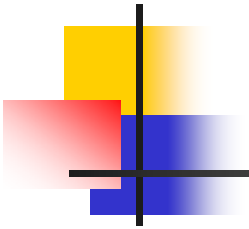
# Верификация параллельных программных и аппаратных систем

Курс лекций

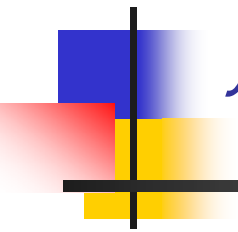
Карпов Юрий Глебович
профессор, д.т.н., зав.кафедрой
“Распределенные вычисления и компьютерные сети”
Санкт-Петербургского политехнического университета

karpov@dcn.infos.ru



# План курса

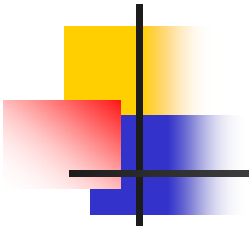
1. Введение
2. Метод Флойда-Хоара доказательства корректности программ
3. Исчисление взаимодействующих систем (CCS) Р.Милнера
4. Темпоральные логики
5. Алгоритм model checking для проверки формул CTL
6. Автоматный подход к проверке выполнения формул LTL
7. Структура Крипке как модель реагирующих систем
8. Темпоральные свойства систем
9. Система верификации Spin и язык Promela. Примеры верификации
10. Применения метода верификации model checking
11. BDD и их применение
12. Символьная проверка моделей
13. Количественный анализ дискретных систем при их верификации
14. Верификация систем реального времени ( I )
15. Верификация систем реального времени ( II )
16. Консультации по курсовой работе



# *Лекция 3*

*Алгебры процессов*

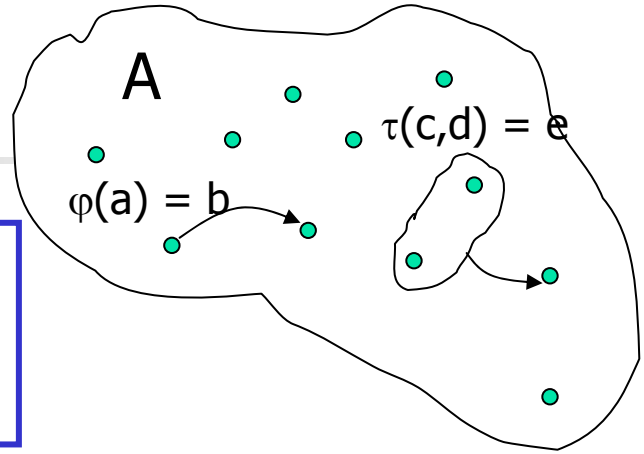
- *Calculus of Communicating Systems (CCS) Р.Милнера*
- *$\pi$ - исчисление – базовая модель мобильной коммуникации*
- *Communicated Sequential Processes (CSP) А.Хоара*



# CCS – Calculus of Communicating Systems

- Исчисление CCS (алгебра взаимодействующих процессов) было разработано by Robin Milner в 1980 (Эдинбургский университет)
- Алгебра процессов – это язык спецификации взаимодействующих процессов, функционирующих параллельно. Отражает только параллелизм и взаимодействие процессов по именованным каналам
- Попытка формализовать параллельные программы для их анализа (проверки свойств)
- Литература
  - Robin Milner. *Calculus of Communicating Systems*. Lecture Notes in Computer Science. v.91, 1980
  - Ю.Карпов. *Формальное описание и верификация протоколов на основе CCS*. Автоматика и вычислительная техника, N6, 1986

# Алгебра – что это такое?



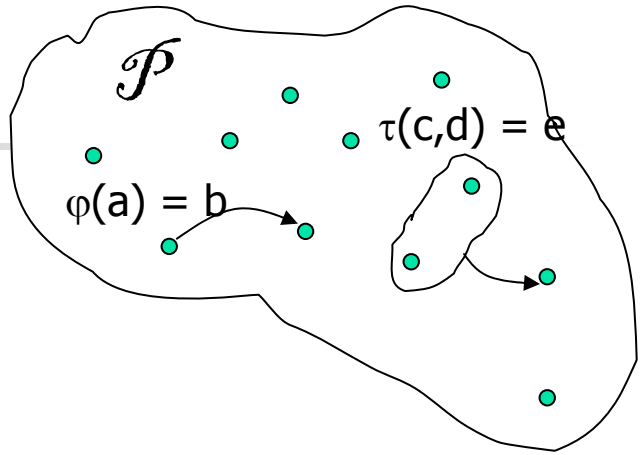

**A**=(A, Ω) – алгебра

A – носитель алгебры {a, b, c, ...)

Ω- конечное множество операций { φ, τ, ...} из A в A

- Алгебра – это множество объектов одной природы и набор операций над этими объектами, результат которых – объект того же множества
- Идея Р.Милнера: можно рассматривать параллельную композицию процессов как процесс. После выполнения действия процесс тоже остается процессом
- Т.е. операции || композиции и действия процессов дают в результате объект той же природы. Отсюда – и алгебра процессов
- Какие действия? С точки зрения внешнего наблюдателя - только внутренний переход и взаимодействие с окружением (другими процессами)
- Итак, рассматриваем процесс как активную сущность. Поведение процесса можно определить, как действие одного шага (его внутренний переход или внешняя коммуникация), после которого процесс становится другим процессом, который тоже может что-то дальше делать

# Алгебра процессов

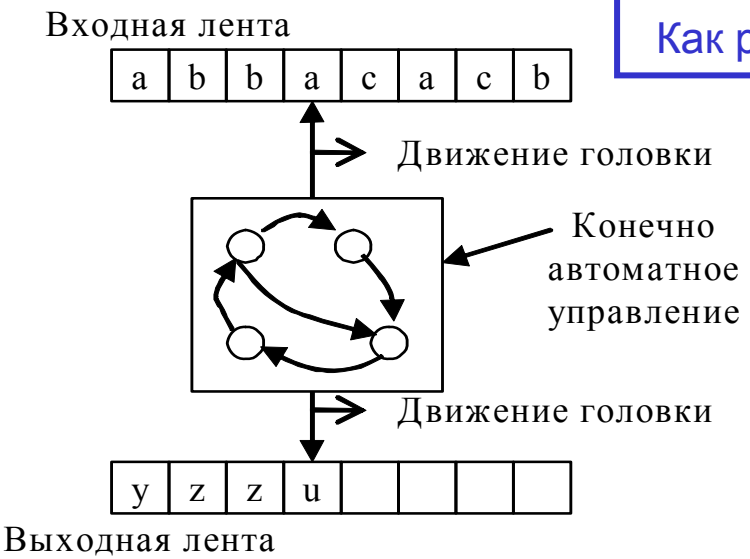



- Можно построить формулы, описывающие операции процессов И параллельное взаимодействие процессов, в результате этих действий снова получается процесс
- Следовательно, процессы имеют алгебраическую структуру, их можно рассматривать, как элементы некоторой Алгебры процессов
- Основная задача алгебры – это проверка эквивалентности: будут ли, например, эквивалентны результаты при всех a и b

  в Булевой алгебре: эквивалентны ли $a \Rightarrow b$ и $\neg a \vee b$ ???
- У процессов, описанных аналитически, можно исследовать свойства
  - как и во всякой алгебре, можно ввести понятие эквивалентности: когда два совершенно разные выражения P и Q, описывающие процессы, будут эквивалентны?
  - Как определить понятие эквивалентности для алгебры процессов?
- $P \approx Q$ *если С ТОЧКИ ЗРЕНИЯ ВНЕШНЕГО НАБЛЮДАТЕЛЯ процессы P и Q ведут себя одинаково*

# Конечные автоматы и их взаимодействие

**Конечный автомат**




$A=(S,X,Y,s0,\delta)$

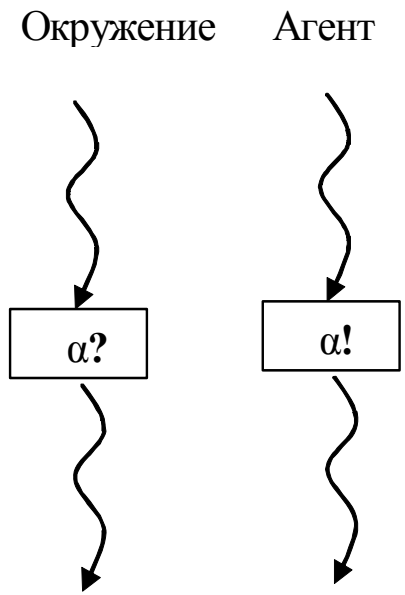
$\delta: S\times X \rightarrow S\times Y$

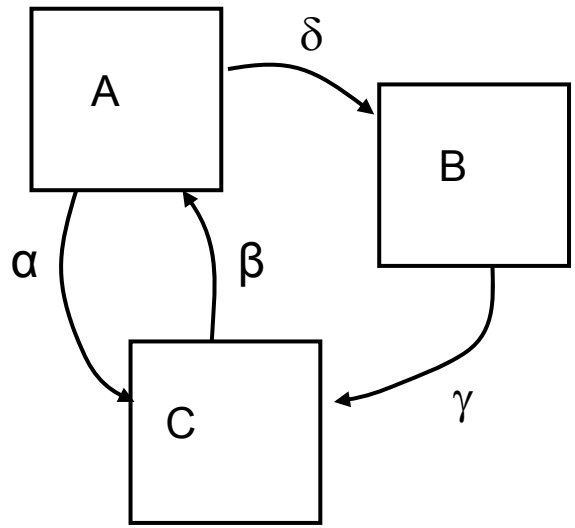
**Программа**:

1. $(s,a)\rightarrow(p,y)$;
2. $(q,b)\rightarrow(s,z)$;
3. …

Как определить ВЗАИМОДЕЙСТВИЕ автоматов?

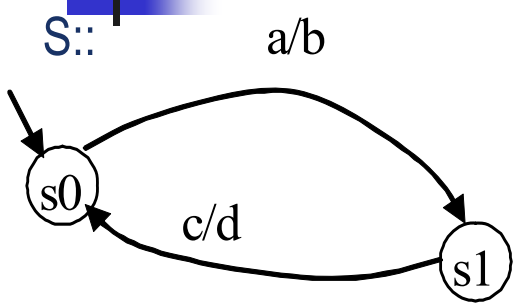
Как реализовать и описать синхронизацию процессов?

