Process Mining Based on Clustering: A Quest for Precision

A.K. Alves de Medeiros[1], A. Guzzo[2], G. Greco[2], W.M.P. van der Aalst[1], A.J.M.M. Weijters[1], B. van Dongen[1], and D. Saccà[2]

[1] Eindhoven University of Technology, P.O. Box 513, 5600MB, Eindhoven, The Netherlands. E-mails: {a.k.medeiros, w.m.p.v.d.aalst, a.j.m.m.weijters, b.f.v.dongen}@tue.nl

[2] University of Calabria, Via Bucci 41C, 187036 Rende (CS), Italy E-mails: guzzo@icar.cnr.it, ggreco@mat.unical.it, sacca@unical.it

Abstract. Process mining techniques attempt to extract non-trivial and useful information from event logs recorded by information systems. For example, there are many process mining techniques to automatically discover a process model based on some event log. Most of these algorithms perform well on structured processes with little disturbances. However, in reality it is difficult to determine the scope of a process and typically there are all kinds of disturbances. As a result, process mining techniques produce spaghetti-like models that are difficult to read and that attempt to merge unrelated cases. To address these problems, we use an approach where the event log is clustered iteratively such that each of the resulting clusters corresponds to a coherent set of cases that can be adequately represented by a process model. The approach allows for different clustering and process discovery algorithms. In this paper, we provide a particular clustering algorithm that avoids over-generalization and a process discovery algorithm that is much more robust than the algorithms described in literature [1]. The whole approach has been implemented in ProM.

Keywords: Process Discovery, Process Mining, Workflow Mining, Disjunctive Workflow Schema, ProM Framework

1 Introduction

The basic idea of *process mining* is to discover, monitor and improve *real* processes (i.e., not assumed processes) by extracting knowledge from event logs [1]. Today many of the tasks occurring in processes are either supported or monitored by information systems (e.g., ERP, WFM, CRM, SCM, and PDM systems). However, process mining is not limited to information systems and can also be used to monitor other operational processes or systems (e.g., web services, care flows in hospitals, and complex devices like wafer scanners, complex X-ray machines, high-end copiers, etc.). All of these applications have in common that *there is a notion of a process* and that *the occurrences of tasks are recorded in so-called event logs*. Assuming that we are able to log events, a wide range of *process mining techniques* comes into reach. The basic idea of process mining is to learn from observed executions of a process and it can be used to (1) *discover* new models (e.g., constructing a Petri net that is able to reproduce the observed behavior), (2) check the *conformance* of a model by checking whether

Table 1. Example of an event log (with 300 process instances) for the process of a one-day conference. Each row refers to process instances that follow a similar pattern in terms of tasks being executed. The first row corresponds to 80 process instances that all followed the event sequence indicated.

Identifier	Process instance	Frequency
1	Start, Get Ready, Travel by Car, Conference Starts, Give a Talk, Join Guided Tour, Join Dinner, Go Home, Pay for Parking, Travel by Car, End.	80
2	Start, Get Ready, Travel by Train, Conference Starts, Give a Talk, Join Guided Tour, Join Dinner, Go Home, Travel by Train, End.	68
3	Start, Get Ready, Travel by Car, Conference Starts, Join Guided Tour, Join Dinner, Go Home, Pay for Parking, Travel by Car, End.	81
4	Start, Get Ready, Travel by Train, Conference Starts, Join Guided Tour, Join Dinner, Go Home, Travel by Train, End.	71

the modeled behavior matches the observed behavior, and (3) *extend* an existing model by projecting information extracted from the logs onto some initial model (e.g., show bottlenecks in a process model by analyzing the event log).

