On Representing, Purging, and Utilizing Change Logs in Process Management Systems

Stefanie Rinderle[1], Manfred Reichert[2], Martin Jurisch[1], and Ulrich Kreher[1]

[1] Dept. DBIS, University of Ulm, Germany
{stefanie.rinderle, martin.jurisch, ulrich.kreher}@uni-ulm.de
[2] Informations Systems Group, University of Twente, The Netherlands
m.u.reichert@utwente.nl

Abstract. In recent years adaptive process management technolgy has emerged in order to increase the flexibility of business process implementations and to support process changes at different levels. Usually, respective systems log comprehensive information about changes, which can then be used for different purposes including process traceability, change reuse and process recovery. Therefore the adequate and efficient representation of change logs is a crucial task for adaptive process management systems. In this paper we show which information has to be (minimally) captured in process change logs and how it should be represented in a generic and efficient way. We discuss different design alternatives and show how to deal with noise in process change logs. Finally, we present an elegant and efficient implementation approach, which we applied in the ADEPT2 process management system. Altogether the presented concepts provide an important pillar for adaptive process management technology and emerging fields (e.g., process change mining).

1 Introduction

The management of log information is crucial in different areas of information systems. One prominent example are transaction logs in database systems which allow to restore a consistent database state after transaction abortions or system crashes. Log information is also exploited for analysis in fields like data mining [1], online analytical processing [2], and process mining [3]. Current process management systems (PMS) maintain comprehensive *execution logs* which capture events related to the start and completion of process activities [4,5].

A key requirement for BPM technology becoming more and more important in practice is (runtime) adaptivity; i.e., the ability of the PMS to support (dynamic) changes at the process type as well as the process instance level. Several approaches have been discussed in literature (e.g. [6,4,5]), and a number of prototypes demonstrating the high potential of adaptive PMS have emerged [7,8]. Obviously, with the introduction of adaptive PMS we obtain additional runtime information about process executions not explicitly captured in current execution logs. This information can be useful in different context and should therefore be managed in respective *change logs*. Change log entries may contain

S. Dustdar, J.L. Fiadeiro, and A. Sheth (Eds.): BPM 2006, LNCS 4102, pp. 241–256, 2006.

information about the type of a change, the applied change operations and their parameterizations, the time the change happened, etc. (cf. Fig. 1).

The kind of change information being logged and the way this information is represented are crucial for the usefulness of change logs. To our best knowledge there has been no profound work related to these fundamental issues so far. Several use cases appear when dealing with change log management. First, execution logs themselves are not sufficient to *restore* the logical structure of a process instance to which ad-hoc changes have been applied (e.g., insertion or deletion of activities). Instead, additional information from change logs is needed. Second, *change traceability* is an important requirement for any adaptive information system. In the medical domain, for example, all deviations from standard procedures have to be recorded for legal reasons. Third, the logged information can be utilized if similar situations re-occur and a previous process change shall be *reused*. Fourth, conflicts between changes concurrently applied to the same process (instance) can be detected based on change log information [9]; i.e., *conflict analyses* can be based on the logged information.

Traceability and change reuse are requirements mainly related to the user level since change information is then presented to and possibly used by human actors. By contrast, restoring process structures after changes and analyzing concurrent changes for the absence of conflicts concern the system level and usually do not involve user interaction. Furthermore, comparable to the use of execution logs in connection with process mining, we must be able to deal with noise in change logs, i.e., information which is unnecessary, irrelevant, or even wrong. Purging change logs from such noise is an important prerequisite, for example, for comparing (conflicting) changes, for reasoning about change effects, and for change mining. However, providing specific *views* on change logs, which hide noisy information, is useful for better user assistance as well, e.g., by providing a homogeneous view on process changes or facilitating their reuse. In summary, the following challenges emerge with respect to change log management:

- How shall change log information be represented in order to meet the described requirements? Which representation form is appropriate at the user level and which one is needed at the system level?
- How can we create purged views on change logs at the *user level* (e.g., to hide 'noise' from users)?
- How can we efficiently store and manage change log information at the *system level*?

In our previous work on adaptive process management (e.g., [10,5]) we have introduced a theoretical framework for dealing with changes at both the process type and the process instance level. In particular we have put emphasis on formal correctness issues arising in connection with dynamic process changes at different levels. In this paper we tackle the above mentioned challenges and introduce a mature approach for representing change information in adaptive PMS. This approach is based on a set of well-defined change operations (applicable at different levels), on change transactions, and on change logs. Further we describe how to create special views on change logs which purge these logs from noisy

