Multi-Phase Process Mining: Aggregating Instance Graphs into EPCs and Petri Nets

B.F. van Dongen, and W.M.P. van der Aalst

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{b.f.v.dongen, w.m.p.v.d.aalst}@tm.tue.nl

Abstract. The goal of *process mining* is to automatically generate a process model from an event log, e.g., automatically constructing an EPC based on the transaction logs in SAP. Currently available process mining techniques typically try to generate a complete process model from the data acquired in a single step. In this paper, we propose a multi-step approach. First models are generated for each individual process instance (this can be done in many steps). In the final step however, these instance models are aggregated to obtain an overall model for the entire data set. In this paper, we focus on that final step, i.e., aggregating instance graphs. The work is motivated by recent tools and techniques to generate instance models. For example, widely used tools such as ARIS PPM and InConcert generate instance models that can be interpreted by our process mining tool. The result of our multi-step approach can be represented in different types of process models. In this paper, we show a translation from aggregated instance graphs to Petri nets using Event-driven Process Chains (EPCs) as an intermediate step.

Keywords: Process mining, Event-driven Process Chains, Aggregation.

1 Introduction

Increasingly, process-driven information systems are used to support operational business processes. Some of these information systems enforce a particular way of working. For example, Workflow Management Systems (WFMSs) can be used to force users to execute tasks in a predefined order. However, in many cases systems allow for more flexibility. For example transactional systems such as ERP (Enterprise Resource Planning), CRM (Customer Relationship Management) and SCM (Supply Chain Management) are known to allow the users to deviate from the process specified by the system, e.g., in the context of SAP R/3 the reference models, expressed in terms of Event-driven Process Chains (EPCs, cf. [16, 17, 21]), are only used to guide users rather than to enforce a particular way of working. Operational flexibility typically leads to difficulties with respect to performance measurements. The ability to do these measurements, however, is what made companies decide to use a transactional system in the first place.

One way of doing performance measurements is through so-called instance graphs. Instance graphs can be considered to be process models of single executions of a case. These instance graphs are a nice way to show parallelism in process execution and using these instance graphs, many interesting performance characteristics can be calculated and visualized. However, sometimes looking at single instances of a process is too fine-grained. Especially managers usually want to calculate performance measures on a more aggregated level. Therefore, in this paper, we propose a way to aggregate instance graphs to an aggregation graph. Such an aggregation graph represents the execution of a set of cases instead of just one. These aggregation graphs can be used as the starting point for many interesting measurements in the context of Business Process Management. The work presented in this paper is greatly inspired by the commercial tool Aris PPM (Aris Process Performance Monitor, c.f. [15]). This tool indeed visualizes single instances of a process using instance graphs and is capable of aggregating them. However, in contradiction to the algorithms presented here, each installation of Aris PPM requires consultants to *manually* configure the way instances are generated and the way they are to be aggregated.

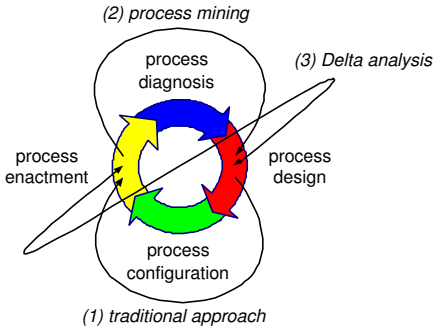

Fig. 1. The BPM life cycle.

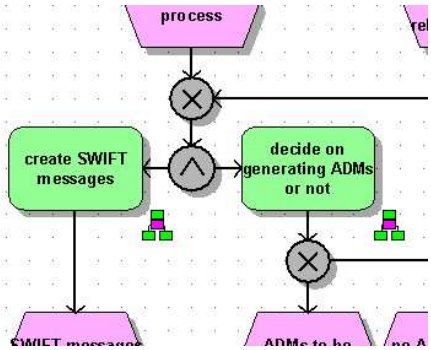

Fig. 2. A partial EPC.

In Figure 1, we show the typical life cycle of a process in Business Process Management. In the traditional approach, you start with a process design. Then, the design is implemented in some process aware information system in the configuration phase. After the process has been running for a while in the enactment phase, diagnostics can be used to come to another (and preferably better) design. The research area of *process mining* focuses mainly on trying to come to a process design using data gathered in the enactment phase. Furthermore, the concept of *delta analysis* is to compare a process as it is actually executed with the way it was intended to be executed. With the approach presented in this paper, we focus on process mining, i.e. we generate a process model from cases that have been executed. The result can then be used for comparison with the original process model (delta analysis).

Processes can be described in several modeling languages. Each language has its own advantages and disadvantages. In this paper, we choose to focus on the modeling language of Event driven Process Chains, or EPCs (c.f. [16, 17]. This

choice is motivated by the fact that it is widely used in industry and is easy to comprehend for people without a background in theoretical computer science. However, the general result is independent of this format, and in Section 5 and Section 6, we show a translation to Petri nets. An example of a part of an EPC can be seen in Figure 2. We introduce EPCs in some more detail in Section 4.

The remainder of this paper is organized as follows. First, we give some general notations and concepts that will be used throughout the paper. Then, in Section 3, we show the formal process of aggregating instance graphs. In Section 4, we show how our format-independent result can be translated to EPCs. In Section 5 and Section 6, we give a translation of our result into Petri nets. The latter translation into Petri nets uses EPCs as an intermediate step. Finally, in Section 7, we present our toolset that supports the processes discussed here, and we conclude the paper with some conclusions.

A short version of the work presented here has been submitted to the International Conference on Cooperative Information Systems (CoopIS05) [12]. This paper extends [12] by adding formal proofs for the various aggregation steps and by adding a translation to Petri nets.

2 Preliminaries

In this section, we introduce the basic concepts behind multi-phase process mining, as well as some basic notations. In process mining, the goal is to construct a formal model describing a set of real life executions. Usually, these models can be visualized as a graph. In this paper, we do this by assuming that for each process instance or case, there is a graph that describes that instance. Then, we sum these individual graphs to an aggregation graph. An example of an algorithm to generate a graph that describes a process instance can be found in [11]. In order to be able to talk about instance and aggregation graphs, we introduce some basic notations.

2.1 General concepts

In general, a graph consists of a set of nodes and a set of edges. We introduce a counting operator # over sets, such that we are able to count the number of elements of a set for which a certain property holds.

Definition 2.1. (Counting operator) Let S be some set. We define a counting operator # over the elements of S, in such a way that we can count the number of elements for which some predicate P holds as $\#_{s \in S} P(s) = \sum_{s \in S} \begin{cases} 1 \text{ if } P(s) \\ 0 \text{ if not } P(s) \end{cases}$
Note that by definition, $S = \emptyset$ implies that $\#_{s \in S} P(s) = 0$.

Since we will be using graphs throughout the paper, we introduce some basic notations for graph-related properties.

Definition 2.2. (Path in a directed graph) Let $G = (N, E)$ be a directed graph with N the set of nodes and $E \subseteq N \times N$ the set of edges. Let $a \in N$ and

$b \in N$. We define a path from a to b as a list of nodes denoted by $\langle n_1, n_2, \ldots, n_k \rangle$ with $k \geq 2$ such that $n_1 = a$ and $n_k = b$ and $\forall_{i \in \{1,\ldots,k-1\}}((n_i, n_{i+1}) \in E)$.

A path in a directed graph is an ordered list of nodes, such that for every node in the list, there is an edge to the next node in the list. Note that we define a path to contain at least two nodes.

Definition 2.3. (Pre-set and Post-set) Let $G = (N, E)$ be a directed graph and let $n \in N$. We define $\overset{G}{\bullet} n = \{m \in N \mid (m, n) \in E\}$ as the pre-set and $n \overset{G}{\bullet} = \{m \in N \mid (n, m) \in E\}$ as the post-set of n with respect to graph G. If the context is clear, the superscript G may be omitted, resulting in $\bullet n$ and $n\bullet$.

Using the pre- and postset of a node, we indicate which nodes have an edge to or from that node respectively.