In this paper, we focus on process discovery. Concretely, we want to construct a process model (e.g., a Petri net) based on an event log where for each case (i.e., an instance of the process) a sequence of events (say tasks) is recorded. Table 1 shows an aggregate view of such a log. This log will be used as a running example and describes the events taking place when people attend a conference. It is a toy example, but it is particularly useful when explaining our approach. Note that in the context of ProM we have analyzed many real-life logs (cf. www.processmining.org). However, the corresponding processes are too difficult to describe when explaining a specific process mining technique. Therefore, we resort to using this simple artificial process as a running example. Each process instance corresponds to a sequence of events (i.e., task executions). Process instances having the same sequence are grouped into one row in Table 1. In total there are 300 process instances distributed over 4 possible sequences. Note that all process instances start with task "Start" and end with task "End". The process instances referred to by the first two rows contain task "Give a Talk" while this event is missing in process instances referred to by the last two rows. This suggests that some people give a talk while others do not. Rows 1 and 3 refer to two occurrences of task "Travel by Car" and the other two rows (2 and 4) refer to two occurrences of task "Travel by Train". This indicates that the people that arrive by car (train) also return by car (train). People that come by car also execute task "Pay for Parking". Note that Table 1 shows only an aggregate view of the real log. In fact, real logs typically contain much more information, e.g., timestamps, transactional information, information on users, data attributes, etc. However, for the purpose of this paper, we can abstract from this information and focus in the information shown in Table 1.

A possible result of a process discovery mining algorithm for the log in Table 1 is depicted in Figure 1. The process is represented in terms of a Petri net [6], i.e.,

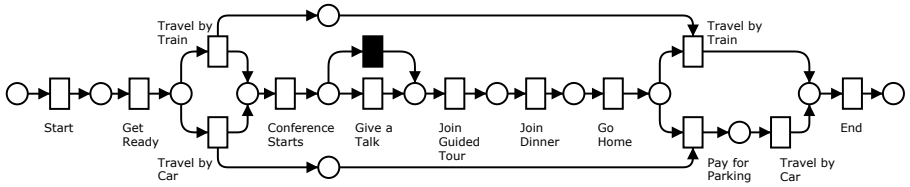

Fig. 1. Example of a mined model for the log in Table 1. This model correctly captures the right level of abstraction for the behavior in the log.

a bipartite directed graph with two node types: *places* and *transitions*. Places (circles) represent states while transitions (rectangles) represent actions (e.g., tasks). A certain state holds in a system if it is *marked* (i.e., it contains at least one *token*). Tokens flow between states by *firing* (or executing) the transitions. A transition is *enabled* (or may fire) if all of its input places have at least one token. When a transition fires, it removes one token from each of its input places and it adds one tokens to each of its output places. More information about Petri nets can be found in [6]. Figure 1 shows that the process can be started by putting a token in the source place at the left and firing the two leftmost transitions. After this, there is a choice to fire "Travel by Train" or "Travel by Car". Firing one of these two transitions results in the production of two tokens: one to trigger the next step ("Conference Starts") and one to "remember" the choice. Note that after task "Go Home" (later in the process) there is another choice, but this choice is controlled by the earlier choice, e.g., people that come by train return by train. Also note that after task "Conference Starts" there is a choice to give a talk or to bypass this task. The black transition refers to a "silent step", i.e., a task not recorded in the log because it does not correspond to a real activity and has only been added for routing purposes.

The model shown in Figure 1 allows for the execution of all process instances in the original log (Table 1). Moreover, the model seems to be at the right level of abstraction because it does not allow for more behavior than the one in the log (e.g., it explicitly portrays that attendees have used the same means of transportation to go to and come back from the conference) and there does not seem to be a way to simplify it without destroying the fit between the log and model. Note that in Figure 1 some of the tasks are duplicated, there are the transitions labeled "Travel by Train" and "Travel by Car" that occur multiple times. Most mining techniques do not allow for this. If we apply a mining algorithm that does not support duplicate tasks, the resulting model could look like the one in Figure 2. Note that, although this model captures (or can reproduce) the behavior in the log (cf. Table 1), it is more general than necessary because: (i) attendees can use different means of transportation to reach and leave the one-day conference, (ii) attendees can skip the whole conference after traveling by car or by train, and (iii) attendees can return to the conference after going home (cf. task "Go Home"). So, this model is not a precise picture of the traces in the log. For such a small log, it is easy to visually detect the points of over-generalizations in the model in Figure 2. However, when mining bigger logs, there is a need for a technique that can automatically identify these points.

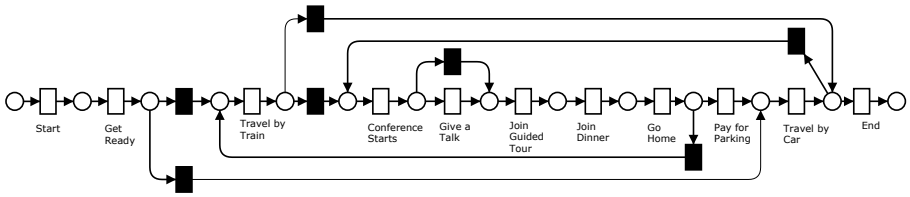

Fig. 2. Example of another mined model for the log in Table 1. This model is more general than necessary.