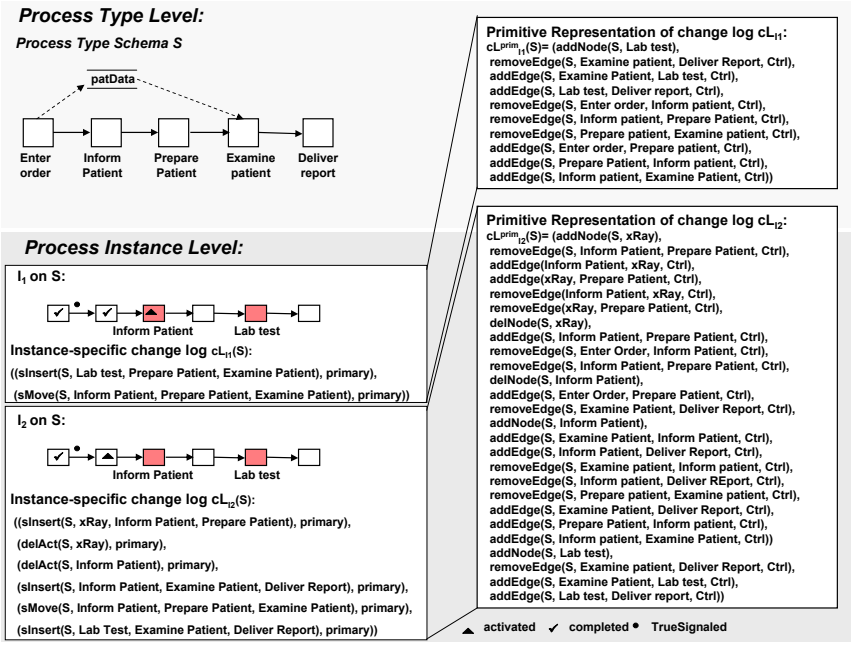


Fig. 1. Change Logs for Modified Process Instances

information (at the user level). Finally, we show how noise-free change logs can be efficiently implemented at the system level.

Sect. 2 deals with basic issues related to change log representation. In Sect. 3 we present an approach for (logically) purging change logs from noise. Sect. 4 shows how change information can be efficiently handled at the system level. Sect. 5 gives an illustrating example. In Sect. 6 we discuss related work and in Sect. 7 we conclude with a summary and an outlook on future work.

2 On Representing Change Logs

We assume a graph–based meta model for defining process templates and representing changes on them. For the sake of simplicity, we restrict our considerations to Activity Nets as, for example, used in MQSeries Workflow [11]. However, our approach can be easily adapted to other process meta models as well.

Logically, a process change is accomplished by applying a sequence of change primitives or operations to the respective process graph (i.e., *process template*). In principle, the change information to be logged can be represented in different ways, which more or less affect the use cases described in Sect. 1. To meet the requirements of these use cases we must find an adequate representation for change log information and appropriate methods for processing it. Independent from the applied (high–level) change operations, for example, we could translate the change into a set of basic change primitives (i.e., graph primitives like

`addNode` or `deleteEdge`). This would still allow us to restore process structures, but also result in a loss of information about change semantics. Consequently, change traceability and conflict analyses would be limited. As an alternative we can explicitly store the applied high–level change operations (incl. their parameterization). We will illustrate both approaches (see also Fig. 1) and discuss their strengths and drawbacks.

We first define the notion of *process template*. For each business process to be supported a process type T is defined. It is represented by a *process template* of which different versions may exist.

Definition 1 (Process Template). *A tuple S with S = (N, D, CtrlEdges, DataEdges, EC) is called a process template, if the following holds:*

- *N is a set of process activities and D a set of process data elements*
- *CtrlEdges* $\subset N \times N$ *is a precedence relation (notation:* $n_{src} \rightarrow n_{dst} \equiv (n_{src}, n_{dst}) \in CtrlEdges$*)*
- *DataEdges* $\subseteq N \times D \times \{read, write\}$ *is a set of read/write data links between process activities and process data elements*
- *EC: CtrlEdges* $\mapsto$ *Conds(D)* $\cup \{TRUE\}$ *where Conds(D) denotes the set of all valid transition conditions on data elements from D.*

For a process template several correctness constraints exist, e.g., (N, CtrlEdges) must be an acyclic graph to ensure the absence of deadlocks (for details see [10,9]).

For defininining changes on a process template two basic approaches (cf. Fig. 2) exist. One approach is to define changes by applying a sequence of basic graph primitives (e.g., inserting or deleting nodes and edges) to the process graph (template). Whether the resulting graph is correct (e.g., does not contain deadlock-causing cycles) or not can be checked, for example, by analyzing the resulting process graph. Tab. 1 summarizes selected change primitives.

Table 1. Examples for Change Primitives on Process Templates

Change Primitive Applied to S	Effects on S
addNode(S,X)	adds node X to template S
delNode(S,X)	deletes node X from template S
addEdge(S,A,B,Ctrl)	adds control edge (A, B) between activities A and B to S
removeEdge(S,A,B,Ctrl)	removes edge (A, B) from S

The other possibility is to use high-level change operations each of which combining change primitives in a certain way (cf. Fig. 2a), e.g., to insert an activity and embed it into the process context. High-level operations comprise more semantics and are characterized by formal pre- and post-conditions. The latter can be used, for example, to ensure correctness when applying a set of operations to a process template. Table 2 presents selected *high-level change operations*. These operations can be applied at the process type as well as the process instance level in order to create or modify process templates. For the

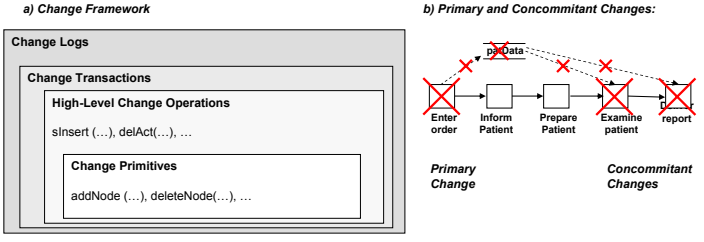


Fig. 2. (a) Overview Change Framework (b) Primary and concomitant Changes

Table 2. Examples for High-Level Change Operations on Process Templates

Change Operation *op* **Applied to S**	opType	subject	paramList	**Effects on S**
		Additive Change Operations		
sInsert(S,X,A,B)	Insert	X	S, A, B	adds activity X between two directly succeeding activities A and B
cInsert(S,X,A,B,c)	Insert	X	S, A, B, sc	adds activity X between two directly succeeding activities A and B as a conditional branch with transition condition c.
		Subtractive Change Operations		
delAct(S,X)	Delete	X	S	deletes activity X from template S and relinks context activities
		Order-Changing Operations		
sMove(S,X,A,B)	Move	X	S, A, B	moves activity X from its current position to the position between directly succeeding activities A and B

two operations *serial move* and *serial insert* Fig. 3 gives more details (incl. pre- and post-conditions and used change primitives).