2.2 Instance graphs

As stated in the introduction, we use a multi-step approach where we first generate models of instances and then aggregate these instance models into an overall process model. For this purpose each process instance is not described by a simple sequence of events but as a directed graph. In every process, there is a set of tasks that has to be executed. Whether or not a specific task is executed, and if so, how often, completely depends on a specific instance of a process. For example an insurance claim handling process can contain a rule that in case of a claim that exceeds a specific amount, a manager has to check the final decision before sending it to the customer. However, if we look at a completed process instance, we know that for all those choices, a certain decision was made. As a result, there is no need to visualize any kind of choice in the representation of a process instance, i.e. every split/join is an AND-split/join. This implies that it suffices to use a normal directed graph. We call such a representation an *instance graph* and each node in that graph represents the execution of a task in the process.[1]

As indicated in the introduction, tools such ARIS PPM (IDS Scheer, [15]), Staffware SPM, and InConcert (TIBCO) generate instance graphs. Figures 3 and 4 show screenshots of Aris PPM. Each of the two figures depicts an instance graph of an insurance process. Both instance graphs are represented in terms of so-called instance EPC. Besides tools such as ARIS PPM directly providing instance graphs, it is also possible to obtain instance graphs via an analysis of "raw" event logs (e.g., transaction logs, audit trails, etc.). For a more detailed description of instance graphs and how they can be generated from a event logs, we refer to [11]. However, we do not limit ourselves to the algorithm presented [11]. Instead, we define an instance graph in a generic way. This explains the term "multi-phase process mining", instead of "two-phase process mining". Generating instance graphs from log files can be done in multiple phases.

[1] In spite of some subtle differences, instance graphs are similar to runs [9] also known as occurrence nets or partially ordered Petri net processes, cf. Section 8.

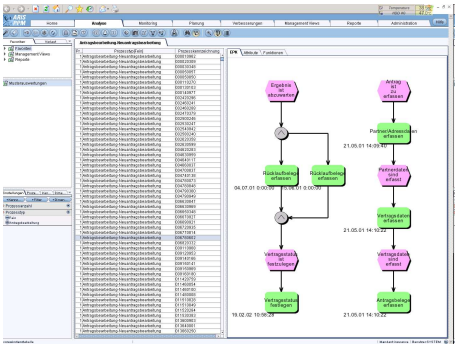

Fig. 3. Aris PPM

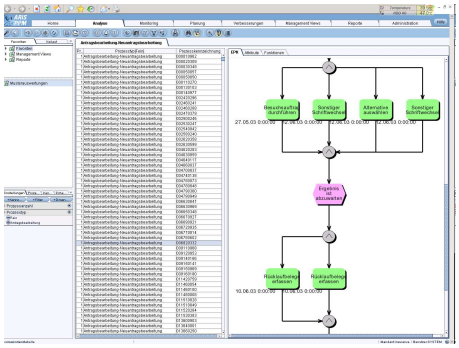

Fig. 4. Aris PPM

Definition 2.4. (Instance Graph) Let T be a set of tasks. We define $IG = (N, E, T, l)$ as an instance graph over T where:

1. $N \neq \emptyset$ is the set of nodes.
2. $E \subseteq N \times N$ is the set of edges.
3. T is a set of tasks.
4. $l : N \rightarrow T$ is a labeling function, mapping nodes onto T.
5. The graph (N, E) is acyclic, i.e. for all paths $\langle n_1, \ldots, n_k \rangle : n_1 \neq n_k$.
6. $\forall_{(a,b)\in E} \langle a, b \rangle$ is the *only* path from a to b.
7. $\forall_{a,b\in N} l(a) = l(b)$ implies that there is a path from a to b or from b to a.

Definition 2.4 shows that instance graphs consist of four parts. N is a set of nodes that represent all task-executions that appeared in that instance. The labeling function l is a function that maps the task executions in N onto tasks that were specified in the information system. Note that there can be multiple executions of the same task, which means that several elements of N will be mapped onto the same element in T. Also, not all elements of T need to be mapped, i.e. not all tasks have to appear in a process instance. Using a set of edges E, we represent causal relations between task executions, i.e. an edge represents the fact that tasks appeared as a result of the execution of another task. Finally, there are three properties that an instance graph has to adhere to (see Definition 2.4):

5 There cannot be any cycles in the graph. A cycle would mean that a task instance is performed more than once. However, this is impossible, since it would be the *task* that is executed twice, resulting in two *instances* or *executions*, and thus, two nodes in the instance graph with the same label.

6 If there is an edge between two nodes, there cannot be another path between these nodes. The reason for this is that an edge represents a direct causal relation. If task a would cause task b to execute and task b would cause task c to be executed, then there is already a causal relation between a and c and the edge (a, c) would be irrelevant.

7 Two executions of the same task cannot appear unrelated. There must be a direct or indirect causal dependency between the two. This property is necessary for the aggregation process and realistic in practice where multiple executions of the same task typically result from a loop.

Figure 5 shows a collection of three instance graphs, where the nodes are represented by their labels. Note that we use an abstract example rather than real life instance graphs such as the ones shown in figures 3 and 4.

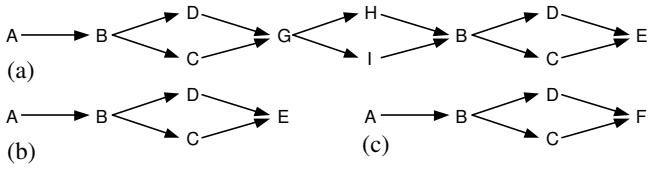

Fig. 5. Example set of instance graphs.

As we stated before, the labeling function l provides the actual task that belongs to a task execution. For our convenience we define the same labeling function on a set of task executions.

Definition 2.5. (Label set) Let $IG = (N, E, T, l)$ be an instance graph. For $S \subseteq N$ we define $l(S) = \bigcup_{s \in S} \{l(s)\}$.

We have now defined how a process instance can be described by an instance graph. These instance graphs have an interesting property with respect to the labeling function. These properties will be used later in the aggregation process.

Property 2.6. (Instance graph properties) Let $IG = (N, E, T, l)$ be an instance graph. For $n \in N$:

- $|n\bullet| = |l(n\bullet)|$, i.e. all nodes in a postset have different labels.
- $|\bullet n| = |l(\bullet n)|$, i.e. all nodes in a preset have different labels

Proof. From the definition of a label set, it is obvious that $|n\bullet| \geq |l(n\bullet)|$. It remains to be proven that $|n\bullet| \not> |l(n\bullet)|$. Assume that $|n\bullet| > |l(n\bullet)|$. Then there exists a $m_1, m_2 \in n\bullet$ such that $l(m_1) = l(m_2)$. From Definition 2.4 we know that there must exist a path $\langle m_1, \ldots, m_2 \rangle$ in IG (or the other way around, for which the proof is symmetrical). Since $m_1 \in n\bullet$ there also exists a path $\langle n, m_1, \ldots, m_2 \rangle$. However, since $(n, m_2) \in E$ this violates the sixth requirement in Definition 2.4. Therefore, $|n\bullet| = |l(n\bullet)|$. For $|\bullet n| = |l(\bullet(n)|$ a similar proof holds. □

In this section, we have presented a formalism to describe process instances in terms of instance graphs. When deploying a system like ARIS PPM [15], this would be the first step of the implementation phase. Using these instances, already basic performance metrics could be calculated. However, in this paper we are interested in the process of aggregating process instances. Therefore, we will show a way to generate a description of a whole process using an algorithm to aggregate a set of instance graphs.

3 Aggregation

In the previous section, we described the result of the first phase (or phases) in multi-phase process mining. This result can be obtained using tools like ARIS PPM (IDS Scheer, [15]) or InConcert (TIBCO), or by applying our algorithm presented in [11] to a simple event log. However, we like to point out again, that we do not limit the aggregation algorithm to any specific way of generating instance graphs, as long as Definition 2.4 is followed.

In this section, we look at the the final phase. This phase consists of aggregating a set of instance graphs. This aggregation allows us to find a description of a set of instances, instead of just one instance. The reason for doing that is twofold. First, it can be of help when monitoring performance. For example it could be possible to give a model for all cases that involved a certain amount of people. This could be useful to indicate whether the amount of people involved influences the throughput time. On the other hand, using the same technique, it is possible to generate a description of all instances, which is the goal in process mining in general.

Since an instance graph shows causal relations between *executions* of tasks, and an aggregation graph will show relations between the tasks *themselves*, we need to take an intermediate step, by defining a projection graph for each instance. Then, we will make an aggregation over these projections.

3.1 Instance graph projections

The first step in the aggregation process is projecting task executions in instance graphs onto tasks. Formally, an instance graph is projected onto its label set and the result is a new graph, where each task that was executed in the process instance is represented by a single node. Note the difference between a node for each *task-execution* in an instance graph and a node for each *task that was executed* in a projection graph. Furthermore, in the projection we introduce a fictive *start* node and a fictive *final* node.

