User Assistance During Process Execution - An Experimental Evaluation of Recommendation Strategies

Christian Haisjackl and Barbara Weber

Department of Computer Science, University of Innsbruck, Austria
{Christian.Haisjackl,Barbara.Weber}@uibk.ac.at

Abstract. In today's changing business environment, flexible Process-aware Information Systems (PAISs) are required to allow companies to rapidly adjust their business processes to changes in the environment. However, increasing flexibility poses additional challenges to the users of flexible PAISs and thus requires intelligent user assistance. To address this challenge we have previously proposed a recommendation service for supporting users during process execution by providing recommendations on possible next steps. Recommendations are generated based on similar past process executions considering the performance goal of the supported process. This paper follows up on this work and suggests additional strategies for generating recommendations. In addition, as major contribution of this paper, we investigate how effectively the recommendation strategies work for different processes and logs of different quality.

1 Introduction

In today's fast changing business environment, flexible Process-aware Information Systems (PAISs) are required to allow companies to rapidly adjust their business processes to changes in the environment [1]. Several proposals on how to deal with this challenge have been made [2, 3, 4, 5, 6] relaxing the strict separation of build-time and run-time. By closely interweaving modeling and execution the above mentioned approaches all provide more maneuvering room for the end-users [5]. In particular, users are empowered to defer decisions regarding the exact control-flow to run-time, when more information is available.

With this increase of flexibility, however, additional challenges are imposed to the users of flexible PAISs. A recently performed experiment at the University of Innsbruck shows that with increased flexibility users with little experience have greater difficulties during process execution, which in the worst case may result in process instances that cannot be properly completed [7]. This trade-off between flexibility and support is also described in [8].

To address the above mentioned challenges and to assist users during process execution, we previously presented an approach for intelligent user assistance in flexible PAISs [9]. In particular, we proposed a recommendation service (including an implementation in ProM) which exploits the information available in event logs to guide users during process execution. The recommendation service provides information to users of a flexible PAIS on how to best proceed with a particular process instance depending on the execution state of that instance to best achieve a certain performance goal (e.g., minimizing cycle time, or maximizing profit). The paper also proposed several simple strategies for calculating recommendations. In this paper, we extend the recommendation strategies introduced in [9] with additional ones. In addition, as major contribution of this paper, an experimental evaluation of the recommendation strategies is conducted and the impact of log quality on recommendation quality is investigated.

The results of our experiment show that even though there is no single recommendation strategy which is always outperforming all the others, log-based recommendations are effective and mostly outperform randomly created process instances. Our data further points to the importance of log quality for obtaining recommendations of high quality.

The remainder of this paper is structured as follows. In Section 2 an overview of the recommendation service is provided. Then, Section 3 introduces different miners for finding similar traces and Section 4 elaborates on different recommendation strategies for generating recommendations. In Section 5, we describe the experiment for evaluating the described recommendation strategies. Finally, we discuss related work in Section 6 and provide conclusions in Section 7.

2 Overview of the Recommendation Service

This section gives an overview of the recommendation service and explains how users are supported during process execution (cf. Fig. 1). For a formalization of the recommendation service see [9]. At any point during process execution the user of a flexible PAIS can ask the recommendation service for support on how to proceed with the execution of a particular process instance (1). The recommendation client then sends the user request containing information about the activities which have already been executed by the users for that particular process instance (i.e., *partial trace*) and all *enabled activities* (i.e., all activities the user is able to execute in the next step for this particular process instance) to the recommendation engine (2). The recommendation request is then passed to the pre-configured *recommendation strategy* (3), which determines the algorithm to be used for calculating recommendations. The strategy then consults one of the *miners* (4) to search the log for traces similar to the partial trace (5). The miner compares the *partial trace* with the traces in the event log (i.e., the *log traces*) and determines how well they fit the partial trace. In addition, for each log trace a *weight* (i.e., a number between zero and one) is calculated reflecting the degree of fit with the partial trace. In addition, the miner provides a *result bag* (with the mining results) which is then passed on, together with the weights, to the strategy for further evaluation. (6). Based on the obtained results the strategy evaluates each of the *enabled activities* (i.e., possible activities to be executed next) in respect to the *performance goal* (e.g., minimizing cycle time, maximizing customer satisfaction) and ranks them accordingly. The resulting list of recommendations is then sent to the server (7) and passed on to the the client (8), which returns the recommendations to the PAIS for displaying them to the user (9). After the process instance is completed, the PAIS sends the information about the recently executed process instance to the recommendation engine to update the log and to allow for learning (10).

