Interval Temporal Logic Semantics of Box Algebra

Maciej Koutny

Newcastle University

with Hanna Klaudel (Evry) and Zhenhua Duan (Xian)

LATA'14, Madrid, March 2014

Motivation

- **Curiosity**

  People often said that ... *temporal logics and Petri nets are fundamentally different formalisms ...* were they right ?

- **Practicality**

  If we can establish a strong semantical connection then ... *analytical techniques could cross-fertilise both fields* ... or at least we can use both kinds of techniques for dealing with a given design

Talk outline

- **Box Algebra (BA)**

  Compositional Petri net model supporting sequence, choice, iteration, parallelism, action synchronisation

- **Interval Temporal Logic (ITL)**

  Temporal logic supporting sequence, choice, iteration, parallelism, variable synchronisation

- **Example of syntax-directed translation from BA to ITL**

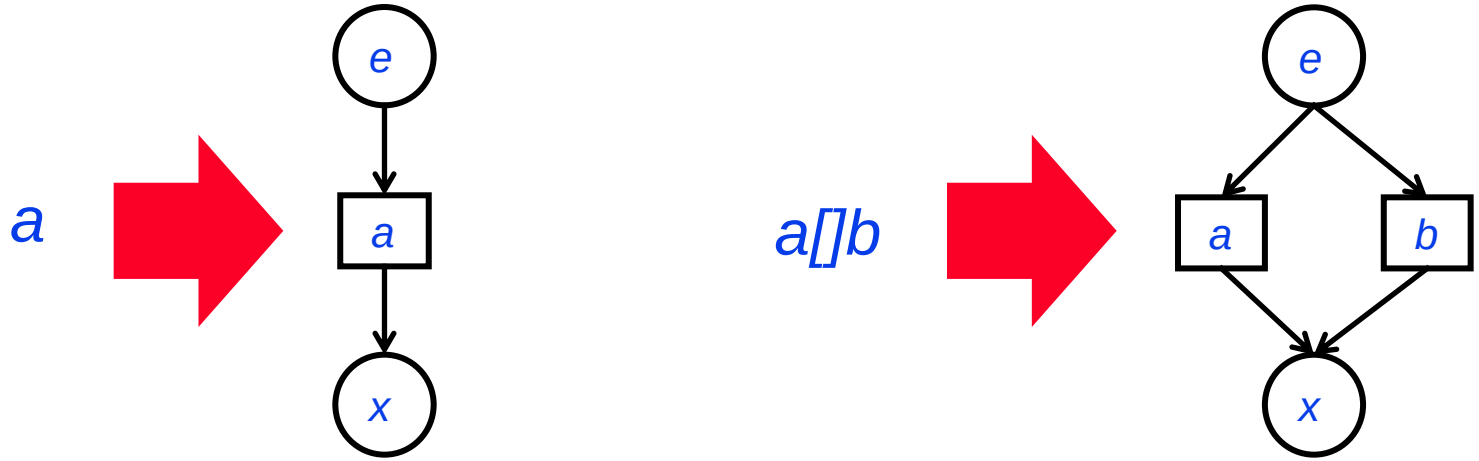
  Behavioural semantics is preserved

Box Algebra

- Compositional Petri net model based on box expressions:

  E := stop | a | E;E | E[]E | [E*E*E]
  F := E sco ρ

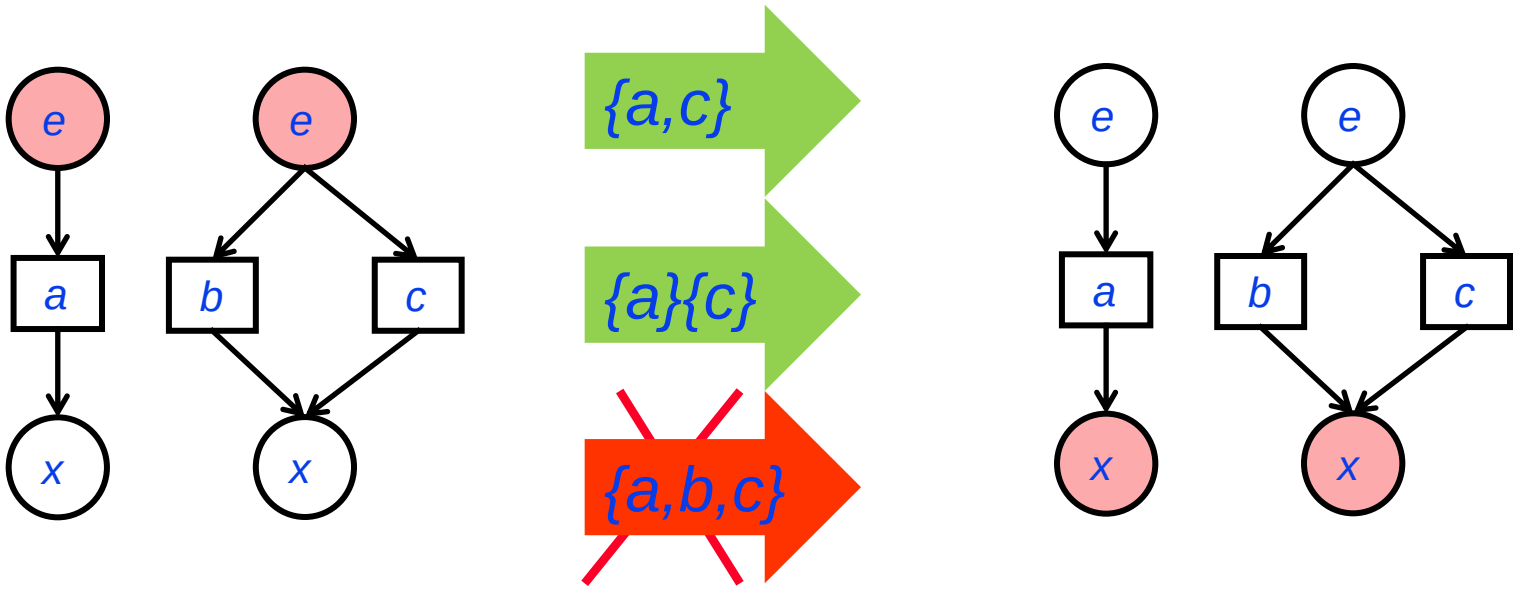
- Syntax-directed mapping generates Petri nets, called boxes, with explicit entry (e) and exit (x) interfaces:



Box Algebra Semantics

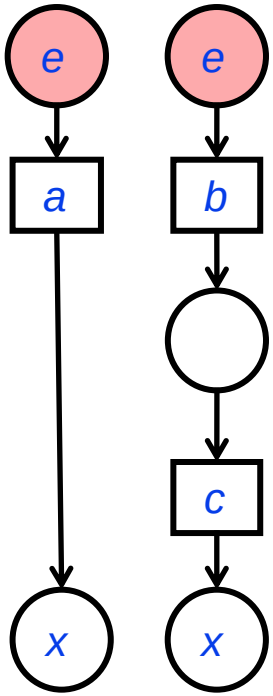
- Mark each e-place
- When a step (set) of transitions is enabled: input places are disjoint and marked
- Execution: unmark all input and mark all output places
- Repeated executions are step sequences

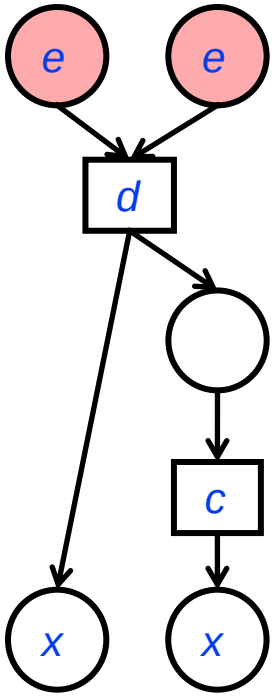
For the box of *a||(b[]c)* we have



Box Algebra Semantics

- Synchronisation glues together transitions
- Synchronised transition behaves as the step made up of the glued transitions



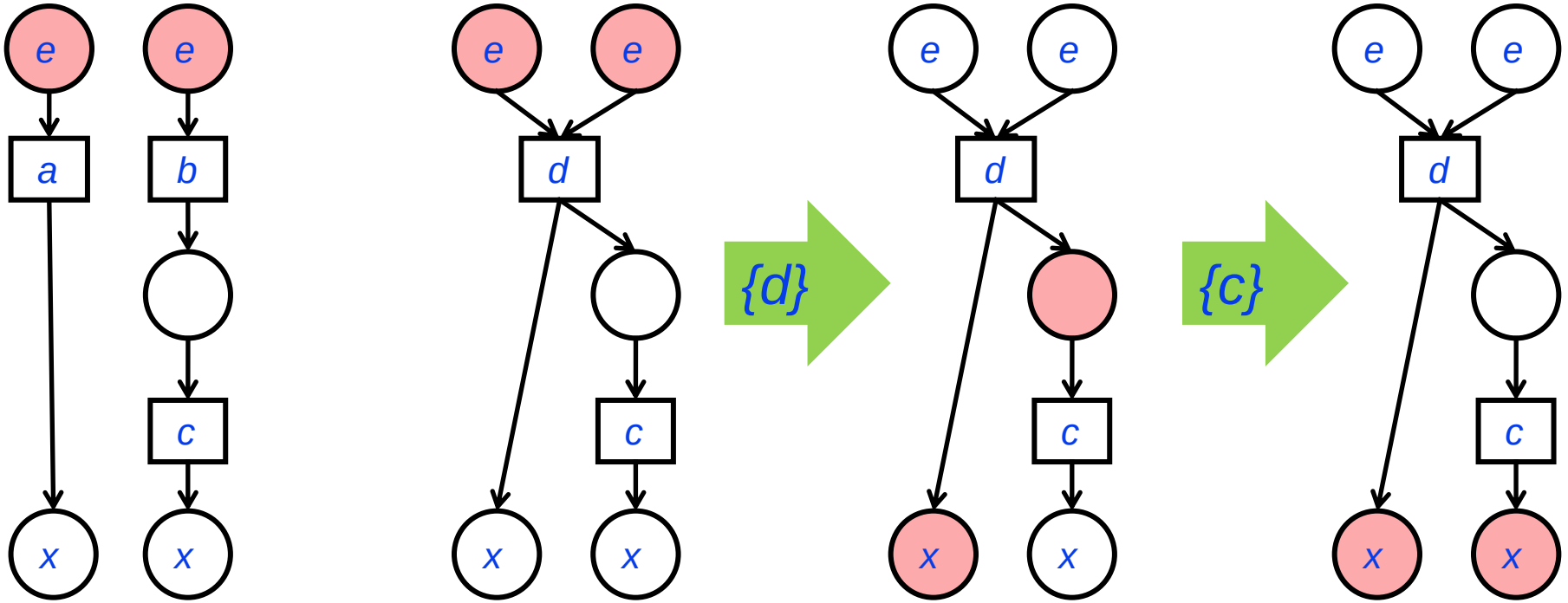


a || *(b;c)*

a || *(b;c)* sco { *a,b* → *d* … }

Box Algebra Semantics

- Synchronisation glues together transitions
- Synchronised transition behaves as the step made up of the glued transitions



a || *(b;c)*

a || *(b;c)* sco { *a,b* → *d* … }

Interval Temporal Logic

- Logic interpreted over discrete intervals of states (valuations of boolean variables):

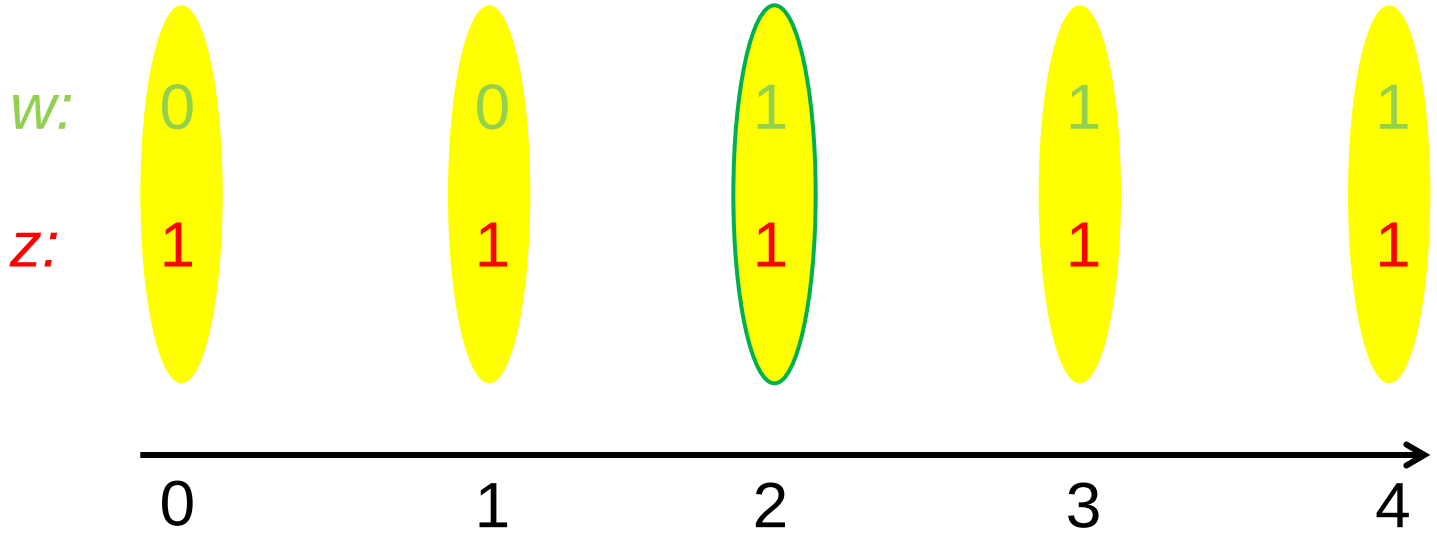
$$\varphi := \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi; \varphi \mid \varphi^* \mid \text{skipstable}(v) \mid \text{flip}(v)$$