Definition 3.1. (Instance Graph Projection) Let $IG = (N, E, T, l)$ be an instance graph. We define $\Pi(IG) = (N', E', T, l')$ as the projection of this instance graph onto T such that:

- $N' = T \cup \{t_s, t_f\}$ is the set of nodes, where $\{t_s, t_f\} \cap T = \emptyset \wedge t_s \neq t_f$.
- $E' = \{(l(a), l(b)) \mid (a, b) \in E\} \cup$
  $\{(t_s, l(b)) \mid b \in N \wedge \ {}^{IG}\!\bullet b = \emptyset\} \cup$
  $\{(l(a), t_f) \mid a \in N \ \wedge a \bullet^{IG} = \emptyset\}$ is the set of edges.
- $l' : N' \cup E' \rightarrow \mathbb{N}$ where
  - for all $a \in T$ we define $l'(a) = \#_{a' \in N}(l(a') = a)$
  - $l'(t_s) = l'(t_f) = 1$
  - for all $(t_s, b) \in E'$ we define $l'((t_s, b)) = 1$
  - for all $(a, t_f) \in E'$ we define $l'((a, t_f)) = 1$
  - for all $(a, b) \in E'$ such that $a \neq t_s \wedge b \neq t_f$, we define $l'((a, b)) = \#_{(a', b') \in E} \ (l(a') = a \wedge l(b') = b)$

In Definition 3.1 we build an instance graph in such a way that all nodes are now elements of the set of tasks, or one of the fictive nodes. Furthermore, edges are constructed in such a way that if two task executions were connected in the instance graph then the tasks are connected in the projection. Finally, for each task execution that was not preceded or succeeded by any other execution there now is an edge connecting the task to the start or final node respectively. The result is a graph where each node lies on a path from the start node to the final node. Furthermore, we introduced a labeling function l' that gives the number of times a task was executed (or an edge was taken) in an instance. Figure 6 shows the instance projections of the three instance graphs of Figure 5.

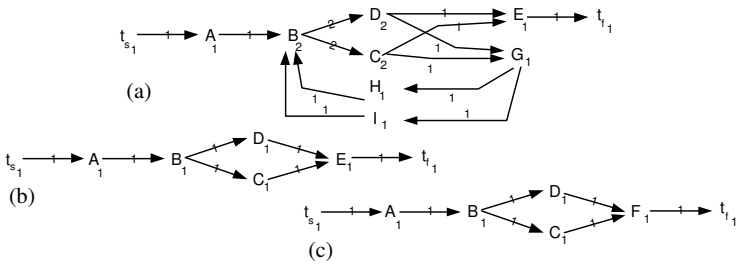

Fig. 6. Set of instance projections.

Let us now consider some interesting properties of instance projections and investigate their relations with instance graphs. The first property shows that when making a projection graph, no causality information is lost. In other words, each path that is present in an instance graph is also present in a projection, if we look at the tasks and not their individual executions.

Property 3.2. (Projection preserves paths) Let $IG = (N, E, T, l)$ be an instance graph and let $\Pi(IG) = (N', E', T, l')$ be the projection of this instance graph. Let $\{n_1, \ldots, n_k\} \subseteq N$ such that $\langle n_1, \ldots, n_k \rangle$ is a path in IG. We show that $\{n' \mid \exists_{1 \leq i \leq k} l(n_i) = n'\} \subseteq N'$ and $\langle l(n_1), \ldots, l(n_k) \rangle$ is a path in $\Pi(IG)$. (Note that $k \geq 2$.)

Proof. Follows directly from Definition 3.1. It is trivial to see that all nodes in the path are respected by the projection. Therefore, it suffices to prove that all the edges from the original path are in the instance projection. Assume $1 \leq i < k$. From the definition of a path we know that (n_i, n_{i+1}) is an edge, such that $(n_i, n_{i+1}) \in E$. From the definition of the projection function, we know that if $(n_i, n_{i+1}) \in E$ then $(l(n_i), l(n_{i+1})) \in E'$. Since this holds for all i such that $1 \leq i < k$, we have shown that $\langle l(n_1), \ldots, l(n_k) \rangle$ is indeed a path in $\Pi(IG)$. □

Property 3.2 shows that each path in an instance is still present in its projection. For an instance graph, we know that every time a node has two outgoing or incoming edges, this represents a parallel split or join respectively. There are no choices, since each node represents a task execution and not a task. Clearly this no longer holds for projection graphs where nodes represent tasks and not individual executions. Therefore, when making an instance graph projection, we

loose explicit information about parallelism. However, we show that we do not loose too much information, i.e. multi-choices and true parallelism are still visible in a projection graph.

In Property 3.3, we give some properties of the labeling function l'. The labeling function of an instance graph projection represents the number of times a task is executed in an instance. Furthermore, it shows how often causality relations caused the execution of a task. For these labels, we again derive some interesting properties that we need in the aggregation process.

Property 3.3. (Label properties) Let $IG = (N, E, T, l)$ be an instance graph and let $\Pi(IG) = (N', E', T, l')$ be the projection of this instance graph.

- $\forall_{(n',m')\in E'} l'((n',m')) \leq l'(n')$, i.e. the labels of all outgoing edges at a node are less or equal to the label of the node.
- $\forall_{n'\in N'} \left(\sum_{(n',m')\in E'} l'((n',m'))\right) \geq l'(n')$, i.e. the sum over the labels of all outgoing edges at a node is greater or equal to the label of the node.
- $\forall_{(n',m')\in E'} l'((n',m')) \leq l'(m')$, i.e. the labels of all incoming edges at a node are less or equal to the label of the node.
- $\forall_{n'\in N'} \left(\sum_{(n',m')\in E'} l'((n',m'))\right) \geq l'(m')$, i.e. the sum over the labels of all incoming edges at a node is greater or equal to the label of the node.

Proof. Let us first consider the first property. Assume that the opposite holds, i.e. there exists an $(n',m') \in E'$ such that $l'((n',m')) > l'(n')$. We will show that this leads to a contradiction. If $n' = t_s$ (or $m' = t_f$), then it is easy to find this contradiction because $l'(n') = 1$ (or $l'(m') = 1$) and $l'((n',m')) = 1$. Therefore, we may assume that $n' \neq t_s$ and $m' \neq t_f$. Using Definition 3.1 we know that $l'((n',m')) = \#_{(n,m)\in E}(l(n) = n' \wedge l(m) = m')$. The assumption $l'((n',m')) > l'(n')$ can only hold if $\exists_{m_1,m_2\in N}(l(m_1) = l(m_2) = m' \wedge (n,m_1) \in E \wedge (n,m_2) \in E)$. However, this violates Property 2.6. Therefore, $\forall_{(n',m')\in E'} l'((n',m')) \leq l'(n')$.

Let us now consider the second property. Assume that the opposite holds, i.e. there exists an $n' \in N'$ such that $\left(\sum_{(n',m')\in E'} l'((n',m'))\right) < l'(n')$. Again we will derive a contradiction. Let $Q = \{n \in N \mid l(n) = n'\}$. From Definition 3.1 we know that $|Q| = l'(n')$. We can partition Q into two subsets Q_1 and Q_2, such that $Q_1 = \{n \in Q \mid |n \overset{IG}{\bullet}| = 0\}$ and $Q_2 = \{n \in Q \mid |n \overset{IG}{\bullet}| > 0\}$. It is clear that $Q_1 \cap Q_2 = \emptyset$ and $Q = Q_1 \cup Q_2$. We can now express a lower bound on $\sum_{(n',m')\in E'} l'((n',m'))$ based on the cardinalities of Q_1 and Q_2. From Definition 3.1 we know that for all elements $q_1 \in Q_1$ there exists $(l(q_1), t_f) \in E'$ such that $l'((l(q_1), t_f)) = 1$. Furthermore, for all elements $q_2 \in Q_2$ there exists at least one $m \in q_2 \overset{IG}{\bullet}$ such that $(l(q_2), l(m)) \in E'$. Note that therefore $l'((l(q_2), l(m))) \geq 1$. Since $Q_1 \cap Q_2 = \emptyset$, we know that $\left(\sum_{(n',m')\in E'} l'((n',m'))\right) \geq |Q_1| + |Q_2|$. However, we know that $|Q_1| + |Q_2| = |Q| = l'(n')$. Therefore we have shown that $\left(\sum_{(n',m')\in E'} l'((n',m'))\right) \geq l'(n')$

The remaining two properties can be proven in a similar way. □

Using the properties from Property 3.3, we derive three properties that show that the type of branching at a task can still be derived from a projection graph. These properties will be used in Section 4 in order to determine which types of connectors should be used in the translation to an EPC.

