Верификация программ на моделях

Лекция № 10

Алгоритмы верификации моделей в системе Spin

Игорь Коннов

План лекции

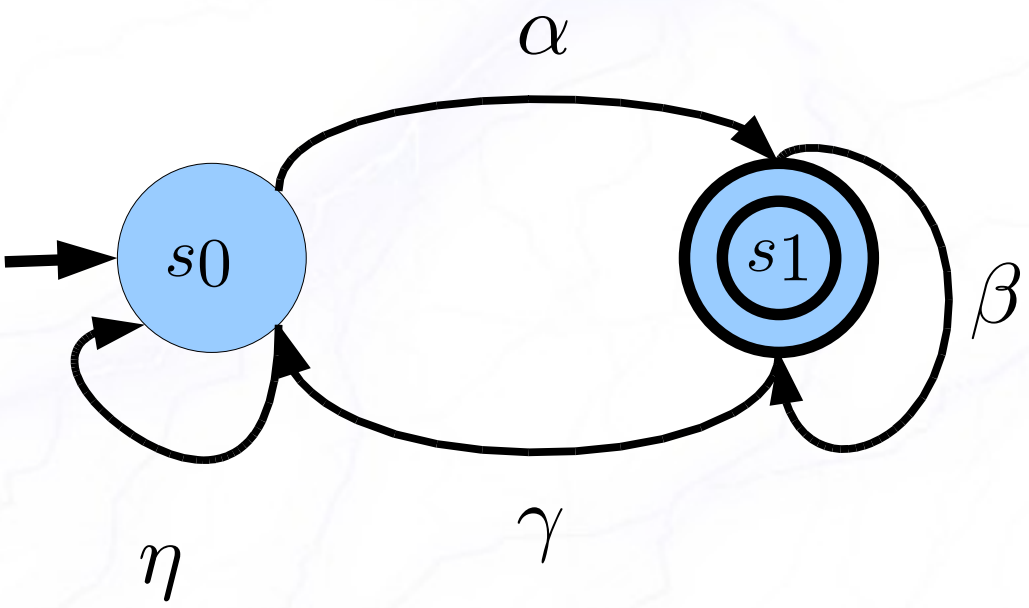
- Определения из предыдущих лекций
- Алгоритм верификации модели для логики линейного времени
- Реализация: поиск в...
- Сложность
- Подводные камни
- Сокращение времени перебора и потребляемой памяти

Автомат Бюхи

- $B = (S, s_0, L, F, T)$

{или $B = (\Sigma, Q, \delta, q_0, F)$, $\Sigma = L$, $Q = S$, $q_0 = s_0$, $\delta = T$ }

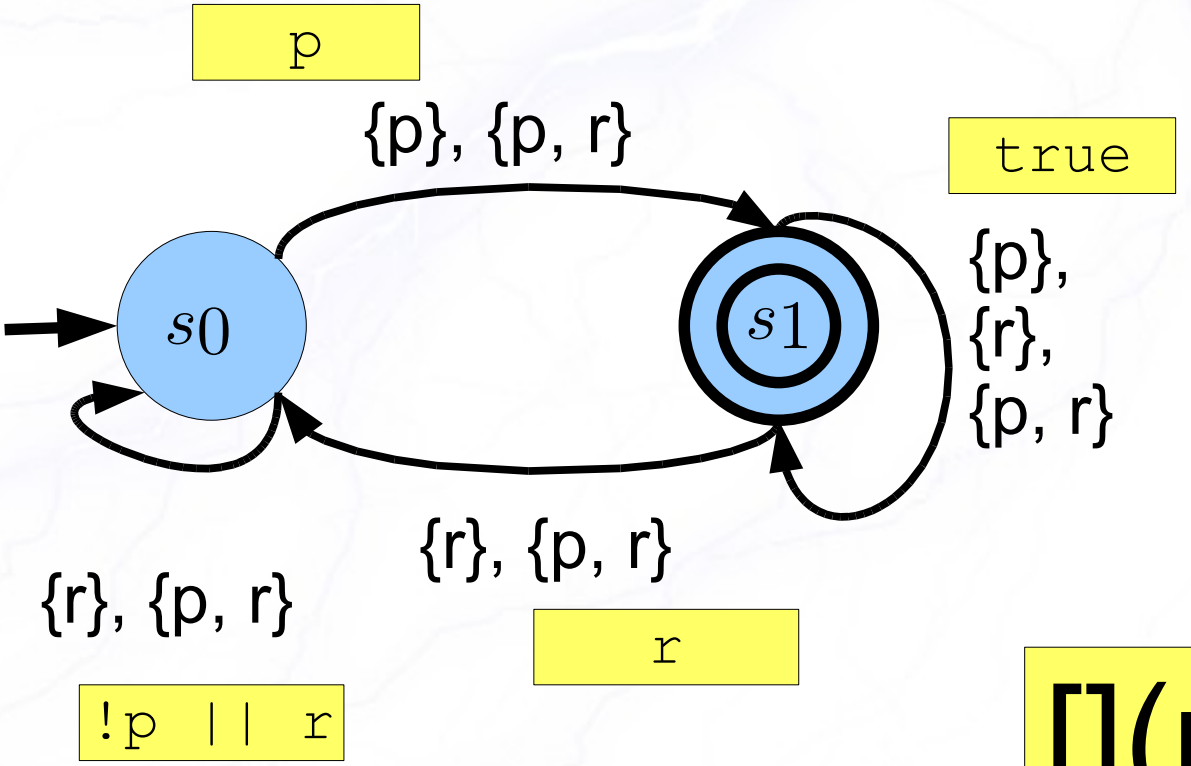
- Автомат над бесконечными словами!



Автомат Бюхи по формуле LTL

$B = (S, s_0, L, F, T)$

{или $B = (\Sigma, Q, \delta, q0, F), \Sigma = L, Q = S, q0 = s0, \delta = T$ }



[](p -> <> r)

Как построить автомат?

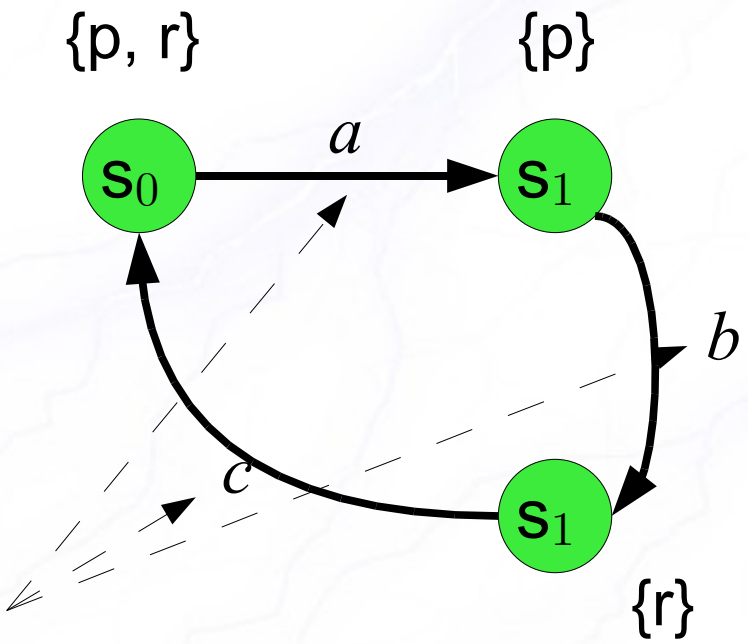
- Формула LTL φ
- Размер автомата $|B\varphi| = O(2^k)$, k – число подформул в формуле φ
- Несколько алгоритмов:
  - Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. Proc. PSTV 1995.
    - Реализован в Spin.
  - P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), Paris, France, July 2001, LNCS 2102, pages 53-65. Springer.
    - http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php

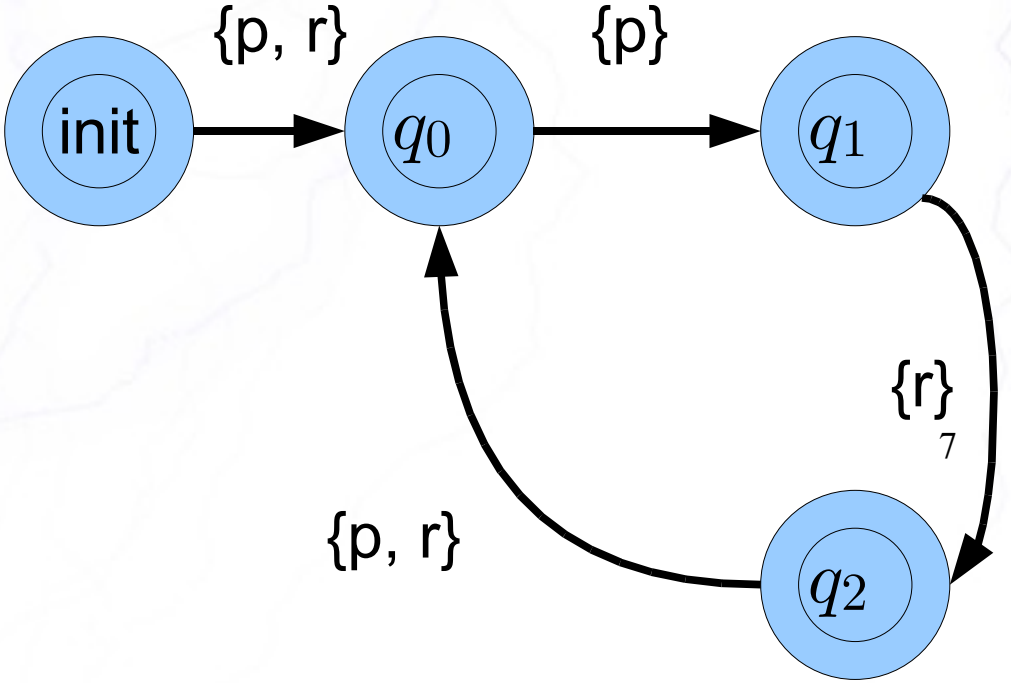
Размеченные системы переходов (LTS)

- TS = $(S, Act, \rightarrow_a, s_0, AP, L)$
- Act – множество действий, $\tau \in Act$
- $\rightarrow_a \in S \times Act \times S$ – тотальное отношение переходов
- $s_0 \in S$ – начальное состояние
- AP – множество атомарных высказываний
- $L: S \rightarrow 2^{AP}$ – функция разметки

LTS и автоматы Бюхи

- Спецификация задаётся автоматом Бюхи
- Как сравнить модель (LTS) и автомат Бюхи?
- Построить автомат Бюхи по LTS



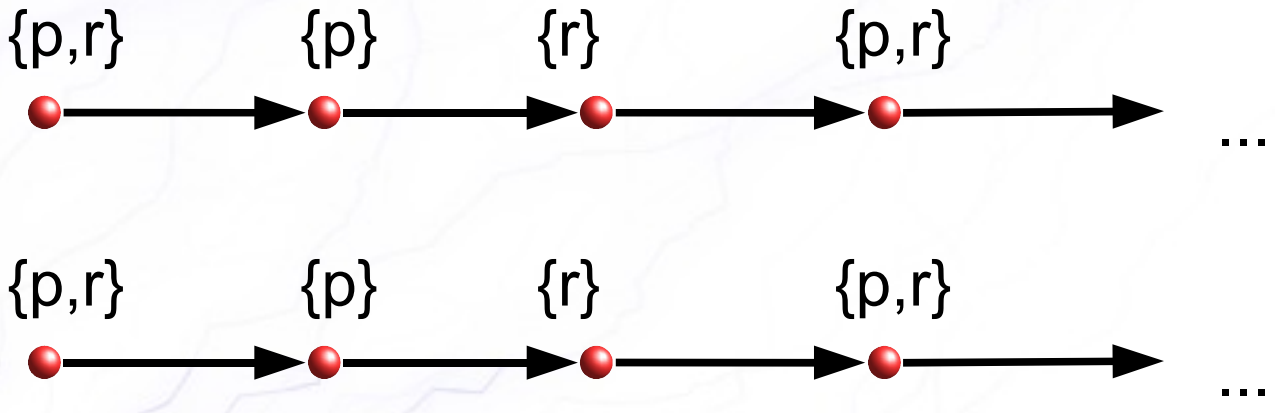


Не интересны (структура Крипке)