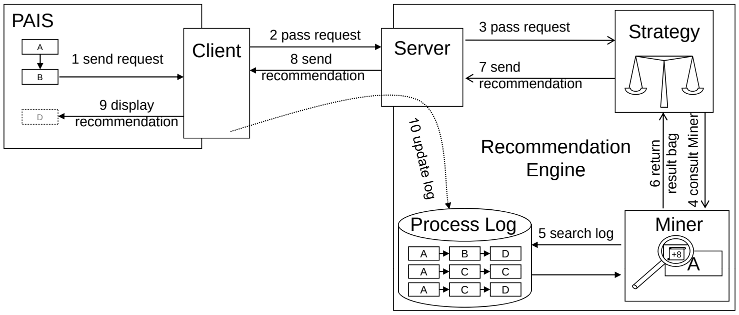


Fig. 1. Overview of the Recommendation Service

3 Finding Similar Log Traces through Miners

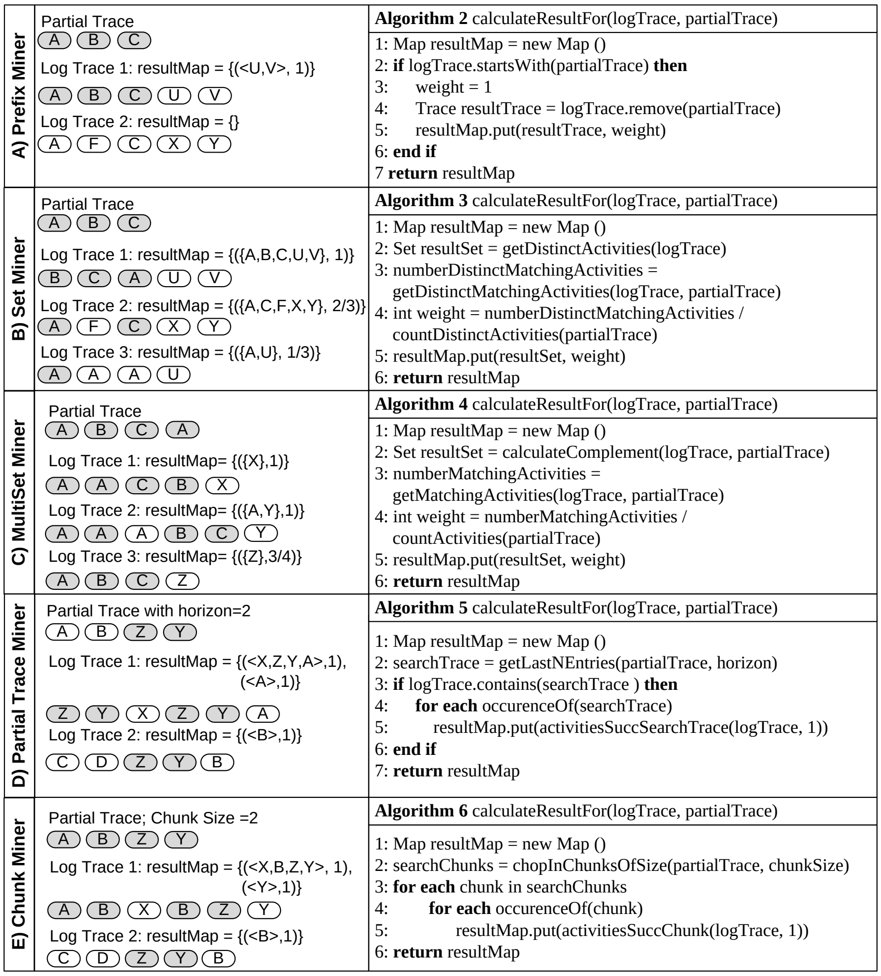
This section introduces different miners for finding similar log traces. In [9], we have already introduced the Prefix Miner, the Set Miner and the MultiSet Miner. This paper adds the Partial Trace Miner, and the Chunk Miner. Each of the miners iterates over the *log* to calculate the result bag for a given *partial trace* (cf. Algorithm 1). Depending on the chosen miner the calculation of the result bag slightly differs (cf. Fig. 2) and is detailed in the following.

Algorithm 1 calculateResultBagFor(logTrace, partialTrace)

```
Bag resultBag = new Bag ()
for each logTrace in log
    resultBag.add(calculateResultFor (logTrace, partialTrace))
return resultBag
```

Prefix Miner: The Prefix Miner considers the exact ordering of activities when comparing the partial trace with a log trace (cf. Fig. 2A). If the partial trace is a prefix of the log trace (Line 2), the log trace obtains a weight of one (Line 3) and the result trace is obtained by removing the partial trace from the log trace (Line 4), otherwise an empty result is returned to the miner (Line 1 + 7). Finally, a resultMap including a result trace and its weight is returned to the miner (Line 5 + 7). Fig. 2A shows an example of the Prefix Miner. Given the partial trace `<A,B,C>`, Log Trace 1 obtains a weight of 1 (since the partial trace is a prefix of the log trace) and result trace `<U,V>`. Log Trace 2, in turn, results in an empty result map since it does not match the partial trace.

