Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications*

Stefanie Rinderle, Manfred Reichert, and Peter Dadam

University of Ulm, Faculty of Computer Science,
Dept. Databases and Information Systems
{rinderle, reichert, dadam}@informatik.uni-ulm.de

Abstract. Adaptive process–aware information systems must be able to support ad–hoc changes of single process instances as well as schema modifications at the process type level and their propagation to a collection of related process instances. So far these two kinds of (dynamic) process changes have been mainly considered in an isolated fashion. Especially for long-running processes, however, it must be possible to adequately handle the interplay between type and instance changes as well. One challenge in this context is to determine whether concurrent process type and process instance changes have the same or overlapping effects on the original process schema or not. Information about the degree of overlap is needed, for example, to determine whether and – if yes – how a process type change can be propagated to individually modified process instances as well. This paper provides a formal framework for dealing with overlapping and disjoint process changes and presents adequate migration strategies depending on the particular degree of overlap. In order to obtain a canonical representation of changes an algorithm is introduced which purges change logs from noisy information. Finally, a powerful proof-of-concept prototype exists.

1 Introduction

To stay competitive at the market for companies it becomes more and more important to adequately support their business by process–aware information systems (PAIS) [1]. Doing so it is not sufficient to implement business processes only once and to let the PAIS then run eternally without any adaptations. In fact the ability to quickly react to market changes or exceptional situations by appropriate process changes is key to success [2,3,4,5,6,7]. Basically, in a PAIS changes can take place at two levels – the *process type* or the *process instance* level. Process type changes become necessary, for example, to adapt the PAIS to optimized business processes or to new laws [8,9]. In particular, applications supporting long-running processes (e.g., handling of leasing contracts or medical treatments) and the process instances controlled by them are affected by such type changes [8,9]. As opposed to this, changes of single process instances have

* This work was done within the research project "Change management in adaptive workflow systems", which is funded by the German Research Community (DFG).

R. Meersman, Z. Tari (Eds.): CoopIS/DOA/ODBASE 2004, LNCS 3290, pp. 101–120, 2004.

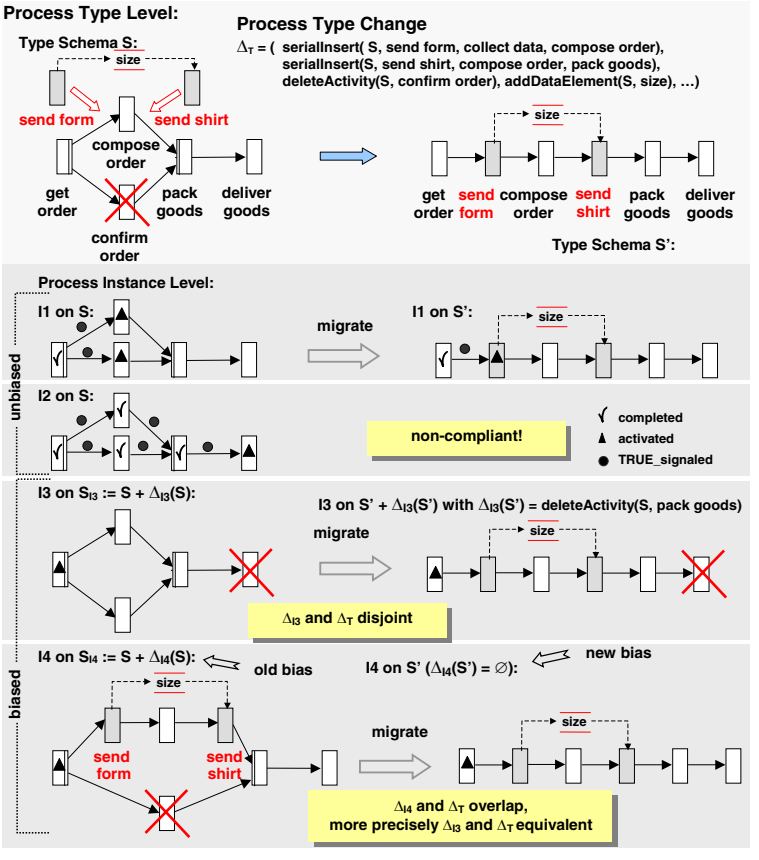

Fig. 1. Process Type and Instance Changes (Example)

often to be carried out in an ad-hoc manner in order to deal with an exceptional situation or evolving process requirements [8,9].

Process type changes are handled by modifying the respective *process schema*. Very often it is desired to *propagate* a process type change to related process instances as well. Process instances for which this is possible are called *compliant*, i.e., they can be *migrated* to the new process schema [3,10]. Adapting a single process instance during runtime, in turn, logically results in an instance-specific schema (i.e., a *process instance schema* differing from the original schema this instance was created from). In the following, we call such individually modified process instances *biased* (e.g., instances I_3 and I_4 in Fig. 1).

Currently there are only few adaptive process management systems (PMS) which support both kinds of changes in one system [7,11]. All these PMS have in common that once an instance has been individually modified (i.e., it possesses an instance-specific process schema due to an ad–hoc change), it can no longer benefit from process type changes; i.e., changes of the schema they were originally

created from. However, doing so is not sufficient in many cases, especially in connection with long-running processes as we have learned from several case studies within medical and automotive environments. In order to come to a complete solution, therefore, it must be possible to propagate process schema changes are carried out at the type level to biased instances as well.

When analyzing the interplay between process type and process instance changes we are faced with several challenges. In [8] we have already discussed the problem of *structural and state–related* conflicts that may arise when propagating a process type change to a biased process instance. Structural conflicts between type and instance changes, for example, may lead to deadlock–causing cycles or incomplete input data for activity executions [8].

Another fundamental issue not treated so far concerns the handling of overlapping type and instance changes; i.e., the handling of concurrent changes[1] on a process schema that partially have the same effects on this schema. In this paper we give insights into fundamental challenges and solution approaches for coping with such *overlapping* changes. One example is depicted in Fig. 1 where process type change Δ_T and process instance change Δ_{I_4} (of instance I_4) both insert activities *send form* and *send shirt* (into schema S). Propagating type change Δ_T to instance–specific schema S_{I_4} would therefore lead to multiple insertion of the same activities. Usually, this would not correspond to the user's intention who, for example, has already anticipated a process optimization by an ad–hoc modification at the instance level. Furthermore Δ_T and Δ_{I_4} both delete the same activity *confirm order*. As a consequence Δ_T actually could not be applied to S_{I_4} since *confirm order* is not longer present.