In order to express more complex changes, high-level change operations can be combined within change transactions (cf. Fig. 2a). This might be needed, for example, if the application of a single change operation would lead to an incorrect process template, but this problem can be overcome by applying a set of concomitant change operations. As example consider the scenario from Fig. 2b). Assume that activity `Enter order` shall be deleted. Due to the existence of data-dependent activities either this change has to be rejected or the two data-dependent activities have to be concomitantly removed to preserve data flow correctness [10]. These concomitant changes must then be carried out within the same change transaction. For change analysis it makes sense to distinguish between such *primary* changes (i.e., changes which initiate the change transactions) and secondary (i.e., *concomitant*) changes (i.e., operations preserving process template correctness afterwards).

As mentioned, process changes may be conducted at the type as well as the instance level. In both cases, several change transactions may be applied during the lifecycle of the process instance or process type respectively. These

sMove(S, X, A, B) with S = (N, D, CtrlEdges, ...)	
Pre-conditions (structural)	- X, A, B ∈ N - ∃ (A, B) ∈ CtrlEdges
Pre-Conditions (state-related)	$NS^S(X)$, $NS^S(B)$ ∈ {NotActivated, Activated}
Change Primitives	- removeEdge(S, A, B, Ctrl) - removeEdge(S, pred(X), X, Ctrl) - removeEdge(S, X, succ(X), Ctrl) - addEdge(S, pred(X), succ(X), Ctrl) - addEdge(S, A, X, Ctrl) - addEdge(S, X, B, Ctrl)
Post-Conditions	- $NS^S(B)$ = Activated → $NS^S(B)$ = NotActivated and $NS^S(X)$ = Activated - $NS^S(B)$ = NotActivated and $NS^S(X)$ = Activated → $NS^S(X)$ = NotActivated
sInsert(S, X, A, B) with S = (N, D, CtrlEdges, ...)	
Pre-conditions (structural)	- X ∉ N - A, B ∈N - ∃ (A, B) ∈ CtrlEdges
Pre-Conditions (state-related)	$NS^S(B)$ ∈ {NotActivated, Activated}
Change Primitives	- addNode(S, X) - removeEdge(S, A, B, Ctrl) - addEdge(S, A, X, Ctrl) - addEdge(S, X, B, Ctrl)
Post-Conditions	$NS^S(B)$ = Activated → $NS^S(B)$ = NotActivated and $NS^S(X)$ = Activated
delAct(S, X) with S = (N, D, CtrlEdges, ...)	
Pre-conditions (structural)	X ∈ N
Pre-Conditions (state-related)	$NS^S(X)$ ∈ {NotActivated, Activated}
Change Primitives	- removeEdge(S, pred(X), X, Ctrl) - removeEdge (S, X, succ(X), Ctrl) - delNode(S, X) - addEdge(S, pred(X), succ(X), Ctrl)
Post-Conditions	$NS^S(X)$ = Activated → NS^S(succ(X)) = Activated
pred(X) (succ(X)) denotes all direct predecessors (successors) of X in S	

Fig. 3. Serial Move/ Serial Insert Operation with Pre- and Post-Conditions (when applying it to a process instance; NS: activity state)

transactions are logically grouped in the *change log* of the instance or type.[1] In Def. 2 we formally define *change transaction* and *change log*. We base this definition on the notion of a process template independent from whether this template is related to a process type or process instance.

Definition 2 (Change Transaction, Change Log). *Let S = (N, D, ...) be a process template. A sequence of change transactions* $cL = < \Delta_1, ..., \Delta_k >$ *applied to S is denoted as process change log. Thereby each change transaction* $\Delta_j := < (op_1^j, cK_1^j), ..., (op_{n_j}^j, cK_{n_j}^j) >$ *(j = 1, ... , k) consists of a sequence of high-level change operations* $op_1^j, \ldots, op_{n_j}^j$ *where either all operations were successfully applied or none of them (atomicity). Flag* $cK_k^j \in \{primary, concomitant\}$ *indicates whether* op_k^j *is a primary change operation or a concomitant one*[2].

In our implementation we maintain additional attributes for change log entries (e.g., time stamps). However this is outside the scope of this paper.

[1] For the sake of readability we use single process instances or process types as granule for a change log.

[2] A change transactions Δ may also consist of exactly one change operation *op*. In this case we write *op* instead of Δ for short and set cK to primary.

Since all transactions Δ_j preserve correctness, the intermediate process templates S_j resulting after the application of change Δ_j are correct. Formally: S + Δ_1:= S_1, S_1 + Δ_2 := S_2, ... , S_{k-1} + Δ_k := S_k are correct process templates. In addition state-related correctness is checked when applying instance changes [5]. However these checks are not based on change logs but on execution logs.

For several reasons it makes sense to maintain both of the aforementioned representation forms for changes in respective logs; i.e., representation of the change as a set of high-level operations and as a set of low-level change primitives. On the one hand, high-level operations are user-friendly and capture more change semantics, on the other hand low-level change primitives enable efficient conflict checks (as we will discuss later on). Therefore, in addition to change log cL (cf. Def. 2) we introduce cL^{prim} which comprises the primitive represenation of cL, i.e., in cL^{prim} the high-level operations from cL are replaced by the change primitives of the respective high-level operations (cf. Fig. 3). As example take the change scenario from Fig. 1 where both representation forms are depicted.

