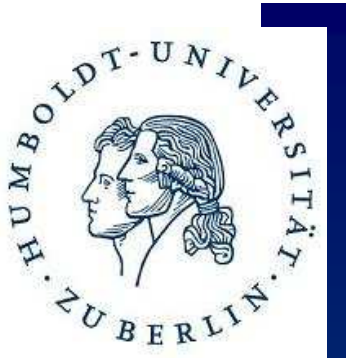
SUMMERSOC 2013

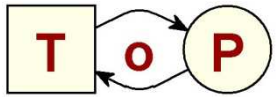


W. Reisig

Conceptual foundations of Service Oriented Computing

-- Formal Methods for SOC --

Prelude



Theory of Programming

Prof. Dr. W. Reisig

Put into a general perspective

Is science driven by theory or by experiment?

Physics:

both

Biology:

so far: more by experiments, with *in*creasing theory

Informatics:

so far: more by experiments (technology),

with *de*creasing theory

Paradigms of computing

Paradigms of computing	*Corresponding Theory*
Classical programming	Computable functions
Object orientation	ADT, Algebr. Spec, logic
SOC	??? Nothing; driven by industry

Basic Assumption

SOC *fundamentally differs* from classical computing

- "always on"
- Non-terminating behaviors
- Not only computers run services,
  but also organizations, machines, persons …

Have the dream of a different world
With rich conceptual foundations
Not just giving better answers,
but soliciting better questions ….

State of the art

What do we see in SUMMERSOC, conceptually?

Plain English,
graphical representations (including BPMN etc)
program code.

May be, this is adequate
to cope with the problems actually considered.

My contributions to recent SUMMERSOCS

- Loose Coupling
- Asynchronous Communication
- Open Petri Nets

Conceptual basis: Petri Nets, Transition systems (automata)

Problems considered:

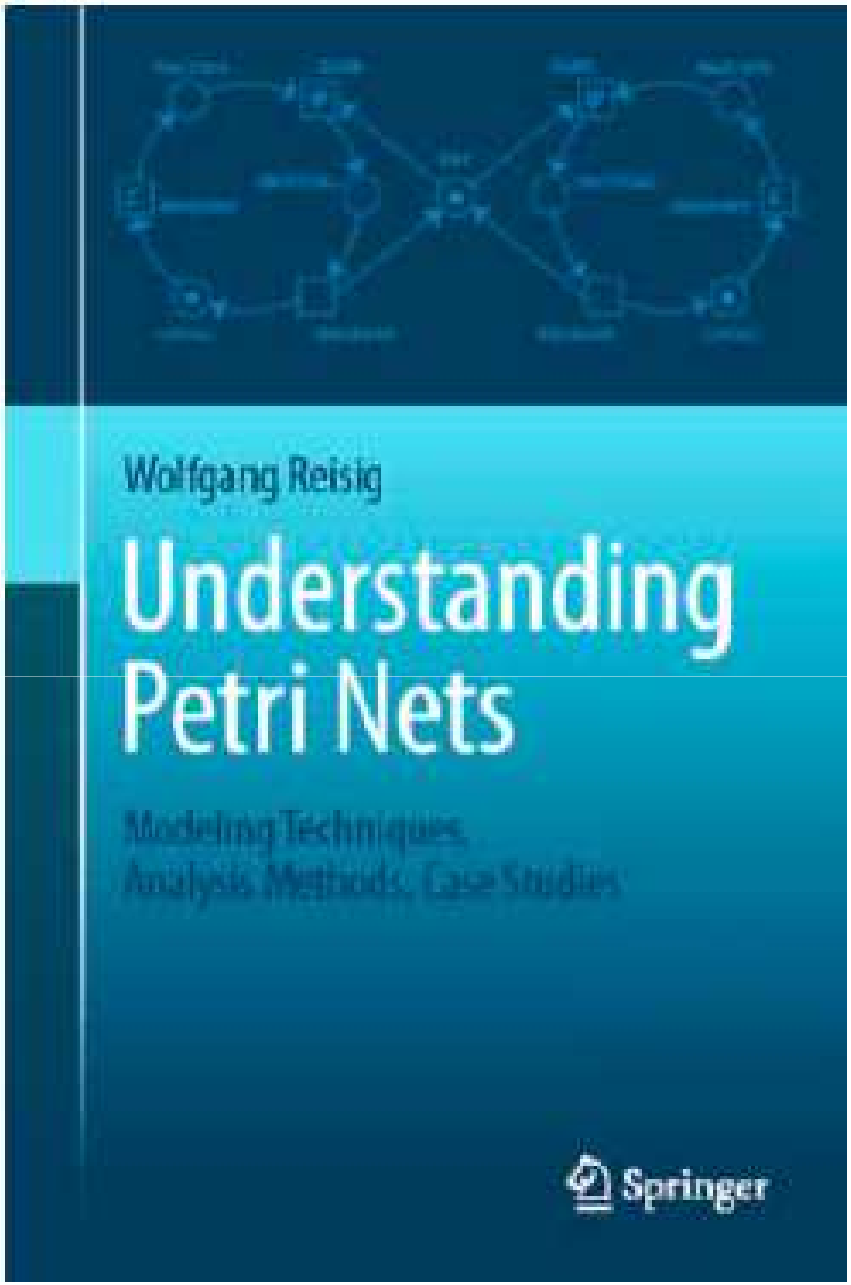
- The partners of a service (to meet a goal)
- Substitute a service by an other one
- Equivalence
- Adapter generation
- Tools

Learned from Frank on Monday:

present a commercial

Leaflets on the reception desk
20% discount

If you like Frank's book
you will also like this one!

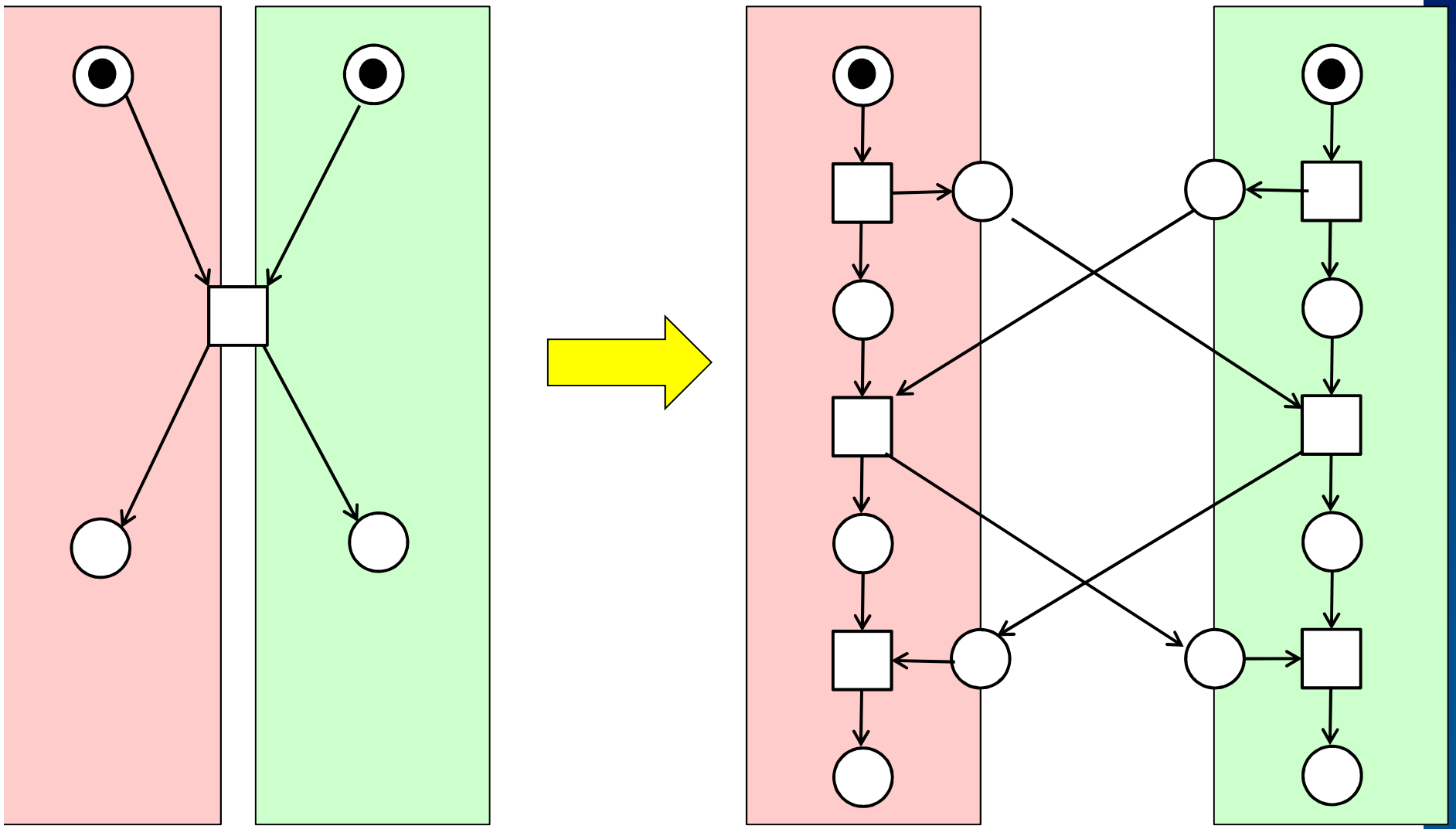


This year:
Synchronous Communication

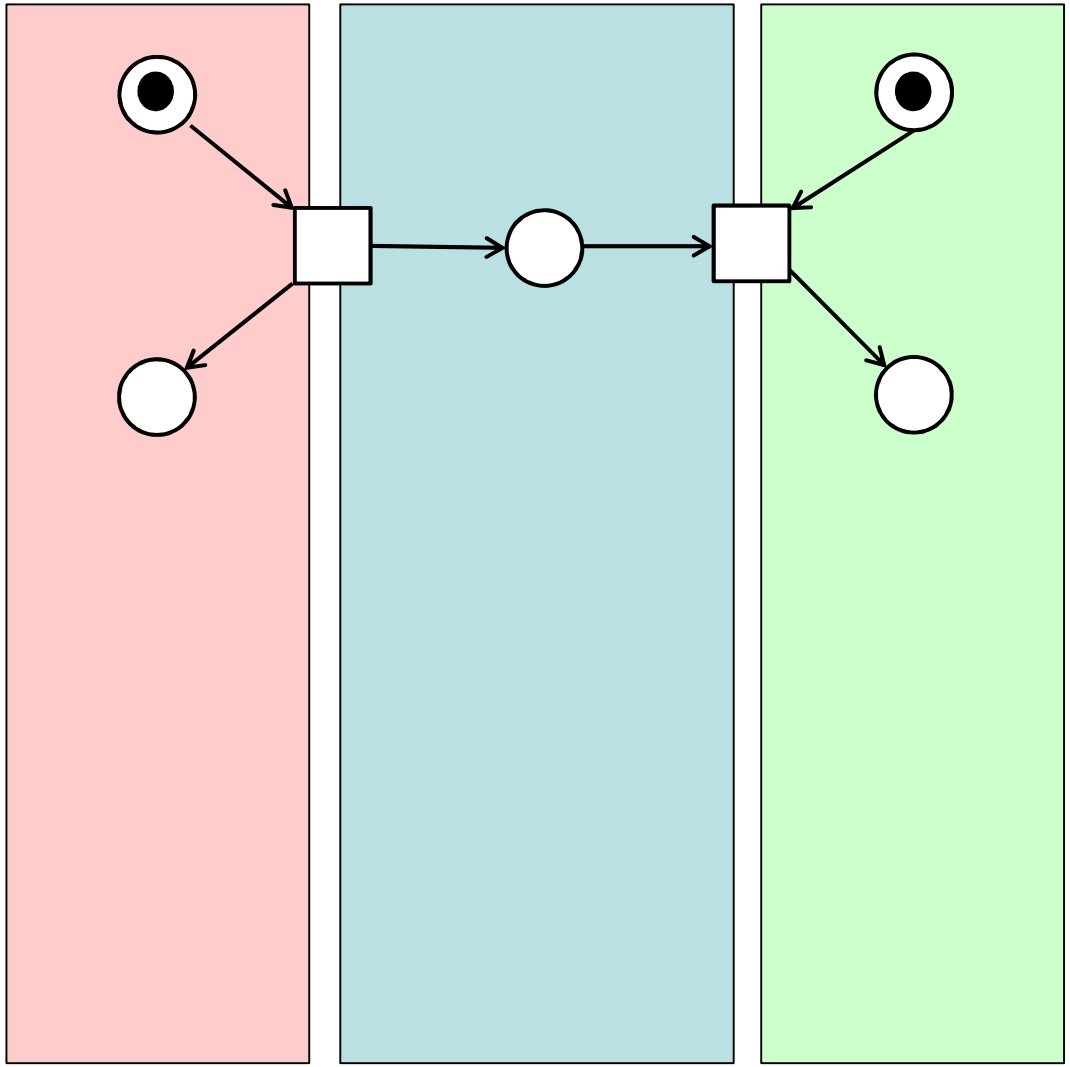
… has deep theory.

Ultimate aim:
Learn from synchronous
communication
for the asynchronous case.

Synchronous modeled asynchronous



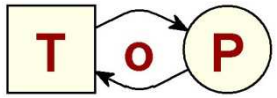
Asynchronous modeled synchronous



End of Prelude

Synchronous Processes etc: their highlights in brief

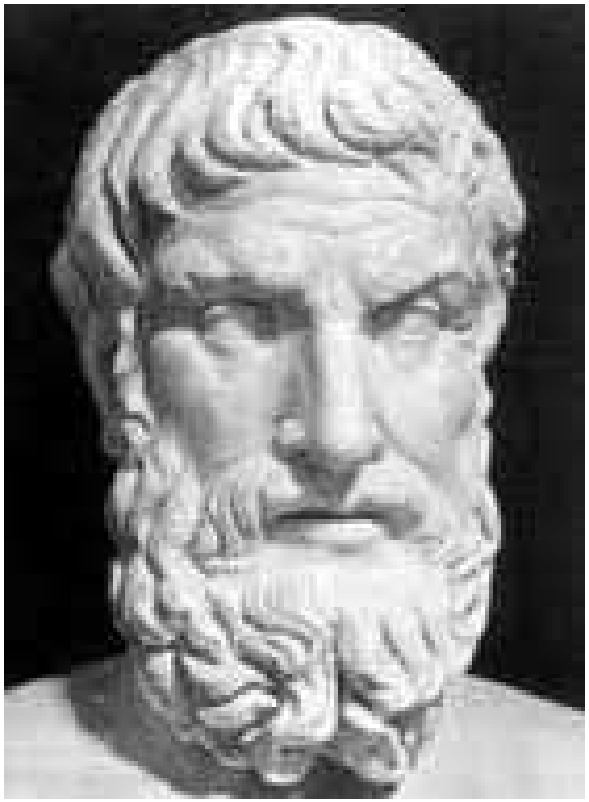
1. ***Their definition: CCS***
2. Their canonical equivalence: Bisimulation
3. Their properties: CTL*
4. Their verification: Model Checking



Theory of Programming

Prof. Dr. W. Reisig

Who is this?



