Multi-phase Process mining: Building Instance Graphs

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. Deploying process-driven information systems is a time-consuming and error-prone task. *Process mining* attempts to improve this by automatically generating a process model from event-based data. Existing techniques try to generate a complete process model from the data acquired. However, unless this model is the ultimate goal of mining, such a model is not always required. Instead, a good visualization of each individual process instance can be enough. From these individual instances, an overall model can then be generated if required. In this paper, we present an approach which constructs an *instance graph* for each individual process instance, based on information in the entire data set. The results are represented in terms of Event-driven Process Chains (EPCs). This representation is used to connect our process mining to a widely used commercial tool for the visualization and analysis of instance EPCs.

Keywords: Process mining, Event-driven process chains, Workflow management, Business Process Management.

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. [13, 14, 19]), 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.

To be able to calculate basic performance characteristics, most systems have their own built-in module. For the calculation of basic characteristics such as the average flow time of a case, no model of the process is required. However, for more complicated characteristics, such as the average time it takes to transfer work from one person to the other, some notion of causality between tasks is required. This notion of causality is provided by the original model of the process, but deviations in execution can interfere with causalities specified there. Therefore, in

this paper, we present a way of defining certain causal relations in a transactional system. We do so without using the process definition from the system, but only looking at a so called process log. Such a process log contains information about the processes as they actually take place in a transactional system. Most systems can provide this information in some form and the techniques used to infer relations between tasks in such a log is called *process mining*.

The problem tackled in this paper has been inspired by the software package *ARIS PPM* (Process Performance Monitor) [12] developed by IDS Scheer. ARIS PPM allows for the visualization, aggregation, and analysis of process instances expressed in terms of *instance EPCs* (i-EPCs). An instance EPC describes the the control-flow of a case, i.e., a single process instance. Unlike a trace (i.e., a sequence of events) an instance EPC provides a graphical representation describing the causal relations. In case of parallelism, there may be different traces having the same instance EPC. Note that in the presence of parallelism, two subsequent events do not have to be causally related. ARIS PPM exploits the advantages of having instance EPCs rather than traces to provide additional management information, i.e., instances can be visualized and aggregated in various ways. In order to do this, IDS Scheer has developed a number of adapters, e.g., there is an adapter to extract instance EPCs from SAP R/3. Unfortunately, these adapters can only create instance EPCs if the actual process is known. For example, the workflow management system Staffware can be used to export Staffware audit trails to ARIS PPM (Staffware SPM, cf. [20]) by taking projections of the Staffware process model. As a result, it is very time consuming to build adapters. Moreover, the approaches used only work in environments where there are explicit process models available.

In this paper, we do not focus on the visualization, aggregation, and analysis of process instances expressed in terms of instance EPC or some other notation capturing parallelism and causality. Instead we focus on the construction of *instance graphs*. An instance graph can be seen as an abstraction of the instance EPCs used by ARIS PPM. In fact, we will show a mapping of instance graphs onto instance EPCs. Instance graphs also correspond to a specific class of Petri nets known as *marked graphs* [17], *T-systems* [9] or *partially ordered runs* [8, 10]. Tools like VIPTool allow for the construction of partially ordered runs given an ordinary Petri net and then use these instance graphs for analysis purposes. In our approach we do not construct instance graphs from a known Petri net but from an event log. This enhances the applicability of commercial tools such as ARIS PPM and the theoretical results presented in [8, 10]. The mapping from instance graphs to these Petri nets is not given here. However, it will become clear that such a mapping is trivial.

In the remainder of this paper, we will first describe a common format to store process logs in. Then, in Section 3 we will give an algorithm to infer causality at an instance level, i.e. a model is built for each individual case. In Section 4 we will provide a translation of these models to EPCs. Section 5 shows a concrete example and demonstrates the link to ARIS PPM. Section 6 discusses related work followed by some concluding remarks.

2 Preliminaries

This section contains most definitions used in the process of mining for instance graphs. The structure of this section is as follows. Subsection 2.1 defines a process log in a standard format. Subsection 2.2 defines the model for one instance.

2.1 Process Logs

Information systems typically log all kinds of events. Unfortunately, most systems use a specific format. Therefore, we propose an XML format for storing event logs. The basic assumption is that the log contains information about specific *tasks* executed for specific *cases* (i.e., process instances). Note that unlike ARIS PPM we do not assume any knowledge of the underlying process. Experience with several software products (e.g., Staffware, InConcert, MQSeries Workflow, FLOWer, etc.) and organization-specific systems (e.g., Rijkswaterstaat, CJIB, and several hospitals) show that these assumptions are justified.