At runtime new *process instances* can be created and executed based on a process template S. Logically, each instance I is associated with an instance-specific process template $S_I := S + cL_I$[3]. S = S(T,V) denotes the original process template from which I was derived, whereby T denotes the process type and V the version of the process type template; cL_I constitutes the instance-specific change log which contains all changes applied to I so far.

The current execution state of I is represented by a marking (NS^{S_I}, ES^{S_I}). It assigns to each activity n and to each control edge e its current status $NS(n)$ or $ES(e)$ respectively. Further, execution history $\mathcal{H}_\mathcal{I}$ captures events related to the start and completion of activities. Based on S, $\mathcal{H}_\mathcal{I}$ and cL^I the current structure and state of instance I can be restored at any point in time.

Definition 3 (Process Instance). *A process instance I is defined by a tuple (T, V, cL^I, M^{S_I}, $\mathcal{H}_\mathcal{I}$, Val^{S_I}) where*

- *T denotes the process type and V the version of the process template S := $S(T,V)$ = (N, D, CtrlEdges, ...) instance I was derived from. We call S the original template of I.*
- *Change log cL^I captures the instance-specific change transactions Δ_i^I (i = 1, ..., n) applied to I so far. We also denote cL^I as bias of I. $S_I := S + cL^I$ (with $S_I = (N_I, D_I, \ldots)$) resulting from the application of cL^I to S is called instance–specific template of I.*
- *M^{S_I} = (NS^{S_I}, ES^{S_I}) describes node and edge markings of I:*
  NS^{S_I}: $N_I \mapsto$ {`NotActivated`, `Activated`, `Running`, `Completed`, `Skipped`}
  ES^{S_I}: ($CtrlEdges_I$) $\mapsto$ {`NotSignaled`, `TrueSignaled`, `FalseSignaled`}
- *$\mathcal{H}_\mathcal{I}$ denotes the execution history of I which captures events related to the start and completion of activities*
- *Val^{S_I} is a function on D_I. It reflects for each data element $d \in D_I$ either its current value or the value* `UNDEFINED` *(if d has not been written yet).*

[3] For unchanged instances $cL_I = \emptyset$ and consequently $S_I = S$ holds.

3 The Logical View – On Purging Change Logs

After having defined the notion of change log we now have a closer look at the information captured by such logs. This makes sense since changes with same effects can be expressed in different ways and therefore be represented by different sets of change operations. As example consider Fig. 1 (left side). Though the changes captured by cL_{I_2} and cL_{I_2} comprise different operations, at the end they have resulted in equal schemes for instances I_1 and I_2. When analyzing cL_{I_2} we can observe that this change log contains operations which do not have any effect (e.g., insertion and immediate deletion of activity `xRay`). Reason for the presence of such changes can be that users either do not act in a goal-oriented way (i.e., they "try out" the change) or, e.g. in the medical domain, certain possible steps (treatments) are first considered and discarded later.

For the mentioned use cases (e.g., change mining, conflict checking) logs should only provide relevant information (about those changes which actually have had effects). By contrast, irrelevant or noisy information make checks or the comparison of changes (as necessary when propagating a process type change to biased process instances) difficult. For traceability reasons, by contrast, the logs should exactly reflect the change transactions as applied (independent from their actual effects). Consequently, change log management should provide different *views* on the stored information depending on the respective use case. In this paper we consider two views, the original change log view (containing all change transactions) and the *purged* change log which only reflects change transactions which actually had an effect on the affected process template.

1. Let S be a process template which is transformed into template S' by applying the operations from change log cL. The first group of changes without any effect on S' are *compensating changes*, i.e., changes mutually compensating their effects. Consider the change log as depicted in Fig. 4: activity `xRay` is first inserted (between `Inform Patient` and `Prepare Patient`) and afterwards deleted by the user. Therefore the associated operations `sInsert(S, xRay, Inform Patient, Prepare Patient)` and `delete(S, xRay)` have no visible effects on S'.
2. The second category of noise in change logs comprises changes which only have hidden effects on S'. Such *hidden changes* always arise when deleting an activity which is then re-inserted at another position. This actually has the effect of a move operation. Consider again Fig. 4 where activity `Inform Patient` is first deleted and then inserted again between `Examine Patient` and `Deliver Report`. The effect behind this is the same as of the move operation `sMove(S, Inform Patient, Examine Patient, Deliver Report)`.
3. There are changes overriding effects of preceding ones (note that a change transaction is an ordered set of operations). Fig. 4 depicts a change log where the effect of the hidden move operation `sMove(S, Inform Patient, Examine Patient, Deliver Report))` is overwritten by operation `sMove (S, Inform Patient, Prepare Patient, Examine Patient)`, i.e., in S' `Inform Patient` is finally placed between `Prepare Patient` and `Examine Patient`.

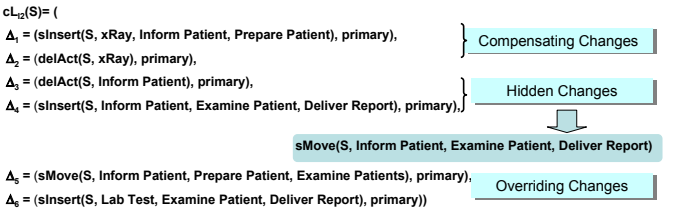


Fig. 4. Different Types of Noise within Change Log

In order to purge a change log from such noise we provide an algorithm for detecting and removing irrelevant or noisy information from change logs. Let $cL =< \Delta_1, ..., \Delta_n >$ be a change log whose application to template $S = (N, D, ..)$ has resulted in template $S' = (N', D', ...)$. We call $N_{cL}^{add} :=$ N' \ N the set of all added activities in S' and $N_{cL}^{del} :=$ N \ N' the set of all deleted activities.