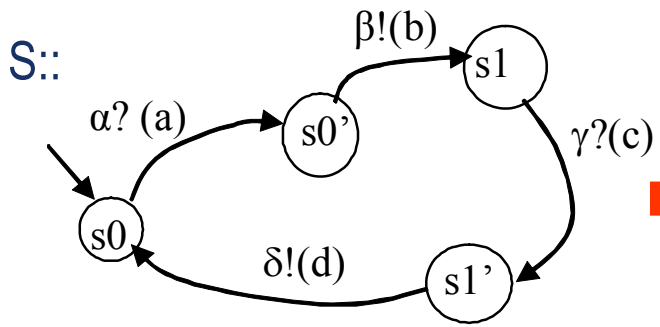



Милнер предложил использовать взаимодействие вместе с синхронизацией – рандеву - по именованным каналам

# Конечные автоматы как процессы



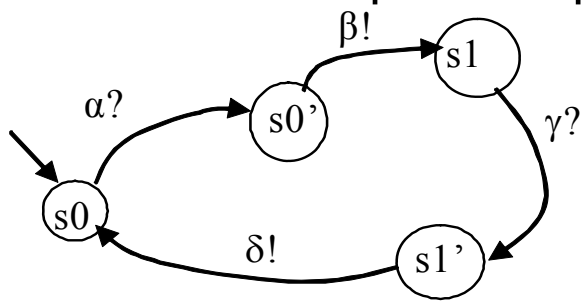

s0 = α?(a).s0';

s0'= β!(b).s1;

s1 = γ? (c).s1';

s1' = δ!(d).s0

Для простоты можно рассматривать только синхронизацию:



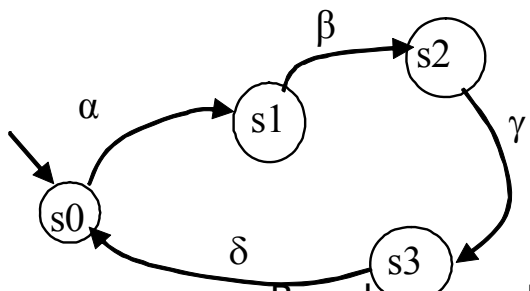

s0 = α?.s0';

s0'= β!.s1;

s1 = γ?.s1';

s1' = δ!.s0

Посылка и прием ничем не отличаются с точки зрения синхронизации:



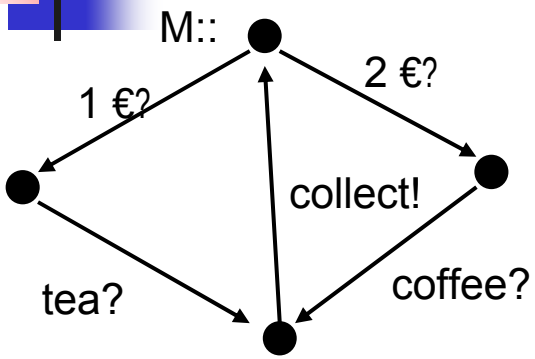

s0 = α.s1;

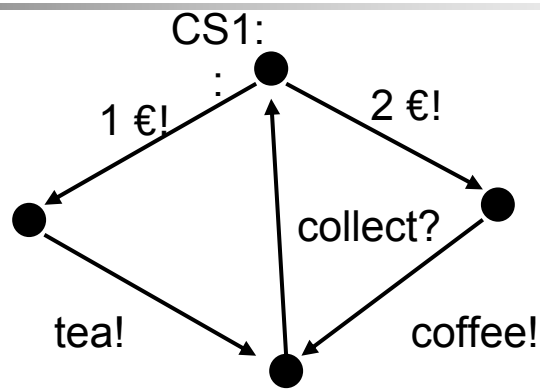
s1= β.s2;

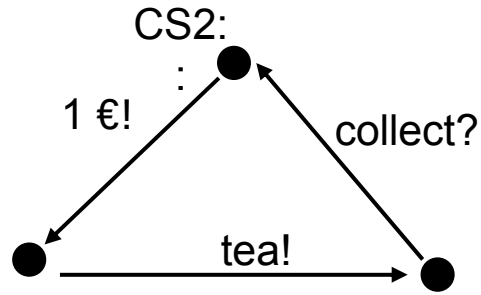
s2 = γ.s3;

s3 = δ.s0

# Пример: Coffee machine




Машина:

Ждет монету, ждет сигнала, что покупатель готов, и выдает напиток

Ученый (Computer Scientist, CS1 и CS2):

Опускает монету, нажимает на кнопку (чай или кофе) и забирает напиток

Машина продает кофе и чай:

M = (1€?. tea? + 2€?.coffee?).collect!.M

Некоторые ученые пьют и то, и другое, некоторые – только что-нибудь одно:

CS1 = (1€!. tea! + 2€!.coffee!).collect?.CS1

CS2 = 1€ !.tea!.collect?.CS2

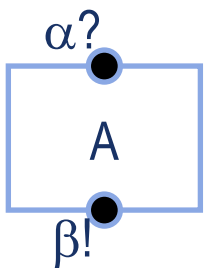
Вся система параллельных процессов :

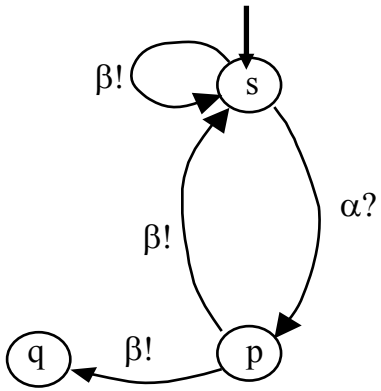
M | CS1 | CS2

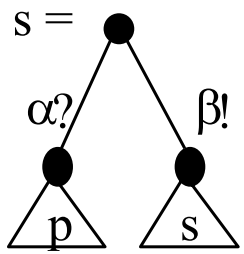
CCS - это алгебраический язык для описания параллельных процессов (а не графический, как, например, Сети Петри)

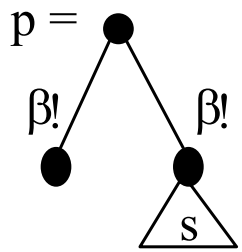
# Алгебра взаимодействующих процессов

Объект (агент) – описываем динамикой его поведения. Агент – это, фактически, будущее поведение процесса, находящегося сейчас в конкретном состоянии

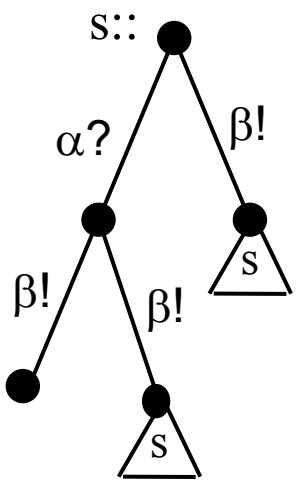


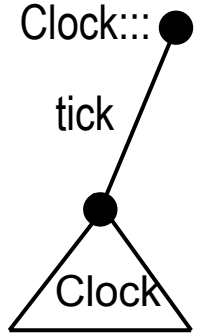




s = α?.p+β!.s;
p = β!.s + β!.q;
q = NIL




s= α? .(β!s + + β!.NIL) + β!s;




Clock = tick. Clock

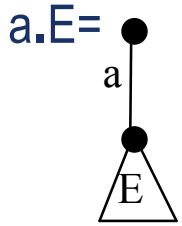
Clock = tick. Clock
= tick. tick. Clock
= tick. tick. tick. Clock

# Операции алгебры процессов

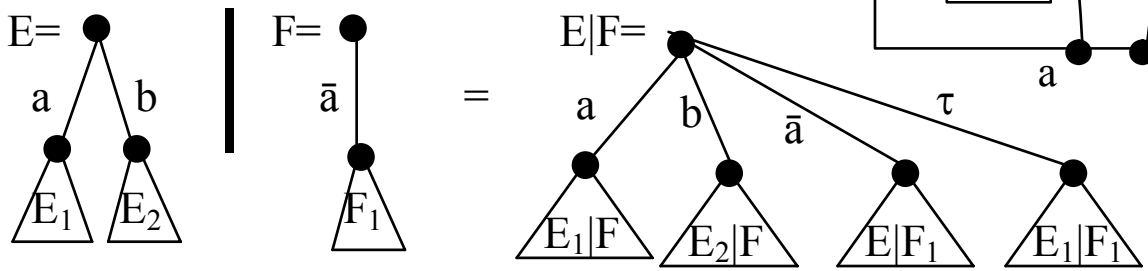
Вводим множество имен A (портов взаимодействия) и коимен $\bar{A}$

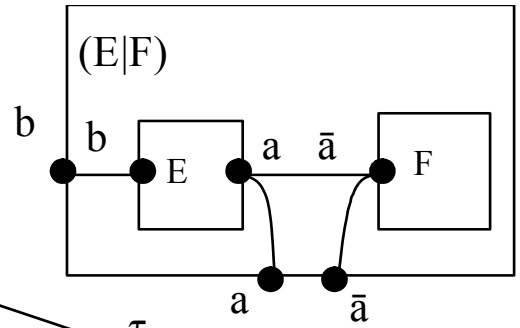
Префикс:

a.E для $a \in A \cup \bar{A} \cup \{\tau\}$

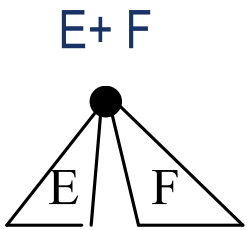



Параллельная композиция: E | F

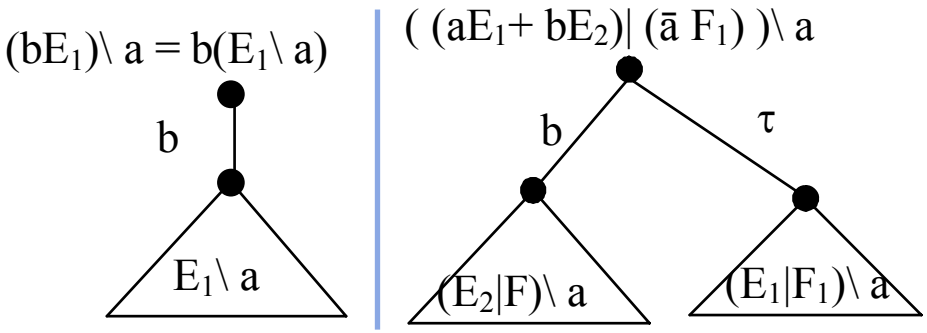


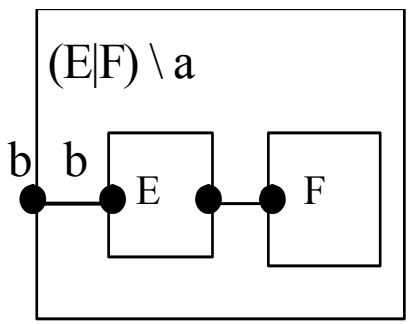




Сумма: E+ F




Ограничение: E \ L




Nil – константа

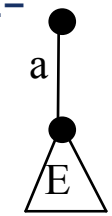
Переименование: E $[a_1/b_1, a_2/b_2, \ldots, a_n/b_n]$

# Семантика операций алгебры процессов

Действие: a E

$a.E \xrightarrow{a} E$

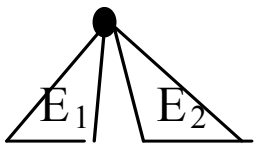
a.E=




Сумма: $E_1 + E_2$

$$\frac{E_i \xrightarrow{a} E,\ i \in I}{\Sigma_{i \in I} E_i \xrightarrow{a} E}$$