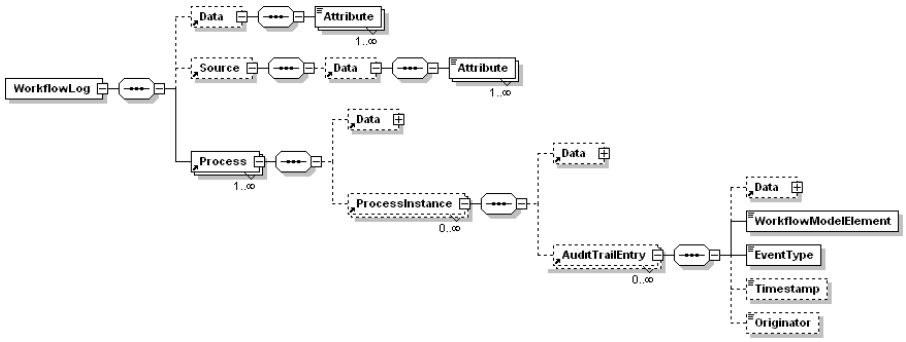

Fig. 1. XML schema for process logs.

Figure 1 shows the schema definition of the XML format. This format is supported by our tools, and mappings from several commercial systems are available. The format allows for logging multiple processes in one XML file (cf. element "Process"). Within each process there may be multiple process instances (cf. element "ProcessInstance"). Each "ProcessInstance" element is composed of "AuditTrailEntry" elements. Instead of "AuditTrailEntry" we will also use the terms "log entry" or "event". An "AuditTrailEntry" element corresponds to a single event and refers to a "WorkflowModelElement" and an "EventType". A "WorkflowModelElement" may refer to a single task or a subprocess. The "EventType" is used to indicate the type of event. Typical events are: "schedule" (i.e., a task becomes enabled for a specific instance), "assign" (i.e., a task instance is assigned to a user), "start" (the beginning of a task instance), " complete" (the completion of a task instance). In total, we identify 12 events. When building an adapter for a specific system, the system-specific events are mapped on these 12 generic events.

As Figure 1 shows the "WorkflowModelElement" and "EventType" are mandatory for each "AuditTrailEntry". There are three optional elements "Data",

"Timestamp", and "Originator". The "Data" element can be used to store data related to the event of the case (e.g., the amount of money involved in the transaction). The "Timestamp" element is important for calculating performance metrics like flow time, service times, service levels, utilization, etc. The "Originator" refers to the actor (i.e., user or organization) performing the event. The latter is useful for analyzing organizational and social aspects. Although each element is vital for the practical applicability of process mining, we focus on the "WorkflowModelElement" element. In other words, we abstract from the "EventType", "Data", "Timestamp", and "Originator" elements. However, our approach can easily be extended to incorporate these aspects. In fact, our tools deal with these additional elements. However, for the sake of readability, in this paper events are identified by the task and case (i.e., process instance) involved.

case identifier	task identifier
case 1	task S
case 2	task S
case 1	task A
case 1	task B
case 2	task B
case 2	task A

Table 1. A process log.

Table 1 shows an example of a small log after abstracting from all elements except for the "WorkflowModelElement" element (i.e., task identifier). The log shows two cases. For each case three tasks are executed. Case 1 can be described by the sequence SAB and case 2 can be described by the sequence SBA. In the remainder we will describe process instances as sequences of tasks where each element in the sequence refers to a "WorkflowModelElement" element. A process log is represented as a bag (i.e., multiset) of process instances.

Definition 2.1. (Process Instance, Process Log) Let T be a set of log entries, i.e., references to tasks. Let T^+ define the set of sequences of log entries with length at least 1. We call $\sigma \in T^+$ a process instance (i.e., case) and $W \in T^+ \rightarrow \mathbb{N}$ a process log.

If $\sigma = t_1 t_2 \ldots t_n \in T^+$ is a process instance of length n, then each element t_i corresponds to "AuditTrailEntry" element in Figure 1. However, since we abstract from timestamps, event types, etc., one can think of t_i as a reference to a task. $|\sigma| = n$ denotes the length of the process instance and σ_i the i-th element. We assume process instances to be of finite length. $W \in T^+ \rightarrow \mathbb{N}$ denotes a bag, i.e., a multiset of process instances. $W(\sigma)$ is the number of times a process instance of the form σ appears in the log. The total number of instances in a bag is finite. Since W is a bag, we use the normal set operators where convenient. For example, we use $\sigma \in W$ as a shorthand notation for $W(\sigma) > 0$.

2.2 Instance Nets

After defining a process log, we now define an instance net. An instance net is a model of one instance. Since we are dealing with an instance that has been

executed in the past, it makes sense to define an instance net in such a way that no choices have to be made. As a consequence of this, no loops will appear in an instance net. For readers familiar with Petri nets it is easy to see that instance nets correspond to "runs" (also referred to as occurrence nets) [8].

Since events that appear multiple times in a process instance have to be duplicated in an instance net, we define an instance domain. The instance domain will be used as a basis for generating instance nets.