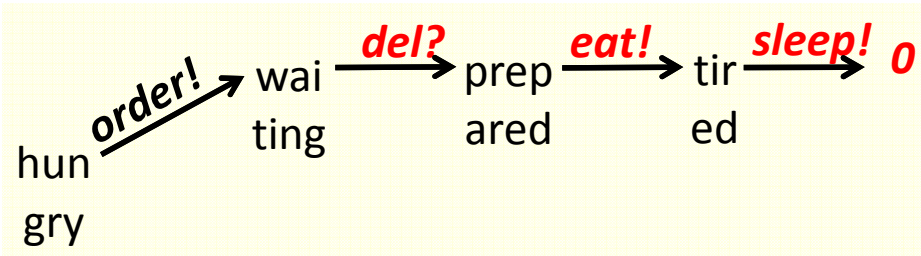
Ἐπίκουρος

Born 341 bc

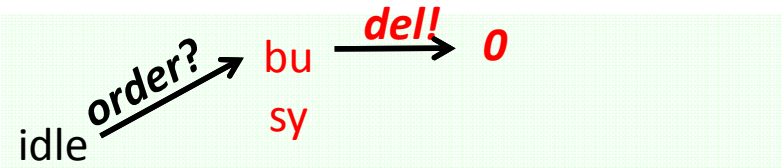
Propagating
the joy of live

After recent
face lifting:

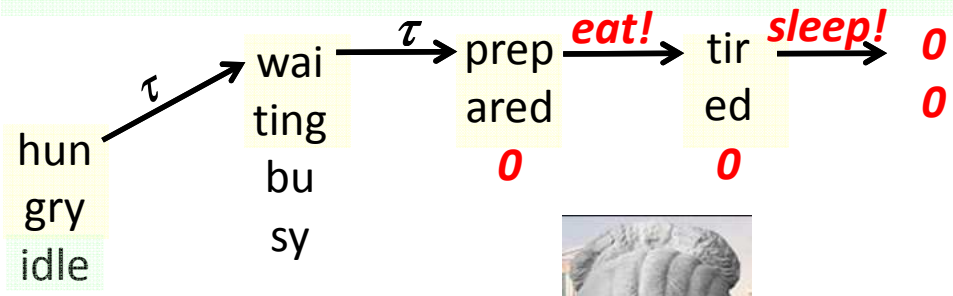
Epikuros orders a pizza



Epikuros = order!. del?. eat!. sleep!. 0



Pizza Hut = order?. del!. 0



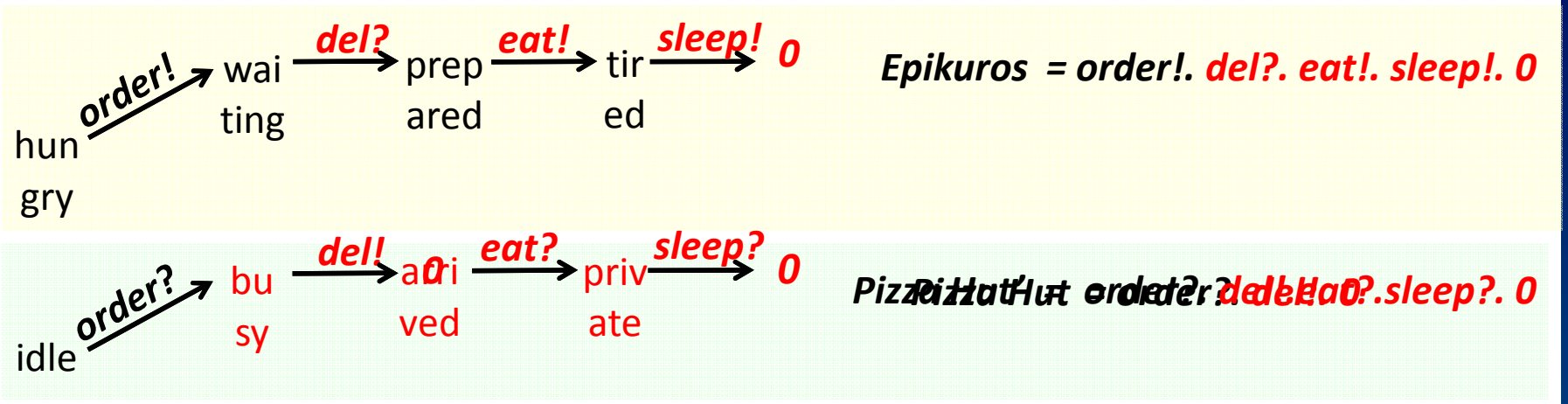
Epikuros|Pizza Hut $\xrightarrow{\tau}$

del?. eat!. sleep!. 0 | del!. 0 $\xrightarrow{\tau}$
eat!. sleep!. 0 | 0

Ξανθίππη = ***eat?. sleep?. 0***

(***Epikuros|Pizza Hut***) | ***Xanthippe*** $\xrightarrow{\tau}$ * ***0|0|0***

Idea: Xanthippe runs the ***delivery***

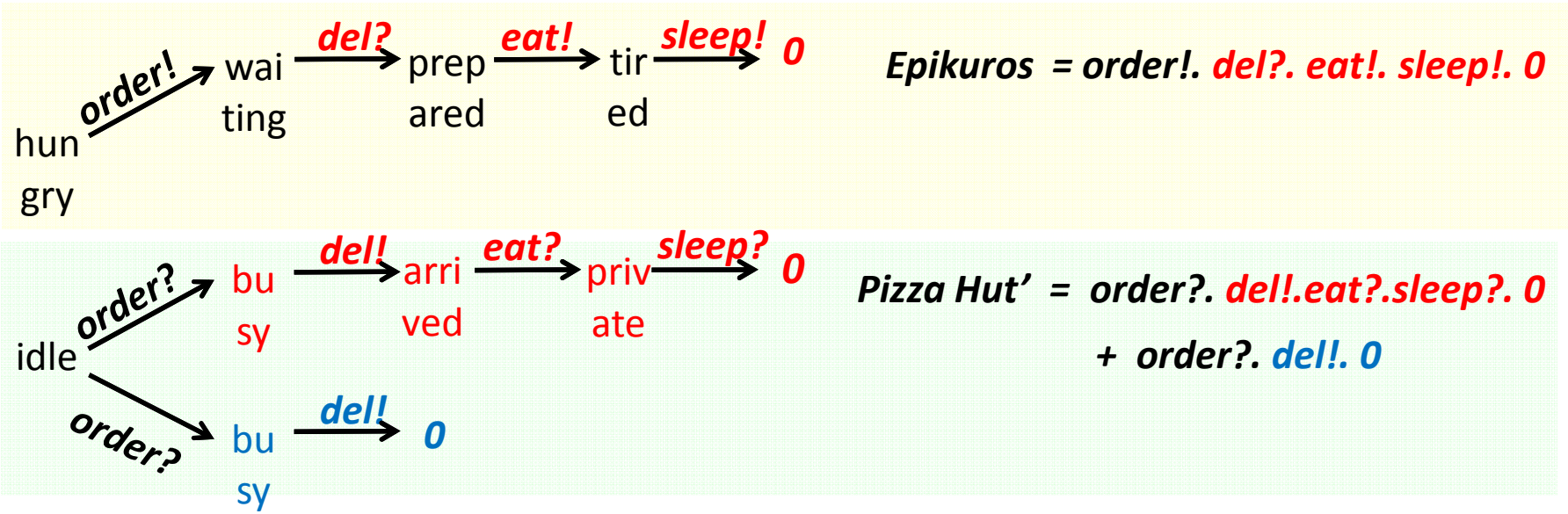


Now : (*Epikuros*|*Pizza Hut'*) →* *0*|*0*

Ξανθίππη = *eat?. sleep?. 0*

Problem: delivery now takes *much* time …
How improve?

… a second delivery branch



… yields two behaviors:

- The known one:
- The new one:

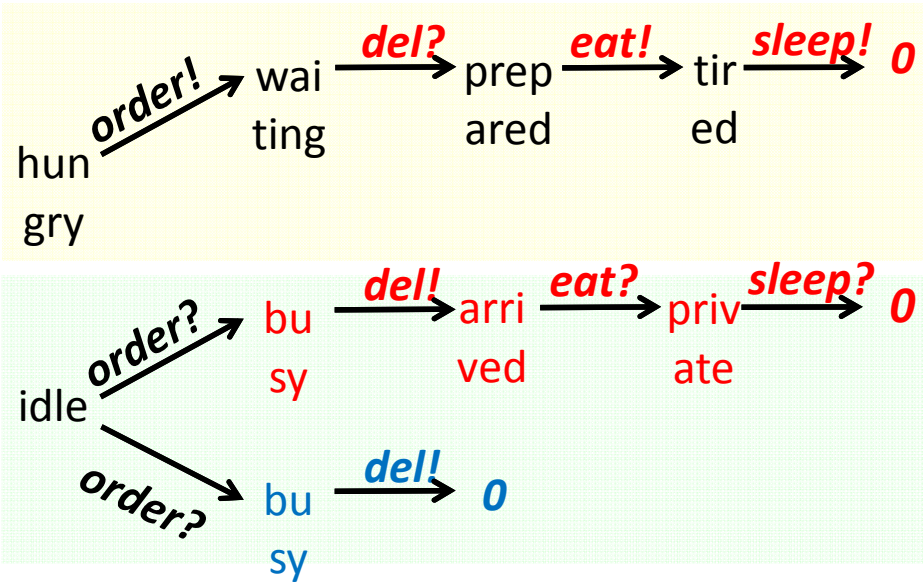
Now : (*Epikuros*|*Pizza Hut'*) →* 0|0

Epikuros|*Pizza Hut'* —τ→ del?. eat!. sleep!. 0 | del!. 0 → ???

Epikuros does not like the blue branch …

Problem: delivery now takes *much* time …
How improve?

How avoid the blue branch ?



Epikuros = order!. del?. eat!. sleep!. 0

Pizza Hut' = order?. del!.eat?.sleep?. 0
+ order?. del!. 0

… yields two behaviors:

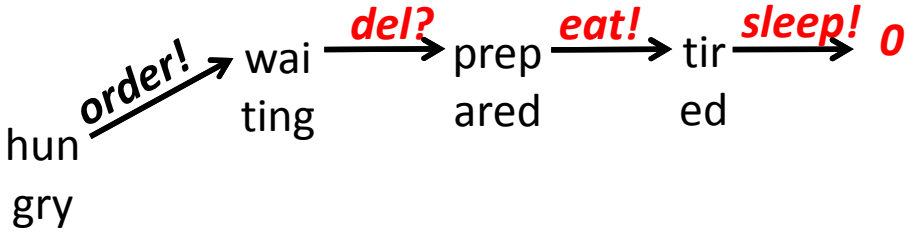
- ***The known one:***
- ***The new one:***

Now : (*Epikuros*|*Pizza Hut'*) $\rightarrow^*$ *0*|*0*

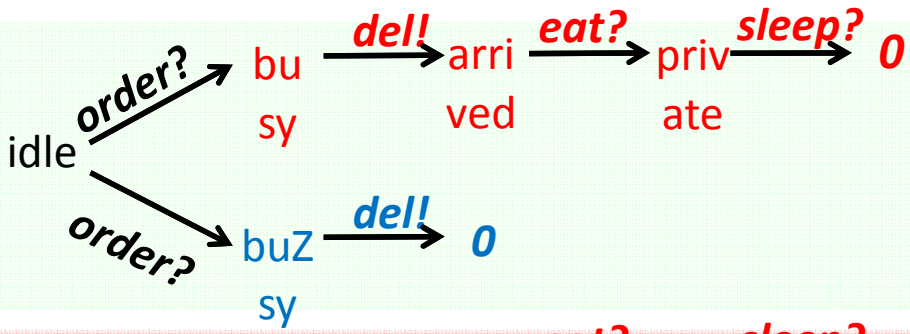
Epikuros*|*Pizza Hut' $\xrightarrow{\tau}$ ***del?. eat!. sleep!. 0 | del!. 0*** $\rightarrow$ ***???***

Epikuros does not like the blue branch …

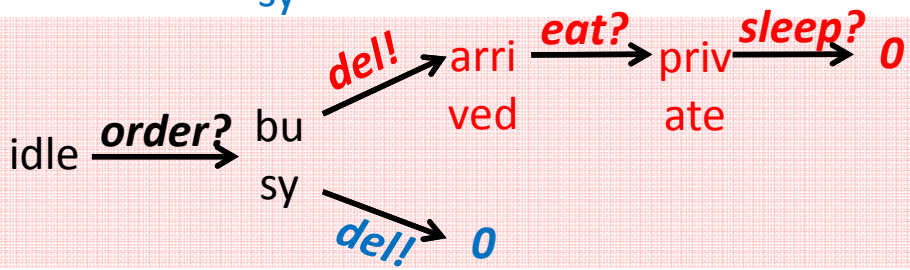
How avoid the blue branch ?



Epikuros = order!. del?. eat!. sleep!. 0



Pizza Hut' = order?. del!.eat?.sleep?. 0
+ order?. del!. 0

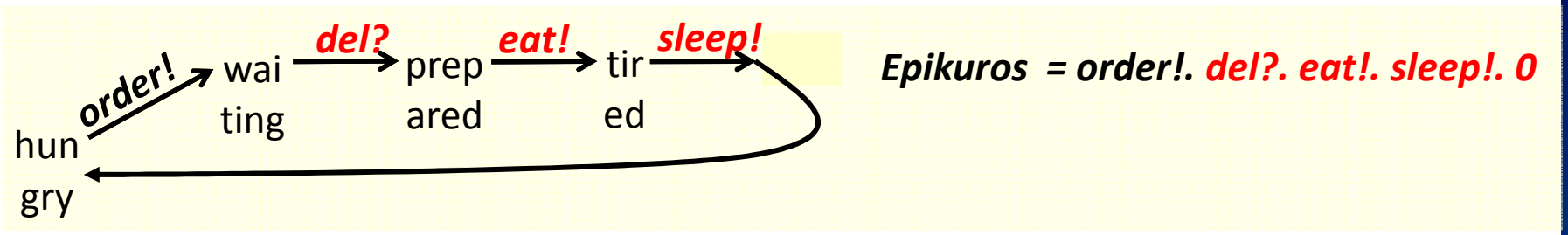


Pizza Hut'' =
order?. (del!.eat?.sleep!. 0 + del!. 0)

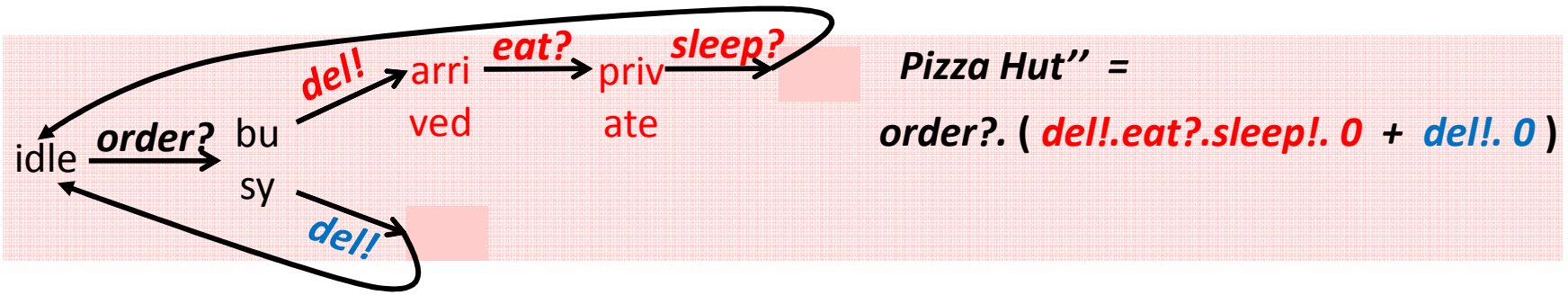
… yields *one* behavior:
Epikuros* | *Pizza Hut''* →* *0|0

***Epikuros* is happy**

Epikuros wants it repeatedly



Epikuros = order!. del?. eat!. sleep!. 0



Pizza Hut'' = order?. (del!.eat?.sleep!. 0 + del!. 0)

Write equations:

Epikuros = order!. del?. eat!. sleep!. Epikuros

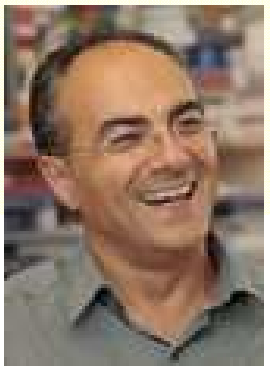
Pizza Hut'' = order?. (del!.eat?.sleep?. . Pizza Hut'' + del!. . Pizza Hut'')

Observe: Epikuros | Pizza Hut'' →* Epikuros | Pizza Hut''

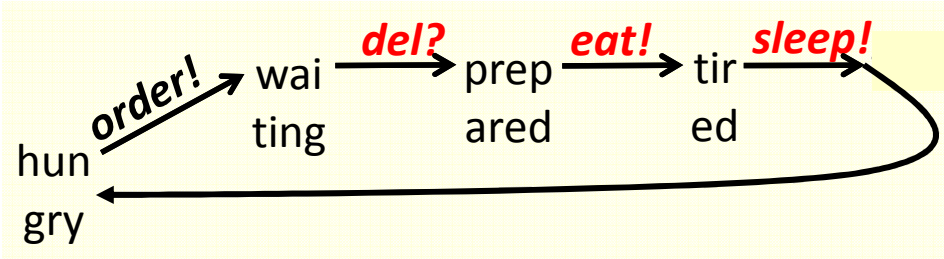
… which leaves him very happy

Epikuros turns jealous of

Ξέρξης



After recent face lifting

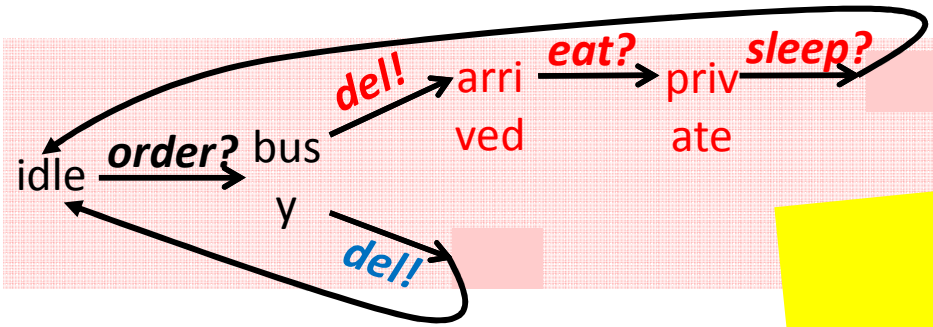


Xerxes = order!. del?. eat!. sleep!. Xerxes

What may happen:

(Epikuros | Pizza Hut'') | Xerxes $\xrightarrow{\tau^*}$

(Epikuros | eat?.sleep?. Pizza Hut'')| eat!. sleep!. Xerxes $\rightarrow$ ...