One prerequisite to adequately deal with such cases is to effectively detect whether (concurrent) process type and process instance changes overlap. Another challenge is to correctly migrate biased process instances to a modified type schema even if the instance–specific changes overlap with the process type change. Basically the problem is that the current representation of the instance–specific schema, which is based on original schema S plus bias $\Delta_I(S)$, must be transformed into a representation based new schema S' plus bias $\Delta_I(S')$. Doing so offers several advantages: If I is actually re–linked to S' it can benefit from further process optimizations of S'. Furthermore, reassigning instances to their actual schema version contributes to an optimal management and redundancy–free storage of process schemes and instances. Looking again at instance I_4 from Fig. 1 we can observe that Δ_T and Δ_I do exactly the same, i.e., they have the same effects on the original process schema S. We therefore call them *equivalent*. For the above reasons, for equivalent changes a desired *migration strategy* would be to abstain from any propagation of Δ_T on I_4 but to re–link or migrate I_4 to S'. In the latter case, representation of I_4 on S' would no longer require maintenance of an instance–specific change, i.e., $\Delta_I(S') = \emptyset$ (cf. instance I_4 on S' in Fig. 1). Assume now that an additional activity *send reminder* has been

[1] In the following, we assume that certain instance–specific changes took place before the process type change occurs. Nevertheless, we call such changes concurrent since they work on the same original process schema.

inserted into I_4. Then Δ_T and Δ_{I_4} would no longer be equivalent but Δ_T be *subsumed* by Δ_{I_4}. For this case an adequate migration strategy is to migrate I_4 to S' (i.e., to re–link I_4 to S') but to further maintain the insertion of *send reminder* as instance–specific change Δ'_{I_4} based on S'. We conclude that for any adaptive PMS it becomes necessary to detect whether process type and process instance changes overlap, and to also determine the *degree of overlap*. This, in turn, is fundamental in order to apply adequate migration strategies.

In this paper we provide fundamental definitions for *disjoint*, *overlapping*, and *equivalent* process changes. Doing so is important in order to be able to provide adequate migration strategies. We illustrate this by means of selected scenarios. Based on formal definitions for disjoint and overlapping process changes we discuss different approaches for detecting them. Thereby *structural*, *operational*, and *hybrid* approaches are presented and estimated along their specific strengths and limitations. We derive an adequate approach to detect to which degree concurrent process changes overlap. This approach comprises a sophisticated method to *purge* unnecessary information (*noise*) from change transaction logs, i.e., we aim at finding a canonical respresentation of change transaction logs. Such noise within change logs, for example, may result from mutually compensating changes. Furthermore, taking purged change transaction logs the necessary information to decide on the degree of overlap between concurrent changes is extracted. Altogether, this method provides the basis for being able to apply adequate migration strategies for any kind of biased instance.

The remainder of this paper is organized as follows: In Section 2.1 we shortly introduce WSM Nets as the process meta model taken to illustrate the presented results. The formal framework – definitions for disjoint, overlapping and equivalent changes – as well as migration strategies are provided in Section 2.2. In Section 3 we discuss different approaches for detecting the degree of overlap between process type and process instance changes and a method to purge noise from change transaction logs in Section 4. We close with a discussion of related work in Section 5 and a summary in Section 6.

2 Disjoint and Overlapping Process Changes

In this paper, we exemplarily use WSM Nets (as for example applied in ADEPT [9]) and the change operations based on them. However, most of the presented results are independent of the used process meta model. Section 2.1 gives background information on WSM Nets necessary for further understanding of the paper. Based on this, Section 2.2 introduces definitions for diesjoint an overlapping changes and exemplarily presents migration strategies for selected scenarios.

2.1 Background Information

A process schema is represented by attributed, serial-parallel process graphs with additional links for synchronizing parallel paths [6].

Definition 1 (WSM Net). *A tuple S = (N, D, NT, CtrlE, SyncE, LoopE, DataE) is called a WSM Net if the following holds:*

- *N is a set (bag) of activities and D a set of process data elements*
- *NT:* $N \mapsto$ {`StartFlow, EndFlow, Activity, AndSplit, AndJoin, XOrSplit, XOrJoin, StartLoop, EndLoop`}
  NT assigns to each node of the WSM Net a respective node type.
- $CtrlE \subset N \times N$ *is a precedence relation*
- $SyncE \subset N \times N$ *is a precedence relation between activities of parallel branches*
- $LoopE \subset N \times N$ *is a set of loop backward edges*
- $DataE \subseteq N \times D \times \{read, write\}$ *is a set of read/write data links between activities and data elements*

A WSM Net S is *structurally correct* if the following constraints hold:

1. S has a unique start node *Start* and a unique end node *End*.
2. Except for nodes *Start* and *End* each activity node of S has at least one incoming and one outgoing control edge $e \in$ CtrlE.
3. $S_{block} := (N, CtrlE, LoopE)$ is structured following a block concept, for which control blocks (sequences, branchings, loops) can be nested but must not overlap.
4. $S_{fwd} = (N, CtrlE, SyncE)$ is an acyclic graph, i.e., the use of control and sync edges must not lead to deadlock-causing cycles.
5. Sync links must not cross the boundary of a loop block; i.e., an activity from a loop block must not be connected with an activity from outside the loop block via a sync link (and vice versa).
6. For activities with mandatory input parameters linked to global data elements it has to be ensured that respective data elements will be always written by a preceding activity at runtime.
7. Parallel write accesses on data elements (and consequently lost updates on them) have to be avoided.

Taking a WSM Net S new process instances can be created and started. *Logically*, each instance I is associated with an instance-specific schema $S_I := S + \Delta_I$ (for unbiased instances $\Delta_I(S) = \emptyset$ and consequently $S_I = S$ holds). The execution state of I is captured by marking function $\mathrm{M}^{S_I} = (\mathrm{NS}^{S_I}, \mathrm{ES}^{S_I})$. It assigns to each activity n its current status $NS(n)$ and to each control edge its marking $ES(e)$. Markings are determined according to well defined marking rules [6], whereas markings of already passed regions and skipped branches are preserved (except loop backs). Concerning data elements, different versions of a data object may be stored, which is important for the context-dependent reading of data elements and the handling of (partial) rollback operations. Formally:

Definition 2 (Process Instance). *A process instance I is defined by a tuple* $(S, \Delta_I, M^{S_I}, Val^{S_I}, \Pi_I^S)$ *where*

- S = (N, D, NT, CtrlE, SyncE, ...) *denotes the process schema I was derived from. We call S the original schema of I.*
- Δ_I *comprises instance-specific changes* $op_1^I, \ldots, op_m^I$ *that have been applied to I so far. We call* Δ_I *the bias of I. Schema* $S_I := S + \Delta_I$ *(with* $S_I = (N_I, D_I, NT, CtrlE_I, \ldots)$*) which results from the application of* Δ_I *to S, is called the instance–specific schema 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}: $(CtrlE_I \cup SyncE_I \cup LoopE_I) \mapsto$
  {`NotSignaled, TrueSignaled, FalseSignaled`}
- 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).*
- $\Pi_I^S = < e_0, \ldots, e_k >$ *is the execution history of I.* $e_0, \ldots, e_k$ *denote the start and end events of activity executions.*

Activities marked as `Activated` are ready to fire and can be worked on. Their status then changes to `Running`. As an example take instance I_1 from Fig. 1: Activity *get order* is completed whereas activity *compose order* is activated. Activities with marking `Skipped` cannot be longer selected for execution.

