Process Mining

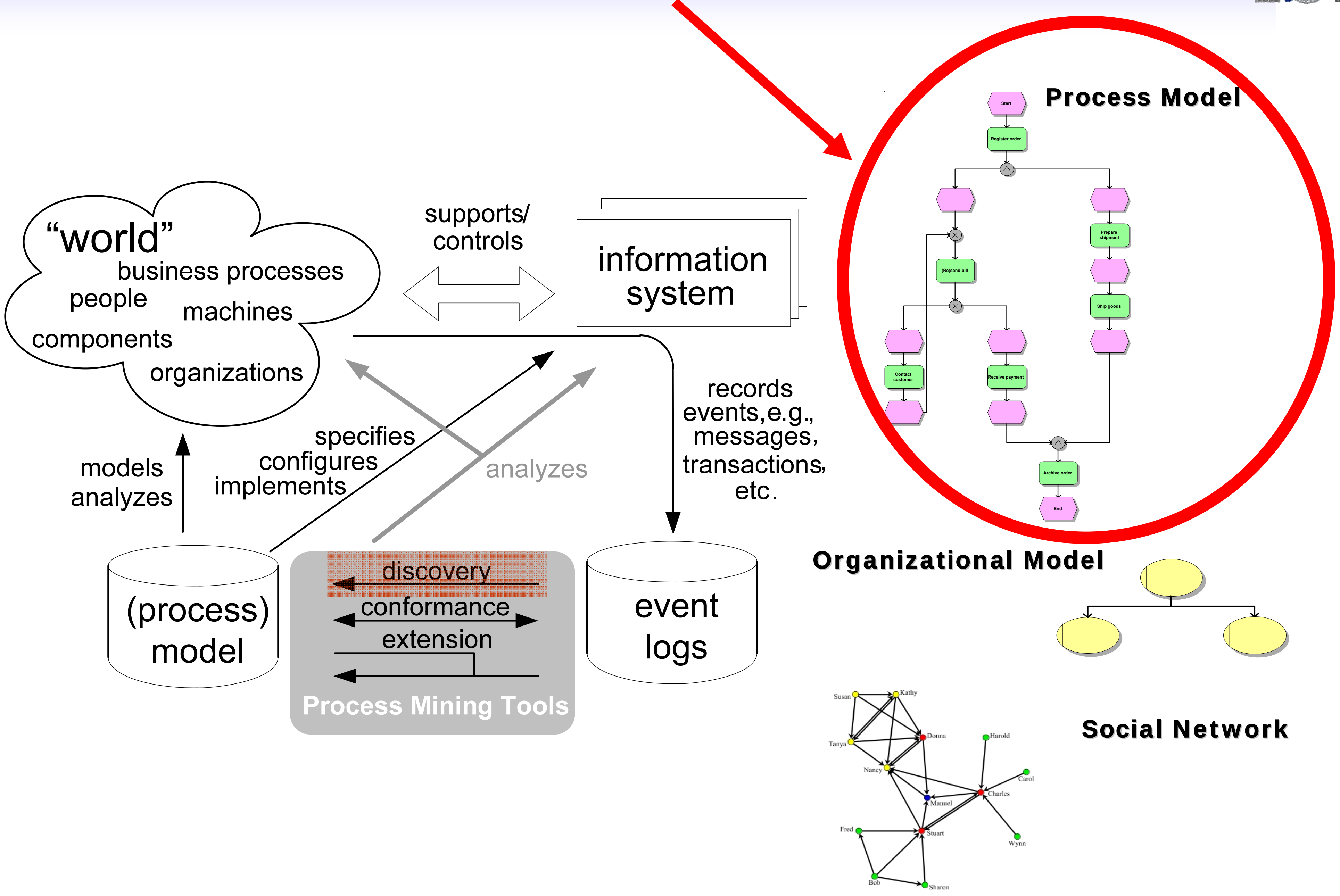
Part II – Workflow discovery algorithms

Induction of Control-Flow Graphs
α-algorithm
Heuristic Miner
Fuzzy Miner

Outline

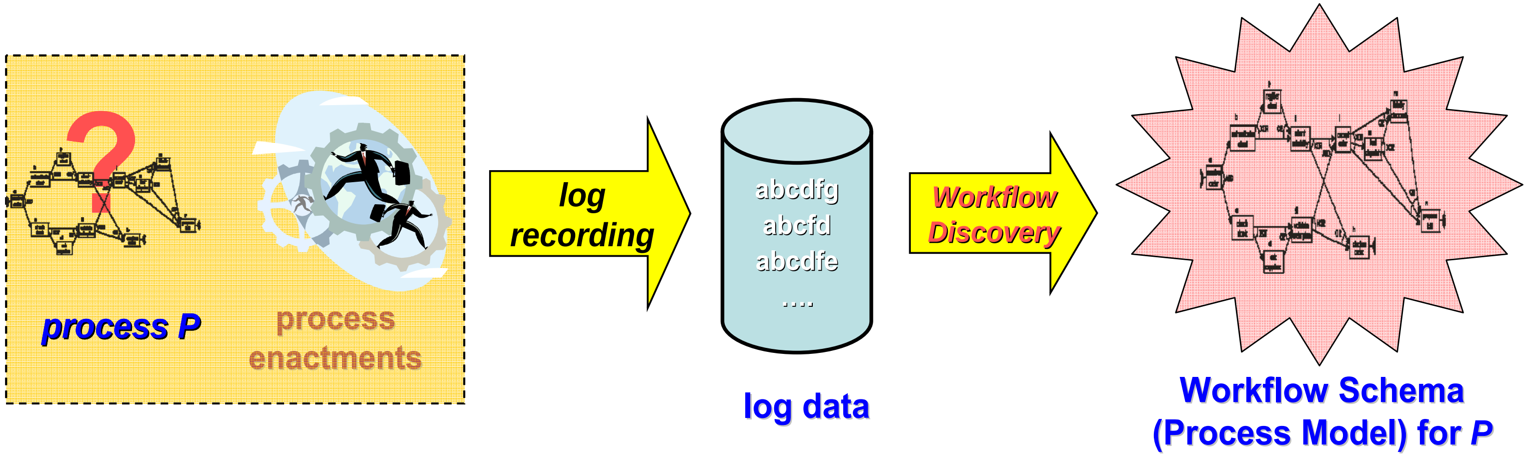
- Part I – Introduction to Process Mining
  - Context, motivation and goal
  - General characteristics of the analyzed processes and logs
  - Classification of Process Mining approaches
- Part II – Workflow discovery
  - Induction of basic Control Flow graphs
  - Other approaches (α-algorithm, Heuristic Miner, Fuzzy mining)
- Part III – Beyond control-flow mining
  - Organizational mining
  - Social net mining
  - Extension algorithms
- Part IV – Evaluation and validation of discovered models
  - Conformance Check
  - Log-based property verification
- Part V – Clustering-based Process Mining
  - Discovery of hierarchical workflow models
  - Discovery of process taxonomies
  - Outlier detection

Control-flow discovery



Workflow (control flow) discovery

- Input: execution data of a process *P* (possibly unknown)
  - log: a list of traces
  - In the simplest case each trace just registers the sequence of tasks performed during one execution of *P*
- Output: a schema for process P
  - captures the P's behavior, by representing all the ways its tasks are executed



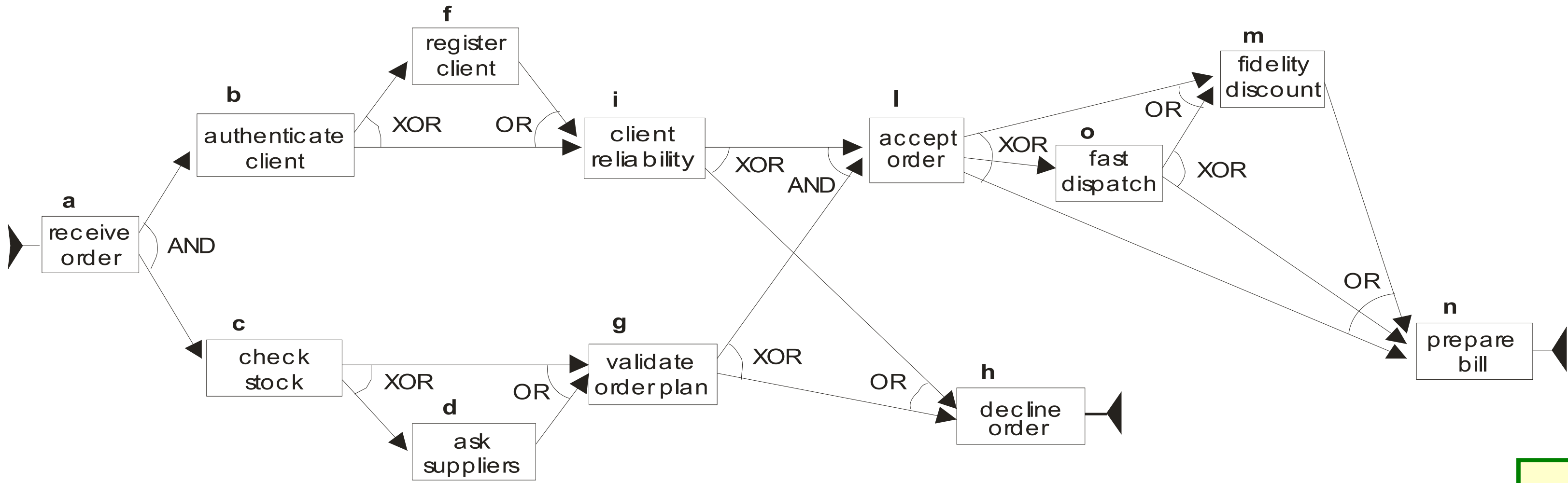
- Usefulness of mined models
  - Help better comprehend process behavior
  - Support process (re)-design (What is the process?)
  - Delta analysis (Are we doing what was specified?)
  - Process Design is often a complex and time consuming task
  - Sometimes, a fully-specified model is not available for the process

Representation of mined models

- A plethora of meta-models for representing workflow models
  - Block-structured languages, Petri Nets, Logics, Process Algebra,…
  - Graph-based languages are a reasonable choice w.r.t. expressiveness, complexity and comprehensibility
    - Most approaches derive some kind of graph over the tasks
    - Few exceptions use alternative techniques (e.g., grammar induction, term rewriting)
- A simple formalism: Control Flow Graph
  - Intuitively specifies which execution flows are allowed across the tasks
    - A labeled, directed graph
    - Each node corresponds to a task (and vice-versa)
    - Each arc represents a (temporal) precedence between two tasks
  - Cardinality constraints further (locally) restricts the possible execution flows