For the sake of readability and without loss of generality we assume that all change transactions Δ_j (j = 1, ..., n) consist of exactly one (primary) change operation op_j (formally: $\forall \Delta_j : \ \Delta_j =< (op_j, \ primary) >$); i.e., we abstain from change transactions comprising multiple operations. However, the algorithm presented in the following can be applied to most complex change transactions as well. Exceptional are only very special cases as the following example shows. Assume that an activity is deleted (primary change) followed by the concomitant deletion of data-dependent steps (e.g., deletion of `Enter order` as depicted in Fig. 2b). Assume further that this activity is re-inserted afterwards, but not all of the other deleted steps. Taking the scenario from Fig. 2b), for example, activities `Enter order` and `Examine` might be re-inserted, but activity `Deliver report` not. Though the primary changes override each other (deletion and insertion of `Enter order`) there is a remaining effect. Consequently the associated change transactions cannot be completely purged from the change log.

Informally the algorithm for purging change logs works as follows: First of all, sets N_{cL}^{add} and N_{cL}^{del} are determined. Taking this information change log cL can be purged. This is accomplished by scanning cL in reverse direction and by determining whether change transaction (operation) $\Delta_j = op_j$ $(j = 1, \ldots, n)$ actually has any effect on S. If so we incorporate $\Delta_j = op_j$ into another – intially empty – change log cL^{purged}. Finally, in order to reduce the number of necessary change log scans to one we use auxiliary sets to memorize which activities, control edges, data elements and data edges have been already treated. The following informal description focuses on the insertion, deletion, and moving of activities in order to get the idea behind the respective algorithm. However, the used methods can be also applied to purge logs capturing information about insertion and deletion of, for example, data elements.

– Assume that we find a log entry $\Delta_j = op_j$ for an operation inserting activity X between activities src and $dest$ into S and that X is not yet present in A (let A be an auxiliary set for which $A = \emptyset$ holds at the beginning), i.e., $\Delta_j = op_j$ is the last change operation within cL which manipulates X. If

X has been already present in S ($X \notin N_{cL}^{add}$) a *hidden change* is found. Consequently, a respective log entry for an operation moving X between *src* and *dest* is created and written into cL^{purged}.

- If log entry $\Delta_j = op_j$ denotes an operation deleting X from S, $X \notin A$, and X is still present in S' ($X \notin N_{cL}^{del}$) we have found a *compensating change*. Therefore $\Delta_j = op_j$ (and the respective insert op.) are left outside cL^{purged}.
- If log entry $\Delta_j = op_j$ denotes an operation moving X to a position between activities *src* and *dest* and $\Delta_j = op_j$ is the last operation within cL having effects regarding X ($X \notin A$) we have to distinguish two cases: If X has been inserted before $\Delta_j = op_j$ ($X \in N_{cL}^{add}$) we write a new log entry in cL^{purged} denoting an operation inserting X between *src* and *dest*. If X has been also present in S ($X \notin N_{cL}^{add}$) we write $\Delta_j = op_j$ unalteredly into cL^{purged}.

A formalization of the method described above is given in Alg. 1. Due to lack of space we restrict this description to serial insert operations. However adopting parallel and branch insertions runs analogously and has been considered in our approach (see [9] for details).

Definition 4 (Purged Change Log). *Let S = (N, D, ...) be a (correct) process template. Let further cL be a change log whose application transforms S into another (correct) process template S' = (N', S', ...). Let* $(N_{cL}^{add} := N' \setminus N$ *and* $N_{cL}^{del} := N \setminus N')$*. Algorithm 1 determines the purged change log* cL^{purged}.

Algorithm 1. `PurgeConsolidate(S, N, N',` $cL = (\Delta_1 = op_1, \ldots, \Delta_n = op_n)) \longrightarrow cL^{purged}$

```
A:=∅; cL^purged = ∅;
N_cL^add := N' \ N; N_cL^del := N \ N';
for i = n to 1 do {
  if (Δ_j = op_j = serialInsert(S, X, src, dest)) {
   if (X ∉ A) {
    A := A ∪ {X}; //X not considered so far
    if(X ∉ N_Δ^add){ //X actually not inserted ⟶ hidden move
     if (src ≠ c_pred(S, X) ∧ dest ≠ c_succ(S, X)[4]){ //X moved to another position?
      cL^purged.addFirst(serialMove(S, X, src, dest))//adds entry at beginning of cL^purged;
    }} else {
     cL^purged.addFirst(serialInsert(S, X, src, dest));}} continue};
 if (Δ_j = op_j = serialMove(S, X, src, dest)) {
   if (X ∉ A) {
    A := A ∪ {X};
    if (X ∈ N_cL^add) {
     cL^purged.addFirst(serialInsert(S, X, src, dest)); } else {
     if (src ≠ c_pred(S, X) ∧ dest ≠ c_succ(S, X)) {
      cL^purged.addFirst(serialMove(S, X, src, dest));}} continue;}
 if (Δ_j = op_j = delete(S, X)) {
   if (X ∉ A) {
    A := A ∪ {X};
    if(X ∈ N_cL^del){
      cL^purged.addFirst(delete(S, X));}}}
  cL^purged.addFirst(op_i);
}
return cL^purged;
```

[4] c_pred(S, X) (c_succ(S, X)) denotes all direct predecessors (successors) of X in S.