Figures 1 and 2 show that process mining algorithms may produce suitable models but also models that are less appropriate. In particular, the more robust algorithms have a tendency to over-generalize, i.e., construct models that allow for much more behavior than actually observed. A related problem is that event logs typically record information related to different processes. For example, there could be process instances related to the reviewing of papers mixed up with the instances shown in Table 1. Note that the distinction between processes is sometimes not very clear and in a way arbitrarily chosen. In our example, one could argue that there are two processes: one for handling people coming by car and the other one for people coming by train. When processing insurance claims, one could argue that there is one process to handle all kinds of insurance claims. However, one could also have a separate process for each of the different types of insurance policies (e.g., fire, flooding, theft, health, and car insurance). *If one attempts to construct a single model for very different cases, the model is likely to be too complex and existing mining techniques are unable to cope without excessive over-generalization.* This is the core problem addressed by the approach presented in this paper.

To address problems related to over-generalization and mixing up different processes in a single model, we propose to use *clustering*. We do not aim to construct one big model that explains everything. Instead we try to cluster similar cases in such a way that for every cluster it is possible to construct a relatively simple model that fits well without too much over-generalization. The idea for such an approach was already mentioned in [4]. However, in this paper we refine the technique and use a much more powerful process mining technique. Moreover, we report on the implementation of this approach in ProM. Both the ProM framework and the described plug-ins are publicly available at www.processmining.org.

The remainder of this paper is organized as follows. Section 2 provides an overview of the approach in this paper. Section 3 explains our implementation for this approach and Section 4 describes how to use this implementation to discover over-generalizations in mined models and common patterns in logs. Section 5 presents related work and Section 6 concludes this paper.

2 Approach

As explained in the introduction, the goal of this paper is to allow for the mining of processes with very diverse cases (i.e., not a homogeneous group of process instances) while avoiding over-generalization. Consider for example the application of process mining to care flows in hospitals. If one attempts to construct a single process model for all patients, the model will be very complex because it is difficult to fit the different kinds of treatments into the same model. The dif-

ferent patient groups are ill-defined and share resources and process fragments. For example, what about the patient that was hit by a car after getting a hearth attack? This patient needs both a knee operation and hearth surgery and belongs to a mixture of patient groups. This example shows that it is not easy to define the boundary of a process and that given the heterogeneity of cases it may be impossible to find a clean structure. Moreover, process mining techniques that are able to deal with less structured processes have a tendency to over-generalize, i.e., the process instances fit into the model but the model allows for much more behavior than what is actually recorded. To address this we suggest to *iteratively split the log in clusters until the log is partitioned in clusters that allow for the mining of precise models.*