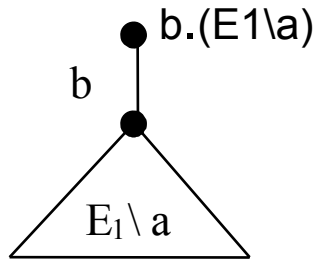
$E_1 + E_2$




Ограничение: $E \setminus L$

$$\frac{E \xrightarrow{a} E_i ;\ a \notin L}{E \setminus L \xrightarrow{a} E_i \setminus L}$$

$(b.E_1) \setminus a = b.(E_1 \setminus a)$



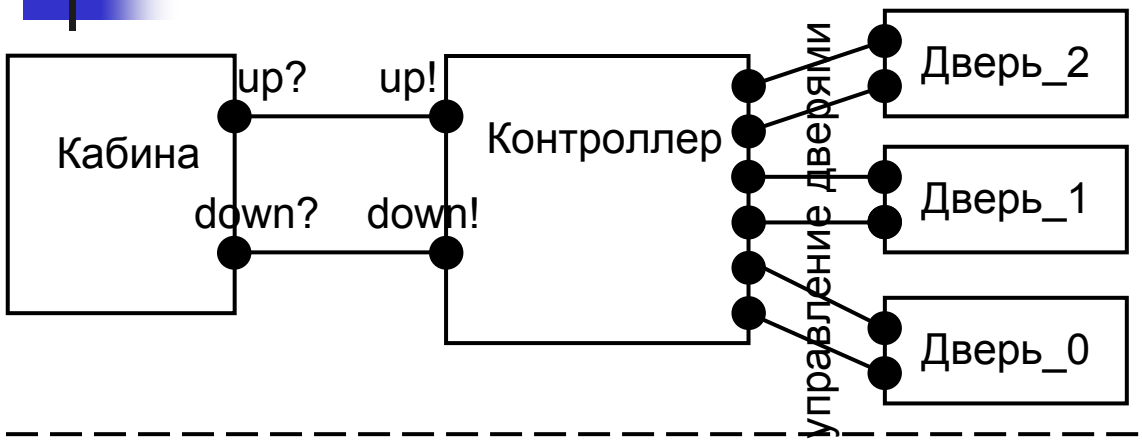

Параллельная композиция: E | F

$$\frac{E \xrightarrow{a} E_i}{E \mid F \xrightarrow{a} E_i \mid F}$$

$$\frac{F \xrightarrow{a} F_i}{E \mid F \xrightarrow{a} E \mid F_i}$$

$$\frac{E \xrightarrow{a} E_i ;\ F \xrightarrow{\bar{a}} F_k}{E \mid F \xrightarrow{\tau} E_i \mid F_k}$$

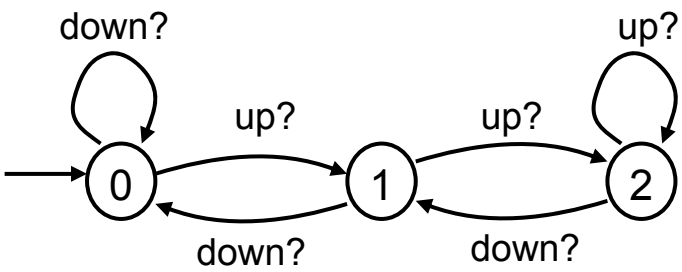
# Пример: Система управления лифтом



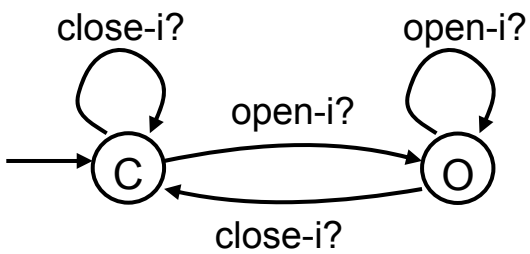

Представив поведение каждого процесса в алгебре CCS, мы можем проверить свойства всей системы

Кабина | Контроллер | Дверь_0 | Дверь_1 | Дверь_2

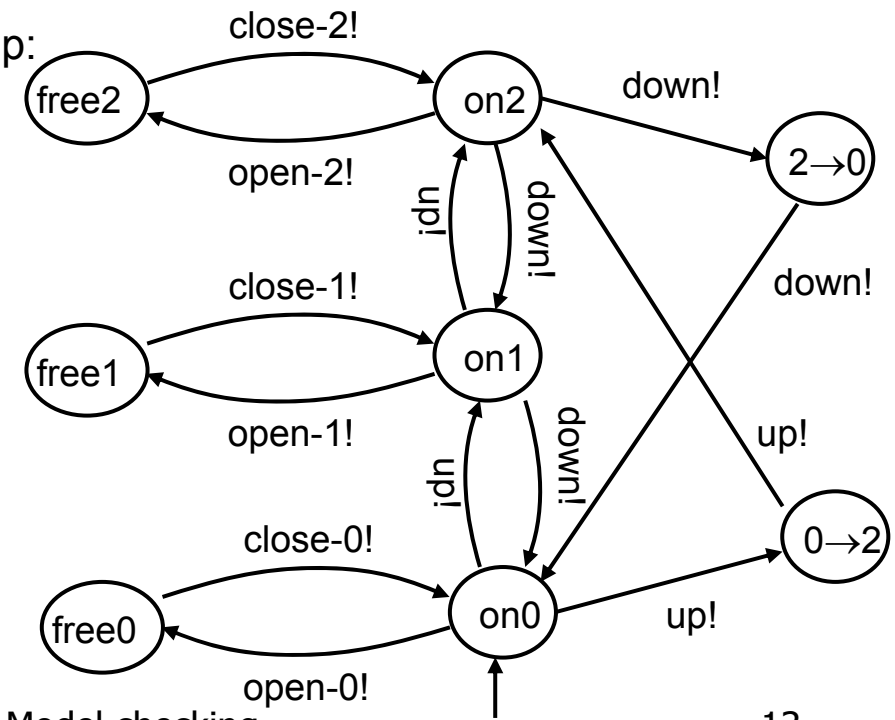
Кабина:



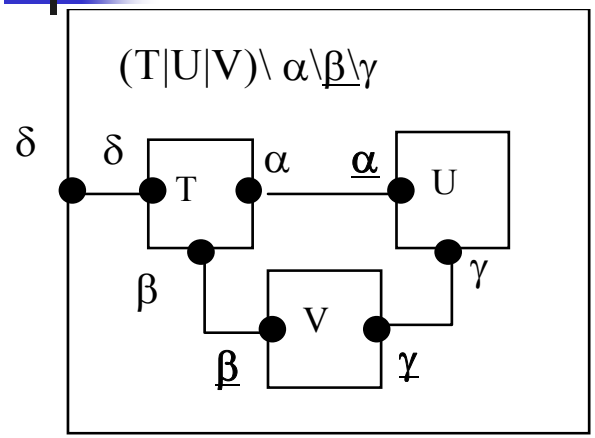

Двери:




Контроллер:



# Пример




$T = \alpha!t + \beta!t_1 + \delta?t_2;$
$U = \alpha?u + \gamma!u_1;$
$V = \beta?v + \gamma?v$

В языке PROMELA:

a?x – прием сообщения, посланного по каналу a, в переменную x

a!v – посылка сообщения v по каналу a

Поведение всей системы

$(T \mid U \mid V)\backslash \alpha \backslash \beta \backslash \gamma =$

$(\ (\alpha!t + \beta!t_1 + \delta?t_2) \mid (\alpha?u+\gamma!u_1) \mid (\beta?v+\gamma?v_1)\ )\backslash \alpha \backslash \beta \backslash \gamma =$

$\delta?(\ t_2 \mid (\underline{\alpha}u+\gamma u_1) \mid (\underline{\beta}v+\gamma v_1)\ )\backslash \alpha \backslash \beta \backslash \gamma$ ( действие $\delta$)

$+ \tau\ (\ t \mid u \mid (\underline{\beta}v+\gamma v_1)\ ) \backslash \alpha \backslash \beta \backslash \gamma$ (взаимодействие по порту $\alpha$)

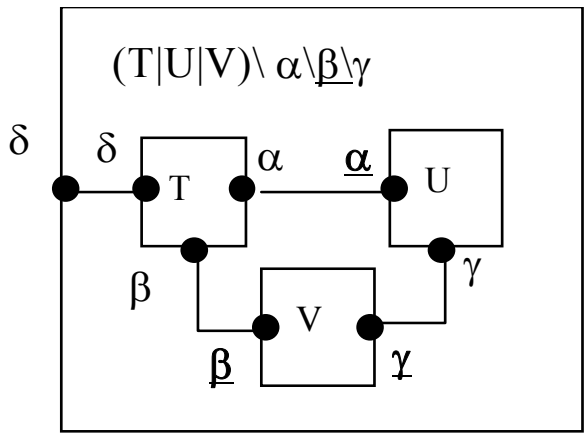
$+ \tau\ (\ t_1 \mid (\underline{\alpha}u+\gamma u_1) \mid v\ )\backslash \alpha \backslash \beta \backslash \gamma$ (взаимодействие по порту $\beta$)

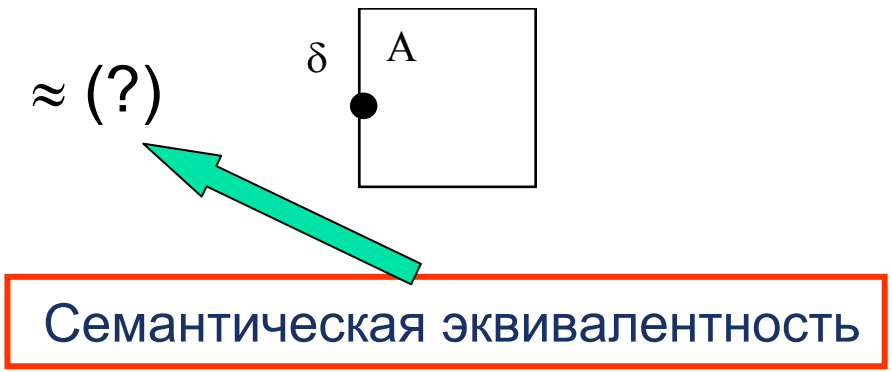
$+ \tau\ ((\alpha t + \beta t_1 + \delta t_2) \mid u_1 \mid v_1)\backslash \alpha \backslash \beta \backslash \gamma$ (взаимодействие по порту $\gamma$)