In Property 3.4 we show that if one task execution in an instance graph never directly precedes more than one other task execution in the same graph, then this has to be an exclusive choice.

Property 3.4. (Exclusive choices can be found in projections) Let $IG = (N, E, T, l)$ be an instance graph and let $\Pi(IG) = (N', E', T, l')$ be the projection of this instance graph. For all $n \in N$ such that $|n {}^{IG}\!\bullet| = 1$ with $l(n) = n'$ and $\forall_{m \in N \wedge l(m)=n'} (|m {}^{IG}\!\bullet| = 1)$ holds that $\left(\sum_{m' \in n'^{\Pi(IG)}\!\bullet} l'((n', m'))\right) = l'(n')$.

Proof. Let $n \in N$ be such that $|n {}^{IG}\!\bullet| = 1$ with $l(n) = n'$ and $\forall_{m \in N \wedge l(m)=n'} (|m {}^{IG}\!\bullet| = 1)$. From Definition 3.1, we know that $l'(n') = \#_{m \in N} l(m) = n'$. Let $N_{out} = \{m \in N \mid \exists_{(a,m) \in E} l(a) = n'\}$. Since $\forall_{m \in N \wedge l(m)=n'} (|m {}^{IG}\!\bullet| = 1)$, we know that $|N_{out}| = l'(n')$. For all $(n', m') \in E'$, we know that $l'((n', m')) = \#_{(n,m) \in N_{out}} l(m) = m'$. Obviously, this implies that $\left(\sum_{m' \in n'^{\Pi(IG)}\!\bullet} l'((n', m'))\right) = l'(n')$.

□

In Property 3.5 we show that if the execution of a task sometimes directly precedes the execution of more than one other task, then this is still visible in the projection graph.

Property 3.5. (Multi-choices can be found in projections) Let $IG = (N, E, T, l)$ be an instance graph and let $\Pi(IG) = (N', E', T, l')$ be the projection of this instance graph. For all $n \in N$ such that $|n {}^{IG}\!\bullet| > 1$ and $l(n) = n'$ holds that $\left(\sum_{m' \in n'^{\Pi(IG)}\!\bullet} l'((n', m'))\right) > l'(n')$.

Proof. Let $n \in N$ be a node such that $|n {}^{IG}\!\bullet| > 1$ and $n' = l(n)$. If n is the only node in the instance graph with label n' then the property holds, since $l'(n') = 1$ and $\left(\sum_{m' \in\ n'^{\Pi(IG)}\!\bullet} l'(n', m')\right) = |n {}^{IG}\!\bullet|$ by definition.
If there exist other nodes with label n' then we partition them into two sets: $N_1 = \{n_1 \in N \mid l(n_1) = n' \wedge |n_1 {}^{IG}\!\bullet| > 0\}$ and $N_2 = \{n_1 \in N \mid l(n_1) = n' \wedge |n_1 {}^{IG}\!\bullet| = 0\}$. Note that $n \in N_1$. Since $N_1 \cap N_2 = \emptyset$, we know by the definition of the projection that $l'(n') = |N_1| + |N_2|$. Furthermore, from that same definition, we know that $\left(\sum_{m' \in\ n'^{\Pi(IG)}\!\bullet} l'(n', m')\right) = \sum_{n_1 \in N_1} (|n_1 {}^{IG}\!\bullet|) + \sum_{n_2 \in N_2} (1)$. The second term here comes from the fact that the nodes in N_2 are final nodes. Therefore, in the projection they get exactly one outgoing edge to t_f. Since we know that $|n_1 {}^{IG}\!\bullet| > 0$ for $n_1 \in N_1$ and $|n {}^{IG}\!\bullet| > 1$, we can conclude that $\sum_{n_1 \in N_1} (|n_1 {}^{IG}\!\bullet|) > |N_1|$. Hence, we deduce that $\left(\sum_{m' \in\ n'^{\Pi(IG)}\!\bullet} l'(n', m')\right) > |N_1| + |N_2| = l'(n')$. □

The final property is stronger. We show that if a certain task execution *always* directly precedes the execution of *the same set of* other tasks, then this remains visible in the instance graph projection.

Property 3.6. (True parallelism can be found in projections) Let $IG = (N, E, T, l)$ be an instance graph and let $\Pi(IG) = (N', E', T, l')$ be the projection of this instance graph. For all $n' \in N'$: $\forall_{m' \in n'^{\Pi(IG)}_{\bullet}} l'((n', m')) = l'(n') \Rightarrow (\forall_{\substack{n \in N \\ l(n)=n'}} l(n^{IG}_{\bullet}) = n'^{\Pi(IG)}_{\bullet}) \vee \{t_f\} = n'^{\Pi(IG)}_{\bullet}$.

Proof. For the proof we have to consider three cases. In each case we assume that the left hand side of the implication holds and prove that the right hand side also holds.

- Assume $n' = t_s$. Since there does not exists a $n \in N$ such that $l(n) = n'$ the property holds.
- Assume $t_f \in n'^{\Pi(IG)}_{\bullet}$. From Definition 3.1, we know that $l'((n', t_f)) = 1$. Hence, $l'(n') = 1$. Therefore, there is only one $n \in N$ such that $l(n) = n'$. This and $(n', t_f) \in E'$ implies $\forall_{m \in N}\ (n, m) \notin E$. From this we can conclude that $\{t_f\} = n'^{\Pi(IG)}_{\bullet}$ and the property holds.
- Assume that $n' \neq t_s$ and $t_f \notin n'^{\Pi(IG)}_{\bullet}$. We assume that there exists a $n \in N$ such that $l(n) = n'$ and $l(n^{IG}_{\bullet}) \neq n'^{\Pi(IG)}_{\bullet}$ and show a contradiction. Assume $m \in N$, $m \notin n^{IG}_{\bullet}$, $l(m) = m'$ and $m' \in n'^{\Pi(IG)}_{\bullet}$. Since $l'((n', m')) = l'(n')$, we know from Definition 3.1 that there must exist three nodes $n_1, m_1, m_2 \in N$ such that $l(n_1) = n'$ and $l(m_1) = l(m_2) = m'$ and $(n_1, m_1) \in E$ and $(n_1, m_2) \in E$. Since $l(m_1) = l(m_2)$, we know that there exists a path $\langle m_1, \ldots, m_2 \rangle$ (or the other way around). Furthermore, $\langle n_1, m_2 \rangle$ is a path from n_1 to m_2. However, combining the two violates the sixth requirement in Definition 2.4 leading to contradiction. Therefore, $\forall_{\substack{n \in N \\ l(n)=n'}} l(n^{IG}_{\bullet}) = n'^{\Pi(IG)}_{\bullet}$.

□

Note that properties 3.4 through 3.6 also hold for their symmetrical counterparts, i.e., considering input sets rather than output sets.

3.2 Aggregating projections

In Section 3.1, we transformed an instance graph into an instance graph projection. This is an important first step in the aggregation process, since we can aggregate two instance projections, while we cannot aggregate instance graphs. Furthermore, we have derived properties for instance graph projections with respect to the labels of nodes and edges. These properties will prove essential in the process of translating the aggregation graph to an EPC. In this section, we give an algorithm to generate an *aggregation graph*. This graph is a summation over a set of instance graph projections. Therefore, the set of nodes is simply the union set of all instance graph projections.

Definition 3.7. (Aggregation graph) Let T be a set of tasks and let S be a set of instance graphs over T. Let Π be a function giving instance graph projections. We define $\alpha(S) = (N_S, E_S, l_S)$ to be the aggregation graph over S such that:

- $N_S = \bigcup_{\substack{IG \in S \\ \Pi(IG)=(N,E,T,l)}} N$, i.e. the set of nodes.
- $E_S = \bigcup_{\substack{IG \in S \\ \Pi(IG)=(N,E,T,l)}} E$, i.e. the set of edges.
- For all $n \in N_S \cup E_S$ we define $l_S(n) = \sum_{\substack{IG \in S \\ \Pi(IG)=(N,E,T,l)}} l(n)$, where we assume that $l(n) = 0$ if $n \notin N \cup E$.

In essence, the aggregation graph is the straightforward sum over a set of instance graph projections. Note that each instance projection contains the start node t_s and the final node t_f. Therefore, the aggregation graph also contains these unique nodes t_s and t_f in such a way that t_s is the only source node and t_f is the only sink node. The labels represent the number of times a task is executed or a causal relation is present in some instance graph.