Pizza Hut''

Perfect for Ἐπίκουρος
(not so nice for Ξέρξης)

Write equations:

Epikuros = order!. del?. eat!. sleep!. Epikuros

Pizza Hut'' = order?. (del!.eat?.sleep?. . Pizza Hut'' + del!. . Pizza Hut'')

Epikuros: **Xerxes, don't engage del !!!** : ***(Epikuros | Pizza Hut'')*** **\ {del}**

Summing up: processes

Given sets *N* (*names*) and *A* (*actions*), $\nu \in N$.

For $a \in A$ let $\alpha = a!$ or $\alpha = a?$; let $B \subseteq A$. *CCS* expressions:

$$P := 0 \mid \nu \mid \alpha.P \mid P_1 + P_2 \mid P_1 | P_2 \mid P \backslash B$$

Epikuros

sleep!. Epikuros

del!.eat?.sleep!. 0 + del!. 0

Epikuros | Pizza Hut''

Epikuros | Pizza Hut'' \ {del}

Equations

$$\nu = P$$

Epikuros = order!. del?. eat!. sleep!. Epikuros

Replacement rules

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$\alpha = a!$ or $\alpha = a?$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$\overline{a!} = a?, \quad \overline{a?} = a!$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \qquad \frac{P \xrightarrow{\alpha} P', \; Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\frac{P \xrightarrow{\alpha} P', \; a \notin B}{P \backslash B \xrightarrow{\alpha} P' \backslash B}$$

$B \subseteq A$

$$\frac{X = P, \quad P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'}$$

Perfect partners reduce to **0**, **0|0**, **0+0** etc

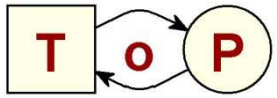
Extensions and variants

Most important:
Message passing.

Replace $\alpha = a!$ by $\alpha = a!(x)$
and $\alpha = a?$ by $\alpha = a?(y)$,
with shared variables x and y

Synchronous Processes etc: their highlights in brief

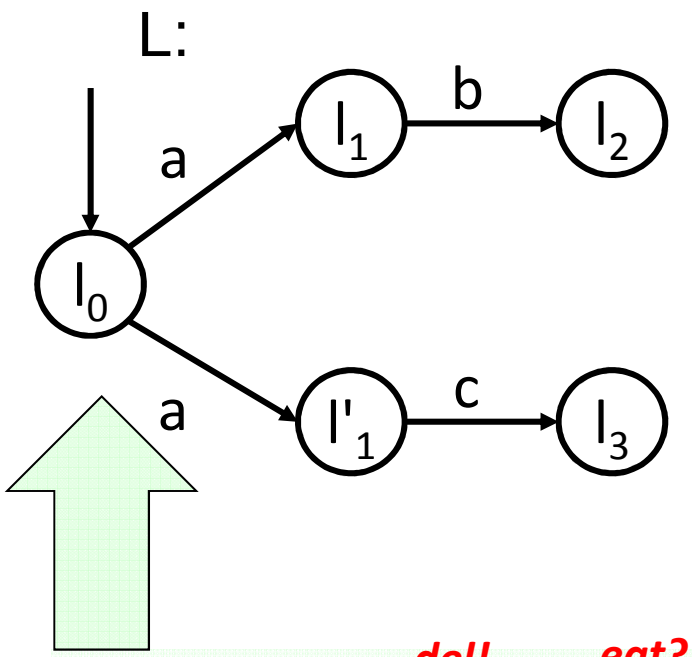
1. **Their definition: CCS**
2. **Their canonical equivalence: Bisimulation**
3. Their properties: Temporal Logic
4. Their verification: Model Checking

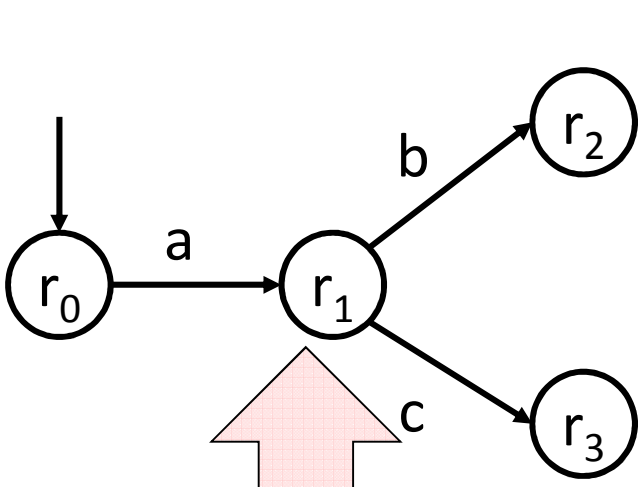


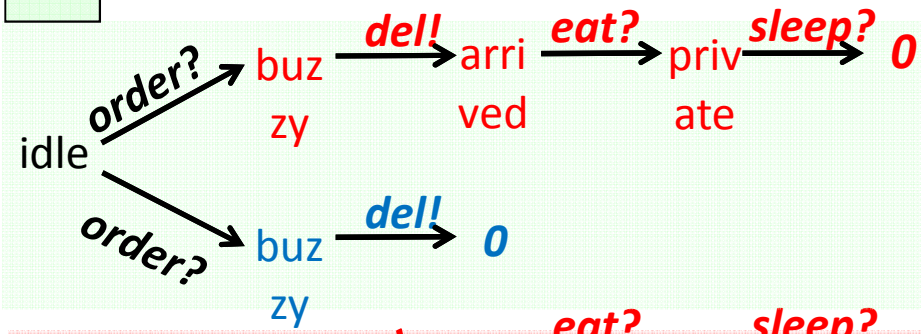
Theory of Programming

Prof. Dr. W. Reisig

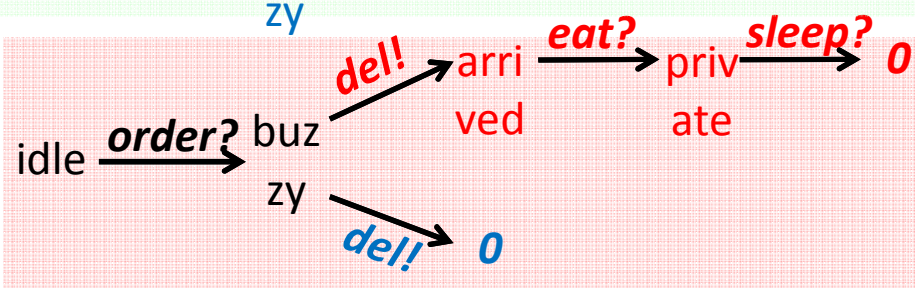
Reminder





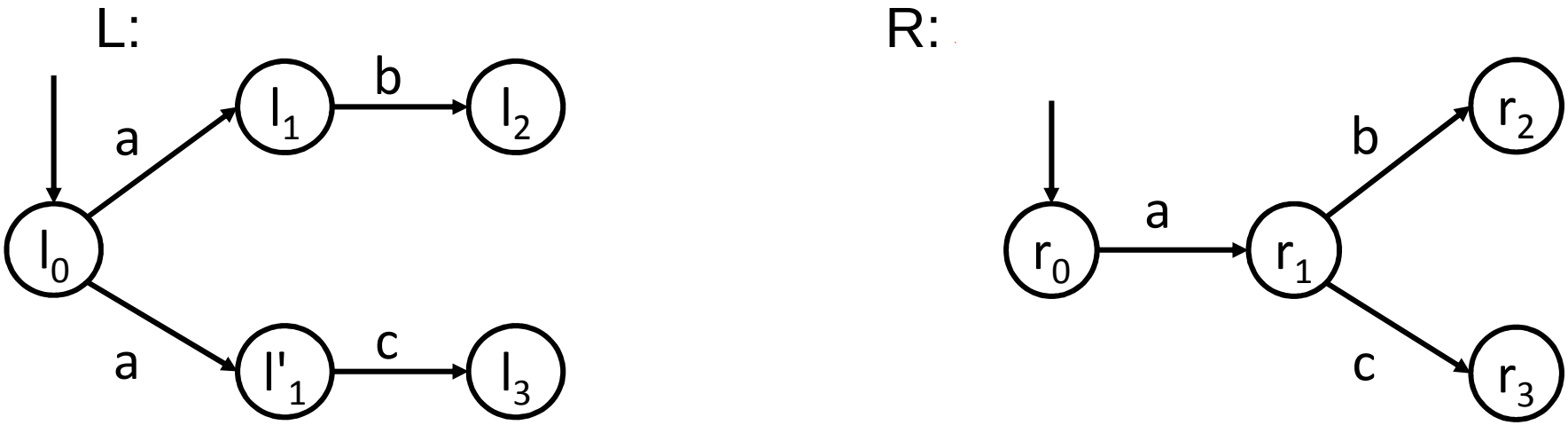


Pizza Hut': ***order?. del!.eat?.sleep?. 0***
+ order?. del!. 0



Pizza Hut'' :
order?. (del!.eat?.sleep!. 0 + del!. 0)

A problem of quivalence



L has two traces: a.b, a.c

„Systems with same traces are equivalent!“

R has same traces.

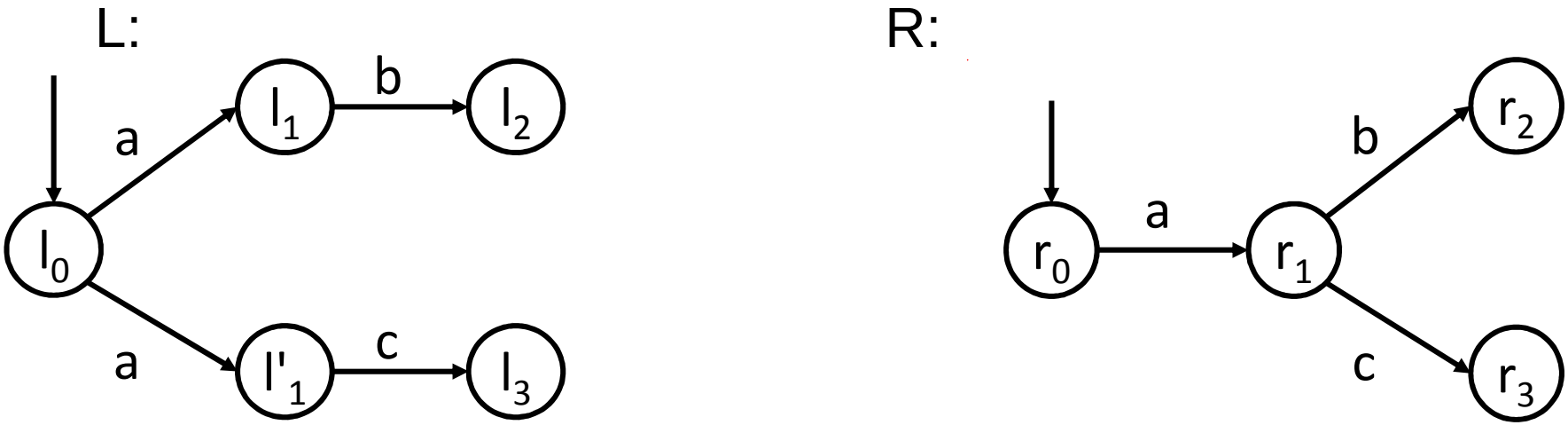
L and R are *not* equivalent, … by no means!

R is „more liberal“ than L :

R *simulates* L

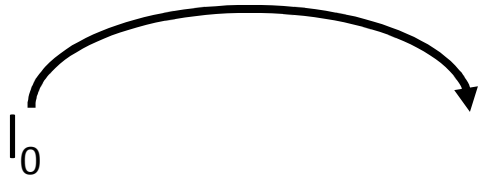
L does *not* simulate R

A problem of quivalence



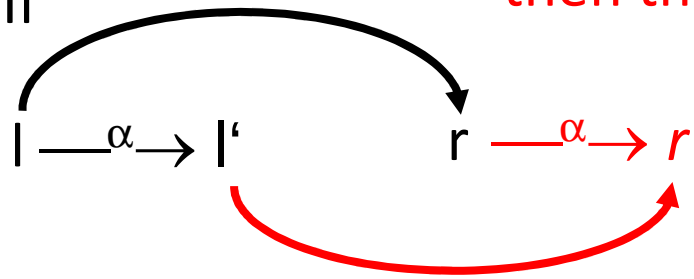
Def. L is simulated by R with ⌒ iff

a)



„ l_0 is simulated by r_0 “

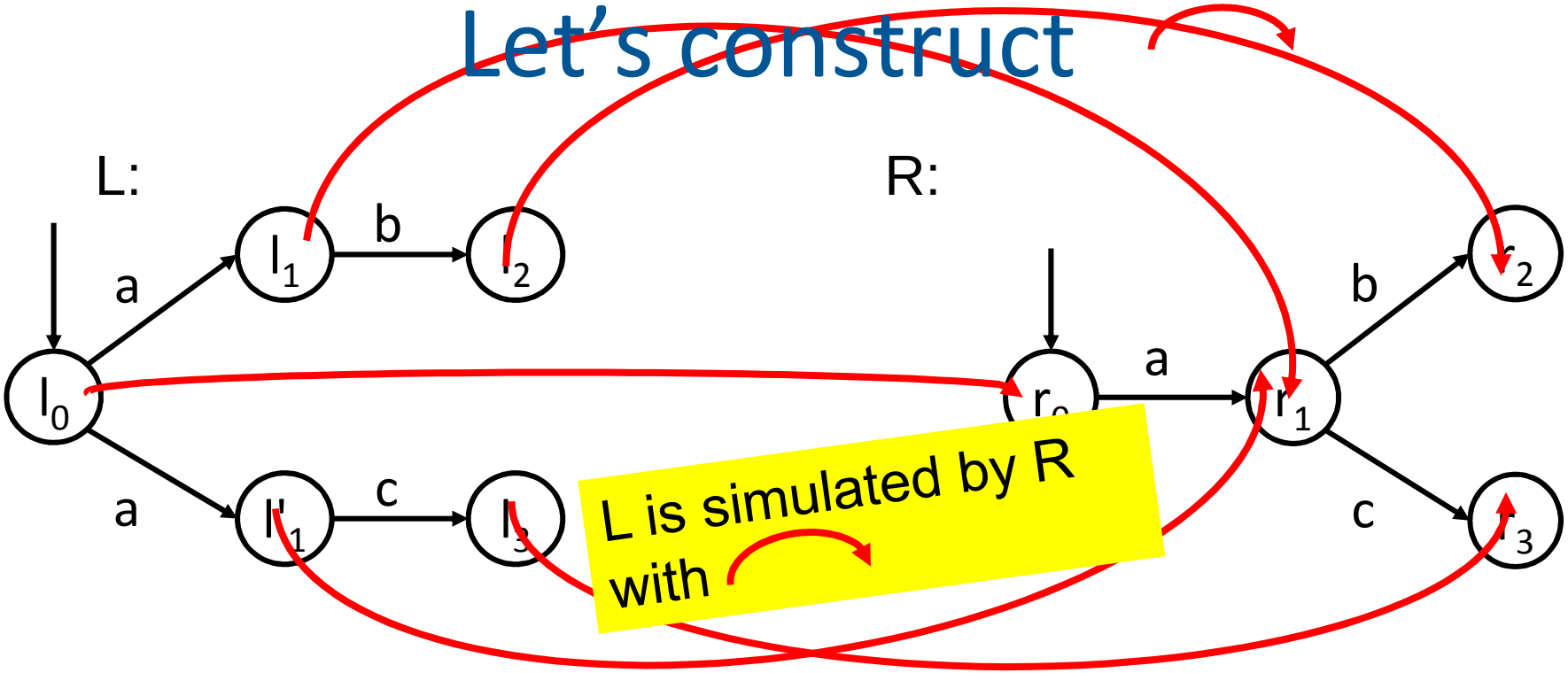
b) If



then there exists

$l \xrightarrow{\alpha} l'$ $r \xrightarrow{\alpha} r'$ and l' is simulated by r'