Table 1 presents a selection of *high-level change operations* which can be used to define or modify WSM Nets. These change operations include formal pre- and post-conditions. They automatically perform the necessary schema transformations whereas schema correctness (cf. correctness constraints 1. – 7. for WSM Nets) is ensured. One typical example of such a change operation is the insertion of an activity and its embedding into the process context.

When applying a series of connected change operations op_i $(i = 1, \ldots, n)$, e.g., when inserting two activities and a data dependency between them, it is often desired to apply either all of these change operations or none of them (atomicity). In order to achieve this, change operations $op_1, \ldots, op_n$ must be carried out within same *change transaction* $\Delta = (op_1, \ldots, op_n)$ (*change* for short).

2.2 Formal Framework

In Sect. 1 we have already introduced the notions of *disjoint* and *overlapping* changes informally. In this section we give formal definitions of these concepts which serve as theoretical underpinning for the following considerations. First of all, we abstract from whether changes are carried out at the type or at the instance level. More precisely, we base our considerations on two arbitrary changes (or change sets) Δ_1 and Δ_2 concurrently applied on the same schema S.

Let S be a (correct) process schema and Δ_1 and Δ_2 two changes which transform S into another (correct) process schema S_1 and S_2 respectively (notation: $S_1 := S + \Delta_1$ and $S_2 := S + \Delta_2$). Generally, disjointness and overlapping are special relations between two changes of the same schema. The challenging question is how to define a relation on changes. Either this can be done by directly

Table 1. A Selection of High-Level Change Operations on WSM Nets

Change Operation *op* **Applied to Schema S**	**Effects on Schema S**
Additive Change Operations	
serialInsert(S, X, A, B)	insertion of activity X between two directly succeeding activities A and B
parallelInsert(S, X, (A, B))	insertion of activity X parallel to control block with start activity A and end activity B
insertSyncEdge(S, src, dest)	insertion of sync edge linking two activities src and dest from parallel execution paths
Subtractive Change Operations	
deleteActivity(S, X)	deletes activity X from schema S
deleteSyncEdge(S, edge)	deletes synchronization edge ∈ SyncE from schema S
Order-Changing Operations	
serialMove(S, X, A, B)	moves activity X from current position to position between directly succeeding activities A and B
Attribute Changing Operations	
changeActivityAttribute(S, X, attr, nV)	changes value of attribute attr of activity X to nV
changeEdgeAttribute(S, edge, attr, nV)	changes value of attribute attr of edge ∈ CtrlE ∪ SyncE to nV
Data Flow Change Operations	
addDataElement(S, d, dom, defVal)	adds data element d with domain dom and default value defVal to S
deleteDataElement(S, d)	deletes data element d from S
addDataEdge(S, (X, d, mode))	adds data edge (X, d, mode) linking activity X with data element d (mode ∈ {read, write})
deleteDataEdge(S, dataEdge))	deletes data edge dataEdge from S

comparing Δ_1 and Δ_2 or by correlating their effects on the original schema S. Effects of Δ_1 and Δ_2 on S, in turn, are reflected by resulting process schemes S_1 and S_2. Consequently, a relation between changes Δ_1 and Δ_2 can be determined by finding a relation between S_1 and S_2. – In the workflow literature several (equivalence) relations for process schemes have been discussed [2,12, 13]. In the context of this work, the relation between concurrent changes affects the *behavior* of the resulting process schemes. Therefore, we base our further considerations on a behavorial equivalence relation for process schemes which is known as *trace equivalence* [10,13].

Definition 3 (Trace Equivalence Between Process Schemes). *Let S_1 and S_2 be two process schemes. S_1 and S_2 are equivalent with respect to their possible traces (formally: $S_1 \equiv_{trace} S_2$) iff each execution history $\Pi_I^{S_1}$ producible on S_1 can be generated on S_2 as well and vice versa.*

Intuitively, two process schemes S_1 and S_2 are trace equivalent if each possible behavior of S_1 (represented by its execution histories) can be simulated by process schema S_2 and vice versa. Based on trace equivalence we now introduce an adequate definition for overlapping and disjoint changes. Intuitively, two change transactions Δ_1 and Δ_2 overlap if they have (partially) the same effects on the underlying process schema S. This is the case if Δ_1 and Δ_2 manipulate the same – already existing – elements of S or insert the same activities into S.

Overlapping effects on already existing elements of a process schema may result from subtractive, order–changing, or attribute–changing operations (cf. Table 1). Subtractive changes that overlap may affect the applicability of Δ_1 on S_2 and vice versa (cf. Fig. 1). Overlapping order–changing and attribute–changing operations may mutually *override* the effects of each other. Assume, for example, that change Δ_1 moves an activity X to position A (resulting in S_1) and Δ_2 moves X to position B (resulting in S_2). Then applying Δ_1 to S_2 would override the effects of Δ_1 and vice versa. Both problems – change applicability and overriding of change effects – can be avoided if Δ_1 and Δ_2 are *commutative*, i.e., applying Δ_2 on S_1 leads to a process schema which is trace equivalent to the process schema that results when applying Δ_1 on S_2. Formally:

Definition 4 (Commutativity of Changes). *Let S be a (correct) schema and Δ_1 and Δ_2 be two changes transforming S into (correct) schema S_1 and S_2 respectively. We call Δ_1 and Δ_2 commutative if the application of Δ_1 to S_2 and the application of Δ_2 to S_1 result in trace equivalent schemes, formally:*

$$\Delta_1,\ \Delta_2 \text{ commutative} \Longleftrightarrow (S + \Delta_1) + \Delta_2 \equiv_{trace} (S + \Delta_2) + \Delta_1$$