- formulas specify how variables behave over intervals
- skipstable(*v*) keeps *v* unchanged between two consecutive states
- flip(*v*) changes *v* between two consecutive states

ITL Semantics

Two parallel processes:

"keep z unchanged" || *"change w at some point"*

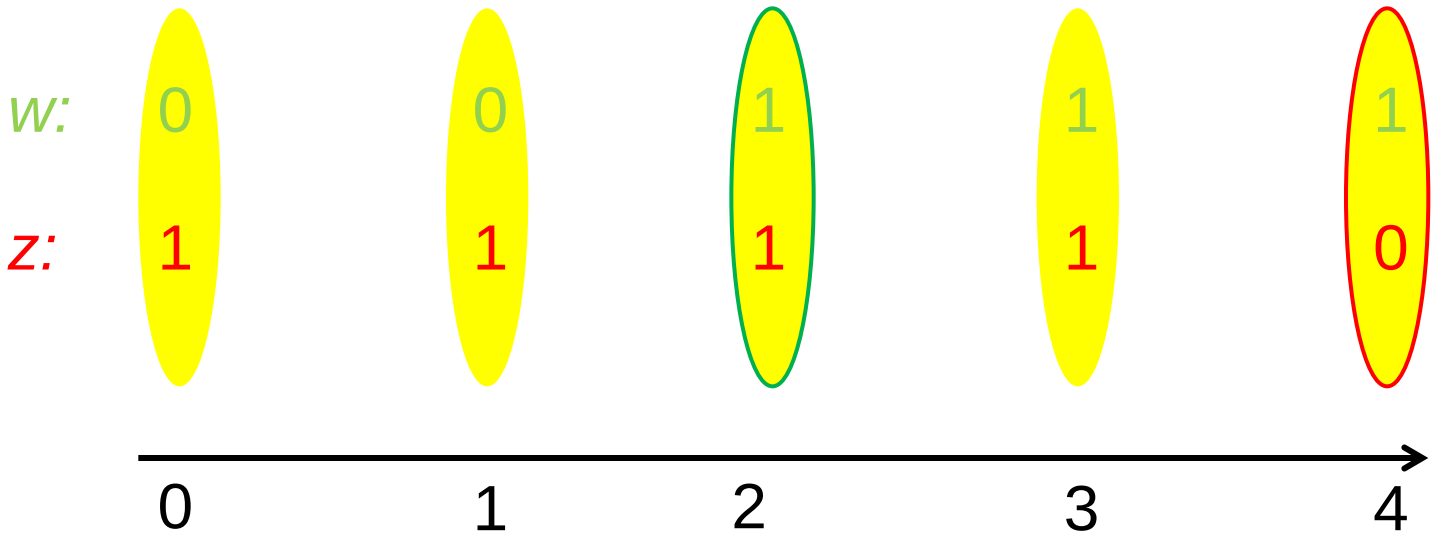


skipstable(z)* ∧ (skipstable(w)*; flip(w); skipstable(w)*)

ITL Semantics

Two parallel processes:

"change z at some point" || *"change w at some point"*



(skipstable(z)*; flip(z); skipstable(z)*) ∧
(skipstable(w)*; flip(w); skipstable(w)*)

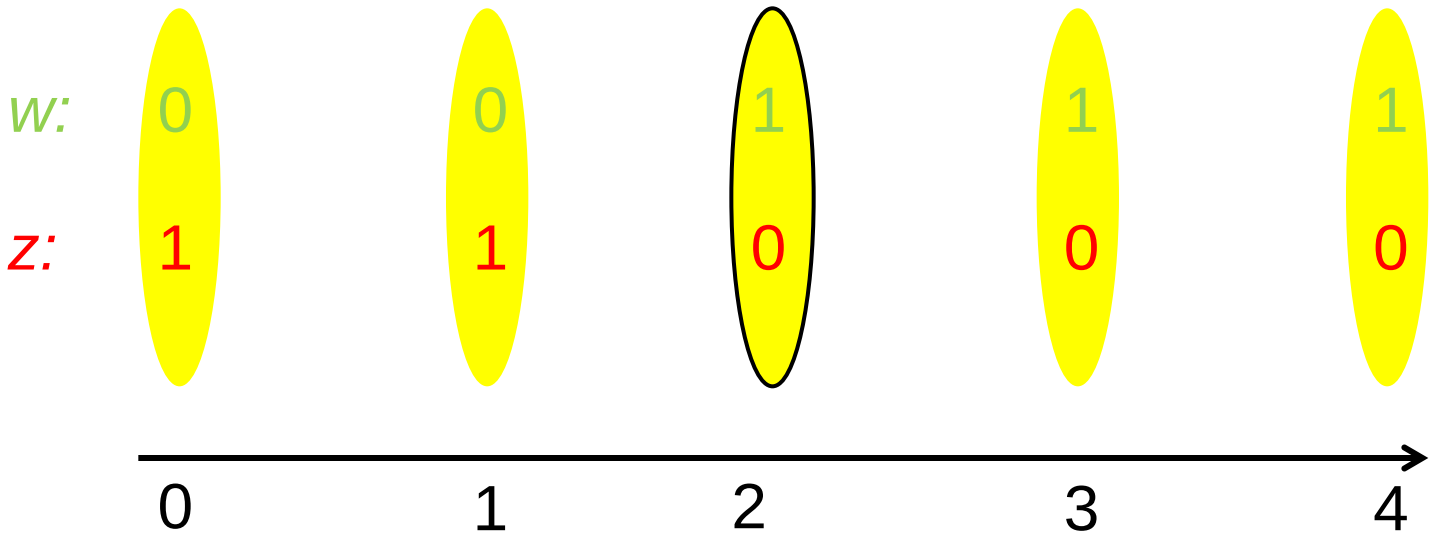
How can we *synchronise*

parallel processes?