Definition 2.2. (Instance domain) Let σ be a process instance such that $\sigma = t_1 t_2 \dots t_n \in T^+$, i.e., $|\sigma| = n$. We define $D_\sigma = \{1 \dots n\}$ as the domain of σ.

Using the domain of an instance, we can link each log entry in the process instance to a specific task, i.e., $i \in D_\sigma$ can be used to represent the i-th element in σ. In an instance net, the instance σ is extended with some ordering relation $\dashv_\sigma$ to reflect some causal relation.

Definition 2.3. (Instance net) Let $N = (\sigma, \dashv_\sigma)$ such that σ is a process instance. Let D_σ be the domain of σ and let $\dashv_\sigma$ be an ordering on D_σ such that:
- $\dashv_\sigma$ is irreflexive, asymmetric and acyclic,
- $\forall_{i,j \in D_\sigma}(i < j \Rightarrow j \not\dashv_\sigma i)$,
- $\forall i, j \in D_\sigma(i \dashv_\sigma j \Rightarrow \nexists_{k \in D_\sigma}(i \dashv_\sigma k \wedge k \dashv_\sigma^+ j)$, where $\dashv_\sigma^+$ is the smallest relation satisfying: $i \dashv_\sigma^+ j$ if and only if $i \dashv_\sigma j$ or $\exists_k(i \dashv_\sigma k \wedge k \dashv_\sigma^+ j)$
- $\forall i, j \in D_\sigma(t_i = t_j \Rightarrow (i \dashv_\sigma^+ j) \vee (j \dashv_\sigma^+ i))$

We call N an instance net.

The definition of an instance net given here is rather flexible, since it is defined only as a set of entries from the log and an ordering on that set. An important feature of this ordering is that if $i \dashv j$ then there is no set $\{k_1, k_2, \dots, k_n\}$ such that $i \dashv k_1, k_1 \dashv k_2, \dots, k_n \dashv j$. Since the set of entries is given as a log, and an instance mapping can be inferred for each instance based on textual properties, we only need to define the ordering relation based on the given log. In Section 3.1 it is shown how this can be done. In Section 4 we show how to translate an instance net to a model in a particular language (i.e., instance EPCs).

3 Mining Instance Graphs

As seen in Definition 2.3, an instance net consists of two parts. First, it requires a sequence of events $\sigma \in T^+$ as they appear in a specific instance. Second, an ordering $\dashv$ on the domain of σ is required. In this section, we will provide a method that infers such an ordering relation on T using the whole log. Furthermore, we will present an algorithm to generate *instance graphs* from these *instance nets*.

3.1 Creating Instance Nets

Definition 3.1. (Causal ordering) Let W be a process log over a set of log entries T, i.e., $W \in T^+ \rightarrow \mathbb{N}$. Let $b \in T$ and $c \in T$ be two log entries. We define a causal ordering $\rightarrow_W$ on W in the following way:

- $b >_W c$ if and only if there is an instance σ and $i \in D_\sigma \setminus \{|\sigma|\}$ such that $\sigma \in W$ and $\sigma_i = b$ and $\sigma_{i+1} = c$,
- $b \triangle_W c$ if and only if there is an instance σ and $i \in D_\sigma \setminus \{|\sigma| - 1, |\sigma|\}$ such that $\sigma \in W$ and $\sigma_i = \sigma_{i+2} = b$ and $\sigma_{i+1} = c$ and $b \neq c$ and not $b >_W b$,
- $b \rightarrow_W c$ if and only if $b >_W c$ and ($c \not>_W b$ or $b \triangle_W c$ or $c \triangle_W b$), or $b = c$.

The basis of the causal ordering defined here, is that two tasks A and B have a causal relation $A \rightarrow B$ if in some process instance, A is directly followed by B and B is never directly followed by A. However, this can lead to problems if the two tasks are in a loop of length two. Therefore, $A \rightarrow B$ also holds if there is a process instance containing ABA or BAB and A nor B can directly succeed themselves. If A directly succeeds itself, then $A \rightarrow A$. For the example log presented in Table 1, $T = \{S, A, B\}$ and causal ordering inferred on T is composed of the following two elements $S \rightarrow_W A$ and $S \rightarrow_W B$.

By defining the $\rightarrow_W$ relation, we defined an ordering relation on T. This relation is not necessarily irreflexive, asymmetric, nor acyclic. This $\rightarrow_W$ relation however can be used to induce an ordering on the domain of any instance σ that has these properties. This is done in two steps. First, an asymmetric order is defined on the domain of some σ. Then, we prove that this relation is irreflexive and acyclic.

