Process Oriented Data Science

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Campus d'Excel·lència Internacional

Josep Carmona
Computer Science Department

Outline

- M1: Process Mining Overview, Positioning & Preliminaries (Event data & Process Models)
- **M2: Process Discovery**
- M3: Conformance Checking
- M4: Process Enhancement

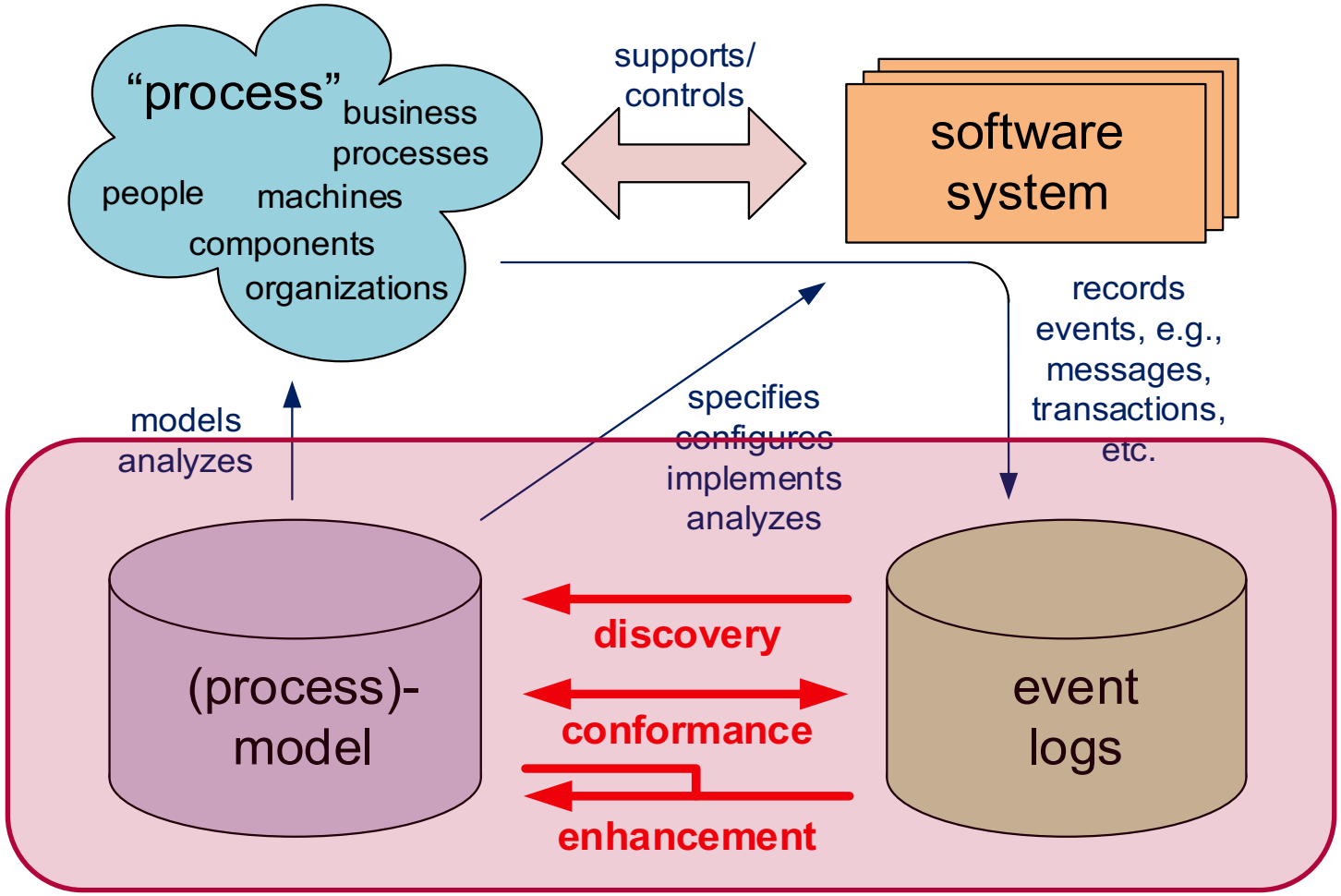
Disclaimer

- Most of the material of this course is taken from my colleagues:
  - **RWTH Aachen (Prof. Wil van der Aalst)**
  - **Humboldt University zu Berlin (Prof. Matthias Weidlich)**
  - Technishe Universiteit Eindhoven (Prof. Boudewijn van Dongen)
  - University of Tartu (Prof. Marlon Dumas)
  - University of Melbourne (Prof. Marcello La Rosa)
  - Technical University of Denmark (Prof. Andrea Burattin)
- Hence, this material is only provided for your learning, please do not share nor publish

Outline

- Introduction to quality metrics
- Process Model Discovery
- The Family of Alpha-Algorithms

The Context


"process"
business
processes
people
machines
components
organizations
supports/
controls
software
system
records
events, e.g.,
messages,
transactions,
etc.
models
analyzes
specifies
configures
implements
analyzes
(process)-
model
discovery
conformance
enhancement
event
logs

Process Model Discovery

- Discovery
  - Create process model for observed behaviour

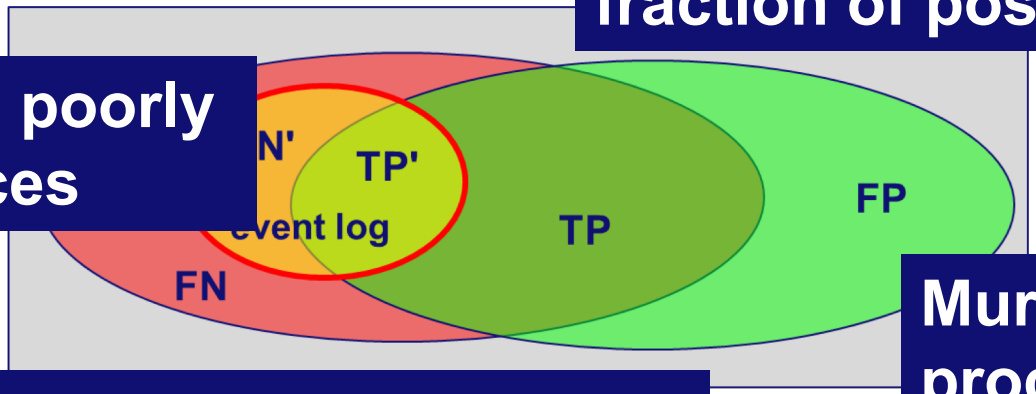
  - But: typically not every possible behaviour (i.e., trace) may have been executed and thus recorded
- Quality dimensions
  - Fitness: model should allow for behaviour seen in log
  - Precision – Generalisation trade-off: model does not allow for behaviour completely unrelated to log, but generalizes to some extent what is seen in the log
  - Simplicity: discovered model should be as simple as possible

Challenges

No negative examples
(cannot see what cannot happen)

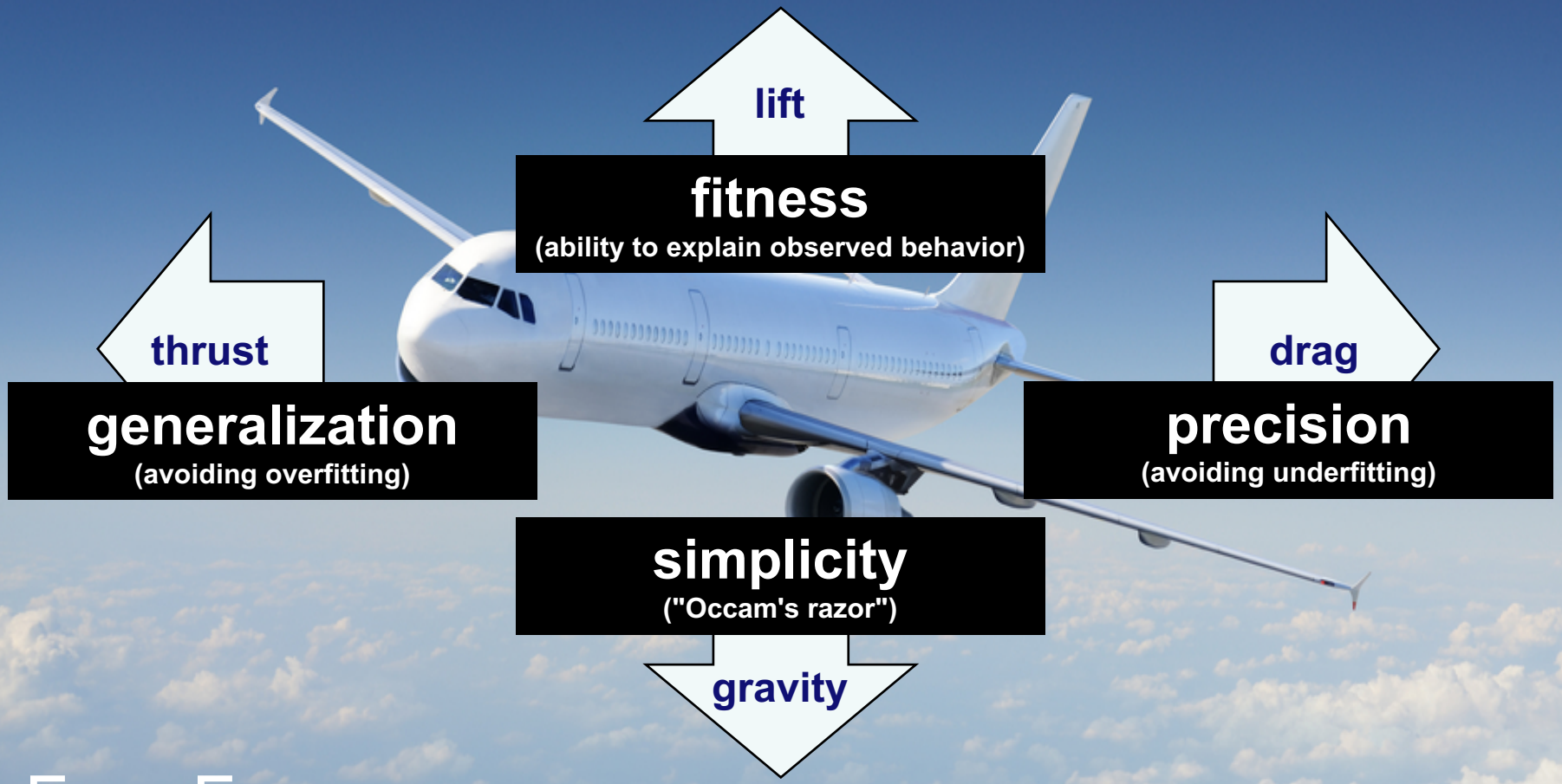
Log contains only a fraction of possible traces

Almost vs poorly fitting traces



In case of loops often infinitely many possible traces

Murphy's law for process mining
(anything is possible, so probabilities matter)


lift
fitness
(ability to explain observed behavior)
thrust
drag
generalization
(avoiding overfitting)
precision
(avoiding underfitting)
simplicity
("Occam's razor")
gravity

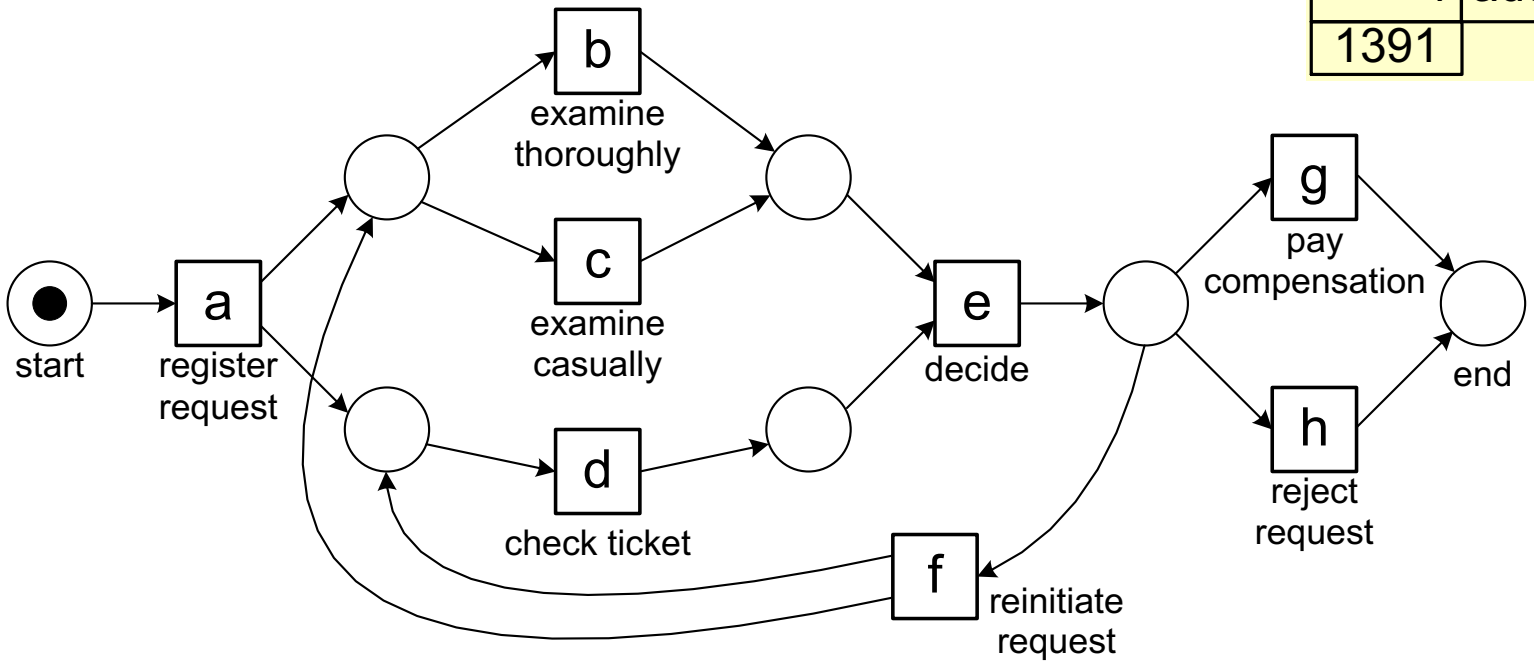
Four Forces

Example Log

#	trace
455	acdeh
191	abdeg
177	adceh
144	abdeh
111	acdeg

Model that seems to be OK …

#	trace
455	acdeh
191	abdeg
...	...
1	adcefdbefcdefdbeg
1391	

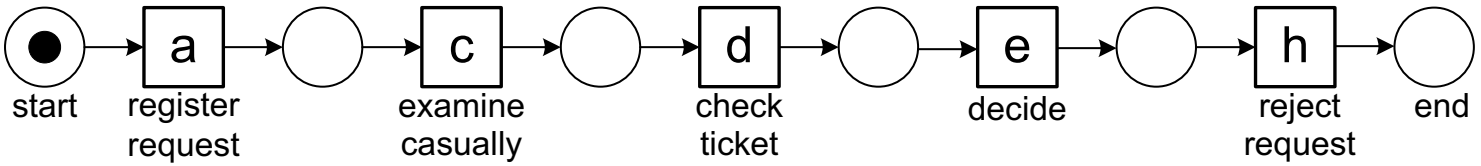


56	adbeh
47	acdefdbeh
38	adbeg
33	acdefbdeh
14	acdefbdeg
11	acdefdbeg
9	adcefcdeh
8	adcefdbeh
5	adcefbdeg
3	acdefbdefdbeg
2	adcefdbeg
2	adcefbdefbdeg
1	adcefdbefbdeh
1	adbefbdefdbeg
1	adcefdbefcdefdbeg
391	

fitness	simplicity	precision	generalization
(observed behavior fits)	("Occam's razor")	(avoiding underfitting)	(avoiding overfitting)

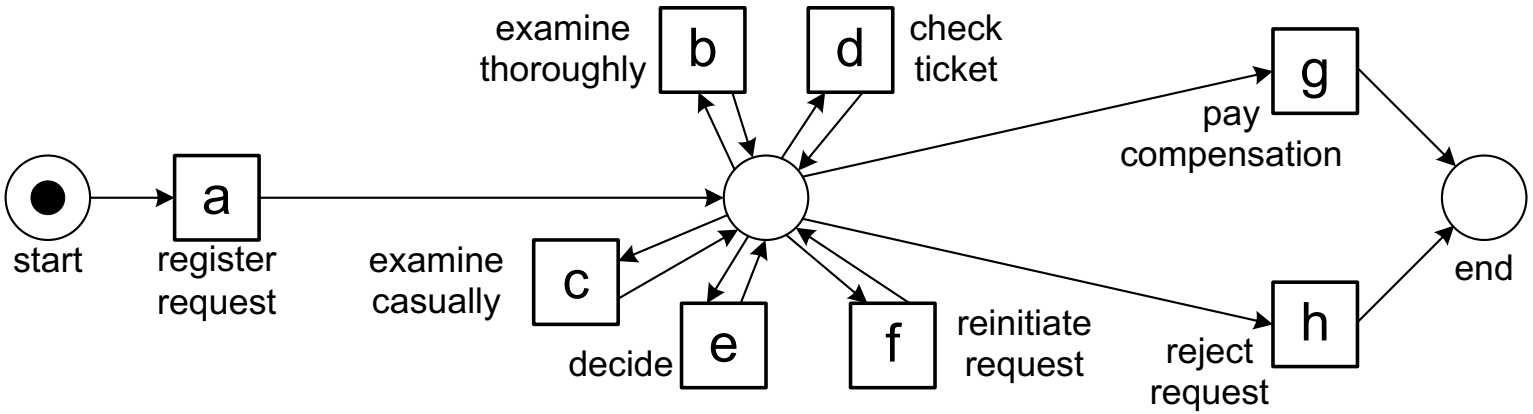
Non-fitting model

#	trace
455	acdeh
191	abdeg
...	...
1	adcefdbefcdefdbeg
1391	



56	adbeh
47	acdefdbeh
38	adbeg
33	acdefbdeh
14	acdefbdeg
11	acdefdbeg
9	adcefcdeh
8	adcefdbeh
5	adcefbdeg
3	acdefbdefdbeg
2	adcefdbeg
2	adcefbdefbdeg
1	adcefdbefbdeh
1	adbefbdefdbeg
1	adcefdbefcdefdbeg
391	

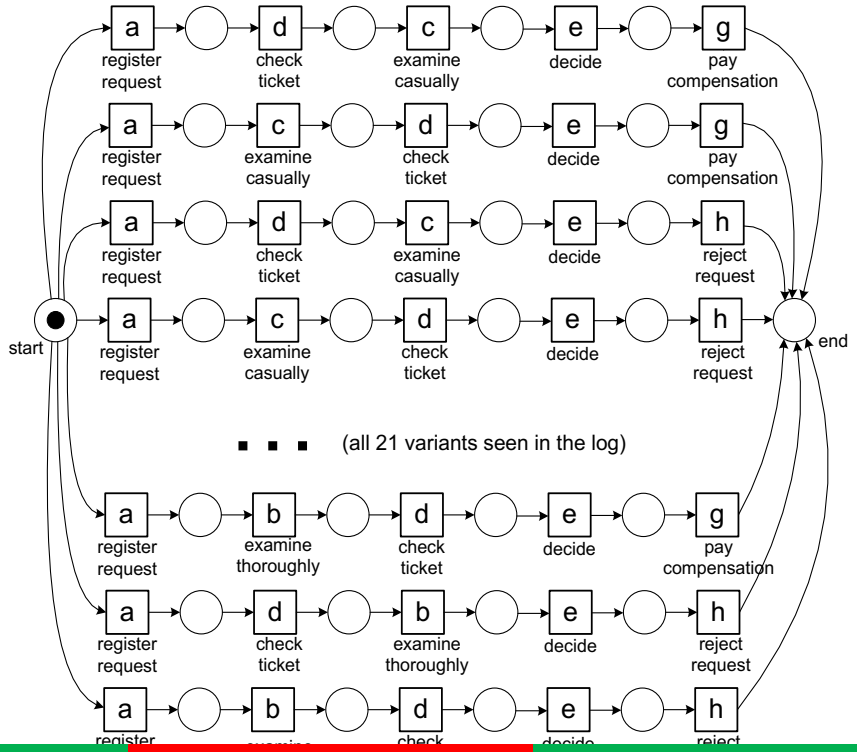
Underfitting model



#	trace
455	acdeh
191	abdeg
177	adceh
144	abdeh
111	acdeg
82	adceg
56	adbeh
47	acdefdbeh
38	adbeg
33	acdefbdeh
14	acdefbdeg
11	acdefdbeg
9	adcefcdeh
8	adcefdbeh
5	adcefbdeg
3	acdefbdefdbeg
2	adcefdbeg
2	adcefbdefbdeg
1	adcefdbefbdeh
1	adbefbdefdbeg
1	adcefdbefcdefdbeg
391	

underfitting

Overfitting model



#	trace
455	acdeh
191	abdeg
177	adceh
144	abdeh
111	acdeg
82	adceg
56	adbeh
47	acdefdbeh
38	adbeg
33	acdefbdeh
14	acdefbdeg
11	acdefdbeg
9	adcefcdeh
8	adcefdbeh
5	adcefbdeg
3	acdefbdefdbeg
2	adcefdbeg
2	adcefbdefbdeg
1	adcefdbefbdeh
1	adbefbdefdbeg
1	adcefdbefcdefdbeg
391	