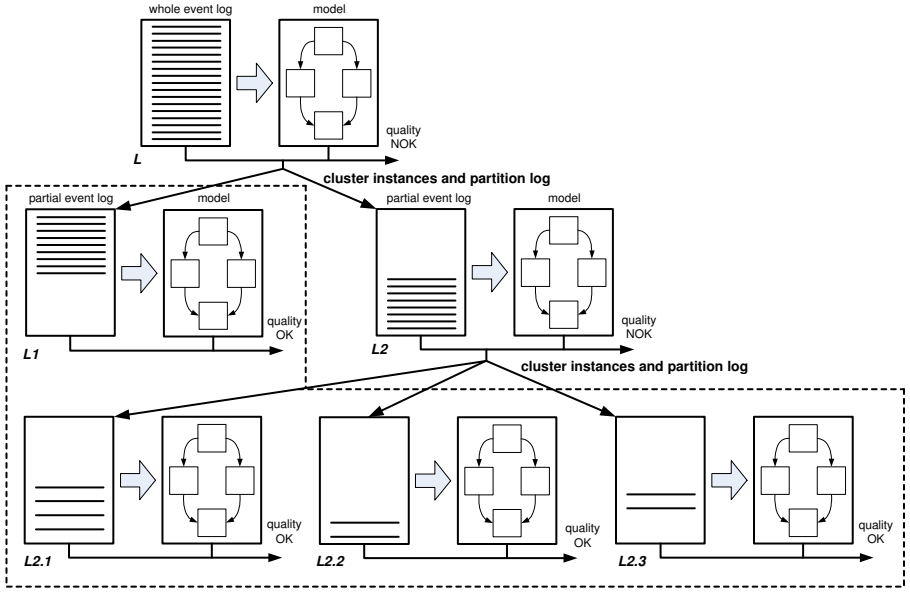

Fig. 3. Overview of the approach: the process instances are iteratively partitioned into clusters until it is possible to discover a "suitable model" for each each cluster.

Figure 3 shows the basic idea behind this approach. First the whole log is considered (denoted by L in Figure 3). Using a discovery algorithm a process model is constructed. By comparing the log and the process model, the quality of the model is measured. If the model has good quality and there is no way to improve it, the approach stops. However, as long as the model is not optimal the log is partitioned into clusters with the hope that it may be easier to construct a better process model for each of the clusters. In Figure 3 log L is split into two logs $L1$ and $L2$, i.e., each process instance of L appears in either $L1$ or $L2$. Then for each cluster, the procedure is repeated. In Figure 3, the quality of the model discovered for cluster $L1$ is good and $L1$ is not partitioned any further. The quality of the model discovered for cluster $L2$ is not OK and improvements are possible by partitioning $L2$ into three clusters: $L2.1$, $L2.2$, and $L2.3$. As Figure 3 shows it is possible to construct suitable process models for each of these three clusters and the approach ends.

Note that the approach results in a hierarchy of clusters each represented by a partial log and a process model. This is quite different from conventional process mining approaches that produce a single model. The leaves of the tree presented in Figure 3 (enclosed by dashed lines) represent clusters that are homogenous enough to construct suitable models.

When applying the approach illustrated by Figure 3, there are basically three decisions that need to be made: *(i) When to further partition a cluster?* The approach starts with considering the whole log as a single cluster. Then this cluster is partitioned into smaller clusters which again may be partitioned into even smaller clusters, etc. Note that this always ends because eventually all clusters contain only one process instance and cannot be split anymore. However, it is desirable to have as few clusters as possible, so a good stopping criterion is needed. Stopping too late, may result in a proces model for every process instance. Stopping too early, may result in low quality process models that try to describe a set of cases that is too heterogeneous;*(ii) How to split a cluster into smaller clusters?* There are many ways to split a cluster into smaller clusters. An obvious choice is to cluster process instances that have a similar "profile" when it comes to task executions, e.g., split the log into a cluster where "A" occurs and a cluster where "A" does not occur. It is also possible to use more refined features to partition process instances. It may also be possible to use data elements, e.g., in case of hospital data it may be wise to cluster patients based on the diagnosis. Different approaches known from the data mining field can be used here; *(iii) What discovery algorithm to use?* To extract a process model from the process instances in the cluster different process mining algorithms can be used. Some algorithms are very robust but yield models that allow for too much behavior. Other algorithms produce models that are unable to replay the existing log, i.e., there are process instances whose behavior is not allowed according to the model.

The approach presented in this paper is inspired by the process mining algorithm described in [4]. Here the approach is termed Disjunctive Workflow Schema (DWS) and particular choices are made for the three questions listed above. For example, a rather weak algorithm is used for process discovery. The algorithm described in [4] is unable to deal with loops, non-free-choice constructs (e.g., the controlled choice in Figure 1 forcing people to take the same means of transportation home), etc. Moreover, the different parts of the approach are tightly coupled in [4]. In our view, it is essential to allow for different techniques to be plugged into the approach illustrated by Figure 3. Hence, this paper improves this earlier work in several directions: (1) the approach is presented independent of a particular algorithm, (2) several improvements have been made (e.g., replacing the process mining algorithm), and (3) the whole approach is implemented in ProM using an architecture that makes it easy to plug-in new clustering algorithms or process discovery algorithms.

The remainder of this paper presents a particular choice for each of the three questions stated before and describes the new plug-ins implemented in ProM. We use the combination of a very robust process mining algorithm (Heuristics Miner) combined with a clustering approach focusing on over-generalization.