Set Miner: In contrast to the Prefix Miner, the Set Miner does not consider the ordering of activities in the log when calculating fitness, but only the presence / absence of activities (cf. Fig. 2B). Thereby, the weight is calculated by dividing


A) Prefix Miner
Partial Trace
A B C
Log Trace 1: resultMap = {(<U,V>, 1)}
A B C U V
Log Trace 2: resultMap = {}
A F C X Y
B) Set Miner
Partial Trace
A B C
Log Trace 1: resultMap = {({A,B,C,U,V}, 1)}
B C A U V
Log Trace 2: resultMap = {({A,C,F,X,Y}, 2/3)}
A F C X Y
Log Trace 3: resultMap = {({A,U}, 1/3)}
A A A U
C) MultiSet Miner
Partial Trace
A B C A
Log Trace 1: resultMap= {({X},1)}
A A C B X
Log Trace 2: resultMap= {({A,Y},1)}
A A A B C Y
Log Trace 3: resultMap= {({Z},3/4)}
A B C Z
D) Partial Trace Miner
Partial Trace with horizon=2
A B Z Y
Log Trace 1: resultMap = {(<X,Z,Y,A>,1), (<A>,1)}
Z Y X Z Y A
Log Trace 2: resultMap = {(<B>,1)}
C D Z Y B
E) Chunk Miner
Partial Trace; Chunk Size =2
A B Z Y
Log Trace 1: resultMap = {(<X,B,Z,Y>, 1), (<Y>,1)}
A B X B Z Y
Log Trace 2: resultMap = {(<B>,1)}
C D Z Y B

Algorithm 2 calculateResultFor(logTrace, partialTrace)

```
1: Map resultMap = new Map ()
2: if logTrace.startsWith(partialTrace) then
3:     weight = 1
4:     Trace resultTrace = logTrace.remove(partialTrace)
5:     resultMap.put(resultTrace, weight)
6: end if
7 return resultMap
```

Algorithm 3 calculateResultFor(logTrace, partialTrace)

```
1: Map resultMap = new Map ()
2: Set resultSet = getDistinctActivities(logTrace)
3: numberDistinctMatchingActivities =
   getDistinctMatchingActivities(logTrace, partialTrace)
4: int weight = numberDistinctMatchingActivities /
   countDistinctActivities(partialTrace)
5: resultMap.put(resultSet, weight)
6: return resultMap
```

Algorithm 4 calculateResultFor(logTrace, partialTrace)

```
1: Map resultMap = new Map ()
2: Set resultSet = calculateComplement(logTrace, partialTrace)
3: numberMatchingActivities =
   getMatchingActivities(logTrace, partialTrace)
4: int weight = numberMatchingActivities /
   countActivities(partialTrace)
5: resultMap.put(resultSet, weight)
6: return resultMap
```

Algorithm 5 calculateResultFor(logTrace, partialTrace)

```
1: Map resultMap = new Map ()
2: searchTrace = getLastNEntries(partialTrace, horizon)
3: if logTrace.contains(searchTrace ) then
4:     for each occurenceOf(searchTrace)
5:         resultMap.put(activitiesSuccSearchTrace(logTrace, 1))
6: end if
7: return resultMap
```

Algorithm 6 calculateResultFor(logTrace, partialTrace)

```
1: Map resultMap = new Map ()
2: searchChunks = chopInChunksOfSize(partialTrace, chunkSize)
3: for each chunk in searchChunks
4:     for each occurenceOf(chunk)
5:         resultMap.put(activitiesSuccChunk(logTrace, 1))
6: return resultMap
```

Fig. 2. Miners for Finding Similar Log Traces

the number of distinct matching activities by the number of distinct activities in the partial trace (Line 3-4). In addition, all distinct activities of the log trace are added to the result set (Line 2). Fig. 2B illustrates the Set Miner for a partial trace `<A,B,C>`. Log Trace 1 obtains a weight of 1, because all activities in the partial trace can also be found in the log trace. Log Trace 2 obtains a weight of 2/3 and Log Trace 3 a weight of 1/3. The result set for Trace 1, for example, is {`A,B,C,U,V`}. The result set for Trace 3, in turn, is {`A,U`}.

MultiSet Miner: Like the Set Miner, the MultiSet Miner does not consider the ordering of activities in the log. However, it takes the number of occurrences of an activity in a log trace into account. Thus, the weight is calculated by dividing the number of matching activities by the number of activities in the partial trace (Line 3-4). In addition, all activities from the log trace minus the activities from the partial trace are added to the result set (Line 2). Fig. 2C shows an example of the MultiSet Miner. Given a partial trace `<A,B,C,A>`, Log Trace 1 and 2 obtain a weight of 1 (i.e., all activities from the partial trace occur exactly as often in the log traces). The result set of Log Trace 1 only contains activity `X`; all other activities are already part of the partial trace. For Log Trace 2 the result set contains activities `A` and `Y`; `A` occurs more frequently in the log trace than in the partial trace and `Y` has not been executed yet. The result set for Log Trace 3 contains activity `Z`, but only with a weight of 3/4 because `A` has been executed in the partial trace once more than in the log trace.