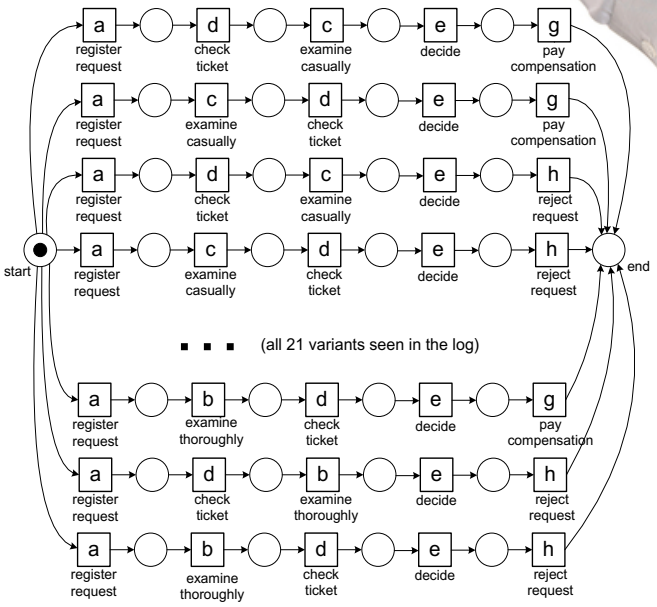
fitness
(observed behavior fits)

simplicity
("Occam's razor")

precision
(avoiding underfitting)

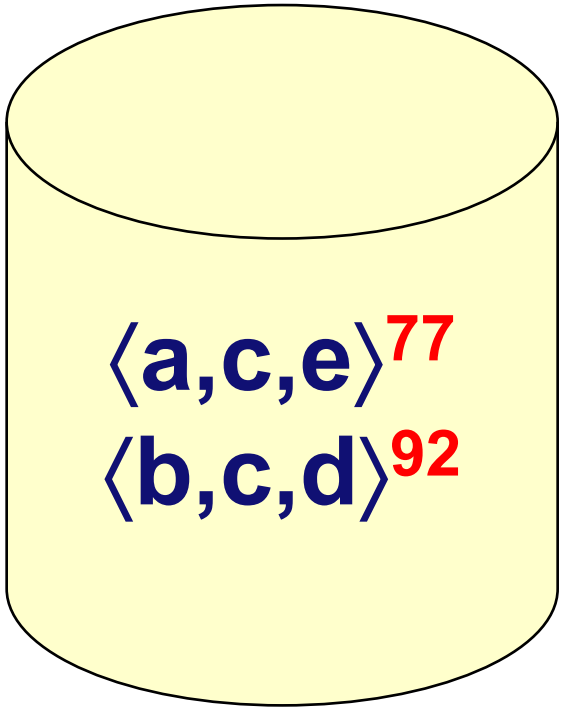
generalization
(avoiding overfitting)

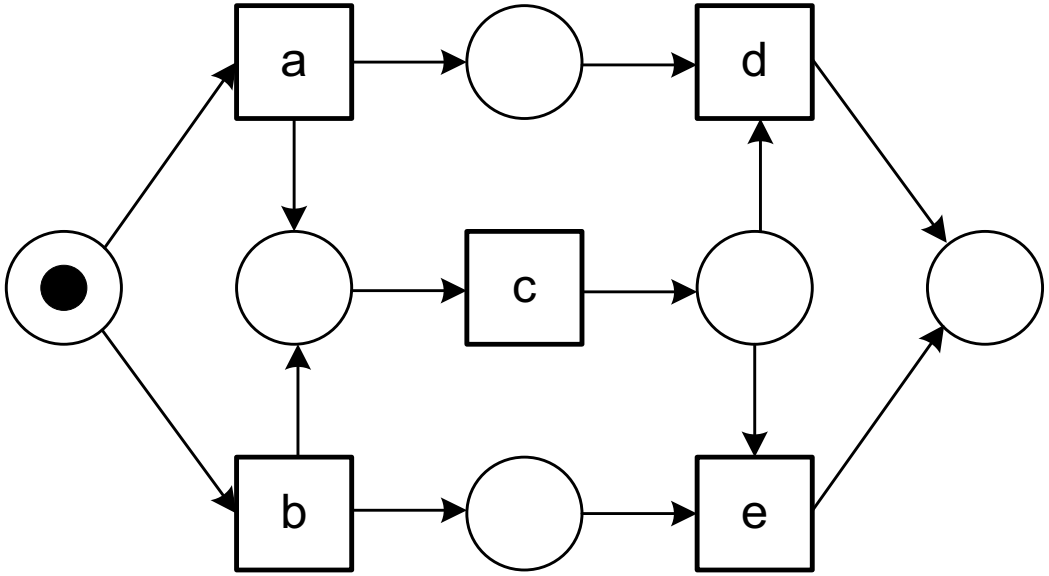
overfitting



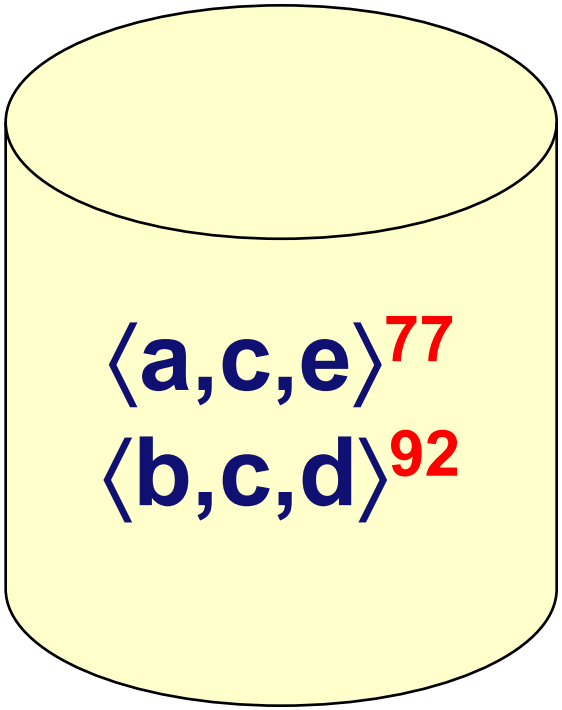
What is the best model ?

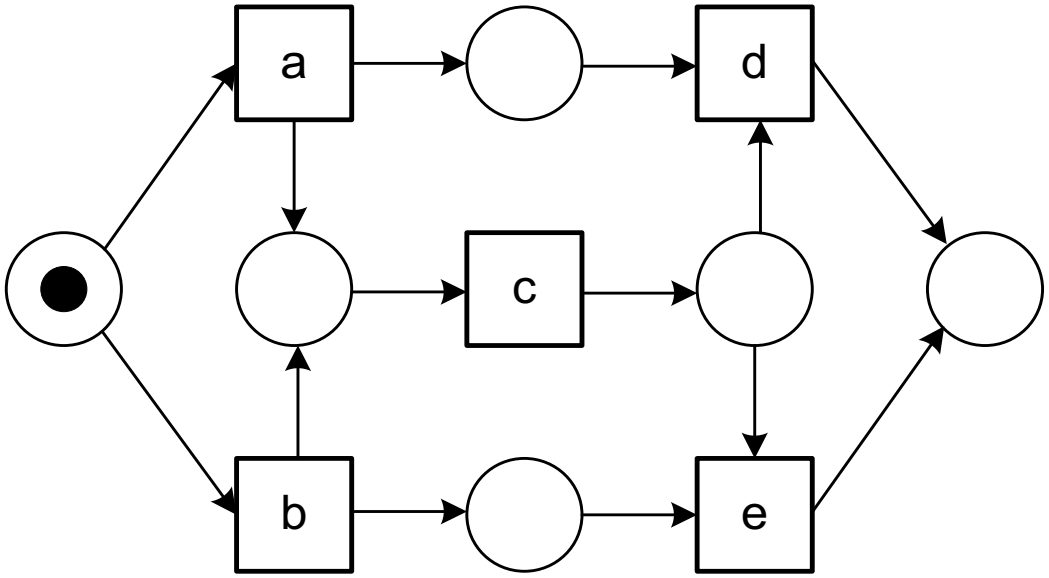
Fitness: good or bad?





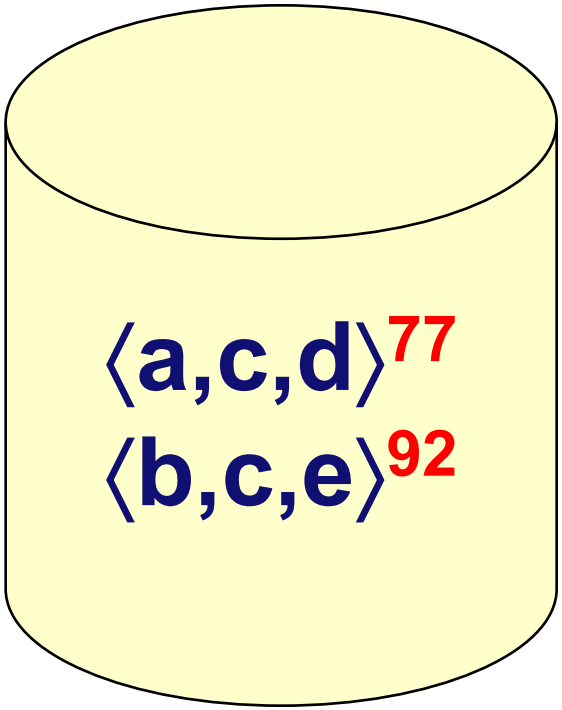
Fitness: bad!

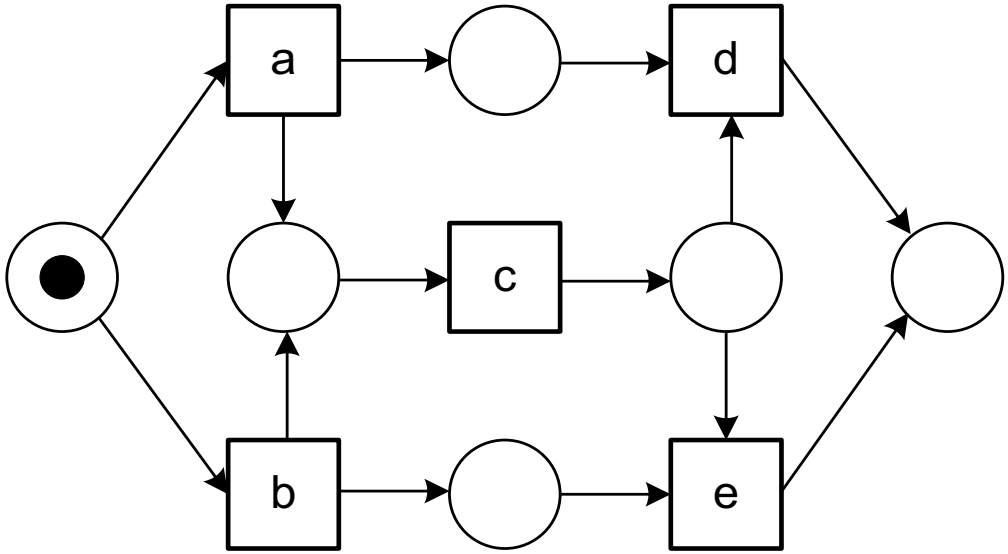




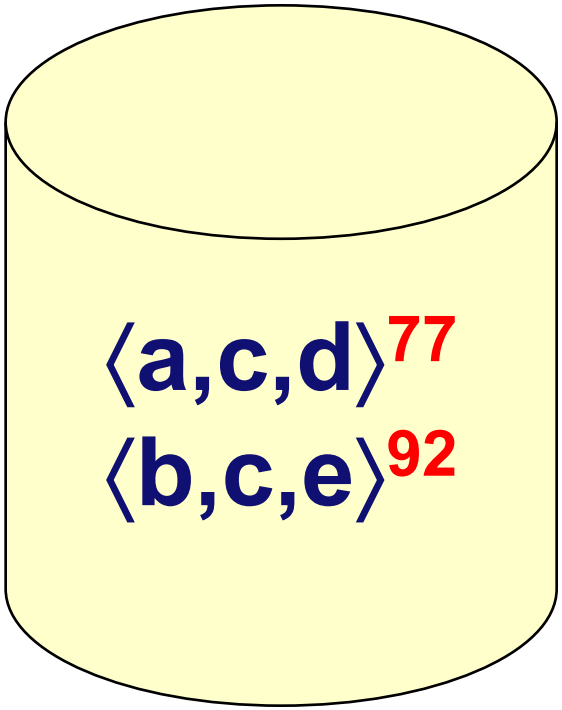
both traces do not fit …

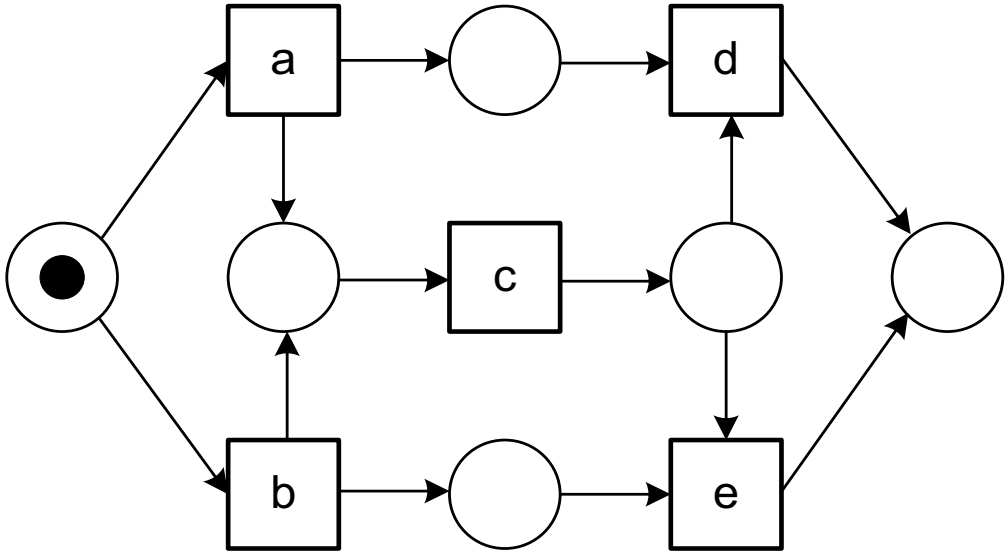
Precision: good or bad?





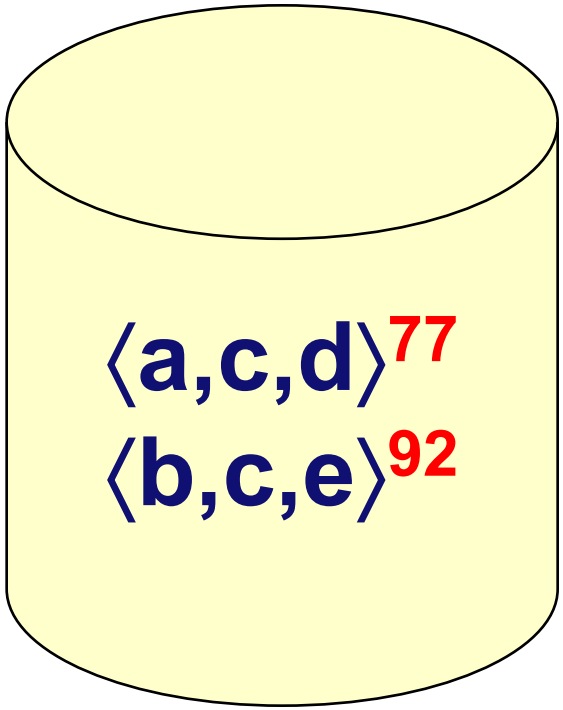
Precision: good!

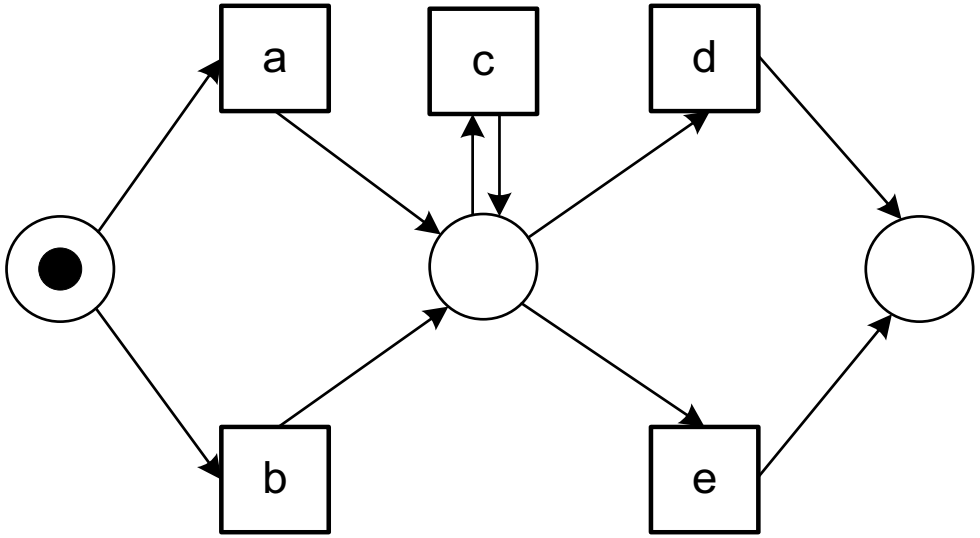




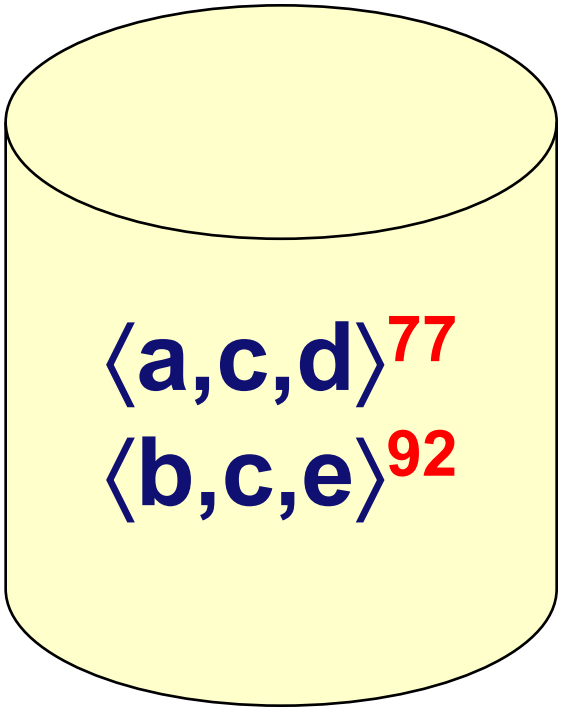
not underfitting…

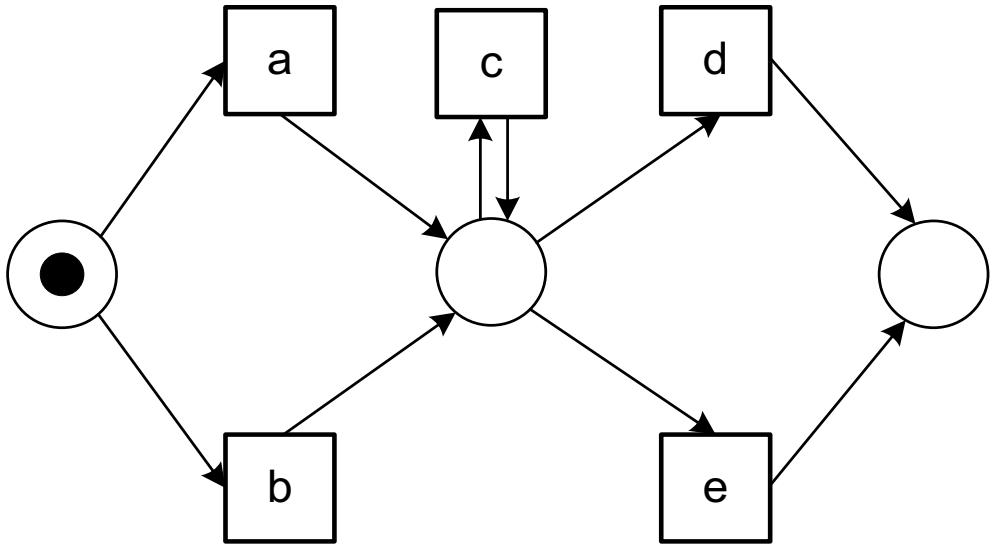
Precision: good or bad?