# Основная идея верификации в CCS

Система –
параллельная композиция процессов

Проверяемое свойство
системы



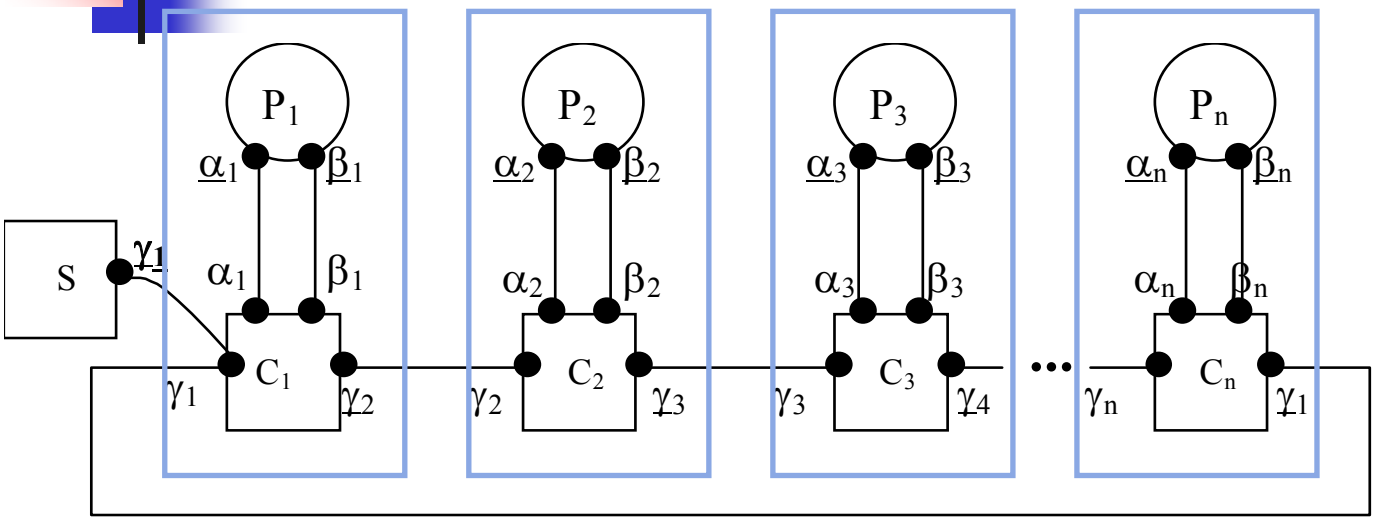




( T | U | V )\ α \ β \ γ ≈(?) A

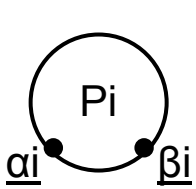
Эквивалентность понимается как невозможность для внешнего наблюдателя различить поведения двух процессов

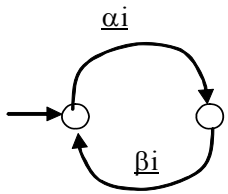
Для этого в алгебре процессов определяется отношение бисимуляции (наблюдаемой эквивалентности)

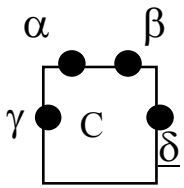
# Пример: планировщик процессов

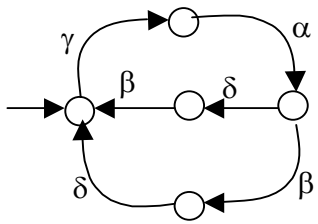



Время не рассматриваем, процессы Pi могут закончиться в произвольном порядке




**Рабочий процесс Pi:** $Pi=\underline{\alpha i}.\underline{\beta i}.Pi;$

ожидание запуска, останов

**Процесс Ci:** $Ci=\{\gamma_{i+1}/\ \beta_i\}\ \gamma i.\alpha i.(\underline{\beta i}.\delta i.Ci+\ \delta i.\underline{\beta i}.Ci);$

ожидание своей очереди, запуск процесса, (ожидание останова процесса и передача информации по цепочке) в любом порядке

**Стартер:** $S = \underline{\gamma_1}.NIL$

**Планировщик:** $Sch=(S|C_1|C_2|...|C_n)\backslash\ \gamma_i\backslash...\backslash\ \gamma_n$

# Планировщик процессов (требования)

**Требования на последовательность сигналов синхронизации в системе:**

1. *Процессы Pi запускаются поочередно* – если синхронизирующую последовательность спроецировать на алфавит $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$, то она станет $(\alpha_1\alpha_2\dots\alpha_n)^\omega$

2. *Каждый процесс Pi может быть запущен повторно только после того, как он завершился* – если синхронизирующую последовательность спроецировать на $\{\alpha_i, \beta_i\}$, то она станет $(\alpha_i\,\beta_i)^\omega$

**Как проверить эти требования?**

Проверка требований осуществляется доказательством эквивалентности двух поведений:
(I) параллельной композиции поведений заданных процессов И
(ii) требуемого (желаемого) поведения:

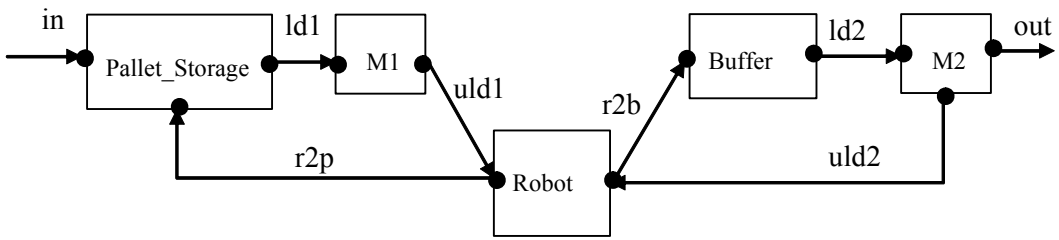
1. $Sch \,||\, (\beta_1^\omega \,|\, \dots \,|\, \beta_n^\omega) \approx? (\alpha_1\alpha_2\dots\alpha_n)^\omega$ - проверка первого требования
2. $Sch \,||\, (\Pi_{k\neq i}\,\alpha_k^\omega \,|\, \Pi_{k\neq i}\,\beta_k^\omega) \approx? (\alpha_i\,\beta_i)^\omega$ - проверка второго требования

Какое отношение эквивалентности имеется в виду?

Как проверить, что два произвольных выражения в алгебре процессов эквивалентны?

# Пример – manufacturing system

Производственная система состоит из 5 агентов:




## Коммуникации:

in – новая заготовка входит в систему

ld1- загрузка в M1

uld1- разгрузка M1

r2b – продукт из руки робота попадает в буфер

r2p – помещение паллеты из руки робота в хранилище

## Спецификации:

Pallet_Storage=in.ld1.r2p.Pallet_Storage

M1=ld1.uld1.M1

Robot = uld1.r2b.Robot+uld2.r2p.Robot

Buffer = r2b.ld2.Buffer

M2 = ld2.out.uld2.M2

Pallet_Storage2 = Pallet_Storage| Pallet_Storage

Pallet_Storage3 = Pallet_Storage2 | Pallet_Storage

Internals = {ld1, uld1,ld2, uld2, r2b, r2p}

Sys = (Pallet_Storage|M1|Robot|Buffer|M2)\Internals

Sys2 = (Pallet_Storage2|M1|Robot|Buffer|M2)\Internals

Sys3 = (Pallet_Storage3 |M1|Robot|Buffer|M2)\Internals

## Внешняя спецификация:

Spec = in.out.Spec

Spec2 =in.Spec2'

Spec2' = out.Spec2 +in.out.Spec2

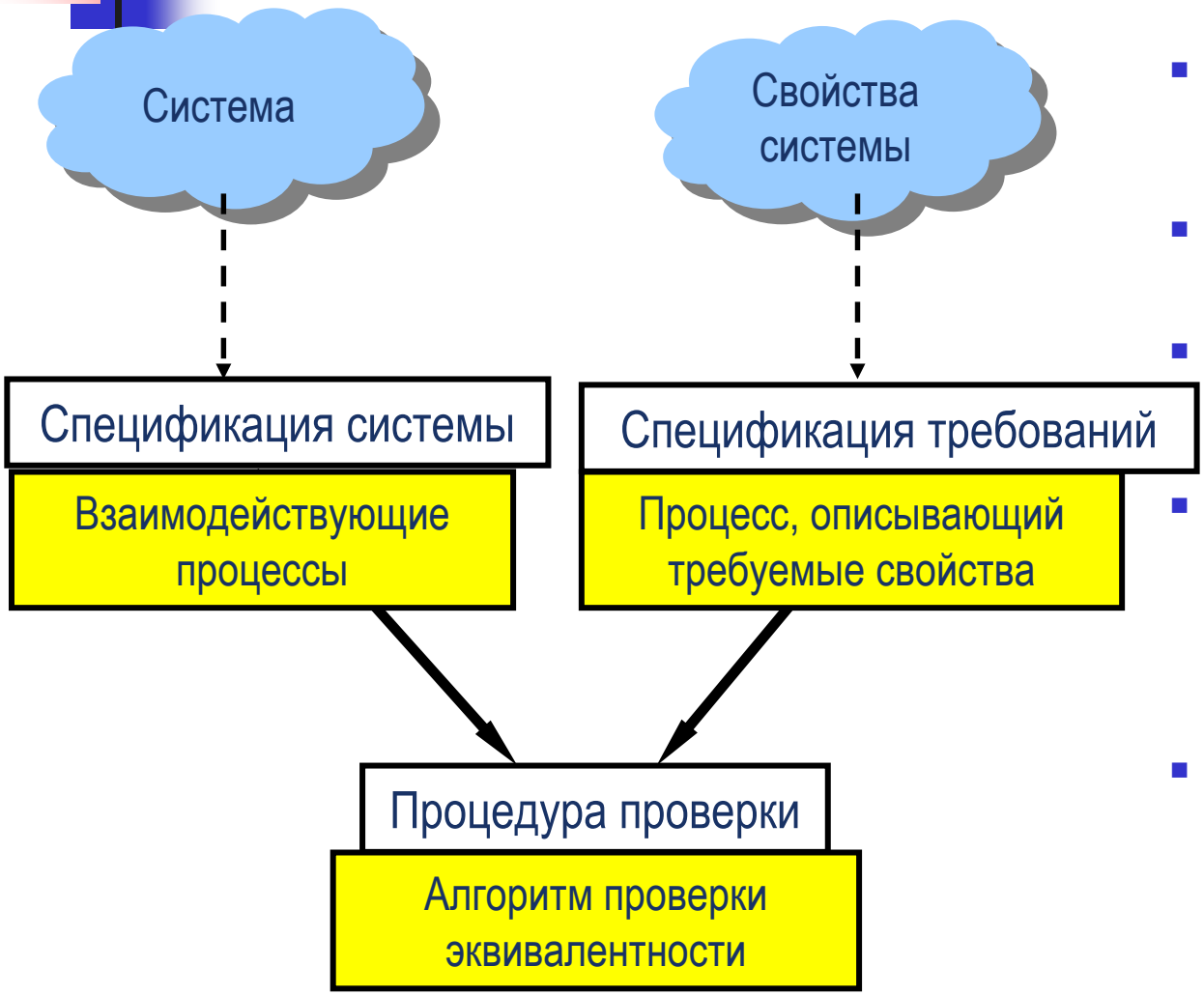
## Анализ (Edinburgh Concurrency Workbench):