Definition 3.2. (Instance ordering) Let W be a process log over T and let $\sigma \in W$ be a process instance. Furthermore, let $\rightarrow_W$ be a causal ordering on T. We define an ordering $\succ_\sigma$ on the domain of σ, D_σ in the following way. For all $i, j \in D_\sigma$ such that $i < j$ we define $i \succ_\sigma j$ if and only if $\sigma_i \rightarrow_W \sigma_j$ and $\nexists_{i<k<j}(\sigma_i \rightarrow_W \sigma_k)$ or $\nexists_{i<k<j}(\sigma_k \rightarrow_W \sigma_j)$.

The essence of the relation defined here is in the final part. For each entry within an instance, we find the *closest* causal predecessor and the *closest* causal successor. If there is no causal predecessor or successor then the entry is in parallel with all its predecessors or successors respectively. It is trivial to see that this can always be done for any process instance and with *any* causal relation.

In the example log presented in Table 1 there are two process instances, case 1 and case 2. From here on, we will refer to case 1 as σ_1 and to case 2 as σ_2. We know that $\sigma_1 = SAB$ and that $D_{\sigma_1} = \{1, 2, 3\}$. Using the causal relation $\rightarrow$ the relation $\succ_{\sigma_1}$ is inferred such that $1 \succ_{\sigma_1} 2$ and $1 \succ_{\sigma_1} 3$. For σ_2 this also applies.

It is easily seen that the ordering relation $\succ_\sigma$ is indeed irreflexive and asymmetric, since it is only defined on i and j for which $i < j$. Therefore, it can easily be concluded that it is *irreflexive* and *acyclic*. Furthermore, the third property holds as well. Therefore we can now define an instance net as $(\sigma, \succ_\sigma)$.

3.2 Creating Instance Graphs

In this section, we present an algorithm to generate an instance graph from an instance net. An instance graph is a graph where each node represents one log entry of a specific instance. These instance graphs can be used as a basis to generate models in a particular language.

Definition 3.3. (Instance graph) Consider a set of nodes N and a set of edges $E \subseteq N \times N$. We call $G = (N, E)_\sigma$ an instance graph of an instance net $(\sigma, \succ_\sigma)$ if and only if the following conditions hold.

1. $N = D_\sigma \cup \{0, |D_\sigma| + 1\}$ is the set of nodes.
2. The set of edges E is defined as $E = E_{rel} \cup E_{initial} \cup E_{final}$, where
   $E_{rel} = \{(n_1, n_2) \in N \times N | (n_1 \succ_\sigma n_2)\}$ and
   $E_{initial} = \{(0, n) \in N \times N | \nexists_{n_1}(n_1 \succ_\sigma n)\}$ and
   $E_{final} = \{(n, |N| - 1) \in N \times N | \nexists_{n_1}(n \succ_\sigma n_1)\}$

An instance graph as described in Definition 3.3 is a graph that typically describes an execution path of some process model. This property is what makes an instance graph a good description of an instance. It not only shows causal relations between tasks but also parallelism if parallel branches are taken by the instance. However, choices are not represented in an instance graph. The reason for that is obvious, since choices are made at the execution level and do not appear in an instance. With respect to these choices, we can also say that if the same choices are made at execution, the resulting instance graph is the same. Note, that the fact that the same choices are made does not imply that the process instance is the same. Tasks that can be done in parallel within one instance can appear in any order in an instance without changing the resulting instance graph.

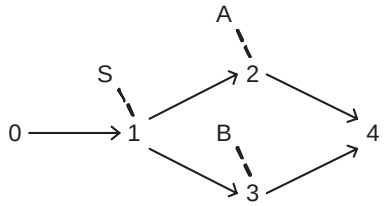

Fig. 2. Instance graph for σ_1.

For case 1 of the example log of Table 1 the instance graph is drawn in Figure 2. Note that in this graph, the nodes 1,2 and 3 are actually in the domain of σ_1 and therefore, they refer to entries in Table 1. It is easily seen that for case 2 this graph looks exactly the same, although the nodes refer to different entries.

In order to make use of instance graphs, we will show that an instance graph indeed describes an instance such that an entry in the log can only appear if all predecessors of that entry in the graph have already appeared in the instance.

Definition 3.4. (Pre- and postset) Let $G = (N, E)_\sigma$ be an instance graph and let $n \in N$. We define $\bullet_G n$ to be the preset of n such that $\bullet_G n = \{n' \in N | (n', n) \in E\}$. We define $n\bullet_G$ to be the postset of n such that $n\bullet_G = \{n' \in N | (n, n') \in E\}$.

Property 3.5. (Instance graphs describe an instance) Every instance graph $G = (N, E)_\sigma$ of some process instance σ describes that instance in such a way that for all $i, j \in N$ holds that for all $j \in \bullet_G i$ implies that $j < i$. This ensures that every entry in process entry σ occurs only after all predecessors in the instance graph have occurred in σ.