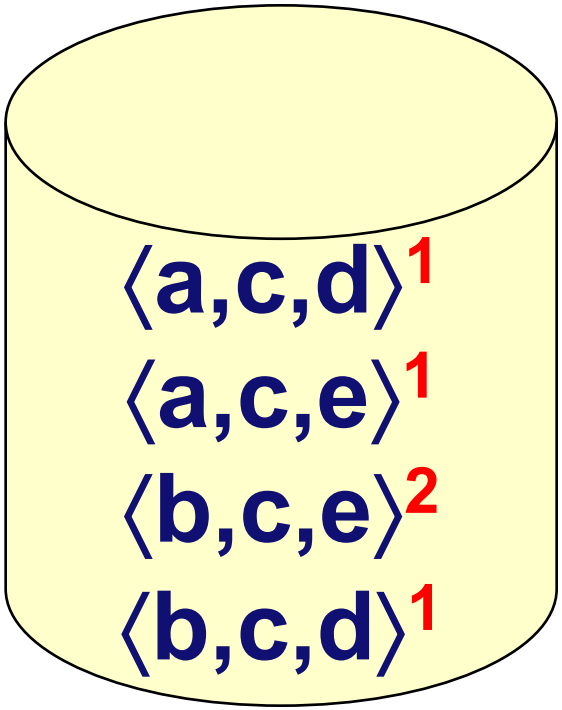
Precision: bad!

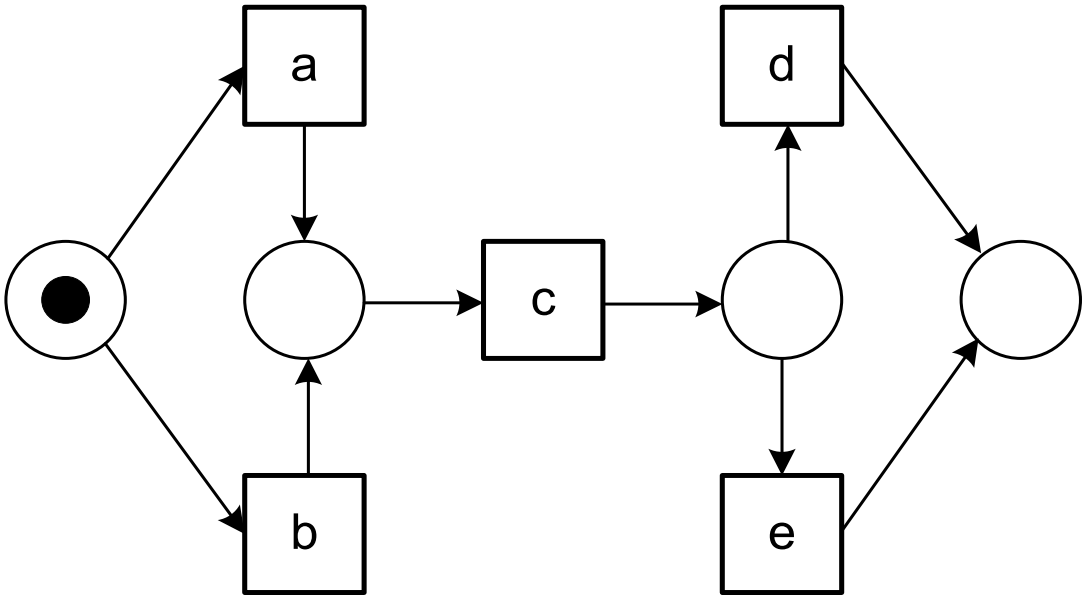




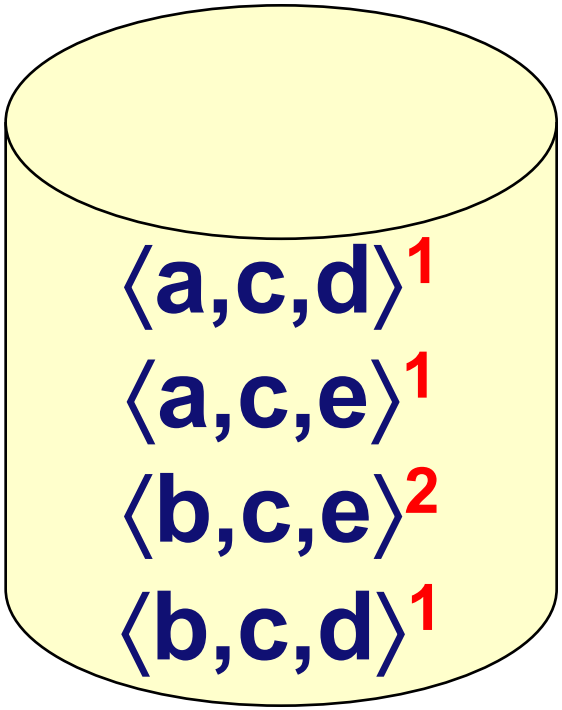
underfitting (allows for highly unlikely behavior) …

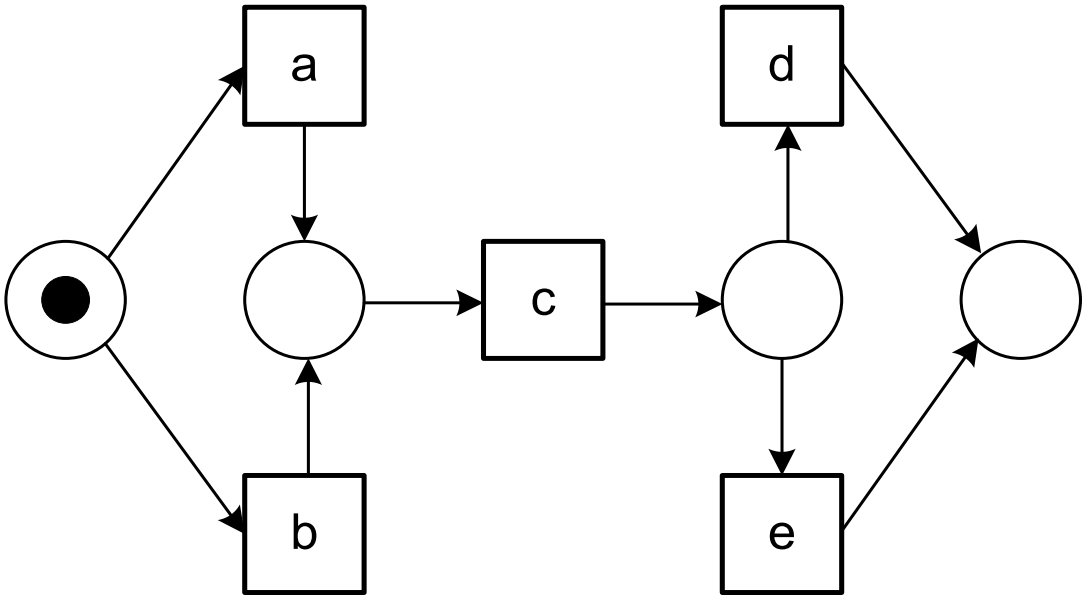
Generalization: good or bad?





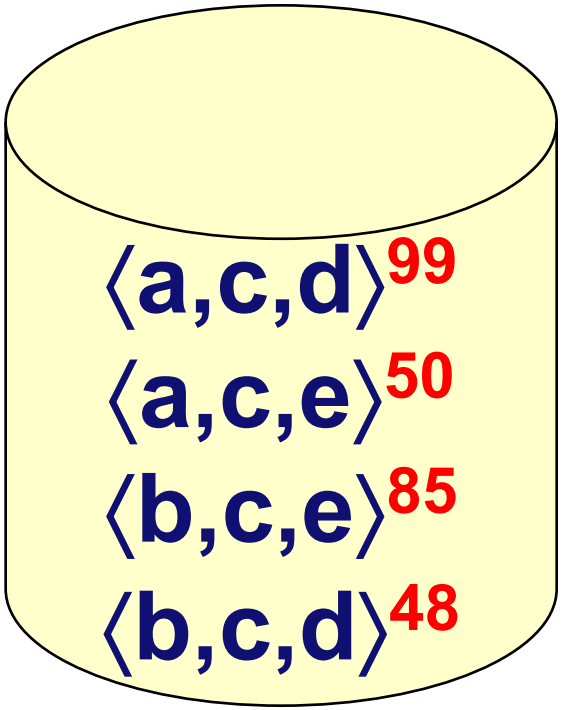
Generalization: bad!

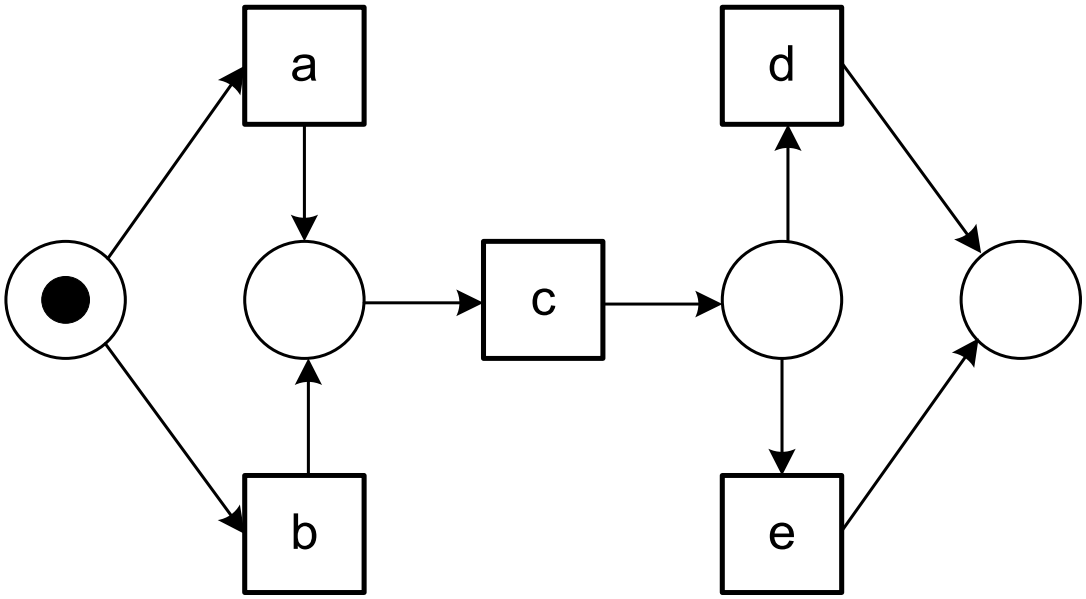




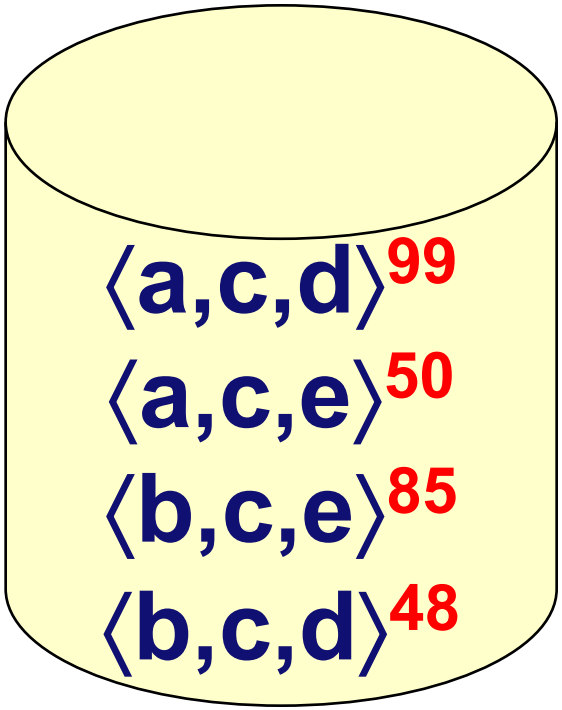
risk of overfitting on 5 example traces …

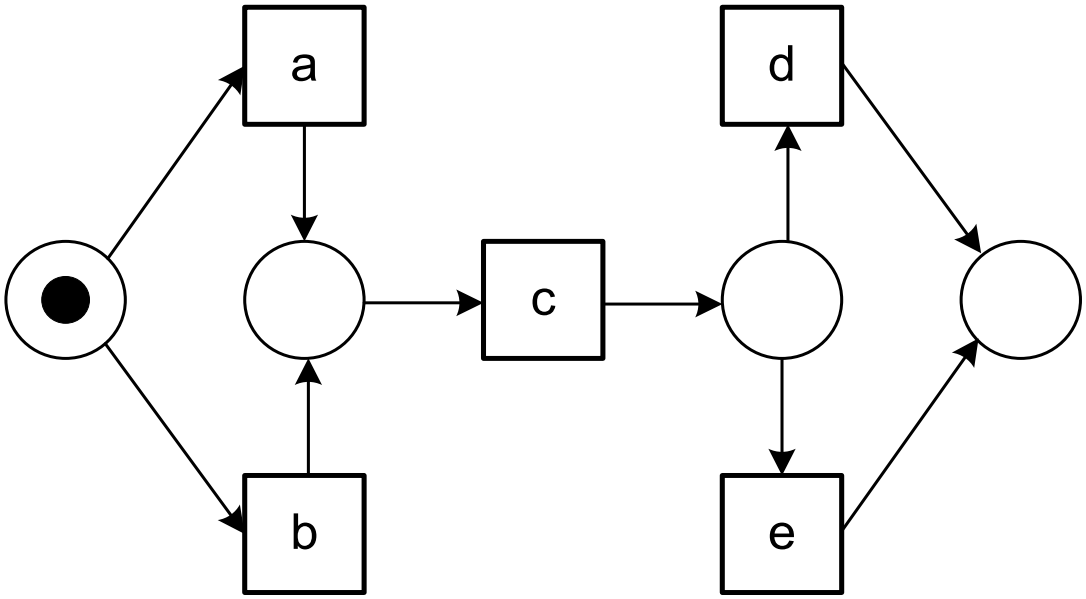
Generalization: good or bad?





Generalization: good!





not overfitting…

Simplicity: good or bad?

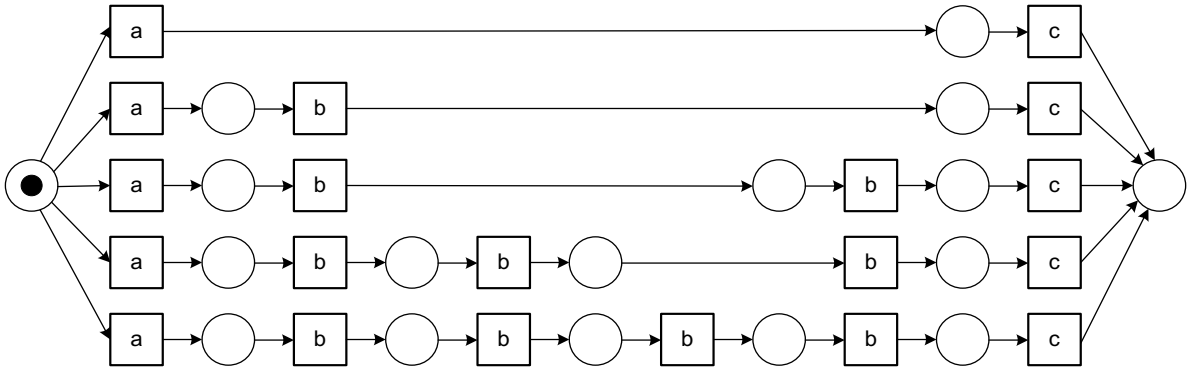
$\langle a,c \rangle^{16}$

$\langle a,b,c \rangle^{8}$

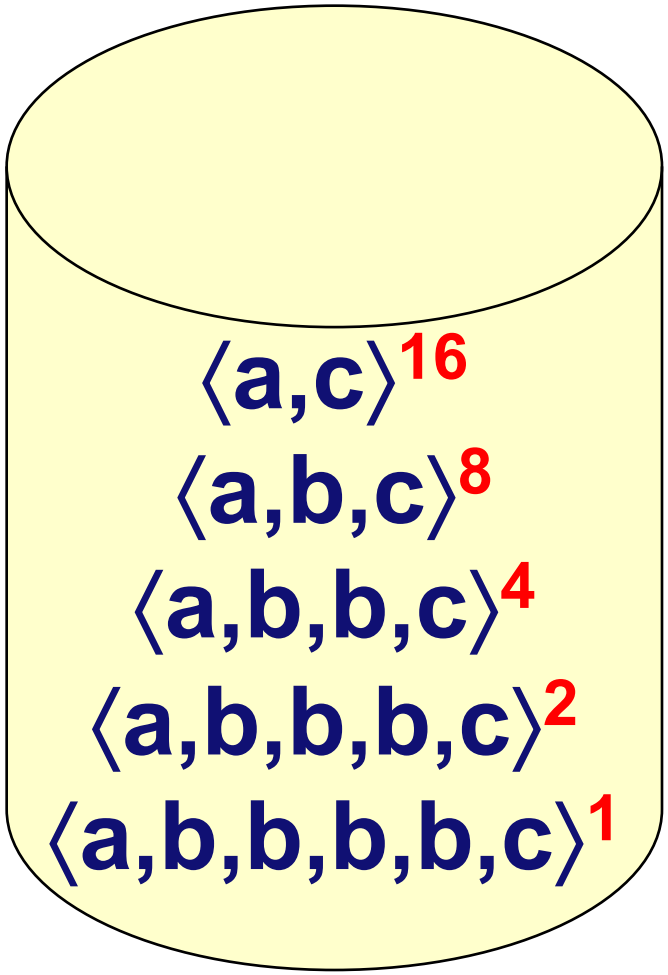
$\langle a,b,b,c \rangle^{4}$

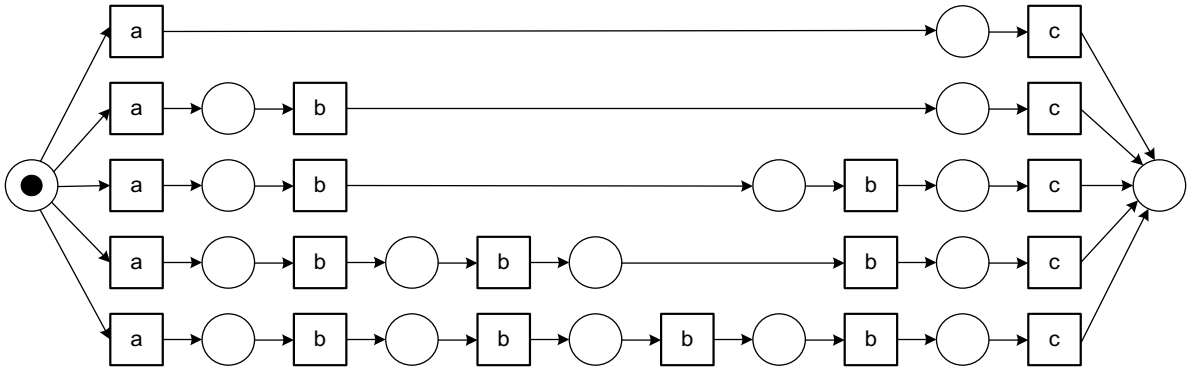
$\langle a,b,b,b,c \rangle^{2}$

$\langle a,b,b,b,b,c \rangle^{1}$



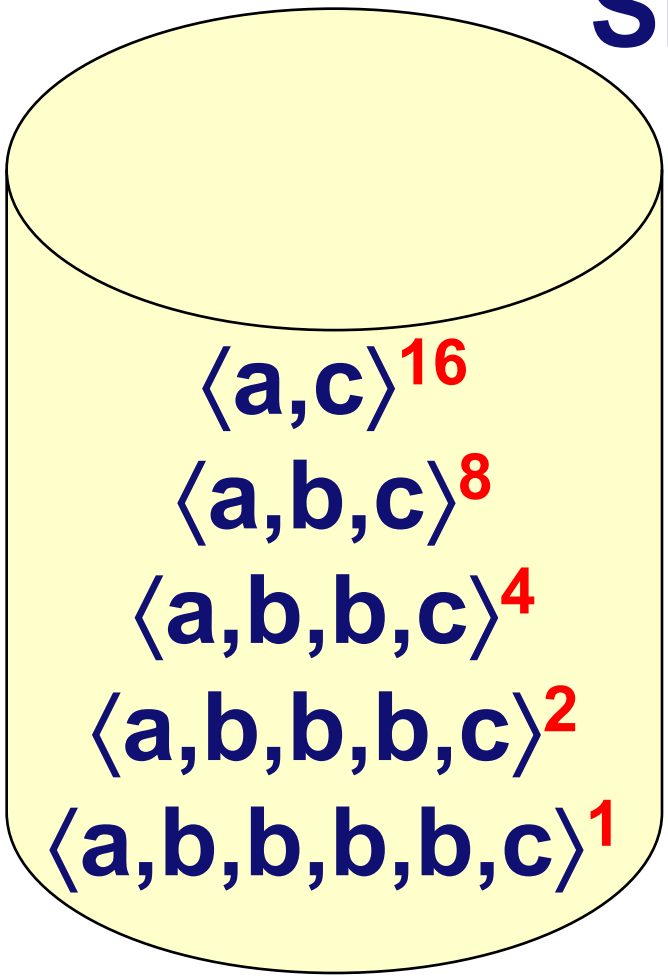
Simplicity: bad!

