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

"world" model

- business processes
- people
- machines
- components
- organizations

information system

supports/controls

models

specifies

analyzes

configures

implements

discovery

conformance

extension

process model

(event) model

process logs

Process Mining Tools

Process Model

Organizational Model

Social Network
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 $<A, E, a_0, A_F, \text{Fork}, \text{Join}>$ 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
    - $\text{Fork}: (A-A_F) \times \{\text{AND, OR, XOR}\}$ and
    - $\text{Join}: (A-\{a_0\}) \times \{\text{AND, OR}\}$

- Example: a CFG for the toy process *Order Management*
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 \( lw \) of \( W \) such that \( s \) is a topological order of \( lw \)
  - 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

Admitted Instances = 20;
Modeled Traces = 276.

\[
soundness(\{W, L\}) = \frac{16}{276} = 5.797\%
\]

\[
completeness(\{W, L\}) = \frac{16}{16} = 100\%
\]
Example: a way to get higher soundness

Modeled Traces = 64

Considered trace Log (L)

\[ s_1: acbfgh \]
\[ s_2: abfcgh \]
\[ s_3: acgbfh \]
\[ s_4: abcgin \]
\[ s_5: abfgimn \]
\[ s_6: acbf gin \]
\[ s_7: acbgfilmn \]
\[ s_8: abc efiln \]
\[ s_9: abefcin \]
\[ s_{10}: acgbefiln \]
\[ s_{11}: abcedfgin \]
\[ s_{12}: acdbefgin \]
\[ s_{13}: abcfdgimn \]
\[ s_{14}: acdbf gin \]
\[ s_{15}: abcdg Fiona \]
\[ s_{16}: acbf gin \]

\[ s_8, s_{11}, s_{12} \] comply with \[ W_1 \cup W_2 \]

\[ \text{soundness}(W_1 \cup W_2, L) = \frac{11}{97} = 11.34\% \]

\[ \text{completeness}(W_1 \cup W_2, L) = \frac{11}{16} = 68.75\% \]
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 (\(\alpha\)-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: \(^\text{^} (\text{and}), \ X (\text{xor}) \text{ and } V (\text{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

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
Dependency graph

- Dependency graph for a log $L$ is a graph $D_L=<A,E>$ such that
  \[ E=\{ (a, b) \mid \exists s \in L, i \in \{1, \ldots, \text{length}(s)-1\} \text{ s.t. } a=s[i] \land b=s[i+1] \}; \]

- 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 $L=\{abcde, adbce, ae\}$

- $a$, $b$ 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 $\mathcal{W}S = (A, E, a_0, A_F, \text{Join}, \text{Fork})$.

Method: Perform the following steps:

1. $\langle A, E \rangle := D_{\mathcal{L}_P}$; //nodes and edges are initially those of the dependency graph
2. for each $(a, b) \in E$ s.t. $a$ and $b$ are parallel in $\mathcal{L}_P$ do //remove cycles
3. $E := E - \{(a, b)\}$;
4. for each $s \in \mathcal{L}_P$ s.t. $\{a, b\} \subseteq \text{tasks}(s)$ do //update edges
5. $\text{pre} := s[i], \text{where } s[i] \to a \land s[i] \to b \text{ and not exists } s[k] \text{ with } k > i \text{ s.t. } s[k] \to a \land s[k] \to b$;
6. $E := E \cup \{(pre, a)\} \cup \{(pre, b)\}$;
7. $\text{post} := s[j], \text{where } a \to s[j] \land b \to s[j] \text{ and not exists } s[h] \text{ with } h < j \text{ s.t. } a \to s[h] \land b \to s[h]$;
8. $E := E \cup \{(a, post)\} \cup \{(b, post)\}$;
9. end for
10. end for
11. $a_0 := s[1], \text{no matter of which trace } s \in \mathcal{L}_P \text{ is selected}; \ A_F := \{a \in A \mid \exists b \in A s.t. a \to b\}$;
12. for each $a \in A$ do //construction of local constraints
13. if $\forall s \in \mathcal{L}_P \text{ s.t. } a \in \text{tasks}(s)$, it holds that $\forall c \text{ s.t. } (a, c) \in E, c \in \text{tasks}(s)$ then $\text{Fork}(a) = \text{AND}$;
14. else if $\forall s \in \mathcal{L}_P \text{ s.t. } a \in \text{tasks}(s), |\{c \mid (a, c) \in E \land c \in \text{tasks}(s)\}| = 1$ then $\text{Fork}(a) = \text{XOR}$;
15. else $\text{Fork}(a) = \text{OR}$;
16. if $\forall s \in \mathcal{L}_P \text{ s.t. } a \in \text{tasks}(s), (c, a) \in E \Rightarrow c \in \text{tasks}(s)$ then $\text{Join}(a) = \text{AND}$;
17. else $\text{Join}(a) = \text{OR}$;
18. end for
19. return $\langle A, E, a_0, A_F, \text{Join}, \text{Fork} \rangle$;

Build the dependency graph and make it coincide with the initial CFG model.