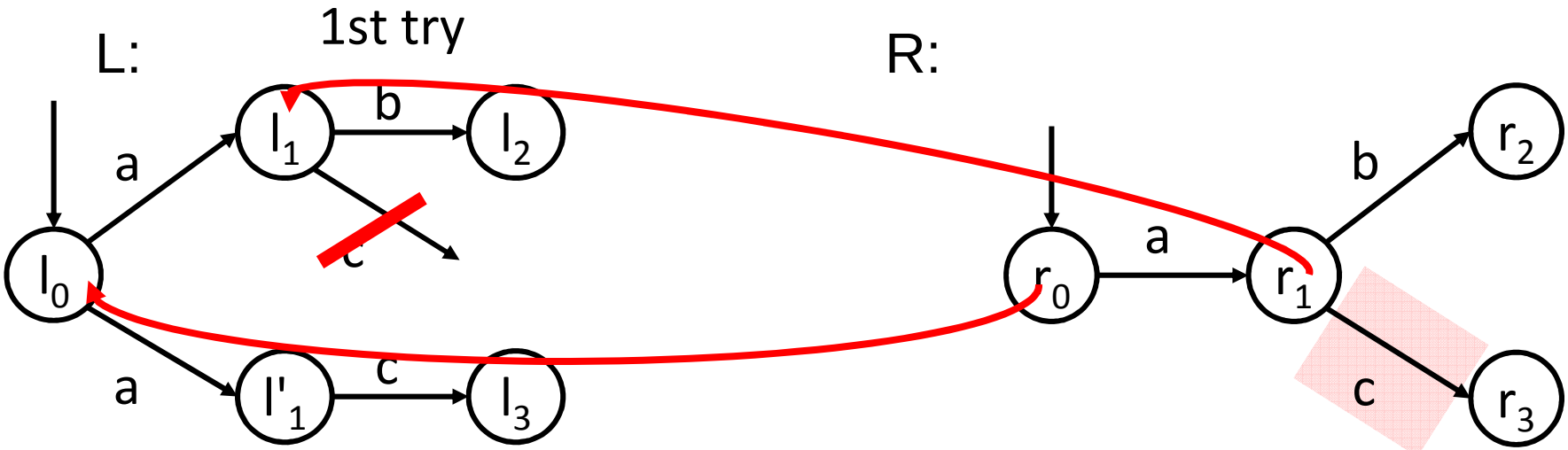
Let's construct



a) l_0 r_0 „ l_0 is simulated by r_0 “

b) If $l \xrightarrow{\alpha} l'$ then there exists $r \xrightarrow{\alpha} r'$ and l' is simulated by r'

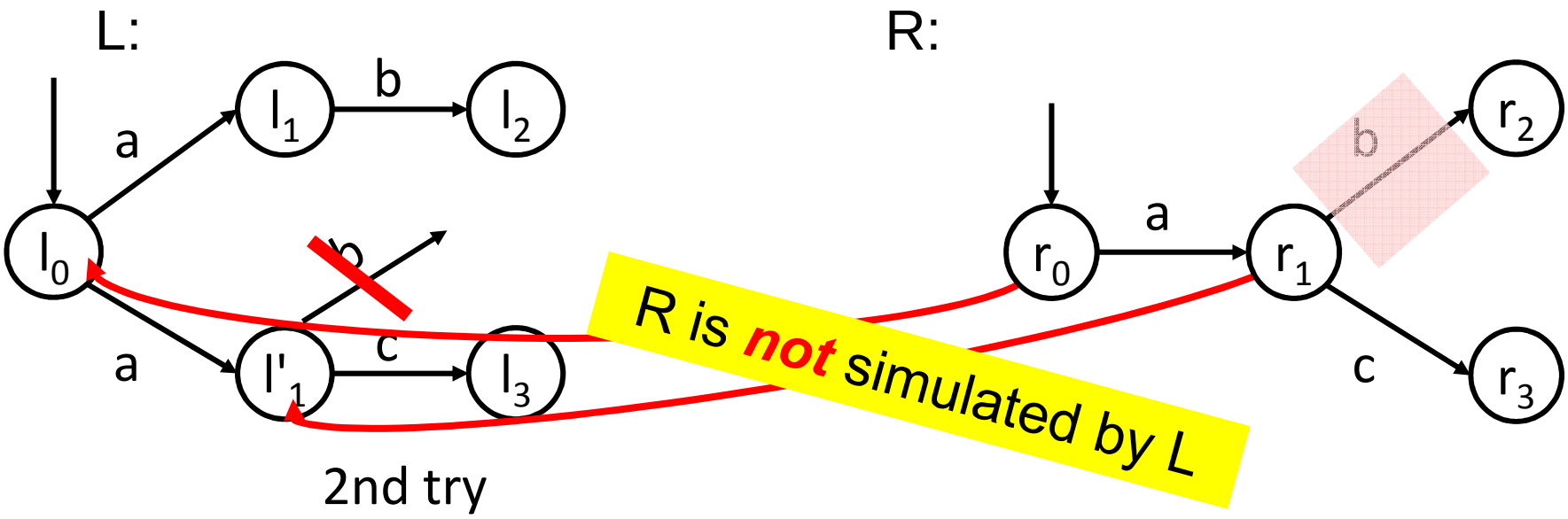
Vice versa



a) l_0 r_0 „ l_0 is simulated by r_0 “

b) If $l \xrightarrow{\alpha} l'$ then there exists $r \xrightarrow{\alpha} r'$ and l' is simulated by r'

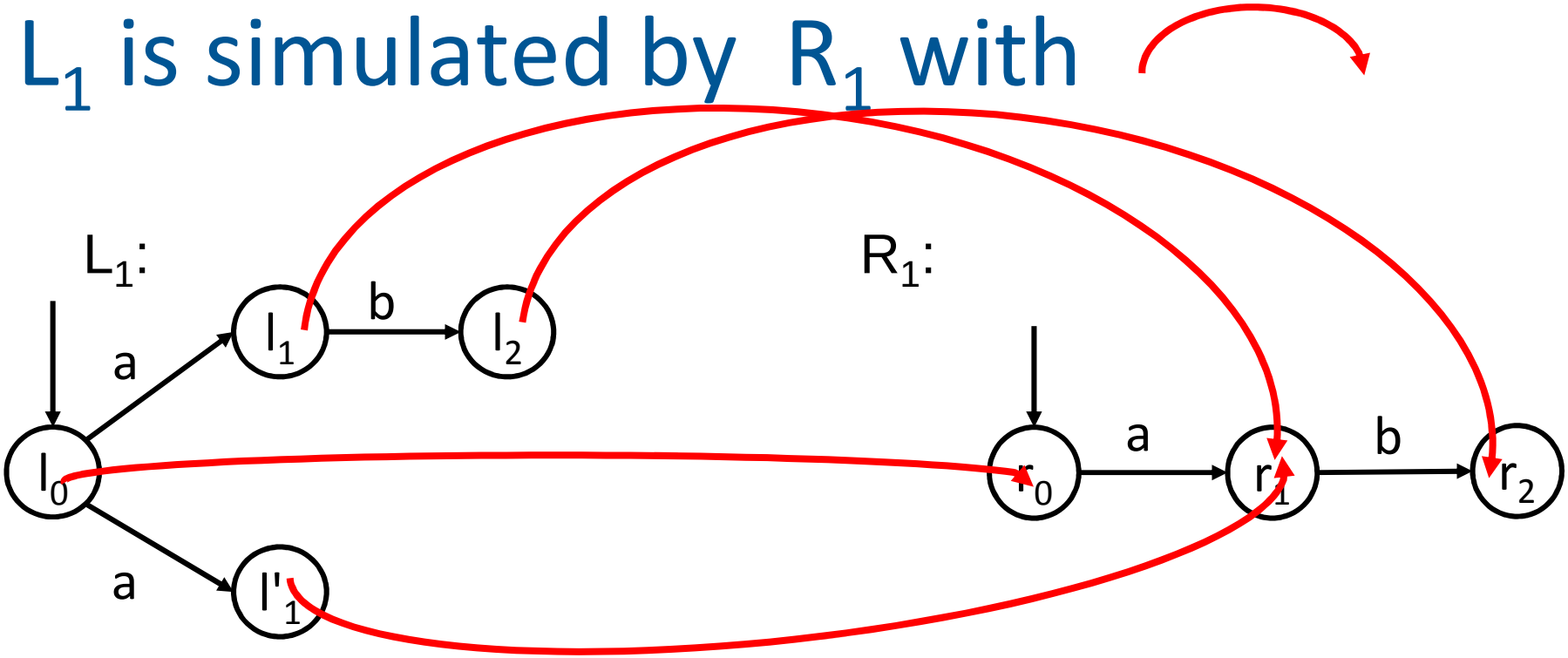
Vice versa



a) l_0 r_0 „ l_0 is simulated by r_0 “

b) If $l \xrightarrow{\alpha} l'$ then there exists $r \xrightarrow{\alpha} r'$ and l' is simulated by r'

L_1 is simulated by R_1 with

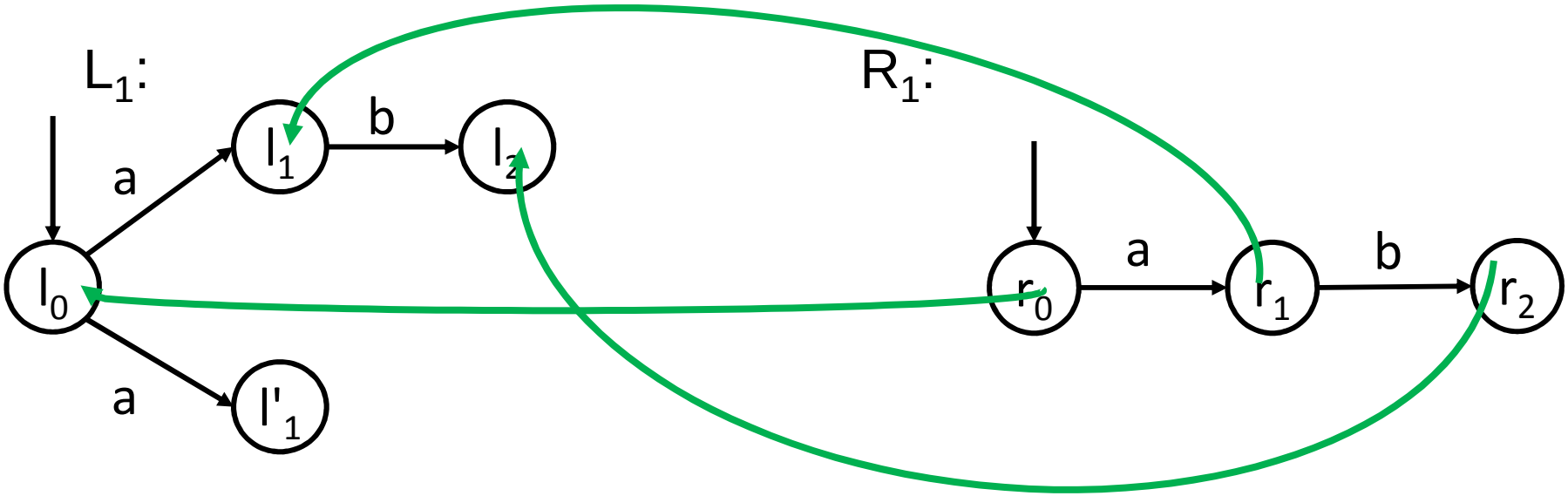


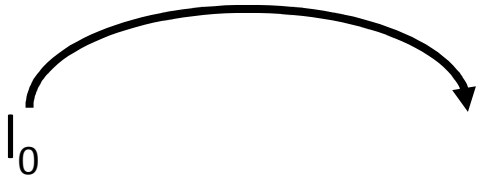
a) l_0 r_0 „ l_0 is simulated by r_0 “

b) If $l \xrightarrow{\alpha} l'$ then there exists $r \xrightarrow{\alpha} r'$ and l' is simulated by r'

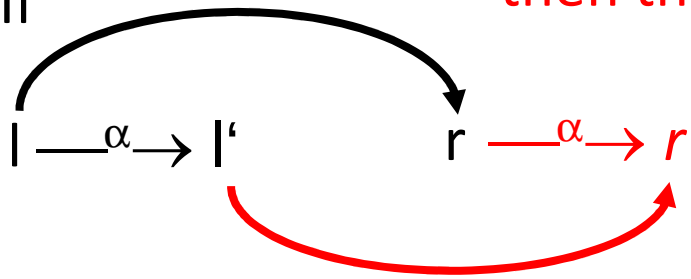
30

R_1 is simulated by L_1 with 



a) 

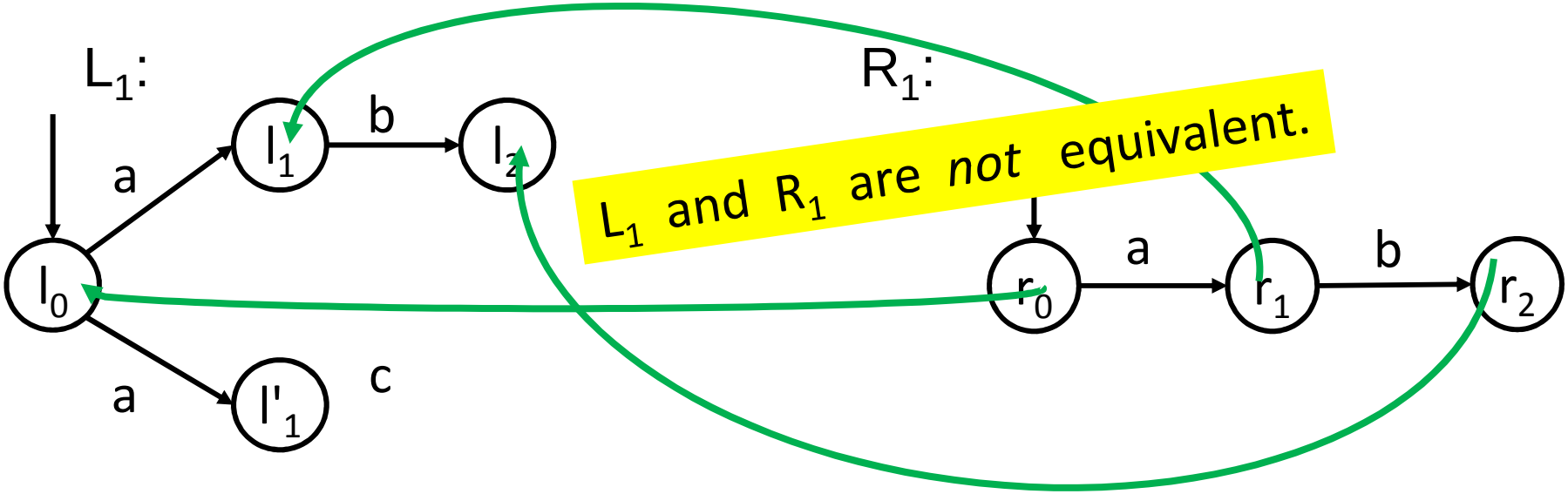
„ l_0 is simulated by r_0 “

b) If 

then there exists

and l‘ is simulated by r‘

R_1 is simulated by L_1 with 

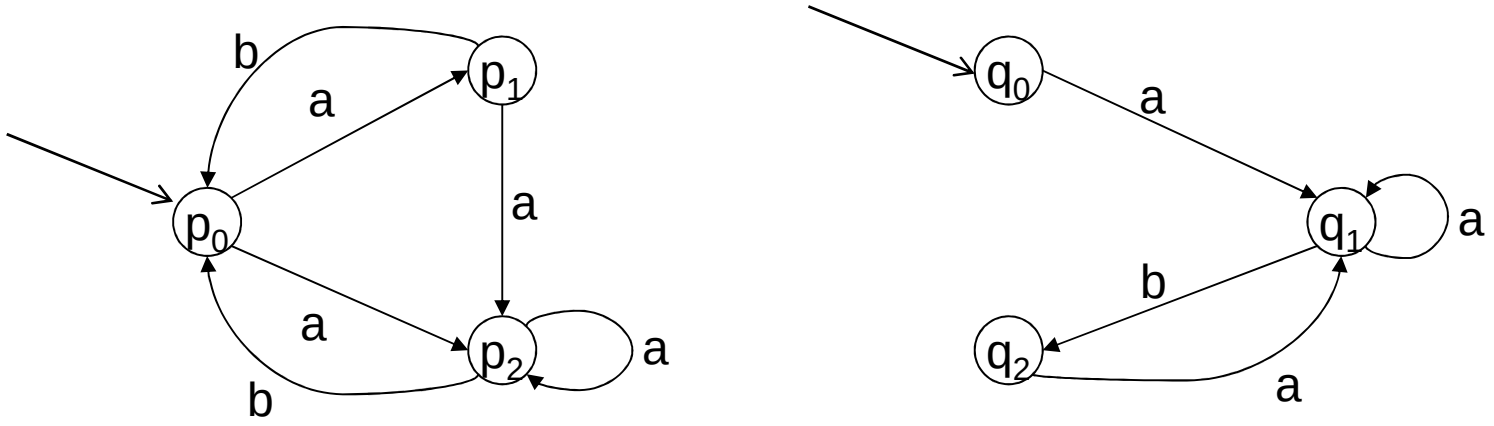


Def. L and R are *equivalent* iff they simulate one another.