**Spec ≈ (?) Sys - Да**

**Spec2 ≈ (?) Sys2 - Да**

**Sys3 - имеет дедлок**

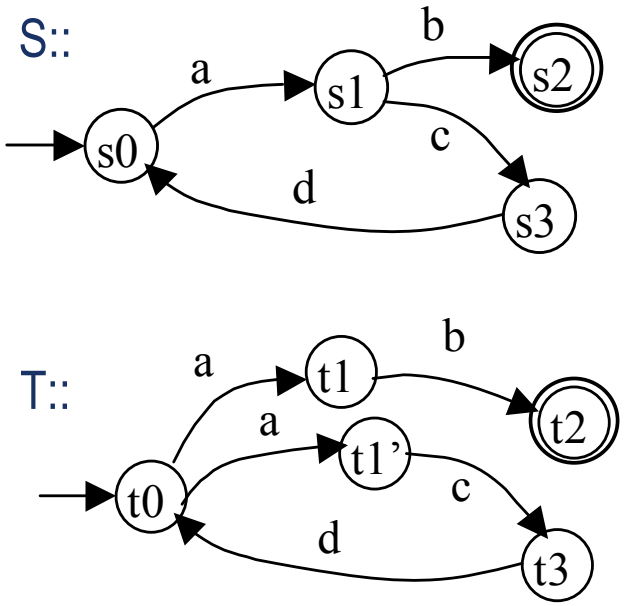
# Подход к верификации в CCS




- Система
  - Несколько параллельных взаимодействующих процессов
- Свойства системы
  - Желаемое поведение на части входов/выходов
- Спецификация системы
  - Описание поведения составляющих процессов на CCS
- Спецификация требований
  - Для каждого свойства – процесс в CCS, который ведет себя на проекции части входов так, как хотел бы пользователь
- Процедура проверки
  - Не очень эффективный алгоритм проверки эквивалентности композиции процессов и процесса, представляющего искомое свойство

**НО: как определяется наблюдаемая эквивалентность?**

# Эквивалентность недетерминированных акцепторов: как построить процедуру проверки?




S и T допускают один и тот же язык (acd)*ab, поэтому S эквивалентен T как языковый акцептор

Доказательство эквивалентности S и T алгебраически

s0 = as1;
s1 = bs2+cs3;
s2 = ε;
s3 = ds0

t0 = at1+ at1';
t1 = bt2;
t1' = ct3;
t2 = ε;
t3 = dt0

Для S:

s0 = as1 = a(bε + cds0) = закон дистрибутивности

abε + acds0 ⇒

используем Arden's rule: s = βs + α ⇒ s = β*α

s0 = (acd)*abε

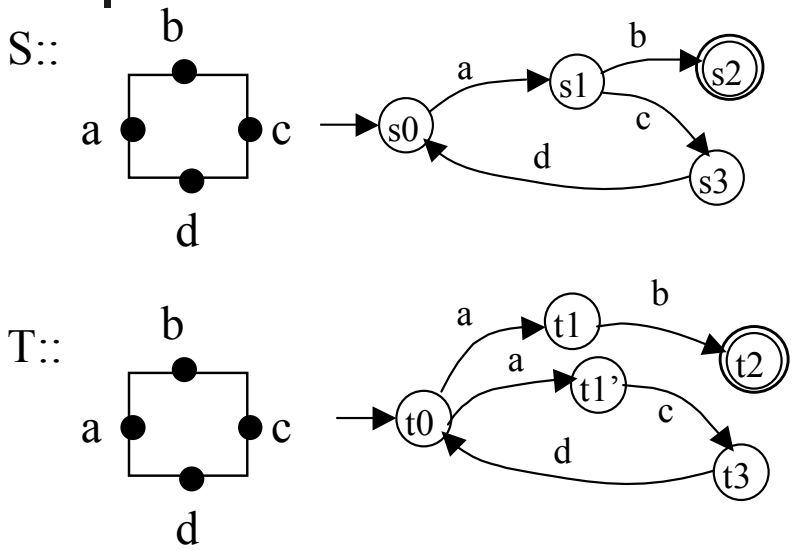
Для T:

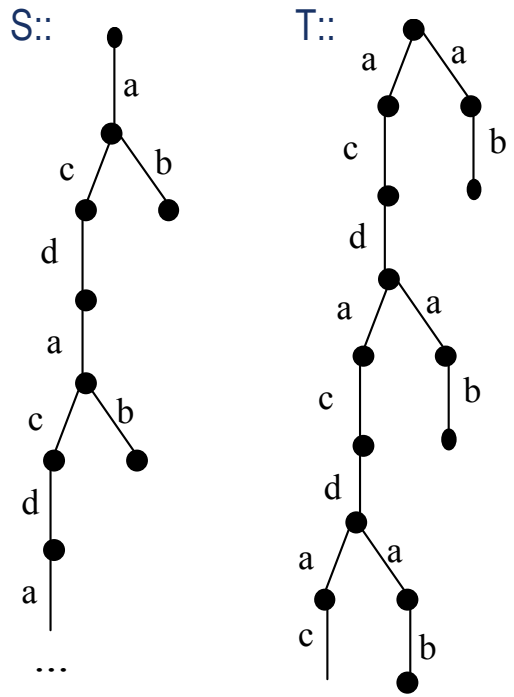
t0 = abε + acd t0 = Используем Arden's rule

t0 =(acd)*abε;

Эквивалентны ли поведения S и T с точки зрения внешнего экспериментатора?

# Недетерминированные акцепторы (процессы)




S и T допускают различные эксперименты, поскольку после подачи a агент T может не принять эксперимент c, а агент S всегда после подачи a принимает эксперимент c. Поэтому S и T не эквивалентны.

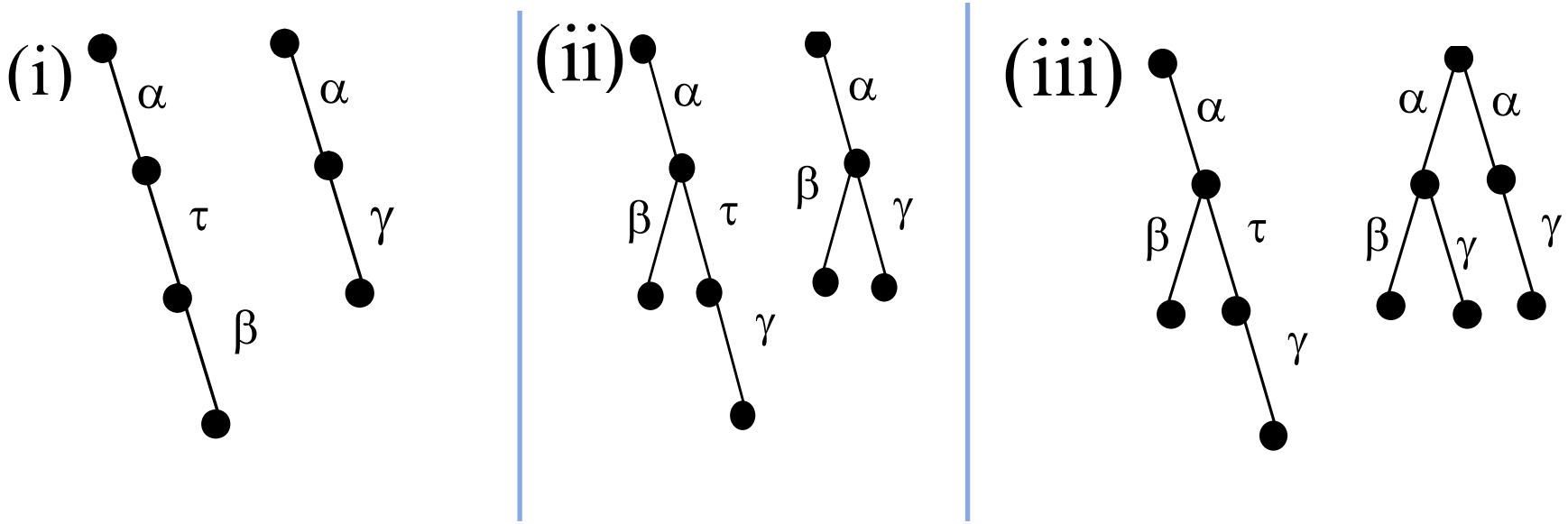
Как определить отношение эквивалентности на деревьях экспериментов?

Р.Милнер представил деревья экспериментов (процессов) выражениями, ввел алгебру процессов и в ней определил отношение эквивалентности

# Эквивалентность деревьев поведений

Наша задача определить понятие эквивалентности процессов через их ВНЕШНЕЕ ПОВЕДЕНИЕ (функционально, не учитывая время)

Какие пары поведений эквивалентны?




Какие аксиомы определяют это отношение поведенческой эквивалентности (бисимуляции)?

# Наблюдаемая эквивалентность поведений

Пусть Δ – множество имен портов, а ***B*** – множество поведений.

Будем писать $\mu.A \text{—}\mu\text{—}> A$ бинарное отношение на множестве поведений.

Определим для каждой (видимой) цепочки $x = \lambda_1\lambda_2 \ldots \lambda_n$ экспериментов по портам Δ (т.е. для любых $x \in \Delta^*$) на множестве ***B*** отношение $=x=>$ так:

$=x=> \;=\; \text{—}\tau^{k0}\text{—}\lambda_1\text{—}\tau^{k1}\text{—}\lambda_2\text{—}\tau^{k2}\text{—}\ldots\lambda_n\text{—}\tau^{kn}\text{—}>$ для некоторых $k_0, k_1, .. k_n$.