Proof. To prove that this is indeed the case for instance graph $G = (N, E)_\sigma$, we consider Definition 3.3 which implies that for "internal nodes" we know that $(n_1, n_2) \in E$ if and only if $n_1 \succ_\sigma n_2$. Furthermore, from the definition of $\succ_\sigma$ we know that $n_1 \succ_\sigma n_2$ implies that $n_1 < n_2$. For the source and sink nodes, it is also easy to show that $n_1 \in \bullet_G n_2$ implies that $n_1 < n_2$ because 0 is the smallest element of N while $|N| - 1$ is the largest. □

Property 3.6. (Strongly connectedness) For every instance graph $G = (N, E)_\sigma$ of some process instance σ holds that the short circuited graph $G' = (N, E \cup \{(|N| - 1, 0)\})$ is strongly connected.[1]

Proof. From Definition 3.3 we know that for all $i \in D_\sigma$ such that there does not exist a $j \in D_\sigma$ such that $j \succ_\sigma i$ holds that $(0, i) \in E$. Furthermore, we know that for all $i \in D_\sigma$ such that there does not exist a $j \in D_\sigma$ such that $i \succ_\sigma j$ holds that $(i, |\sigma| + 1) \in E$. Therefore, the graph is strongly connected if the edge $(|N| - 1, 0)$ is added to E. □

In the remainder of this paper, we will focus on an application of instance graphs. In Section 4 a translation from these instance graphs to a specific model are given.

4 Instance EPCs

In Section 3 instance graphs were introduced. In this section, we will present an algorithm to generate instance EPCs from these graphs. An instance EPC is a special case of an EPC (Event-driven Process Chain, [13]). For more information on EPCs we refer to [13, 14, 19]. These instance EPCs (or i-EPCs) can only contain AND-split and AND-join connectors, and therefore do not allow for loops to be present. These i-EPCs serve as a basis for the tool ARIS PPM (Process Performance Monitor) described in the introduction.

In this section, we first provide a formal definition of an instance EPC. An instance EPC does not contain any connectors other than AND-split and AND-joins connectors. Furthermore, there is exactly one initial event and one final event. Functions refer to the entries that appear in a process log, events however do not appear in the log. Therefore, we make the assumption here that each event uniquely causes a function to happen and that functions result in one or more events. An exception to this assumption is made when there are multiple functions that are the start of the instance. These functions are all preceded by an AND-split connector. This connector is preceded by the initial event. Consequently, all other connectors are preceded by functions and succeeded by events.

Definition 4.1. (Instance EPC) Consider a set of events E, a set of functions F, a set of connectors C and a set of arcs $A \subseteq ((E \cup F \cup C) \times (E \cup F \cup C)) \setminus$

[1] A graph is strongly connected if there is a directed path from any node to any other node in the graph.

$((E \times E) \cup (F \times F))$. We call (E, F, C, A) an instance EPC if and only if the following conditions hold.

1. $E \cap F = F \cap C = E \cap C = \emptyset$
2. Functions and events alternate in the presence of connectors: $\forall_{n_1,n_2 \in E \cup F} \forall_{(c_1,c_2) \in (A \cap (C \times C))^+ \cup I}((n_1, c_1) \in A \ \wedge \ (c_2, n_2) \in A) \Rightarrow (n_1 \in E \Leftrightarrow n_2 \in F)$, where $I = \{(c, c) \mid c \in C\}$.
3. The graph $(E \cup F \cup C, A)$ is acyclic.
4. There exists exactly one event $e_i \in E$ such that there is no element $n \in F \cup C$ such that $(n, e_i) \in A$. We call e_i the initial event.
5. There exists exactly one event $e_f \in E$ such that there is no element $n \in F \cup C$ such that $(e_f, n) \in A$. We call e_f the final event.
6. The graph $(E \cup F \cup C, A \cup \{(e_f, e_i)\})$ is strongly connected.
7. For each function $f \in F$ there are exactly two elements $n_1, n_2 \in E \cup C$ such that $(f, n_1) \in A$ and $(n_2, f) \in A$. Functions only have one input and one output.
8. For each event $e \in E / \{e_i, e_f\}$ there are exactly two elements $n_1, n_2 \in F \cup C$ such that $(e, n_1) \in A$ and $(n_2, e) \in A$. Events only have one input and one output, except for the initial and the final event. For them the following holds. For e_i there is exactly one element $n \in F \cup C$ such that $(e_i, n) \in A$ and for e_f there is exactly one element $n \in F \cup C$ such that $(n, e_f) \in A$.

4.1 Generating Instance EPCs

Using the formal definition of an instance EPC from Definition 4.1, we introduce an algorithm that produces an instance EPC from an instance graph as defined in Definition 3.3. In the instance EPC generated it makes sense to label the functions according to the combination of the task name and event type as they appear in the log. The labels of the events however cannot be determined from the log. Therefore, we propose to label the events in the following way. The initial event will be labeled "initial". The final event will be labeled "final". All other events will be labeled in such a way that it is clear which function succeeds it. Connectors are labeled in such a way that it is clear whether it is a split or a join connector and to which function or event it connects with the input or output respectively.