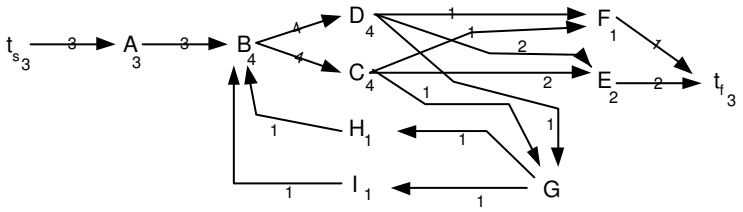

Fig. 7. Labelled aggregation graph.

Figure 7 shows the resulting aggregation graph after aggregating the instance graph projections of the example in Figure 6.

Now we extend some of the results for projections to aggregation graphs (cf. properties 3.3 through 3.6). Note that the labels in the aggregation graphs will later serve as a basis for the conversion of an aggregation graph into an EPC. Using these properties, we will show that no information is lost in the aggregation process.

Property 3.8. (Label properties) Let T be a set of tasks and S a set of instance graphs over T. $\alpha(S) = (N_S, E_S, l_S)$ is the aggregation graph over S.

- $\forall_{(n',m') \in E_S} l_S((n',m')) \leq l_S(n')$, i.e. the labels of all outgoing edges at a node are less or equal to the label of the node.
- $\forall_{n' \in N_S} \left(\sum_{(n',m') \in E_S} l_S((n',m')) \right) \geq l_S(n')$, i.e. the sum over the labels of all outgoing edges at a node is greater or equal to the label of the node.
- $\forall_{(n',m') \in E_S} l_S((n',m')) \leq l_S(m')$, i.e. the labels of all incoming edges at a node are less or equal to the label of the node.
- $\forall_{n' \in N_S} \left(\sum_{(n',m') \in E_S} l_S((n',m')) \right) \geq l_S(m')$, i.e. the sum over the labels of all incoming edges at a node is greater or equal to the label of the node.

Proof. Proving these properties can be done by using Property 3.3 and simple induction over S. □

Similar to Property 3.2 we show that every path in some instance graph is still a path in the aggregation graph. This property reflects the fact that the resulting graph actually represents each instance graph.

Property 3.9. (Aggregation preserves paths) Let T be a set of tasks and S a set of instance graphs over T. $\alpha(S) = (N_S, E_S, l_S)$ is the aggregation graph over S. For all $IG = (N, E, T, l) \in S$ and $\{n_1, \ldots, n_k\} \subseteq N$ such that $\langle n_1, \ldots, n_k \rangle$ is a path in IG: $\{n' \mid \exists_{1 \leq i \leq k} l(n_i) = n'\} \subseteq N_S$ and $\langle l(n_1), \ldots, l(n_k) \rangle$ is a path in $\alpha(S)$.

Proof. Follows directly from Property 3.2 and Definition 3.7. From Property 3.2 we know that if $\{n_1, \ldots, n_k\} \subseteq N$ such that $\langle n_1, \ldots, n_k \rangle$ is a path in some instance graph $IG \in S$ then $\langle l(n_1), \ldots, l(n_k) \rangle$ is a path in $\Pi(IG) = (N', E', T, l')$ and $\{n' \mid \exists_{1 \leq i \leq k} l(n_i) = n'\} \subseteq N'$. From the definition of aggregation we know that $N' \subseteq N_S$ and $E' \subseteq E_S$. Therefore we know that $\{n' \mid \exists_{1 \leq i \leq k} l(n_i) = n'\} \subseteq N_S$ and $\langle l(n_1), \ldots, l(n_k) \rangle$ is a path in $\alpha(S)$. □

Similar to instance graph projections, we show that in an aggregation graph it is possible to find the type of branching at a node. For this, we introduce three properties, similar the properties 3.4 through 3.6. First, we show the aggregative counterpart of Property 3.4, i.e. exclusive choices can be found in aggregation graphs.

Property 3.10. (Exclusive choices can be found) Let T be a set of tasks and S a set of instance graphs over T. $\alpha(S) = (N_S, E_S, l_S)$ is the aggregation graph over S. If for some node $n_S \in N_S$ holds that $\sum_{m \in n_S {}^{\alpha(S)}_{\bullet}} l_S((n_S, m)) = l_S(n_S)$ then for all $IG \in S$, where $IG = (N, E, T, l)$ holds that $\forall_{\substack{n \in N \\ l(n) = n_S}} |n {}^{IG}_{\bullet}| \leq 1$.

Proof. If $n_S = t_s$ then the property holds, because $t_s \notin l(N)$. Therefore, assume $n_S \neq t_s$. Furthermore, assume that there exists a $IG \in S$, where $IG = (N, E, T, l)$ and $n \in N$ with $l(n) = n_S$, such that $|n {}^{IG}_{\bullet}| > 1$. We prove by contradiction that this is not possible. $\Pi(IG) = (N', E', T, l')$ is the projection of IG. From Property 3.5 we know that $\sum_{m \in n_S {}^{\Pi(IG)}_{\bullet}} l'((n_S, m)) > l'(n_S)$. Furthermore, from Property 3.3 we know that for all instance graphs IG', $\sum_{m \in n_S {}^{\Pi(IG')}_{\bullet}} l'((n_S, m)) \geq l'(n_S)$. Using Definition 3.7 we can now conclude that $\sum_{m \in n_S {}^{\alpha(S)}_{\bullet}} l_S((n_S, m)) > l_S(n_S)$. This yields contradiction. □

Besides exclusive choices, it is possible to find true parallelism as well. Property 3.11 shows this for the aggregation graph, similar to Property 3.6 for projection graphs.

Property 3.11. (True parallelism can be found) Let T be a set of tasks and S a set of instance graphs over T. $\alpha(S) = (N_S, E_S, l_S)$ is the aggregation graph over S. For all $n' \in N_S$ and $IG \in S$, where $IG = (N, E, T, l)$: $\left(\forall_{p' \in n' {}^{\alpha(S)}_{\bullet}} l_S((n', p')) = l_S(n') \right) \Rightarrow \left(\left(\forall_{\substack{n \in N \\ l(n) = n'}} l(n {}^{IG}_{\bullet}) = n' {}^{\alpha(S)}_{\bullet} \right) \vee \{t_f\} = n' {}^{\alpha(S)}_{\bullet} \right)$

Proof. We prove this property by induction over the number of instance graphs in S. Assume that S contains only one instance graph IG. Since $\alpha(S) = \Pi(IG)$ in that case, we have proven in Property 3.6 that this property holds. Assume that we have a set of instances S such that for $\alpha(S)$ the property holds. We will show that adding one instance $IG = (N, E, T, l)$ to S, generating a new set S', does not violate the property, i.e., we need to prove that the property holds for $S' = S \cup \{IG\}$ assuming that it holds for S. Consider $n' \in N_{S'}$. If $n' \notin N_S$, then $n' \in l(N)$ and IG is the first instance graph containing n'. Hence we can apply Property 3.6. If $n' \in N_S$, we need to consider two cases:

1. The left-hand side of the implication, i.e., $\forall_{m' \in n'^{\alpha(S)}_{\bullet}} l_S((n', m')) = l_S(n')$ holds for S. If $n' \notin l(N)$, it is easily seen that the property is not violated. Therefore, assume $n' \in l(N)$. Adding instance IG to S can have two results. First, $\forall_{m' \in n'^{\alpha(S')}_{\bullet}} l_S((n', m')) = l_S(n')$ holds. In this case, we know that ${n'}^{\alpha(S)}_{\bullet} = {n'}^{\Pi(IG)}_{\bullet}$ and $\forall_{m' \in n'^{\Pi(IG)}_{\bullet}} l'((n', m')) = l'(n')$ holds. Property 3.6 shows that therefore, $\left(\forall_{\substack{n \in N \\ l(n) = n'}} l(n^{IG}_{\bullet}) = {n'}^{\alpha(S)}_{\bullet}\right) \vee \{t_f\} = {n'}^{\alpha(S)}_{\bullet}$ holds for IG. Combining our assumptions we can conclude that the right hand side of the implication still holds for all instances. Second, $\forall_{m' \in n'^{\alpha(S')}_{\bullet}} l'((n', m')) = l'(n')$ does not hold. This is a trivial case, since now the implication still holds.
2. The left-hand side of the implication does not hold of S. This can only be the case if there is some $m' \in N_S$, such that $l_S((n', m')) < l_S(n')$. Using Property 3.3 it is easily seen that adding an instance will never result in $l_S((n', m')) = l_S(n')$. Therefore, the implication still holds.

This concludes the proof by induction. □

Note that for instance graph projections, we also defined the multi-choice as a branching type (recall Property 3.5). Obviously, this is the case when the type of branching is neither an exclusive choice, nor a parallel split. Therefore, there is no need to derive this property for aggregation graphs separately. Note that properties 3.10 and 3.11 also hold for their symmetric counter parts (input sets rather than output sets).