Thus commutativity is a first property for characterizing disjoint changes. However, it is not strong enough to cover disjointness of additive changes (e.g., insertions of new activities) as well. In particular, commutativity does not exclude the (undesired) multiple insertion of the same activity (cf. Fig. 1). In order to avoid this effect, we additionally claim that the sets of activities which are newly inserted by Δ_1 and Δ_2 respectively have to be disjoint. Formally:

Definition 5 (Disjoint and Overlapping Changes). *Let S = (N, D, CtrlE, SyncE, DataE, ...) be a WSM Net and Δ_1 and Δ_2 be two change transactions which transform S into WSM Nets S_1 and S_2 with*
$S_i = (N_i, D_i, CtrlE_i, SyncE_i, ...), i = 1, 2$

I) We denote Δ_1 and Δ_2 as disjoint (notation: $\Delta_1 \cap \Delta_2 = \emptyset$) iff the following properties hold:
(1) Δ_1 and Δ_2 are commutative (cf. Def. 4)
(2) $(N_1 \setminus N) \cap (N_2 \setminus N) = \emptyset$[2]

II) We denote Δ_1 and Δ_2 as overlapping (notation: $\Delta_1 \cap \Delta_2 \neq \emptyset$) if they are not disjoint.

As it can be seen from Def. 5 the notion of overlapping concurrent changes is still relatively rough. As indicated in the introduction it is possible to further classify overlapping changes according to their degree of overlap. One of these subclasses is formed by *equivalent* changes, i.e., changes which have exactly the same effects on original schema S. Formally:

Definition 6 (Equivalent Change Transactions). *Let S be a WSM Net and Δ_1 and Δ_2 be two change transactions which transform S into WSM Nets S_1*

[2] We abstract from realization details regarding the concurrent insertion of the same activity. Informally, two process activities are considered as equal iff they use the same activity template and the same semantic identifier.

and S_2. Then Δ_1 and Δ_2 are equivalent, i.e., $\Delta_1 \equiv \Delta_2$ iff S_1 and S_2 are trace equivalent (cf. Def. 3). Formally:

$$\Delta_1 \equiv \Delta_2 \Longleftrightarrow S_1 \equiv_{trace} S_2$$

A very interesting application of Def. 5 and Def. 6 is the correct handling of concurrent process type and process instance changes as described in Section 1. More precisely, based on the particular degree of overlap between process type change Δ_T and process instance change Δ_I (which can be determined based on Def. 5 and 6) different migration strategies have to be applied. To illustrate this, in the following, we present the migration strategies for disjoint and equivalent process type and instance changes.

Policy 1 (Migrating Instances With Disjoint Bias). *Let S be a (correct) process type schema and Δ_T be a process type change which transforms S into another (correct) type schema S'. Let further $I = (S, \Delta_I, \ldots)$ be a process instance on S with instance–specific schema $S_I := S + \Delta_I$. Finally, let Δ_T and Δ_I be disjoint changes (cf. Def. 5), i.e., $\Delta_T \cap \Delta_I = \emptyset$. Then:*

I can correctly migrate to S' preserving Δ_I on S', i.e., $I = (S', \Delta_I, \ldots) :\Longleftrightarrow$

1. *$S_I^* := (S + \Delta_I) + \Delta_T$ is a correct schema (according to the structural correctness constraints 1. – 7. set out for the used process meta model); i.e., Δ_T can be correctly applied to $S_I = S + \Delta_I$* **(Structural Correctness)**.
2. *I is compliant with S_I^*; i.e., the (reduced) execution history Π_I^S of I on S_I can be produced on S_I^* as well* **(State-Related Correctness)**.[3]

We call the migration strategy introduced in Policy 1 the *standard migration case*. When applying it to an instance I, which is both structurally and state–related compliant with S', we actually propagate Δ_T to I and migrate I to S' preserving instance–specific change Δ_I on S'. Generally, migrating a process instance I for which instance change Δ_I overlaps with type change Δ_T is called the *advanced migration case*. As discussed above, adequate strategies for this case depend on the *degree of overlap* between process type and instance changes. It ranges from *equivalence* of the changes (cf. Def. 6) to minor overlapping between them. To give an idea of these advanced strategies we sketch the one for dealing with *equivalent* process type and process instance changes.

Policy 2 (Migrating Instances With Equivalent Bias).

Let S be a (correct) process type schema and Δ_T be a process type change which transforms S into another (correct) type schema S'. Let further $I = (S, \Delta_I, \ldots)$ be a process instance on S with instance execution schema $S_I := S + \Delta_I$. Finally let Δ_T and Δ_I be equivalent changes, i.e., $\Delta_T \equiv \Delta_I$. Then I can correctly migrate to S' with resulting bias $\Delta_I = \emptyset$ on S', i.e., $I = (S', \emptyset, \ldots)$.

[3] How to efficiently ensure compliance and how to automatically adapt instance markings when migrating them to the changed process type schema is extensively discussed in [14].

If an instance change Δ_I is equivalent with process type change Δ_T the advanced migration strategy is to re–link instance I to the new process type schema S' without applying any further changes or checks. In the sequel, instance change Δ_I is nullified due to the application of Δ_T, i.e., $\Delta_I(S') = \emptyset$.

An example is depicted in Fig. 1 where instance change Δ_{I_4} is equivalent with type change Δ_T (obviously S' and S_{I_4} are trace equivalent). Consequently, we can re–link I_4 to S' and we can set $\Delta_{I_4}(S') = \emptyset$. Due to lack of space, for dealing with further degrees of overlap we refer to [15].

3 Detecting the Degree of Overlap Between Concurrent Process Changes

Let S be a (correct) process schema and let $I = (S, \Delta_I, \ldots)$ be a (biased) process instance on S (with bias Δ_I). Let further Δ_T be a type change transforming S into another (correct) process schema S'. Then the challenging question arises whether Δ_T and Δ_I are disjoint or whether they are overlapping each other (cf. Def. 5). A naive solution would be to directly check Def. 5. Doing so would require materialization of resulting process schemes $S_{(\Delta_T,\Delta_I)} := (S + \Delta_T) + \Delta_I$ and $S_{(\Delta_I,\Delta_T)} := (S + \Delta_I) + \Delta_T$ and explicit verification of trace equivalence between $S_{(\Delta_T,\Delta_I)}$ and $S_{(\Delta_I,\Delta_T)}$. However, this approach is not applicable in practice for three reasons:

1. Δ_T cannot be always applied to $S_I := S + \Delta_I$ and vice versa Δ_I to $S' := S + \Delta_T$ (e.g., if Δ_T and Δ_I delete the same activities). Consequently, $S_{(\Delta_T,\Delta_I)}$ and $S_{(\Delta_I,\Delta_T)}$ respectively cannot be materialized.
2. Even if $S_{(\Delta_T,\Delta_I)}$ and $S_{(\Delta_I,\Delta_T)}$ can be materialized the verification of trace equivalence would require to determine all execution histories producible on $S_{(\Delta_T,\Delta_I)}$ and $S_{(\Delta_I,\Delta_T)}$. This, in turn, would demand reachability analyses for both schemes resulting in exponential complexity.
3. Assume that we can materialize both $S_{(\Delta_T,\Delta_I)}$ and $S_{(\Delta_I,\Delta_T)}$ and determine all possible execution histories. Nevertheless we would have to replay all these execution histories on the mutually other process schema. Due to the possibly large number of creatable execution histories and their large volume a severe performance penalty can be caused.