Control Flow Graph (CFG) models

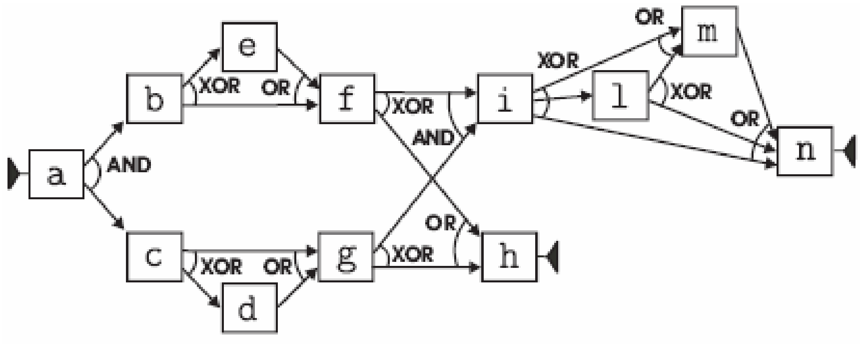
- A CFG schema *W* for *P* is a tuple <A, E, a_0, A_F, *Fork, Join*> where:
  - A is a finite set of *activities* (also nodes or tasks);
  - $E \subseteq (A - A_F) \times (A - \{a_0\})$ is an acyclic relation of precedence among activities;
  - $a_0 \in A$ is the starting activity, $A_F \subseteq A$ is the set of final activities;
  - Local constraints are expressed through the functions
    - ***Fork***: $(A - A_F)\alpha\{AND, OR, XOR\}$ and
    - ***Join:*** $(A - \{a_0\})\alpha\{AND, OR\}$
- Example: a CFG for the toy process *Order Management*



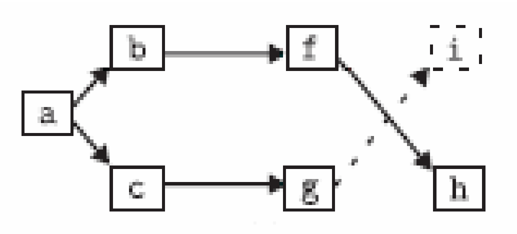
abficgln,
acbidpegln
abficdgh

CFG models: executions

schema ***W***



- **Instance** of W :
  - Connected sub-graph of S's CFG, containing at least the starting activity and one final activity, compliant with the constraints



- **Trace** of the process *P*:
  - A sequence of *P*'s tasks
- A trace *s* is **compliant** with the schema *W* if there is at least an instance *Iw* of W such that *s* is a topological order of *Iw*
  - Es: the trace *abfcgh* is compliant with the instance, while the traces *afbcgh* and *afblm* are not

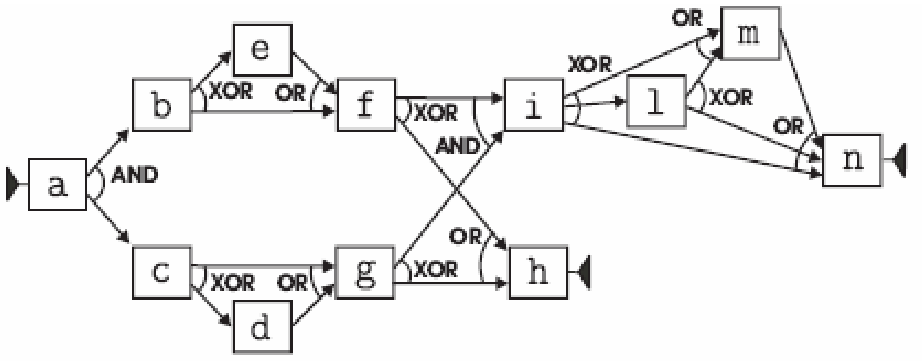
Conformance of a CFG schema w.r.t. a log

Two criteria to compare a (mined) model *W* with a given log *L:*

- Completeness:
  - the percentage of traces in the log that are compliant with *W* – the larger the more complete
- Soundness:
  - the percentage of traces that can be generated from *W* that actually occur in the log – the larger the sounder.

CFG conformance: Example

Schema W



Admitted Instances = 20;

Modeled Traces = 276.

soundness({W, L})=16/276

=5.797%

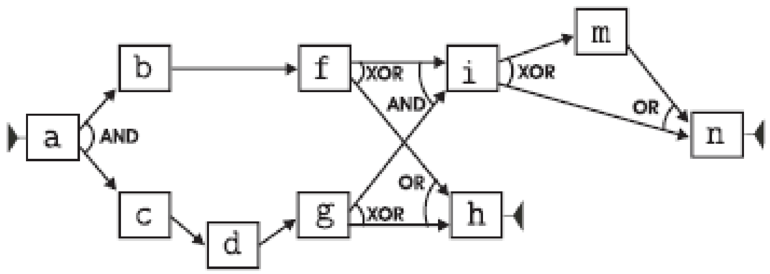
Log L

s_1 : acbgfh	s_9 : abefcgin
s_2 : abfcgh	s_{10} : acgbefiln
s_3 : acgbfh	s_{11} : abcedfgin
s_4 : abcgfin	s_{12} : acdbefgin
s_5 : abfcgimn	s_{13} : abcfdgimn
s_6 : acbfgiln	s_{14} : acdbfgimn
s_7 : acbgfilmn	s_{15} : abcdgfimn
s_8 : abcegfiln	s_{16} : acbfdgin

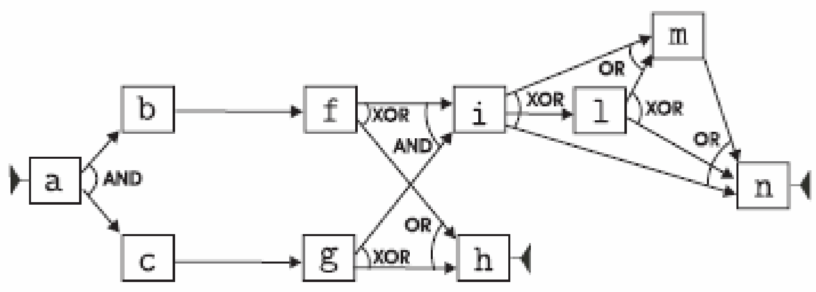
completeness({W, L})=16/16

=100%

Example: a way to get higher soundness



W_1



W_2

Modeled Traces = 64

Modeled Traces = 33

Considered trace Log (L)

s_1 : acbgfh	s_9 : abefcgin
s_2 : abfcgh	s_{10} : acgbefiln
s_3 : acgbfh	s_{11} : abcedfgin
s_4 : abcgfin	s_{12} : acdbefgin
s_5 : abfcgimn	s_{13} : abcfdgimn
s_6 : acbfgiln	s_{14} : acdbfgimn
s_7 : acbgfilmn	s_{15} : abcdgfimn
s_8 : abcegfiln	s_{16} : acbfdgin

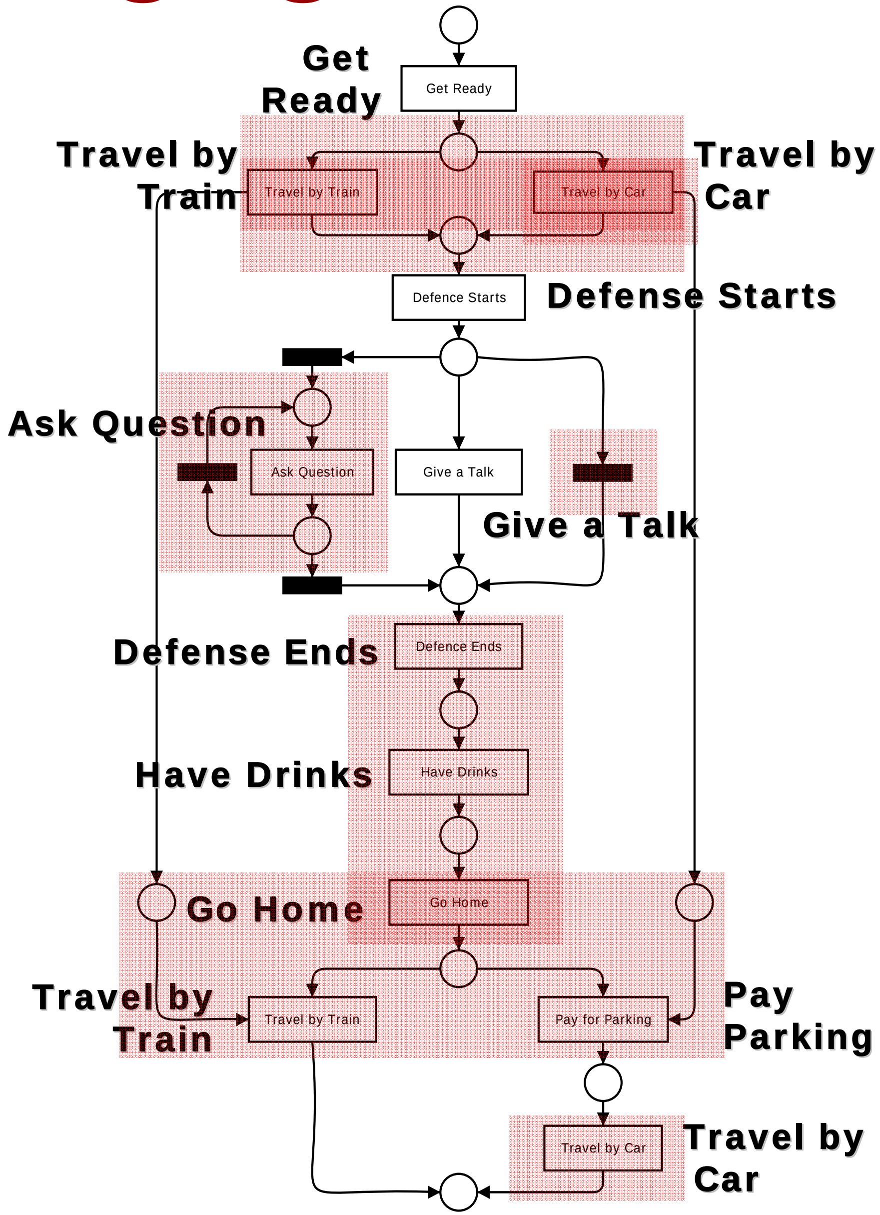
$s_{8,\ldots,12}$ comply with $W_1 \cup W_2$