Partial Trace Miner: Like the Prefix Miner, the Partial Trace Miner takes the ordering of activities into consideration (cf. Fig. 2D). However, instead of comparing the entire partial trace with the log traces, it only considers the last n activities of the partial trace (denoted as horizon) (Line 2). All activities succeeding the found search trace(s) are considered as result traces (Line 3-6). Given a partial trace `<A,B,Z,Y>` and a horizon of two, both Log Trace 1 and Log Trace 2 obtain a weight of one (i.e., the last two activities of the partial trace are contained in both log traces). For Log Trace 1 two result traces are obtained, `<X,Z,Y,A>` for the first match and `<A>` for the second match. The result trace for Log Trace 2 is `<B>`.

Chunk Miner: Like the Partial Trace Miner, the Chunk Miner does not compare the entire partial trace with the log traces (cf. Fig. 2E). Instead the partial trace is divided into chunks of size n (i.e., sliding window of size n), each of which is then compared with the log trace (Line 2). All activities succeeding any of the found chunks are considered as result traces (Line 3-5). Fig. 2E shows an example of the Chunk Miner. Given a partial trace `<A,B,Z,Y>` and a chunk size of two, the trace is divided into chunks `<A,B>,<B,Z>,<Z,Y>`, which are then compared with the log traces. Both Log Trace 1 and Log Trace 2 obtain a weight of 1 (i.e., at least one chunk is contained in both log traces). Result traces for Log Trace 1 are `<X,B,Z,Y>` and `<Y>`, while Log Trace 2 results in trace `<B>`.

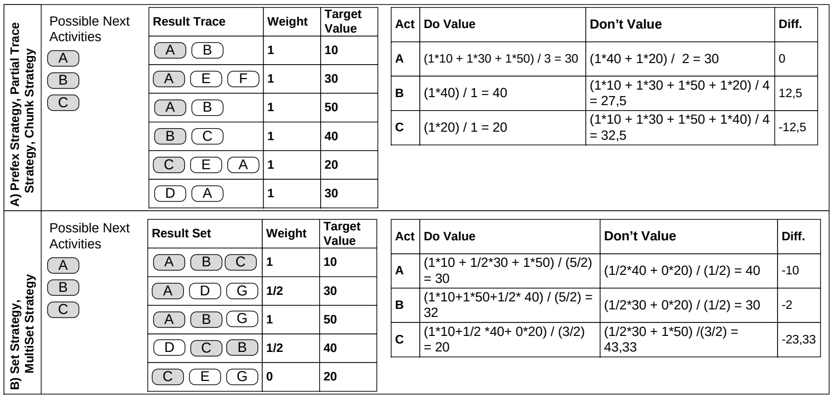
4 Strategies for Generating Recommendations

The miners introduced in the previous section are responsible for weighting log traces according to their fit with the partial trace and provide a result bag containing the mining results (cf. Algorithm 1) which is then taken by the strategies as input for generating recommendations. In particular, based on this information the strategies evaluate all *enabled activities* (i.e., possible next activities) in respect to the *performance goal* (e.g., minimize cycle time). For this, the strategies calculate for each activity a so-called *do value* representing the *expected target value* (e.g., cycle time) a user obtains when executing that particular activity. In addition, a *don't value* representing the expected target value after executing any other activity is calculated. For predicting the expected target value for executing or not executing a particular activity the strategies use historic data from the event log. In particular, a *target function* is applied to all log traces calculating their *target values* (e.g., cycle time for executing the trace). This information is then combined with the weighting information provided by the miners to calculate the do and don't values. The difference of do and don't values is then used by the strategies to provide a sorting of the enabled activities.

If the performance goal is to minimize (maximize) a certain target value (e.g., cycle time) recommendations are sorted by increasing (decreasing) order.

In [9], we have introduced the Randomized Strategy, the Prefix Strategy, the Set Strategy and the MultiSet Strategy. This paper adds the Partial Trace Strategy and the Chunk Strategy. The following describes them briefly.

Randomized Strategy: The Randomized Strategy randomly picks one of the possible next tasks and recommends this task for execution. This strategy can be used to create random traces and is used as baseline for the experiments.



A) Prefex Strategy, Partial Trace Strategy, Chunk Strategy

Possible Next Activities
A
B
C

Result Trace	Weight	Target Value
A B	1	10
A E F	1	30
A B	1	50
B C	1	40
C E A	1	20
D A	1	30

Act	Do Value	Don’t Value	Diff.
A	(1*10 + 1*30 + 1*50) / 3 = 30	(1*40 + 1*20) / 2 = 30	0
B	(1*40) / 1 = 40	(1*10 + 1*30 + 1*50 + 1*20) / 4 = 27,5	12,5
C	(1*20) / 1 = 20	(1*10 + 1*30 + 1*50 + 1*40) / 4 = 32,5	-12,5

B) Set Strategy, MultiSet Strategy

Possible Next Activities
A
B
C

Result Set	Weight	Target Value
A B C	1	10
A D G	1/2	30
A B G	1	50
D C B	1/2	40
C E G	0	20