For these reasons we have to find better suited approaches to verify Def. 5 for Δ_T and Δ_I. The information we can use for this purpose comprises process schemes S, S_I, and S' and changes Δ_T and Δ_I. Intuitively, taking this information we come to the following three kinds of approaches (cf. Fig. 2): *(1) structural approaches* which directly compare process schemes S, S_I, and S', *(2) operational approaches* directly contrasting changes Δ_I and Δ_T (i.e., looking at the two sets of applied change operations), and *(3) hybrid approaches* (cf. Sect. 4) combining approaches (1) and (2). In the following we present these variants and systematically rate their particular stenghts and limitations.

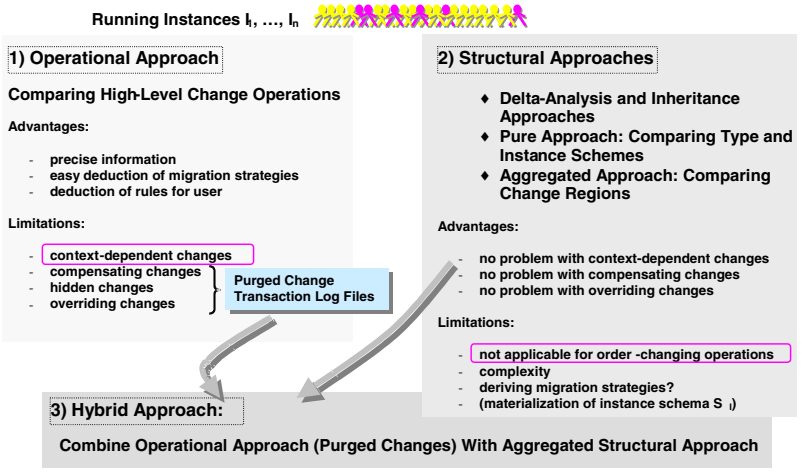

Fig. 2. Approach Overview to Detect Overlapping of Changes

3.1 Structural Approaches

The essence of all structural approaches is to compare process type schema $S' := S + \Delta_T$ with process instance schema $S_I := S + \Delta_I$ in order to gain information about the degree of overlap between Δ_T and Δ_I. A promising approach to analyze the difference between two process schemes, the so called *Delta Analysis*, has been presented in [16] and used by v.d. Aalst and Basten in [12]. In [12] Delta Analysis is based on four inheritance relations on process schemes. Roughly speaking a process schema S_1 is a subclass of process schema S_2 if it can do everything S_2 can do and more. With this, for example, v.d. Aalst and Basten determine the *Greatest Common Divisor (GCD)* for process schemes S_1 and S_2 which represents the common superclass of S_1 and S_2. Though this approach is very promising it cannot be adopted to the problem described in this paper since it shows the reverse line of attack as the following example illustrates:

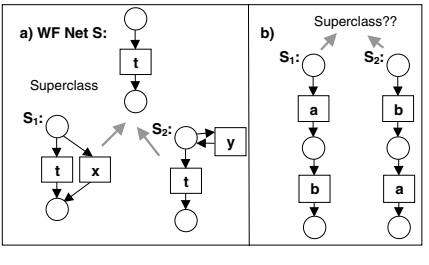

Fig. 3. Determining the Greatest Common Divisor (Examples)

Consider process schemes S_1 and S_2 (represented by WF Nets [2] – a Petri Net based formalism) as depicted in Fig. 3a). Applying the approach presented by v.d. Aalst and Basten [12] we start from process schemes S_1 and S_2 and determine the common superclass S. By contrast, in our approach we already have common divisor S and derive process type schema S' and process instance schema S_I by applying Δ_T and Δ_I respectively.

However, considering the Delta Analysis approach we can already recognize one common limitation of all structural approaches: they are not able to adequately deal with order–changing operations. One example is depicted in Fig. 3b) where we cannot find a process schema which represents a common behavior for schemes S_1 and S_2.

As a second possibility, consider the so called *pure structural approach* (cf. Fig. 2). Here we exploit the set–based representation of WSM Nets (cf. Sect. 2.1) and directly compare activity sets N' and N_I, edge sets $CtrlE'$ and $CtrlE_I$, $SyncE'$ and $SyncE_I$, $DataE'$ and $DataE_I$, $LoopE'$ and $LoopE_I$, and data element sets D' and D_I regarding the two process schemes

- $S' = (N', D', NT, CtrlE', SyncE', LoopE', DataE')$ and
- $S_I = (N_I, D_I, NT, CtrlE_I, SyncE_I, LoopE_I, DataE_I)$.

However, doing so is unnecessarily expensive. Actually we do not have to compare "whole" activity and edge sets since they have been derived starting with same original schema S, i.e., starting with the same activity and edge sets. In other words we already know a common divisor $S = (N, D, \ldots)$ for S' and S_I. Therefore we can reduce complexity by exploiting the common ancestry of S' and S_I what results in a third method which we call *aggregated structural approach* (cf. Fig. 2). More precisely, the aggregated structural approach works by comparing differences between process type schema S' and original schema S and between process instance schema S_I and original schema S. These differences can be easily determined by building the following difference sets:

- $N^{add}_{\Delta_T} := N' \setminus N$ and $N^{add}_{\Delta_I} := N_I \setminus N$
- $N^{del}_{\Delta_T} := N \setminus N'$ and $N^{del}_{\Delta_I} := N \setminus N_I$
- $CtrlE^{add}_{\Delta_T} := CtrlE' \setminus CtrlE$ and $CtrlE^{add}_{\Delta_I} := CtrlE_I \setminus CtrlE$
- and so on (cf. [17])