3 Implementation

This section explains how we have answered the questions raised in Section 2 and describes the resulting plug-ins that were implemented in ProM.

3.1 When to Further Partition a Cluster?

A cluster should be further partitioned when its mined model allows for more behavior than what is expressed by the traces in the cluster. So, in our approach, generalizations in the model are captured by *selecting features* (or structural patterns) that, while being in principle executable according to the model, have never been registered in the log. If such discrepancies can be identified, then the model is not an accurate representation for the process underlying the log. Hence, we are given some evidence that the model has to be further specialized into a set of different, more specific use cases.

To identify the relevant features, we use an *A-priori* [2] like approach. The idea is to incrementally generate sequences of tasks by extending a sequence of length n with another task, and to subsequently check for their frequency in the log[3]. A relevant feature is basically a sequence for which this incremental extension cannot be carried out by guaranteeing that the resulting sequence is as frequent as its two basic constituents. Hence, a relevant feature is a sequence, say $t_1, ..., t_n$, together with a task, say t_{n+1} such that (cf. Figure 4): (i) $t_1, ..., t_n$ is *frequent*, i.e., the fraction of projected log traces in which the sequence occurs is greater than a fixed threshold (called *sigma*); (ii) t_n, t_{n+1} is also frequent with respect to the same threshold *sigma*; but,(iii) the whole sequence $t_1, ..., t_n, t_{n+1}$ is *not* frequent, i.e., its occurrence is smaller than some threshold *gamma*.

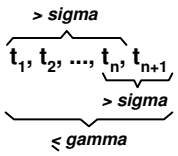

Fig. 4. Feature selection. The subparts ("$t_1...t_n$" and "t_nt_{n+1}") of the feature should occur more than *sigma* times in the projected log traces while the whole sequence ("$t_1...t_{n+1}$") should occur at most *gamma* times.

3.2 How to Split a Cluster into Smaller Clusters?

We use the *k-means* method [5] to split a cluster into sub-clusters. *k-means* is a clustering algorithm based on Euclidian distance in vectorial spaces. It works by finding central points (or centroids) over which a set of vectors are clustered. Every cluster has a centroid. The parameter k determines the number of centroids (and, therefore, the number of clusters). Consequently, to reuse the *k-means* clustering method, we have designed an approach for producing a flat representation of the traces by projecting each of them on the relevant features identified as outlined in Section 3.1. In particular, only the most relevant m features are considered. The relevance of each feature is measured on the basis

[3] Note that log traces are projected while checking for the frequency of these sequences in the log. During the projection, only the tasks that are in a sequence are kept in the log traces.

of the frequency of its occurrences in the log (the more frequent, the more relevant). Once features have been selected, each trace in the log is processed and associated with a vector in a m-dimensional vectorial space: for each feature, the corresponding entry of the vector is set to a value that is proportional to the fraction of the feature actually occurring in the trace. In the extreme case where the feature does not characterize the trace, this value is set to 0. When the whole feature matches the trace, this value is set to 1. After the log traces have been projected in the vectorial space, the *k-means* clustering method takes place.

3.3 What Discovery Algorithm to Use?

We have selected the *Heuristics Miner* (HM) [9] to use as the mining algorithm in our approach. The HM can deal with noise and can be used to express the main behavior (i.e., not all the details and exceptions) registered in an event log. It supports the mining of all common constructs in process models (i.e., sequence, choice, parallelim, loops, invisible tasks and some kinds of non-free-choice), except for duplicate tasks. Therefore, the HM is a more robust algorithm than the mining algorithm originally used in [4]. The HM algorithm has two main steps. In the first step, a *dependency graph* is built. In the second step, the *semantics of the split/join points* in the dependency graph are set. Due to the lack of space, in this paper we do not elaborate in the relation between these two steps and the threshold values used by the HM. The interested reader is referred to [9].

The next subsection introduces the two plug-ins that were implemented in ProM to support the choices explained so far in this section.