soundness$(W_1 \cup W_2, L) = 11/97 = 11.34\%$

completeness$(W_1 \cup W_2, L) = 11/16 = 68.75\%$

Other representation languages: Petri nets

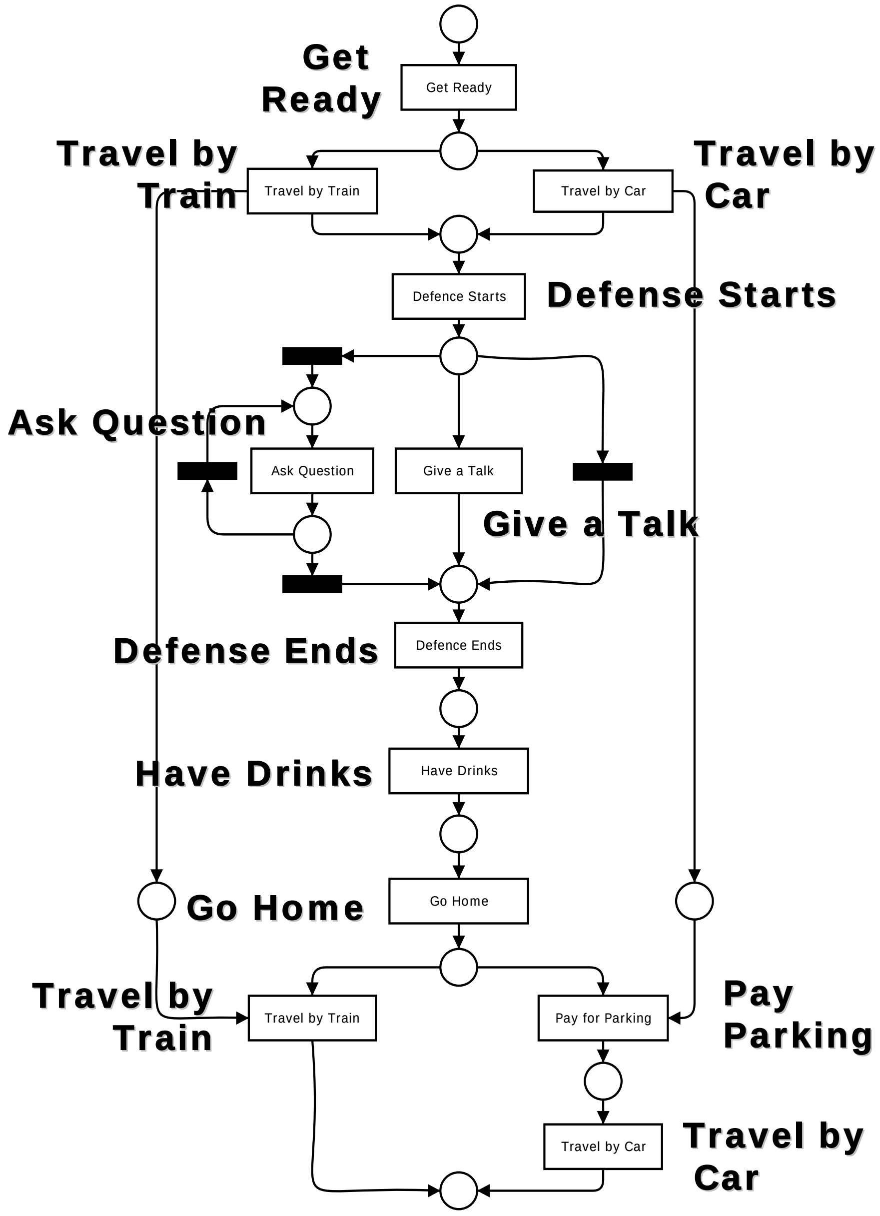
- Sequence
- Splits
- Joins
- Loops
- Non-Free Choice
- Invisible Tasks
- Duplicate Tasks



Other representation languages: Petri nets

- Sequence
- Splits
- Joins
- Loops
- Non-Free Choice
- Invisible Tasks
- Duplicate Tasks

+ noise in logs!



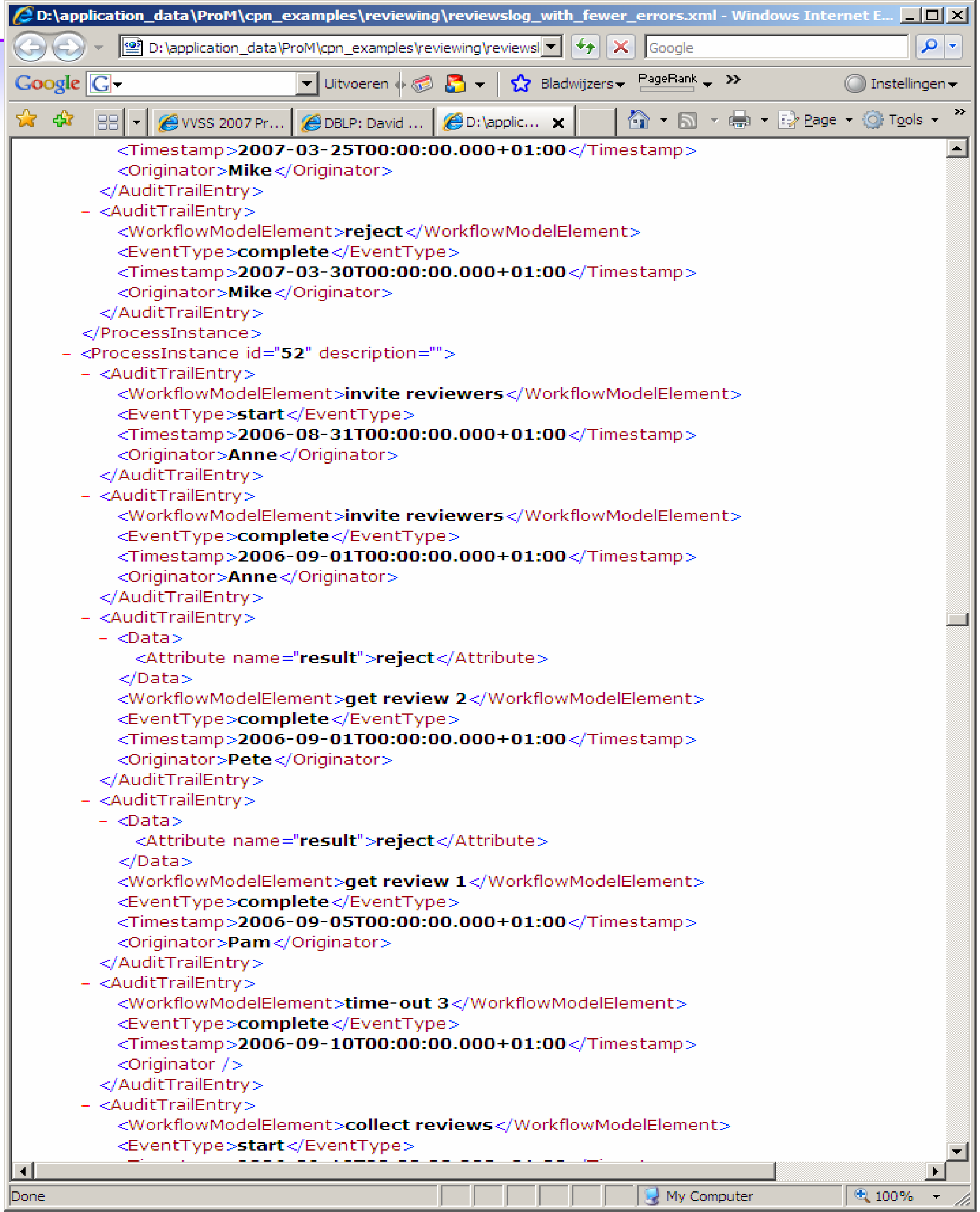
Toy example: paper reviewing

Event log:

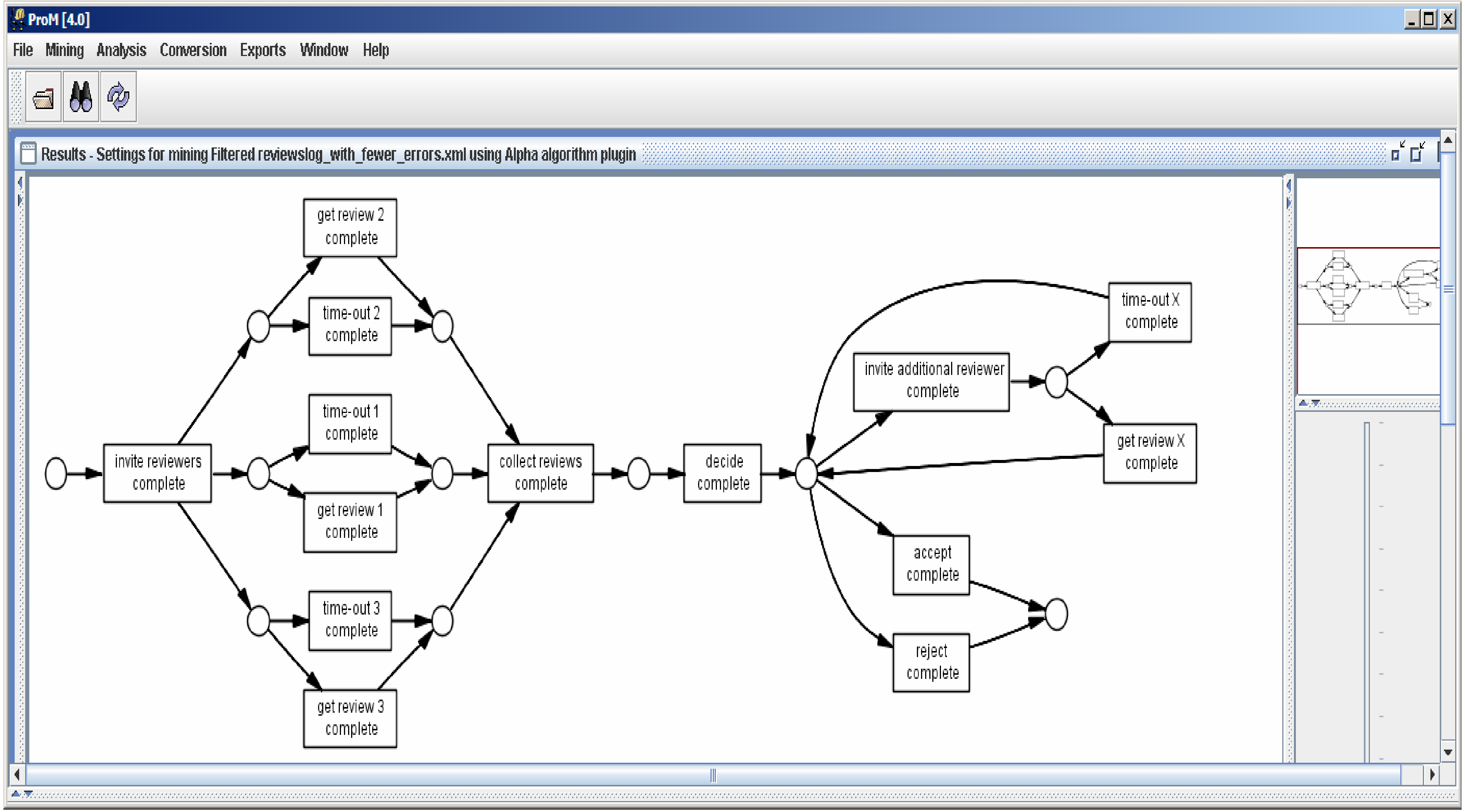
- processes
  - process instances
    - events

Per event:

- activity name
- (event type)
- (originator)
- (timestamp)
- (data)