Act	Do Value	Don’t Value	Diff.
A	(1*10 + 1/2*30 + 1*50) / (5/2) = 30	(1/2*40 + 0*20) / (1/2) = 40	-10
B	(1*10+1*50+1/2* 40) / (5/2) = 32	(1/2*30 + 0*20) / (1/2) = 30	-2
C	(1*10+1/2 *40+ 0*20) / (3/2) = 20	(1/2*30 + 1*50) /(3/2) = 43,33	-23,33

Fig. 3. Examples of Strategies

Prefix Strategy, Partial Trace Strategy, Chunk Strategy: All these strategies use the same method for calculating the do and don’t values and only differ in terms of the used miner. They all consider the first task of each result trace (from the result bag) for calculating the do and don’t values. Traces which do not contain an enabled activity at position one of the result trace are discarded (e.g., Trace 6 in Fig. 3A). For each enabled activity (activities `A,B,C` in Fig. 3A) the do value is then calculated as the weighted average of target values of all traces where the respective activity can be found at position one. The don’t value, in turn, is the weighted average of target values of all traces having another enabled activity at position one in the result trace. Activity `A` in Fig. 3A, for example, obtains a do value of 30 (i.e., weighted average Traces 1-3) and a don’t value of 30 (i.e., weighted average Traces 4-5).

Set Strategy, MultiSet Strategy: Unlike the Prefix Strategy, the Set Strategy and MultiSet Strategy consider all activities of each result set in the result bag for calculating the do and don’t values. Results sets which do not contain any of the enabled activities are discarded (e.g., Trace 6 in Fig. 3B). For each possible next activity the do value is calculated as the weighted average of target values of all result sets containing that particular activity. The don’t value, in turn, is the weighted average of target values of all result sets which do not contain that particular activity (but any other enabled activity). Fig. 3B shows five traces as returned from the Set/MultiSet Miner including their weights. The possible next tasks are activities `A,B,C`. Activity `A`, for example, obtains a do value of 30 (i.e., weighted average Traces 1-3) and a don’t value of 40 (i.e., Trace 4).

5 Evaluation

To evaluate the effectiveness of the suggested recommendation strategies, an experiment has been conducted. The design of the experiment is explained in Section 5.1. Section 5.2 discusses the major results of our experiment.

5.1 Experimental Design

This section introduces the experiment goal, its objects, independent variables and the considered response variable.
Experiment Goal: The main goal of our experiment is to investigate how effectively the recommendation strategies described in Section 4 work depending on the business process and on the log quality.
Object: The recommendation strategies were tested using six distinct processes. *Process A* consists of five activities `A,B,C,D,E` which all need to be executed exactly once, no matter in which order. Usually, the cycle time for executing this process instance is 50. However, if Activity `B` is executed exactly before Activity `C` the cycle time will be reduced to 35 (due to reduced set-up times). This process model has already been used in [9] and can thus serve as a reference.
Process B is relatively well structured and consists of a parallel branching at the beginning, which is followed by an exclusive branching where one activity from the set {`G,H,I`} has to be executed. The cycle time will be 60 if `G` is executed, 50 if `H` is executed and 35 if `I` is executed.
Process C comprises ten activities from which exactly three have to be executed, whereby each activity can only be chosen once. If activities `A`, `B` and `C` (in any order) are executed in one process instance the cycle time will be 35. For all other cases the cycle time will be 50.
Process D: Similar to Process C, Process D offers a pool of ten activities from which exactly three distinct activities have to be executed. Activities `A`, `B` and `C` have a cycle time of 10, all other activities have a cycle time of 20. Depending on which activities are executed, the cycle time will thus be 30, 40, 50 or 60.
Process E: provides a pool of 14 activities, which all can be executed at most once. Half of them have a cycle time of 10, the other half has a cycle time of 20. Exactly six of these activities must be executed, resulting in a cycle time of either 60, 70, 80, 90, 100, 110 or 120.
Process F comprises activities `A,B,C,D,E,F`. First, a sequence `<A,B,C>` is executed, followed either by a loop consisting of a sequence `<D,C>` (which can be executed up to three times) or activity `E`. Finally, activity `F` is performed. Whenever `D` is executed the cycle time is reduced by 20 (due to reduced set-up times of activity `E`) resulting in cycle times of 40, 60, 80 and 100.
Independent Variables: In our experiment we consider the recommendation strategy and the log qualiy as independent variables. For variable *recommendation strategy* we consider all strategies described in Section 4. Variable *log quality*, in turn, respresents how well the instances in the log fulfill the performance goal (i.e., minimizing cycle time). Since the process models used for our experiment have different cycle times, mean cycle time of all instances in the log could not be

used as a measure for log quality. Instead, we define log quality as the number of instances in the log falling into a particular cycle time category. The factor levels for variable log quality are calculated according to Algorithm 7 (cf. Fig. 4A), considering the number of cycle time categories of each process and a log size of 30 process instances. Fig. 4B illustrates the obtained factor levels for Process B. A log with factor level `[0;15;15]`, for example, contains 15 instances of cycle time category 2 (50) and 15 instances of category 3 (60).

A.)