3.4 Implemented Plug-ins

Two ProM plug-ins have been implemented to provide for the mining of precise models: *DWS Mining* and *DWS Analysis*. The *DWS Mining* plug-in implements the full-cycle of the approach in Figure 3. This plug-in starts with an event log and uses the Heuristic Miner (cf. Section 3.3) to mine a model for this log. Afterwards, the plug-in tries to detect relevant features for this model (cf. Section 3.1). If features are found, the plug-in clusters this log based on *k-means* (cf. Section 3.2). If further sub-clusters can be identified, a model is again automatically mined by the HM for each sub-cluster. This iterative procedure continues until no further clustering is possible. Figure 5 shows a screenshot of applying the DWS Mining plug-in to the log in Table 1. As can be seen, *the window for specifying the settings* has two parts: one for setting the parameters used by the HM (cf. Figure 5(a)) and another for setting the parameters for the feature selection and clustering (cf. Figure 5(b)). The parameters for the HM correspond to the thresholds mentioned in Section 3.3. The first three parameters in the panel in Figure 5(b) are respectively used to determine the *sigma*, *gamma* and k threshold explained in sections 3.1 and 3.2. Note that other three extra parameters are provided ("(Max.) Length of features", "(Max.) Number of splits" and "(Max.) Number of features") which allow for determining upper bounds to the algorithm. The parameter "(Max.) Length of features" sets the length of the sequences up to which the discovery of the frequent features is carried out. In many

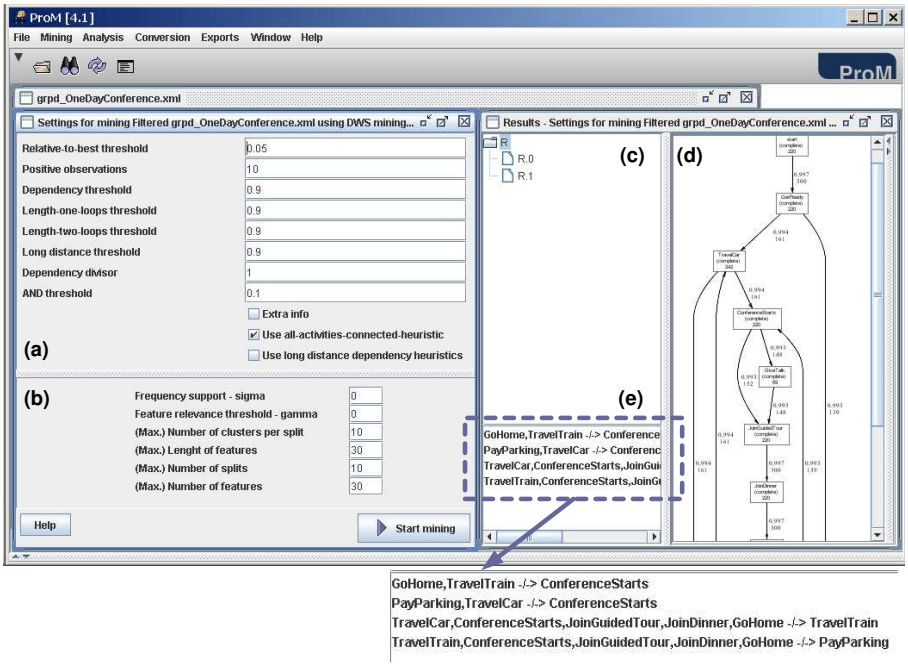

Fig. 5. Screenshot of the *DWS mining* plug-in

practical situations, this parameter may be set to 2, meaning that one looks for features of the form t_1, t_2 with a task t_3 such that t_1, t_2, t_3 is not frequent. Larger values for this length may be desirable for models involving many tasks, especially when the paths between the starting task and some final one involve lots of tasks. The parameter "(Max.) Number of splits" is an upper bound on the total number of splits to be performed by the algorithm. The higher its value, the deeper the resulting hierarchy can be. The parameter "(Max.) Number of features" defines the dimension of the feature space over which the clustering algorithm is applied. Note that "(Max.) Number of features" should be greater than k (i.e. parameter "(Max.) Number of clusters per split"). In general, larger values of "(Max.) Number of features" lead to higher quality in the clustering results but it requires more computational time. The *window for showing the results* has three panels: the hierarchy of the found clusters (cf. Figure 5(c)), the model mined for each cluster[4] (cf. Figure 5(d)), and the set of relevant features (cf. Figure 5(e)) used to split each cluster. The subcomponents of each feature are separated by the symbol "-/->". The substrings on the left and right side of this symbol respectively correspond to $t_1 ... t_n$ and t_{n+1} in Figure 5. As can be seen at the bottom of this figure, four features have been found for the settings in this example. These features reflect the generalizations already discussed in

[4] The model mined by the HM for this example is just like the one in Figure 2. However, instead of Petri nets, the HM uses *Heuristics nets* as the notation to represent models. Due to the lack of space and the fact that the two models allow for the same behavior, we will not provide an explanation about Heuristics nets. The interested reader is referred to [9].