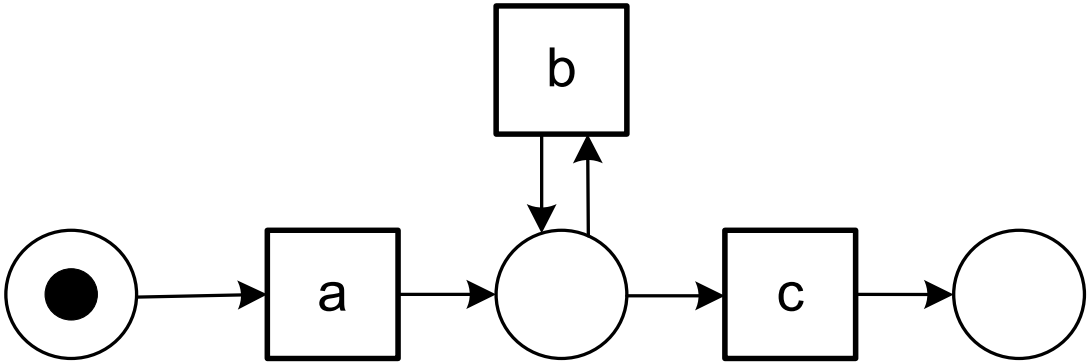




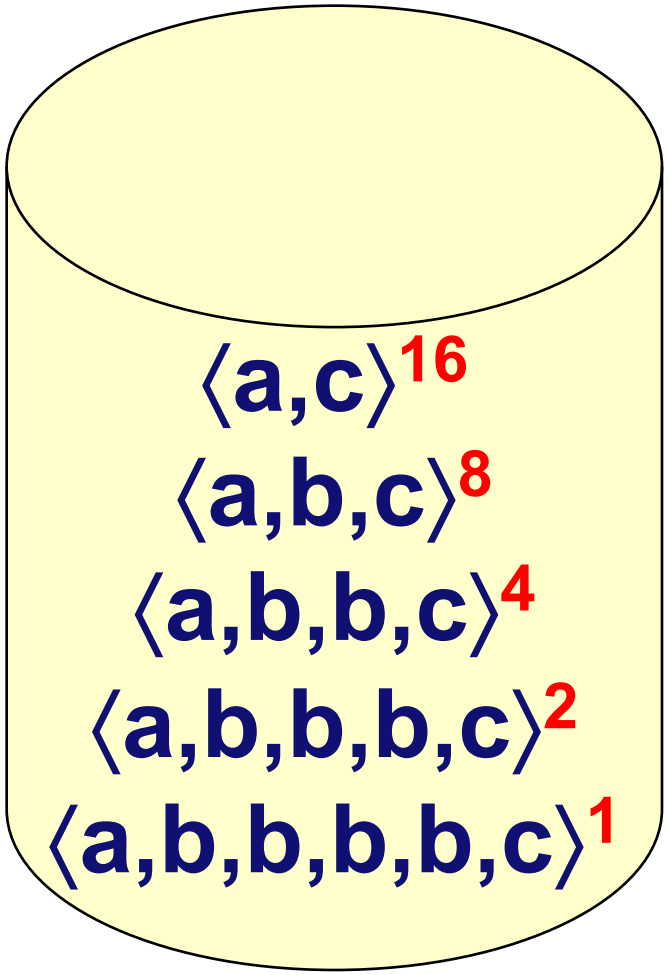
too complex/specific…

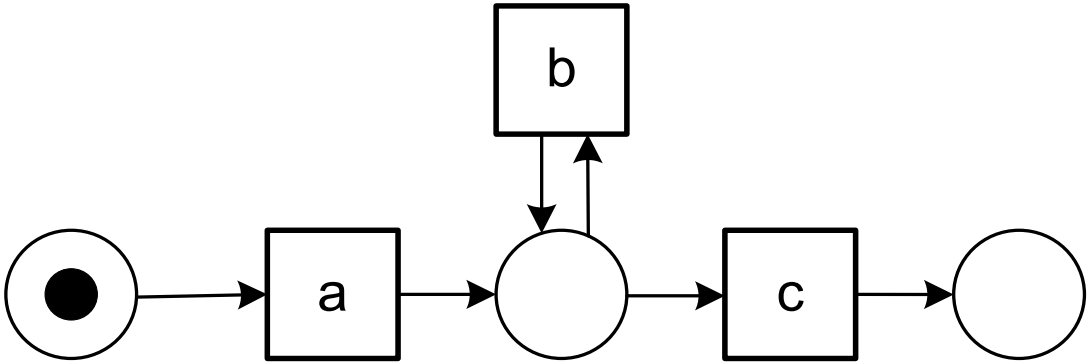
Simplicity: good or bad?



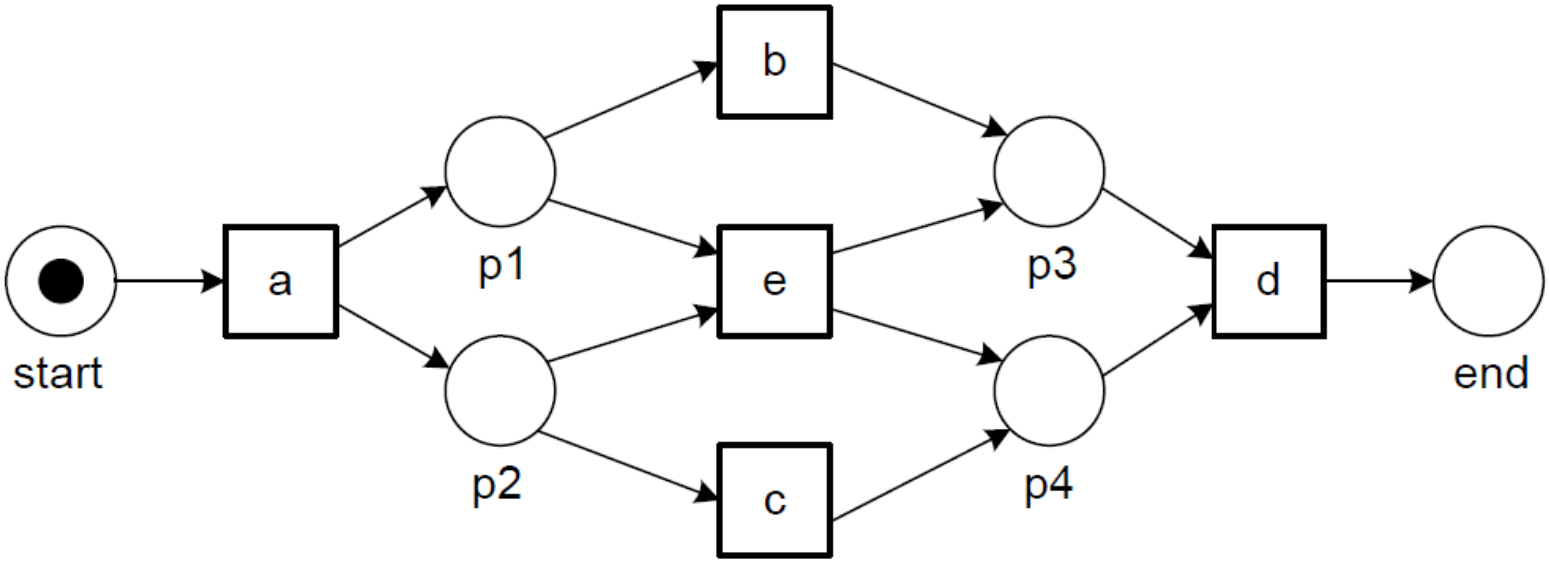


Simplicity: good!





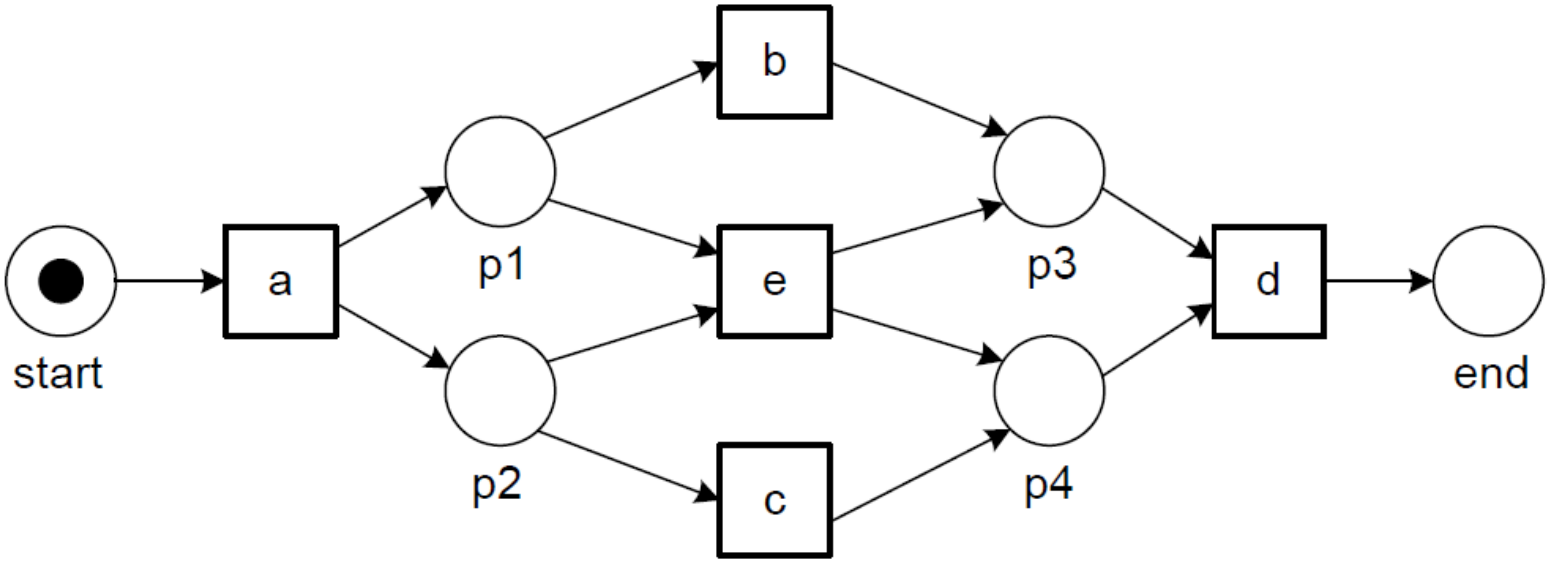
Discovery Example



$$L_1 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^2, \langle a,e,d\rangle]$$

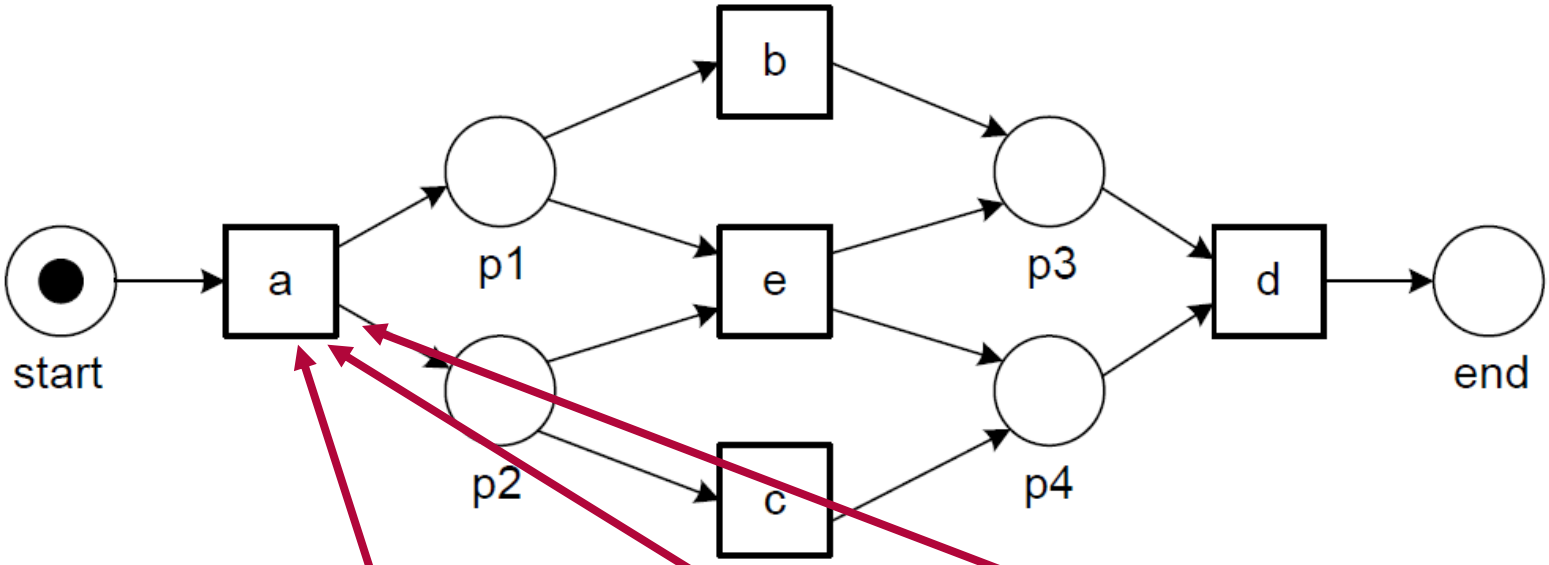
L_1 contains the sequence <a,b,c,d> three times

Discovery Example



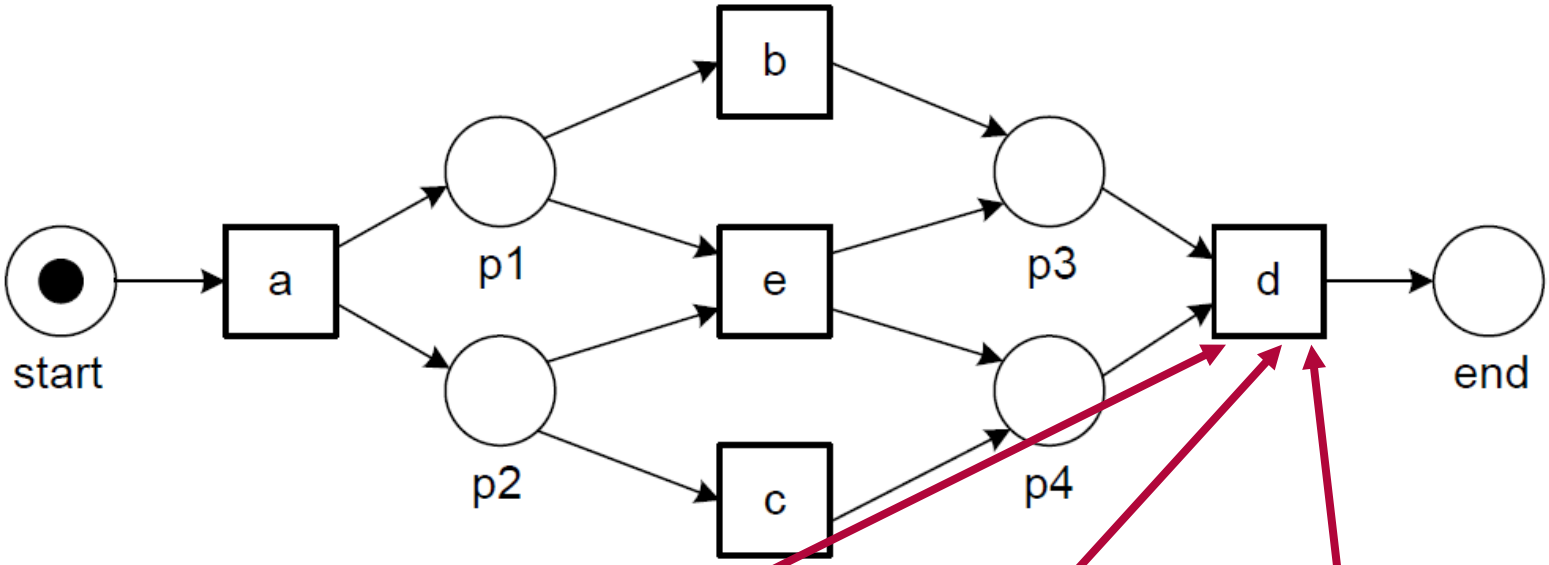
$$L_1 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^2, \langle a,e,d\rangle]$$

Discovery Example



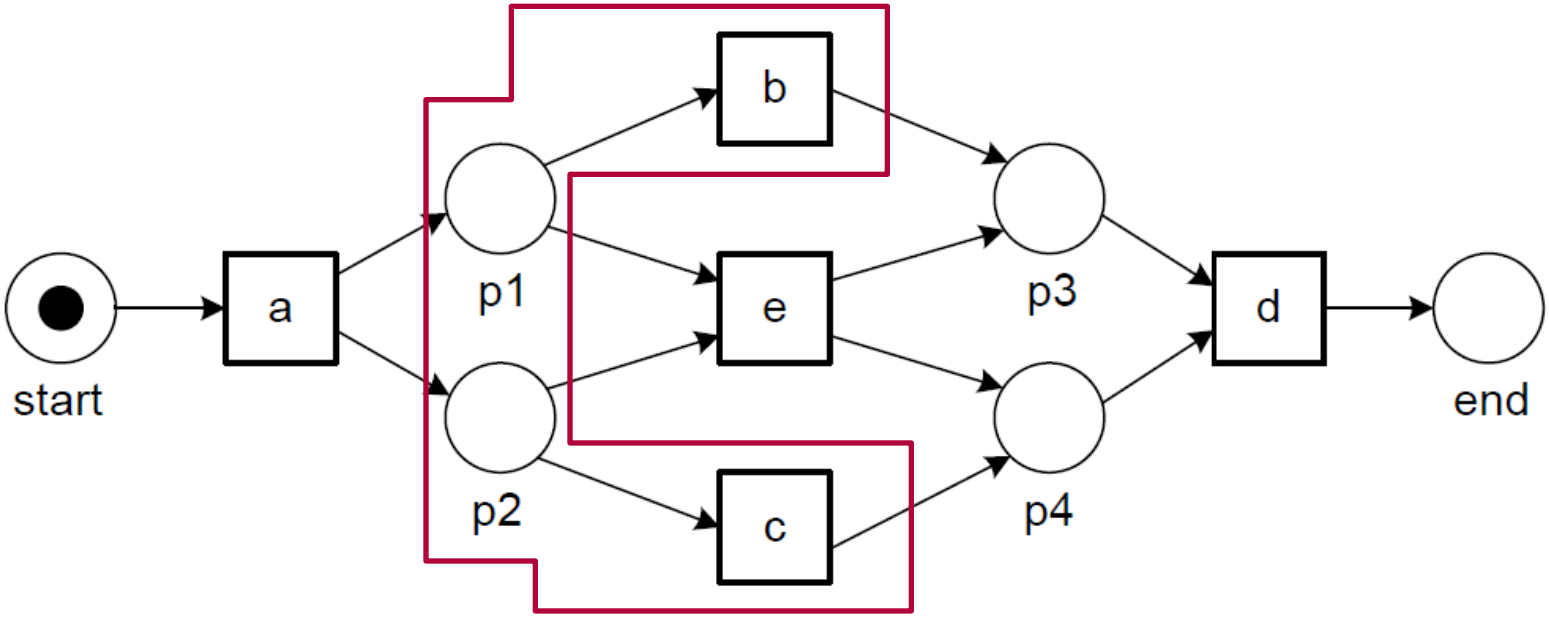
$$L_1 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^2, \langle a,e,d\rangle]$$