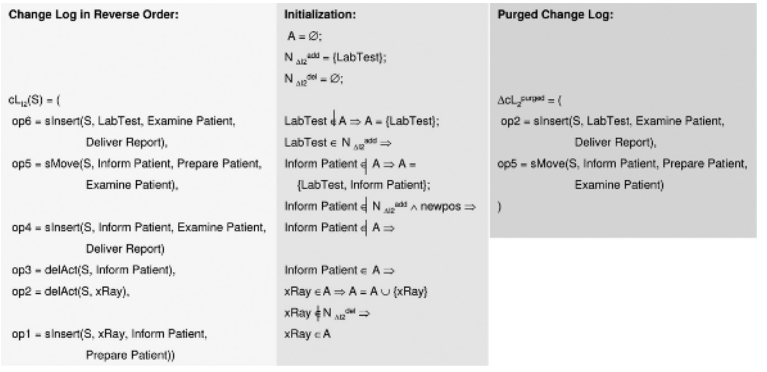


Fig. 5. Purging the Change Log of Instance I_2 (cf. Fig. 4)

Figure 5 depicts how change log cL_{I2} from Fig. 4 is purged resulting in purged change log cL^{purged}. This view just contains those change transactions (operations) which actually have had an effect on the instance-specific template.

Altogether purging change logs in the described way results in a specific, logical view on the conducted changes. This view may, for example, be presented to users if an overview on the actual change effects on the original process template is required. As we will discuss in the next section, at the system level a more efficient approach becomes necessary.

4 The Implementation View – The Delta Layer Concept

In this section we present concepts for representing changes at the system level which have been implemented within the ADEPT prototype. Before presenting the delta layer concept in more detail, some background information on the general representation of process type and process instance templates is needed. Fig. 6a illustrates an approach which has been implemented by several adaptive PMS [8,12]. The *process logic* (e.g., control and data flow) is encapsulated within object *process template* which represents the *process type*. *Instance objects* representing process instances solely contain runtime information (like activity execution states or – logically – the content of data elements). The associated process type is expressed by a reference to the respective process template object. Following this approach, all instances of a given process type reference the same template object. We chose this representation since the necessary storage space is significantly reduced – especially for a large number of running instances – compared to storing a process description for each instance in a redundant way.

In order to reflect the difference between template and instance objects (e.g., after instance changes) we introduced the delta layer concept (cf. Fig. 6b). The delta layer is represented by an object which has the same interfaces as the process template object and therefore offers the same operations. As difference between the delta layer object and the template object the delta layer object

does not reflect the whole process graph but only those parts of the process template which have been changed by instance-specific modifications. Therefore, together with the template object the delta layer object allows to restore the instance-specific template of biased instances. The instance object which represents a biased instance does no longer reference the associated template object but the delta layer object. The delta layer object itself references the original template object and therefore preserves the assocation between instance and process type. Unchanged instances directly reference the original process template object further on.

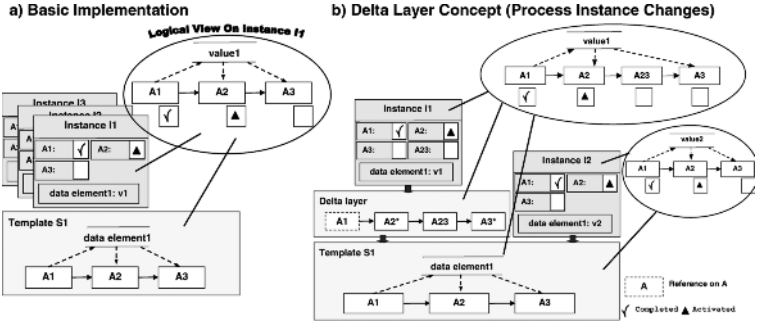


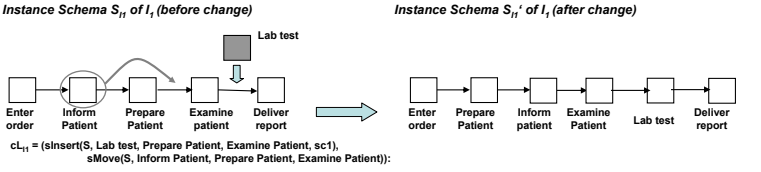
Fig. 6. On Representing Process Template and Process Instance Objects

Fig. 7 depicts how the delta layer concept is realized. As discussed in Sect. 2, at the system level, the (high-level) change operations are translated into change primitives which directly operate on node and edge sets. We represent change information by change log cL and its primitive representation cL^{prim}. The change primitives captured by cL^{purged} are directly stored within the delta layer (e.g., information about added and deleted nodes and edges). For change log cL_{I1}, for example, the set of added nodes and edges as well as the set of deleted edges exactly reflect the "difference" between templates S_{I1} and S'_{I1}.

The "self-purging" effect of storing changes within a delta layer is illustrated by Fig. 8. Change log cL_{I2} contains noise, i.e., information which has to be purged from the change log in order to obtain a "minimal" view on the change effects. Using the delta layer this purging effect is automatically achieved since the change primitives overwrite unnecessary information automatically. For compensating change operations `sInsert(S, xRay, Inform Patient, Prepare Patient)` and `delAct (S, xRay)`, for example, first control edge `(Inform Patient, Prepare Patient)` is removed and re-inserted afterwards such that this change has no effect within the delta layer.