Section 1. The results in Figure 5 are discussed in Section 4. The *DWS Analysis* plug-in is very similar to the DWS mining one. However, it has the advantage that the approach is decoupled from a specific mining plug-in. The input of the DWS Analysis plug-in is a log *and* a model. Its output is a set of partitions (or clusters) for this log. No sub-clusters are provided because the user can again choose which mining plug-ins to use for each of the resulting clusters.

The next section describes how to use the parameters in these two plug-ins to detect (i) over-generalizations in mined models and (ii) frequent patterns in the log.

4 Detecting Over-Generalizations and Common Patterns

A (mined) model is over-general when it can generate behavior that cannot be derived from the log. In other words, certain sequences (or features) are possible in the model but do not appear in the log. Using this reasoning, over-generalizations can be detected by the DWS analysis or mining plug-ins whenever we (i) set the *sigma* and *gamma* parameters to 0, (ii) allow for feature sizes that are *at least* as big as the number of tasks in the model and (iii) set a maximum number of features so that all points of over-generalization are kept in the list of identified features. As an illustration, consider the screenshot in Figure 5. This figure shows the result of applying the DWS mining plug-in to the model in Figure 2 linked to the log in Table 1. Note that the DWS mining plug-in successfully detected *four* points of over-generalizations in the model. For instance, the first feature (see bottom-right) states that the task "ConferenceStarts" was never executed after the sequence "GoHome,TravelTrain" has happened, although the sequence "TravelTrain,ConferenceStarts" appears in the log. Actually, the first two features indicate that attendees did not return to the conference after going home and the last two features reflect that attends always used the same means of transportation while reaching of leaving the conference. Note that the algorithm did not capture the over-generalizations for the sequences "GetReady,TravelTrain,End" and "GetReady,TravelCar,End" because features are detected based on their occurrences in *projected* traces in a log. So, if we project the traces in the log in Table 1 to contain only the tasks in these two sequences, they both will occur in the log and, therefore, are not relevant features. This example shows that the DWS analysis plug-in *not only finds out that the model is more general than necessary, but it also indicates the points of over-generalization*. This information is especially useful when many parts of a model capture the right level of abstraction and just a few parts do not. In these situations, the DWS analysis or mining plug-ins could guide a designer in modifying the model to make it more precise.

Our approach can also be used to identify common patterns in the log. In these situations, the values for the *sigma* and *gamma* parameters may overlap because the whole feature may also be a common pattern in the log. For instance, if one wants to find out the patterns that happen at least in half of the traces in the log, one can set $sigma = 0.5$ and $gamma = 1$. With these settings, one is saying that the subparts of the feature (i.e., "$t_1...t_n$" and "$t_n t_{n+1}$") happen in more than 50% of the traces in the log and its combination (i.e., "$t_1...t_n t_{n+1}$")

happens in (i) all traces, (ii) some traces or (iii) none of the traces in the log. As an illustration, let us try to find out the most frequent patterns (above 50%) in the log in Table 1. The results are in Figure 6. As expected, all the patterns identified in the list of features returned by the DWS analysis plug-in occur in process instances 1 and 3 in the log. Together, these process instances correspond to 53.6% of the behavior in the log (cf. Table 1). Based on the ten identified features, the log was partitioned into two clusters: "R.0" contains the process instances 1 and 3, and "R.1" has the process instances 2 and 4.