Discovery Example



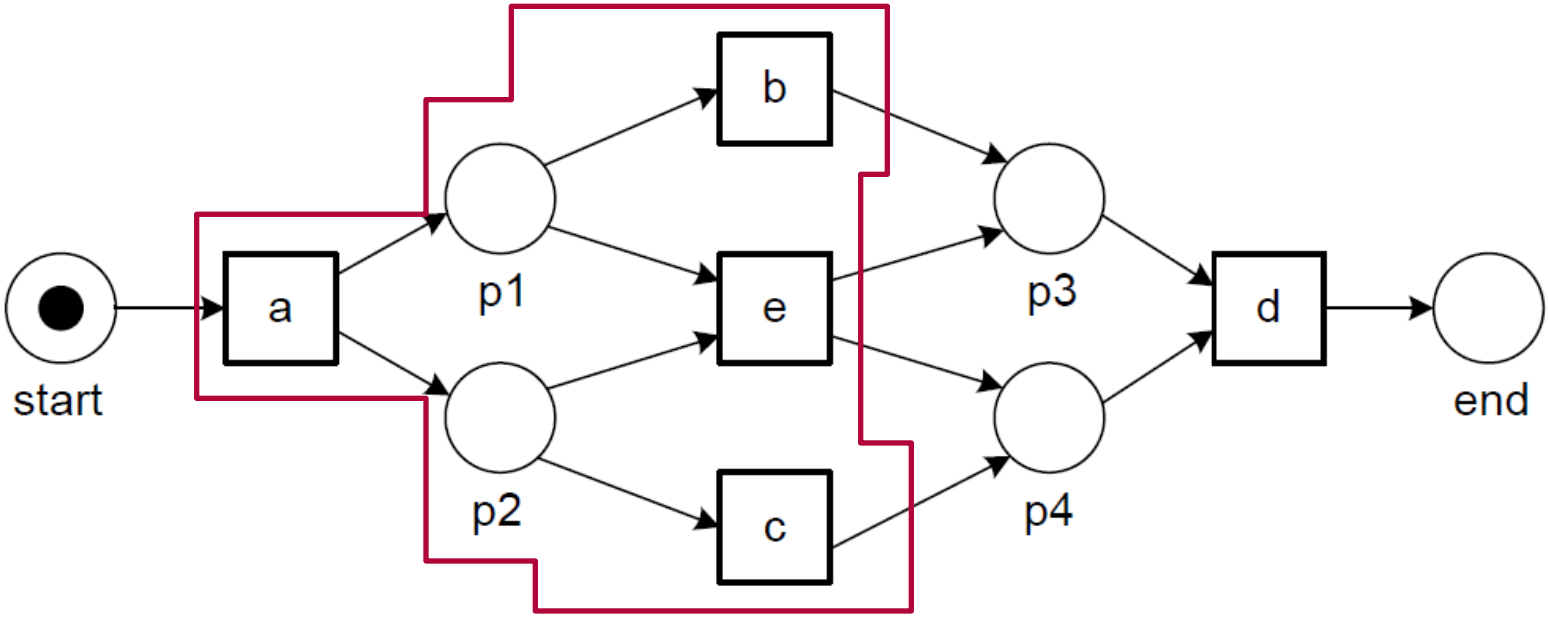
$$L_1 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^2, \langle a,e,d\rangle]$$

Discovery Example



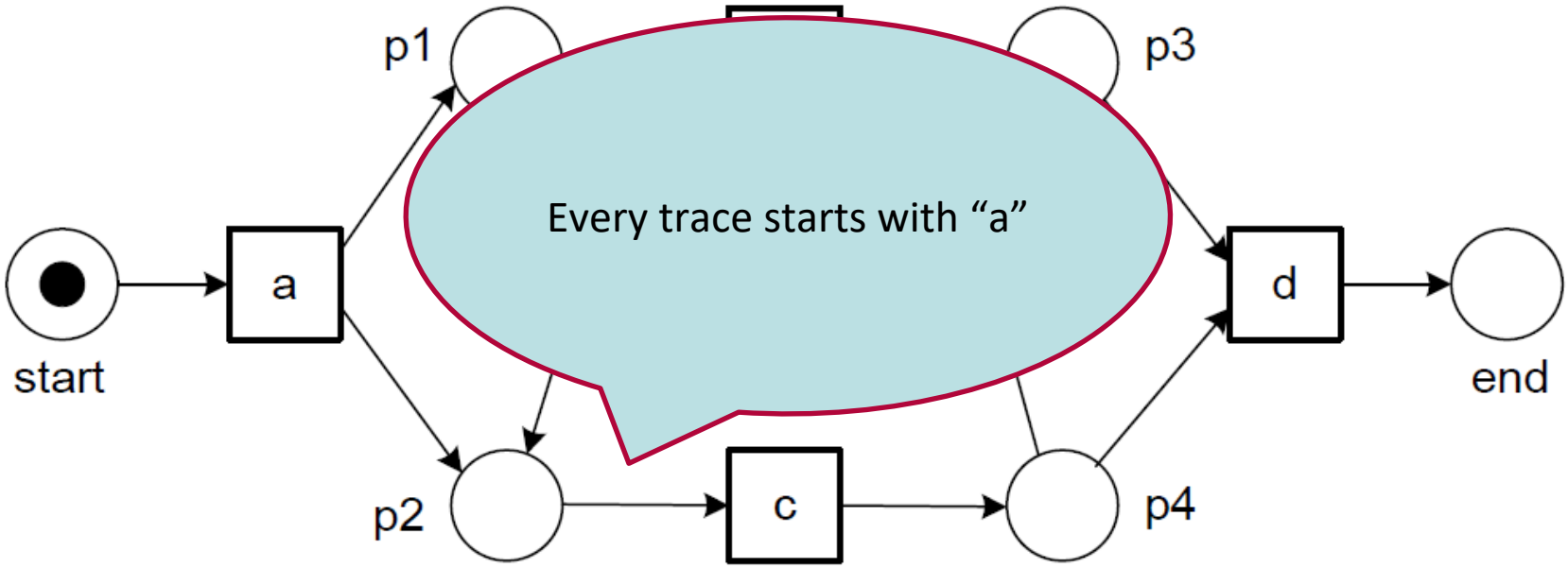
$$L_1 = [\langle a, b, c, d \rangle^3, \langle a, c, b, d \rangle^2, \langle a, e, d \rangle]$$

Discovery Example



$$L_1 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^2, \langle a,e,d\rangle]$$

Discovery Example



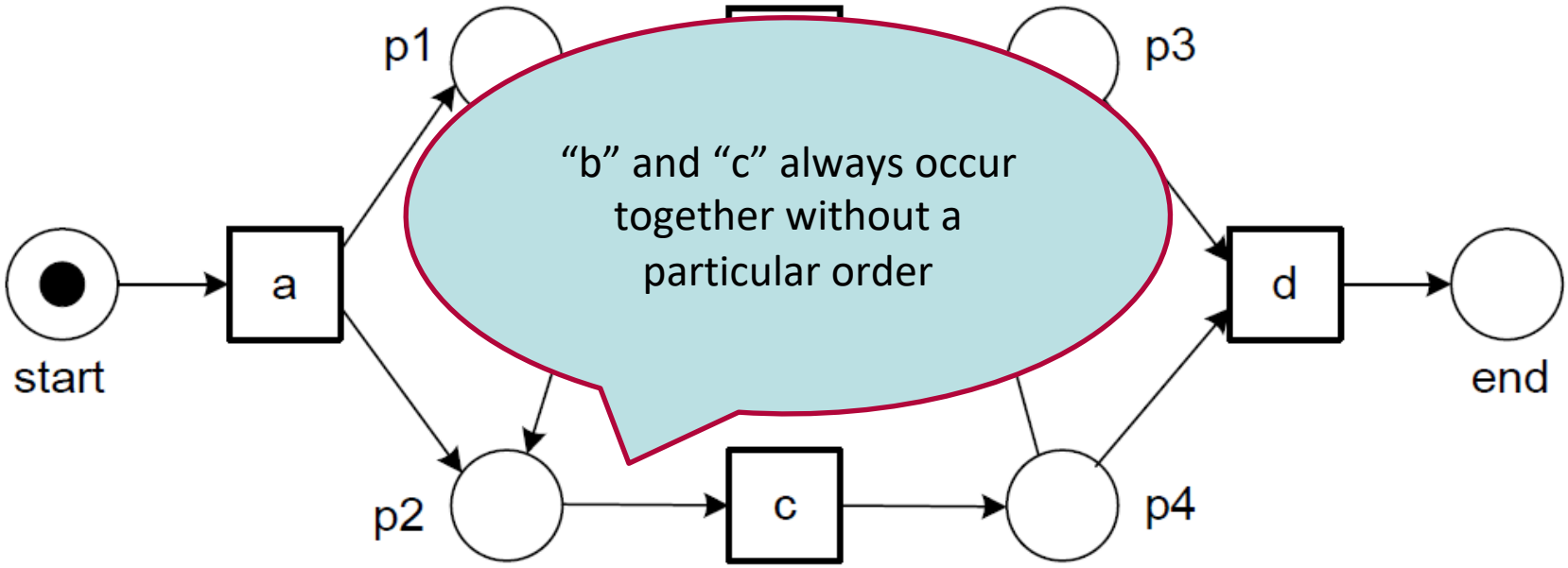
$$L_2 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^4, \langle a,b,c,e,f,b,c,d\rangle^2, \langle a,b,c,e,f,c,b,d\rangle,$$
$$\langle a,c,b,e,f,b,c,d\rangle^2, \langle a,c,b,e,f,b,c,e,f,c,b,d\rangle]$$

Discovery Example

Every trace ends with "d"

$$L_2 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^4, \langle a,b,c,e,f,b,c,d\rangle^2, \langle a,b,c,e,f,c,b,d\rangle,$$
$$\langle a,c,b,e,f,b,c,d\rangle^2, \langle a,c,b,e,f,b,c,e,f,c,b,d\rangle]$$

Discovery Example



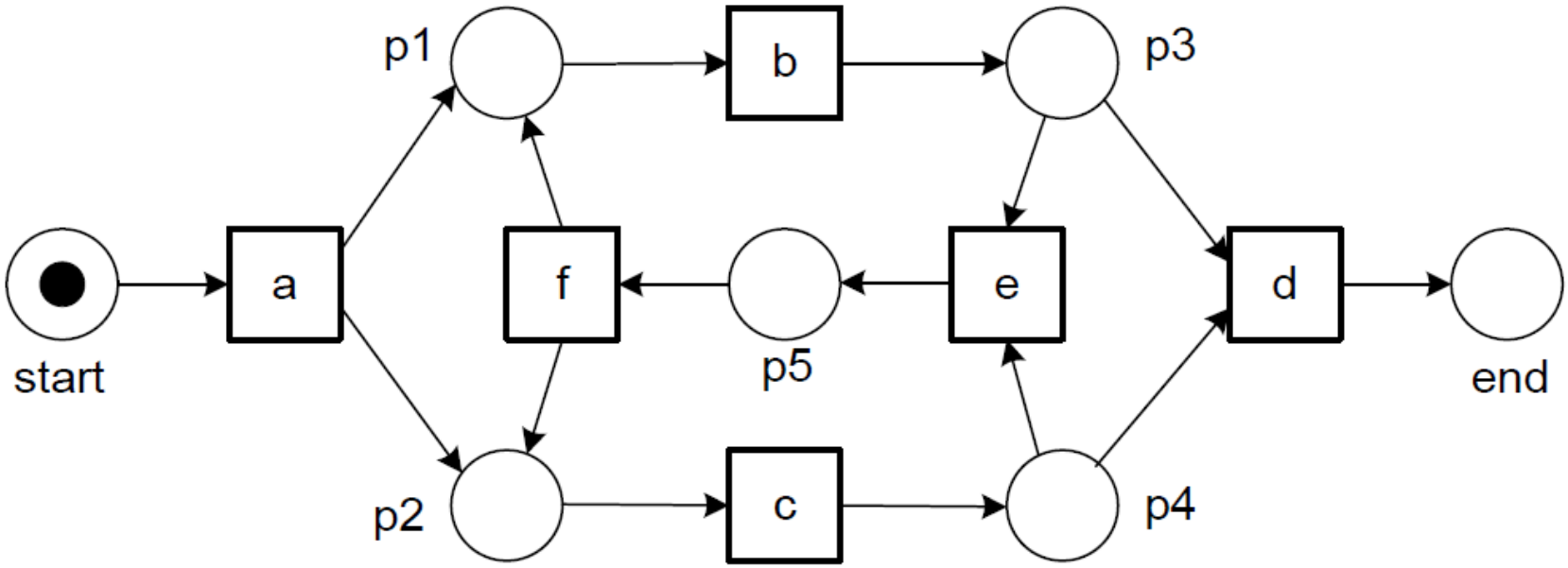
$$L_2 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^4, \langle a,b,c,e,f,b,c,d\rangle^2, \langle a,b,c,e,f,c,b,d\rangle,$$
$$\langle a,c,b,e,f,b,c,d\rangle^2, \langle a,c,b,e,f,b,c,e,f,c,b,d\rangle]$$

Discovery Example

Every "e" is followed by an "f",
every "f" is preceded by an "e"

$$L_2 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^4, \langle a,b,c,e,f,b,c,d\rangle^2, \langle a,b,c,e,f,c,b,d\rangle,$$
$$\langle a,c,b,e,f,b,c,d\rangle^2, \langle a,c,b,e,f,b,c,e,f,c,b,d\rangle]$$

Discovery Example



$$L_2 = [\langle a,b,c,d\rangle^3, \langle a,c,b,d\rangle^4, \langle a,b,c,e,f,b,c,d\rangle^2, \langle a,b,c,e,f,c,b,d\rangle,$$
$$\langle a,c,b,e,f,b,c,d\rangle^2, \langle a,c,b,e,f,b,c,e,f,c,b,d\rangle]$$

Alpha-family Discovery Algorithms

- Simple algorithm: α – algorithm
  - Detect concurrent execution of activities
  - Relatively easy, certain properties can be proven
  - Not robust against *noise*, incomplete or erroneous logs
  - Therefore, nice for illustration, but limited use in practice
- Extensions: α+(+) – Algorithms
  - α+ and α++ extend α – Algorithm to broaden the spectrum of constructs that may be discovered
  - Still not robust against *noise*
- Robust algorithms will be discussed later

Workflow Log

- General notion of event log
  - Format of log entries:
    (timestamp, caseID, activity, additional attributes …)
  - Various abstractions can be applied

- One abstraction: the notion of a workflow log
  - Set of all distinct sequences of activity executions
  - Let T be a set of activities (aka tasks)
  - Let T^* the set of all finite sequences over T
  - $\sigma \in T^*$ *is a trace, all tasks* in σ *belong to the same case*
  - $W \subseteq T^*$ *is a workflow log*

No cardinalities

No caseIDs, …

Assumption: Each task occurs at most once in a process model

Workflow Log

Traces:

Case 1: ABCD

Case 2: ACBD

Case 3: ABCD

Case 4: ACBD

Case 5: EF

No cardinalities, no caseIDs

Workflow log:

W = {ABCD, ACBD, EF}

```
case 1 : task A
case 2 : task A
case 3 : task A
case 3 : task B
case 1 : task B
case 1 : task C
case 2 : task C
case 4 : task A
case 2 : task B
case 2 : task D
case 5 : task E
case 4 : task C
case 1 : task D
case 3 : task C
case 3 : task D
case 4 : task B
case 5 : task F
case 4 : task D
```

Ordering Relations

Log-based ordering relations for a pair of tasks $a, b \in T$ in a workflow log W:

- Direct successor

  $a >_W b$ iff b directly follows a in at least one trace
- Causality

  $a \rightarrow_W b$ iff $a >_w b$ and not $b >_w a$
- Concurrency

  $a \parallel_W b$ iff $a >_w b$ and $b >_w a$
- Exclusiveness

  $a \#_W b$ iff not $a >_w b$ and not $b >_w a$

Ordering Relations Example

Workflow log:

W = {ABCD, ACBD, EF}

1)

A>B
A>C
B>C
B>D
C>B
C>D
E>F

2)

$A \rightarrow B$
$A \rightarrow C$
$B \rightarrow D$
$C \rightarrow D$
$E \rightarrow F$

3)

B||C
C||B

4)

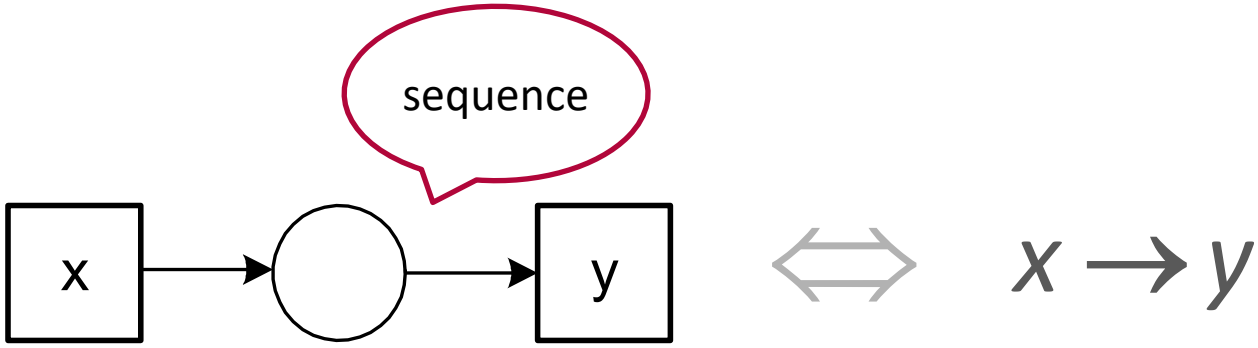
A # A
A # D
A # E
A # F
B # B
B # E
B # F
C # C
C # E
C # F
D # D
D # E
D # F
E # E
F # F
...

```
case 1 : task A
case 2 : task A
case 3 : task A
case 3 : task B
case 1 : task B
case 1 : task C
case 2 : task C
case 4 : task A
case 2 : task B
case 2 : task D
case 5 : task E
case 4 : task C
case 1 : task D
case 3 : task C
case 3 : task D
case 4 : task B
case 5 : task F
case 4 : task D
```

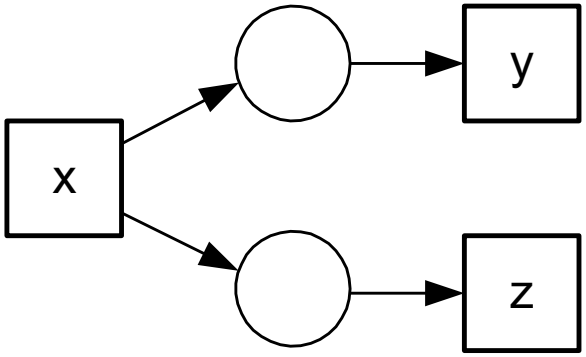
α-Algorithm Idea

Idea: create workflow net based on the ordering relations, such that the ordering is obeyed by the net

Realisation: derive a Petri net fragment from the each entry of the ordering relations

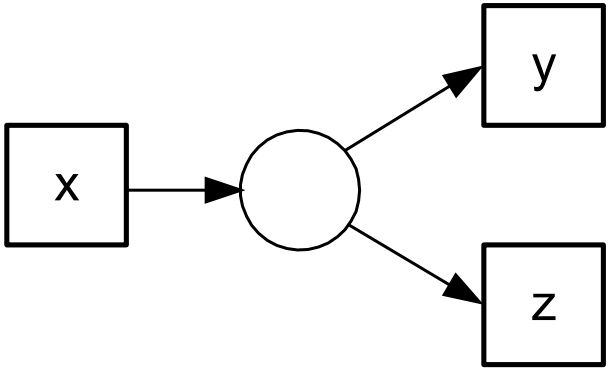


α-Algorithm Idea (cont'd)



AND-split

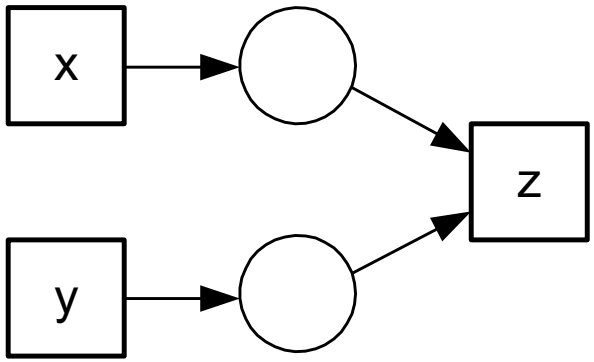
$x \rightarrow y, x \rightarrow z$ and $y \,||\, z$



XOR-split

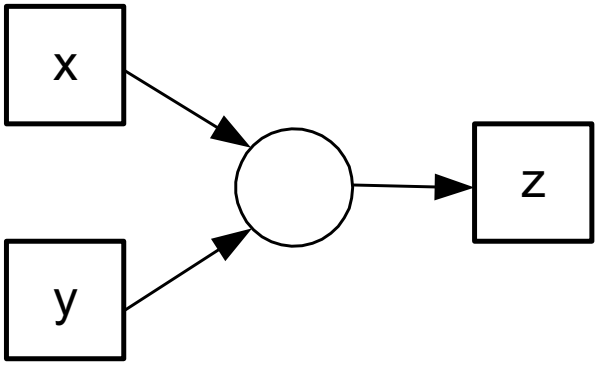
$x \rightarrow y, x \rightarrow z$ and $y \# z$

α-Algorithm Idea (cont'd)



AND-join

$x \rightarrow z,\ y \rightarrow z$
and $x \mid\mid y$



XOR-join

$x \rightarrow z,\ y \rightarrow z$
and $x \# y$

α-Algorithm

Let *W* be a workflow log over *T*. $\alpha(W)$ is defined as follows:

1. $T_W = \{ t \in T \mid \exists_{\sigma \in W}\, t \in \sigma \}$,
2. $T_I = \{ t \in T \mid \exists_{\sigma \in W}\, t = \mathit{first}(\sigma) \}$,
3. $T_O = \{ t \in T \mid \exists_{\sigma \in W}\, t = \mathit{last}(\sigma) \}$,
4. $X_W = \{ (A,B) \mid A \subseteq T_W \wedge B \subseteq T_W \wedge \forall_{a \in A} \forall_{b \in B}\, a \rightarrow_W b \wedge \forall_{a1,a2 \in A}\, a_1 \#_W a_2 \wedge \forall_{b1,b2 \in B}\, b_1 \#_W b_2 \}$,
5. $Y_W = \{ (A,B) \in X_W \mid \forall_{(A',B') \in X_W}\, A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B') \}$,
6. $P_W = \{ p_{(A,B)} \mid (A,B) \in Y_W \} \cup \{i_W, o_W\}$,
7. $F_W = \{ (a, p_{(A,B)}) \mid (A,B) \in Y_W \wedge a \in A \} \cup \{ (p_{(A,B)}, b) \mid (A,B) \in Y_W \wedge b \in B \} \cup \{ (i_W, t) \mid t \in T_I \} \cup \{ (t, o_W) \mid t \in T_O \}$,
8. $\alpha(W) = (P_W, T_W, F_W)$.