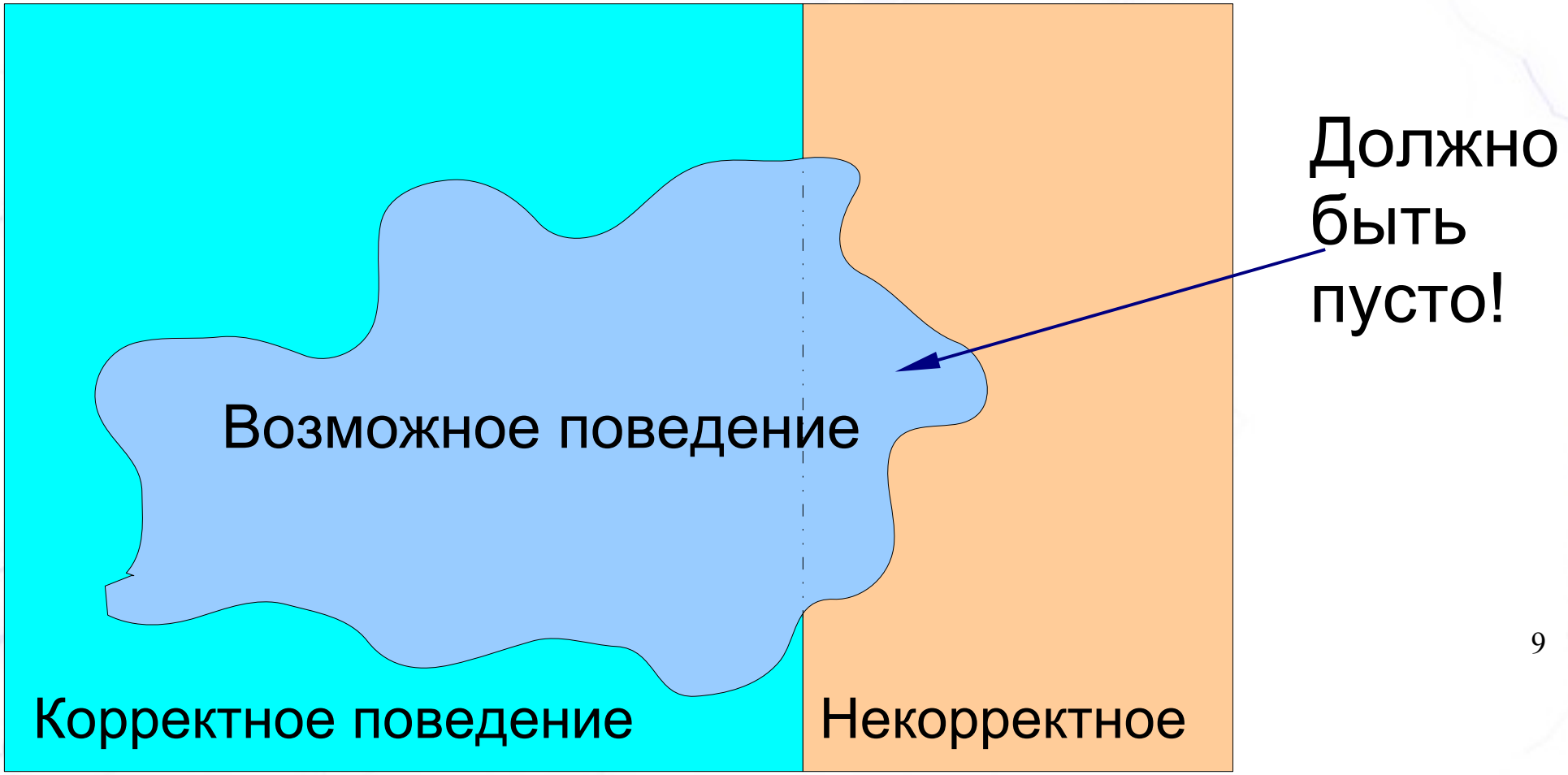
Проверка выполнимости

- Формула LTL φ и соответствующий автомат Бюхи Bφ
- LTS M и соответствующий автомат Бюхи B_M
- Проверить $\mathcal{L}(B_M) = \mathcal{L}(B\varphi)$
- Перебрать все вычисления?

Бесконечное число вычислений



Верификация программы на модели



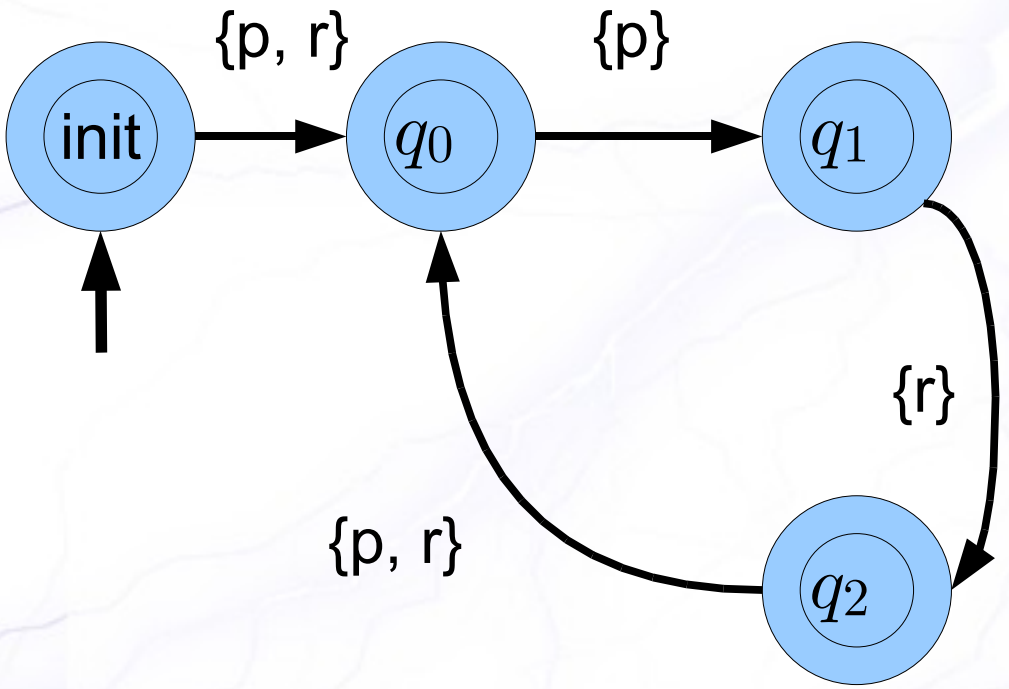
Проверка на пустоту

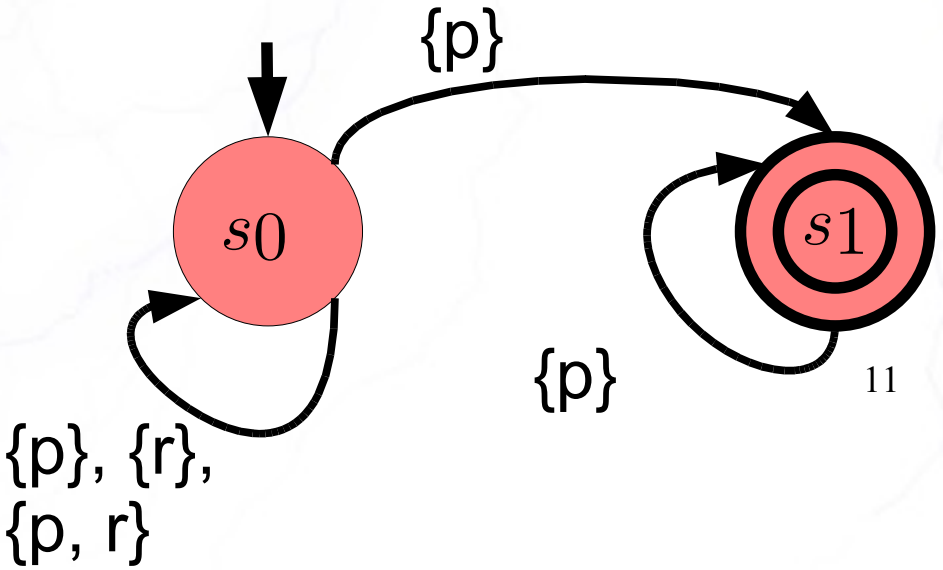
- Вместо $\mathcal{L}(B_M) = \mathcal{L}(B\varphi)$ можно проверить:
  - $\mathcal{L}(B_M) \cap \mathcal{L}(B\neg\varphi) = \varnothing$
  - Надо уметь строить пересечение языков
  - Надо уметь проверять язык на пустоту
- Можно проверить:
  - $\mathcal{L}(B_M \otimes B\neg\varphi) = \varnothing$
  - Так и делает Spin!

Задачи разрешимы!

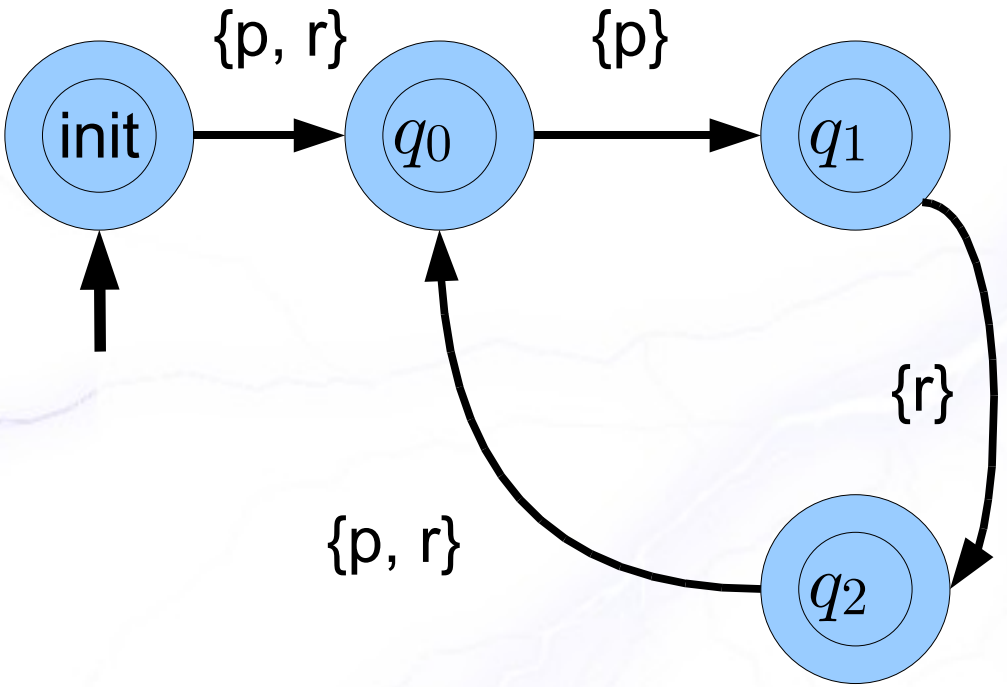
Синхронная композиция

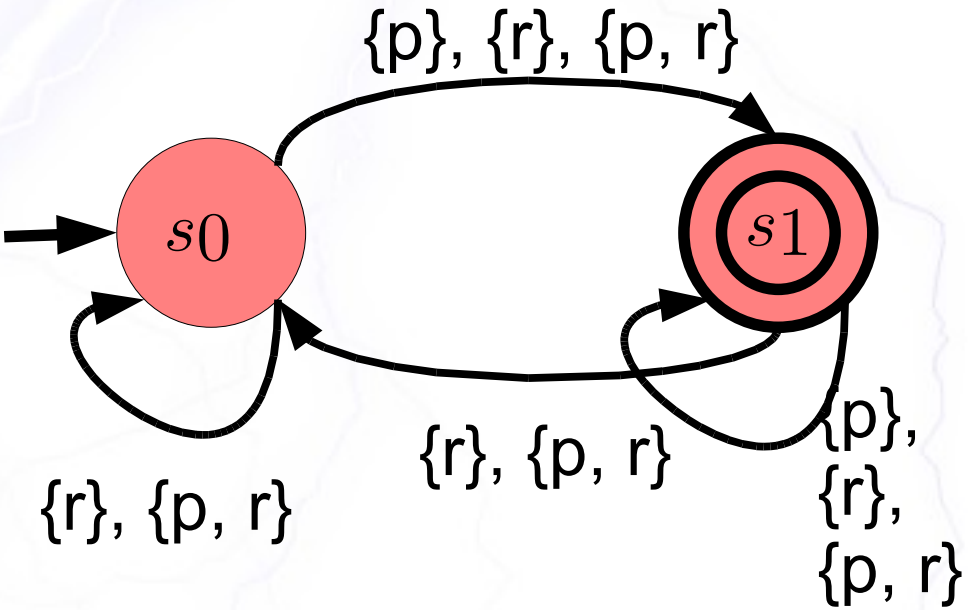
- Сложно ли построить $B_M \otimes B\neg\varphi$?