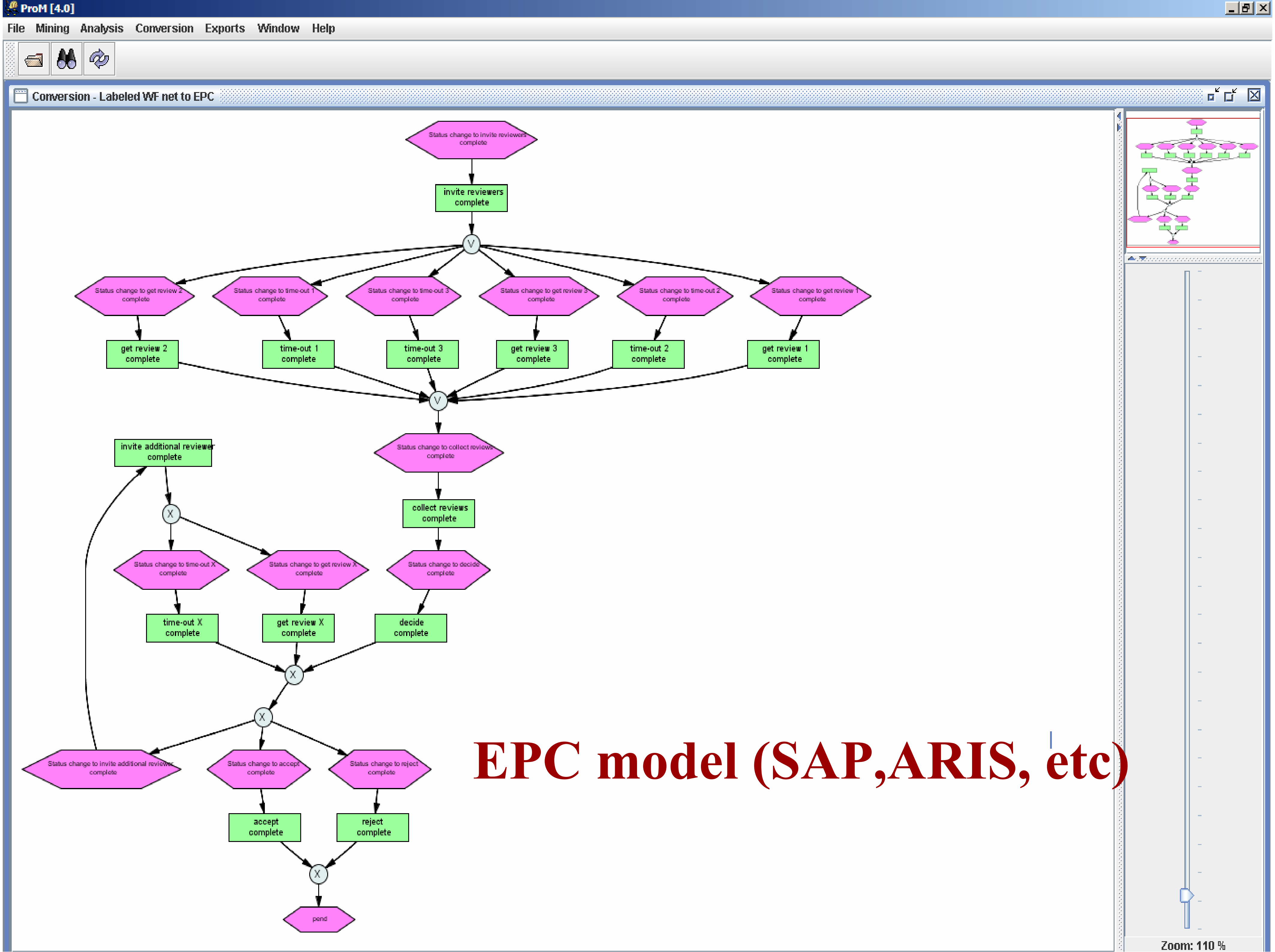
A discovered Petri net model (α-algorithm)

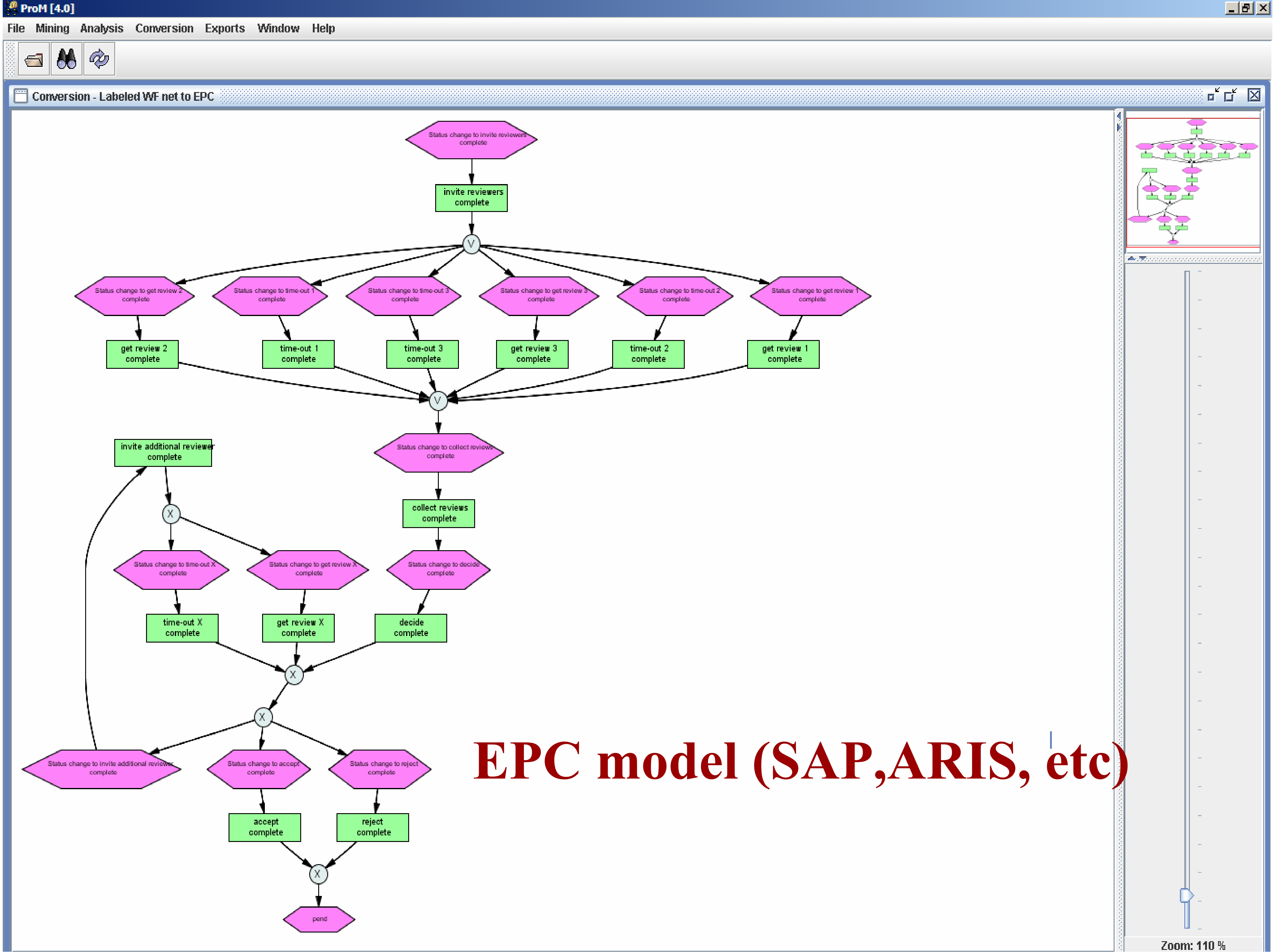


Other workflow languages: EPCs

- EPC= Event Driven Process Chain
- An EPC consists of three kinds of elements, which define the flow of a business process as a chain of events.
  - **Functions**: A function corresponds to an activity (task, process step) which needs to be executed.
  - **Events**: Events describe the situation before and/or after a function is executed.
  - **Connectors**: There are three types of connectors: ^ (and), X (xor) and V (or).
- Functions, events and connectors can be connected with edges in such a way that the following rules apply:
  - Events have at most one incoming edge and at most one outgoing edge.
  - Functions have precisely one incoming edge and precisely one outgoing edge.
  - Connectors have either one incoming edge and multiple outgoing edges, or multiple incoming edges and one outgoing edge.
  - In every path, functions and events alternate.
    - No two functions are connected and no two events are connected, not even when there are connectors in between.

EPC model (SAP,ARIS, etc)





EPC model (SAP,ARIS, etc)

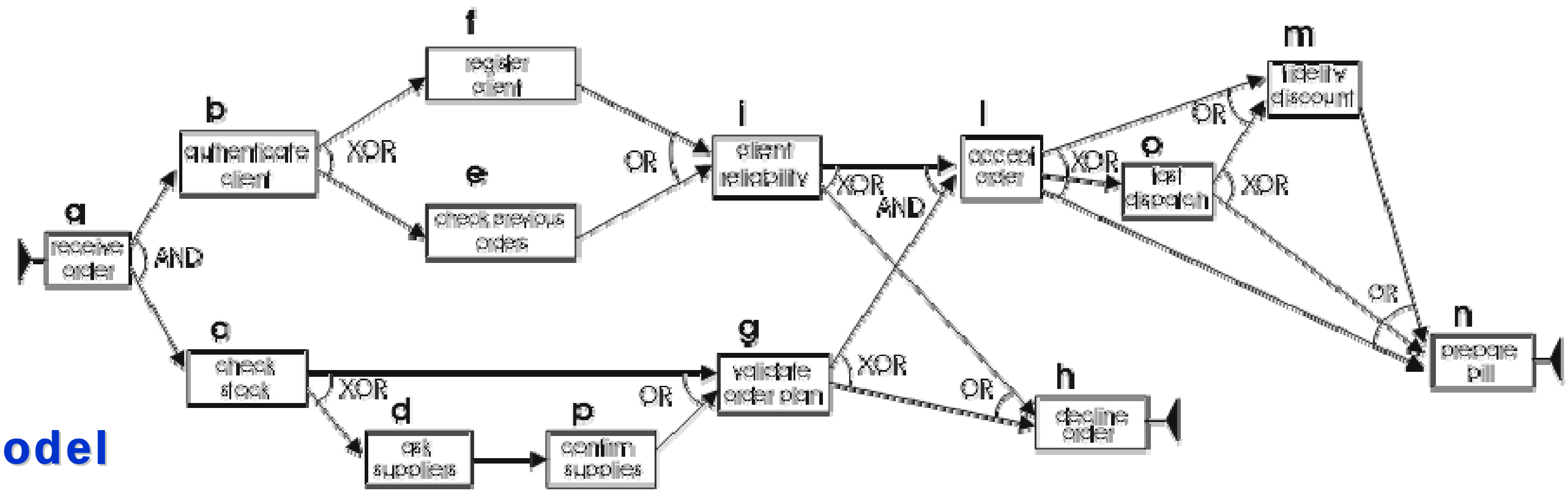
Workflow discovery algorithms: the case of CFG models

s_1: *acdbfgih*	s_5: *abicglmn*	s_9: *abficgln*	s_{13}: *abcidglmn*
s_2: *abficdgh*	s_6: *acbiglon*	s_{10}: *acgbfilon*	s_{14}: *acdbiglmn*
s_3: *acgbfih*	s_7: *acbgilomn*	s_{11}: *abcfdigln*	s_{15}: *abcdgilmn*
s_4: *abcgiln*	s_8: *abcfgilon*	s_{12}: *acdbfigln*	s_{16}: *acbidgln*

Event Log

Basic induction scheme

1. Mine a **Dependency Graph** encoding a minimal set of precedence links
2. Mine a set of cardinality (local) constraints, based on simple statistics
3. Introduce support thresholds to handle noisy data



Mined Model

Dependency graph

- Dependency graph for a log L is a graph D_L=<*A*,*E*> such that

  E={ (a, b) | ∃s∈L, i∈{1,..., length(s)-1} s.t. a=s[i] ∧ b=s[i+1]};

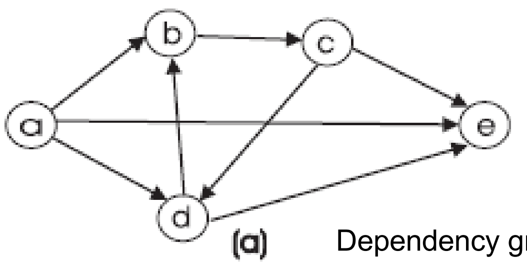
- Parallel activities

  Two activities *a* and *b* are parallel in *L*, if they occur in some cycle of D_L

- Precedence

  The activity *a* precedes *b in L*, denoted with $a \rightarrow b$, if *a* and *b* are not parallel and there is a path from *a* to *b* in D_L

Example: Log *L={abcde, adbce, ae}*



Dependency graph

- *a*, *b* and *c* are parallel activities in L;
- $a \rightarrow b$;
- $b \rightarrow e$;

Basic Workflow Discovery scheme

Input: A log $\mathcal{L}_P$.