Algorithm 7 calculateLogQualityCategories(numberCategories, logSize)

```
ArrayList categoriesList = new ArrayList ()
int[] logCategory = new int[numberCategories]
logCategory[0] = 0
logCategory[numberCategories-1] = logSize
categoriesList.add(logCategory)
for (int i =  numberCategories-1; i >0; i-- )
    for (int j = 1; j <= logSize; j++ )
        logCategory = new int[numberCategories]
        logCategory[numberCategories-1] = logSize-j
        logCategory[i-1] = j
        categoriesList.add(logCategory)
return categoriesList
```

B.) Process B: 3 cycle time categories

[0;0;30]
[0;1;29]
...
[0;29;1]
[0;30;0]
[1;0;29]
...
[29;0;1]
[30;0;0]

C.)

	Categories	Instances	Quality Pts.	Rec. Quality
Process B, Prefix Strategy, Log Quality Category [0;15;15]				
Category 1: cycle time 35	1,00	0	0,00	0%
Category 2: cycle time 50	0,50	30	15,00	50%
Category 3: cycle time 60	0,00	0	0,00	0%
Overall Recommendation Quality			**15 (30)**	**50%**

Fig. 4. Calculating Factor Levels of Log Quality

Response Variable: The response variable in our experiment is the *recommendation quality* of our recommendation strategies when applied to a given process using a log of a given quality. Thereby, recommendation quality measures how well the recommendations fulfill the performance goal. For each process and log quality we measure how often a particular cycle time is obtained when applying a particular recommendation strategy. As illustrated in Fig. 4C, for example, the application of the Prefix Strategy for Process B and a log quality of `[0;15;15]` (i.e., log with 15 instances of cycle time 50 and 15 instances of cycle time 60) resulted 0 times in cylce time 35 and 30 times in cycle time 50 and 0 times in cycle time 60. Depending on the cycle time, different weights are assigned as follows: the best cycle time is weighted with 1 and the worst one with 0. The remaining cycle times obtain values in between. By multiplying the number of process instances with the respective weights, quality points are calculated for each strategy. Since for every combination of process, log quality and strategy 30 process instances are created, a particular strategy can obtain at most 30 quality points for a particular combination. The recommendation quality is then measured as the sum of quality points dived by the maximum number of quality points, for example, 50% in Fig. 4C.