α-Algorithm

Let W be a workflow log over T. $\alpha(W)$ is defined as follows:

1. $T_W = \{ t \in T \mid \exists_{\sigma \in W}\, t \in \sigma \}$,
2. $T_I = \{ t \in T \mid \exists_{\sigma \in W}\, t = \mathit{first}(\sigma) \}$,
3. $T_O = \{ t \in T \mid \exists_{\sigma \in W}\, t = \mathit{last}(\sigma) \}$,
4. $X_W = \{ (A,B) \mid A \subseteq T_W \wedge B \subseteq T_W \wedge \forall_{a \in A} \forall_{b \in B}\, a \rightarrow_W b \wedge \forall_{a1,a2 \in A}\, a_1 \#_W a_2 \wedge \forall_{b1,b2 \in B}\, b_1 \#_W b_2 \}$,
5. $Y_W = \{ (A,B) \in X_W \mid \forall_{(A',B') \in X_W}\, A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B') \}$,
6. $P_W = \{ p_{(A,B)} \mid (A,B) \in Y_W \} \cup$
7. $F_W = \{ (a,p_{(A,B)}) \mid (A,B) \in Y_W$
   $\{ (p_{(A,B)},b) \mid (A,B) \in Y_W \wedge$
   $\{ (i_W,t) \mid t \in T_I \} \cup \{ (t,o_W$
8. $\alpha(W) = (P_W,T_W,F_W)$.

Result is a WF-net:

P_W : Set of places

T_W : Set of transitions

F_W : Flow relation

α-Algorithm

Let W be a workflow log over T. $\alpha(W)$ is defined as follows:

1. $T_W = \{ t \in T \mid \exists_{\sigma \in W}\ t \in \sigma \}$,
2. $T_I = \{ t \in T \mid \exists_{\sigma \in W}\ t = \mathit{first}(\sigma) \}$,
3. $T_O = \{ t \in T \mid \exists_{\sigma \in W}\ t = \mathit{last}(\sigma) \}$,
4. $X_W = \{ (A,B) \mid A \subseteq T_W \wedge B \subseteq T_W \wedge \forall_{a1,a2 \in A}\ a_1 \#_W a_2 \wedge \forall_{b1,b2 \in B}$ [illegible] $\}$,
5. $Y_W = \{ (A,B) \in X_W \mid \forall_{(A',B') \in X_W} (A,B) = (A',B') \}$,
6. $P_W = \{ p_{(A,B)} \mid (A,B) \in Y_W \} \cup$
7. $F_W = \{ (a,p_{(A,B)}) \mid (A,B) \in Y_W \wedge$
   $\{ (p_{(A,B)},b) \mid (A,B) \in Y_W \wedge$
   $\{ (i_W,t) \mid t \in T_I \} \cup \{ (t,o_W)$
8. $\alpha(W) = (P_W,T_W,F_W)$.

Derive set of transitions T_W from all traces.

$\mathit{first}(\sigma)$ [$\mathit{last}(\sigma)$] is the first [last] transition in trace σ.

T_I [T_O] is the set of all initial [final] transitions.

$W = \{ABCD, ACBD, EF\}$

$T_W = \{A, B, C, D, E, F\}$

$T_I = \{A, E\}$

$T_O = \{D, F\}$

α-Algorithm

Let *W* be a workflow log over *T*. $\alpha(W$

$T_I = \{A, E\}$

$T_O = \{D, F\}$

$\{ (i_W,t) \mid t \in T_I\} = \{(i_W,A), (i_W,E)\}$

$\{ (t,o_w) \mid t \in T_O\} = \{(D,o_w), (F,o_w)\}$

i_W is initial place, o_W is final place.

Step 7 connects them to transitions in T_I and T_O.

$\wedge \ \forall_{a \in A} \forall_{b \in B} a \rightarrow_W b \ \wedge$

$_W b_2 \},$

5. $Y_W = \{ (A,B) \in X_W \mid \ \forall_{(A',B') \in X_W} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B') \},$
6. $P_W = \{ p_{(A,B)} \mid (A,B) \in Y_W \} \cup \{i_W,o_W\},$
7. $F_W = \{ (a,p_{(A,B)}) \mid (A,B) \in Y_W \wedge a \in A \} \cup \{ (p_{(A,B)},b) \mid (A,B) \in Y_W \wedge b \in B \} \cup \{ (i_W,t) \mid t \in T_I\} \cup \{ (t,o_W) \mid t \in T_O\},$
8. $\alpha(W) = (P_W,T_W,F_W).$

α-Algorithm

Let *W* be a workflow log over *T*. $\alpha(W)$ is defined as follows:

1. $T_W = \{ t \in T \mid \exists_{\sigma \in W}\ t \in \sigma \}$,
2. $T_I = \{ t \in T \mid \exists_{\sigma \in W}\ t = first(\sigma) \}$,
3. $T_O = \{ t \in T \mid \exists_{\sigma \in W}\ t = last(\sigma) \}$
4. $X_W = \{ (A,B) \mid A \subseteq T_W \wedge B \subseteq T_W$
   $\forall_{a1,a2 \in A}\ a_1 \#_W a_2 \ \wedge \ \forall_{b1,b2 \in B}\ b_1$
5. $Y_W = \{ (A,B) \in X_W \mid \forall_{(A',B') \in X_W} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B') \}$,
6. $P_W = \{ p_{(A,B)} \mid (A,B) \in Y_W \} \cup \{i_W, o_W\}$,
7. $F_W = \{ (a, p_{(A,B)}) \mid (A,B) \in Y_W \wedge a \in A \} \cup$
   $\{ (p_{(A,B)}, b) \mid (A,B) \in Y_W \wedge b \in B \} \cup$
   $\{ (i_W, t) \mid t \in T_I \} \cup \{ (t, o_W) \mid t \in T_O \}$,
8. $\alpha(W) = (P_W, T_W, F_W)$.

All other places are denoted by $p_{(A,B)}$, with A and B being preceding / succeeding transitions.

Place is created, iff $a \rightarrow_{\mathbf{w}} b$.

Let *W* be a workflow log c

1. $T_W = \{ t \in T \mid \exists_{\sigma \in W} t \in$
2. $T_I = \{ t \in T \mid \exists_{\sigma \in W} t =$
3. $T_O = \{ t \in T \mid \exists_{\sigma \in W} t =$
4. $X_W = \{ (A,B) \mid A \subseteq T_W \wedge B \subseteq T_W \wedge \forall_{a \in A} \forall_{b \in B} a \rightarrow_W b \wedge \forall_{a1,a2 \in A} a_1 \#_W a_2 \wedge \forall_{b1,b2 \in B} b_1 \#_W b_2 \}$,
5. $Y_W = \{ (A,B) \in X_W \mid \forall_{(A',B') \in X_W} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A', B') \}$

Some places are joint to represent XOR-splits/joins instead of AND-splits/joins. To distinguish those, relations X_W and Y_W are defined.

$(A,B) \in X_W$ if causality between all A and all B and elements in A (B) are never direct successors.

P_W = { $p_{(A,B)}$ | $(A,B) \in Y_W$ } ∪ { i_W, t }

$\alpha(W) = (P_W$

A→B
A→C
B→D
C→D
E→F

B||C
C||B

$T_W = \{A, B, C, D, E, F\}$

$X_W = \{ (\{A\},\{B\}), (\{A\},\{C\}), (\{B\},\{D\}), (\{C\},\{D\}), (\{E\},\{F\})\}$

Since B||C, tuple ({A}, {B,C}) and ({B,C},{D}) are not in X_W

α-Algorithm

Let W be a workflow log over T. $\alpha(W)$ is defined as follows:

1. $T_W = \{ t \in T \mid \exists_{\sigma \in W}\, t \in \sigma \}$,
2. $T_I = \{ t \in T \mid \exists_{\sigma \in W}\, t = \mathit{first}(\sigma) \}$,
3. $T_O = \{ t \in T \mid \exists_{\sigma \in W}\, t = \mathit{last}(\sigma) \}$,
4. $X_W = \{ (A,B) \mid A \subseteq T_W \wedge B \subseteq T_W \wedge \forall_{a \in A} \forall_{b \in B}\, a \rightarrow_W b \wedge \forall_{a1,a2 \in A}\, a_1 \#_W a_2 \wedge \forall_{b1,b2 \in B}\, b_1 \#_W b_2 \}$,
5. $Y_W = \{ (A,B) \in X_W \mid \forall_{(A',B') \in X_W}\, A \subseteq A' \wedge B \subseteq B' \Rightarrow (A,B) = (A',B') \}$,
6. $P_W = \{ p_{(A,B)} \mid (A,B) \in Y_W \} \cup \{i_W, o_W\}$,
7. $F_W = \{ (a, p_{(A,B)}) \mid (A,B) \in Y_W \wedge a \in A \} \cup$

Y_W is derived from X_W, by taking the largest sets in terms of set containment.

$Y_W = \{ (\{A\},\{B\}), (\{A\},\{C\}), (\{B\},\{D\}), (\{C\},\{D\}), (\{E\},\{F\}) \}$

It holds $Y_W = X_W$ since $\forall\ (A,B) \in X_W : |A| = 1 \wedge |B| = 1$

α-Algorithm

$Y_W = \{ (\{A\},\{B\}), (\{A\},\{C\}), (\{B\},\{D\}), (\{C\},\{D\}), (\{E\},\{F\})\}$

$P_W = \{ p_{(\{A\},\{B\})}, p_{(\{A\},\{C\})}, p_{(\{B\},\{D\})}, p_{(\{C\},\{D\})}, p_{(\{E\},\{F\})}, i_W, o_W\}$

$F_W = \{(A,p_{(\{A\},\{B\})}), (A,p_{(\{A\},\{C\})}), (B,p_{(\{B\},\{D\})}), (C,p_{(\{C\},\{D\})}), (E,p_{(\{E\},\{F\})}), (p_{(\{A\},\{B\})},B), (p_{(\{A\},\{C\})},C), (p_{(\{B\},\{D\})},D), (p_{(\{C\},\{D\})},D), (p_{(\{E\},\{F\})},F), (i_W,A), (i_W,E), (D, o_W), (F, o_W)\}$

4. $X_W = \{ (A,B) \mid A \subseteq T_W \wedge B \subseteq T_W \wedge \forall_{a \in A} \forall_{b \in B} a \rightarrow_W b \wedge \forall_{a1,a2 \in A} a_1 \#_W a_2 \wedge \forall_{b1,b2 \in B} b_1 \#$ …

5. $Y_W = \{ (A,B) \in X_W \mid \forall_{(A',B') \in X_W} A$ … $(A',B') \}$,

Use Y_W to create places and the flow relation.

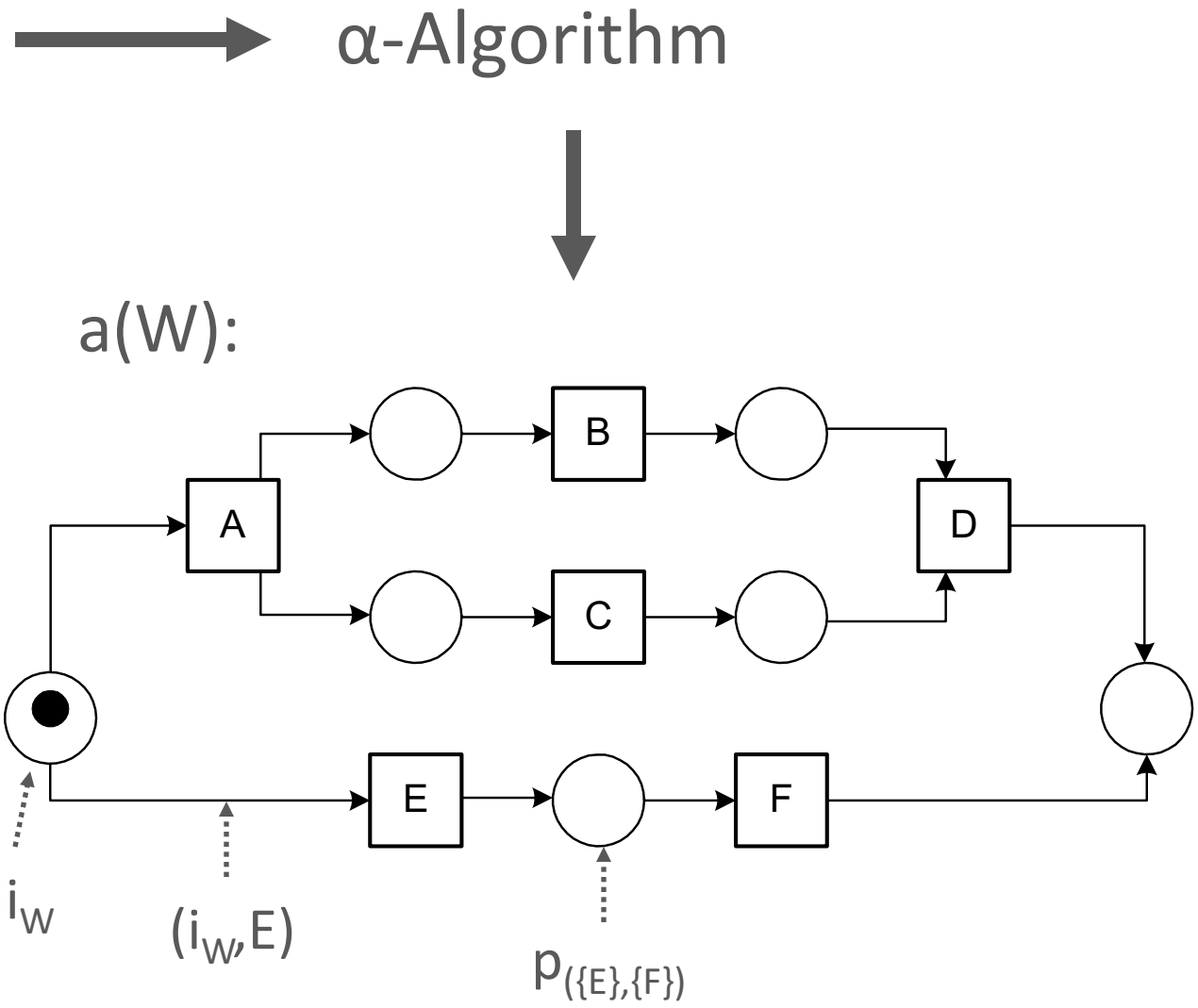
6. $P_W = \{ p_{(A,B)} \mid (A,B) \in Y_W \} \cup \{i_W,o_W\}$,

7. $F_W = \{ (a,p_{(A,B)}) \mid (A,B) \in Y_W \wedge a \in A \} \cup \{ (p_{(A,B)},b) \mid (A,B) \in Y_W \wedge b \in B \} \cup \{ (i_W,t) \mid t \in T_I\} \cup \{ (t,o_W) \mid t \in T_O\}$,

8. $\alpha(W) = (P_W,T_W,F_W)$.

α-Algorithm Example

```
case 1 : task A
case 2 : task A
case 3 : task A
case 3 : task B
case 1 : task B
case 1 : task C
case 2 : task C
case 4 : task A
case 2 : task B
case 2 : task D
case 5 : task E
case 4 : task C
case 1 : task D
case 3 : task C
case 3 : task D
case 4 : task B
case 5 : task F
case 4 : task D
```



Log Completeness

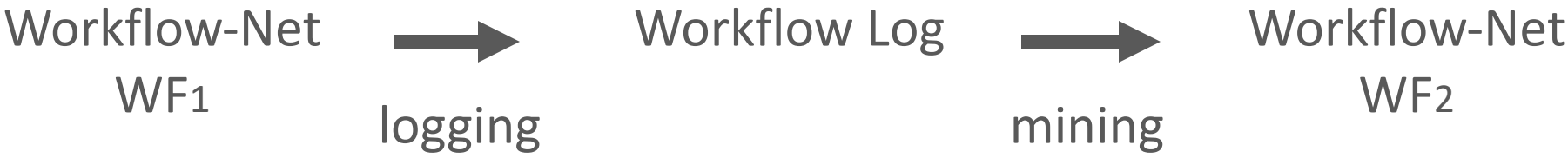
Which information is required to be in the log?

- Consider the example given for the α – algorithm and assume trace EF for instance 5 is missing in the log
- Then, the correct model cannot be created
- Assume that every traces has to occur in the log?
  - Problem: number of required instances increases dramatically
  - Consider the parallel execution of 10 tasks
  - This yields 10! = 3.628.800 possible traces!

Log Completeness

- Required completeness for the α-algorithm
  - Completeness w.r.t. direct successorship ($>_W$)
  - Whenever two tasks may occur as direct successors, this must be observed in at least *one* trace
  - Example: concurrent execution of five tasks, A,B,C,D,E
  - If ABCDE and BACED are observed, BACDE is not required for log completeness!
- This completeness criterion reduced the number of required traces for highly concurrent processes dramatically

Rediscovery Problem

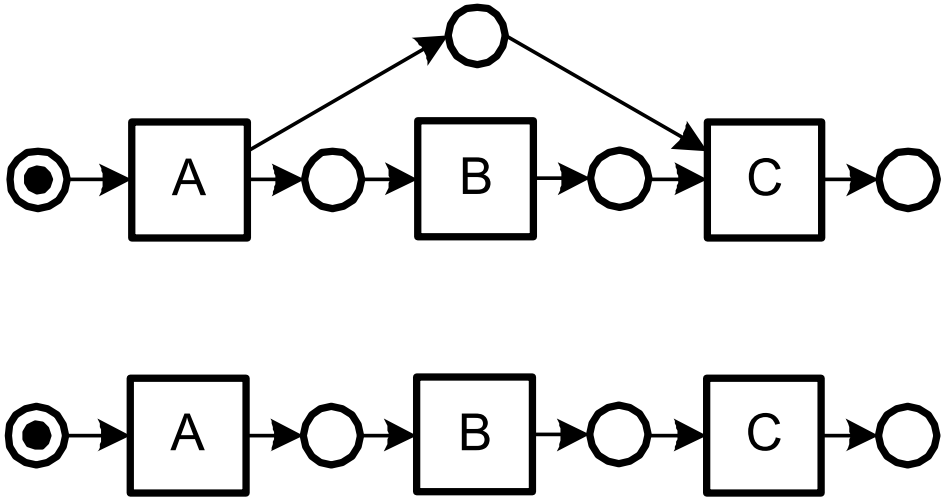


For which class of WF-nets can we guarantee that WF_1 and WF_2 are equivalent if logging is complete according to introduced notion?

Structural and behavioural assumptions on WF_1

Implicit Places

- The presence or absence of implicit places does not change the behaviour of a net system
  - Hence, process models with implicit places cannot be re-discovered
  - Not really an issue, since there are no consequences for the behaviour

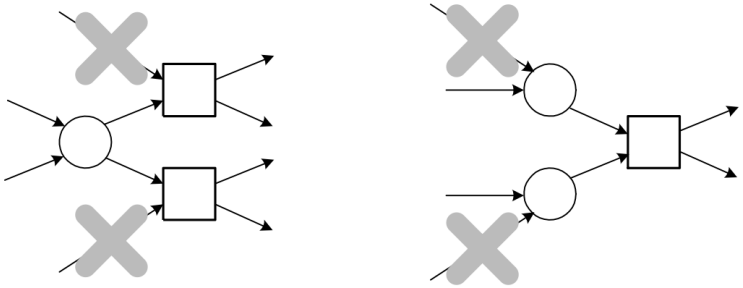


Structured Workflow-nets

Structured WF-nets (SWF-nets) are structural subclass:

Definition 4.3. (SWF-net) A WF-net $N = (P, T, F)$ is an *SWF-net* (Structured workflow net) iff:

1. For all $p \in P$ and $t \in T$ with $(p, t) \in F$: $|p\bullet| > 1$ implies $|\bullet t| = 1$.
2. For all $p \in P$ and $t \in T$ with $(p, t) \in F$: $|\bullet t| > 1$ implies $|\bullet p| = 1$.
3. There are no implicit places.