Output: A workflow schema $\mathcal{WS} = \langle A, E, a_0, A_F, \mathit{Join}, \mathit{Fork} \rangle$.

Method: Perform the following steps:

$\langle A, E \rangle := D_{\mathcal{L}_P}$; *//nodes and edges are initially those of the dependency graph*
for each $(a, b) \in E$ s.t. a and b are *parallel* in $\mathcal{L}_P$ **do** *//remove cycles*
  $E := E - \{(a, b)\}$;
  for each $s \in \mathcal{L}_P$ s.t. $\{a, b\} \subseteq \mathit{tasks}(s)$ **do** *//update edges*
    $pre := s[i]$, where $s[i] \rightarrow a \wedge s[i] \rightarrow b$ and not exists $s[k]$ with $k > i$ s.t. $s[k] \rightarrow a \wedge s[k] \rightarrow b$;
    $E := E \cup \{(pre, a)\} \cup \{(pre, b)\}$;
    $post := s[j]$, where $a \rightarrow s[j] \wedge b \rightarrow s[j]$ and not exists $s[h]$ with $h < j$ s.t. $a \rightarrow s[h] \wedge b \rightarrow s[h]$;
    $E := E \cup \{(a, post)\} \cup \{(b, post)\}$;
  end for
end for
$a_0 := s[1]$, no matter of which trace $s \in \mathcal{L}_P$ is selected; $A_F := \{a \in A \mid \nexists b \in A \text{ s.t. } a \rightarrow b\}$;
for each $a \in A$ **do** *//construction of local constraints*
  if $\forall s \in \mathcal{L}_P$ s.t. $a \in \mathit{tasks}(s)$, it holds that $\forall c$ s.t. $(a, c) \in E$, $c \in \mathit{tasks}(s)$ **then** $\mathit{Fork}(a) =$ AND;
  else if $\forall s \in \mathcal{L}_P$ s.t. $a \in \mathit{tasks}(s)$, $|\{c \mid (a, c) \in E \wedge c \in \mathit{tasks}(s)\}| = 1$ **then** $\mathit{Fork}(a) =$ XOR;
  else $\mathit{Fork}(a) =$ OR;
  if $\forall s \in \mathcal{L}_P$ s.t. $a \in \mathit{tasks}(s)$, $(c, a) \in E \Rightarrow c \in \mathit{tasks}(s)$ **then** $\mathit{Join}(a) =$ AND;
  else $\mathit{Join}(a) =$ OR;
end for
return $\langle A, E, a_0, A_F, \mathit{Join}, \mathit{Fork} \rangle$;

Build the dependency graph And make it coincide with the initial CFG model

Removal of cycles

Remove, from *E*, all edgee between parallel activities

Connect the vertices of the the edge, with preceding nodes.

Connect the vertices of the removed edge with following nodes

Identification of the first node a_0 and the set of final nodes A_F.

Basic Workflow Discovery Scheme

Input: A log $\mathcal{L}_P$.

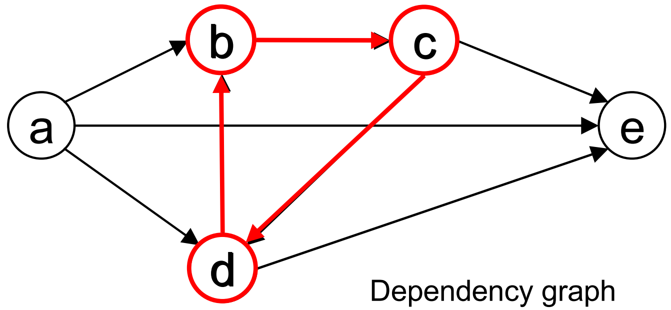
Output: A workflow schema $\mathcal{WS} = \langle A, E, a_0, A_F, \mathit{Join}, \mathit{Fork} \rangle$.

Method: Perform the following steps:

$\langle A, E \rangle := D_{\mathcal{L}_P}$; *//nodes and edges are initially those of the dependency graph*
for each $(a, b) \in E$ s.t. a and b are *parallel* in $\mathcal{L}_P$ **do** *//remove cycles*
  $E := E - \{(a, b)\}$;
  for each $s \in \mathcal{L}_P$ s.t. $\{a, b\} \subseteq \mathit{tasks}(s)$ **do** *//update edges*
    $\mathit{pre} := s[i]$, where $s[i] \to a \wedge s[i] \to b$ and not exists $s[k]$ with $k > i$ s.t. $s[k] \to a \wedge s[k] \to b$;
    $E := E \cup \{(\mathit{pre}, a)\} \cup \{(\mathit{pre}, b)\}$;
    $\mathit{post} := s[j]$, where $a \to s[j] \wedge b \to s[j]$ and not exists $s[h]$ with $h < j$ s.t. $a \to s[h] \wedge b \to s[h]$;
    $E := E \cup \{(a, \mathit{post})\} \cup \{(b, \mathit{post})\}$;
  end for
end for
$a_0 := s[1]$, no matter of which trace $s \in \mathcal{L}_P$ is selected; $A_F := \{a \in A \mid \nexists b \in A \text{ s.t. } a \to b\}$;
for each $a \in A$ **do** *//construction of local constraints*
  if $\forall s \in \mathcal{L}_P$ s.t. $a \in \mathit{tasks}(s)$, it holds that $\forall c$ s.t. $(a, c) \in E$, $c \in \mathit{tasks}(s)$ **then** $\mathit{Fork}(a) = \texttt{AND}$;
  else if $\forall s \in \mathcal{L}_P$ s.t. $a \in \mathit{tasks}(s)$, $|\{c \mid (a, c) \in E \wedge c \in \mathit{tasks}(s)\}| = 1$ **then** $\mathit{Fork}(a) = \texttt{XOR}$;
  else $\mathit{Fork}(a) = \texttt{OR}$;
  if $\forall s \in \mathcal{L}_P$ s.t. $a \in \mathit{tasks}(s)$, $(c, a) \in E \Rightarrow c \in \mathit{tasks}(s)$ **then** $\mathit{Join}(a) = \texttt{AND}$;
  else $\mathit{Join}(a) = \texttt{OR}$;
end for
return $\langle A, E, a_0, A_F, \mathit{Join}, \mathit{Fork} \rangle$;

Derive local constraints

Example: Algorithm simulation



Dependency graph

Precedences:

a→b	b→c	c→b	d→b
a→c	b→d	c→d	d→c
a→d	b→e	c→e	d→e
a→e			

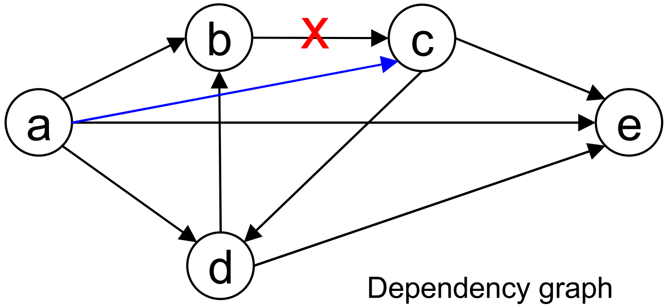
Parallel activities:

- b, c, d

Edges to remove:

- (b, c),
- (b, d),
- (c, d).

Example: Algorithm simulation



Dependency graph

Edges to remove: (b, c), (d, b), (c, d).

log *L* = {abcde, adbce, ae},

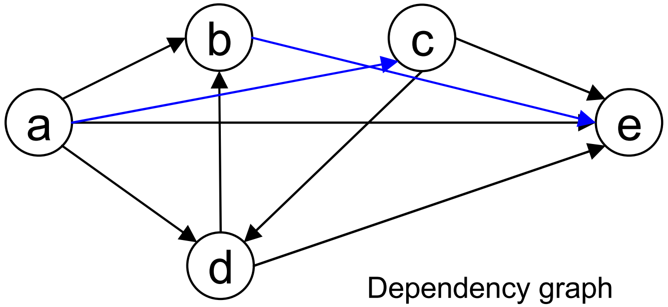
pre

a→b	b→c	c→b	d→b
a→c	b→d	c→d	d→c
a→d	b→e	c→e	d→e
a→e			

E:= {(a, b), (a, d), (a, e),
~~(b, c)~~, (c, d), (c, e), ∪ *{(a,c)};*
(d, b), (d, e)}

for each $(a, b) \in E$ s.t. a and b are *parallel* in $\mathcal{L}_P$ **do** *//remove cycles*
$E := E - \{(a, b)\}$;
for each $s \in \mathcal{L}_P$ s.t. $\{a, b\} \subseteq tasks(s)$ **do** *//update edges*
$pre := s[i]$, where $s[i] \rightarrow a \wedge s[i] \rightarrow b$ and not exists $s[k]$ with $k > i$ s.t. $s[k] \rightarrow a \wedge s[k] \rightarrow b$;
$E := E \cup \{(pre, a)\} \cup \{(pre, b)\}$;
$post := s[j]$, where $a \rightarrow s[j] \wedge b \rightarrow s[j]$ and not exists $s[h]$ with $h < j$ s.t. $a \rightarrow s[h] \wedge b \rightarrow s[h]$;
$E := E \cup \{(a, post)\} \cup \{(b, post)\}$;
end for
end for

Example: Algorithm simulation



Dependency graph

Edges to remove: (b, c), (d, b), (c, d).