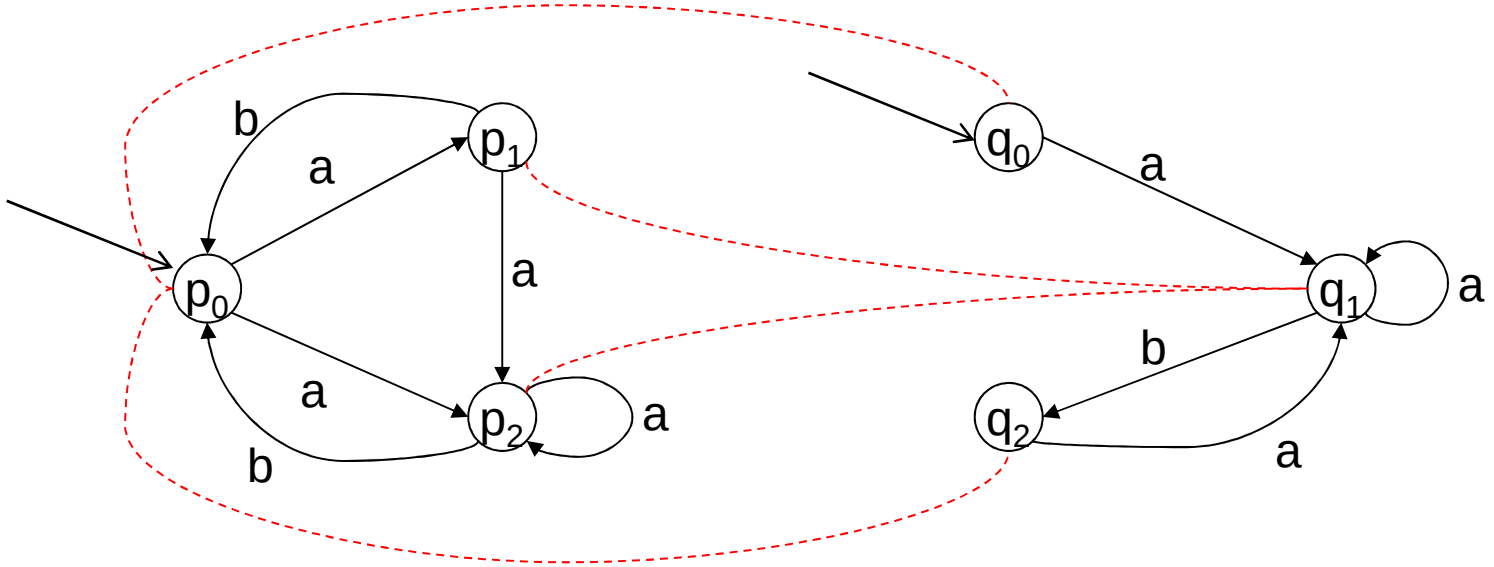
Def. L and R are *equivalent* iff for some  ,
R simulates L by  , and
L simulates R by $(\frown)^{-1}$.

(L and R are ***bisimular***)

Bisimulation: yet another example



Bisimulation: yet another example



Why so complicated?

Why not

Def. L and R are *equivalent* iff they simulate one another.

Why no rules

$\mathbf{0+0} \xrightarrow{\tau} \mathbf{0}$

$\mathbf{0|0} \xrightarrow{\tau} \mathbf{0}$

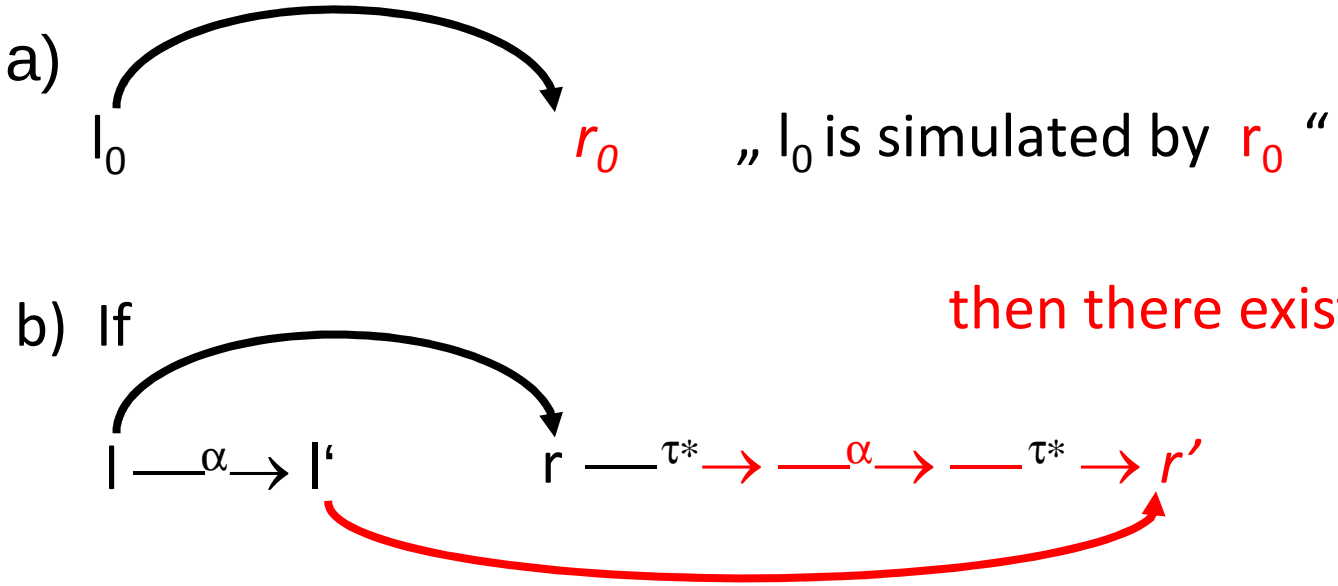
?

Because we want *Compositionality* (equivalence to be a *congruence*):

Theorem.
Let P, Q, R be processes,
Let P and Q be equivalent,
written $P \sim Q$.

Then
$P+R \sim Q+R$
$\alpha.P \sim \alpha.Q$
$P|R \sim Q|R$
$P/B \sim Q/B$

Variant: L is *wekly simulated* by R

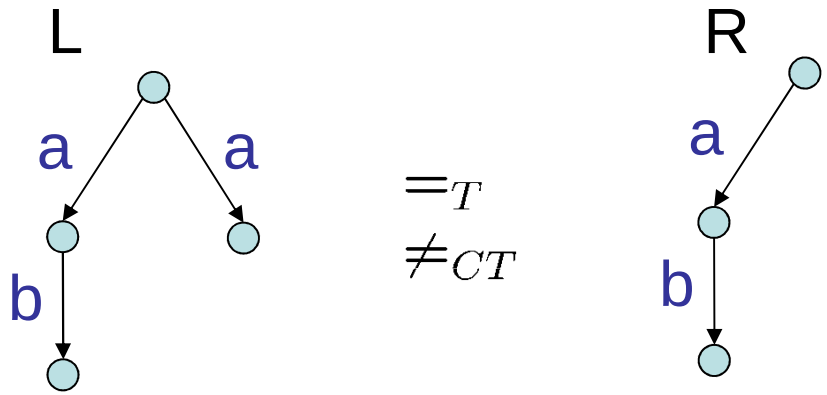


Caution!

Weak bisimulation is no congruence

Complete Trace Equivalence

Combining termination and choice …

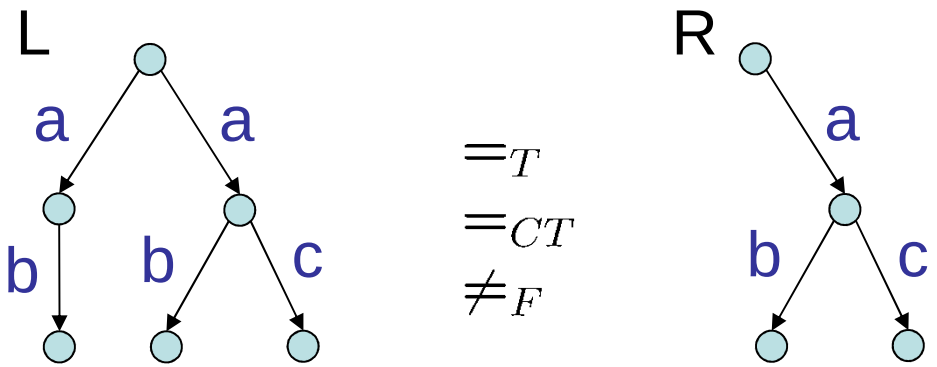


a is a complete trace of L but not of R

Failure Equivalence of a set M of actions

Def.: For an action w and a set of actions M:
[w,M] is a *failure pair* of P iff P may do a step

$P \xrightarrow{w} Q$ and no action of M is enabled in Q.

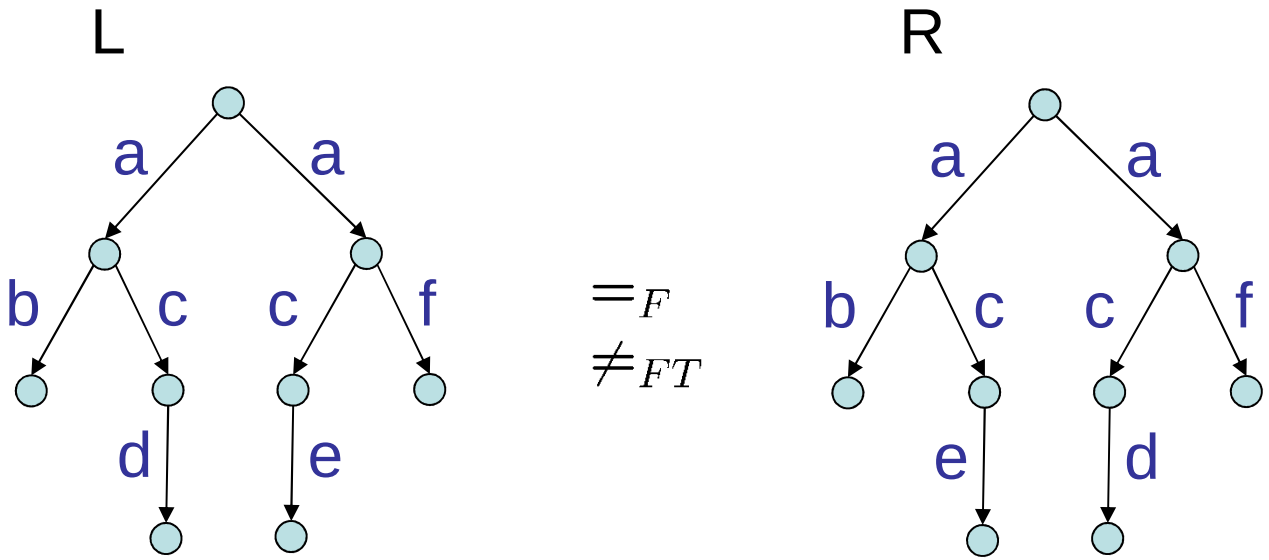


[a,{c}] is a failure pair of L but not of R

Failure Trace Equivalence

… like Failure equivalence.

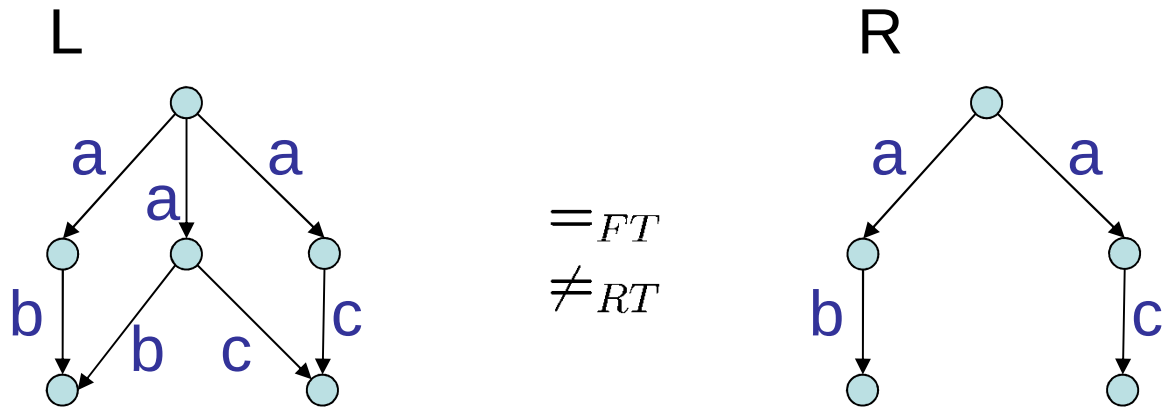
But now you continue along a trace



a {f} c {e} d is a failure trace of L but not of R

Ready Trace Equivalence

In a trace, between each two actions,
present the alternative actions.

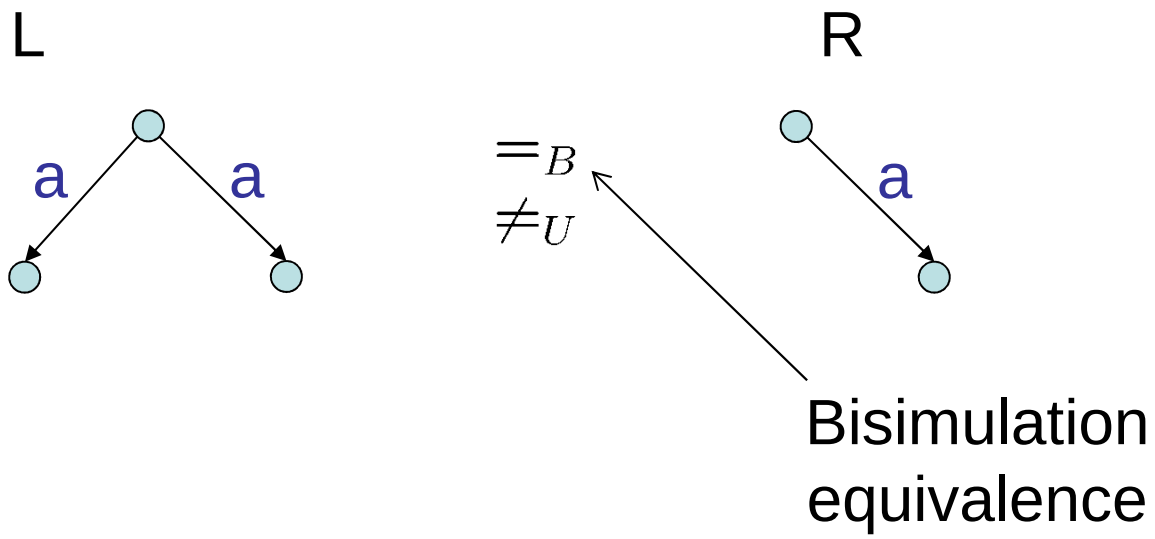


[a,{c},b] is a ready trace of L but not of R

Tree Equivalence

Unfold the transition systems as trees

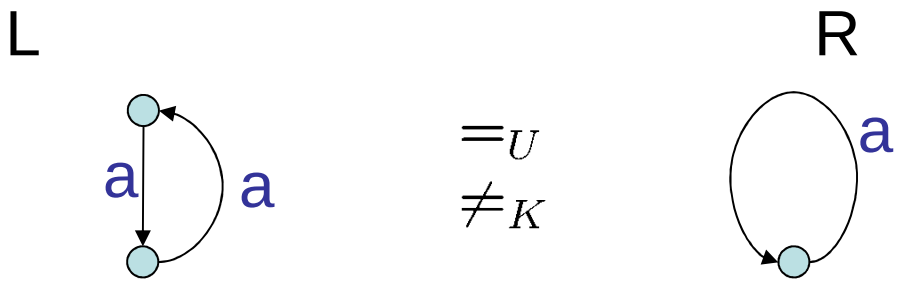
L $=_U$ R iff both trees are isomorpic



Structural Equivalence

Equivalence:

L $=_K$ R iff the transtion systems are isomorphic



Further equivalences

Ready equivalence

Ready Simulation equivalence

Ready Trace Simulation equivalence

Completed Simulation equivalence

Failure Simulation equivalence

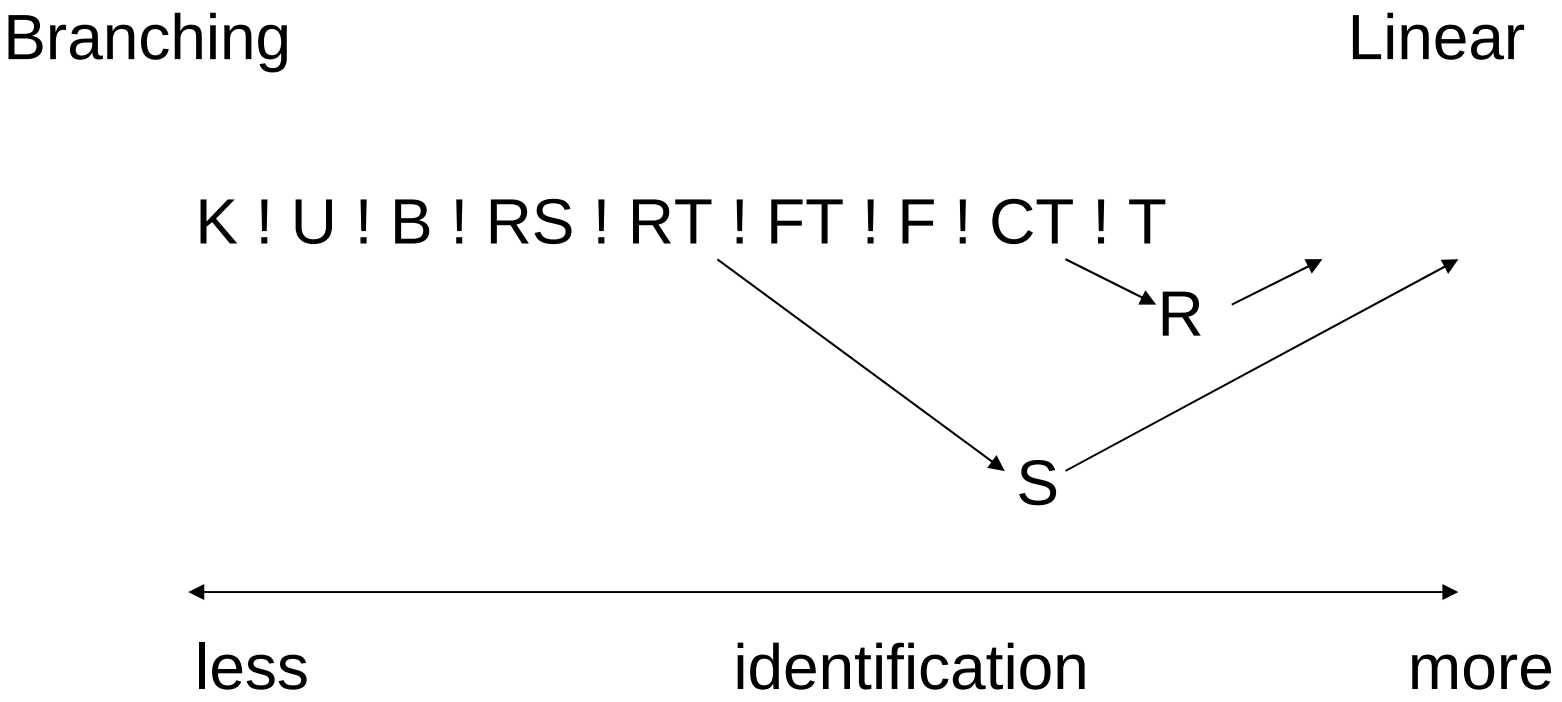
Failure Trace Simulation equivalence

Simulation equivalence

…

152 ones

The Linear Time – Branching Time Spectrum



Synchronous Processes etc: their highlights in brief

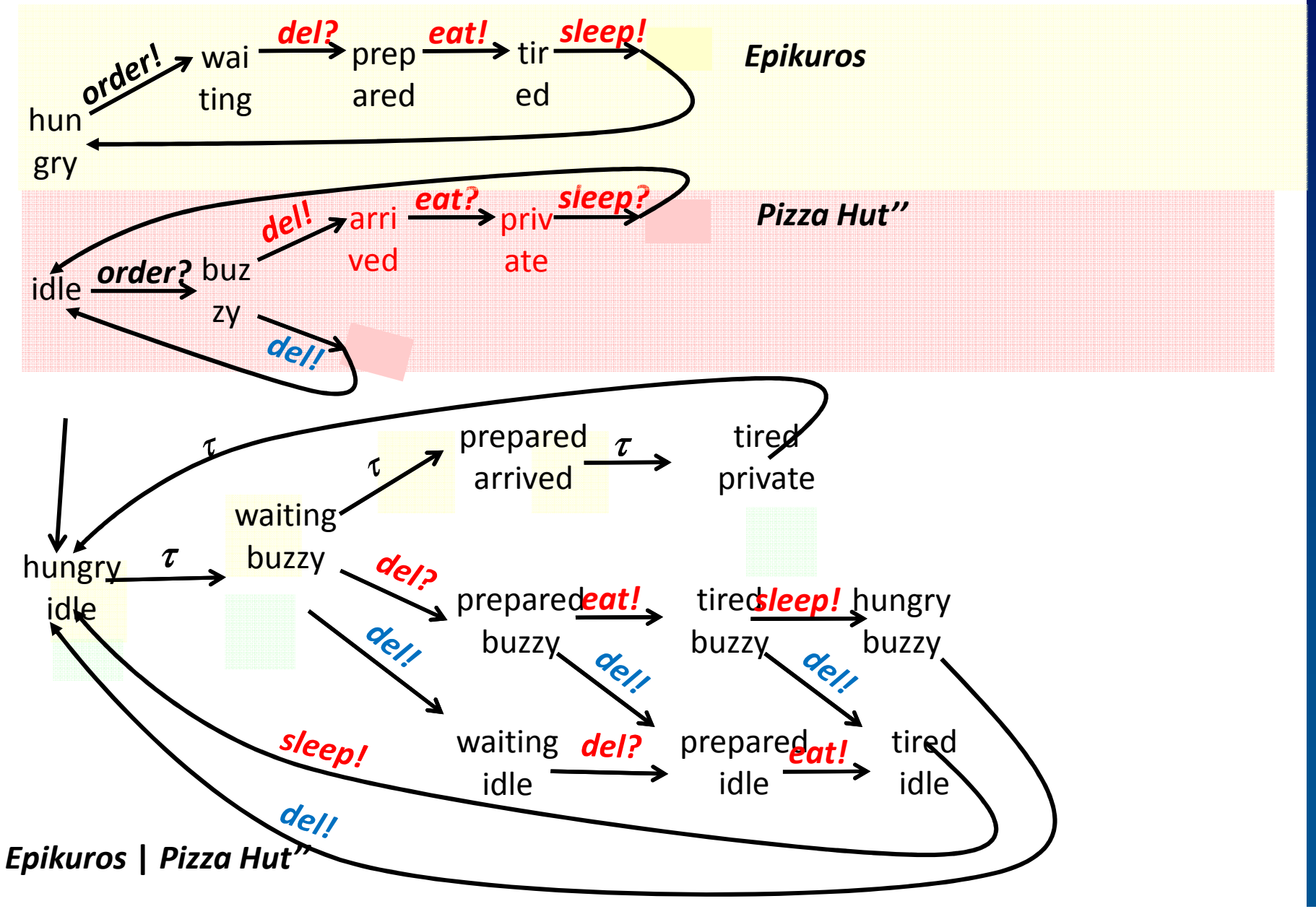
1. Their definition: CCS
2. Their canonical equivalence: Bisimulation
3. **Their properties: Temporal Logic**
4. Their verification: Model Checking