A first example is depicted in Fig. 4a). Both Δ_T and Δ_{I_1} serially insert activity X at the same position ("between B and C") into S_1 whereas Δ_{I_2} serially inserts another activity Y between A and B. Obviously, Δ_T and Δ_{I_1} overlap since they offend against claim (2) for disjoint changes (cf. Def. 5). Using the aggregated structural approach, we obtain $N^{add}_{\Delta_T} = N^{add}_{\Delta_{I_1}} = \{X\}$. This corresponds to the expected result, i.e., the multiple insertion of same activity X. Regarding instance I_2 on S_1, Δ_T and Δ_{I_2} are disjoint according to Def. 5. Application of the aggregated structural approach results in $N^{add}_{\Delta_T} \cap N^{add}_{\Delta_{I_2}} = \emptyset$, $N^{del}_{\Delta_T} \cap N^{del}_{\Delta_{I_2}} = \emptyset$, $CtrlE^{add}_{\Delta_T} \cap CtrlE^{add}_{\Delta_{I_2}} = \emptyset$, and $CtrlE^{del}_{\Delta_T} \cap CtrlE^{del}_{\Delta_{I_2}} = \emptyset$. Interpreting this result, we can state that Δ_T and Δ_{I_2} are disjoint.

These first two examples from Fig. 4a) show that the aggregated structural approach works fine for insert (and delete) operations. Reason is that we are able to precisely determine which activities have been inserted or deleted. In contrast, for move operations the aggregated structural approach (and consequently the pure structural approach) may be too imprecise[4]. Fig. 4b) shows a respective

[4] It is not sufficient to map a move operation onto respective delete and insert operations. Since activities are not really deleted or inserted structural approaches fail.

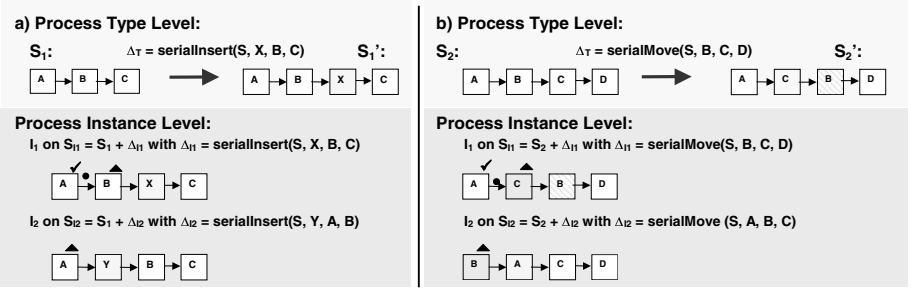

Fig. 4. Inserting and Moving Activities (Examples)

example: For all three changes on schema S_2, $N^{add}_{\Delta_T} = N^{add}_{\Delta_{I_1}} = N^{add}_{\Delta_{I_2}} = \emptyset$ and $N^{del}_{\Delta_T} = N^{del}_{\Delta_{I_1}} = N^{del}_{\Delta_{I_2}} = \emptyset$ holds (no activity has actually been inserted or deleted). Determining the sets of newly inserted and deleted control edges for Δ_T and Δ_{I_1} yields $CtrlE^{add}_{\Delta_T} = CtrlE^{add}_{\Delta_{I_1}} = \{(A, C), (C, B), (B, D)\}$ and $CtrlE^{del}_{\Delta_T} = CtrlE^{del}_{\Delta_{I_1}} = \{(A, B), (B, C), (C, D)\}$ respectively. From this result we can conclude that $\Delta_T \cap \Delta_{I_1} \neq \emptyset$. Comparing the respective edge sets for Δ_T and Δ_{I_2} again we obtain: $CtrlE^{add}_{\Delta_T} \cap CtrlE^{add}_{\Delta_{I_2}} \neq \emptyset$ and $CtrlE^{del}_{\Delta_T} \cap CtrlE^{del}_{\Delta_{I_2}} \neq \emptyset$. This indicates that $\Delta_T \cap \Delta_{I_2} \neq \emptyset$ holds. However, these results are too imprecise since in both cases we cannot exactly determine which activity has been actually moved. In case Δ_T and Δ_{I_1} are solely based on structural considerations, activity C as well as activity B could have been moved. When comparing Δ_T with Δ_{I_2} we can only conclude that these changes actually overlap but we are not able to make further statements. Both effects – not knowing which activities have been moved and imprecise statements about overlapping – are aggravated if change transactions comprise several move operations. In summary, taking this imprecise information it is not possible to derive adequate migration strategies.

3.2 Operational Approach

A solution to overcome the drawback of structural approaches in conjunction with order–changing operations – not knowing which activities have been actually moved – may be to directly compare applied changes Δ_T and Δ_I. Obviously, Δ_T and Δ_I contain precise information about applied changes in general and about actually moved activities in particular. However, this operational approach also shows limitations. As summarized in Fig. 2 change transaction logs may contain information about change operations which actually have no or only hidden effects on the underlying process schema. Reason is that the users who define changes (i.e., the process designer or the end user) do not always act in a goal–oriented way when modifying a process schema. In fact they may try out the best solution resulting in noisy information within the change logs:

1. The first group of changes without any effects on S' are *compensating changes*, i.e., changes mutually compensating their effects. A simple exam-

ple is depicted in Fig. 5 where activity Z is first inserted (between F and G) and afterwards deleted by the user. Therefore the respective operations `serialInsert(S,Z,F,G)` and `delete(S,Z)` 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 from deleting an activity which is then inserted again at another position. This actually has the effect of a move operation. An example is given in Fig. 5 where activity E is first deleted an then inserted again between Y and G. The effect behind is the same as of the respective move operation `serialMove(S, E, Y, G)`.
3. There are changes overriding effects of predeceding changes (note that a change transaction is an ordered series of single change operations). Again consider Fig. 5 where the effect of the hidden move operation `serialMove(S, E, Y, G)` (cf. 2.) is overwritten by move operation `serialMove(S, E, F, G)`, i.e., in S' activity E is finally placed between F and G.