ITL Semantics

Two synchronised parallel processes:

"change z at some point" || *"change w **AND** z at some point"*



(skipstable(z)*; flip(z); skipstable(z)*) ∧

(skipstable(w)*; flip(w) ∧ flip(z); skipstable(w)*)

Box Algebra vs ITL

- Syntactically similar: sequence, choice, parallelism
- However semantical models

  step sequences
  intervals

  are very different

Box Algebra vs ITL

- Syntactically similar: sequence, choice, parallelism
- However semantical models

  step sequences
  intervals

  are very different
- BUT are they really so different ?

  in circuit design, events are changes of voltage level

  in intervals, variables change in step-like manner

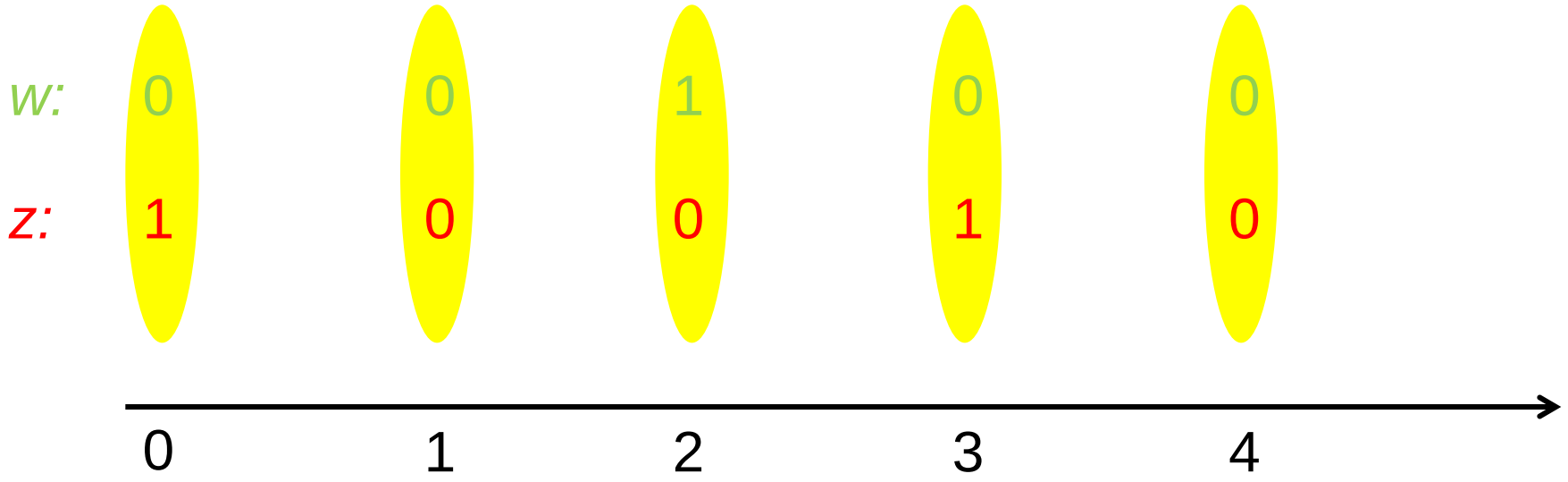
Box Algebra vs ITL

Basic idea of the proposed solution:

- associate a variable with each transition
- step is executed = corresponding variable are flipped

Equivalent executions

ITL: *w* and *z* are variables



{ *z* } { *w* } { *z*, *w* } { *z* }

BA: *w* and *z* are transitions

Client/Server example

Client

```
...
(send.req; receive.ans); local.upd
...
```

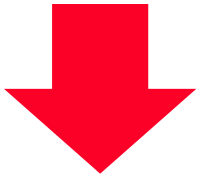
Server

```
...
if
case 1: receive.req; send.ans
case 2: local.upd
fi
...
```

Client/Server example

Client

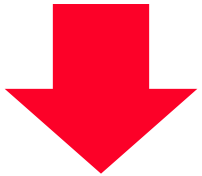
```
...
(send.req; receive.ans); local.upd
...
```



Client = (s.req; r.ans); l.upd

Server

```
...
if
case 1: receive.req; send.ans
case 2: local.upd
fi
...
```

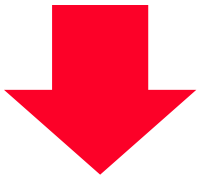


Server = (r.req; r.ans) [] l.upd

Client/Server example

Client

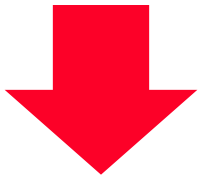
```
...
(send.req; receive.ans); local.upd
...
```