After aggregating a set of instance graphs into an aggregation graph the result has to be visualized. Although the aggregation graph as shown in Figure 7 contains all the information needed, it is still hard for a human to understand. Therefore, we translate the graph into a format that is more intuitive. The modeling language we use here is called *Event Driven Process Chains* (EPCs). The choice for EPCs is based on the fact that it is a well known concept, and it allows for all necessary routing constructs.

4 Transformation into EPCs

EPCs are an intuitive modeling language to model business processes introduced by Keller, Nüttgens and Scheer in 1992 [16]. The language was initially not

intended to be a formal specification of a business process [3]. An EPC consists of three main elements. Combined, these elements define the flow of a business process as a chain of events. The elements used are:

Functions
: The basic building blocks are 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. Functions are linked by events.

Connectors
: Connectors can be used to connect functions and events. This way, the flow of control is specified. There are three types of connectors: $\wedge$ (and), $\times$ (xor) and $\vee$ (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.

In this section, we will transform an aggregation graph as defined in Definition 3.7 into an EPC. In the labelled aggregation graph, the labels of the nodes represent the number of times a task is executed in the log. The labels of the edges represent the number of times an edge is taken in some instance. It is easily seen that there are three possibilities for the incoming as well as for the outgoing edges of a node.

1. The *sum* over the labels of all incoming (outgoing) edges of some node equals the label of the node. (Recall Property 3.10)
2. *All* the labels of *all* incoming (outgoing) edges of some node equal the label of the node. (Recall Property 3.11)
3. *Not all* the labels of all incoming (outgoing) edges of some node equal the label of the node, and the *sum* over the labels is *greater* than the label of the node.

We use these three points to determine the type of connectors we use in the transformation process to EPCs. These points are symmetrical for ingoing and outgoing edges at all nodes. Therefore, if we talk about incoming edges, we also mean the symmetrical case for outgoing edges. Using Property 3.10 and Property 3.11, transforming an aggregation graph into an EPC is straightforward and we only give a sketch of the translation algorithm.

Definition 4.1. (Translation to EPC) Let S be a set of instance graphs and let $AG = (N_S, E_S, l_S)$ be the labelled aggregation graph over S. We define the aggregated EPC $EPC = (F_{EPC}, E_{EPC}, C_{EPC}, A_{EPC}, l_{EPC})$ in such a way that:

- $F_{EPC} = N_S \setminus \{t_f\}$ is the set of functions,
- $E_{EPC} = \{e_n \mid n \in N_S\}$ is the set of events,
- C_{EPC} is the set of connectors,
- A_{EPC} is the set of edges,
- $l_{EPC} : C_{EPC} \to \{and, xor, or\}$ is a function typifying the connectors.

The construction of the set of connectors, the set of edges and the labeling function of this EPC is defined in Figure 8.

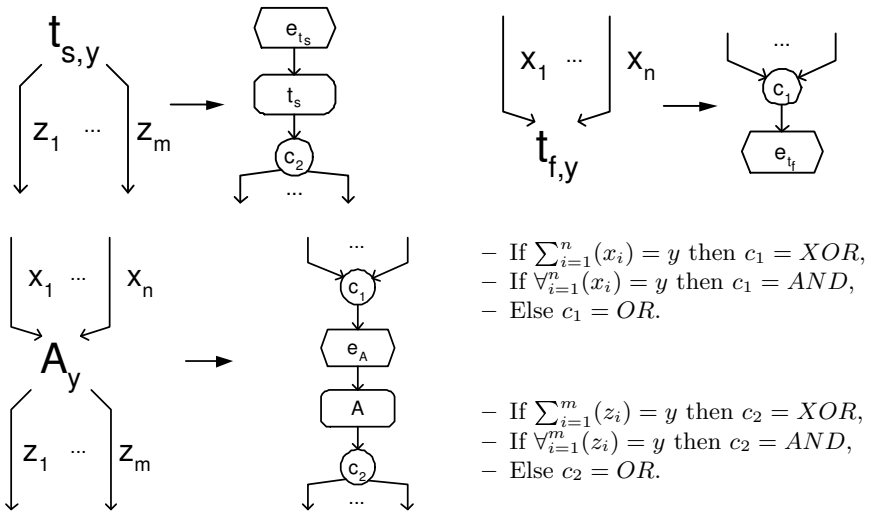

Fig. 8. Transformation rules from graph to EPC (top:start/final nodes, bottom: "normal" nodes).

Figure 8 defines the transformation of the aggregation graph into an EPC. These rules are rather straightforward. First, each node n from the aggregation graph is translated into a combination of an event e_n and a function n and an edge is added in between. Then, the incoming and outgoing edges at all nodes are transformed into connectors. Note that for connectors that have only one ingoing or outgoing arc, the type is ambiguous, since both *and* and *xor* can be applied. Therefore, we remove all connectors that have only one ingoing and one outgoing edge, and replace them with a normal edge. For the initial and final nodes t_s and t_f, special rules are made. In Figure 9 the result of this procedure is shown for the aggregation graph of Figure 7.

An important property of the resulting EPC is that it is indeed capable of reproducing all original instance graphs. This can easily be seen from the fact that an EPC is merely a syntactically different representation of a labelled aggregation graph. In Property 3.9 we have show that each path in an instance

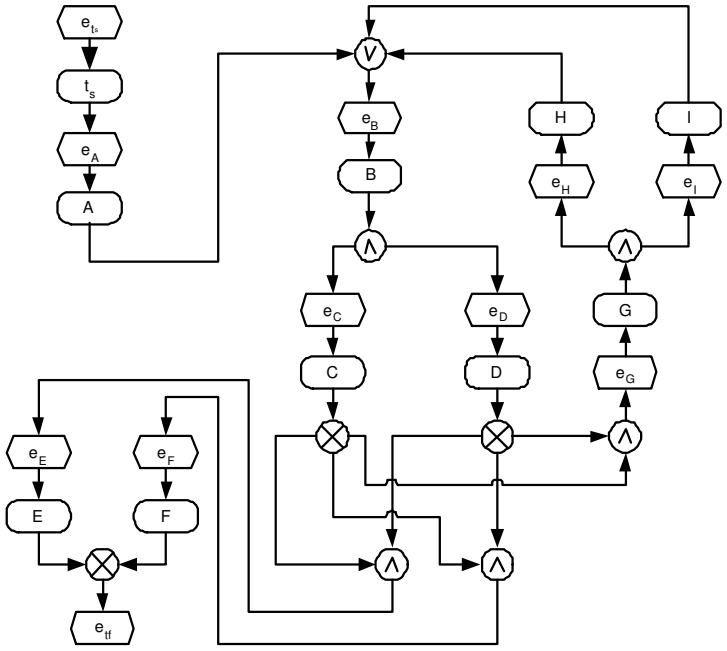

Fig. 9. Aggregated EPC.

graph is also a path in an aggregation graph. Therefore, it is also a path in an aggregated EPC. In properties 3.10 and 3.11 we have shown that the types of splits and joins can be found in an aggregation graph. Based on these properties, the transformation into an EPC is made. Therefore, it is clear that for the aggregated EPC the same properties hold. All the instance graphs that we started with in the first place are executions of the EPC. However, the EPC can allow for more behavior, especially when *or*-connectors are used.

Even though we have shown that the aggregated EPC does represent all instances it is still not the best result possible with this approach. The reason for this is the fact that *or*-connectors appear in the EPC. Using the original set of instances, it is possible that we can be more specific with respect to these connectors. In order to be able to express this, we will use Petri nets. In Section 5 we will first introduce a naive translation algorithm. Then, in Section 6 we will restrict the behavior of the Petri net generated to the original set of instance graphs.

5 Transformation into Petri nets

In this section, we will transform an aggregated EPC as defined in Definition 4.1 into a Petri net. The translation will be naive, which means that we translate *or*-connectors into a large set of transitions. The result of this transformation

process will be a Petri net with two types of transitions. The first type of transitions is formed by the so-called visible transitions. These transitions correspond to the functions in the original EPC. The second type of transitions is formed by the set of invisible transitions. These transitions correspond to connectors in the EPC. We use a variant of the classic Petri-net model, namely Place/Transition nets. For an elaborate introduction to Petri nets, the reader is referred to [10, 19, 20]. In this section, we will not consider soundness [1, 8] of the Petri net. However, in Section 6 we will show that it is easy to prove that the restricted Petri net is relaxed sound [8].