Note:
Sufficiently expressive to model most process-related control-flow structure sequences, concurrency, exclusive choices, etc.

Soundness (recap)

- Behavioural correctness criterion for WF-nets
  - Processes terminate in proper final state
  - Final state is indeed characterised unambiguously
  - All activities can contribute to process execution

- Recall: WF-net has initial place *i* and final place *o*
  - Overload notation and refer to *i* and *o* also as the markings that put one token into i and o, respectively, and no token in any other place

Soundness Formal Def.

Definition 6.3 A workflow system (PN, i) with a workflow net $PN = (P, T, F)$ is *sound*, if and only if

- For every state M reachable from state i there exists a firing sequence leading from M to o, i.e.,

$$\forall M (i \xrightarrow{*} M) \implies (M \xrightarrow{*} o)$$

- State o is the only state reachable from state i with at least one token in place o, i.e.,

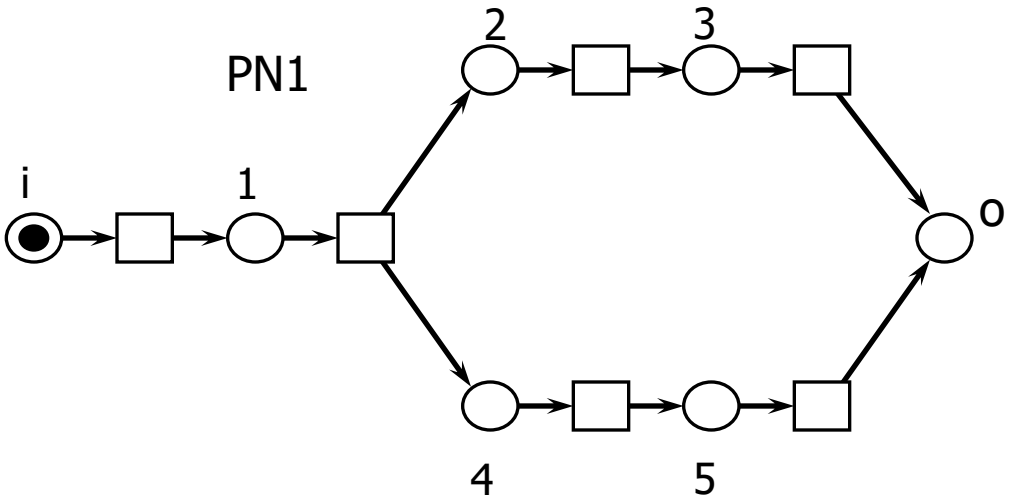
$$\forall M (i \xrightarrow{*} M \wedge M \geq o) \implies M = o$$

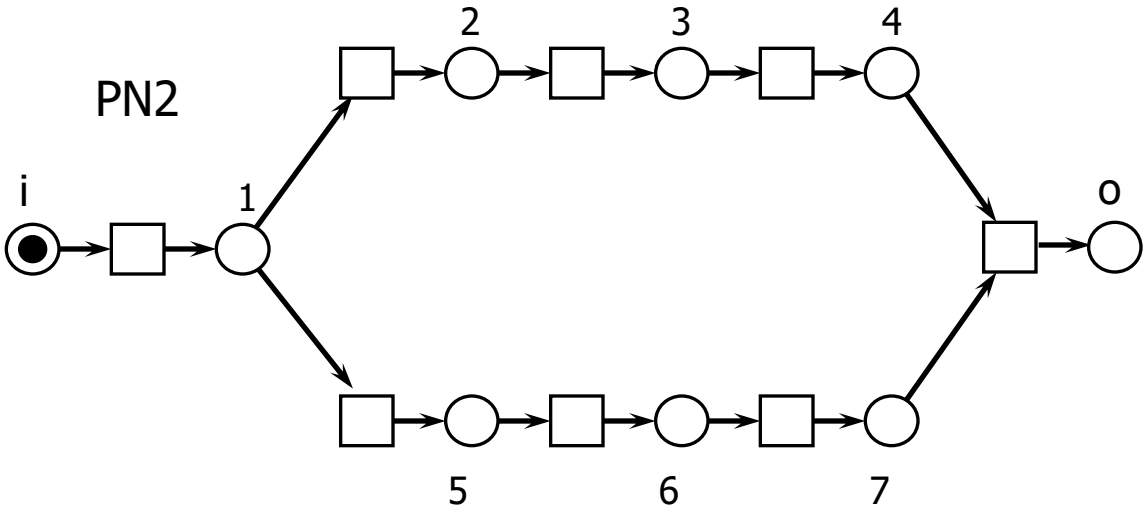
- There are no dead transitions in the workflow net in state i, i.e.,

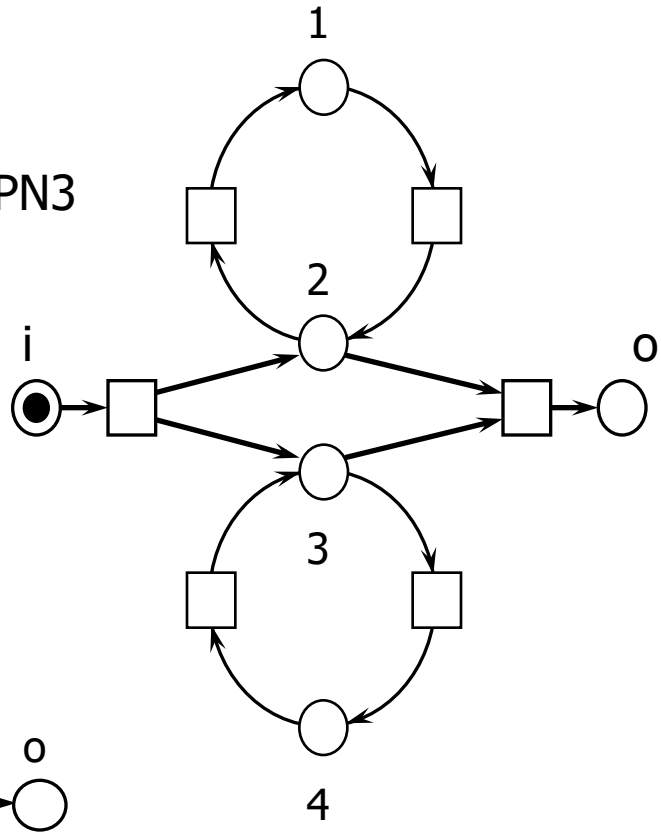
$$(\forall t \in T)\ \exists M, M' : i \xrightarrow{*} M \xrightarrow{t} M'$$

$\diamond$

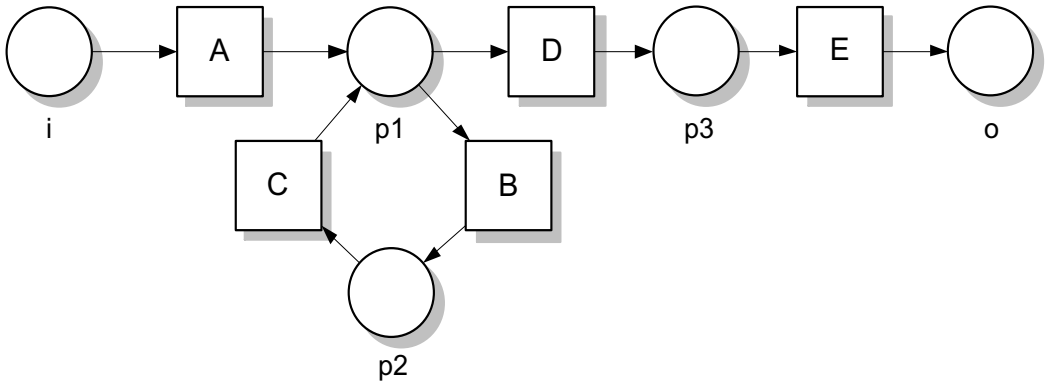
Soundness Examples

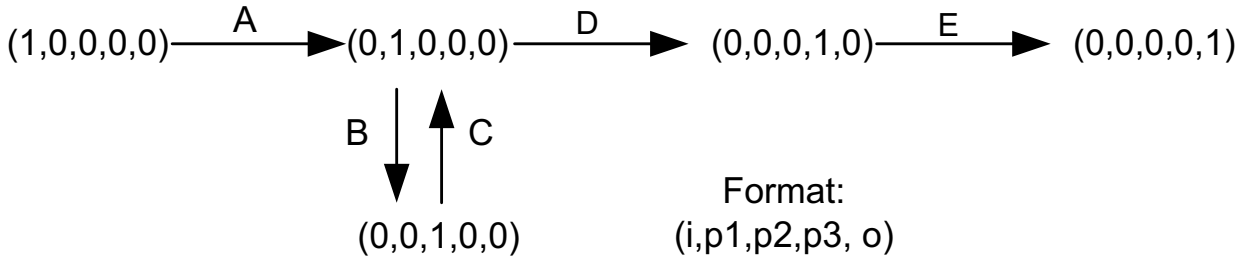






Reachability Graph (recap)





Checking Soundness

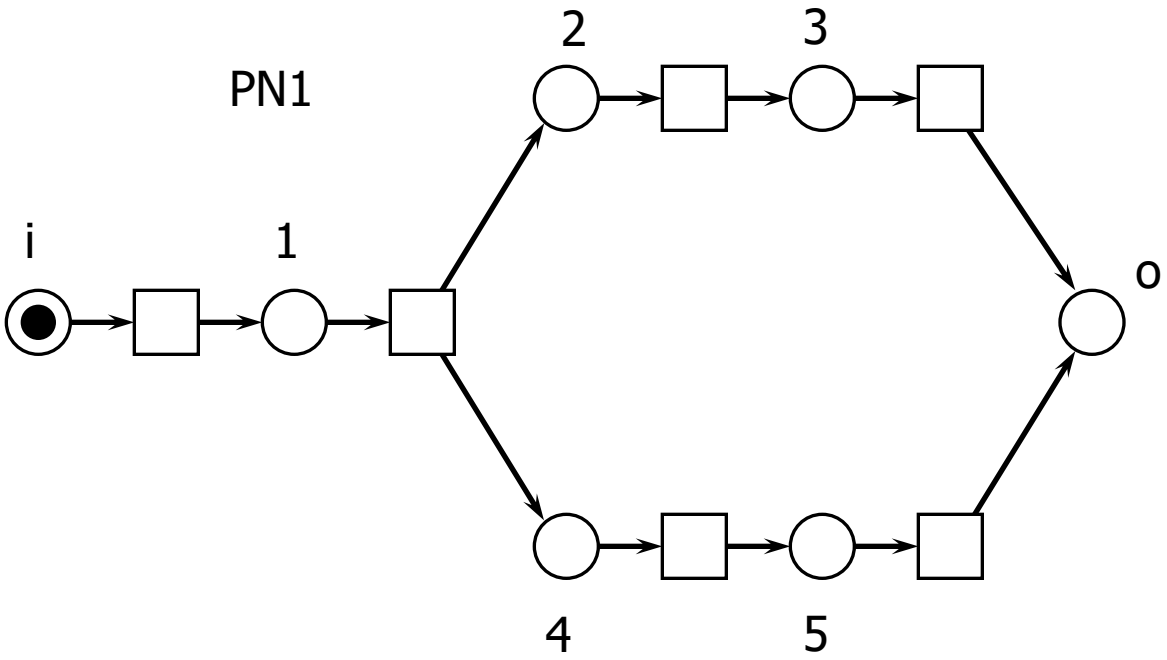
- Check soundness based on the reachability graph (RG)

- Procedure
  1. Check whether from each node in the RG, there is a path to state *o*
  2. Check whether place *o* is marked only in state *o*
  3. Check whether each transition occurs in the RG

Soundness Examples (cont'd)

PN1 is not sound

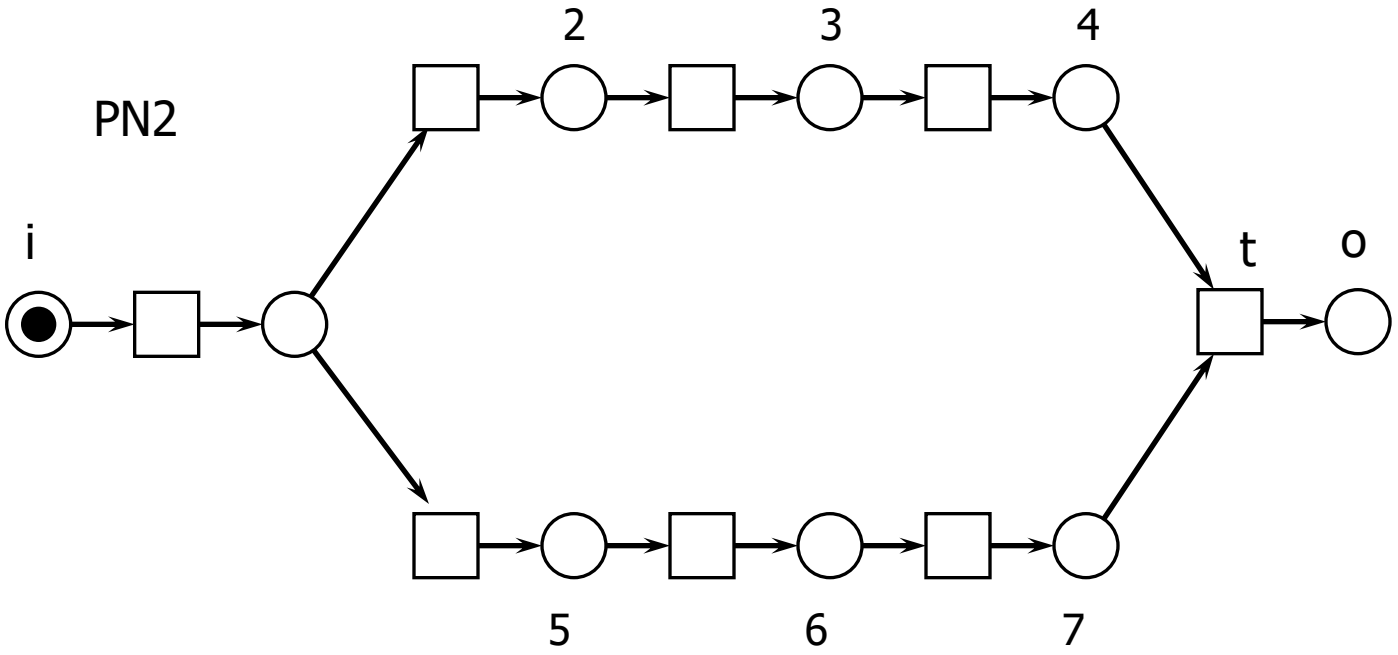
- Once there is a token in place o, there are other tokens remaining in the net
- For instance, reaching state [p4,o] violates requirement (2)



Soundness
Examples (cont'd)

PN2 is not sound

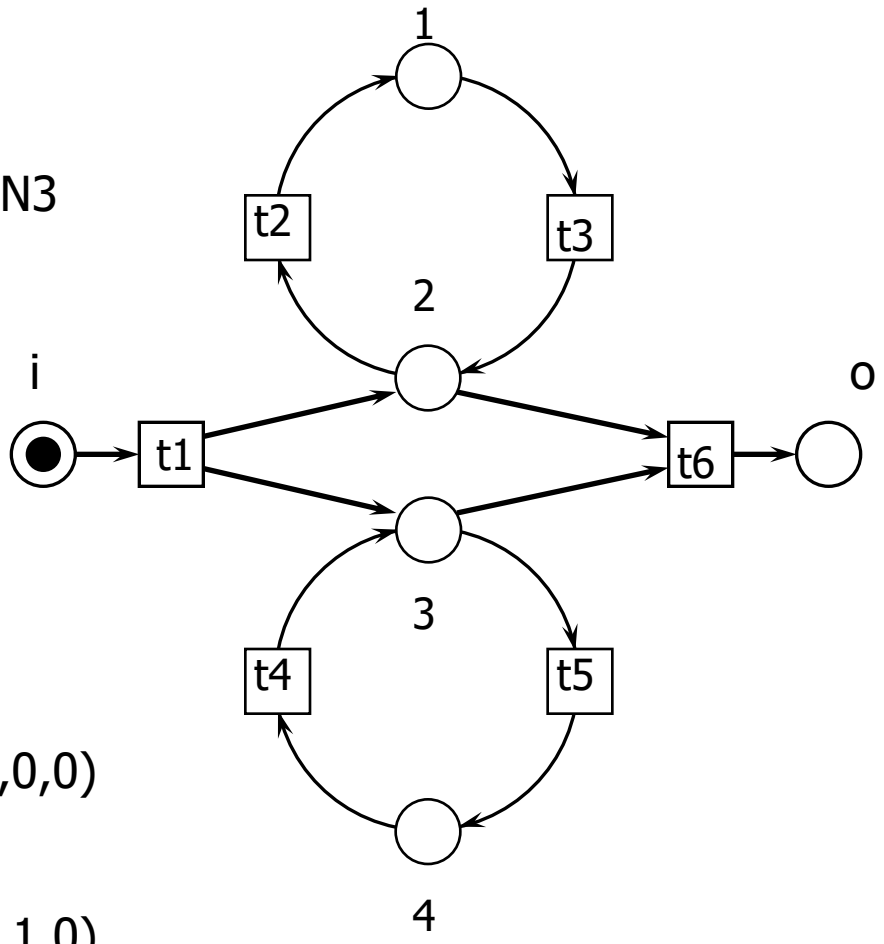
- No termination
- State [p4] is a deadlock and violates requirement (1)
- Transition t can never fire, which violates requirement (3)

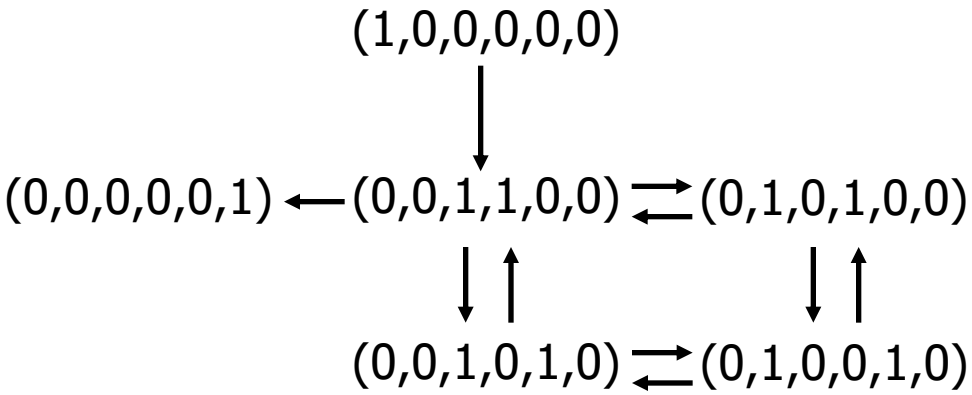


Soundness Examples (cont'd)

PN3 is sound

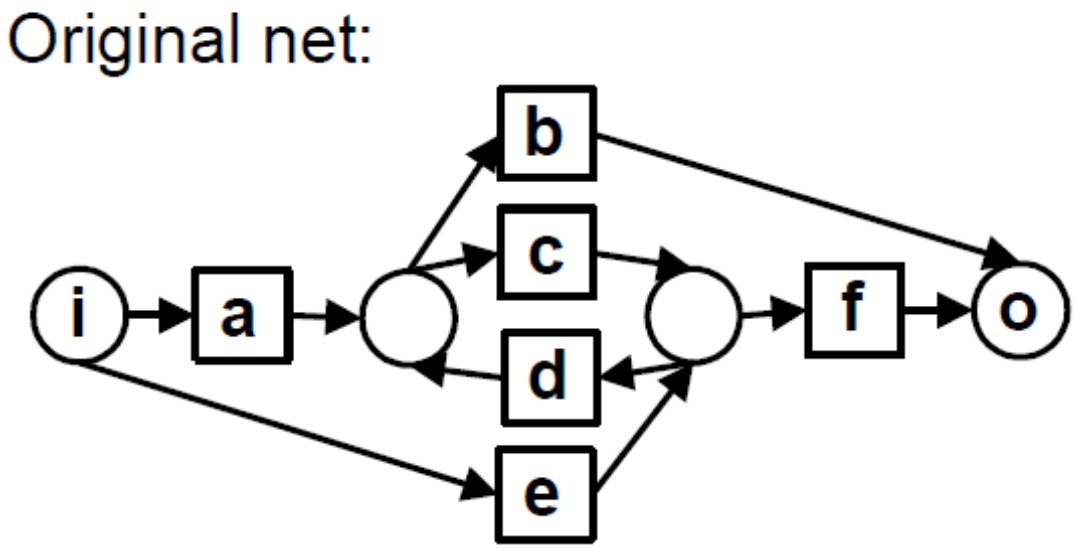
- All requirements are satisfied
- Prove based on RG





What can be rediscoverd ?