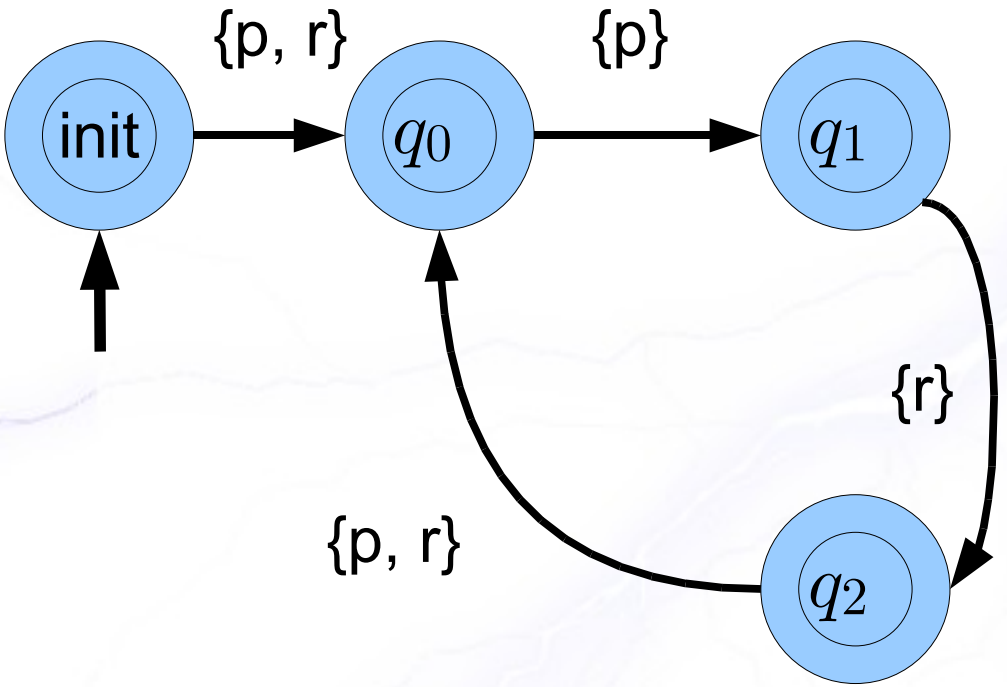


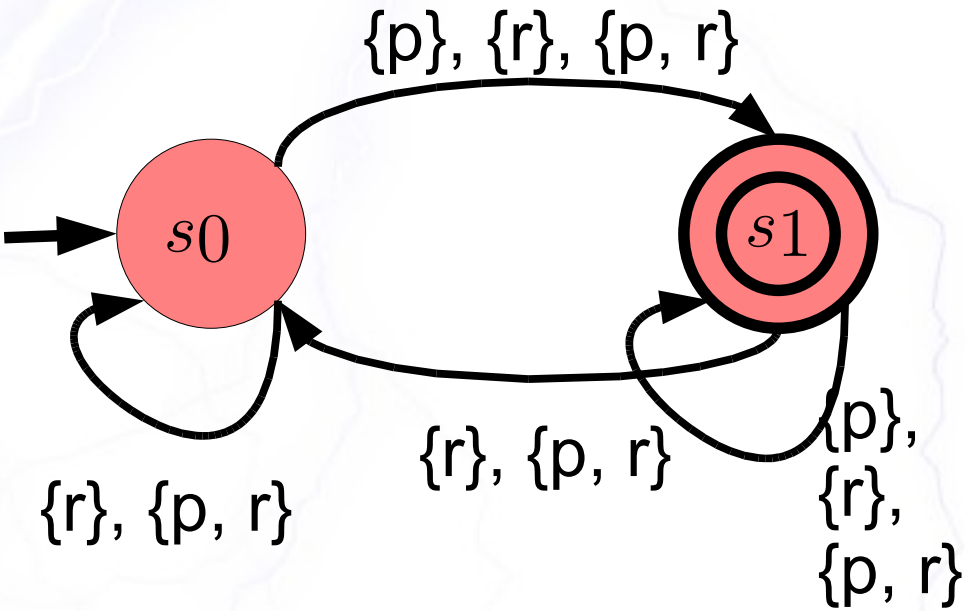
Синхронная композиция (2)





Синхронная композиция (2)





Слово из $\mathcal{L}(\mathrm{B}_{\mathrm{M}} \otimes \mathrm{B}\neg\varphi)$

Контрпример!

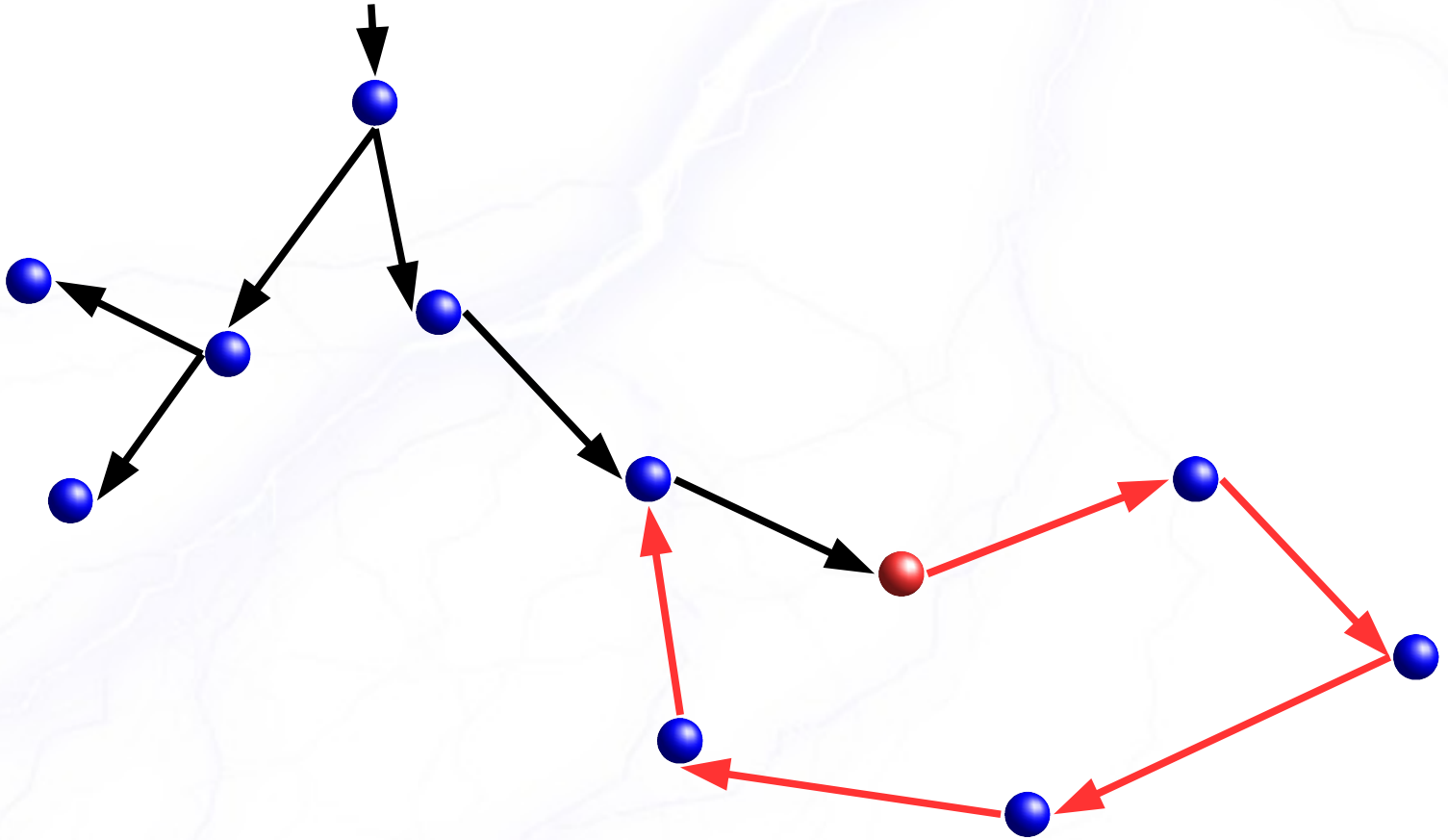
Промежуточные выводы

- Строим $\mathcal{L}(B_M) \cap \mathcal{L}(B\neg\varphi)$ и проверяем на пустоту
- Как видно из предыдущего примера, совсем необязательно строить полное пересечение
- Достаточно найти одно вычисление
- ...или убедиться, что нет ни одного вычисления

Поиск в... глубину

- Как найти бесконечное вычисление-контрпример или убедиться в том, что его нет?
- Автоматы B_M и $B_{\neg\varphi}$ конечны
- Достаточно найти достижимый цикл, содержащий допускающее состояние!
- Двойной поиск в глубину:
  - Первый уровень ищет допускающее состояние
  - Второй уровень ищет цикл из заданного допускающего состояния