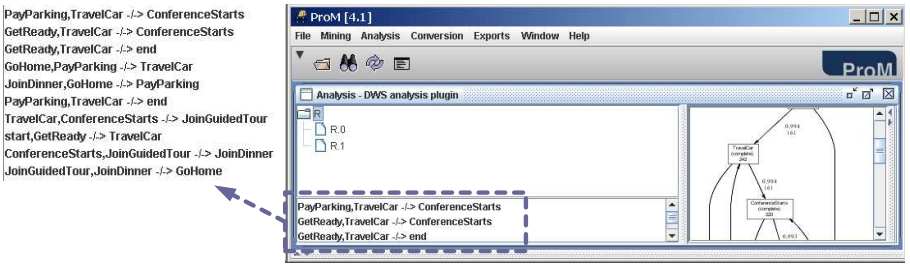

Fig. 6. Screenshot of the result of applying the *DWS analysis* plug-in to detect *common patterns in the log*. The model, log and configuration parameters are the same as in Figure 6, except for the parameters "Frequency support - sigma" and "Frequency relevance threshold - gamma" which were respectively set to 0.5 and 1.

5 Related Work

This section reviews the techniques that that have been used to detect over-general mined models in the process mining domain. Greco et al. [4] have defined the *soundness* metric, which receives a model and a log as input and calculates the percentage of traces that a model can generate and are not in the log. Since the log is assumed to be exhaustive, this metric only works for *acyclic* models. Rozinat et al. [7] have created two notions of *behavioral appropriateness* (a_B), both based on a model and a log. The *simple* a_B measures the average number of enabled tasks while replaying a log in a model. The problem with this metric is that it does not take into account *which* tasks are enabled. The *complex* a_B calculates the so-called "sometimes" predecessor and successor binary relations for tasks in a log and tasks in a model. The problem here is that the relations are global and binary. So, the approach cannot identify over-generalizations that involve more than two tasks. Alves de Medeiros et al. [8] have defined the *behavioral precision* (B_P) and *behavioral recall* (B_R) metrics, which work by checking how much behavior two models have in common with respect to a given log. Although the metrics quantify the degree of over-generalization in a mined model, they have the drawback that they require a base model (in addition to the mined model and the log). Van Dongen et al. [3] have defined the *causal footprint* metric, which assesses behavior similarity of two models based on their *structure*. Like the behavioral precision/recall metrics, the problem with the causal footprint is that a base model is also required. The approach presented in this paper differs from the previously discussed ones because it not only detects that a mined model is over-general, but it also highlights where the over-general points

are. Additionally, it only requires a log and a model to run (i.e., no need for a base model).

6 Conclusions and Future Work

This paper has introduced an approach for mining precise models by clustering a log. The approach has been implemented as two ProM plug-ins: the *DWS analysis* and the *DWS mining*. The DWS analysis plug-in can be used to detect *points of over-generalization* in a model or *frequent patterns* in a log. The DWS mining plug-in provides a way to mine a hierarchical tree of process models by using the *Heuristics Miner*, a pre-existing ProM plug-in that is robust to noise and can handle most of the common control-flow constructs in process models. By decoupling the feature selection and clustering steps from a specific mining algorithm, this paper has shown how to broaden the reach of the techniques in [4] such that other process mining algorithms can easily use them. Future work will focus on (i) allowing for the definition of intervals for the thresholds *sigma* and *gamma*, and (ii) removing the constraint that features are possible (sub-)paths in a model. This way it would be possible to identify features whose two sub-components are not directly connected.

References

1. W.M.P. van der Aalst and A.J.M.M. Weijters, editors. *Process Mining*, volume 53 of *Special Issue of Computers in Industry*. Elsevier Science Publishers, Amsterdam, 2004.
2. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In J.B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB*, pages 487–499. Morgan Kaufmann, 1994.
3. B.F. van Dongen, J. Mendling, and W.M.P. van der Aalst. Structural Patterns for Soundness of Business Process Models. In *EDOC '06: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 116–128, Washington, DC, USA, 2006. IEEE Computer Society.
4. G. Greco, A. Guzzo, L. Pontieri, and D. Sacca. Discovering expressive process models by clustering log traces. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1010–1027, 2006.
5. A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a Review. *ACM Computing Surveys*, 31(3):264–323, 1999.
6. 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.
7. A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In Christoph Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, Berlin, 2005.
8. Wil M. P. van der Aalst, A. K. Alves de Medeiros, and A. J. M. M. Weijters. Process equivalence: Comparing two process models based on observed behavior. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2006.
9. A.J.M.M. Weijters, W.M.P. van der Aalst, and A.K. Alves de Medeiros. Process Mining with HeuristicsMiner Algorithm. BETA Working Paper Series, WP 166, Eindhoven University of Technology, Eindhoven, 2006.