Removal of cycles:
Remove, from $E$, all edges between parallel activities.

Connect the vertices of 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 \( \mathcal{WS} = \{A, E, a_0, A_F, \text{Join, Fork}\} \).

Method: Perform the following steps:

1. \( \langle A, E \rangle := D_{\mathcal{L}_P} \);  
   //nodes and edges are initially those of the dependency graph
2. for each \((a, b) \in E\) s.t. \(a\) and \(b\) are parallel in \(\mathcal{L}_P\) do  
   //remove cycles
   3. \(E := E - \{(a, b)\}\);
4. for each \(s \in \mathcal{L}_P\) s.t. \(\{a, b\} \subseteq \text{tasks}(s)\) do  
   //update edges
   5. \(\text{pre} := s[i]\), where \(s[i] \to a \land s[i] \to b\) and not exists \(s[k] \text{ with } k > i\) s.t. \(s[k] \to a \land s[k] \to b\);
6. \(E := E \cup \{(\text{pre}, a)\} \cup \{(\text{pre}, b)\}\);
7. \(\text{post} := s[j]\), where \(a \to s[j] \land b \to s[j]\) and not exists \(s[k] \text{ with } h < j\) s.t. \(a \to s[h] \land b \to s[h]\);
8. \(E := E \cup \{(a, \text{post})\} \cup \{(b, \text{post})\}\);
9. end for
10. end for
11. \(a_0 := s[1]\), no matter of which trace \(s \in \mathcal{L}_P\) is selected;  
   \(A_F := \{a \in A \mid \lnot b \in A \text{ s.t. } a \to b\}\);
12. for each \(a \in A\) do  
   //construction of local constraints
   13. if \(\forall s \in \mathcal{L}_P\) s.t. \(a \in \text{tasks}(s)\), it holds that \(\forall c \text{ s.t. } (a, c) \in E, c \in \text{tasks}(s)\) then \(\text{Fork}(a) = \text{AND}\);
   14. else if \(\forall s \in \mathcal{L}_P\) s.t. \(a \in \text{tasks}(s)\), \(|\{c \mid (a, c) \in E \land c \in \text{tasks}(s)\}| = 1\) then \(\text{Fork}(a) = \text{XOR}\);
   15. else \(\text{Fork}(a) = \text{OR}\);
   16. if \(\forall s \in \mathcal{L}_P\) s.t. \(a \in \text{tasks}(s), (c, a) \in E \Rightarrow c \in \text{tasks}(s)\) then \(\text{Join}(a) = \text{AND}\);
   17. else \(\text{Join}(a) = \text{OR}\);
18. end for
19. return \(\langle A, E, a_0, A_F, \text{Join, Fork}\rangle\);
Example: Algorithm simulation

Dependency graph

Precedences:

<table>
<thead>
<tr>
<th>a→b</th>
<th>b→c</th>
<th>c→b</th>
<th>d→b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a→c</td>
<td>b→d</td>
<td>c→d</td>
<td>d→c</td>
</tr>
<tr>
<td>a→d</td>
<td>b→e</td>
<td>c→e</td>
<td>d→e</td>
</tr>
<tr>
<td>a→e</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parallel activities:
• b, c, d

Edges to remove:
• (b, c),
• (b, d),
• (c, d).
Example: Algorithm simulation

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

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

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

\[
\log L = \{abcde, adbce, ae\},
\]

\[
E := \{(a, b), (a, d), (a, e), (b, c), (c, d), (c, e), (d, b), (d, e)\}
\]

2 for each (a, b) \in E such that a and b are parallel in \(L_P\) do  //remove cycles
3 \quad E := E \setminus \{(a, b)\};
4 for each \(s \in L_P\) such that \{(a, b)\} \subseteq \text{tasks}(s) do  //update edges
5 \quad pre := s[i], where \(s[i] \rightarrow a\) \land s[i] \rightarrow b \land \text{not exists} s[k] \text{ with } k > i \text{ s.t. } s[k] \rightarrow a \land s[k] \rightarrow b;
6 \quad E := E \cup \{(pre, a)\} \cup \{(pre, b)\};
7 \quad post := s[j], where a \rightarrow s[j] \land b \rightarrow s[j] \land \text{not exists} s[h] \text{ with } h < j \text{ s.t. } a \rightarrow s[h] \land b \rightarrow s[h];
8 \quad E := E \cup \{(a, post)\} \cup \{(b, post)\};
9 end for
10 end for
Example: Algorithm simulation