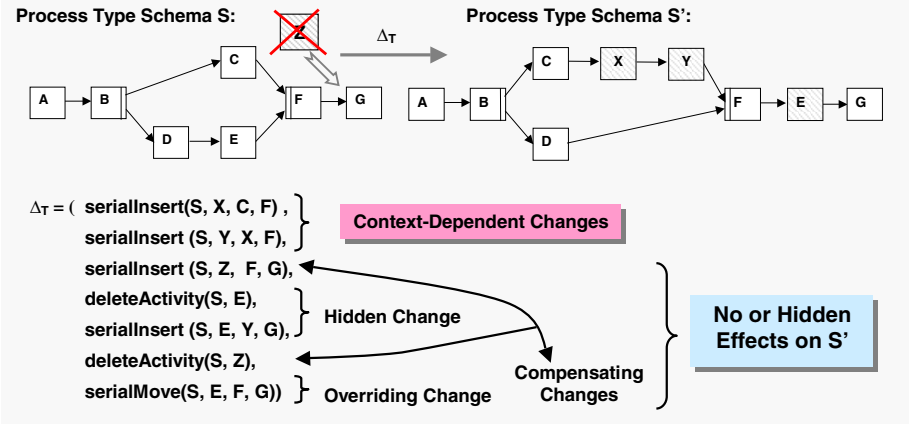

Fig. 5. Process Type Change Transaction (Example)

However, the presence of compensating, hidden, or overriding changes within a change transaction is a cumbersome but conquerable problem. Reason is that we can find methods to *purge* a change transaction from these kinds of changes (cf. Alg. 1). Doing so is essential in order to find a canonical and minimal view on change logs. This, in turn, is necessary to be able to determine which activities actually have been moved by a change.

A much more severe limitation of the operational approach is its disability to adequately deal with *context–dependent changes*, i.e., changes which are mutually based on each other. An example is depicted in Fig. 5: First, activity X is inserted serially between C and F. Based on this a second activity Y is inserted between X and F. Obviously, the second insertion uses the newly added activity of the first insertion as change context.

Why are such context–dependent process type and process instance changes critical when applying the operational approach? Fig. 6 illustrates the underlying problem. Obviously, Δ_T and Δ_I are equivalent since S' and S_I are trace equivalent. Unfortunately, this equivalence relation cannot be determined based on the depicted change transaction logs since Δ_T and Δ_I have inserted activities X, Y and Z in different orders. Therefore the operational approach sketched so far would only detect an overlapping (multiple insertion of same activities) but not be able to determine the degree of overlap, i.e., the total equivalence between Δ_T and Δ_I.

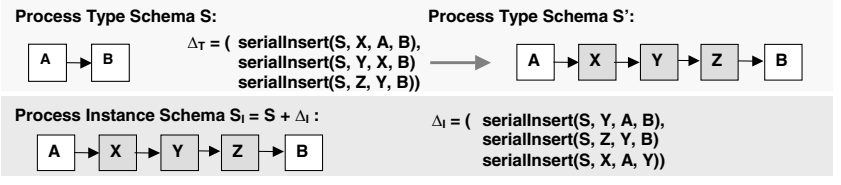

Fig. 6. Equivalent Process Type and Instance Changes (Example)

At this point an important conclusion is that structural approaches have no problems with context–dependent changes. Consider again Fig. 6. Applying the aggregated structural approach (cf. Sect. 3.1) we get $N^{add}_{\Delta_T} = N^{add}_{\Delta_I}$, $N^{del}_{\Delta_T} = N^{del}_{\Delta_I}$, $CtrlE^{add}_{\Delta_T} = CtrlE^{add}_{\Delta_I}$, and $CtrlE^{del}_{\Delta_T} = CtrlE^{del}_{\Delta_I}$ and therefore $\Delta_T \equiv \Delta_I$ holds.

In summary, at this point we have the following situation (cf. Fig. 2): Structural approaches are able to cope with context–dependent changes as well as with compensating, hidden and overriding changes. Reason is that structural approaches are based on the actual effects on a process schema. However, they are unable to adequately deal with order–changing operations. In contrast, when applying the operational approach we are able to precisely determine which activities have been moved but we are not able to handle context–dependent changes. Altogether, in the following section we combine both methods to a *hybrid approach* in order to exploit the particular strengths and to overcome the particular limitations.

4 The Hybrid Approach

The hybrid approach presented in the following combines elements of structural and operational approaches (cf. Sect. 3). How this approach works in general is presented in Sect. 4.1. How we can apply the hybrid approach to concurrent process type and instance changes is illustrated in Sect. 4.2.

4.1 Purging Change Logs and Consolidated Activity Sets

Let S be a (correct) process schema and let Δ be a change which transforms S into another correct process schema $S' := S + \Delta$. Informally, the hybrid ap-

proach works as follows: First, the activity sets actually inserted into and deleted from S – N_{Δ}^{add} and N_{Δ}^{del} (cf. Sect. 3.1) – are determined (*structural approach*). Taking this information the change log capturing Δ is *purged*. More precisely, this purging is accomplished by scanning the log of change $\Delta = (op_1, \ldots, op_n)$ in reverse order and by determining for each change operation op_i $(i = 1, \ldots, n)$ whether it actually has any effects on S. If so we incorporate op_i into a new – intially empty – change log Δ^{purged}. Finally, in order to reduce the number of necessary change log scans to one we use an auxiliary set A to memorize which activities have been already handled. In detail, the following considerations are made when determining Δ^{purged}:

- Assume that we find a log entry op_i for an operation inserting activity X between activities *src* and *dest* into S and that X is not yet present in A, i.e., op_i is the last change operation within Δ which manipulates X. If X has been already present in S ($X \notin N_{\Delta}^{add}$) a *hidden change* is found (cf. Sect. 3.2). Consequently, a respective log entry for an operation moving X between *src* and *dest* is created and written into Δ^{purged}.

- If log entry op_i denotes an operation deleting activity X from S and $X \notin A$ but X is still present in S' ($X \notin N_{\Delta}^{del}$) then we have found a *compensating change*. Therefore op_i (and the respective insert operation) are left outside Δ^{purged}.

- If log entry op_i denotes an operation moving activity X to a position between activities *src* and *dest* and op_i is the last operation within Δ which has effects regarding X ($X \notin A$) we have to distinguish between two cases: If X has been inserted before op_i ($X \in N_{\Delta}^{add}$) we write a new log entry in Δ^{purged} denoting an operation inserting X between *src* and *dest*. If X has been also present in S ($X \notin N_{\Delta}^{add}$) we write op_i unalteredly into Δ^{purged}.

In the following, the *consolidated activity sets* $(N_{\Delta}^{add}, N_{\Delta}^{del}, N_{\Delta}^{move})$ (cf. Def. 7) will serve as the basis for determining the degree of overlap between changes. Note that N_{Δ}^{add} and N_{Δ}^{del} can be determined using the aggregated structural approach (cf. Sect. 3.1) but we have to use purged change logs (operational approach) in order to obtain N_{Δ}^{move}.

A formalization of the method described above is given in Alg. 1. For the sake of simplicity we restrict this description to serial insert operations. However adopting parallel and branch insertions runs analogously.