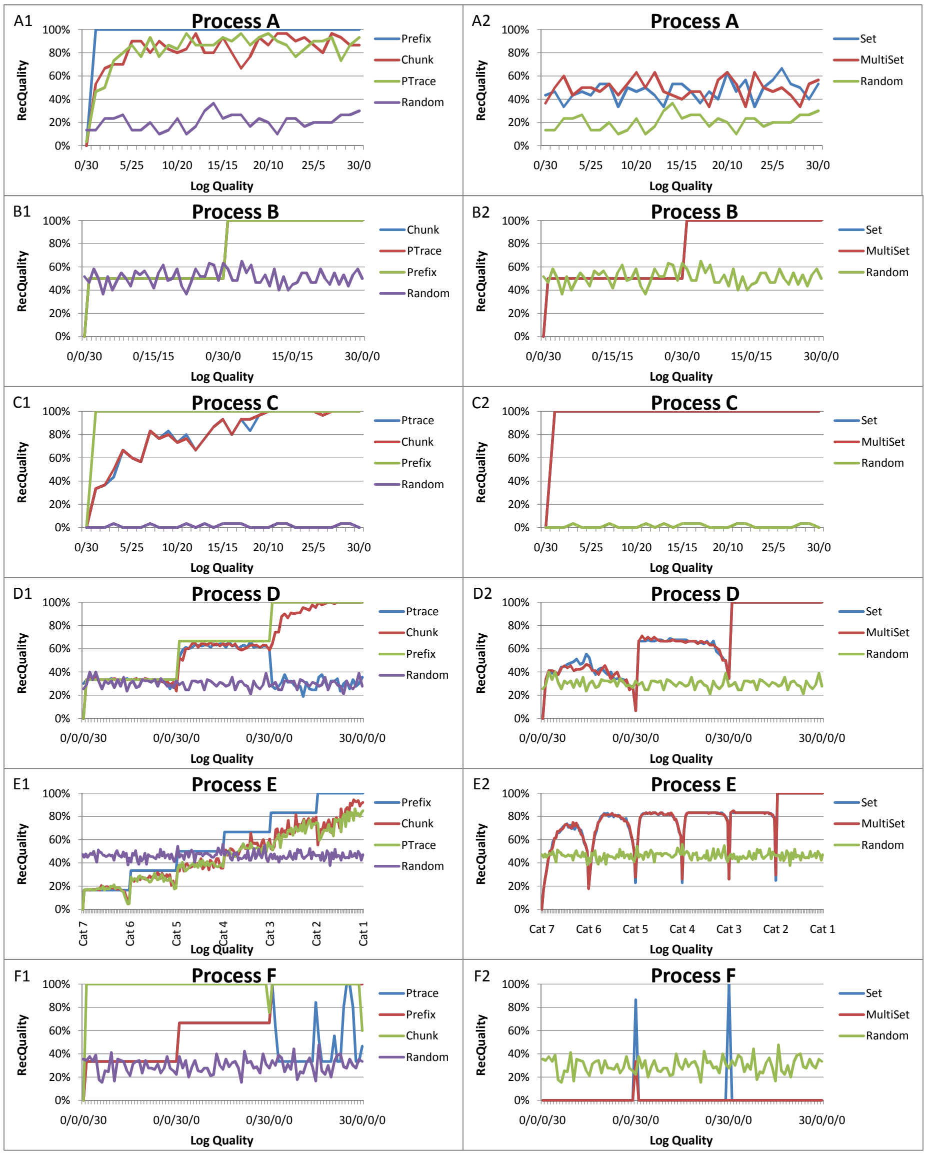
5.2 Experimental Results

This section summarizes major results and illustrates how effectively our recommendation strategies work depending on the log quality (cf. Fig. 5).

Regarding Process A, the Prefix Strategy obtained a recommendation quality of 100% (except for category `[0;30]`) and consistently outperformed all other strategies. This shows that the abstractions used by the other strategies are less appropriate for Process A indicating that they disregard information relevant for that particular process. Process B causes little problems for any of the recommendation strategies (i.e., all strategies led to a recommendation quality correspoding to the best process instance in the log). For Process C, the Prefix Strategy, the Set Strategy and the MultiSet Strategy turned out to be well suited and delivered a recommendation quality of 100% (except for category `[0;30]`). For Process D, the Prefix, Set and MultiSet Strategies showed a good recommendation performance. Again, the Prefix Strategy obtained, for all log categories, a recommendation quality corresponding to the best process instance in the log. The Set Strategy and the MultiSet Strategy, in turn, could even outperform the best process instance in the log for 30 and 32 categories respectively. For additional 44 and 46 categories the recommendation quality corresponds to the log quality. However, for 15 categories both strategies obtained a recommendation quality below the best process instances in the log. The biggest deterioration in quality can be observed for categories `[0;30;0;0]` and `[0;0;30;0]` (cf. Fig. 5D2). In these two cases the log only contains process instances with the same cycle time, leading to the same do and don't values for all activities and thus to a random selection. Regarding Process E, the Set Strategy and the MultiSet Strategy are particularly well suited and delivered for 114 out of 181 categories a recommendation quality better than the best process instance in the log. For additional 60 categories, these strategies obtained a recommendation quality corresponding to the best process instance in the log. Again, like illustrated in Fig. 5E2, a deterioration in quality can be observed whenever the log contains process instances all belonging to the same cycle time category. Finally, regarding Process F the Chunk Strategy showed the best performance[1]. For all categories except `[0;0;0;0]`, `[30;0;0;0]` and `[0;30;0;0]`, the Chunk Strategy obtained a recommendation quality of 100% and outperformed for 59 out of 91 categories the best processs instances in the log (cf. Fig. 5F1). The outliers for categories `[30;0;0;0]` and `[0;30;0;0]` can be explained by looking at both the characteristics of Process F and the do and don't value calculation. After having executed activity `C` there is a choice between performing activity `D` or `E`. However, since the log contains for categories `[30;0;0;0]` and `[0;30;0;0]` 30 process instances with the same traces, all comprising both activity `D` and `E`, the same difference of do and don't values is obtained for both activities. Thus, one of these activities is chosen on a random basis. For Process F both the Set Strategy and the MultiSet Strategy performed very badly, leading to a recommendation quality of 0% for amost all categories. The poor performance of these two strategies can again be explained by the way how do and don't values are calculated. When it comes to selecting between activities `D` or `E`, activity `E` is preferred (as it is mandatory and thus has an undefined don't value, while `D` only appears in some of the traces). For categories `[0;30;0;0]` and `[0;0;30;0]` two peaks can be observed (cf. Fig.

---

[1] Note that we used a chunk size of three in the experiment



Cat 1: 0/0/0/0/0/0/30; Cat 2: 0/0/0/0/0/30/0; Cat 3: 0/0/0/0/30/0/0; Cat 4: 0/0/0/30/0/0/0; Cat 5: 0/0/30/0/0/0/0; Cat 6: 0/30/0/0/0/0/0;
Cat 7: 30/0/0/0/0/0/0

Fig. 5. Experimental Results

5F2). For these two special cases, the same difference of do and don't values is obtained for both activities `D` and `E` and a random selection is performed.

Since no strategy is always outperforming all other strategies, these findings have important implications for the selection of an appropriate recommendation strategy. A consistently good performance is delivered by the Prefix Strategy which provides a recommendation quality of 100% whenever at least one trace in the log has the optimal target value. Consequently, the Prefix Strategy is the best choice whenever logs of high quality are available. However, a downside of the Prefix Strategy is that this strategy never provides recommendations which are better than the best process instance in the log. In contrast, the Set Strategy and the MultiSet Strategy have the potential to outperform the best process instances in the log (e.g., Process D and E). As a drawback of these strategies, however, it has to be considered that they are only suitable for selected processes (e.g., not Process F) and have troubles when all process instances of the log belong to a single cycle time category (cf. outliers in Fig. 5D2 and Fig. 5E2). In the majority of cases, both the Partial Trace Strategy and the Chunk Stategy are outperformed by the Prefix Strategy. However, the Chunk Strategy might bear some potential for processes comprising loops (e.g., Process F).