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

\[ \log L = \{abcde, adbce, ae\} \]

\[ E := \{(a, b), (a, d), (a, e), (b, c), (c, d), (c, e), (d, b), (d, e)\} \]

\[ \forall \{a,e\} \cup \{b,e\} \]

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

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

log \( L = \{abcde, adbce, ae\} \),

\[ E := \{(a, b), (a, d), (a, e), (b, c), (c, d), (c, e), (d, b), (d, e)\} \cup \{(a, c)\} \cup \{(b, e)\} \]

4 for each \( s \in \mathcal{L}_P \) s.t. \( \{a, b\} \subseteq \text{tasks}(s) \) do //update edges
5 \( \text{pre} := s[i] \), where \( s[i] \rightarrow a \land s[i] \rightarrow b \) and not exists \( s[k] \) with \( k > i \) s.t. \( s[k] \rightarrow a \land s[k] \rightarrow b \);
6 \( E := E \cup \{(\text{pre}, a)\} \cup \{(\text{pre}, b)\} \);
7 \( \text{post} := s[j] \), where \( a \rightarrow s[j] \land b \rightarrow s[j] \) and not exists \( s[h] \) with \( h < j \) s.t. \( a \rightarrow s[h] \land b \rightarrow s[h] \);
8 \( E := E \cup \{(a, \text{post})\} \cup \{(b, \text{post})\} \);
9 end for
Example: Algorithm simulation

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

\[ \log L = \{abcde, adbce, ae\} \]

\[ E := \{(a, b), (a, d), (a, e),
(b, c), (c, d), (c, e),
(d, b), (d, e)\} \cup \{(a, c)\} \cup \{(b, e)\} \]

---

4 for each \( s \in \mathcal{L}_P \) s.t. \( \{a, b\} \subseteq \text{tasks}(s) \) do
5 \quad \text{//update edges}
6 \quad \text{pre} := s[i], \text{ where } s[i] \rightarrow a \land s[i] \rightarrow b \text{ and not exists } s[k] \text{ with } k > i \text{ s.t. } s[k] \rightarrow a \land s[k] \rightarrow b; \quad \\
7 \quad E := E \cup \{(\text{pre}, a)\} \cup \{(\text{pre}, b)\}; \quad \\
8 \quad \text{post} := s[j], \text{ where } a \rightarrow s[j] \land b \rightarrow s[j] \text{ and not exists } s[h] \text{ with } h < j \text{ s.t. } a \rightarrow s[h] \land b \rightarrow s[h]; \quad \\
9 \quad E := E \cup \{(a, \text{post})\} \cup \{(b, \text{post})\}; \quad \\
10 \text{end for
Example: Algorithm simulation

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

log \( L = \{abcde, adbce, ae\} \)

\[ \begin{array}{cccc}
  a \to b & b \to c & c \to b & d \to b \\
  a \to c & b \to d & c \to d & d \to c \\
  a \to d & b \to e & c \to e & d \to e \\
  a \to e
\end{array} \]

\[ E := \{(a, b), (a, d), (a, e), (b, c), (c, d), (c, e), (d, b), (d, e)\} \cup \{(a, c)\} \cup \{(b, e)\} \]

```
4     for each \( s \in \mathcal{L}_P \) s.t. \( \{a, b\} \subseteq \text{tasks}(s) \) do  //update edges
5       \text{pre} := s[i], where \( s[i] \to a \land s[i] \to b \) and not exists \( s[k] \) with \( k > i \) s.t. \( s[k] \to a \land s[k] \to b \);
6       \text{E} := \text{E} \cup \{(\text{pre}, a)\} \cup \{(\text{pre}, b)\};
7       \text{post} := s[j], where \( a \to s[j] \land b \to s[j] \) and not exists \( s[h] \) with \( h < j \) s.t. \( a \to s[h] \land b \to s[h] \);
8       \text{E} := \text{E} \cup \{(a, post)\} \cup \{(b, post)\};
9     end for
```
Example: Algorithm simulation