*Отношение =x=> прозрачно для любого числа ненаблюдаемых действий, которые агент может выполнить в процессе наблюдаемого эксперимента x.*

***Определение***.

Два процесса P и Q являются наблюдаемо эквивалентными (находятся в отношении бисимуляции, обозначается $P \approx Q$), если для любой цепочки внешних воздействий (для любого $x \in \Delta^*$) справедливо:

*a*) если P =x=> P', то существует такое Q', что Q =x=> Q' и $P' \approx Q'$

*b*) если Q =x=> Q', то существует такое P', что P =x=> P' и $P' \approx Q'$

# Отношение бисимуляции ‘≈’

Отношение ‘≈’ есть предел цепочки отношений $\approx_0$, $\approx_1$, $\approx_2$, … $\approx_k$ …, где:

1) $\approx_0 = B^2$, (т.е. любые два поведения находятся в отношении $\approx_0$ )

2) $\approx_{k+1}$: $P \approx_{k+1} Q$ если и только если для любой цепочки $x \in \Delta^*$ :

a) если $P =x=> P'$, то существует такое $Q'$, что $Q =x=> Q'$ и $P' \approx_k Q'$

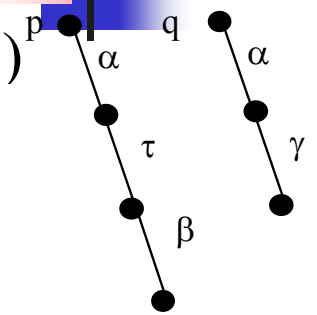
b) если $Q =x=> Q'$, то существует такое $P'$, что $P =x=> P'$ и $P' \approx_k Q'$

Отношение языковой эквивилентности конечных автоматов – это отношение $\approx_1$

Фактически, проверяется не только допустимость цепочек экспериментов, но и эквивалентность всех промежуточных состояний

# Примеры проверки эквивалентности

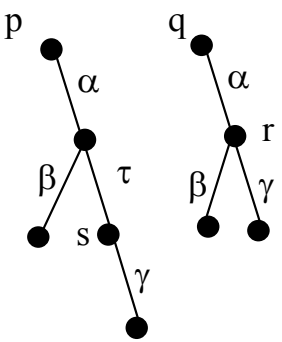
(i)




Верно, что $p \approx_0 q$ (любые два поведения находятся в этом отношении);

Неверно, что $p \approx_1 q$: они допускают разные языки

(ii)



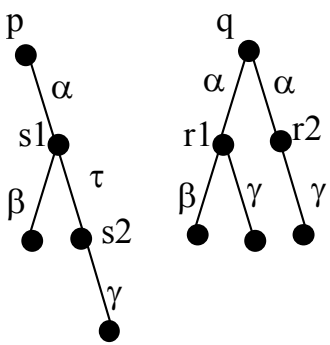

Верно, что $p \approx_0 q$,

Верно, что $p \approx_1 q$, оба допускают язык $\{\varepsilon, \alpha, \alpha\beta, \alpha\gamma\}$

Неверно, что $p \approx_2 q$: эксперимент $\alpha$ приводит p в состояние s, допускающее язык $\{\varepsilon, \gamma\}$, а единственное состояние r, в которое переходит q после подачи $\alpha$, допускает язык $\{\varepsilon, \beta, \gamma\}$.

Следовательно, $(s \not\approx_1 r)$, и поэтому $(p \not\approx q)$

(iii)




Верно, что $p \approx_0 q$.

Верно, что $p \approx_1 q$, оба допускают язык $\{\varepsilon, \alpha, \alpha\beta, \alpha\gamma\}$.

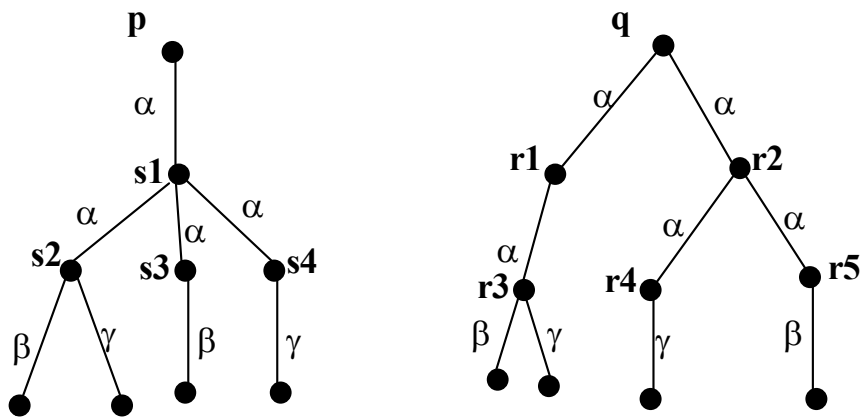
Верно, что $p \approx_2 q$: эксперимент $\alpha$ приводит p в состояния s1, для которого в процессе q после такого же эксперимента есть эквивалентное состояние r1; то же и для s2. Обратное тоже верно.

Поэтому $p \approx q$

# Примеры проверки эквивалентности (2)

Верно, что $p \approx_0 q$.

Верно, что $p \approx_1 q$, оба допускают язык $\{\varepsilon, \alpha, \alpha\alpha, \alpha\alpha\beta, \alpha\alpha\gamma\}$.




Верно, что $p \approx_2 q$:

Действительно:

Для $p \xrightarrow{\alpha} s1$ есть $q \xrightarrow{\alpha} r1$ и $s1 \approx_1 r1$.

Для $p \xrightarrow{\alpha\alpha} s2$ есть $q \xrightarrow{\alpha\alpha} r3$ и $s2 \approx_1 r3$.

Для $p \xrightarrow{\alpha\alpha} s3$ есть $q \xrightarrow{\alpha\alpha} r5$ и $s3 \approx_1 r5$.

Для $p \xrightarrow{\alpha\alpha} s4$ есть $q \xrightarrow{\alpha\alpha} r4$ и $s4 \approx_1 r4$.

Обратно:

Для $q \xrightarrow{\alpha} r1$ есть $p \xrightarrow{\alpha} s1$ и $s1 \approx_1 r1$.

Для $q \xrightarrow{\alpha} r2$ есть $p \xrightarrow{\alpha} s1$ и $s1 \approx_1 r2$.

Для $q \xrightarrow{\alpha\alpha} r3$ есть $p \xrightarrow{\alpha\alpha} s2$ и $s2 \approx_1 r3$.

Для $q \xrightarrow{\alpha\alpha} r4$ есть $p \xrightarrow{\alpha\alpha} s4$ и $s4 \approx_1 r4$.

Для $q \xrightarrow{\alpha\alpha} r5$ есть $p \xrightarrow{\alpha\alpha} s3$ и $s3 \approx_1 r5$.

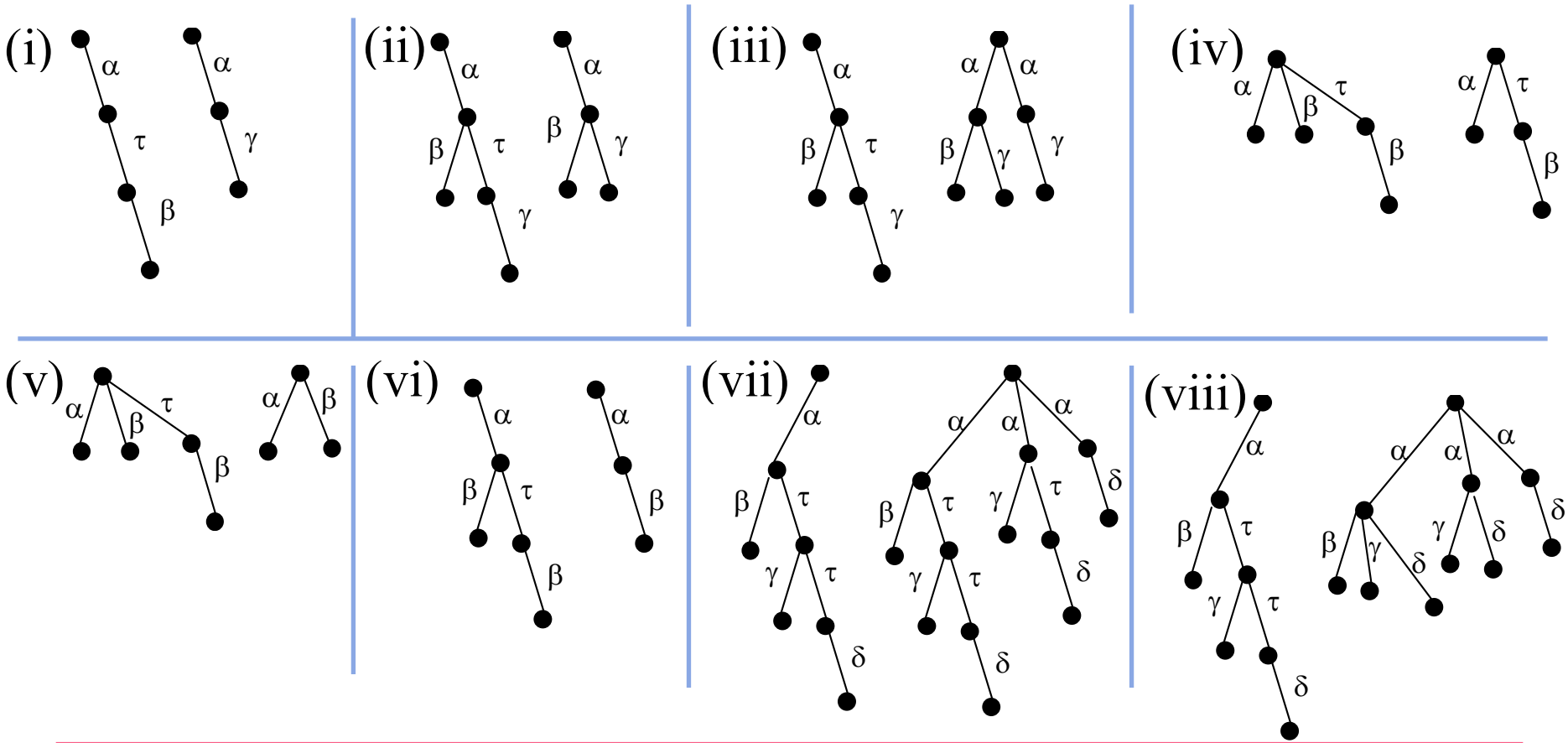
Но! $p \not\approx_3 q$: действительно:

$p \xrightarrow{\alpha} s1$, $q \xrightarrow{\alpha} r1$ и $q \xrightarrow{\alpha} r2$,
однако, $s1 \not\approx_2 r1$ и $s1 \not\approx_2 r2$ .

Для каждого k можно построить агенты, которые k-наблюдаемо эквивалентны, но не являются k+1- наблюдаемо эквивалентными

# Эквивалентность деревьев поведений (2)

Какие пары поведений эквивалентны?




Первые три – проанализированы. На экзамене каждому будет дан какой-либо из оставшихся примеров

# Аксиомы бисимуляции

Аксиомы, справедливые для всех выражений CCS:

- $E1 + (E2+E3) \approx (E1+E2)+E3$
- $E1 + E2 \approx E2+E1$;
- $E + E \approx E$;
- $E + \tau E \approx E$;
- $E + NIL \approx E$;
- $\mu (E + \tau E1) \approx \mu(E + \tau E1) + \mu E1$;
- $\mu \tau E \approx \mu E$;
- $E + \tau (E + E1) \approx \tau( E + E1)$

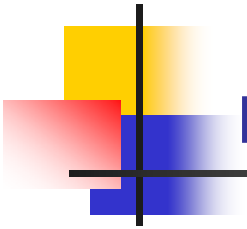
***Определение.*** Конгруэнцией на носителе алгебры называется такая эквивалентность $\rho$, которая согласована со всеми операциями алгебры. А именно, для каждой операции f справедливо $f(a_1, \ldots, a_n) \rho f(b_1, \ldots, b_n)$ если все пары $(a_i,b_i)$ удовлетворяют условию $a_i \rho b_i$

Эквивалентность $\approx$ не является конгруэнцией: $\tau E \approx E$, но $E1+\tau E \not\approx E1+E$;

Максимальная конгруэнция, меньшая $\approx$, называется наблюдаемой конгруэнцией.