Definition 4.2. (Converting instance graphs to EPCs) Let W be a process log and let $G = (N_G, E_G)_\sigma$ be an instance graph for some process instance $\sigma \in W$. To create an instance EPC, we need to define the four sets E, F, C and A.

- The set of functions F is defined as $F = \{f_i \mid i \in D_\sigma\}$. In other words, for every entry in the process instance, a function is defined.
- The set of events E is defined as $E = \{e_{f_i} \mid f_i \in F \text{ and } \exists_{j \in D_\sigma}(j \succ_\sigma i)\} \cup \{e_{initial}, e_{final}\}$. In other words, for every function there is an event preceding it, unless it is a minimal element with respect to $\succ_\sigma$. Furthermore, there is an initial event $e_{initial}$ and a final event e_{final}.

– The set of connectors C is defined as $C = C_{split} \cup C_{join} \cup C_i \cup C_f$ where
$C_{split} = \{c_{(split,f_i)} \mid f_i \in F \wedge |i \bullet_G| > 1\}$ and
$C_{join} = \{c_{(join,e_{f_i})} \mid e_{f_i} \in E \wedge |\bullet_G i| > 1\}$ and
$C_i = \{c_{(split,e_{initial})} \mid |0 \bullet_G| > 1\}$ and
$C_f = \{c_{(join,e_{final})} \mid |\bullet_G (|N_G| - 1)| > 1\}$.
Here, the connectors are constructed in such a way that connectors are always preceded by a function, except in case the process starts with parallel functions, since then the event $e_{initial}$ is succeeded by a split connector.
– The set of arcs A is defined as $A = A_{ef} \cup A_{fe} \cup A_{split} \cup A_{join} \cup A_i \cup A_f$ where
$A_{ef} = \{(e_{f_i}, f_i) \in (E \times F)\}$ and
$A_{fe} = \{(f_i, e_{f_j}) \in (F \times E) \mid (i,j) \in E_G \wedge |i \bullet_G| = 1 \wedge |\bullet_G j| = 1\}$
$A_{split} = \{(f_i, c_{(split,f_i)}) \in (F \times C_{split})\} \cup$
$\{(c_{(split,f_i)}, e_{f_j}) \in (C_{split} \times E) \mid (i,j) \in E_G \wedge |\bullet_G j| = 1\} \cup$
$\{(c_{(split,f_i)}, c_{(join,e_{f_j})}) \in (C_{split} \times C_{join}) \mid (i,j) \in E_G\}$ and
$A_{join} = \{(c_{(join,e_{f_i})}, e_{f_i}) \in (C_{join} \times E)\} \cup$
$\{(f_i, c_{(join,e_{f_j})}) \in (F \times C_{join}) \mid (i,j) \in E_G \wedge |i \bullet_G| = 1\}$ and
$A_i = \{(e_{initial}, c_{(split,e_{initial})}) \in (E \times C_i)\} \cup$
$\{(c_{(split,e_{initial})}, f_i) \in (C_i \times F) | (0,i) \in E_G\}$ and
$A_f = \{(c_{(join,e_{final})}, e_{final}) \in (C_f \times E)\} \cup$
$\{f_i, (c_{(join,e_{final})}) \in (F \times D_f) | (i, (|N_G| - 1)) \in E_G\}$.

It is easily seen that the instance EPC generated by Definition 4.2 is indeed an instance EPC, by verifying the result against Definition 4.1.

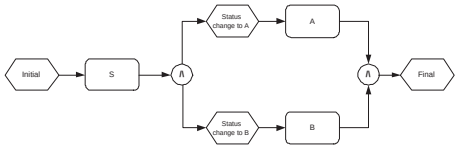

Fig. 3. Instance EPC for σ_1.

In definitions 3.3 and 4.1 we have given an algorithm to generate an instance EPC for each instance graph. The result of this algorithm for both cases in the example of Table 1 can be found in Figure 3. In Section 5 we will show the practical use of this algorithm to ARIS PPM.

5 Example

In this section, we present an example illustrating the algorithms described in sections 3 and 4. We will start from a process log with some process instances. Then, we will run the algorithms to generate a set of instance EPCs that can be imported into ARIS PPM.