5 Illustrating Example

We illustrate the different concepts presented in this paper by means of an example – a process template evolution with related instance migrations. Consider



cL^{prim}_{I1} =
(addNode(S, Lab test),
removeEdge(S, Examine patient, Deliver Report, Ctrl),
addEdge(S, Examine Patient, Lab test, Ctrl),
addEdge(S, Lab test, Deliver report, Ctrl),
removeEdge(S, Enter order, Inform patient, Ctrl),
removeEdge(S, Inform patient, Prepare Patient, Ctrl),
removeEdge(S, Prepare patient, Examine patient, Ctrl),
addEdge(S, Enter order, Prepare patient, Ctrl),
addEdge(S, Prepare Patient, Inform patient, Ctrl),
addEdge(S, Inform patient, Examine Patient, Ctrl))

Delta Layer for S_H':

inNodes	newEdges	Type	delEdges	Type
Lab test	(Examine Patient, Lab test)	Ctrl	(Examine Patient, Deliver report)	Ctrl
	(Lab test, Deliver report)	Ctrl	(Enter order, Inform Patient)	Ctrl
	(Enter order, Prepare patient)		(Inform Patient, Prepare Patient)	
	(Prepare Patient, Inform patient)		(Prepare Patient, Examine patient)	
	(Inform patient, Examine patient)			

Copy of Internal Representation of S_H

Node	Edges	EType	Data
Enter order	(Enter order, Inform Patient)	Ctrl	…
Inform Patient	(Inform Patient, Prepare Patient)	Ctrl	…
Prepare Patient	(Prepare Patient, Examine patient)	Ctrl	…
Examine Patient	(Examine patient, Deliver report)	Ctrl	…
Deliver report			

Copy of Internal Representation of S_H'

Node	Edges	EType	Data
Enter order	(Enter order, Prepare Patient)	Ctrl	…
Inform Patient	(Prepare Patient, Inform Patient)	Ctrl	…
Prepare Patient	(Inform Patient, Examine patient)	Ctrl	…
Examine Patient	(Examine patient, Lab test)	Ctrl	…
Lab test	(Lab test, Deliver report)	Ctrl	…
Deliver report			

Fig. 7. Process Instance Changes Stored within Delta Layer

the scenario depicted in Fig. 8: Instances I_1, I_2 and I_3 were derived from process type template S and have been individually modified. For I_1 and I_2 activity `Lab test` was inserted between `Examine patient` and `Deliver report`, and activity `Inform patient` was moved to the position between `Prepare patient` and `Examine patient`. For I_3 activity `Inform patient` was moved to the same position as for I_1 and I_2 but, by contrast, activity `Deliver report` was deleted. The instance changes are captured by the logs cL_{I_1}, cL_{I_2}, and cL_{I_3} where cL_{I_2} contains noisy information. The purged view on cL_{I_2} as well as the primitive representations of all change logs are depicted in Fig. 8 as well.

Taking this scenario assume that the process type template S is modified by inserting activity `Lab test` between activities `Examine patient` and `Deliver report` and by moving activity `Inform patient` to the position between `Prepare patient` and `Examine patient`. The associated change log cL^{T1} and the delta layer for the new template version S' capture these changes. When migrating I_1, I_2, and I_3 to S' (after performing required correctness checks [5]) the delta layers of I_1, I_2, and I_3 are purged by the delta layer of S'. This becomes necessary since the instance delta layers must not capture information about changes which are already reflected by the delta layer of the new template version after their migration. For I_1 and I_2, for example, all instance-specific changes are already captured by the delta layer of S'. Thus the delta layer and the resulting change log based on S' become empty. For I_3 the already captured move operation of `Inform patient` is purged from the delta layer of I_3 on S', but the change primitives reflecting the deletion of activity `Deliver report` are still kept. With this the delta layer of I_3 on S' exactly represents the difference between the instance-specific template of I_3 and S'.

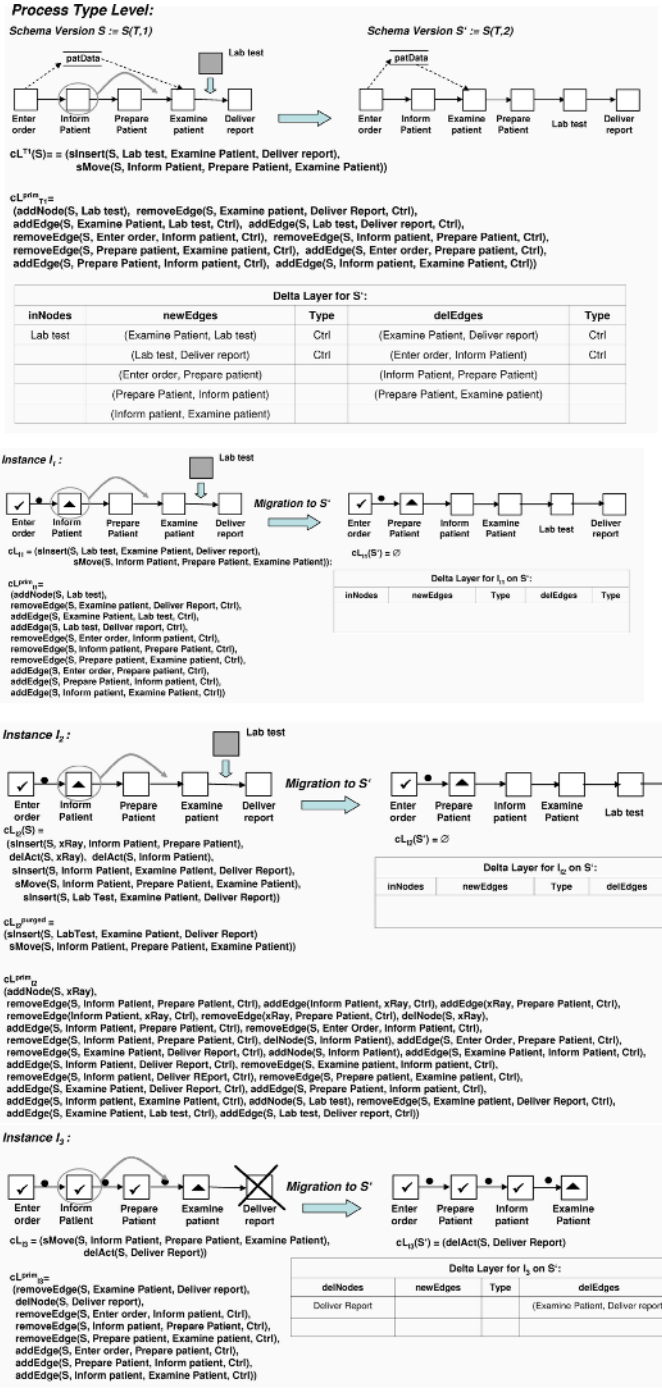


Fig. 8. Process Template Evolution (Example)