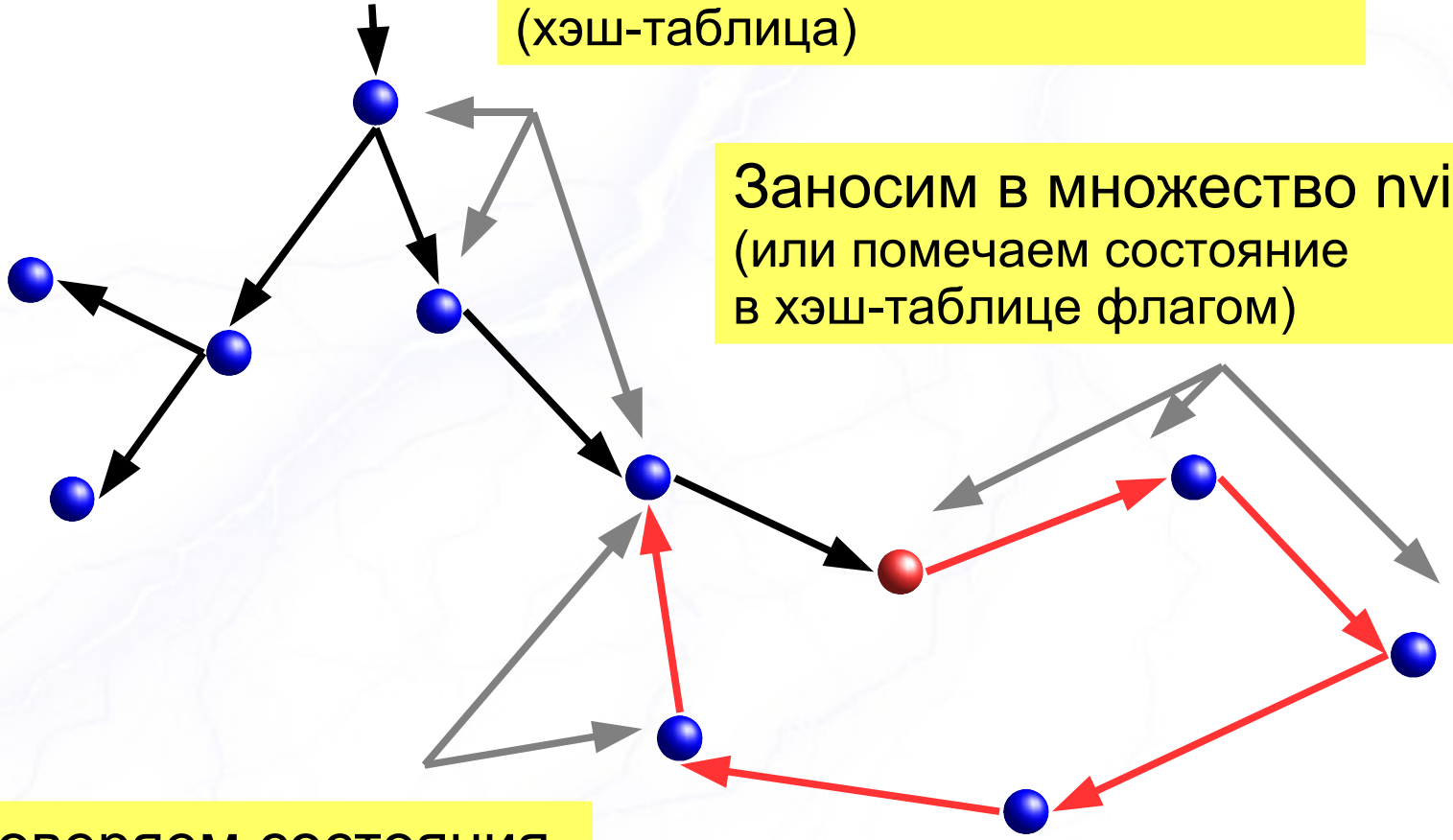
Двойной поиск в глубину



Организация поиска

Заносим в множество visited
(хэш-таблица)

Заносим в множество nvisited
(или помечаем состояние
в хэш-таблице флагом)



Проверяем состояния
в стеке поиска первого
уровня

Алгоритм проверки пустоты

```
procedure emptiness
   for all q0 in Q0 do
      dfs1(q0);
   terminate(false);
end procedure
```

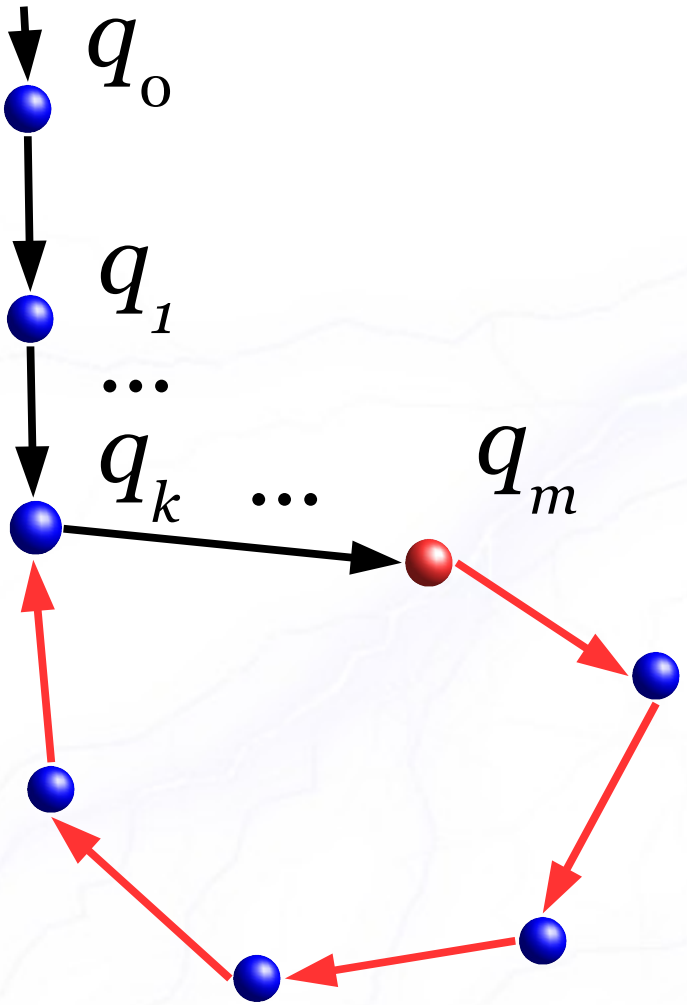
Внешний поиск

```
procedure dfs1(q)
  var q';
  hash_insert(q); // добавить q в visited
  for all q' in successors(q) do  // q -> q'
    if not hash_find(q')
      dfs1(q');
  if is_accepting(q)
    dfs2(q);
end procedure
```

Вложенный поиск

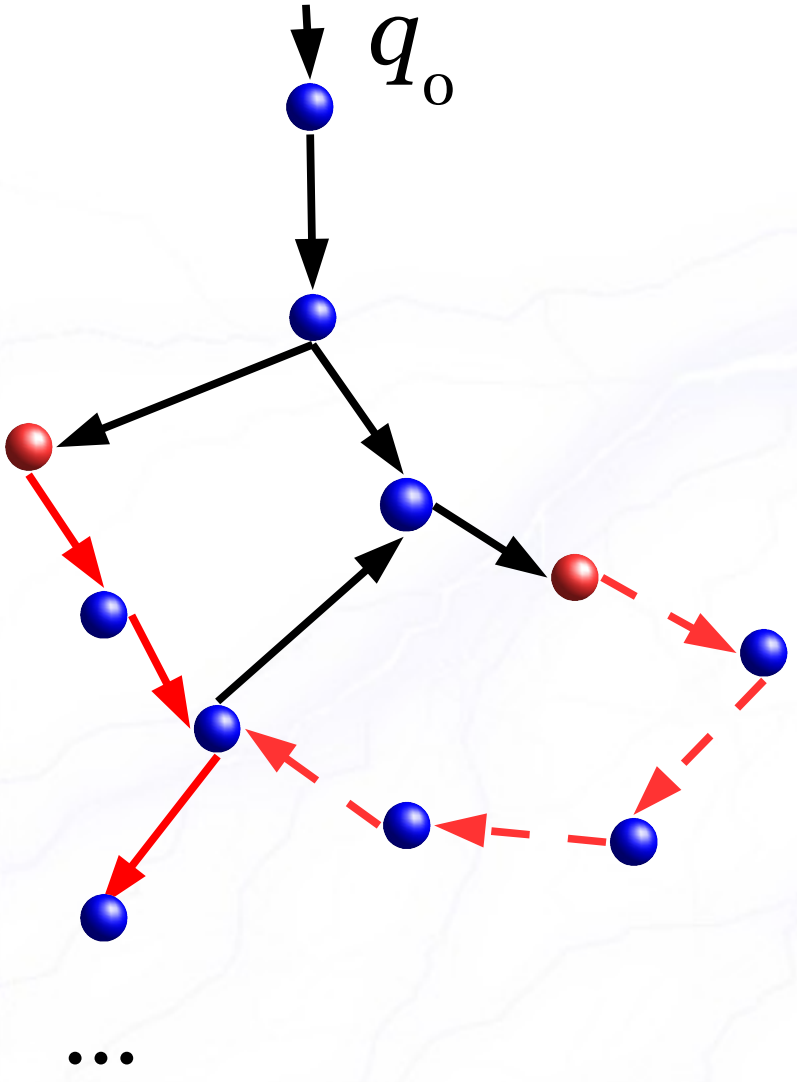
```
procedure dfs2(q)
  var q';
  flag(q); // пометить сост. q в хэш-таблице
  for all q' in successors(q) do  // q -> q'
    if q' in dfs1_stack
      terminate(true); // обнаружен цикл
    else if not has_flag(q')
      dfs2(q');
end procedure
```

Почему же алгоритм работает?



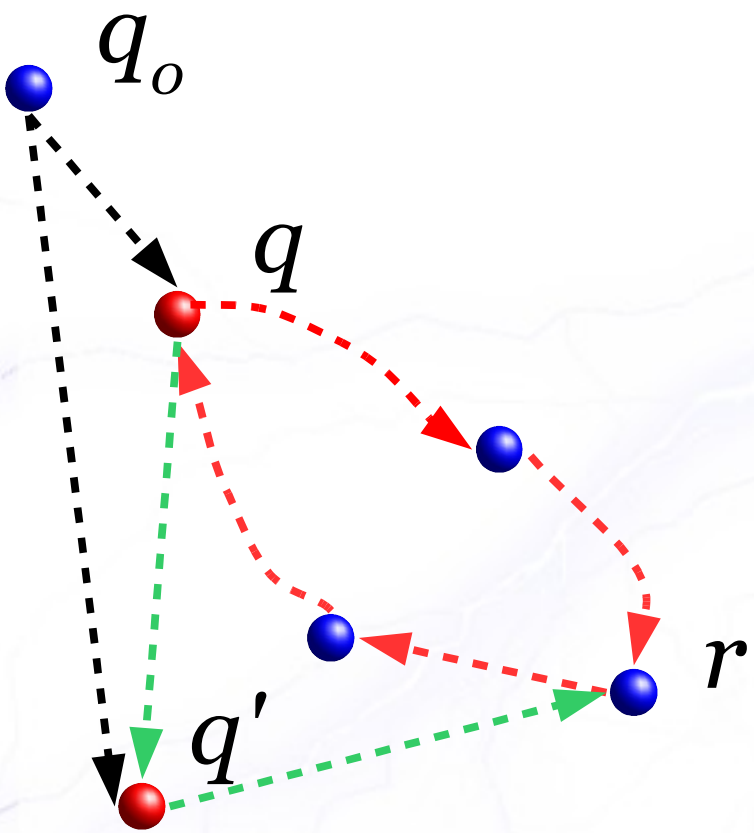
- Путь $q_0 \to q_1 \to \ldots \to q_k \to \ldots \to q_m \to \ldots \to q_k \to \ldots \to q_k \to \ldots$ принадлежит языку $\mathcal{L}(B_M \otimes B\neg\varphi)$
- Если цикл через допускающее состояние найден, то контрпример построен.

Можно ли пропустить цикл?

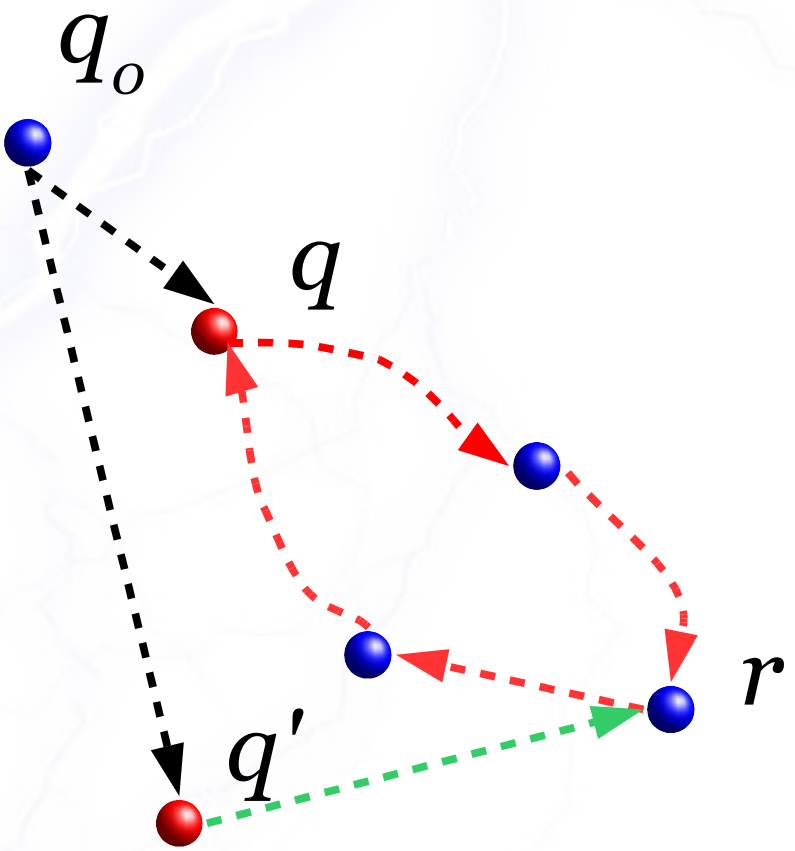


- Можно ли пропустить цикл, попав в состояние, помеченное ранее *dfs*2?
- Рассмотрим первый пропущенный цикл.