log *L* = {abcde, adbce, ae},

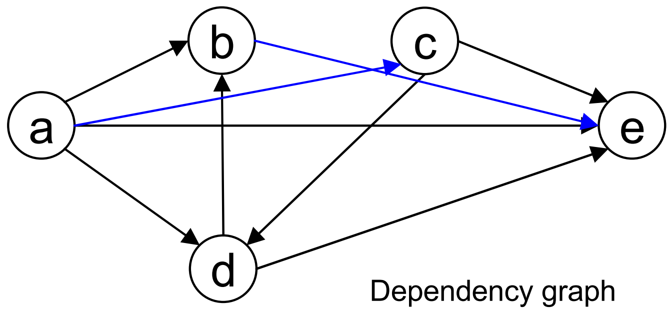
post

a→b	b→c	c→b	d→b
a→c	b→d	c→d	d→c
a→d	b→e	c→e	d→e
~~a→e~~			

E:= {(a, b), (a, d), (a, e),
~~(b, c)~~, (c, d), (c, e), $\cup$ {(a,c)} $\cup$ {(b,e)}
(d, b), (d, e)}

for each $s \in \mathcal{L}_P$ s.t. $\{a, b\} \subseteq tasks(s)$ **do** //*update edges*
  $pre := s[i]$, where $s[i] \to a \wedge s[i] \to b$ and not exists $s[k]$ with $k > i$ s.t. $s[k] \to a \wedge s[k] \to b$;
  $E := E \cup \{(pre, a)\} \cup \{(pre, b)\}$;
  $post := s[j]$, where $a \to s[j] \wedge b \to s[j]$ and not exists $s[h]$ with $h < j$ s.t. $a \to s[h] \wedge b \to s[h]$;
  $E := E \cup \{(a, post)\} \cup \{(b, post)\}$;
end for

Example: Algorithm simulation



Dependency graph

Edges to remove: (b, c), (d, b), (c, d).

log L = {abcde, adbce, ae},

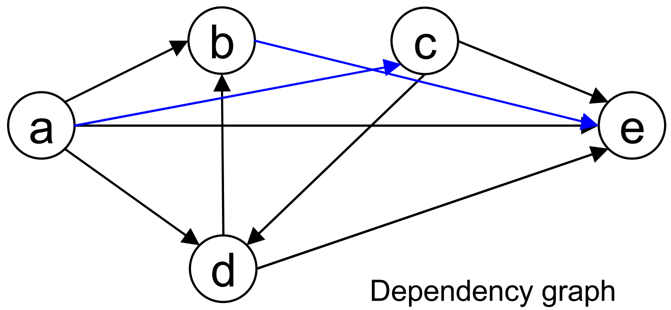
pre

a→b	b→c	c→b	d→b
a→c	b→d	c→d	d→c
a→d	b→e	c→e	d→e
a→e			

E:= {(a, b), (a, d), (a, e),
(~~b, c~~), (c, d), (c, e), $\cup$ *{(a,c)}* $\cup$ *{(b,e)}*
(d, b), (d, e)}

for each $s \in \mathcal{L}_P$ s.t. $\{a, b\} \subseteq tasks(s)$ **do** //*update edges*
  $pre := s[i]$, where $s[i] \rightarrow a \wedge s[i] \rightarrow b$ and not exists $s[k]$ with $k > i$ s.t. $s[k] \rightarrow a \wedge s[k] \rightarrow b$;
  $E := E \cup \{(pre, a)\} \cup \{(pre, b)\}$;
  $post := s[j]$, where $a \rightarrow s[j] \wedge b \rightarrow s[j]$ and not exists $s[h]$ with $h < j$ s.t. $a \rightarrow s[h] \wedge b \rightarrow s[h]$;
  $E := E \cup \{(a, post)\} \cup \{(b, post)\}$;
end for

Example: Algorithm simulation



Dependency graph

Edges to remove: (b, c), (d, b), (c, d).

log L = {abcde, adbce, ae},

post

a→b	b→c	c→b	d→b
a→c	b→d	c→d	d→c
a→d	b→e	c→e	d→e
a→e			

E:= {(a, b), (a, d), (a, e),
~~(b, c)~~, (c, d), (c, e), $\cup$ {(a,c)} $\cup$ {(b,e)}
(d, b), (d, e)}

for each $s \in \mathcal{L}_P$ s.t. $\{a, b\} \subseteq tasks(s)$ **do** *//update edges*

$pre := s[i]$, where $s[i] \rightarrow a \wedge s[i] \rightarrow b$ and not exists $s[k]$ with $k > i$ s.t. $s[k] \rightarrow a \wedge s[k] \rightarrow b$;

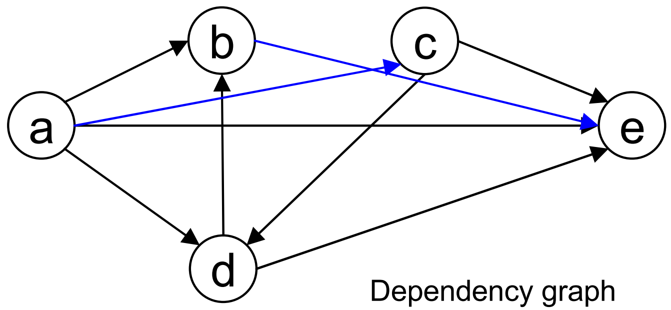
$E := E \cup \{(pre, a)\} \cup \{(pre, b)\}$;

$post := s[j]$, where $a \rightarrow s[j] \wedge b \rightarrow s[j]$ and not exists $s[h]$ with $h < j$ s.t. $a \rightarrow s[h] \wedge b \rightarrow s[h]$;

$E := E \cup \{(a, post)\} \cup \{(b, post)\}$;

end for

Example: Algorithm simulation



Dependency graph

Edges to remove: (b, c), (d, b), (c, d).

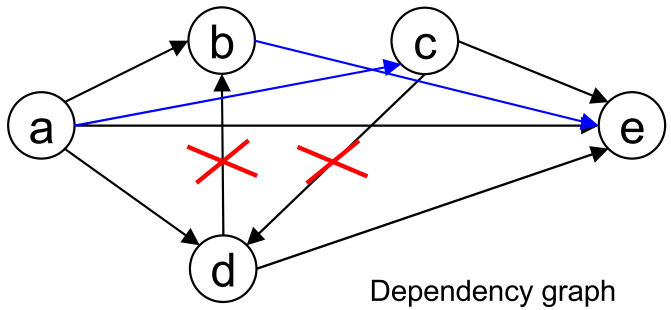
log L = {abcde, adbce, ae},

a→b	b→c	c→b	d→b
a→c	b→d	c→d	d→c
a→d	b→e	c→e	d→e
a→e			

E:= {(a, b), (a, d), (a, e),
(~~b, c~~), (c, d), (c, e), $\cup$ {(a,c)} $\cup$ {(b,e)}
(d, b), (d, e)}

```
for each s ∈ L_P s.t. {a, b} ⊆ tasks(s) do          //update edges
   pre := s[i], where s[i] → a ∧ s[i] → b and not exists s[k] with k > i s.t. s[k] → a ∧ s[k] → b;
   E := E ∪ {(pre, a)} ∪ {(pre, b)};
   post := s[j], where a → s[j] ∧ b → s[j] and not exists s[h] with h < j s.t. a → s[h] ∧ b → s[h];
   E := E ∪ {(a, post)} ∪ {(b, post)};
end for
```

Example: Algorithm simulation



Dependency graph

Edges to remove: (b, c), (d, b), (c, d).

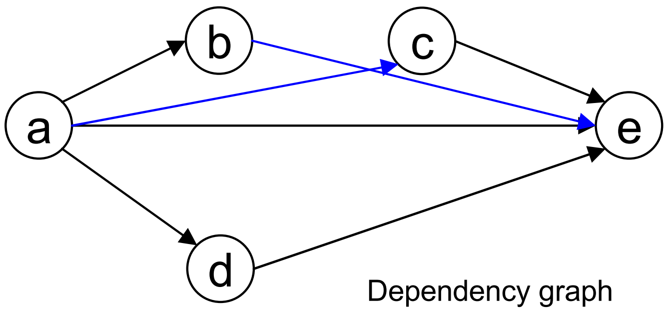
log L = {abcde, adbce, ae},

a→b	b→c	c→b	d→b
a→c	b→d	c→d	d→c
a→d	b→e	c→e	d→e
a→e			

E:= {(a, b), (a, d), (a, e),
(~~b, c~~), (c, d), (c, e), $\cup$ *{(a,c)}* $\cup$ *{(b,e)}*
(~~d, b~~), (~~d, e~~)}

```
for each (a, b) ∈ E s.t. a and b are parallel in L_P do        //remove cycles
  E := E − {(a, b)};
  for each s ∈ L_P s.t. {a, b} ⊆ tasks(s) do          //update edges
    pre := s[i], where s[i] → a ∧ s[i] → b and not exists s[k] with k > i s.t. s[k] → a ∧ s[k] → b;
    E := E ∪ {(pre, a)} ∪ {(pre, b)};
    post := s[j], where a → s[j] ∧ b → s[j] and not exists s[h] with h < j s.t. a → s[h] ∧ b → s[h];
    E := E ∪ {(a, post)} ∪ {(b, post)};
  end for
end for
```

Example: Algorithm simulation



Dependency graph

Edges to remove: (b, c), (d, b), (c, d).

log L = {abcde, adbce, ae},

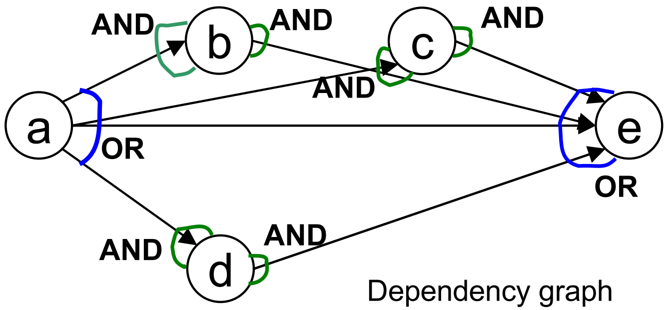
a→b	b→c	c→b	d→b
a→c	b→d	c→d	d→c
a→d	b→e	c→e	d→e
a→e			

E:= {(a, b), (a, d), (a, e),
(~~b, c~~), (c, d), (c, e), ∪ {(a,c)} ∪ {(b,e)}
(~~d, b~~), (~~d, e~~)}

- a_0:= a;
- A_F:= {e}

$a_0 := s[1]$, no matter of which trace $s \in \mathcal{L}_P$ is selected; $A_F := \{a \in A \mid \nexists b \in A \text{ s.t. } a \rightarrow b\}$;

Example: Algorithm simulation



Dependency graph

log L = {abcde, adbce, ae},

A = {a, b, c, d, e},

E:= {(a, b), (a, d), (a, e),
~~(b, c)~~, (c, d), (c, e), $\cup$ {(a,c)} $\cup$ {(b,e)}
~~(d, b)~~, ~~(d, e)~~}