Все аксиомы, приведенные выше – аксиомы наблюдаемой конгруэнции

СМЫСЛ КОНГРУЭНЦИИ: Любое подвыражение можно заменять ему конгруэнтным в любом контексте, и новое выражение будет наблюдаемо эквивалентным старому. Это – один из основных приемов преобразования выражений алгебры поведений



# Применение CCS

- Были разработаны автоматизированные системы, выполняющие анализ в рамках CCS – например, Edinburgh Concurrency Workbench
- Известно, что протоколы очень трудно проверять. Были разработаны три языка спецификации протоколов:
  - SDL – язык состояний и переходов (в двух эквивалентных - графической и программной - формах). Не является формальным
  - Language of Temporal Ordering Specifications (Lotos) стандарт ISO (1989г.), основан на CCS. Позволяет формально специфицировать поведение протокола на высоком уровне абстракции. Позволяет использовать множество правил трансформации поведения и отношение эквивалентности, можно проверить НЕКОТОРЫЕ свойства поведения. Но в общем проблема верификации систем, описанных на Lotos, неразрешима
  - Estelle – также формальный язык спецификации протоколов, основан на расширенной модели КА

# π-calculus (pi-исчисление) – алгебра мобильных процессов

## Р.Милнер расширил CCS для возможности формального анализа процессов мобильной коммуникации

- Мобильность процессов и изменение конфигурации систем НЕ ПОКРЫВАЕТСЯ CCS
- pi-calculus – это расширение исчисления CCS для того, чтобы работать с системами процессов с изменяемой конфигурацией (мобильными процессами и реконфигурируемыми связями)
- В дополнение к простейшему CCS, в пи-исчислении можно создавать новые имена каналов и передавать их другим процессам при коммуникации || процессов. Процесс, принявший имя канала, может взаимодействовать по этому новому каналу с окружением
- Это исчисление позволяет описывать распределенные и мобильные вычисления, оно включает λ-исчисление Черча и машины Тьюринга. Служит базой для других теорий. Создано множество расширений этого исчисления, в частности, для ООП

Robin Milner. Communicating and Mobile Systems: the π-calculus, 1999, Cambridge
D.Sangiorgi, D.Walker. The π-calculus: a Theory of Mobile Processes, 2001, Cambridge
B.Victor, F.Moller. The Mobility Workbemch – a tool for the π-calculus. In CAV'94

# CCS и π-Calculus

**ППФ:**

**0** – пустой процесс, Nil

P, Q, ... - имена процессов (бесконечное множество имен)

P|Q – параллельная композиция

a(x).P или a(x)!.P – вывод по каналу a значения x

a(y).P или a(y)?.P – ввод по каналу a нового значения для y

!P – репликация P (порождение нового || процесса P)

{v/x} P – в терме P все свободные вхождения x заменяются на v

{new x} P – в процессе P имя x является локальным

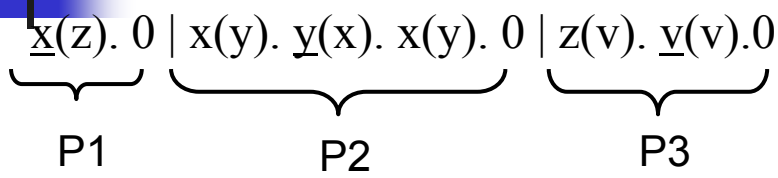
Monadic calculus – передача одного значения

Poliadic calculus – передача вектора значений

!x(z).y(z).0 – что это?

| | |
|---|---|
| CCS: | $a.\ P \mid a.Q \xrightarrow{\tau} P|Q$ |
| CCS с передачей значений: | $a(v).\ P \mid a(x).Q \xrightarrow{\tau} \{v/x\}\ P|Q$ |
| π-исчисление: | $x\ (y).\ P \mid x\ (z).Q \xrightarrow{\tau} \{y/z\}\ P|Q$ но y и z теперь – и имена каналов |
| ***Пример*** выражения π-исчисления: | $x(z).Q \mid x(y).\ y(u).\ 0 \xrightarrow{\tau} Q \mid z(u).\ 0$ |

# Пример поведения с передачей значений




Что происходит?

***ШАГ 1***. Первые два процесса взаимодействуют по каналу x.
Во втором процессе P2: y:=z, т.е. переменная y - имя канала – приняла значение z.

Поведение всей системы после первого шага:

$0 \mid \underline{z}(x).x(z).0 \mid z(v).\underline{v}(v).0$ первый процесс стоит

***ШАГ 2***. Передача теперь идет через z. Процесс P2 передает по z значение x, а P3 принимает это значение, присваивая переменной v значение x, теперь в P3 v=x

Поведение после второго шага:

$0 \mid x(z).0 \mid \underline{x}(x).0$

3. Последний шаг в функционировании процессов – это взаимодействие второго и третьего процессов по каналу x, передается имя x этого канала. В процессе P2 оно присваивается переменной z:

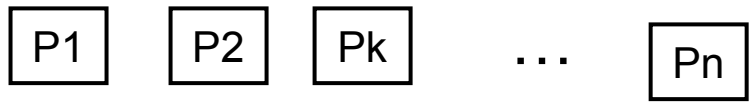
Поведение после третьего шага:

$0 \mid 0 \mid 0$ все процессы стоят

# Пример: обслуживание серверами клиентских процессов

Имеем пул серверов Si и множество клиентских процессов Pk, которым время от времени нужно обслуживание. Каждый сервер может обслужить любой процесс, если свободен.

Как организовать работу?





РЕШЕНИЕ:

Вводим дополнительный процесс-диспетчер.

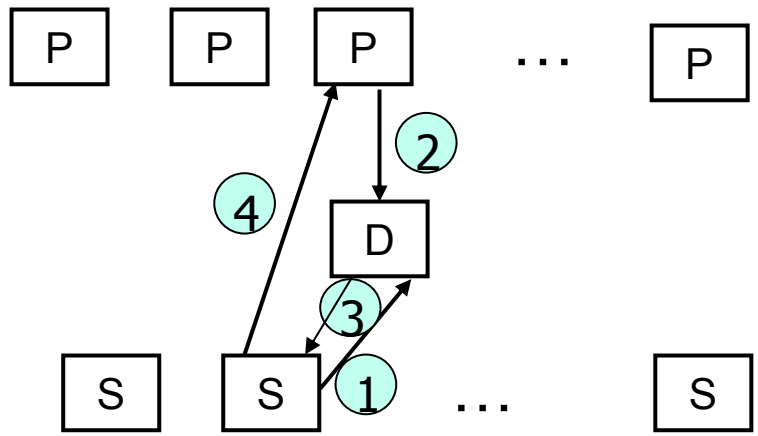
Сценарий:

1. Когда один из серверов свободен, он извещает об этом диспетчера

2. Клиентский процесс для получения обслуживания обращается к диспетчеру и регистрируется - передает свое имя (адрес)

3. Диспетчер передает имя зарегистрировавшегося клиента свободному серверу

4. Сервер связывается с клиентом по его имени, получает от него запрос и обслуживает его, возвращая ему запрошенную информацию, после чего снова становится готовым к обслуживанию



# Формальное описание системы с двумя серверами

В CCS такая реконфигурация процессов невозможна

***Повторяем сценарий*** (обслуживание клиентов несколькими серверами):

1. Когда любой из серверов готов, он извещает об этом диспетчера
2. Клиентский процесс для получения обслуживания обращается к диспетчеру и регистрируется - передает свое имя (адрес) и запрос
3. Диспетчер передает имя и запрос зарегистрировавшегося клиента свободному серверу
4. Сервер связывается с клиентом по его имени, получает от него запрос и обслуживает его, возвращая ему запрошенную информацию, после чего снова становится готовым

***Диспетчер*** ждет сигнала от очередного клиента с приемом его имени (N) и запроса (s), ждет сигнала от готового сервера, передает ему имя и запрос очередного клиента

$$D = checkin\ (N, s)\ .\ (next_1\ .\ \underline{ans_1}(N,s)\ .\ D + next_2\ .\ \underline{ans_2}(N,s)\ .\ D)$$

***i-й сервер*** посылает информацию о готовности, получает в ответ имя клиентского процесса и запрос, после чего выдает ***этому клиенту*** требуемые им данные

$$S_i = \underline{next_i}\ .\ ans_i\ (N, s)\ .\ \underline{N}\ (rez(s))\ .\ S_i$$

Клиентский процесс посылает свое имя и запрос (data), получает на свое имя результат

$$P(name, data) = \underline{checkin}\ (name, data)\ .\ name\ (x)\ .\ P'\ (x)$$

# CSP – Communicating Sequential Processes (Xoap)

- Язык для описания взаимодействующих процессов. CSP поддержан элегантной математической теорией и инструментами верификации. Удобен для моделирования и анализа систем, которые включают сложный обмен сообщениями, в частности, криптопротоколов
- Книга *Communicating Sequential Processes* опубликована в 1985. В 2006 эта книга была третьей по числу ссылок на нее в области Computer Science
- Программа на CSP описывает параллельно функционирующие последовательные процессы, взаимодействующие рандеву
- Применение CSP для разработки safety critical systems:
  - использован для верификации процессорного конвейера и Virtual Channel Processor – вынесенного процессора для управления внешней коммуникацией
  - Bremen Institute for Safe Systems и Daimler-Benz Aerospace построили модель отказоустойчивой системы управления (23 000 строк кода) для использования в Международной космической станции. Модель была проанализирована с целью проверки того, что в системе нет блокировок и ливлоков. Верификатор вскрыл множество ошибок, которые было трудно обнаружить тестированием
- Язык CSP использован в транспьютерах, на нем основан язык Оккам

# OCCAM и транспьютеры

- В 1984 г. фирма INMOS Ltd. (Бристоль, Англия) объявила о выпуске микропроцессоров нового типа для параллельного программирования – транспьютеров (transistor computer), с языком программирования OCCAM (язык интерпретировался аппаратно)
- Уильям Оккам (*William of Ockham)* ~1285
  английский философ, францисканский монах из Оккама (Южная Англия). «Бритва Оккама» — методологический принцип: *«Не следует привлекать новые сущности без самой крайней необходимости»*.
- Транспьютеры – параллельно функционирующие процессоры- сложный суперскалярный конвейерный процессор
- OCCAM – это реализация алгебры процессов CSP Хоара, в котором процессы взаимодействуют по рандеву
- Реализация взаимодействия: если процессы в разных процессорах, то взаимодействие между ними реализуется по реальным физическим каналам, если процессы в одном процессоре, то взаимодействие между ними моделируется виртуальными каналами