Варианты

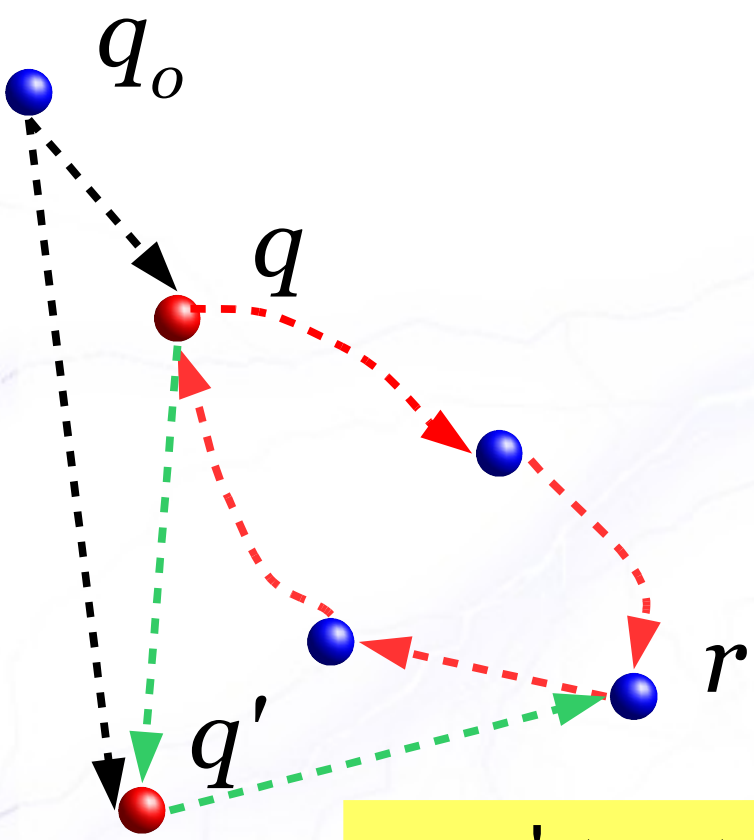


Вариант 1



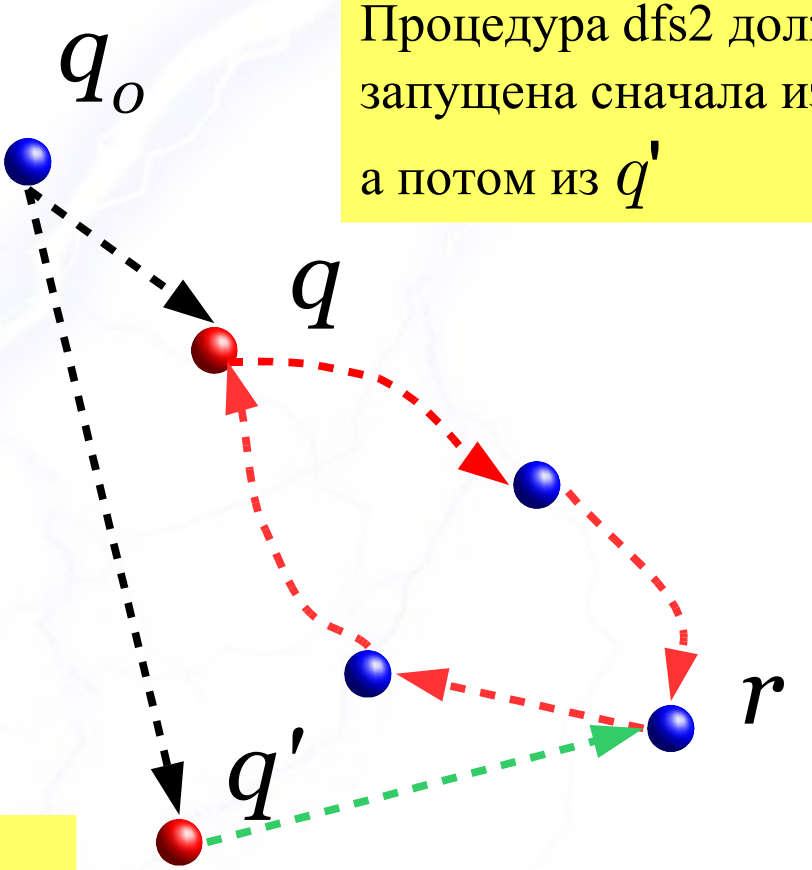
Вариант 2

Варианты



Цикл q'→…→r
→…→q→…
пропущен раньше

Вариант 1



Процедура dfs2 должна быть запущена сначала из q,
а потом из q'

Вариант 2

Корректность алгоритма

Теорема (Корректность). Алгоритм двойного поиска в глубину строит контрпример для задачи пустоты автомата Бюхи B тогда и только тогда, когда $\mathcal{L}(B)$ не пусто.

Сложность проверки

- Сложность: $O(n_1 * n_2)$, где n_1, n_2 – число состояний в автоматах B_M и $B\neg\varphi$.
- То есть сложность проверки формулы LTL на LTS M оценивается как $O((m+n) * 2^k)$, где m и n – число переходов и состояний в M, а k – число подформул в φ.

Подводные камни

- Не все так просто.
- Операции с множествами:
  - Вставка в *visited*, *nvisited* и *stack*
  - Проверка на принадлежность *visited*, *nvisited* и *stack*
- В оценке предполагается, что вставка и проверка на принадлежность занимает $O(1)$.

Ресурсы: время и память

- Храним *visited* и *stack* в хэш-таблицах (*nvisited* – дополнительный флаг)
- Операции *O*(1), но быстро заканчивается память
- Решение 1: сжатие структур, представляющих состояние (-DCOLLAPSE)
  - Режим упаковки вектора состояний; коэффициент сжатия от 80% до 90%
  - Всё равно может довольно быстро закончиться память

Ресурсы: время и память (2)

- Решение 2: хэш-таблица фиксированного размера с несколькими хэш-функциями (-DHC) (HASHCOMPACT):
  - Режим упаковки вектора состояний; отображает вектор состояний на число из 32+16 бит и сохраняется состояние в хэш-таблице (Wolper's hash-compact method).
  - Рано или поздно, придётся разрешать коллизии.

Ресурсы: время и память (3)

- Решение 3: множества представляются битовой шкалой без разрешения коллизий (-DBITSTATE):
  - Вместо полного перебора используются алгоритмы supertrace/bitstate.
  - Коллизии не разрешаются. Состояние считается просмотренным, если по вычисленному хэш-ключу установлен бит.
  - Аппроксимация!
  - Можно потерять вычисления.
  - Вычисляется оценка покрытия.

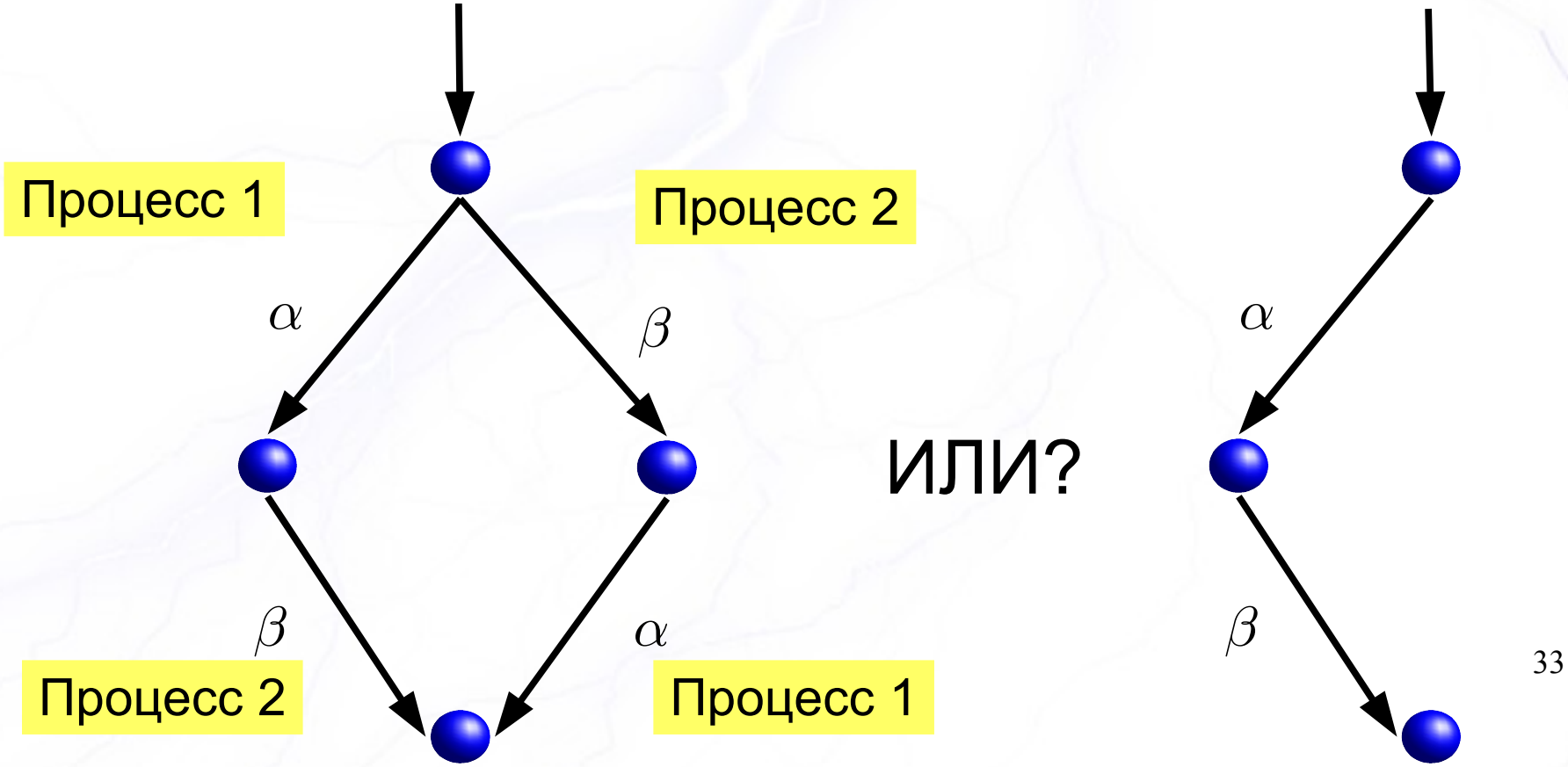
Другие способы представления множеств

- Если не хватает памяти:
  - можно пожертвовать временем,
  - не можем ждать годами,
  - нужен компромисс между потреблением времени и памяти.
- Об этом на следующей лекции...

Почему нужно так много ресурсов?