5.1 A process log

Consider a process log consisting of the following traces.

case identifier	task executions
case 1	S_1, A_2, B_3, F_4, C_5, D_6, H_7, G_8, T_9
case 2	S_1, A_2, C_3, B_4, E_5, H_6, F_7, G_8, T_9
case 3	S_1, A_2, D_3, B_4, C_5, F_6, H_7, G_8, T_9
case 4	S_1, A_2, E_3, B_4, C_5, H_6, F_7, G_8, T_9
case 5	S_1, A_2, B_3, D_4, F_5, H_6, C_7, G_8, T_9
case 6	S_1, A_2, B_3, E_4, F_5, H_6, C_7, G_8, T_9
case 7	S_1, A_2, B_3, F_4, D_5, C_6, H_7, G_8, T_9
case 8	S_1, A_2, B_3, F_4, E_5, C_6, H_7, G_8, T_9
case 9	S_1, A_2, D_3, C_4, B_5, H_6, F_7, G_8, T_9
case 10	S_1, A_2, C_3, E_4, H_5, B_6, F_7, G_8, T_9

Table 2. A process log.

The process log in Table 2 shows the execution of tasks for a number of different instances of the same process. To save space, we abstracted from the original names of tasks and named each task with a single letter. The subscript refers to the position of that task in the process instance.

Using this process log, we will first generate the causal relations from Definition 3.1. Note that casual relations are to be defined between tasks and not between log entries. Therefore, the subscripts are omitted here. This definition leads to the following set of causal relations: $\{S \rightarrow A,\ A \rightarrow B,\ A \rightarrow C,\ A \rightarrow D,\ A \rightarrow E,\ B \rightarrow F,\ D \rightarrow H,\ E \rightarrow H,\ F \rightarrow G,\ C \rightarrow G,\ H \rightarrow G,\ G \rightarrow T\}$.

Using these relations, we generate instance graphs as described in Section 3 for each process instance. Then, these instance graphs are imported into ARIS PPM and a screenshot of this tool is presented (cf. Figure 5).

5.2 Instance graphs

To illustrate the concept of instance graphs, we will present the instance graph for the first instance, "case 1". In order to do this, we will follow Definition 3.2 to generate an instance ordering for that instance. Then, using these orderings, an instance graph is generated. Applying Definition 3.2 to case 1 in the log presented in Table 2 using the casual relations given in Section 5.1 gives the following instance ordering: $0 \succ 1, 1 \succ 2,\ 2 \succ 3,\ 3 \succ 4,\ 4 \succ 8,\ 8 \succ 9,\ 2 \succ 5,\ 5 \succ 8,\ 2 \succ 6,\ 6 \succ 7,\ 7 \succ 8,\ 8 \succ 9, 9 \succ 10$.

Using this instance ordering, an instance graph can be made as described in Definition 3.3. The resulting graph can be found in Figure 4. Note that the instance graphs of all other instances are isomorphic to this graph. Only, the numbers of the nodes change.

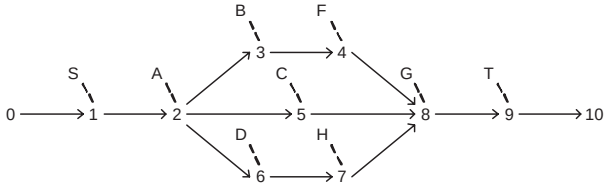

Fig. 4. Instance graph for case 1.

For each process instance, such an instance graph can be made. Using the algorithm presented in Section 4 each instance can than be converted into an instance EPC. These instance EPCs can be imported directly into ARIS PPM for further analysis. Here, we would like to point out again that our tools currently provide an implementation of the algorithms in this paper, such that the instance EPCs generated can be imported into ARIS PPM directly. A screenshot of this tool can be found in Figure 5 where "case 1" is shown as an instance EPC. Furthermore, inside the boxed area, the aggregation of some cases is shown. Note that this aggregation is only part of the functionality of ARIS PPM. Using graphical representations of instances, a large number of analysis techniques is available to the user. However, creating instances without knowing the original process model is an important first step.

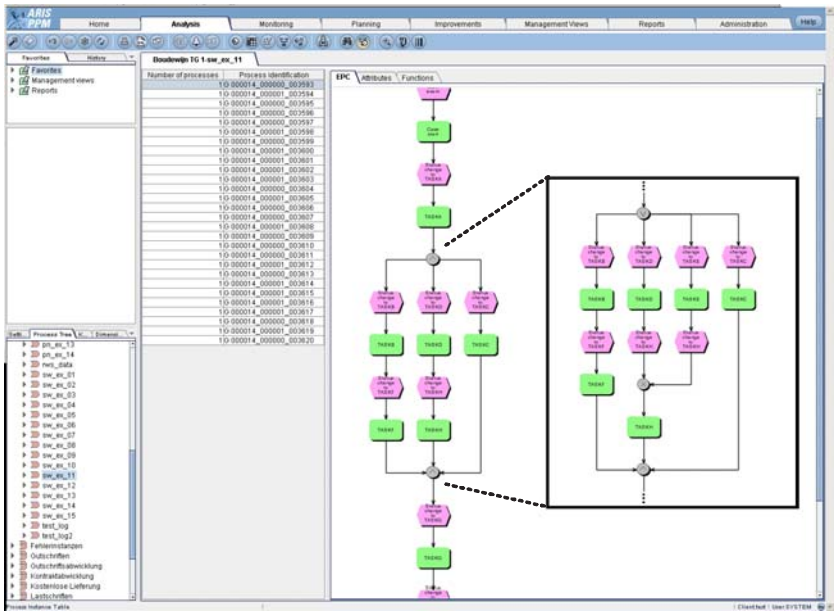

Fig. 5. ARIS PPM screenshot.