Limitations: The presented results should be considered recognizing various limitations. First, we evaluated the different recommendation strategies using six different example processes. Although the used processes have different characteristics (i.e., cycle time depends on odering of activities (Process A), presence/absence of activities (Process B-D), execution of loop activities (Process F)), generalizability of our results is certainly limited. Second, the examples used for the evaluation are synthetic examples and not real-world processes. Thus, further evaluations based on more realistic examples have to be conducted.

6 Related Work

The need for user support in flexible systems has been recognized by the research community and several proposals have been made to tackle this issue. Related work in the context of adaptive PAISs aims at facilitating structural process adaptations through change reuse [10, 11]. While the focus of this work is on user support in exceptional situations, our recommendation service assists users during process execution by providing recommendations on what enabled activity to best execute next. In the context of late binding and late modeling basic support for the reuse of previously defined strategies is offered [3, 12]. Similar to our approach, [13] suggests the usage of recommendations to guide users during process execution to meet performance goals of the process best (e.g., lowest cost, shortest remaining cycle time). Unlike our approach, the recommendations are not based on a log, but on a product data model and are thus tailored towards product based workflows. Related to our work is also the approach described in [14] which aims at predicting the completion time of a particular process instance. In contrast to this work, our recommendation service aims at predicting which steps should be executed to achieve certain performance goals best.

7 Conclusion

The increasing flexibility of existing PAISs goes along with an increasing need for user assistance and support. To address this challenge we have previously proposed a recommendation service for assisting users during process execution to best meet the processes' performance goals [9]. This paper extends this work by proposing additional recommendation strategies for calculating recommendations on how to best proceed with a partially executed process instance. To evaluate how effectively the strategies work and to investigate the impact of log quality on recommendation quality we tested through an experiment. Our results indicate that there is no single recommendation strategy which always outperforms all the others. Therefore, depending on the process context a suitable strategy has to be chosen. In addition, our experiments show that traces created using the described recommendation strategies mostly outperform randomly created logs. Moreover, our data also points to the importance of log quality for obtaining high quality recommendations. Future work aims at further experiments based on both real-life scenarios and simulated experiments to obtain more insights into the performance of our recommendation strategies.

References

1. v.d. Aalst, W., Jablonski, S.: Dealing with Workflow Change: Identification of Issues an Solutions. IJCSES **15**(5) (2000) 267–276
2. Reichert, M., Dadam, P.: ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows Without Losing Control. JIIS **10**(2) (1998) 93–129
3. Adams, M., ter Hofstede, A., Edmond, D., van der Aalst, W.: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. In: Coopis'06. (2006)
4. Sadiq, S., Sadiq, W., Orlowska, M.: A Framework for Constraint Specification and Validation in Flexible Workflows. Information Systems **30**(5) (2005) 349 – 378
5. Pesic, M., Schonenberg, M., Sidorova, N., v. d. Aalst, W.: Constraint-Based Workflow Models: Change Made Easy. In: CoopIS'07. (2007) 77–94
6. van der Aalst, W., Weske, M., Grünbauer, D.: Case Handling: A New Paradigm for Business Process Support. DKE **53**(2) (2005) 129–162
7. Zugal, S.: Agile versus Plan-Driven Approaches to Planning - A Controlled Experiment. Master's thesis, University of Innsbruck (October 2008)
8. Dongen, B., Aalst, W.: A meta model for process mining data. In: In Proceedings of the CAiSE WORKSHOPS. (2005) 309–320
9. Schonenberg, H., Weber, B., van Dongen, B., van der Aalst, W.: Supporting flexible processes through log-based recommendations. In: Proc. BPM'08. (2008) 51–66
10. Weber, B., Reichert, M., Wild, W., Rinderle-Ma, S.: Providing Integrated Life Cycle Support in Process-Aware Information Systems. IJCIS **18**(1) (2009) 115–165
11. Minor, M., Tartakovski, A., Schmalen, D., Bergmann, R.: Agile Workflow Technology and Case-Based Change Reuse for Long-Term Processes. International Journal of Intelligent Information Technologies **1**(4) (2008) 80–98
12. Lu, R., Sadiq, S.W.: Managing process variants as an information resource. In: BPM06. (2006) 426–431
13. Vanderfeesten, I., Reijers, H., van der Aalst., W.: Product based workflow support: A recommendation service for dynamic workflow execution. Technical Report BPM-08-03, BPMcenter.org (2008)
14. W.M.P. van der Aalst, M.S., Song, M.: Time prediction based on process mining. Technical Report BPM-09-04, BPMcenter.org (2009)