Client = (s.req; r.ans); l.upd

Server

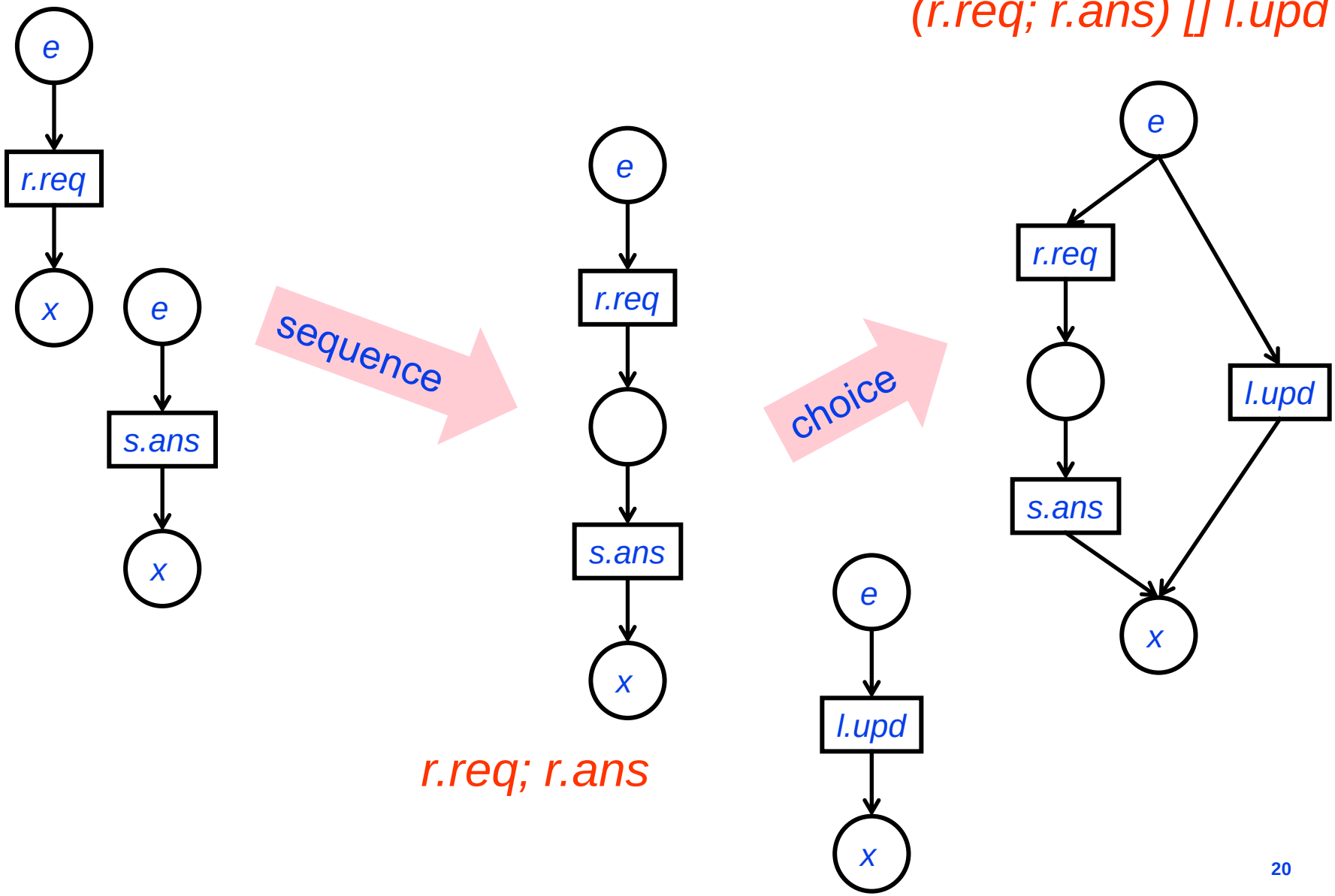
```
...
if
case 1: receive.req; send.ans
case 2: local.upd
fi
...
```



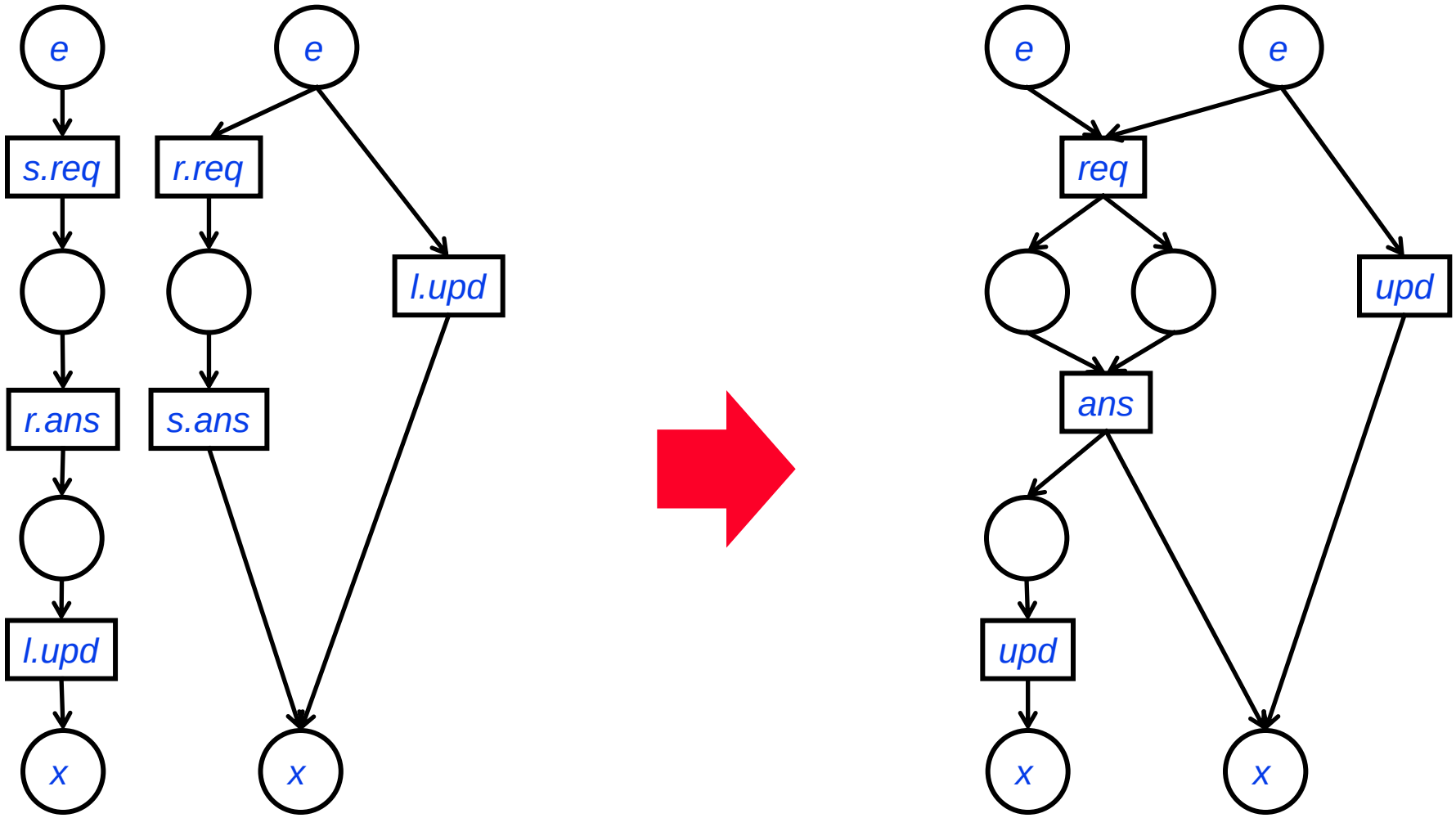
Server = (r.req; r.ans) [] l.upd

System
=
Client || Server sco { s.req, r.req → req
s.ans , r.ans → ans
l.upd → upd }

Server box


(r.req; r.ans) [] l.upd
e
r.req
x
e
s.ans
x
sequence
e
r.req
s.ans
x
r.req; r.ans
choice
e
l.upd
x
e
r.req
l.upd
s.ans
x

