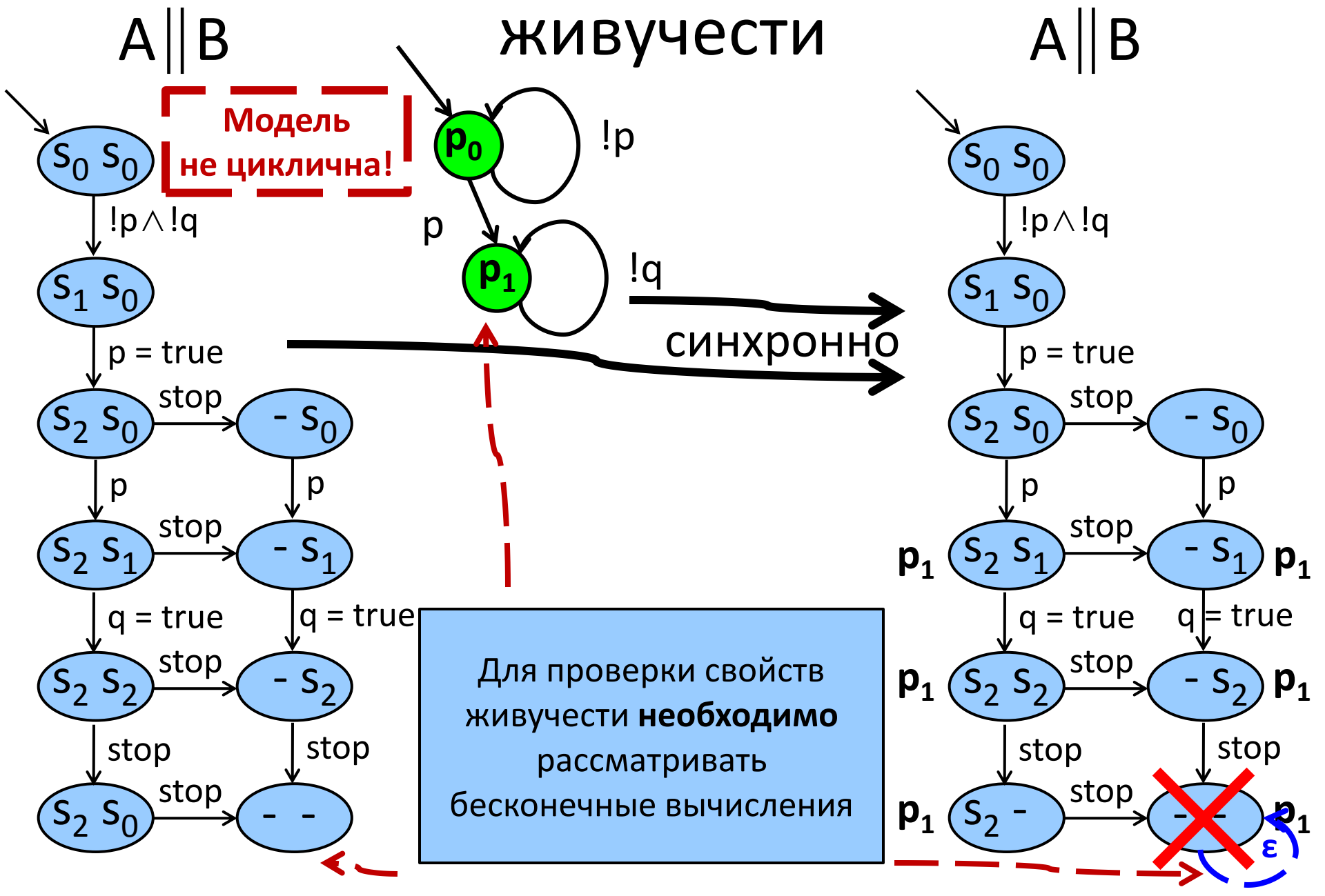
Мы можем заключить о нарушении свойства, только если обнаружим удовлетворяющую условию **потенциально бесконечную** последовательность состояний

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?