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

log $L = \{abcde, adbce, ae\}$,

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

2 for each $(a, b) \in E$ s.t. $a$ and $b$ are parallel in $\mathcal{L}_P$ do //remove cycles
3 $E := E - \{(a, b)\}$;
4 for each $s \in \mathcal{L}_P$ s.t. $\{a, b\} \subseteq tasks(s)$ do //update edges
5 $pre := s[i]$, where $s[i] \rightarrow a \land s[i] \rightarrow b$ and not exists $s[k]$ with $k > i$ s.t. $s[k] \rightarrow a \land s[k] \rightarrow b$;
6 $E := E \cup \{(pre, a)\} \cup \{(pre, b)\}$;
7 $post := s[j]$, where $a \rightarrow s[j] \land b \rightarrow s[j]$ and not exists $s[h]$ with $h < j$ s.t. $a \rightarrow s[h] \land b \rightarrow s[h]$;
8 $E := E \cup \{(a, post)\} \cup \{(b, post)\}$;
9 end for
10 end for
Example: Algorithm simulation

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

$\log L = \{abcde, adbce, ae\}$,

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

- $a_0 := a$;
- $A_F := \{e\}$
Example: Algorithm simulation

\[
\begin{align*}
\log L &= \{\text{abcde, adbce, ae}\}, \\
A &= \{a, b, c, d, e\}, \\
E &= \{(a, b), (a, d), (a, e), (b, c), (c, d), (c, e), (d, b), (d, e)\} \\
&\cup \{(a, c)\} \cup \{(b, e)\}
\end{align*}
\]

- \(a_0 := a\)
- \(A_F := \{e\}\)

```
12  for each \(a \in A\) do
13     //construction of local constraints
14     if \(\forall s \in L_P\) s.t. \(a \in \text{tasks}(s)\), it holds that \(\forall c\) s.t. \((a, c) \in E\), \(c \in \text{tasks}(s)\) then \(\text{Fork}(a) = \text{AND}\);
15     else if \(\forall s \in L_P\) s.t. \(a \in \text{tasks}(s)\), \(|\{c | (a, c) \in E \land c \in \text{tasks}(s)\}| = 1\) then \(\text{Fork}(a) = \text{XOR}\);
16     else \(\text{Fork}(a) = \text{OR}\);
17     if \(\forall s \in L_P\) s.t. \(a \in \text{tasks}(s)\), \((c, a) \in E \Rightarrow c \in \text{tasks}(s)\) then \(\text{Join}(a) = \text{AND}\);
18     else \(\text{Join}(a) = \text{OR}\);
19  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

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
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):**

  (a)

  ![Diagram](image)

  (b)

  ![Diagram](image)

  (c)

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

  (a)

  ![Diagram](image)

  (b)

  ![Diagram](image)

  (c)
Multi-phase mining: Example (2)

- **Instance graphs:**

  ![Instance graphs](image)

- **Aggregated graph model:**

  ![Aggregated graph model](image)
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

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
**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:**
  \[ x > y \iff \text{for some case } x \text{ is directly followed by } y \]

- **Causality:**
  \[ x \rightarrow y \iff x > y \text{ and not } y > x \]

- **Parallel:**
  \[ x || y \iff x > y \text{ and } y > x \]

- **Unrelated:**
  \[ x \# y \iff \text{not } x > y \text{ and not } y > x \]
From the ordering relations to the Petri net

$x \rightarrow y$

$x \rightarrow z, y \rightarrow z, \text{ and } x \# y$

$x \rightarrow y, x \rightarrow z, \text{ and } y || z$

$x \rightarrow y, x \rightarrow z, \text{ and } y \# z$

$x \rightarrow z, y \rightarrow z, \text{ and } x || y$
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. $T_I = \{ t \in T \mid \exists \sigma \in W \; t = \text{first}(\sigma) \}$,
3. $T_O = \{ t \in T \mid \exists \sigma \in W \; t = \text{last}(\sigma) \}$,
4. $X_W = \{ (A,B) \mid A \subseteq T_W \land B \subseteq T_W \land \forall a \in A \forall b \in B \; a \rightarrow_W b \land \forall a_1,a_2 \in A \; a_1 \#_W a_2 \land \forall b_1,b_2 \in B \; b_1 \#_W b_2 \}$,
5. $Y_W = \{ (A,B) \in X \mid \forall (A',B') \in X A \subseteq A' \land B \subseteq B' \Rightarrow (A,B) = (A',B') \}$,
6. $P_W = \{ p_{(A,B)} \mid (A,B) \in Y_W \} \cup \{ i_W,o_W \}$,
7. $F_W = \{ (a,p_{(A,B)}) \mid (A,B) \in Y_W \land a \in A \} \cup \{ (p_{(A,B)},b) \mid (A,B) \in Y_W \land b \in B \} \cup \{ (i_W,t) \mid t \in T_I \} \cup \{ (t,o_W) \mid t \in T_O \}$, and
8. $\alpha(W) = (P_W,T_W,F_W)$. 

\(\alpha\)-algorithm - Formalization