Altogether, the change log management illustrated by this example meets all imposed requirements. The applied changes are still traceable at type and instance level due to the full change logs being kept (e.g., change log for I_2). The purged view on, for example, change log cL_{I_2} may be helpful for reusing the change operation. At the system level, the delta layers provide the information necessary for restoring instance-specific templates at any point in time. Furthermore, they constitue the basis for checks (e.g., regarding possible overlaps between changes) and for correctly determining the resulting delta layers and instance-specific changes after instance migration.

6 Related Work

As discussed the management of log information plays an important role in different areas. Examples are recovery in DBMS or data analyses in the context of data mining [1], online analytical processing [2], and process mining [3]. For process mining a meta model representation for execution logs based on MXML format has been developed [13]. In particular, for OLAP and process mining views on logs are built as well (e.g., by clustering [1] or filtering [14]). However, none of these approaches has dealt with change logs so far. Therefore the framework for change log management presented in this paper can be used as basis for an optimized mining of advanced aspects in adaptive PMS (e.g., change mining).

In general, adaptivity in PMS has been a hot topic in literature for many years. Most approaches have focussed on process instance or process type changes and related correctness issues [6,4]. Some approaches have also dealt with both kinds of changes in one system [7,5,8]. However, the representation and organization of the changes themselves has been left pretty vague so far. The approach presented in this paper is complementary to this work.

There are only few approaches dealing with an efficient implementation of advanced process management functionality, [15,7]. So far, they have neglected issues related to change log management. The functionality of existing prototypes are mostly restricted to buildtime and runtime simulations. Using such simulations it can be shown that the particular functionality is realized in principle, but not how it can be implemented in a performant way in practice. Our ADEPT system is one of the very few available research prototypes for adaptive, high-performance process management [12].

7 Summary and Outlook

We have presented an approach for the management of change logs in PMS facing requirements of different uses cases. In order to meet these requirements we have distinguished between the representation of change information at the user and the system level (high-level operations vs. primitives). Based on this we have defined change primitives and operations as well as change transactions. A special view on change logs, the so called purged change logs, has been introduced in order to present the actual change effects to users (e.g., for reuse purposes). For

the system level, we have presented the counterpart based on change primitives stored within a delta layer. An example on correctness checks in the context of process template evolution and individually modified process instances has illustrated the presented concepts.

In future we want to use our change management approach for advanced application scenarios. One example is the mining of change logs in order to, for example, derive process type changes from process instance logs. Furthermore, the presented results are to be transferred to other types of change logs (e.g., logs capturing information on changes of organizational models [16]) as well. Finally we intend to formalize our approach to derive change logs from delta layer information which can be used, for example, to calculate differences between changes. This is necessary, for example, to store correct instance-specific changes after migration to a changed process type template.

References

1. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Academic Press (2001)
2. Bauer, A., Günzel, H.: Data Warehouse Systems. dpunkt (2004)
3. v.d. Aalst, W., van Dongen, B., Herbst, J., Maruster, L., Schimm, G., Weijters, A.: Workflow mining: A survey of issues and approaches. DKE **27** (2003) 237–267
4. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. DKE **24** (1998) 211–238
5. Rinderle, S., Reichert, M., Dadam, P.: Flexible support of team processes by adaptive workflow systems. Distributed and Parallel Databases **16** (2004) 91–116
6. v.d. Aalst, W., Basten, T.: Inheritance of workflows: An approach to tackling problems related to change. Theoret. Comp. Science **270** (2002) 125–203
7. Kochut, K., Arnold, J., Sheth, A., Miller, J., Kraemer, E., Arpinar, B., Cardoso, J.: IntelliGEN: A distributed workflow system for discovering protein-protein interactions. DPD **13** (2003) 43–72
8. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: HICSS-34. (2001)
9. Rinderle, S.: Schema Evolution in Process Management Systems. PhD thesis, University of Ulm (2004)
10. Reichert, M., Dadam, P.: ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. JIIS **10** (1998) 93–129
11. Leymann, F., Altenhuber, W.: Managing business processes as an information ressource. IBM Systems Journal **33** (1994) 326–348
12. Reichert, M., Rinderle, S., Kreher, U., Dadam, P.: Adaptive process management with ADEPT2. In: ICDE'05. (2005) 1113–1114
13. van Dongen, B., van der Aalst, W.: A meta model for process mining data. In: CAiSE'05 Workshops. (2005) 309–320
14. van Dongen, B., de Medeiros, A., Verbeek, H., Weijters, A., van der Aalst, W.: The ProM framework: A new era in process mining tool support. In: ICATPN'05. (2005) 444–454
15. Weske, M.: Object-oriented design of a flexible workflow management system. In: ADBIS98. (1998) 119–131
16. Rinderle, S., Reichert, M.: On the controlled evolution of access rules in cooperative information systems. In: CoopIS'05. (2005) 238–255