6 Related Work

The idea of process mining is not new [1, 3, 5–7, 11, 12, 15, 16, 18, 21] and most techniques aim at the control-flow perspective. For example, the α-algorithm allows for the construction of a Petri net from an event log [1, 5]. However, process mining is not limited to the control-flow perspective. For example, in [2] we use process mining techniques to construct a social network. For more information on process mining we refer to a special issue of Computers in Industry on process mining [4] and a survey paper [3]. In this paper, unfortunately, it is impossible to do justice to the work done in this area. To support our mining efforts we have developed a set of tools including EMiT [1], Thumb [21], and MinSoN [2]. These tools share the XML format discussed in this paper. For more details we refer to www.processmining.org.

The focus of this paper is on the mining of the control-flow perspective. However, instead of constructing a process model, we mine for instance graphs. The result can be represented in terms of a Petri net or an (instance) EPC. Therefore, our work is related to tools like ARIS PPM [12], Staffware SPM [20], and VIPTool [10]. Moreover, the mining result can be used as a basis for applying the theoretical results regarding partially ordered runs [8].

7 Conclusion

The focus of this paper has been on mining for instance graphs. Algorithms are presented to describe each process instance in a particular modelling language. From the instance graphs described in Section 3, other models can be created as well. The main advantage of looking at instances in isolation is twofold. First, it can provide a good starting point for all kinds of analysis such as the ones implemented in ARIS PPM. Second, it does not require any notion of completeness of a process log to work. As long as a causal relation is provided between log entries, instance graphs can be made. Existing methods such as the α-algorithm [1, 3, 5] usually require some notion of completeness in order to rediscover the entire process model. The downside thereof is that it is often hard to deal with noisy process logs. In our approach noise can be filtered out before implying the causal dependencies between log entries, without negative implications on the result of the mining process.

ARIS PPM allows for the aggregation of instance EPCs into an aggregated EPC. This approach illustrates the wide applicability of instance graphs. However, the aggregation is based on simple heuristics that fail in the presence of complex routing structures. Therefore, we are developing algorithms for the integration of multiple instance graphs into one EPC or Petri net. Early experiments suggest that such a two-step approach alleviate some of the problems existing process mining algorithms are facing [3, 4].

References

1. W.M.P. van der Aalst and B.F. van Dongen. Discovering Workflow Performance Models from Timed Logs. In Y. Han, S. Tai, and D. Wikarski, editors, *International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002)*, volume 2480 of *Lecture Notes in Computer Science*, pages 45–63. Springer-Verlag, Berlin, 2002.
2. W.M.P. van der Aalst and M. Song. Mining Social Networks: Uncovering interaction patterns in business processes. In M. Weske, B. Pernici, and J. Desel, editors, *International Conference on Business Process Management*, volume 3080 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, Berlin, 2004.
3. 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.
4. W.M.P. van der Aalst and A.J.M.M. Weijters, editors. *Process Mining*, Special Issue of Computers in Industry, Volume 53, Number 3. Elsevier Science Publishers, Amsterdam, 2004.

5. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. QUT Technical report, FIT-TR-2003-03, Queensland University of Technology, Brisbane, 2003. (Accepted for publication in IEEE Transactions on Knowledge and Data Engineering.).
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. 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.
9. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.
10. J. Desel, G. Juhas, R. Lorenz, and C. Neumair. Modelling and Validation with VipTool. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 380–389. Springer-Verlag, 2003.
11. 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.
12. IDS Scheer. ARIS Process Performance Manager (ARIS PPM). http://www.ids-scheer.com, 2002.
13. 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.
14. G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation*. Addison-Wesley, Reading MA, 1998.
15. 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.
16. M. zur Mühlen and M. Rosemann. Workflow-based Process Monitoring and Controlling - Technical and Organizational Issues. In R. Sprague, editor, *Proceedings of the 33rd Hawaii International Conference on System Science (HICSS-33)*, pages 1–10. IEEE Computer Society Press, Los Alamitos, California, 2000.
17. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
18. M. Sayal, F. Casati, and M.C. Shan U. Dayal. Business Process Cockpit. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 880–883. Morgan Kaufmann, 2002.
19. A.W. Scheer. *Business Process Engineering, Reference Models for Industrial Enterprises*. Springer-Verlag, Berlin, 1994.
20. Staffware. Staffware Process Monitor (SPM). http://www.staffware.com, 2002.
21. 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.