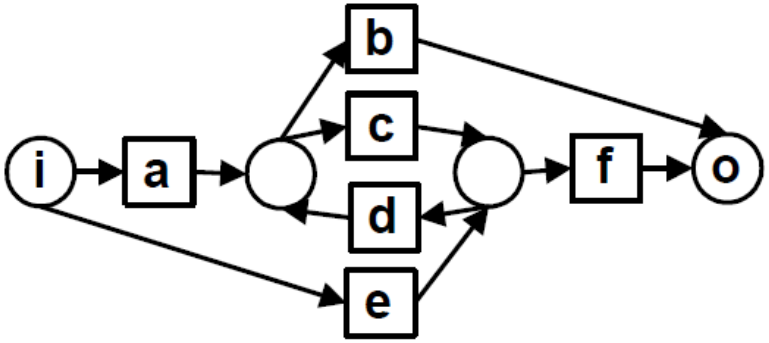
- Consider Loops



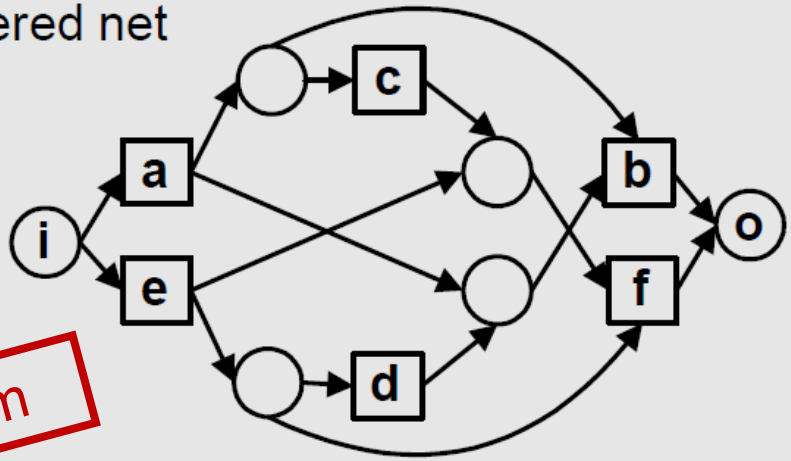
What can be rediscoverd ?

- Short loops are a problem:

Original net:



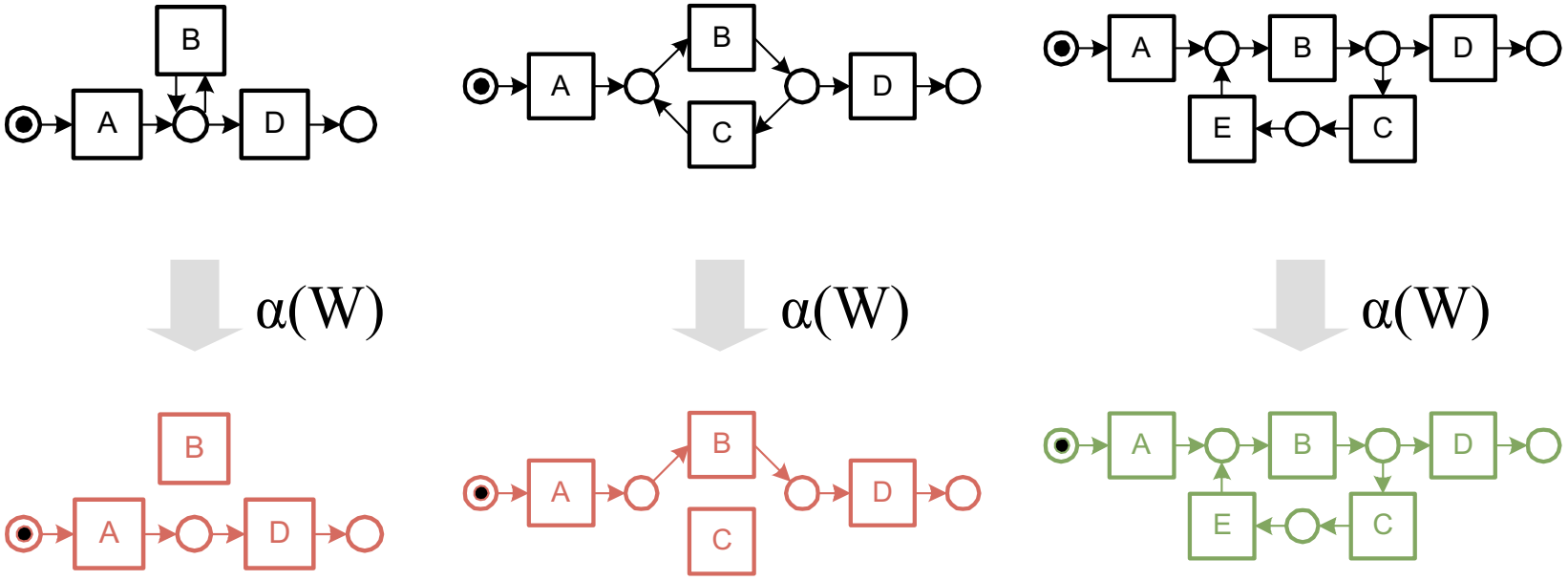
Discovered net



α-algorithm

Short Loops

- Short loops (length 1 or 2) pose an issue for rediscoverability by the α-algorithm:

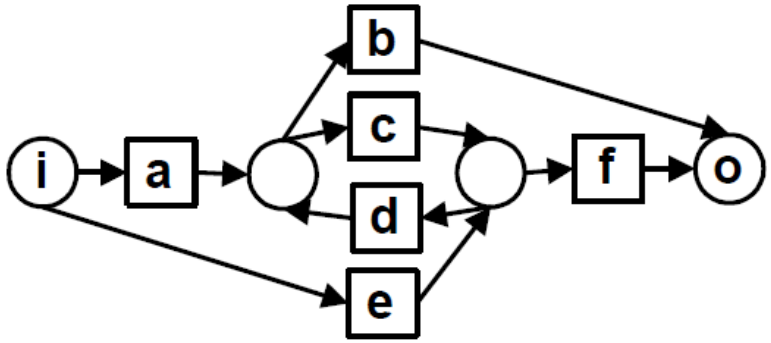


What can be rediscovered ?

- There is a proof for: The α-algorithm can rediscover a sound SWF-net without short loops, if the event log is complete according to introduced notion (based on direct successorship)

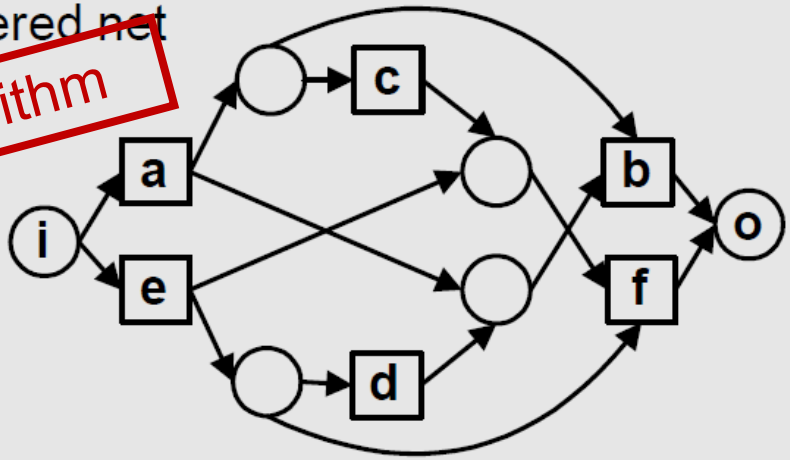
- Next:
  - Extend α-algorithm, so that sound SWF-nets with short loops can be rediscovered
  - This yields the α+-algorithm

Original net:



Discovered net

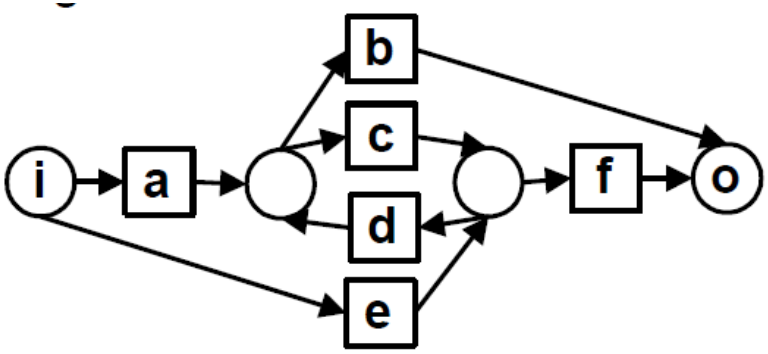
α-algorithm



α+ - Algorithm

- Loops of length 2
  - Issue: concurrent execution of activities and their execution in a loop cannot be distinguished
  - Approach: adapt log completeness criterion
  - Completeness so far: all direct successorships *xy* of transitions are observed at least once
  - Loop completeness: all successorship triples of transitions *xyx* are observed at least once

- Example for WF-net system
  - *…cdc…*
  - *…dcd…*



α+ - Algorithm

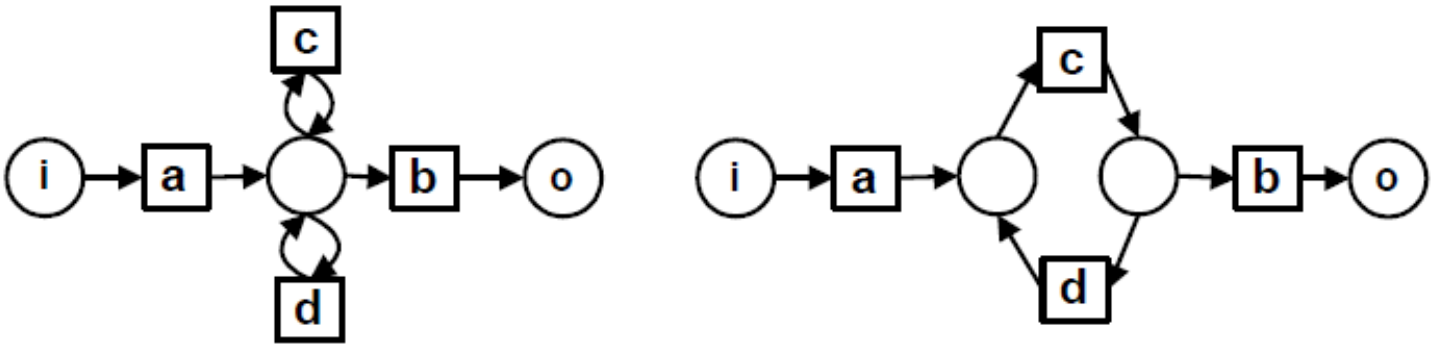
- Redefine ordering relations
  - Direct successor $a >_{\mathbf{w}} b$ and exclusiveness $a \#_{\mathbf{w}} b$ as before
  - Causality
    $a \rightarrow_{\mathbf{w}} b$ iff $a >_{w} b$ and (not $b >_{w} a$ or there exists a trace containing *aba*)
  - Concurrency
    $a \parallel_{\mathbf{w}} b$ iff $a >_{w} b$ and $b >_{w} a$ and there exists no trace containing *aba*

- Hence: loops of length 2 and concurrency can be distinguished

α+ - Algorithm

- Issue: Loops of length 2 and 1 cannot be distinguished

- Both nets produce
  - *…cdc…*
  - *…cdcd…*



α+ - Algorithm

- Solution to cope with loops of length 1
  - Pre-processing: identify transitions in loop of length 1 by sequence …*aa*…
  - Remove all these transitions from all observed sequences
  - Mining: Proceed as before with redefined ordering relations and the assumption of loop completeness of the log
  - Post-processing: insert loops of length 1 into constructed net
  - Find place *p* to insert loop for transition *a*
    - Find transitions, which are direct predecessors of *a*, but not direct successors of *a*
    - Find transitions, which are direct successors of *a*, but not direct predecessors of *a*
    - Input/output places of these transitions indicate where the loop has to be inserted

α+ - Algorithm

Log:

- ab, acdcb, acccb, adcddb, adddb

After pre-processing:

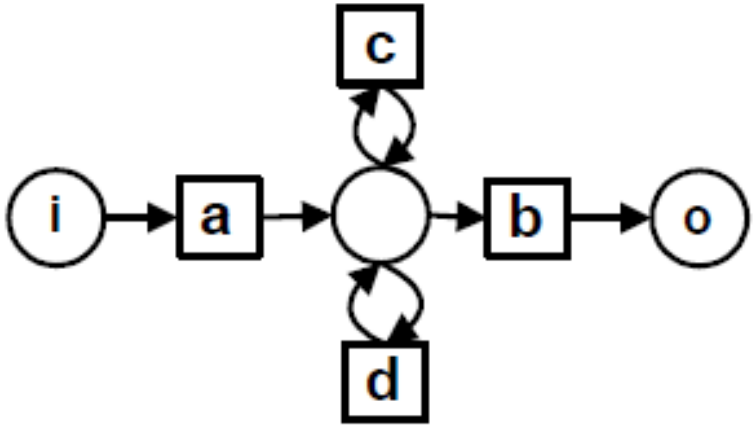
- ab, ab, ab, ab, ab
- Short loops of length 1 for c and d

Ordering relations:

- a → b

Net constructed by presented algorithm

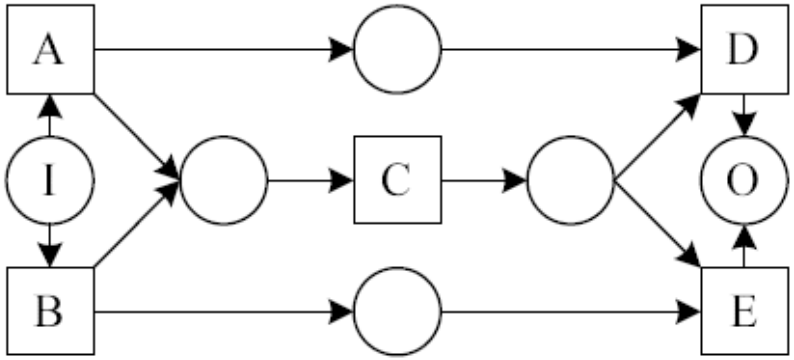
Net after post-processing

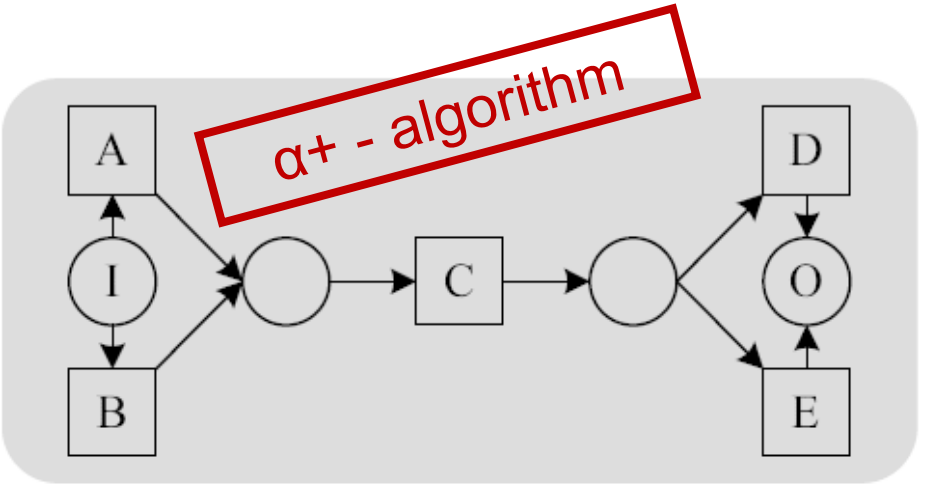


α++ - Algorithm

- α+ - algorithm fails to rediscover indirect dependencies imposed by non-free choice constructs (excluded by the notion of SWF-nets)

- Example:
  - Dependencies between A and D, and B and E
  - But: …AD… and …BE… are never observed to discover these dependencies
  - Complete log: ACD, BCE

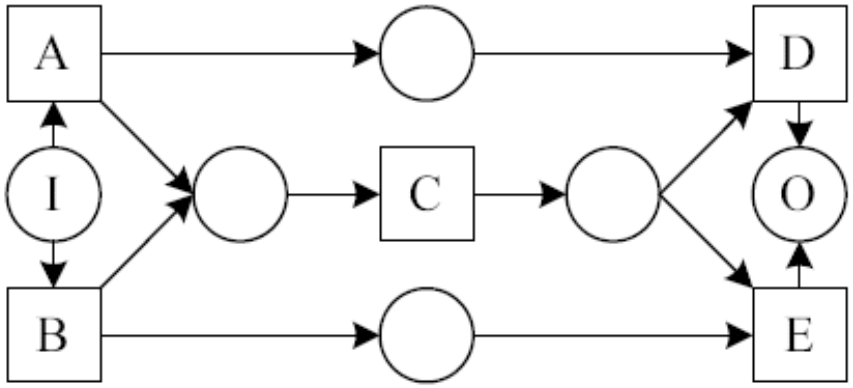




α++ - Algorithm

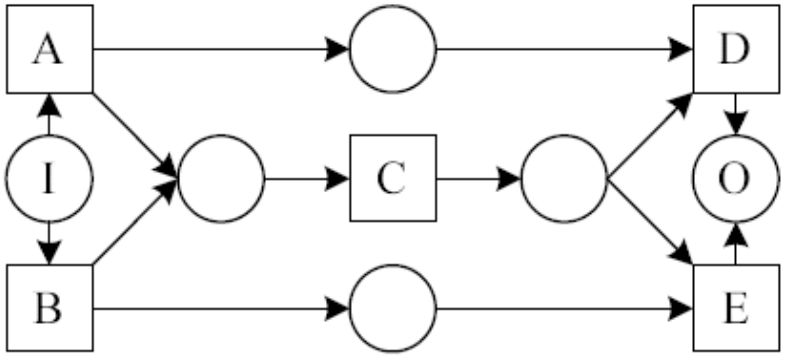
- Direct vs. indirect dependencies:
  - So far, consider only direct dependency in terms of successorship relation of transitions a and b
    - Yields: output place of a is input place of b

  - Next: consider indirect dependency between transitions a and b
    - Output place of a is input place of b
    - No reachable marking, such that firing of a in this marking enables b
    - Reachable marking for which holds: firing of a *can lead to marking*, in which b is enabled



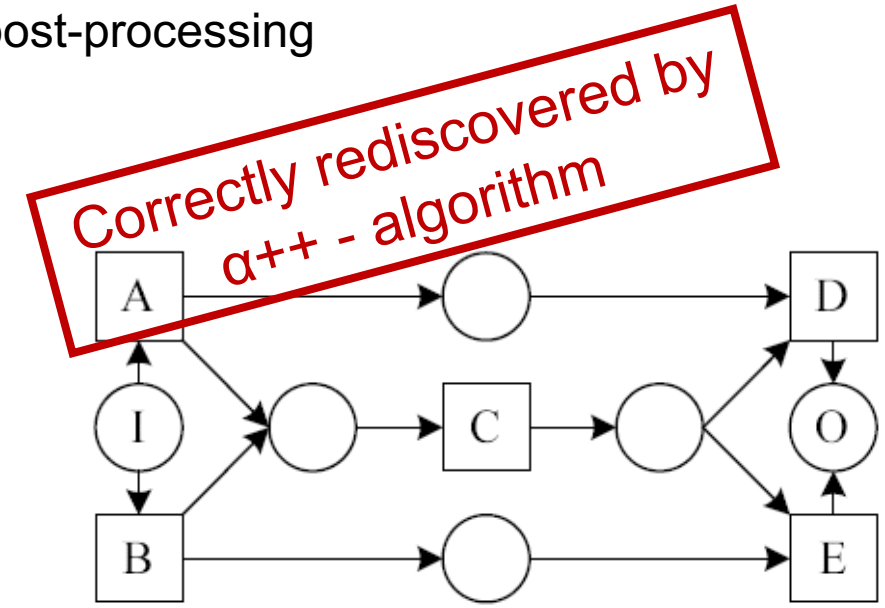
α++ - Algorithm

- Additional behavioural relations
  - XOR-split relation
    a <| b iff a # b and there exists a c such that c → a and c → b
  - XOR-join relation
    a |> b iff a # b and there exists a c such that a → c and b → c
  - Indirect successor
    a >> b iff not a > b and there exists a reachable firing sequence in which a happens before b, such that there is no XOR-split or -join between a and b
- Example
  - Complete log: ACD, BCE
  - A |> B, D <| E, A >> D, B >> E



α++ - Algorithm

- Approach
  - Based on additional relations, identify implicit dependencies
  - These implicit dependencies lead to additional places and flow arcs in the net system
  - These places may redundant in the sense that they do not change the behaviour of the net system
  - If so, they are removed as part of post-processing
- Result
  - Many non-free-choice constructs are discovered
  - But: no formal result on completeness for this net class



Take Away

Process Discovery produces models that can be evaluated through quality metrics: fitness, precision, generalization and simplicity

Family of α-algorithms for discovering certain Petri net subclasses

Completeness, rediscoverability, concurrency detection are key

α-algorithms do not consider noise, making them not practical often