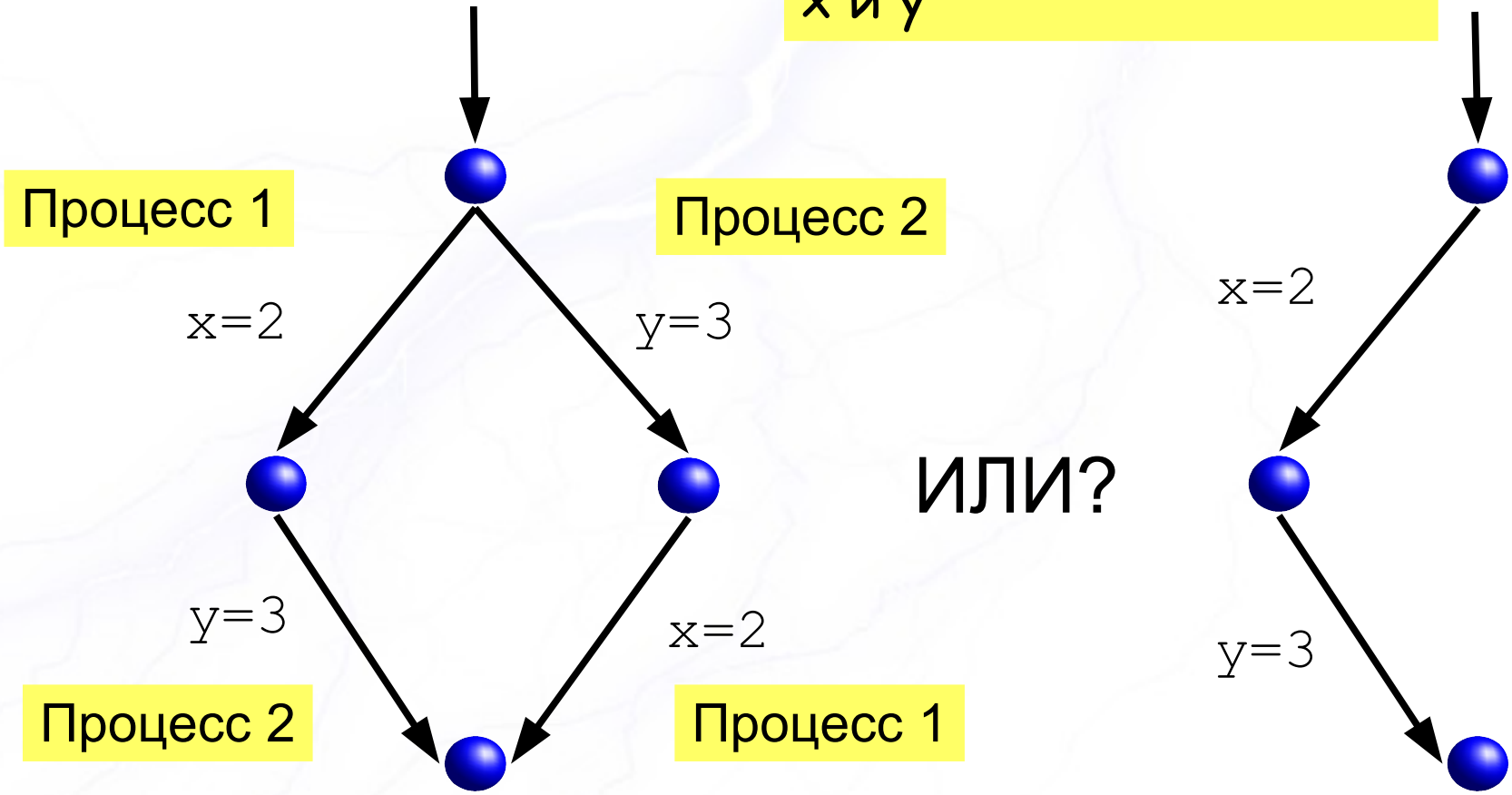
- Большие векторы состояний
- Комбинаторный взрыв:
  - необходимо рассмотреть всевозможные варианты поведения различных процессов
  - Всегда ли это необходимо?

Перебор состояний

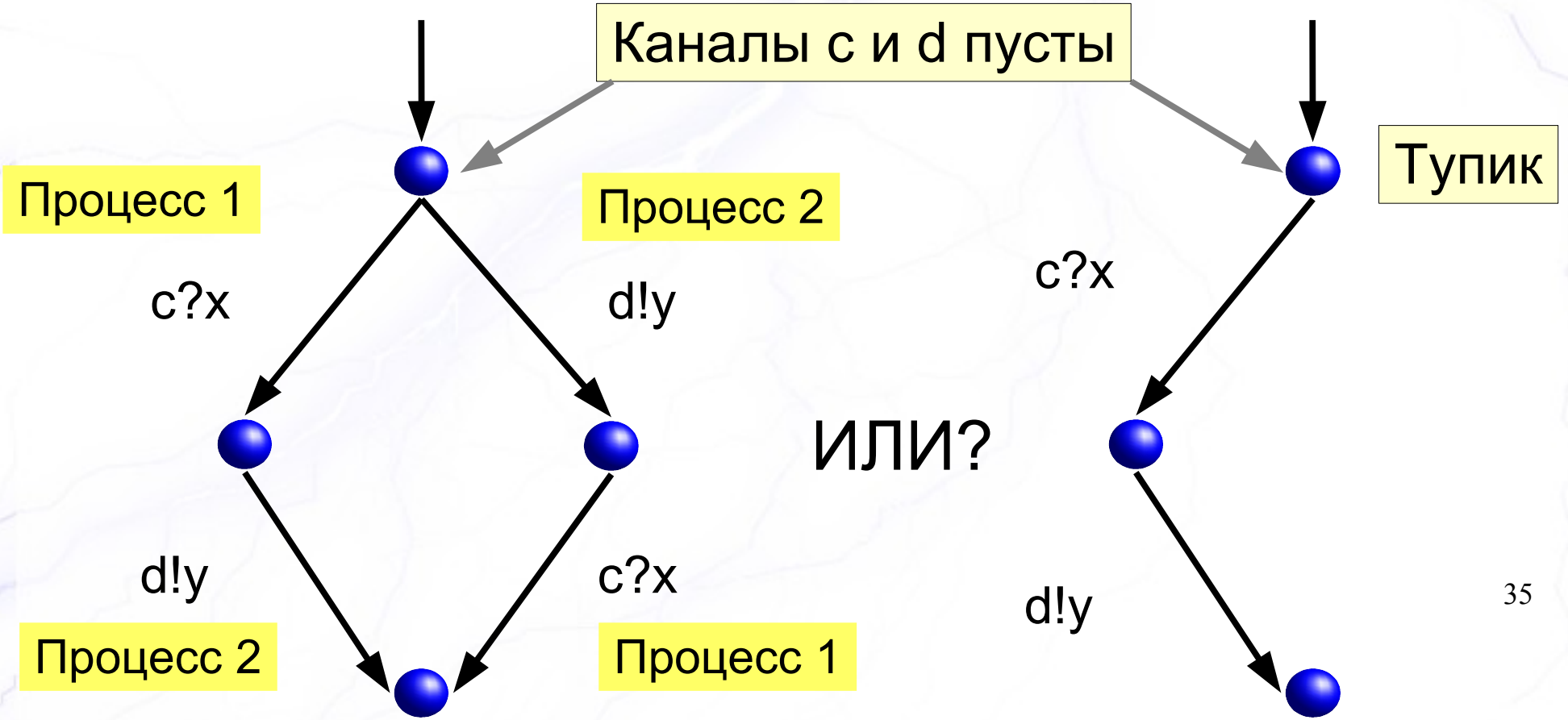


Перебор состояний

В спецификациях
не учитывается порядок
x и y



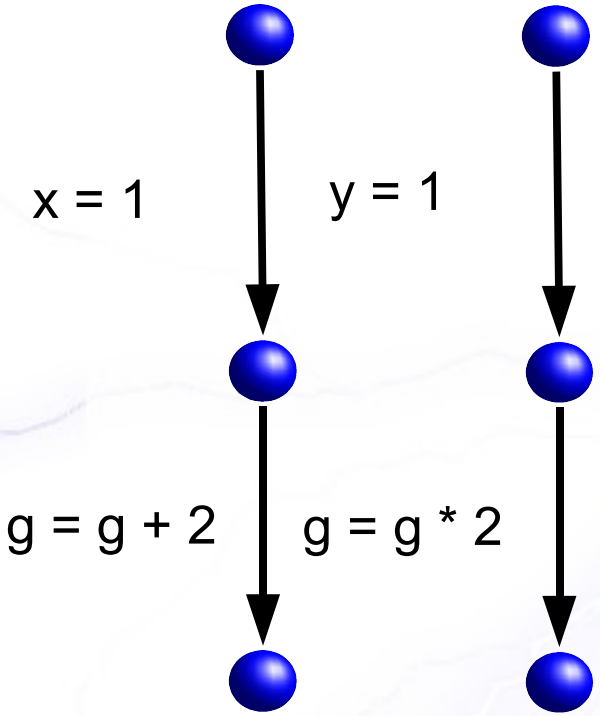
Такой приём не всегда корректно работает

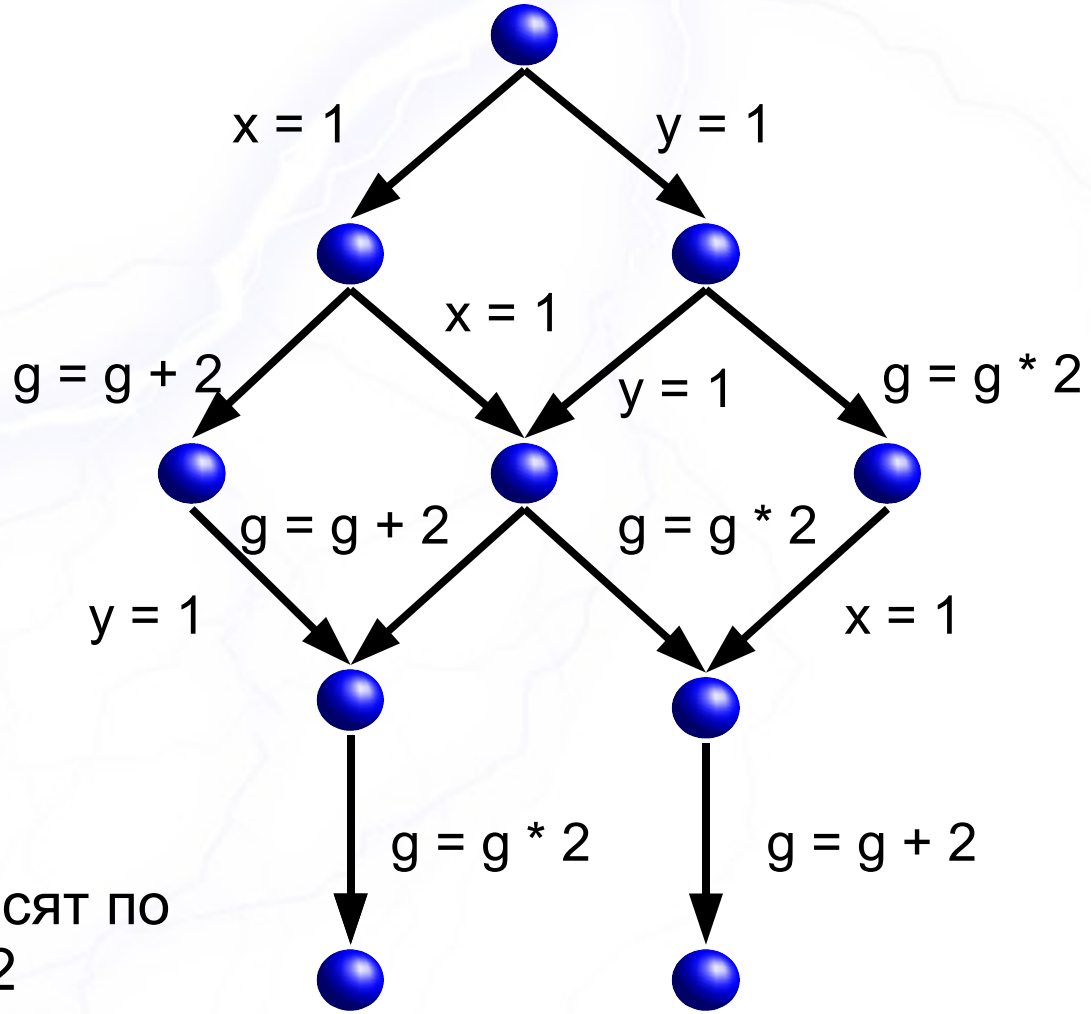


Редукция частичных порядков

- На множестве состояний композиции определяется частичный порядок $\leq$.
- Если $s_2 \leq s_1$, то переход $s_1 \rightarrow s_2$ можно пропустить при просмотре.
- *Доказывается*, что порядок $\leq$ сохраняет выполнимость (и не выполнимость) спецификации.
- Запрещается флагом -DNOREDUCE.
- Как выбрать порядок $\leq$? Эвристика.