Now we present a sketch of the actual transformation from an EPC into a Petri net. Just as for the translation of the aggregation graph onto the aggregated EPC, the translation of the EPC onto the Petri net is straightforward. Note that we do not consider subtle semantical problems of EPCs as discussed in [3]. Instead we use the more pragmatic approach described in [8].

To explicitly state the difference between transitions that correspond to functions and transitions that correspond to connectors, we will split up the set of transitions T into T_v (visible transitions) and T_{iv} (invisible transitions). However, the resulting net can be interpreted as an ordinary Place/Transition net $(P, T_v \cup T_{iv}, F)$.

Definition 5.1. (Sketch of Transformation Algorithm)
Let $EPC = (F_{EPC}, E_{EPC}, C_{EPC}, A_{EPC}, l_{EPC})$ be an EPC. We define the Petri net transformation of this EPC as $PN = (P, T_v \cup T_{iv}, F)$. The construction of this Petri net is done in the following way:[2]

1. Remove all events from the EPC except e_{t_s} and e_{t_f}, and rebuild arcs. For all events $e \in E \setminus \{e_{t_s}, e_{t_f}\}$ we find $n_1, n_2 \in F_{EPC} \cup C_{EPC}$ such that $\{(n_1, e), (e, n_2)\} \subseteq A_{EPC}$ and we change A_{EPC} in the following way: $A_{EPC} := (A_{EPC} \cup \{(n_1, n_2)\}) \setminus \{(n_1, e), (e, n_2)\}$.
2. Create a visible transition for each function except the function t_s. $T_v := F_{EPC} \setminus \{t_s\}$.
3. Create an invisible transition for the function t_s. $T_{iv} := \{t_s\}$.
4. Create a place for each arc. $P := \{p_a \mid a \in A_{EPC}\}$. Note that here A_{EPC} is the set created after the changes in the first step.
5. For each *and* connector in the EPC, create an invisible transition. $T_{iv} := T_{iv} \cup \{t_c \mid c \in C_{EPC} \wedge l_{EPC}(c) = and\}$.
6. For each *xor* connector in the EPC, create a number of invisible transitions. $T_{iv} := T_{iv} \cup \{t_{c,a,b} \mid c \in C_{EPC} \wedge l_{EPC}(c) = xor \ \wedge \ (a, c) \in A_{EPC} \wedge (c, b) \in A_{EPC}\}$.
7. For each *or* connector in the EPC, create a number of invisible transitions. $T_{iv} := T_{iv} \cup \{t_{c,V,W} \mid c \in C_{EPC} \wedge l_{EPC}(c) = or \ \wedge \ V \subseteq \{a | (a, c) \in A_{EPC}\} \ \wedge \ V \neq \emptyset \wedge \ W \subseteq \{b | (c, b) \in A_{EPC}\} \ \wedge \ W \neq \emptyset\}$.

[2] Note that in this paper we only give a sketch of the algorithm. Note that the full algorithm has been implemented in the ProM framework described in Section 7.

8. Update all connections, i.e., construct F based on the set of places P, the set of transitions $T_v \cup T_{iv}$ and the original connections in A_{EPC}. (A complete formalization of the algorithm is beyond the scope of this paper.)

The sketch of the algorithm presented in Definition 5.1 shows the intuition behind the translation. Events are removed from the model since they were artificially added in the transformation from an instance graph to an EPC and do not provide information. This does not hold for the initial and final event, since they are transformed into places. Functions are transformed into visible transitions and these transitions are connected through places. An exception is made for the function t_s. This function is translated into a invisible transition, since it does not represent a task in the log file. Furthermore, it is easily seen that the resulting Petri net has exactly one place with no incoming arcs and exactly one place without outgoing arcs.

The Petri net we now have generated will allow for all instances to execute for the same reasons the EPC does. However, it allows for more behavior than just these instances. The reason for this is in the *or* split and join connectors. An *or* split (join) with k outputs (inputs) allows for $2^k - 1$ transitions but not every transition will be possible. Moreover, specific splits will need to be matched to specific joins. Therefore, in Section 6, we will restrict the behavior of the net to the set of instance graphs it was generated from, by removing some invisible transitions.

6 Restricting Petri nets to instances

In this section, we will restrict the behavior of an aggregated Petri net by looking at the instances it was generated from. These instances are mapped onto the Petri net by playing the "token game". Each invisible transition is included, if it had to be fired in any of the instances. If this isn't the case, it will be removed. In order to decide whether an invisible transition should be removed, we need to define the in- and out-sets of invisible transitions. These sets represent the *and*-join/split that a transition represents.

Definition 6.1. (In/Out sets) Let $PN = (P, T_v \cup T_{iv}, F)$ be a Petri net with visible and invisible transitions. We define $in : T_{iv} \rightarrow \mathcal{P}(T_v)$ as the function giving the in-set and $out : T_{iv} \rightarrow \mathcal{P}(T_v)$ as a function giving the out-set of an invisible transition, where (note the recursion/fixpoint)

- $in(t_{iv}) := \{t \in T_v \mid t\bullet \cup \bullet t_{iv} \neq \emptyset\} \cup \bigcup_{\substack{t' \in T_{iv} \\ t'\bullet \cup \bullet t_{iv} \neq \emptyset}} in(t')$
- $out(t_{iv}) := \{t \in T_v \mid \bullet t \cup t_{iv}\bullet \neq \emptyset\} \cup \bigcup_{\substack{t' \in T_{iv} \\ \bullet t' \cup t_{iv}\bullet \neq \emptyset}} out(t')$

Informally the in and out-set of a transition can be explained as follows. If in some execution sequence transition t fires, then this means that before, all the transitions in the in-set have fired and as a result all the transitions in the out-set will eventually become enabled. However, after the firing of the transitions in the in set and before firing the transitions in the out set, there can be firings of

other invisible transitions. More about these semantics can be found in [2], where the invisible transitions are referred to as "silent transitions". Since the invisible transitions introduced in the conversion of an EPC belong to a connector that is either a split or a join, either the corresponding in-set or out-set has cardinality one respectively.

Using in and out sets, we can restrict the Petri net.

Definition 6.2. (Restricted Petri net) Let S be a set of instance graphs and let $EPC = (F_{EPC}, E_{EPC}, C_{EPC}, A_{EPC}, l_{EPC})$ be the aggregated EPC over S (Definition 4.1) and let $PN = (P, T_v \cup T_{iv}, F)$ be the Petri net translation of EPC (Definition 5.1). We define the restricted Petri net as $PN_r = (P, T_v \cup T'_{iv}, F')$ as follows:

- $T'_{iv} = \{t \in T_{iv} \mid \exists_{(N,E,T,l)\in S} \exists_{\substack{N_1 \subseteq N \\ l(N_1)=in(t)}} \exists_{\substack{N_2 \subseteq N \\ l(N_2)=out(t)}} \wedge \forall_{\substack{n_1 \in N_1 \\ n_2 \in N_2}} (n_1, n_2) \in E\}$,
- $F' = F \cap ((P \cup T'_{iv} \cup T_v) \times (P \cup T'_{iv} \cup T_v))$.

The restrictions from Definition 6.2 remove invisible transitions that do not appear in any instance graph. See Figure 10 for PN_r based on the running example. Note that these transitions will always be translations of *or*-connectors. The reason for this is that for each *and* connector, there is only one invisible transition and Property 3.11 shows that this transition is present in at least one instance. For *xor* connectors, there are multiple transitions, but from Property 3.10 we know that they all appear in at least one instance. For *or*-connectors, it is clear that we do not remove too many transitions. In other words, we do not introduce places without input or output arcs. Furthermore, by definition, every invisible transition that is not removed is fired in some instance.

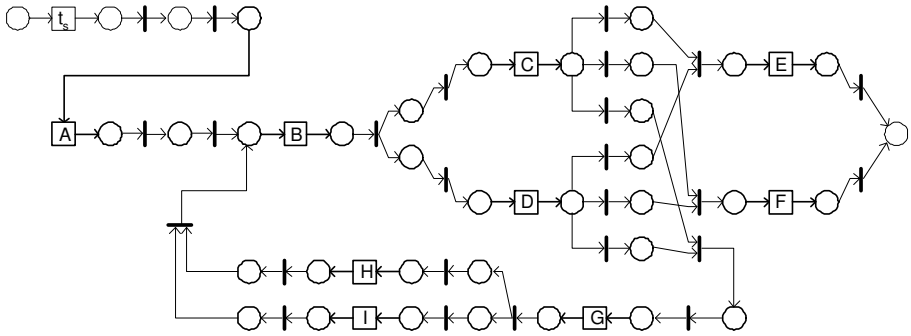

Fig. 10. Aggregated Petri net based on Figure 7 restricted to the instances.