# Язык OCCAM: структура программы

- Примитивные процессы

```
keyboard ? x            /* ввод с клавиатуры значения в переменную x  */
screen ! y              /* вывод на экран значения переменной y  */
x:=x+1                  /* оператор присваивания  */
```

- Конструкторы SEQ, PAR, ALT

```
SEQ                     /* процессы, выполняющиеся последовательно */
   buffer.in   ?   x
   buffer.out  !   x
```

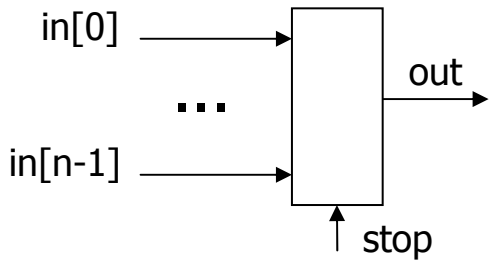
-

```
PAR                     /* процессы, выполняющиеся параллельно   */
   x := x+1
   z :=x*x
```

-

```
ALT                     /* несколько guarded commands – защищенных процессов */
  count1 < 100  &  c1 ? data
    SEQ
       count1 := count1 + 1
       merged ! data
  count2 < 100 & c2 ? data
    SEQ
       count2 := count2 + 1
       merged ! data
  status ? request
    PAR
       out ! count
       in ?  x
```

# Простой пример программы на языке Оккам




Буфер с несколькими входами и одним выходом

```
VAR going :
SEQ
    going := TRUE
    WHILE going
        VAR x :                      /* описание переменной x  */
        ALT
          ALT i = [ 0 FOR n ]        /* принимает по любому готовому входу,
              in [ i ] ? x               передает принятое значение на выход  */
                 out ! x
          stop ? ANY                 /* до тех пор, пока не будет принят сигнал
              going := FALSE             синхронизации по входу stop  */
```

Повторение имеет форму: [ *первый* FOR *количество* ]

ANY – фиктивная переменная, используется для синхронизации по событиям (синхронизации)

# Язык OCCAM: пример программы мультиплексора


По нескольким входным каналам поступают потоки байтов.

Как только по какому-нибудь из них поступит полная пачка (слово) (например, 100 байтов), ее нужно передать по выходному каналу

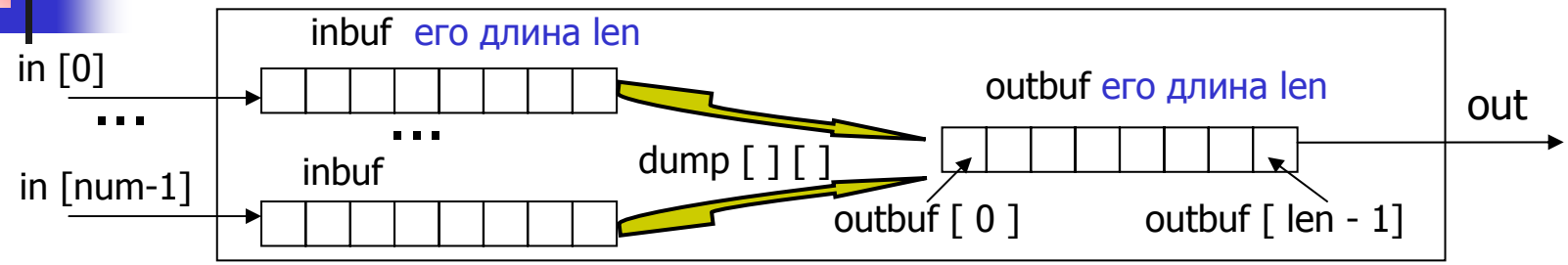
Как реализовать такую программу???

Конечно, это должны быть параллельно функционирующие процессы, между ними должна быть синхронизация

Построить такую программу на обычном языке довольно сложно

На Оккаме это делается очень изящно

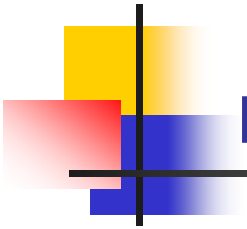
# Язык OCCAM: пример программы мультиплексора




```
DEF num = 8, len = 100
PROC multiplexor ( CHAN in [ num ],  out )
      CHAN dump [ num ] [ len ]          /* параллельно: 1) прием в inbuf по in и выдача в outbuf  по dump
      PAR                                                 2) прием в outbuf  по dump и выдача в out  * /
            PAR i = [ 0 FOR num ]              /* параллельно: прием в inbuf [ i ] по in и выдача  по dump */
                  VAR inbuf [ len ] :
                  WHILE TRUE                    /*  бесконечный цикл приема в inbuf, потом выдачи из него */
                        SEQ
                              SEQ  j = [ 0 FOR len ]   /*последовательно вводим,  параллельно выводим */
                                    in [ i ] ? inbuf [ j ]
                              PAR j = [ 0 FOR  len ]
                                    dump [ i ] [ j ] ! inbuf [ j ]      /*параллельно выводим из ячеек inbuf */
            WHILE TRUE                                  /* бесконечный цикл приема в outbuf и выдачи из него */
                  VAR outbuf [ len ] :
                  ALT i = [ 0 FOR num ]
                        dump [ i ] [ 0 ]  ?  outbuf [ 0 ]          /*  защита: если готов 0 элемент, то вводим
                              SEQ                                     параллельно из какого-нибудь inbuf, а потом
                                    PAR k = [ 1 FOR len-1 ]          последовательно выводим из outbuf по out */
                                          dump [ i  ] [ j ] ? outbuf [ j ]
                                    SEQ j = [ 0 FOR len ]
                                          out ! outbuf [ j ]
```

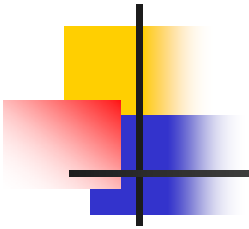
# Заключение (алгебры процессов)

- CCS – первая попытка формализации параллельных взаимодействующих процессов на основе логико-алгебраического подхода для верификации, отражающей параллелизм, коммуникацию, недетерминизм, а не операционного исполнимого графического формализма, как СП)
- Разработаны системы для проверки наблюдаемой эквивалентности (бисимуляции) заданных процессов.
- Для CCS разработано расширение с введением формализмов, описывающих передачу значений при взаимодействии. Инструментов, поддерживающих верификацию систем с передачей значений, нет
- Был стандартизирован язык Lotos, основанный на CCS, для формальной спецификации и анализа протоколов
- Взаимодействие процессов по рандеву используется во многих моделях представления параллельных процессов, в частности в языке PROMELA
- В $\pi$-исчислении вдобавок к простейшему CCS, можно создавать новые имена каналов и передавать их другим процессам, в связи с чем можно описывать мобильные процессы, процессы с реконфигурацией и т.п.
- $\pi$-исчисление сейчас интенсивно исследуется. На нем основано множество расширений, например, формализм спецификации и анализа криптографических протоколов



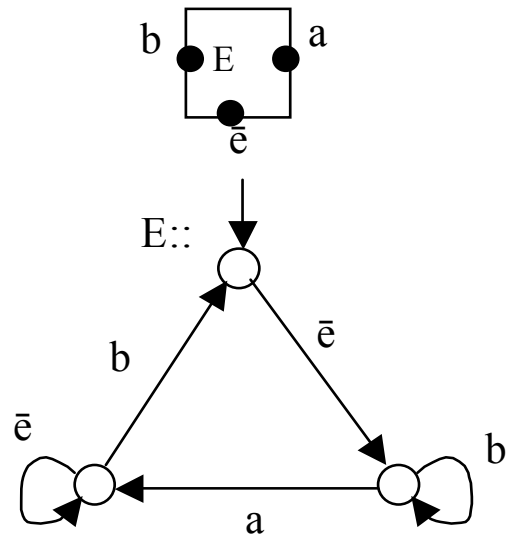
# Персоналии

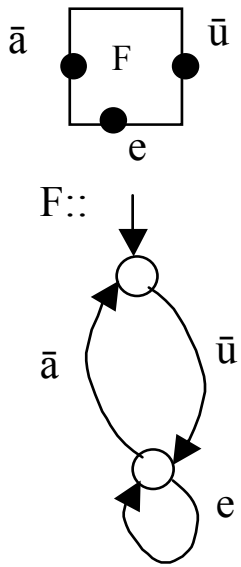
- Робин Милнер – (1934 – 2010) – широко известный британский ученый. Внес три основных вклада в CS:
  - первый инструмент для автоматического доказательства теорем,
  - теоретическую основу для анализа параллельных процессов (CCS)
  - pi-исчисление.
- Награжден премией Тьюринга в 1994 г.
- Сэр Антони Чарльз Хоар (р. 1934) был профессором в Оксфорде, сейчас - senior researcher в Microsoft Research

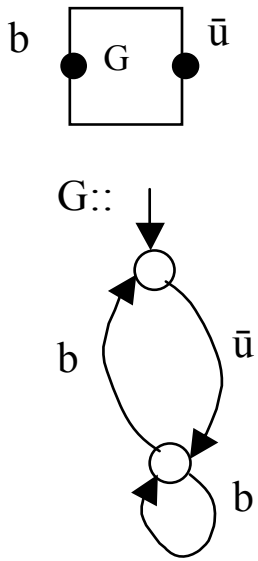


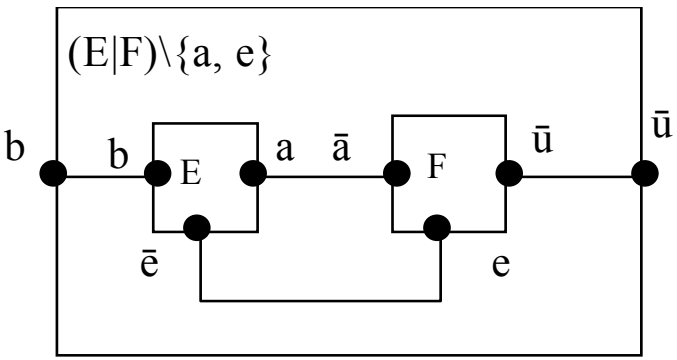
# Спасибо за внимание

# Пример задачи на экзамене

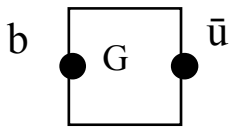



≈ (?)




- Заданы три процесса, E, F и G.
- Проверить, будет ли параллельная композиция (E|F)\ {a,e} наблюдаемо эквивалентна процессу G

# Задача на π-исчисление

Проанализировать работу системы, описывающей произвольное число серверов (модификация предыдущей задачи)

$D = checkin\ (n, s)\ .next\ (a).\underline{a}\ (n,s)\ .\ D$

$S_i = \underline{next}\ (ans_i)\ .\ ans_i\ (n, s).\ \underline{n}\ (rez(s)\ )\ .\ S_i$

$P(name, data) = \underline{checkin}\ (name, data)\ .\ name\ (x)\ .\ P'$