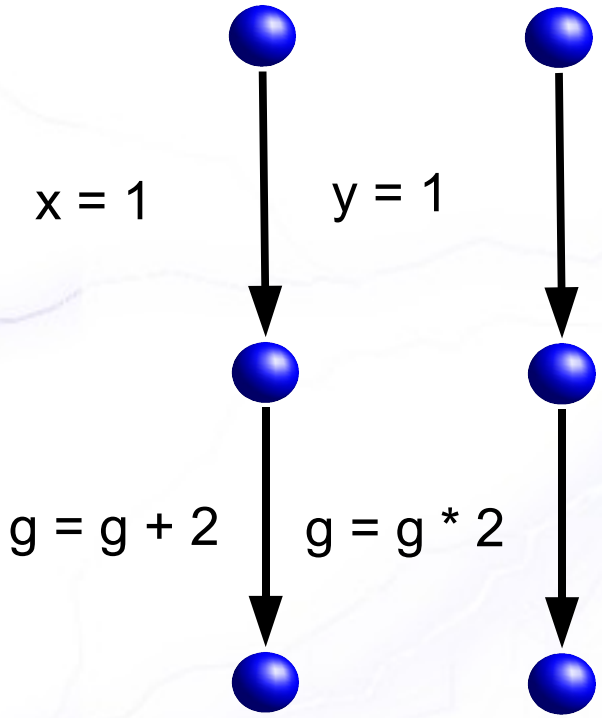
Зависимости





Только два оператора зависят по данным: g = g + 2 и g = g * 2
Остальные операторы независимы по данным!

Зависимость по данным и управлению



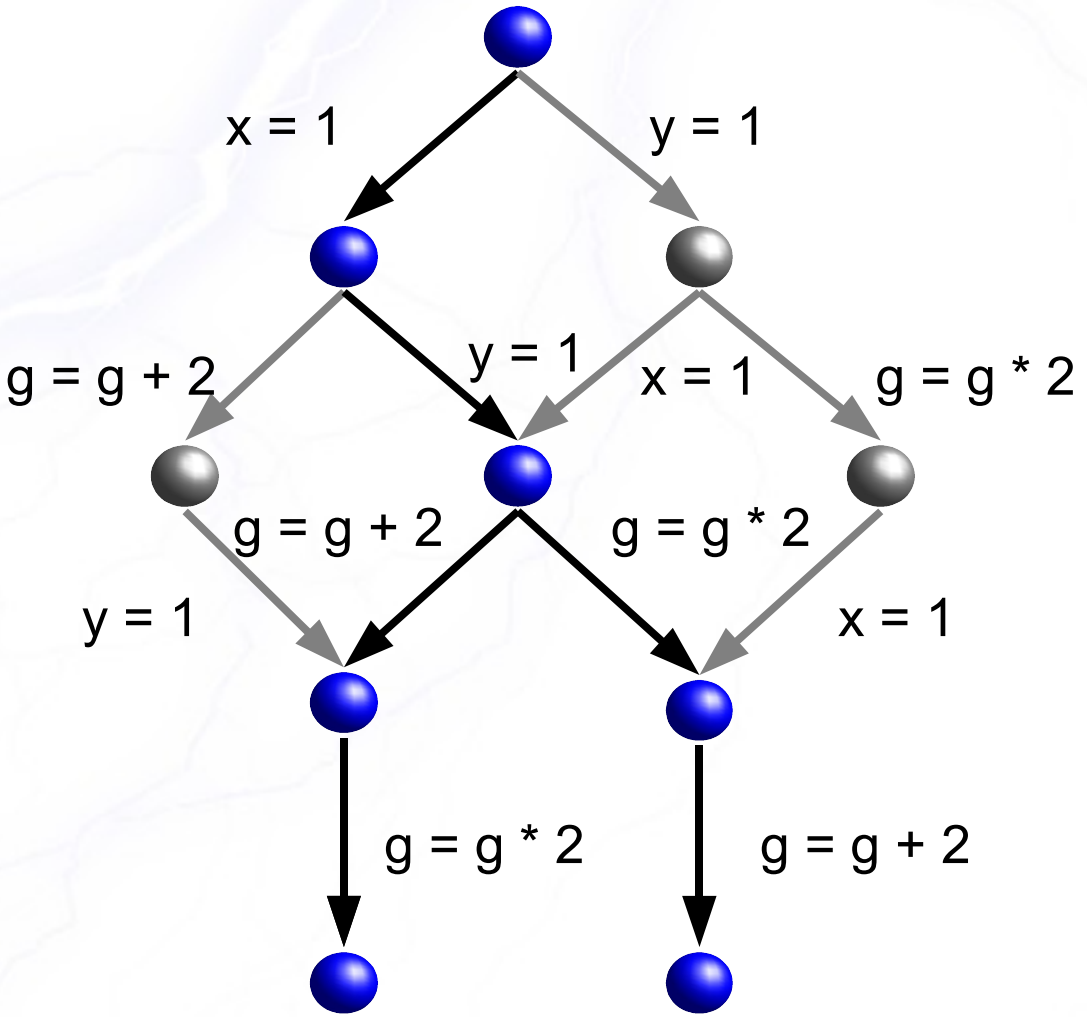
	x = 1	y = 1	g = g + 2	g = g * 2
x = 1		*I*	*Control*	*I*
y = 1	*I*		*I*	*Control*
g = g + 2	*Control*	*I*		*Data*
g = g * 2	*I*	*Control*	*Data*	

I – независимые операторы
Control – зависимость по управлению
Data – зависимость по данным

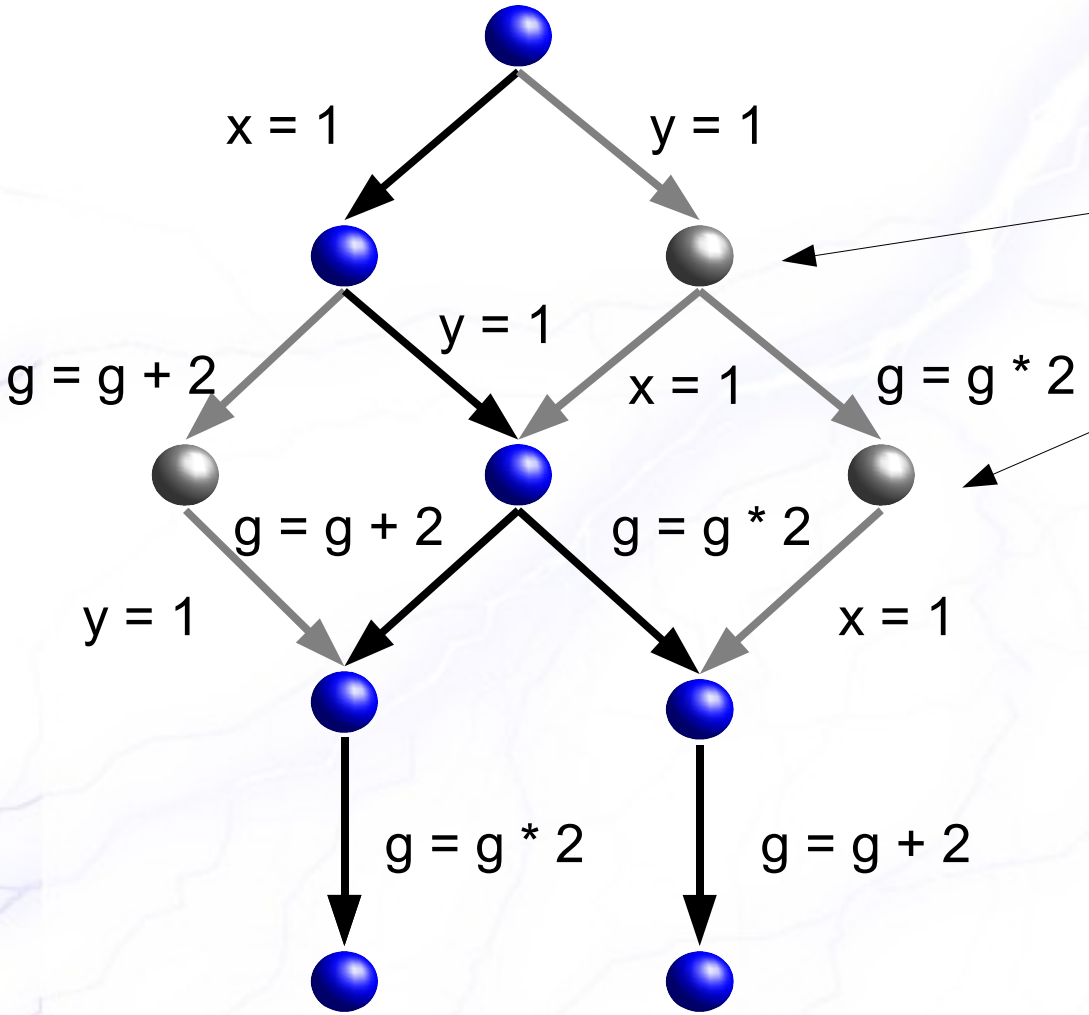
Использование зависимостей для редукции

Независимые операторы:

x =1 и y = 1
x = 1 и g = g * 2
y = 1 и g = g + 2



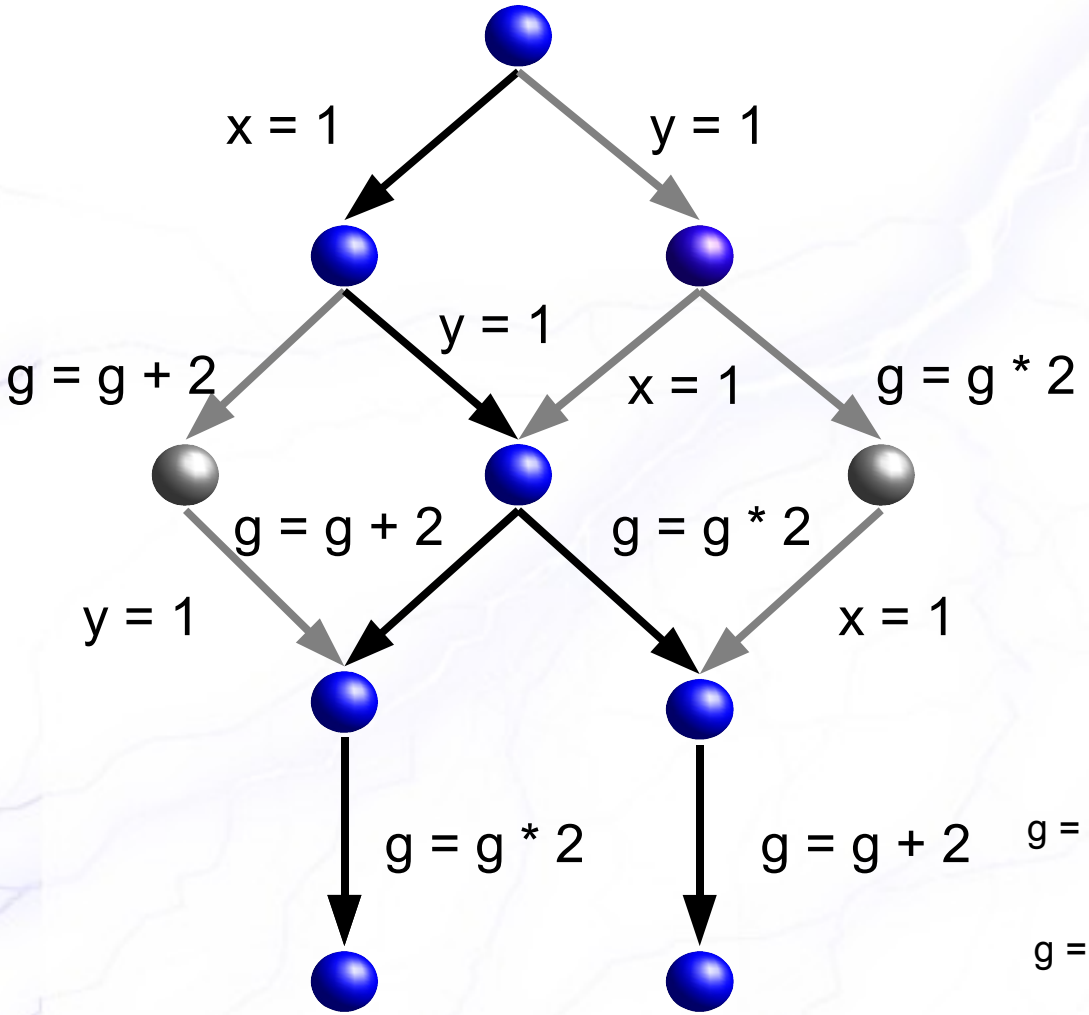
Наблюдаемые переменные



[] (x >= y)

Необходимо также учитывать зависимость по наблюдаемым переменным (visibility)!