Without proof we mention that the resulting Petri net is relaxed sound [8], i.e., for every transition there is a sound firing sequence leading from the initial state (token in source place) to the final state (token in sink place). Note that the resulting Petri net is not (strongly) sound [1] because of or-joins that do not follow or-splits. By only looking at the sound firing sequences (i.e., restricting the behavior based on relaxed soundness, cf. [8]), we get an *executable semantics* for the aggregated EPCs and therefore also *for the aggregation graphs*. This was

exactly the goal of our multi-phase mining approach. Based on event logs we derive an executable process model.

7 Tool support

Process mining has many applications in industry. However, for the different techniques to be useful in industry, good tool support is essential. The work presented in this paper is supported by two tools. In this section, we briefly discuss both tools. The main difference between the tools is that one of them is freely available, while the second is a commercial tool. Moreover, the latter tool focuses on performance analysis and the generation of instances rather than the construction of an aggregated process model and it requires deep knowledge of the process under consideration.

7.1 The ProM-framework

The ProM-framework is a framework for the implementation of process mining techniques. The tool has been developed at the Technical University of Eindhoven and is freely available from *www.processmining.org*. The tool provides support for a large number of process mining plug-ins at different levels. It contains for example a component for the mining of social networks. It provides a standard way for working with different models such as Petri nets and EPCs as well as components for the automatic layout of these models. The algorithms presented in this paper are available in the ProM framework under the name "Multi Phase mining plug-in". Figure 11 shows a screenshot of the Prom-framework, showing the same EPC as Figure 9, only with a different layout.

7.2 ARIS Process Performance Monitor

The ARIS Process Performance Monitor (PPM) of IDS Scheer is a commercial tool for business process monitoring. The idea behind ARIS PPM is very similar to the algorithms presented in this paper. In ARIS PPM, every case is represented as a so-called *instance-EPC* (cf. instance graph). These instance-EPCs can be aggregated into an aggregated EPC in a similar way as described here. However, this is not the main feature of this tool and the rules used are quite different. The main focus of ARIS PPM is on the analysis of the performance of processes. It can make nice graphical representations of all kinds of properties on a case level, but also on an aggregated level. Figures 3 and 4 show screenshots of the ARIS PPM client.

ARIS PPM can be considered to be the primary motivation for the research presented in this paper and we expect that some of the ideas presented in this paper will be adopted by ARIS PPM. In fact, IDS Scheer is currently in the process of adding some functionality of the ProM-framework to ARIS PPM.

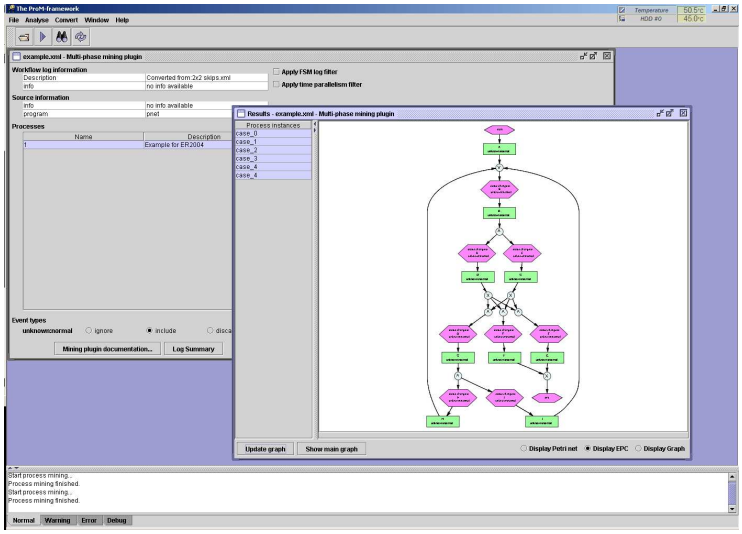

Fig. 11. The ProM-framework.

8 Related work

Existing approaches for mining the process perspective [4–7, 14, 18, 22] are fundamentally different from the approach presented in this paper, i.e., they only use one mining phase (other than things like user-assisted process mining). It is impossible to give an overview of the work op process mining. Instead we refer to [4, 5] which present an elaborate overview of process mining literature.

For the readers familiar with Petri net theory, it is important to discuss the relation between this work and the work on *regions* [13] and the work on *runs* [9]. Process mining is quite different from constructing a Petri net on the basis of regions because our notion of completeness is much weaker than the typical assumption when using regions (i.e., a complete transition system) [13]. For process mining one assumes that the log contains only a fraction of the possible behavior. Furthermore, it is important to mention that instance graphs can be seen as runs [9] also known as occurrence nets or partially ordered Petri net processes. However, runs are typically generated from a Petri net rather than used to construct a Petri net. Another problem is that approaches such as the one described in [9] typically link places in the run to places in the original Petri net. When mining event logs there is no information about places. Nevertheless it is interesting to further investigate the relations between multi-phase mining and synthesis problems in Petri nets.

As indicated in the introduction, a short version of this paper has been submitted to the International Conference on Cooperative Information Systems (CoopIS05) [12]. However, in [12] the formal proofs and the translation to Petri nets are missing.

9 Conclusion

In this paper, we introduced an algorithm to aggregate a number of instance graphs. An instance graph can be seen as the representation of an execution of a business process for a single case. Such an execution may have concurrent paths, but cannot contain any choices. Commercial products such as ARIS PPM show that it is interesting to inspect instance graphs. However, at the same time there is the desire to aggregate sets of instance graphs. In this paper, we have defined an algorithm to support this. From a set of instance graphs, we first build an aggregation graph. We provided translations for this graph into the well-known modeling language of EPCs. One of the most interesting results of our efforts is that the resulting EPC actually represents all instances. Moreover, we can prove that for the resulting model a deadlock-free, executable semantics exists for the model. Via the translation to EPCs, we can also aggregate instance graphs into a Petri net. The translation allows for too much behavior. However, by restricting the Petri net or assuming relaxed soundness as a control semantics, the "abundance of behavior" can be removed.

Acknowledgements

The authors would like to thank IDS Scheer for supporting the mining research and for providing their ARIS Process Performance Monitor (ARIS PPM).

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst, A.K. Alves de Medeiros, and A.J.M.M. Weijters. Genetic Process Mining. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (to appear)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2005.
3. W.M.P. van der Aalst, J. Desel, and E. Kindler. On the Semantics of EPCs: A Vicious Circle. In M. Nüttgens and F.J. Rump, editors, *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–80, Trier, Germany, November 2002. Gesellschaft für Informatik, Bonn.
4. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
5. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
6. R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*, pages 469–483, 1998.
7. J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

8. J. Dehnert and W.M.P. van der Aalst. Bridging the Gap Between Business Models and Workflow Specifications. *International Journal of Cooperative Information Systems*, 13(3):289–332, 2004.
9. J. Desel. Validation of Process Models by Construction of Process Nets. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 110–128. Springer-Verlag, Berlin, 2000.
10. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
11. B.F. van Dongen and W.M.P. van der Aalst. Multi-Phase Process Mining: Building Instance Graphs. In P. Atzeni, W. Chu, H. Lu, S. Zhou, and T.W. Ling, editors, *International Conference on Conceptual Modeling (ER 2004)*, volume 3288 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, Berlin, 2004.
12. B.F. van Dongen and W.M.P. van der Aalst. Multi-phase Process mining: Aggregating Instance Graphs. In *submitted to COOPIS 2005*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2005.
13. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.
14. J. Herbst. A Machine Learning Approach to Workflow Management. In *Proceedings 11th European Conference on Machine Learning*, volume 1810 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag, Berlin, 2000.
15. IDS Scheer. ARIS Process Performance Manager (ARIS PPM): Measure, Analyze and Optimize Your Business Process Performance (whitepaper). IDS Scheer, Saarbruecken, Gemany, http://www.ids-scheer.com, 2002.
16. G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.
17. G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation*. Addison-Wesley, Reading MA, 1998.
18. A.K.A. de Medeiros, W.M.P. van der Aalst, and A.J.M.M. Weijters. Workflow Mining: Current Status and Future Directions. In R. Meersman, Z. Tari, and D.C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 389–406. Springer-Verlag, Berlin, 2003.
19. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
20. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
21. A.W. Scheer. *Business Process Engineering, Reference Models for Industrial Enterprises*. Springer-Verlag, Berlin, 1994.
22. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.