- a_0:= a;
- A_F:= {e}

for each $a \in A$ **do** //*construction of local constraints*
  if $\forall s \in \mathcal{L}_P$ s.t. $a \in tasks(s)$, it holds that $\forall c$ s.t. $(a, c) \in E$, $c \in tasks(s)$ **then** $Fork(a) =$ AND;
  else if $\forall s \in \mathcal{L}_P$ s.t. $a \in tasks(s)$, $|\{c \mid (a, c) \in E \wedge c \in tasks(s)\}| = 1$ **then** $Fork(a) =$ XOR;
  else $Fork(a) =$ OR;
  if $\forall s \in \mathcal{L}_P$ s.t. $a \in tasks(s)$, $(c, a) \in E \Rightarrow c \in tasks(s)$ **then** $Join(a) =$ AND;
  else $Join(a) =$ OR;
end for

Outline

- Part I – Introduction to Process Mining
  - Context, motivation and goal
  - General characteristics of the analyzed processes and logs
  - Classification of Process Mining approaches
- Part II – Workflow discovery
  - Basic CFG induction algorithm
  - Other algorithms (α-algorithm, Heuristic Miner, Fuzzy mining)
- Part III – Beyond the control-flow mining perspective
  - Organizational mining
  - Social net mining
  - Extension algorithms
- Part IV – Evaluation and validation of discovered models
  - Conformance Check
  - Log-based property verification
- Part V – Clustering-based Process Mining
  - Discovery of hierarchical process models
  - Discovery of process taxonomies
  - Outlier detection

Workflow discovery algorithms

- Multi-phase PM
- α-algorithm
- Heuristics Miner
- Genetic PM
- Fuzzy Miner

Outline

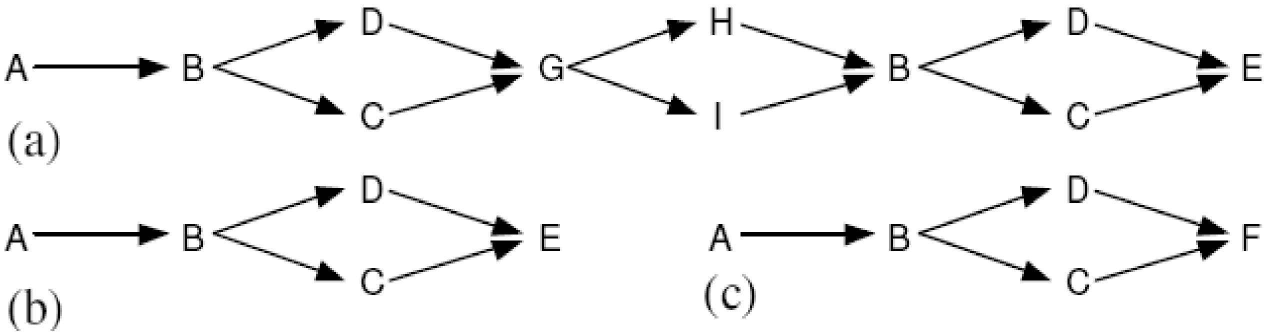
- Part I – Introduction to Process Mining
  - Context, motivation and goal
  - General characteristics of the analyzed processes and logs
  - Classification of Process Mining approaches
- Part II – Workflow discovery
  - Basic CFG induction algorithm
  - Other algorithms (α-algorithm, Heuristic Miner, Fuzzy mining)
- Part III – Beyond the control-flow perspective
  - Organizational mining
  - Social net mining
  - Extension techniques
- Part IV – Evaluation and validation of discovered models
  - Conformance Check
  - Log-based property verification
- Part V – Advanced Process Mining approaches
  - Discovery of hierarchical process models
  - Discovery of process taxonomies
  - Outlier detection in a process mining setting

Multi-phase mining

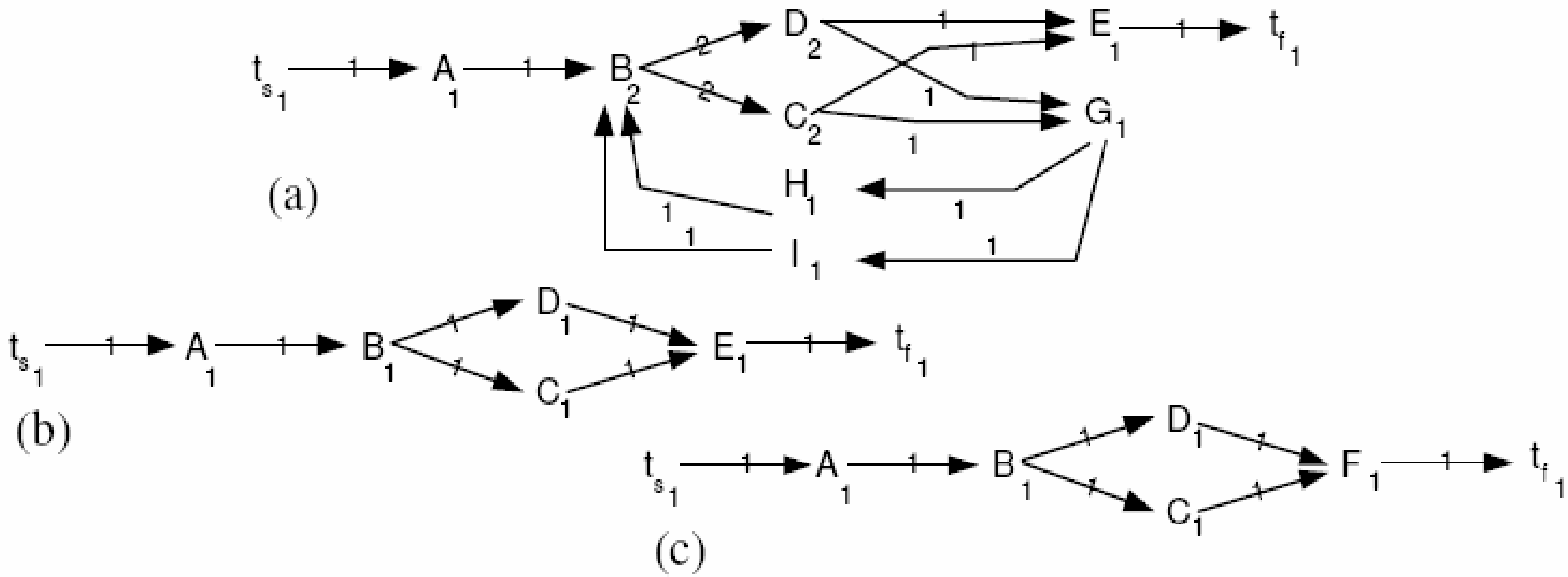
- Main steps
  - Convert each log trace into an *execution graph*, where each node corresponds to the execution of a task
    - a task label can appear multiple times!
  - Convert each instance graph into an *instance graph*
    - each node is associated with a single task
    - both nodes and edges are labelled with occurrence counters
    - a fictive start node and a fictive final node are introduced
  - Merge the *instance graphs* into an *aggregated graph model*
    - The model is simply the union of all the *instance graphs*
    - Arc/node counters are summed up
  - Convert the CFG model into an EPC

Multi-phase mining: Example

- *Execution graphs (acyclic)*:



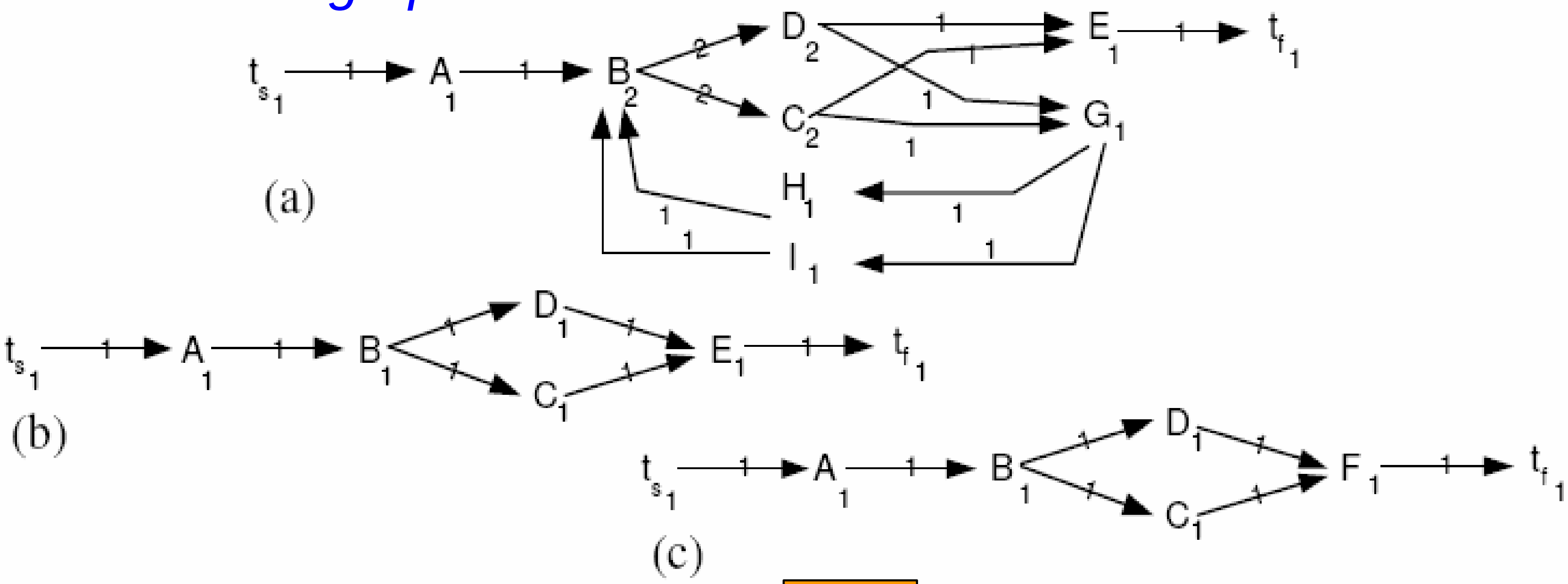
- *Instance graphs (some cycles can be created)*



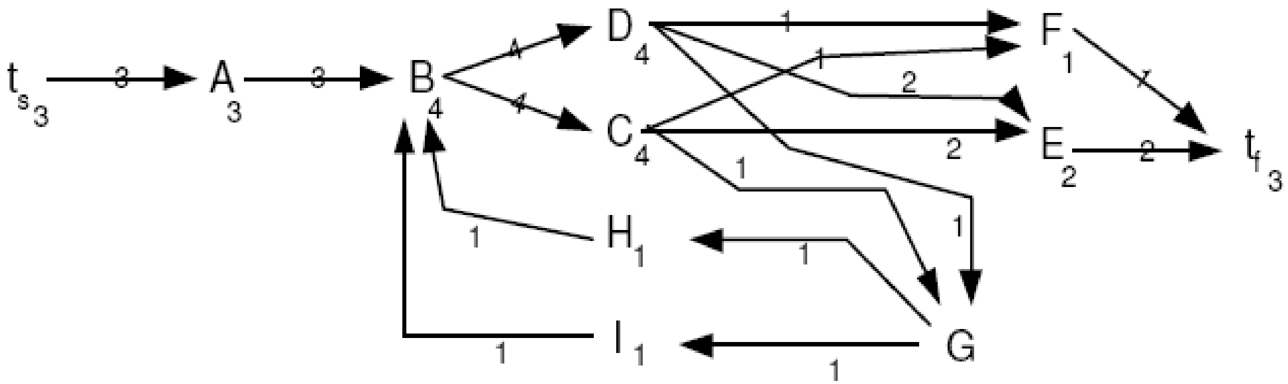
any log
node)