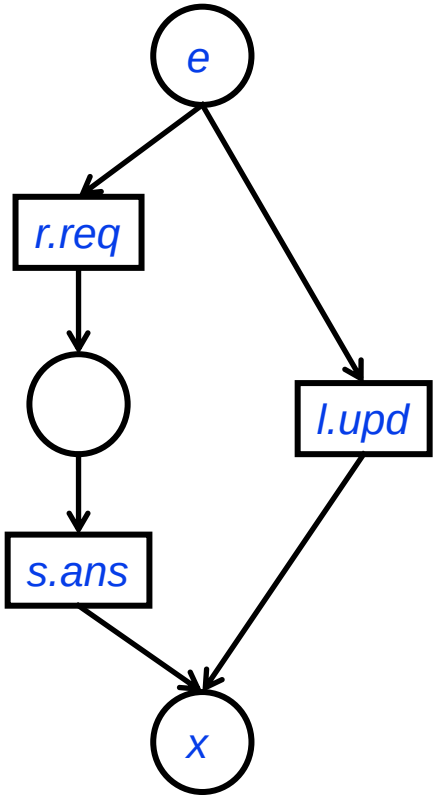
System box



Client || Server

System

Server formula



fks(*r.req* | *s.ans,l.upd*) ; ***fks***(*s.ans* | *r.req,l.upd*)

V

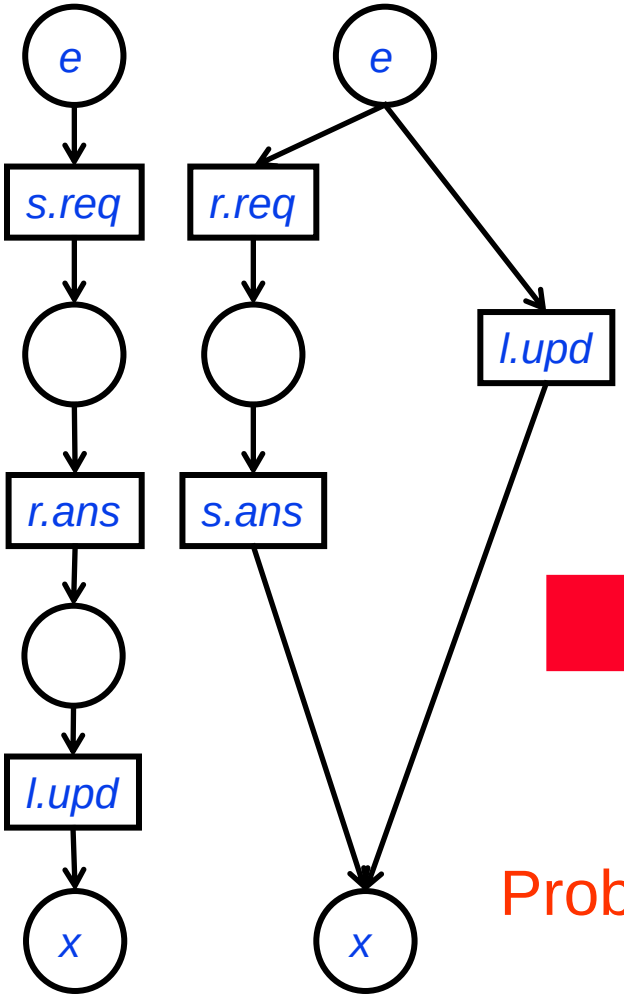
fks(*l.upd* | *r.req,s.ans*)

itl(Server)

fks(*l.upd* | *r.req, s.ans*) means:

flip l.upd & keep stable r.req and s.ans

Client||Server formula



itl(*Client || Server*)

fks(*s.req* | ..) ; ***fks***(*r.ans* | ..) ; ***fks***(*l.upd* | ..)

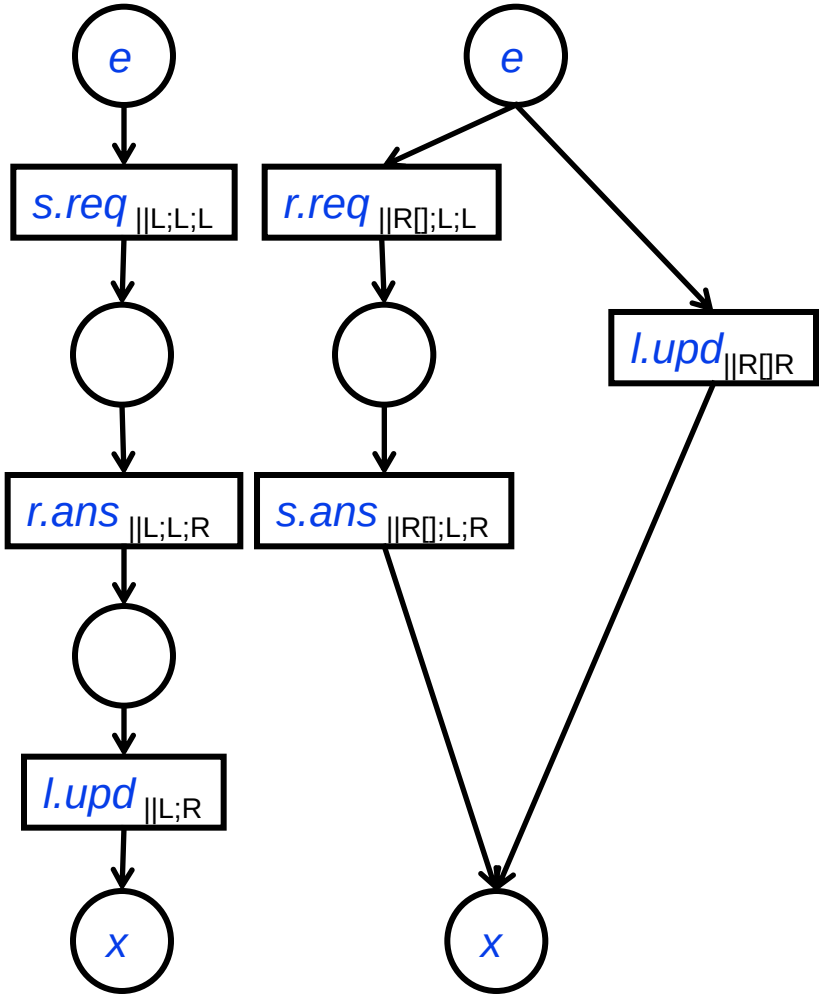
∧

(***fks***(*r.req* | ..) ; ***fks***(*s.ans* | ..)) ∨ ***fks***(*l.upd* | ..)