Humboldt Universität Informatik

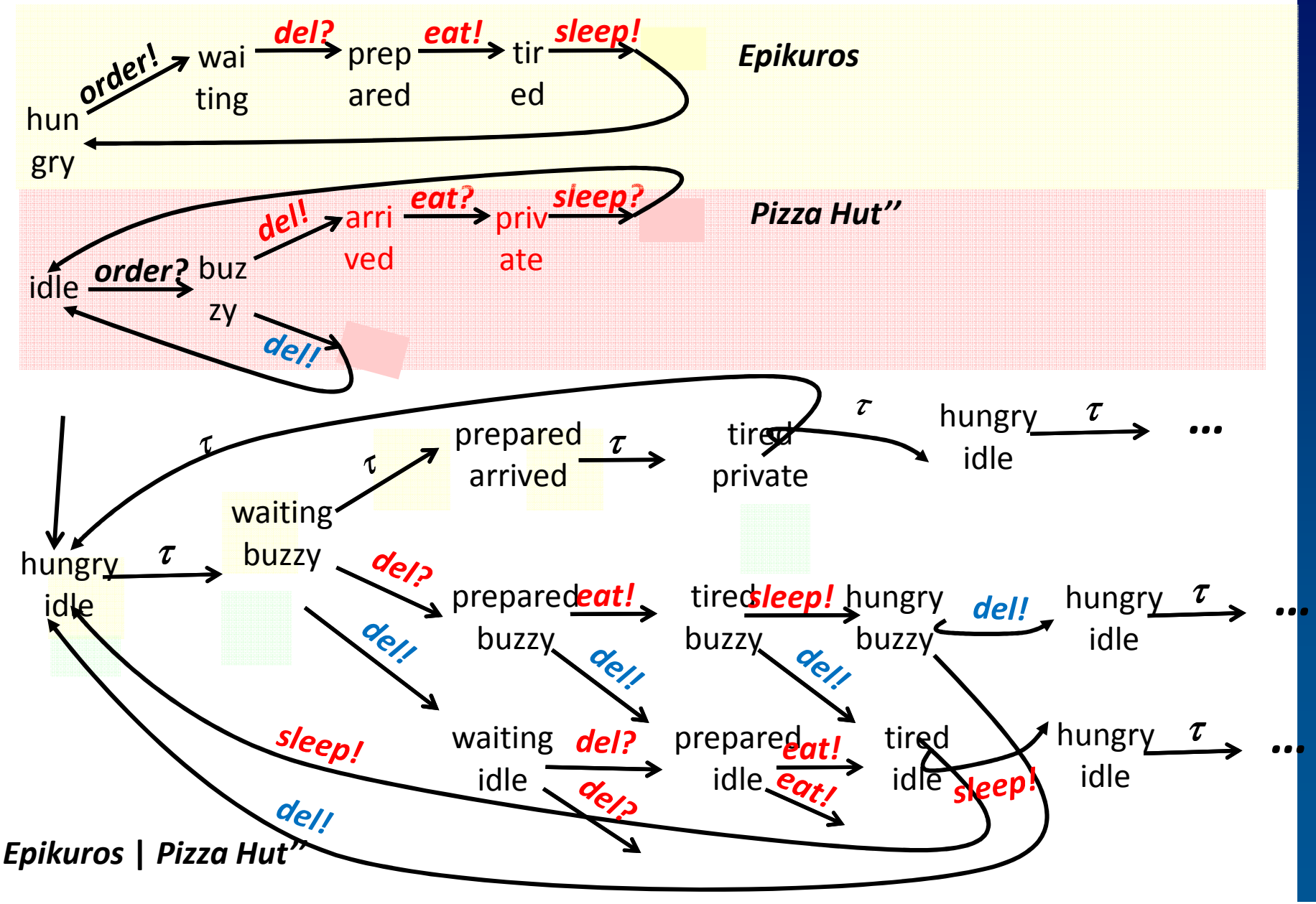
T o P
Theory of Programming

Prof. Dr. W. Reisig

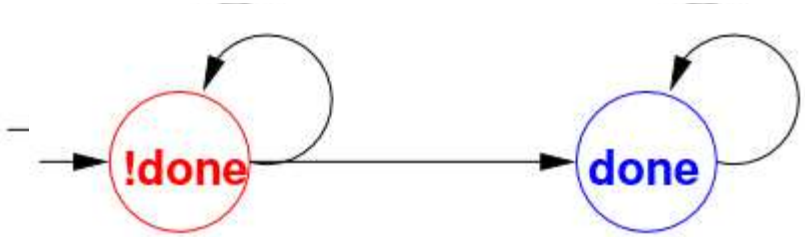
The computation graph

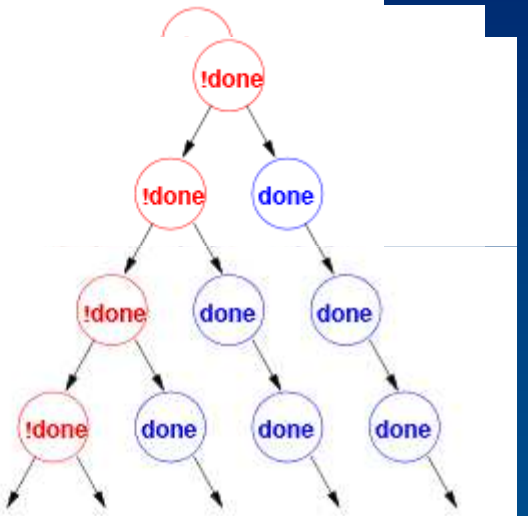


Make it a tree

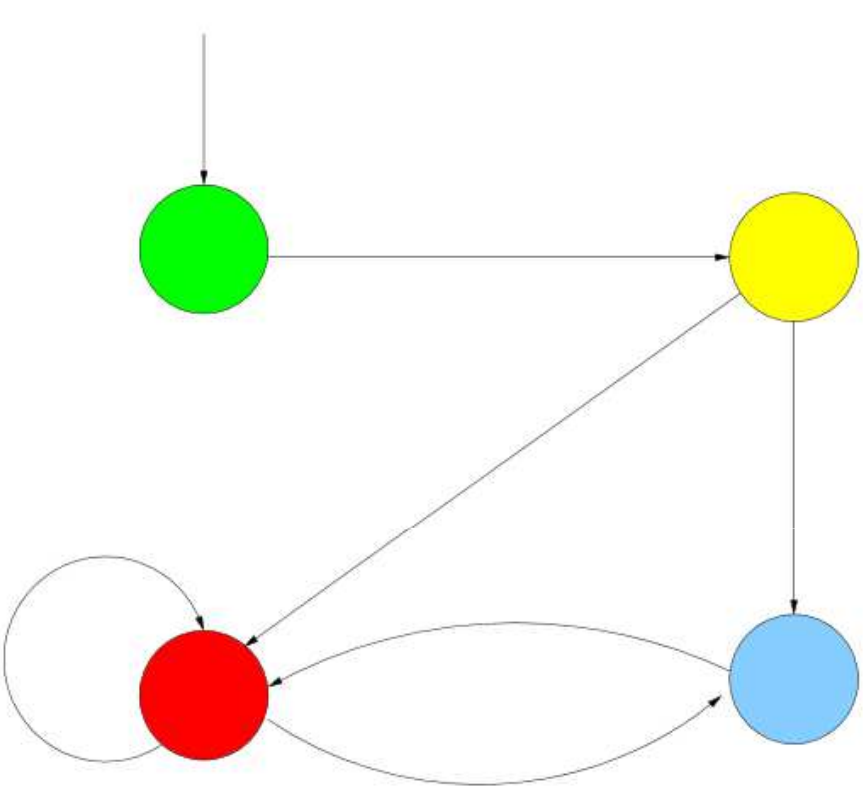


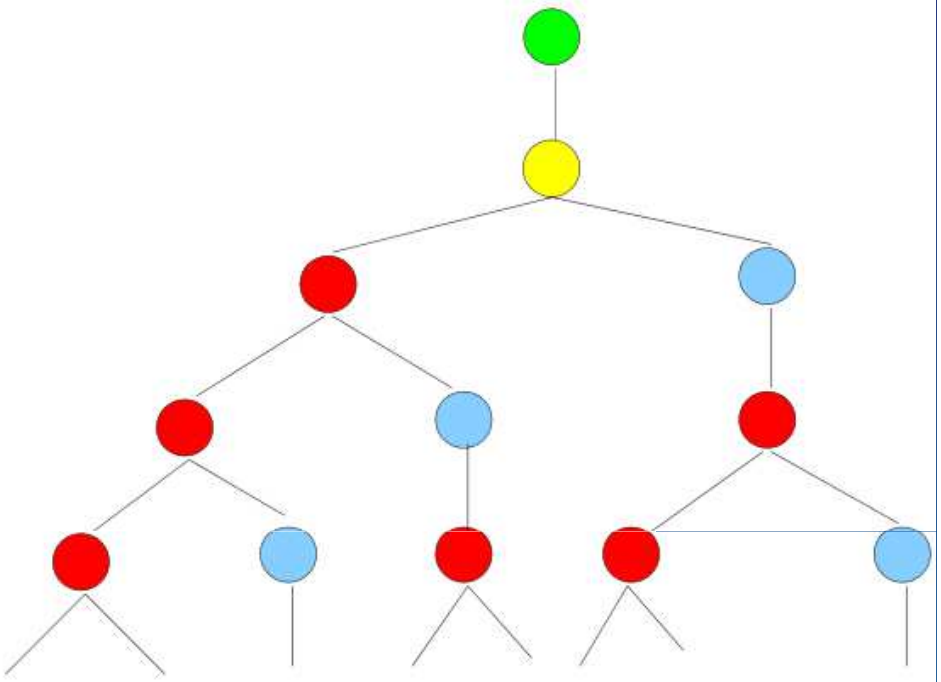
From a transition graph to its tree





Once more: a process and its tree





Computation Tree Logic *CTL**

p = 

finally p

globally p

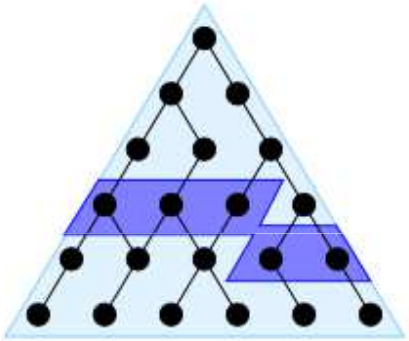
next p

p until q

Computation Tree Logic *CTL**

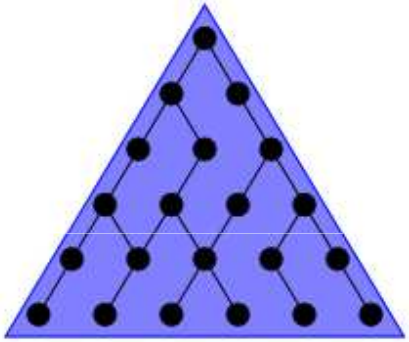
p = 

finally p



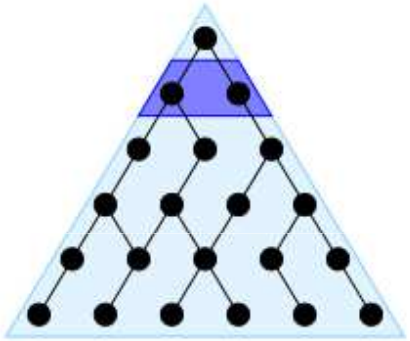
AF p

globally p



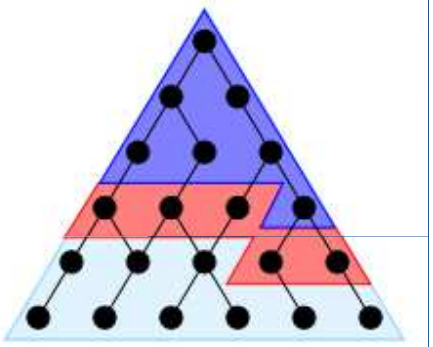
AG p

next p

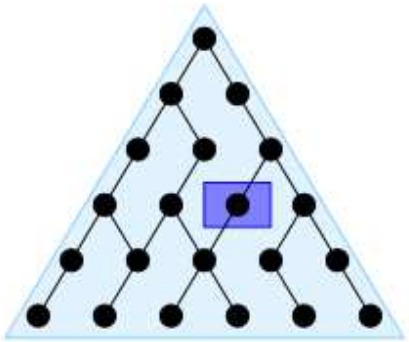


AX p

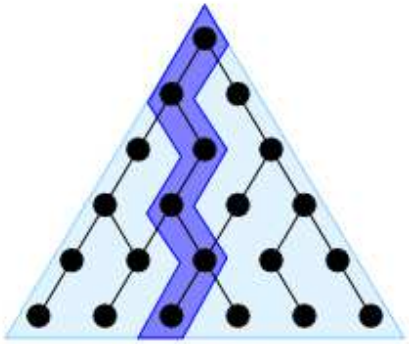
p until q



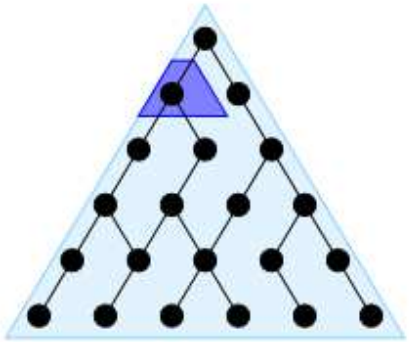
A [p U q]



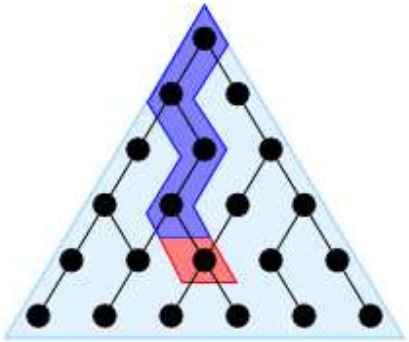
EF p



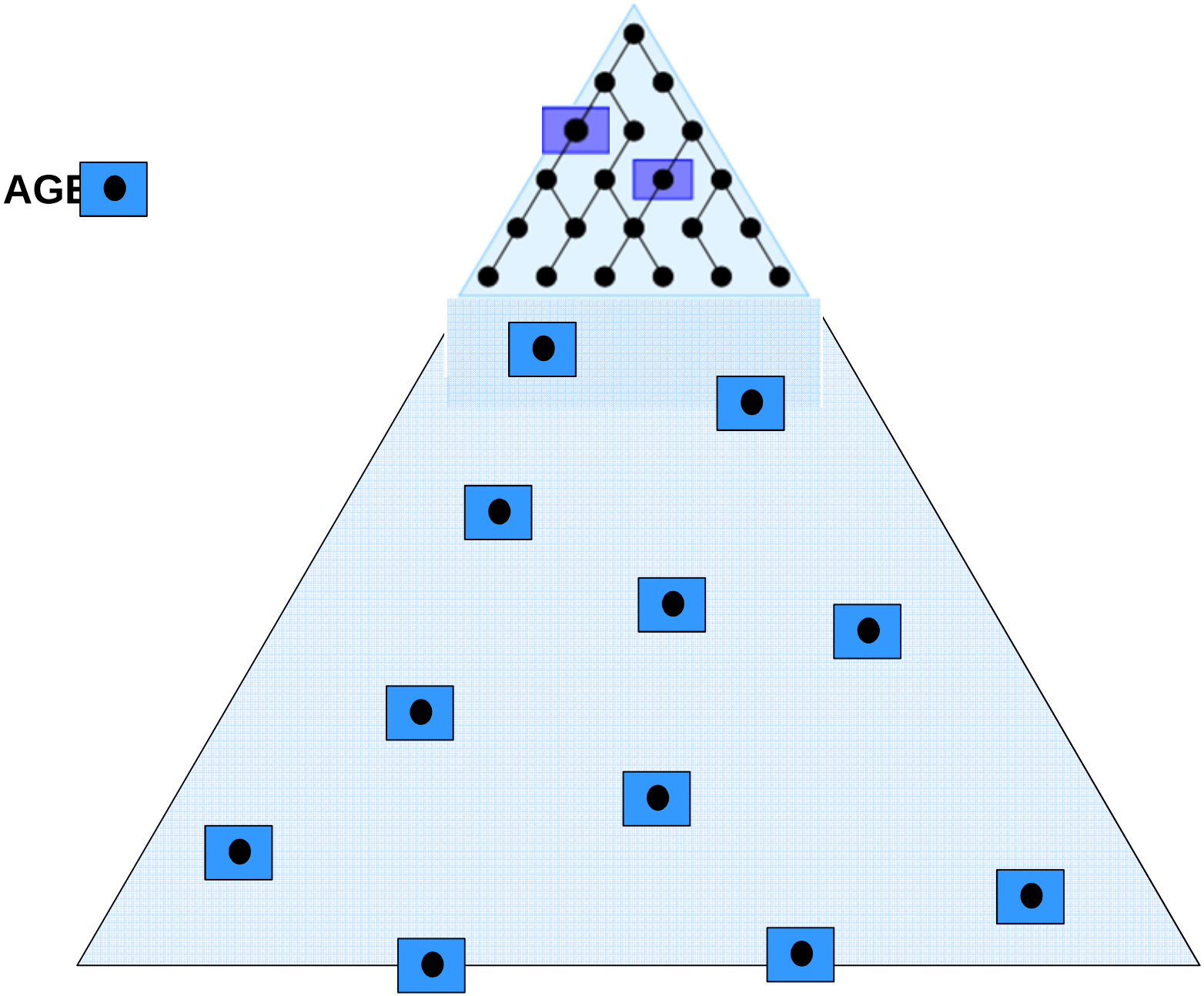
EG p



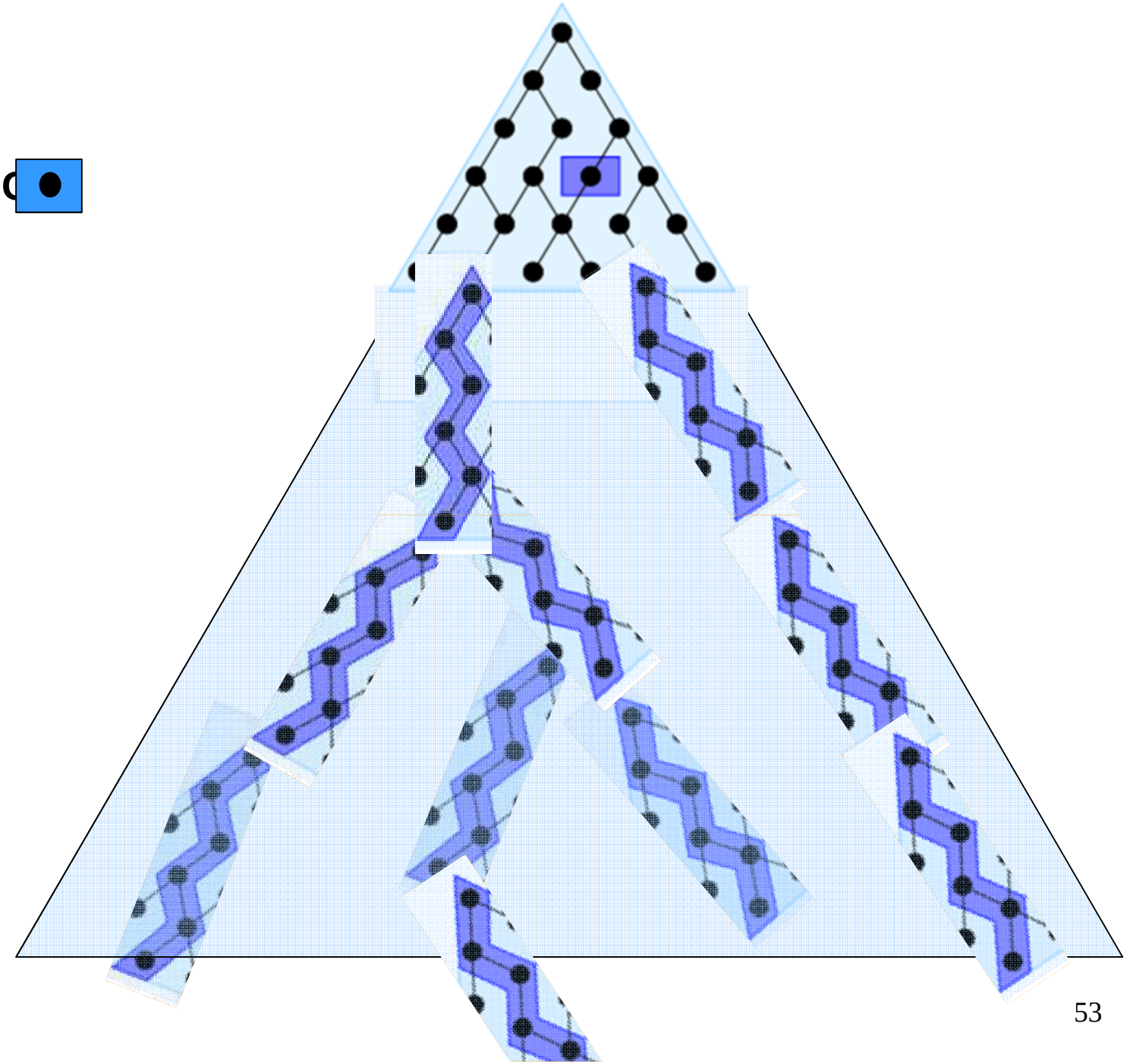
EX p



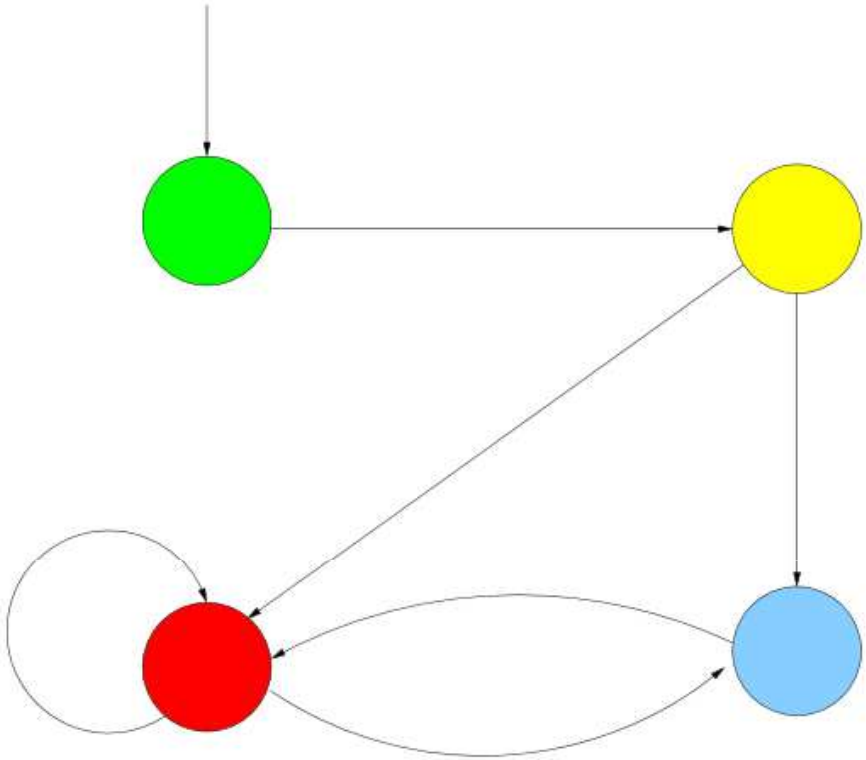
E [p U q]

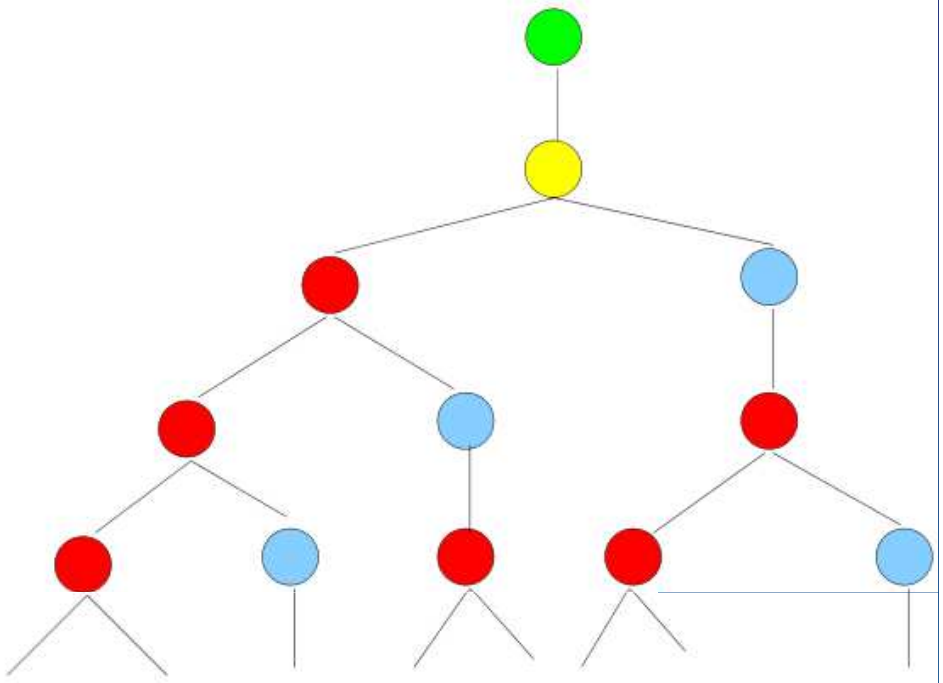

AGE

AGEC



Valid formulas





EX 

AX 

AG ( v EX )

AGEF 

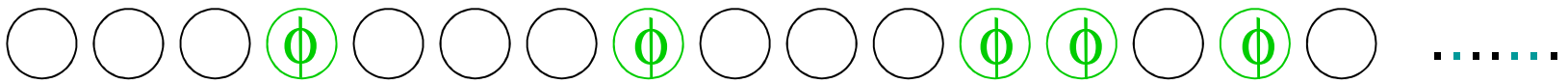
EFG 

Typical applications

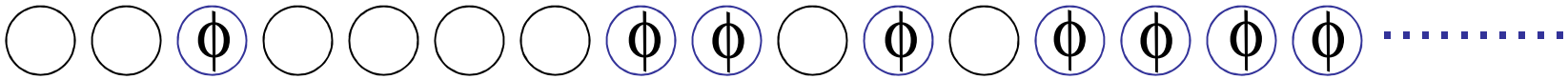
“Never something bad happens”	AG *safely*
“No deadlock reachable”	AG *enabled*
„You can click to reach p“	EF *p*
“Whatever happens – you will succeed”	AF *Goal*
“Each requirement is followed by an acknowledgement”	AG(*req* u AF *ack*)
“It makes sense to wait”	AG AF *avail*
“You always can properly terminate”	AG EF *exit*

Combining F and G

G F ϕ = ϕ holds infinitely often



F G ϕ = ϕ stabilizes



G (ϕ ◆ F ψ) = ϕ leads to ψ

Tautologies: F G F ϕ ◆ G F ϕ G F G ϕ ◆ F G ϕ

Why not just First order logic (predicate logic)?

Example:

Whenever process A sends a message to process B, then B eventually sends an acknowledgement to A.