Редукция



	x = 1	y = 1	g = g + 2	g = g * 2
x = 1		*P*	*Control*	*I*
y = 1	*P*		*I*	*Control*
g = g + 2	*Control*	*I*		*Data*
g = g * 2	*I*	*Control*	*Data*	

Правила редукции

Переходы t1 и t2 независимы в состоянии s:

- t1 ∈ enabled(s) и t2 ∈ enabled(s)
- t1 ∈ enabled(t2(s)) и t2 ∈ enabled(t1(s))
- Нет зависимости по данным и наблюдаемым свойствам

• Сильная независимость: переходы t1 и t2 сильно независимы, если они независимы в любом состоянии s: t1 ∈ enabled(s) и t2 ∈ enabled(s)

Правила редукции (2)

Безопасные переходы:

- Переход *t1* безопасен, если он независим от всех других переходов системы *(правило 1)*
- Оператор *i* условно безопасен для условия *c*, если он безопасен во всех состояниях, в которых выполняется условие *c (правило 2)*

Симуляция

- Как формально обосновать корректность редукции?
- Обычно для обоснования правил редукции используются отношения симуляции

Сильная симуляция

M1 = ($S1, Act, \rightarrow 1, s_{01}, AP, L1$)

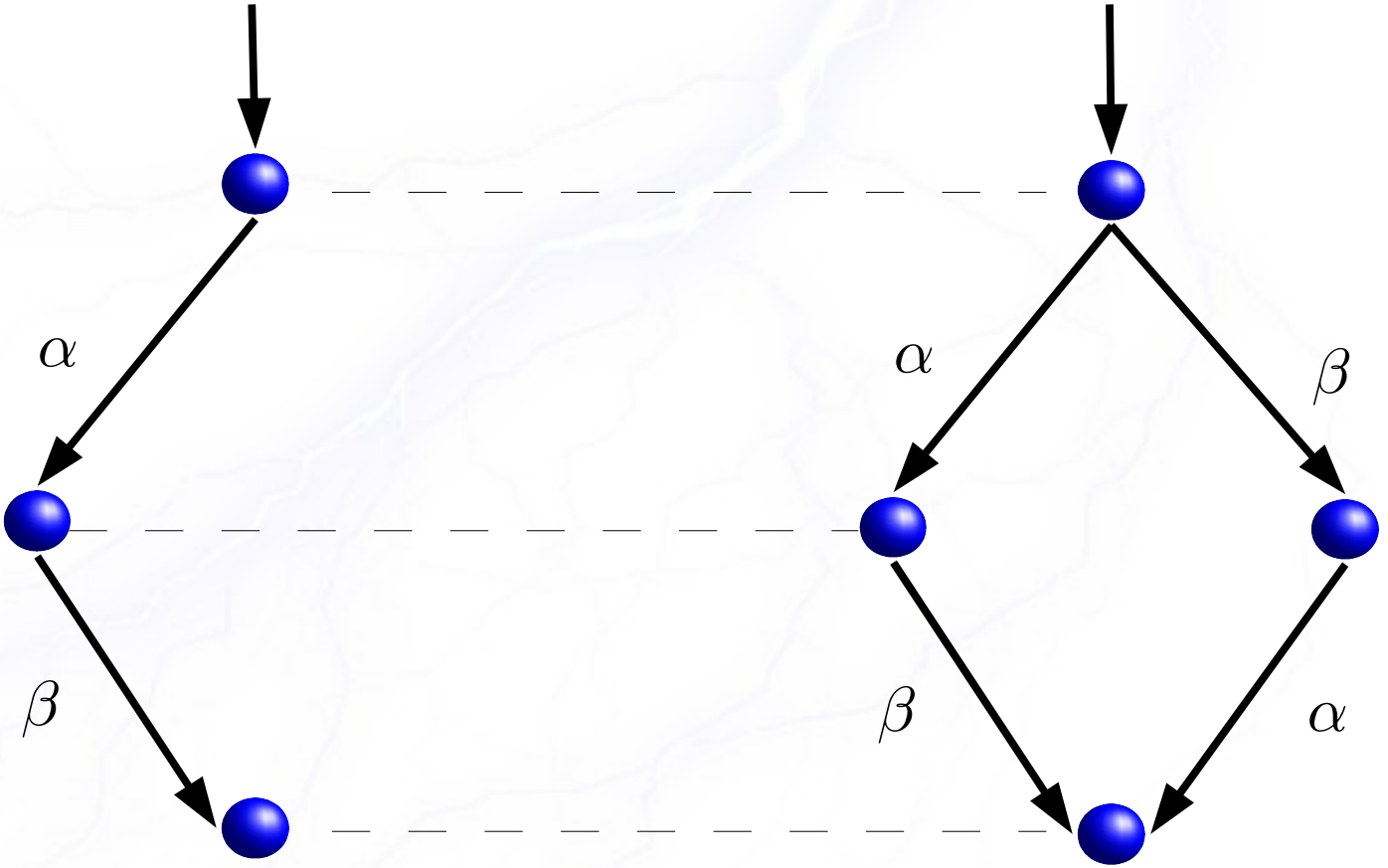
M2 =($S2, Act, \rightarrow 2, s_{02}, AP, L2$)

Отношение H $\in$ S_1 x S_2 называется симуляцией, если для любых $(s_1, s_2) \in H$:

для любого $(s_1, a, u_1) \in \rightarrow_1$, то найдётся такой переход $(s_2, a, u_2) \in \rightarrow_1$, в котором $(u_1, u_2) \in H$

$M_1 \leq M_2$, если найдётся H: $(s_{01}, s_{02}) \in H$

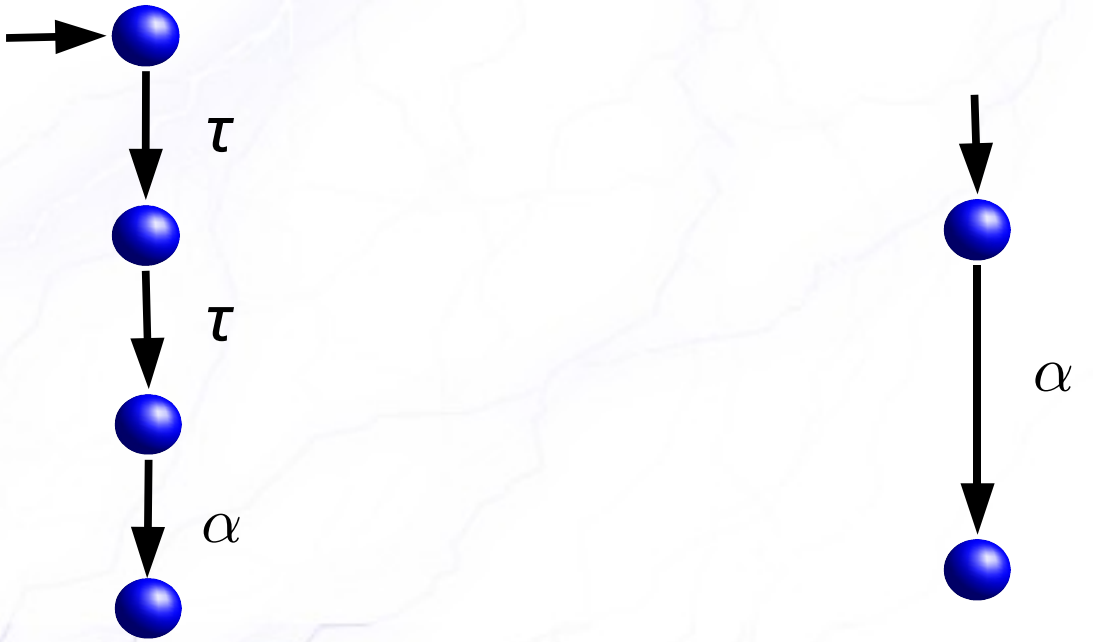
Сильная симуляция (пример)



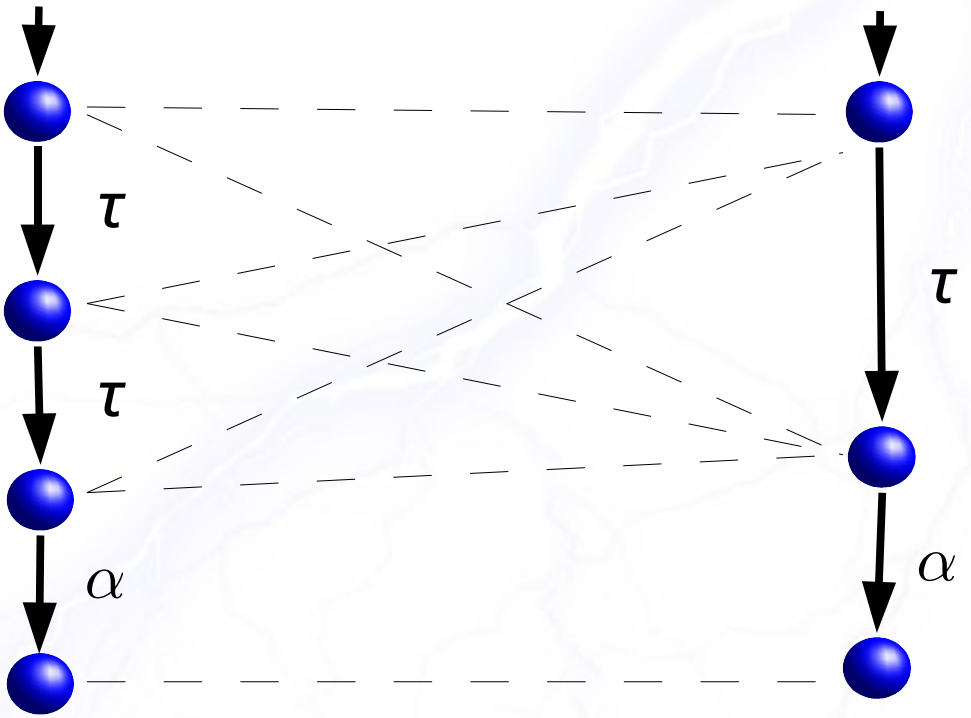
Свойства сильной симуляции

Если $M_1 \leq M_2$ и на M_2 выполняется φ из ACTL* (в частности LTL!), то на M_1 также выполняется φ.

Строгое соответствие переходов:



Слабая симуляция



Теряются свойства, **не инвариантные** к прореживанию
(LTL: оператор ne**X**t).

Спасибо за внимание!

При подготовке лекции использовались идеи из курсов лекций Дж. Хольцмана и Ю.Г. Карпова