Problem:

flipping of *l.upd* must happen in both sub-formulas creating false synchronisation !!!

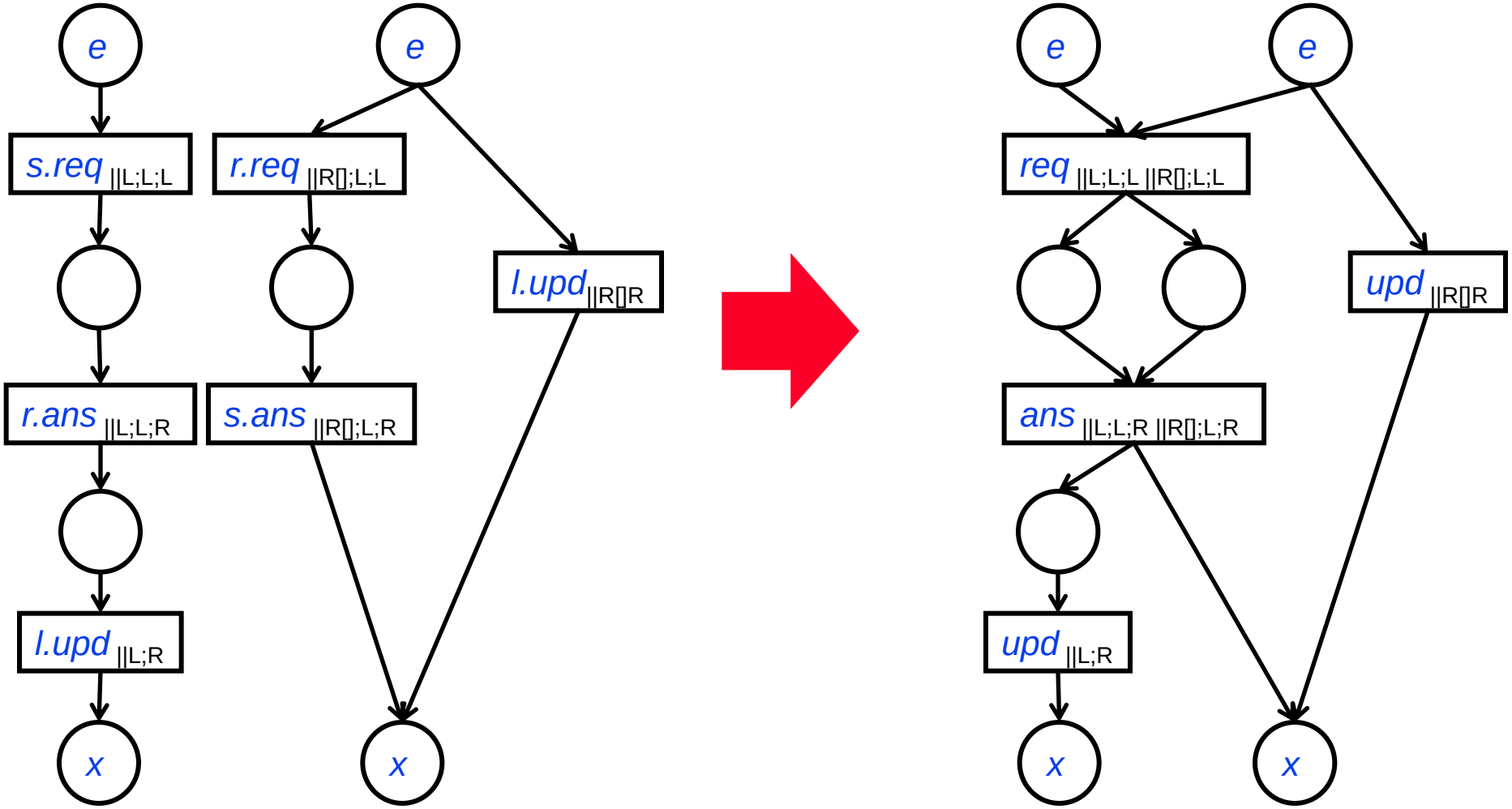
Client||Server formula



Solution:

use syntax paths in addition to labels

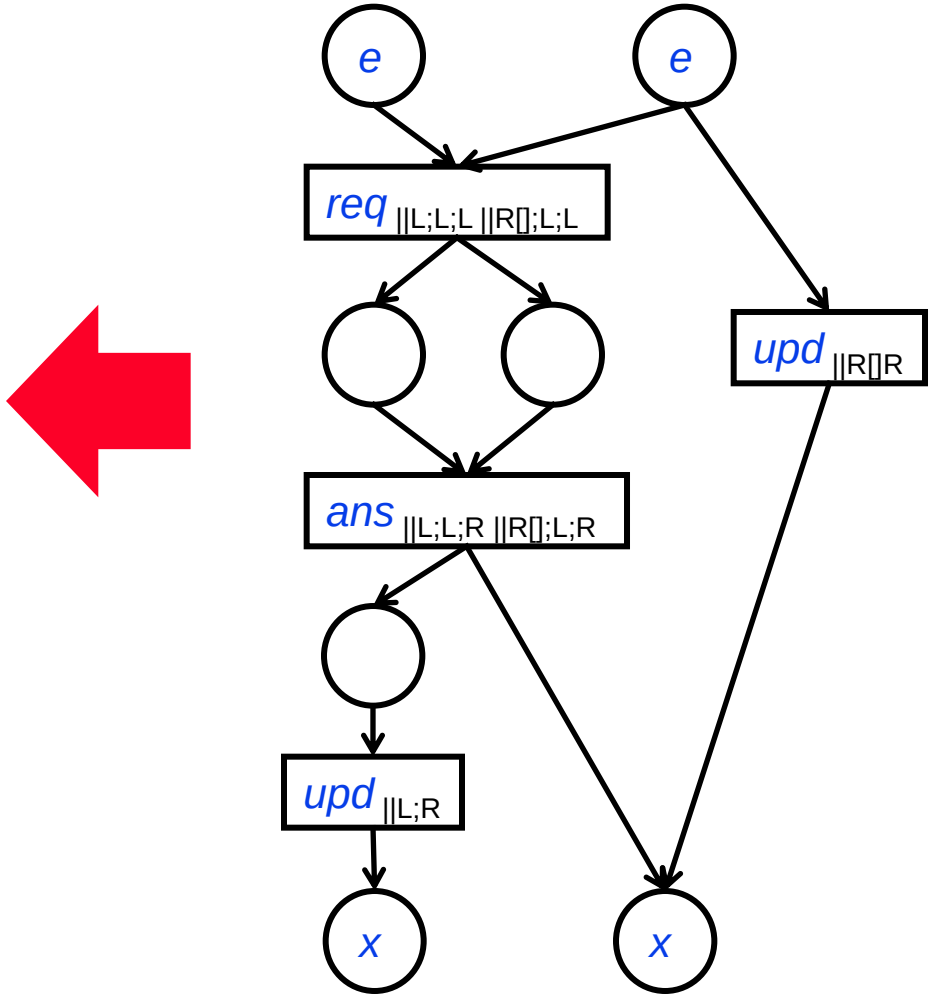
Client||Server formula



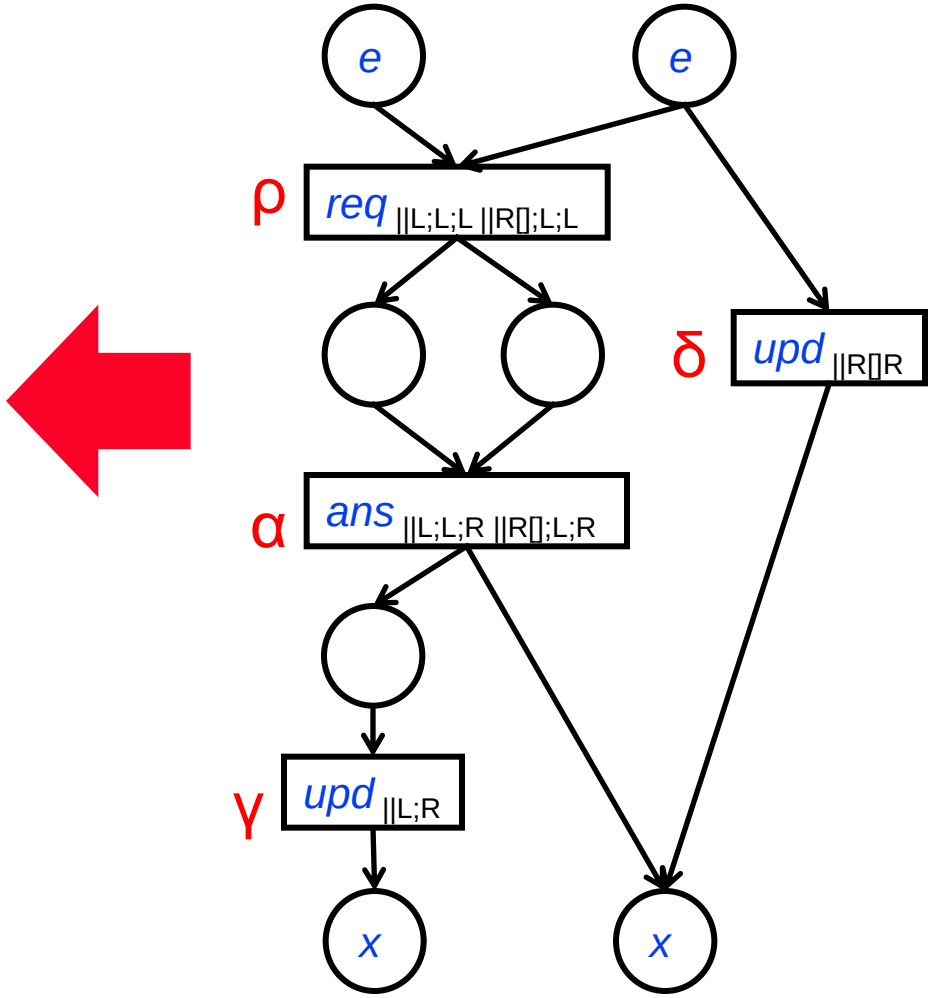
Solution:

use syntax paths in addition to labels

System formula



System formula



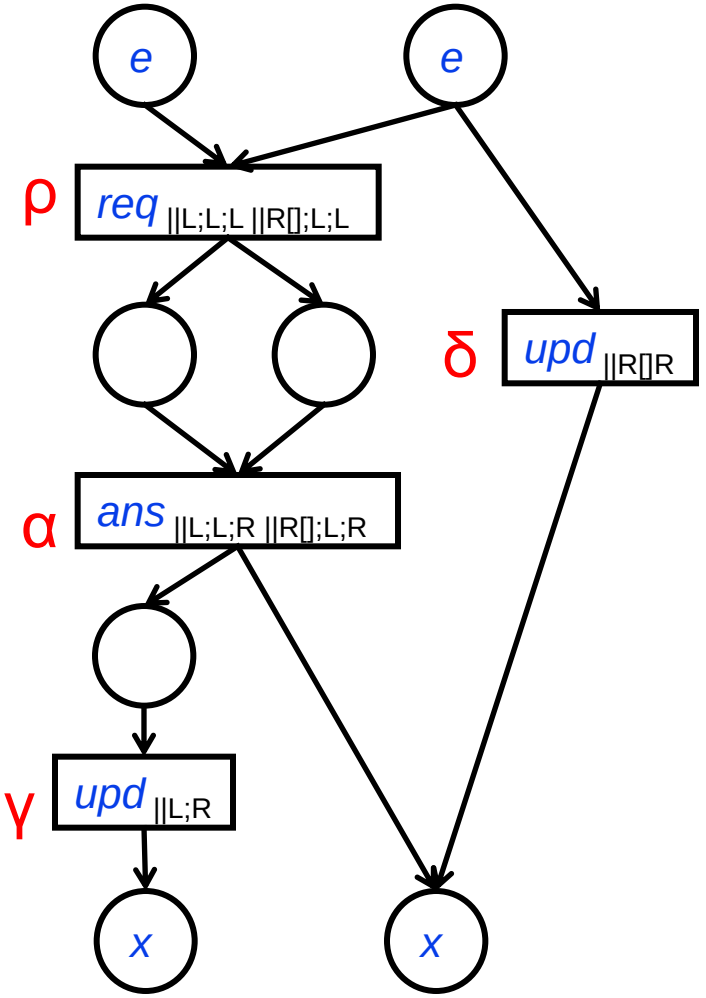
System formula

fs(ρ | ..) ; ***fs***(α | ..) ; ***fs***(γ | ..)

∧

(***fs***(ρ | ..) ; ***fs***(α | ..)) ∨ ***fs***(δ | ..)

itl(System)



Main result

THEOREM

box(E) and *itl(E)* generate equivalent behaviours

Concluding remarks

- We obtained a translation of compositional Petri nets into behaviourally equivalent ITL formulas
- We plan to investigate how other operators of ITL map into net compositions

Thank you!