First order:

$\forall$ t (send(A,B,t) $\rightarrow$ $\forall$ t' (greater(t',t) $\rightarrow$ send(B,A,t')))

CTL*:

AG (Send (A,B) $\rightarrow$ AF Send (B,A))

Expressiveness

Why just ***THIS*** logic?

Theorem.

Two states are bisimilar

iff they satisfy the same *CTL** properties.

Consequence:

Specify a system in terms of CTL*.

This may yield various different implementations.

They all are bisimular.

Synchronous Processes etc: their highlights in brief

1. Their definition: CCS
2. Their canonical equivalence: Bisimulation
3. Their properties: CTL*
4. **Their verification: Model Checking**

Humboldt Universität Informatik

Theory of Programming

Prof. Dr. W. Reisig

Why verify a system design?

to prove its correctness (theoretically)

To find subtle mistakes (practically)

In contrast: Testing
Testing shows presence of mistakes,
but not their absence (E. Dijkstra)

Verification techniques

manually	Hoare Logic, Invariants, descending functions, ...
interactive	Theorem proving
automatically	Model Checking

Model Checking

Aim: Show that a CTL* formula ϕ holds in a transition system T .

Idea: Visit each state of T and derive its properties.
Combine the results to prove ϕ

First relevant results: 1986

Brake through: 1992

… a success story
with a fundamental problem:
state explosion

State Explosion

Assume: 2.4 GHz, sufficient store,
one new state per clock cycle:
how many states can you visit?

2,400,000,000 per second

144,000,000,000 per minute

8,840,000,000,000 per hour

207,360,000,000,000 per day

75,738,240,000,000,000 per year

1,514,764,800,000,000,000,000,000 since big bang
($< 10^{28}$)

Systems with 10^{28} states

Theoretically: 90 boolean variables

Praktically: 200 boolean variables (in distributed sytems)

Milestones of Model Checking:

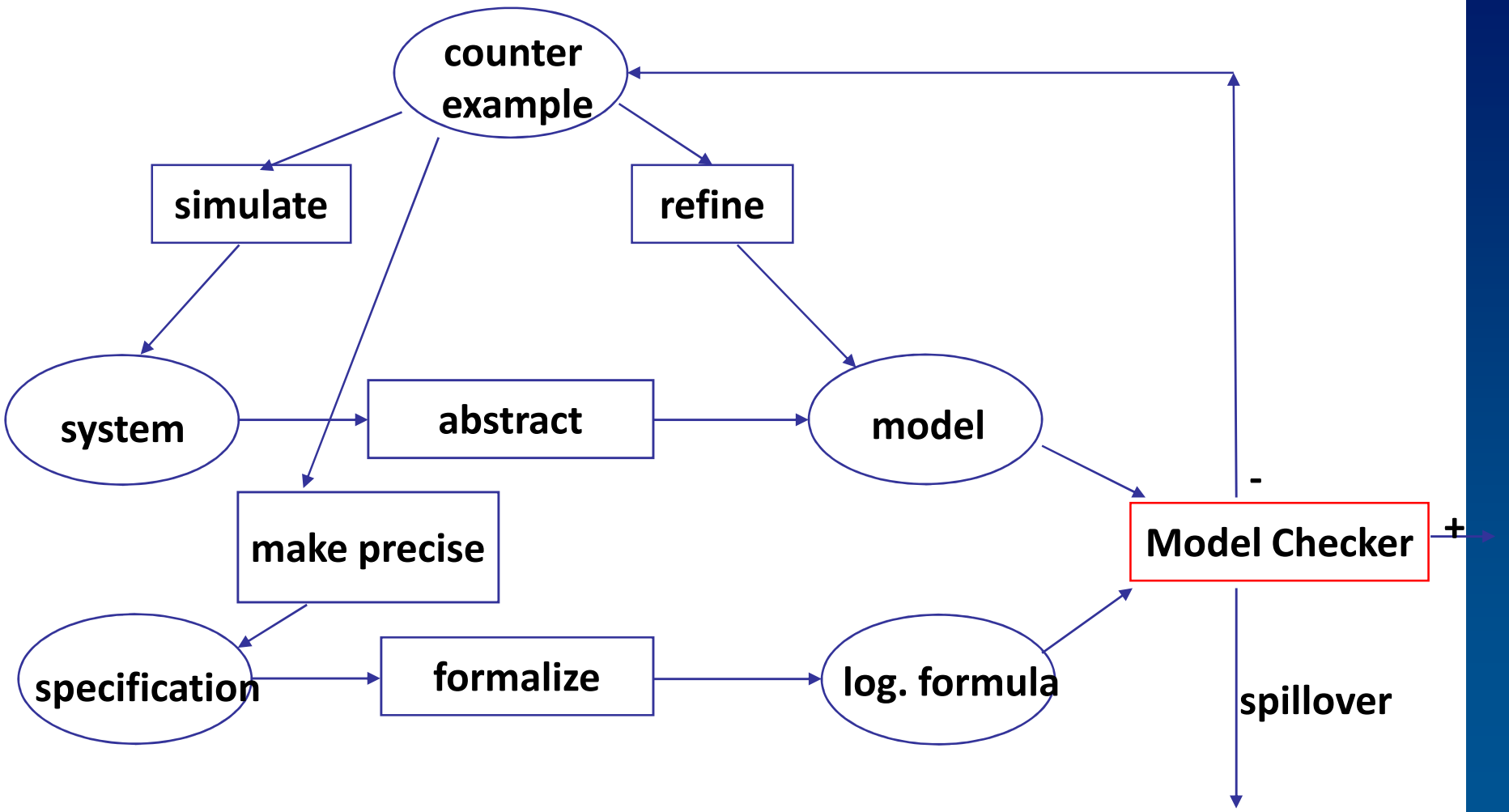
1986: 10^6
1992: 10^{20}
1996: 10^{100}
2000: 10^{1000}

A miracle?
Cheating?
Clever technolgy?

Supporting techniques:

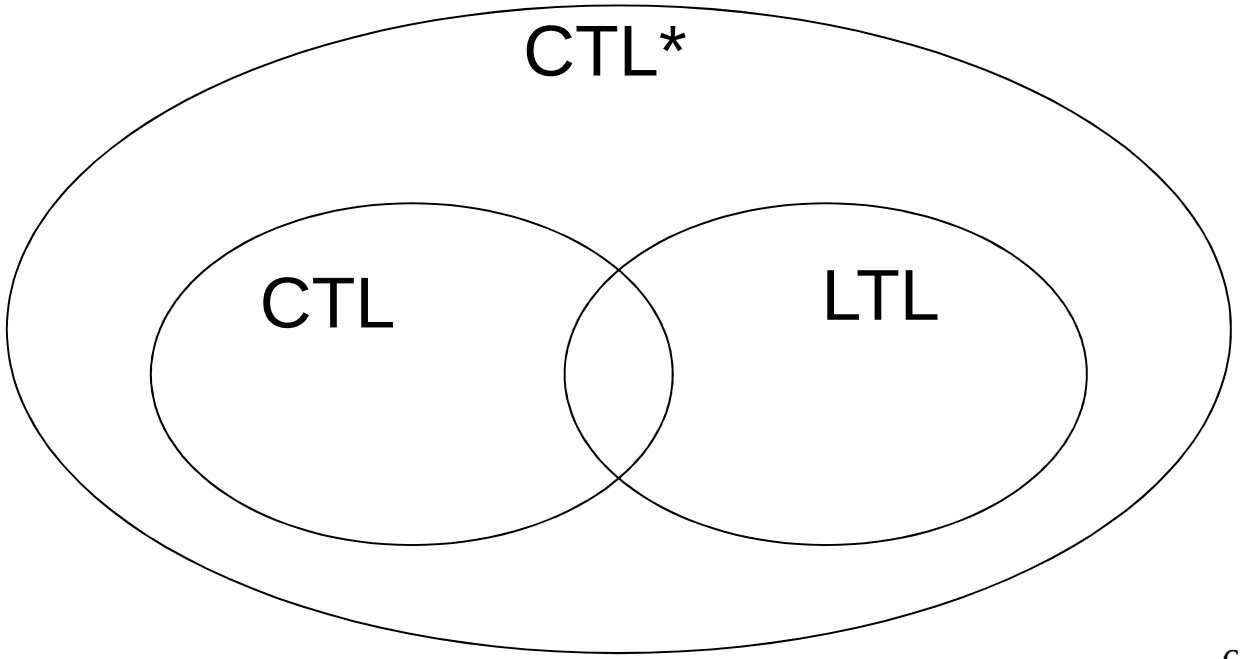
Abstract interpretation,
Symbolic Model checking.

Model Checking: How to use it



Efficient algorithms

… not for CTL*,
but for subsets of it



Path Formulas

proposition p

$p \text{ ” } (s_0\ s_1\ s_2\ s_3 \ldots)$ iff $p \text{ ” } s_0$

X *path formula*

$X\ \phi \text{ ” } (s_0\ s_1\ s_2\ s_3 \ldots)$ iff $\phi \text{ ” } (s_1\ s_2\ s_3 \ldots)$

F *path formula*

$F\ \phi \text{ ” } (s_0\ s_1\ s_2\ s_3 \ldots)$ iff $\phi \text{ ” } (s_i\ s_{i+1}\ s_{i+2} \ldots)$ for some i

G *path formula*

$G\ \phi \text{ ” } (s_0\ s_1\ s_2\ s_3 \ldots)$ iff $\phi \text{ ” } (s_i\ s_{i+1}\ s_{i+2} \ldots)$ for all i

path formula U *path formula*

$\phi\ U\ \psi \text{ ” } (s_0\ s_1\ s_2\ s_3 \ldots)$ iff ...

State Formulas

E *path formula*

E ϕ ” s iff for some path π starting at s holds: ϕ ” π

A *path formula*

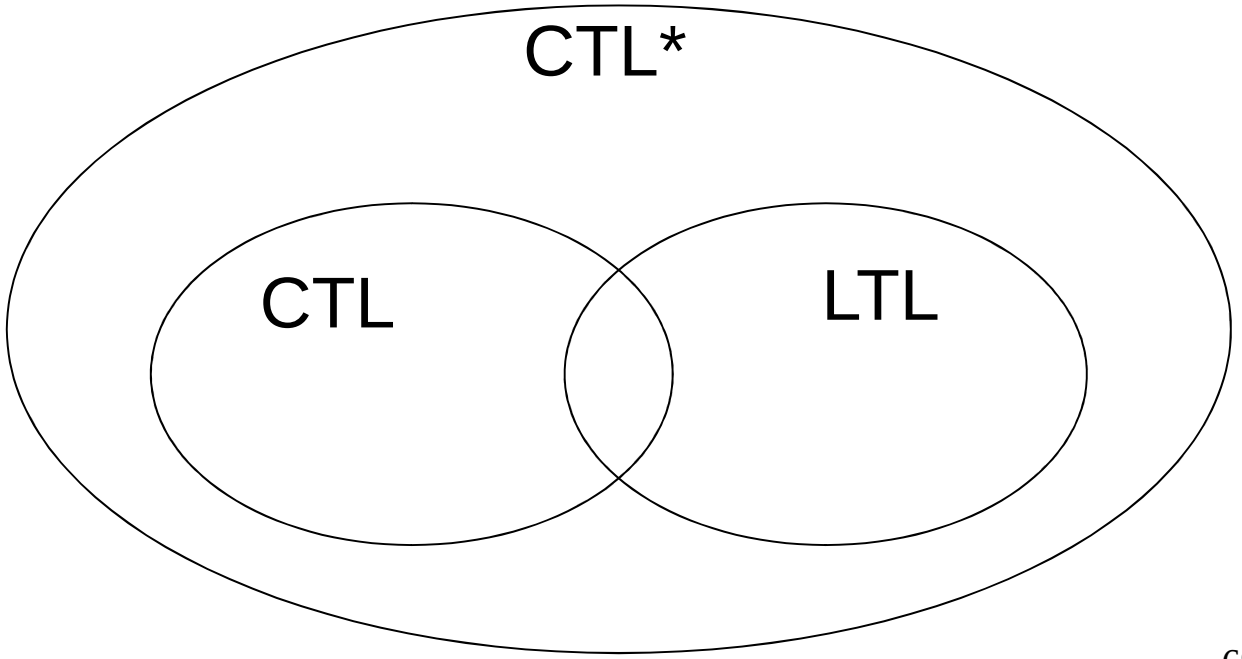
A ϕ ” s iff for each path π starting at s holds: ϕ ” π

Efficient algorithms

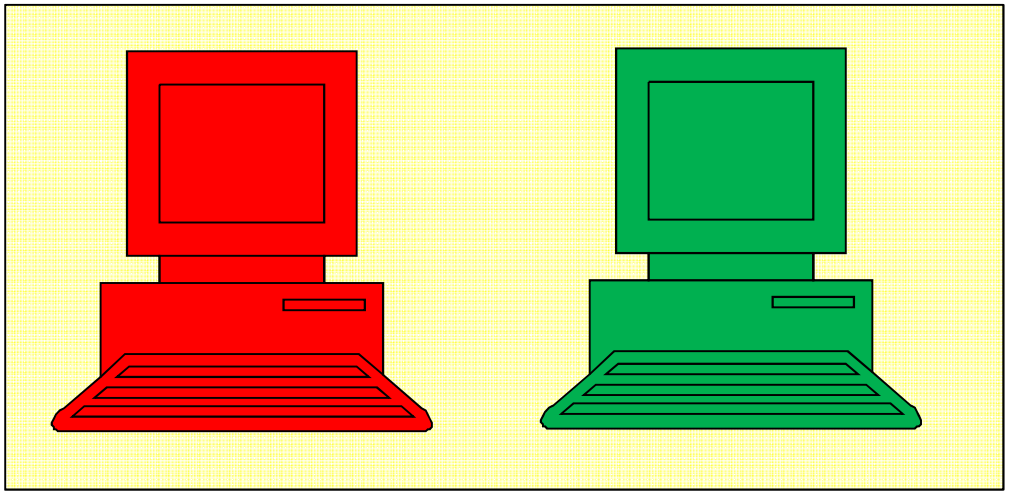
CTL* : $O(2^{|\phi|} |TS|)$

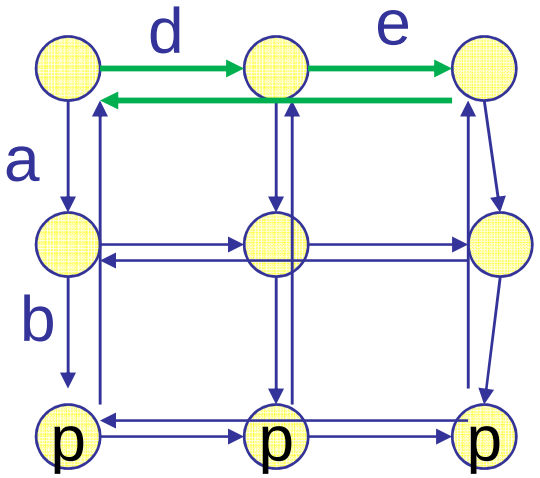
LTL: Only path formulas : $O(2^{|\phi|} |TS|)$

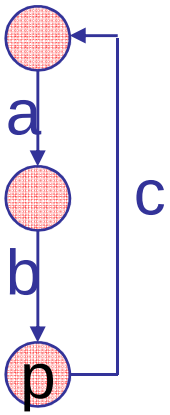
CTL: Only state formulas: $O(|\phi| |TS|)$

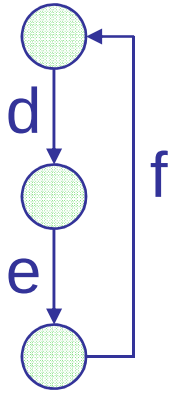


Fairness









GFp ✞ 

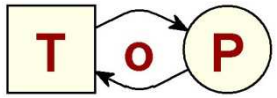
GFp ” 

Take ***GFp*** as part of the specification of 

W. Reisig

Synchronous Processes etc: their highlights in brief

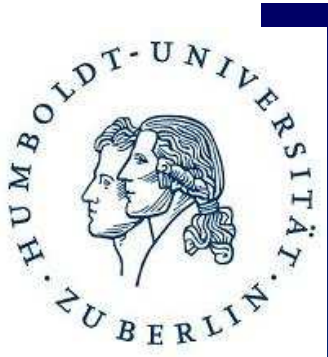
1. Their definition: CCS
2. Their canonical equivalence: Bisimulation
3. Their properties: Temporal Logic
4. **Their verification: Model Checking**



Theory of Programming

Prof. Dr. W. Reisig

The end



W. Reisig

Processes etc:

their highlights in brief

There are dozens of versions of

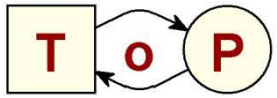
1. Their definition: CCS
2. Their canonical equivalence: Bisimulation
3. Their properties: CTL*
4. Their verification: Mod

process algebras

equivalences

temporal logics

tools



Theory of Programming

Prof. Dr. W. Reisig