Multi-phase mining: Example (2)

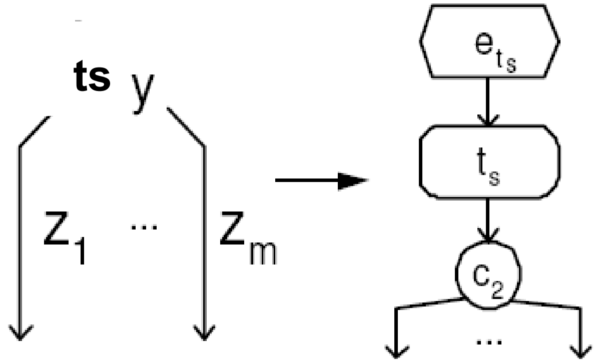
- *Instance graphs:*

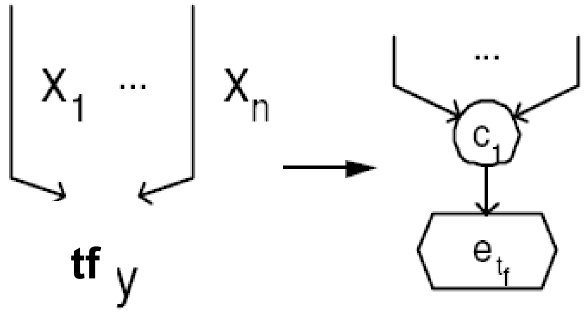


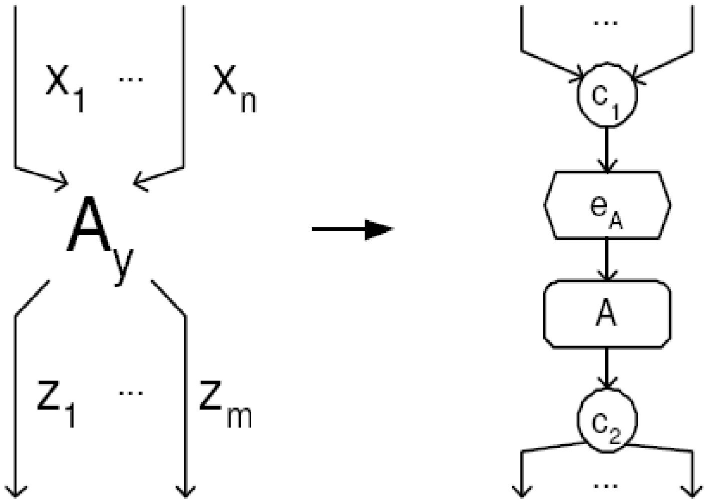
- *Aggregated graph model:*



Multi-phase mining: deriving an EPC



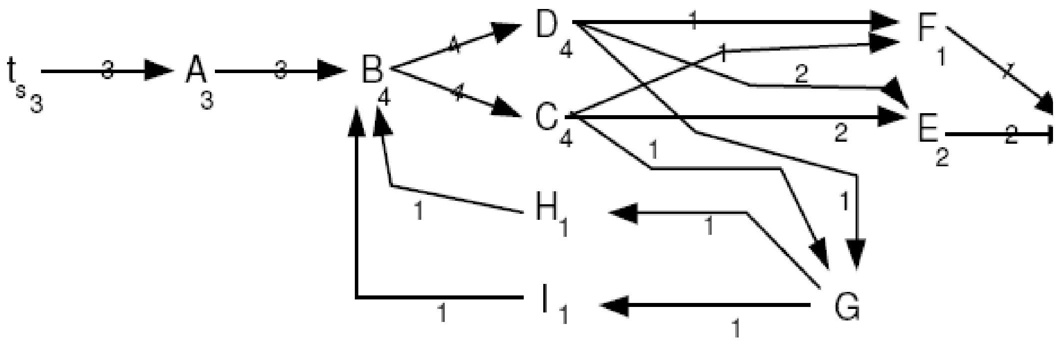




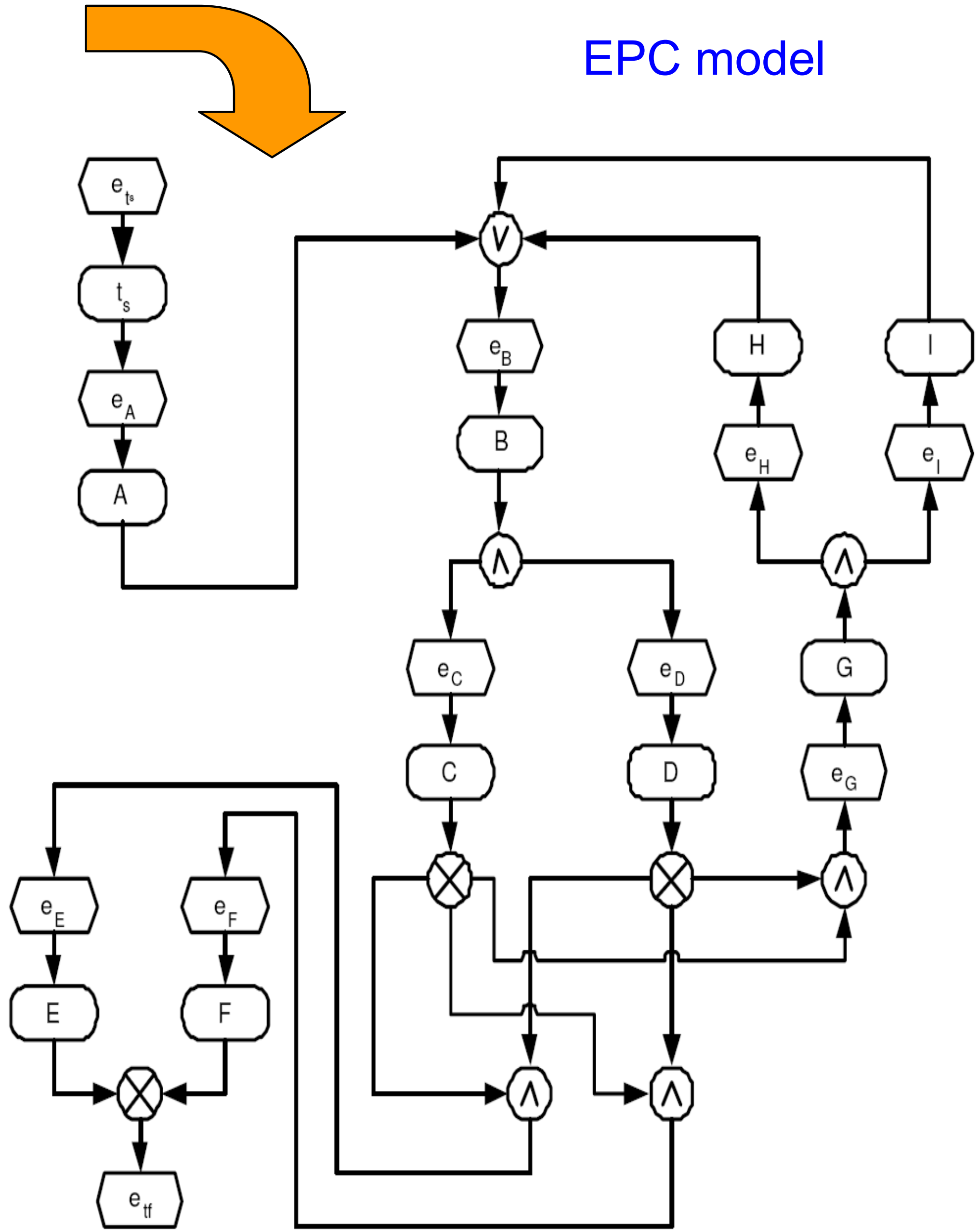
- If $\sum_{i=1}^{n}(x_i) = y$ then $c_1 = XOR$,
- If $\forall_{i=1}^{n}(x_i) = y$ then $c_1 = AND$,
- Else $c_1 = OR$.

- If $\sum_{i=1}^{m}(z_i) = y$ then $c_2 = XOR$,
- If $\forall_{i=1}^{m}(z_i) = y$ then $c_2 = AND$,
- Else $c_2 = OR$.

Multi-phase mining: Example (3)



Aggregated graph model



EPC model

Workflow discovery algorithms

- Multi-phase PM
- α-algorithm
- Heuristics Miner
- Genetic PM
- Fuzzy Miner

Outline

α-algorithm

Output: a Petri net

Method:

- Read the input log
- Get the set of tasks
- Infer a set of ordering relations
- Build the net based on inferred relations
- Return the net

α-algorithm - Ordering Relations >,→,||,#

- **Direct succession**:

  x>y iff for some case x is directly followed by y

- **Causality**:

  x→y iff x>y and not y>x

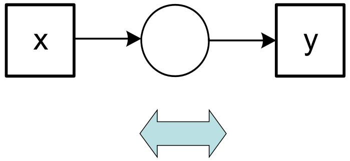
- **Parallel**:

  x||y iff x>y and y>x

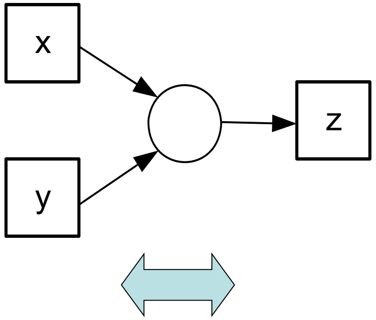
- **Unrelated**:

  x#y iff not x>y and not y>x

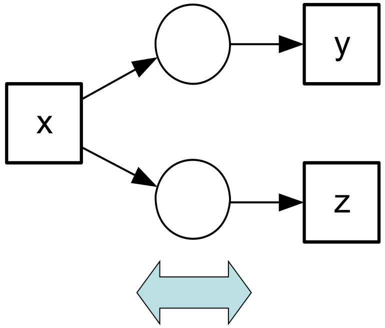
From the ordering relations to the Petri net



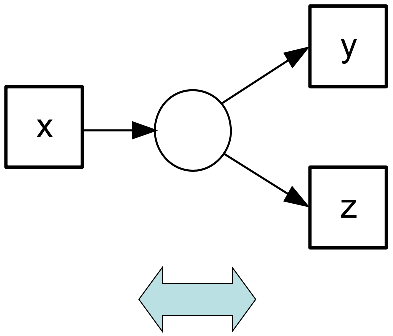
x→y



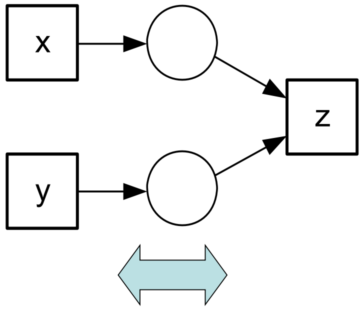
x→z, y→z, and x#y



x→y, x→z, and y||z



x→y, x→z, and y#z



x→z, y→z, and x||y

α-algorithm - Formalization

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