Definition 7 (Purged Change Transaction; Consolidated Activity Sets). *Let S = (N, D, ...) be a (correct) process schema. Let further Δ be a change which transforms S into another (correct) process schema S' = $(N', S', \ldots)$. Then the purged representation of Δ, Δ^{purged} and the consolidated activity sets $(N_{\Delta}^{add}, N_{\Delta}^{del}, N_{\Delta}^{move})$ can be determined by applying Algorithm 1.*

Algorithm 1. PurgeConsolidate(S, N, N', $\Delta = (op_1, \ldots, op_n)$) $\longrightarrow$ (Δ^{purged}, ($N_\Delta^{add}, N_\Delta^{del}, N_\Delta^{move}$))

```
A:=∅; Δ^purged = ∅;
N_Δ^add := N' \ N; N_Δ^del := N \ N';
for i = n to 1 do {
  if (op_i = 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)[5]){ //X moved to another position?
      Δ^purged.addFirst(serialMove(S, X, src, dest))//adds entry at beginning of Δ^purged;
      N_Δ^move := N_Δ^move ∪ {X};}} else {
     Δ^purged.addFirst(serialInsert(S, X, src, dest));}} continue};
 if (op_i = serialMove(S, X, src, dest)) {
   if (X ∉ A) {
    A := A ∪ {X};
    if (X ∈ N_Δ^add) {
     Δ^purged.addFirst(serialInsert(S, X, src, dest)); } else {
     if (src ≠ c_pred(S, X) ∧ dest ≠ c_succ(S, X)) {
      Δ^purged.addFirst(serialMove(S, X, src, dest));
      N_Δ^move := N_Δ^move ∪ {X};}} continue;}
 if (op_i = delete(S, X)) {
   if (X ∉ A) {
    A := A ∪ {X};
    if(X ∈ N_Δ^del){
      Δ^purged.addFirst(delete(S, X));}}}
  Δ^purged.addFirst(op_i);
}
return (Δ^purged, (N_Δ^add, N_Δ^del, N_Δ^move));
```

4.2 Application to Concurrent Process Type and Instance Changes

A practically relevant application of the hybrid approach introduced in Sect. 4.1 is to determine the degree of overlap between concurrent process type and process instance changes. We illustrate this by the following example:

Fig. 7 shows the mode of operation of Alg. 1 applied to the log of change Δ_T in Fig. 5. Initially, Alg. 1 determines the sets of newly inserted and deleted activities regarding schema S, i.e., $N_{\Delta_T}^{add} = \{X, Y\}$ and $N_{\Delta_T}^{del} = \emptyset$. Based on this information change log Δ_T is traversed once (in reverse direction) and purged from noisy operations op_6, op_5, op_4, op_3. Algorithm 1 finishes with purged change transaction Δ_T^{purged} = (serialInsert(S, X, C, G), serialInsert(S, Y, X, G), serialMove(S, E, F, G)) (cf. Fig. 7). Based on this purged change log the set of activities actually moved by Δ_T can be determined as $N_{\Delta_T}^{move} = \{E\}$. Together with the set of newly inserted and deleted activities we obtain consolidated activity sets $(N_{\Delta_T}^{add}, N_{\Delta_T}^{del}, N_{\Delta_T}^{move}) = (\{X, Y\}, \emptyset, \{E\})$.

Purging change logs from noisy information has several advantages: First, the purged form of a change log can be used as the canonical representation of this change, i.e., if we have to compare changes (what we actually have to do when determining the degree of overlap between them) we can use the purged form as an adequate basis. Furthermore, purged change logs are also sufficient to determine the difference between changes. This, for example, is necessary if we want to calculate the instance bias after migration to the changed schema (if

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

Change Log (in reverse order):	Initialization:	Purged Change Log
	$A = \emptyset$;	
	$N_{\Delta T}^{add} = \{X, Y\}$;	
	$N_{\Delta T}^{del} = \emptyset$;	
$\Delta_{T=}$ (		$\Delta^{purged}{}_{T=}$ (
op_7 = serialMove(S, E, F, G),	$E \notin A \Rightarrow A = \{E\}$;	
	$E \notin N_{\Delta T}^{add} \wedge$ new pos. $\Rightarrow$	op_3 = serialMove(S, E, F, G),
op_6 = deleteActivity(S, Z),	$Z \notin A \Rightarrow A = \{E, Z\}$;	
	$Z \notin N_{\Delta T}^{del}$;	
op_5 = serialInsert (S, E, Y, G),	$E \in A$;	
op_4 = deleteActivity(S, E),	$E \in A$;	
op_3 = serialInsert (S, Z, F, G),	$Z \in A$;	
op_2 = serialInsert (S, Y, X, F),	$Y \notin A \Rightarrow A = \{E, Z, Y\}$;	
	$Y \in N_{\Delta T}^{add} \Rightarrow$	op_2 = serialInsert(S, Y, X, F),
op_1 = serialInsert(S, X, C, F))	$X \notin A \Rightarrow A = \{E, Z, Y, X\}$;	
	$X \in N_{\Delta T}^{add} \Rightarrow$	op_1 = serialInsert(S, X, C, F),

Fig. 7. Purging A Change Log (Example)

bias and respective type change are not disjoint or equivalent). A more detailed treatment of these issues can be found in [17].

5 Discussion

In the workflow literature, there are many approaches either dealing with process type changes ("schema evolution") or single process instance changes [11,7,2,3, 4,5]. Thereby, main focus has been put on providing appropriate correctness criteria for deciding about compliance of unbiased instances. Although there are some approaches [7,11] that provide common support for process type and instance changes there is no **interplay** between them. $WASA_2$ [7], for example, realizes changes of single process instances by deriving a new schema version with exactly one running instance. Consequently, individually modified instances are excluded from further process type changes.

Commutativity (cf. Sect. 2.2) is an important property in the context of concurrent changes in cooperative applications. In [18], operations commute if the state changes on an object as well as the values returned by the operations are independent of the order in which they are executed. Wäsch and Klas claim that concurrent changes on complex objects can be correctly carried out if they are commutative followed by a *history merge* of the respective changes [19]. In this paper, we use commutativity to define disjointness of changes. However, we do not restrict correctness of concurrent changes on commutativity but we provide advanced solutions for non commutative and therefore overlapping changes.

As discussed in Section 3.1 an interesting structural approach to compare process schemes is the Delta Analysis based on inheritance relations [12]. The used inheritance relations as well as our definition of disjointness and overlapping are based on equivalence notions between process schemes. V.d. Aalst and Basten use *branching bisimilarity* as equivalence relation [12,2,20,21]. There are

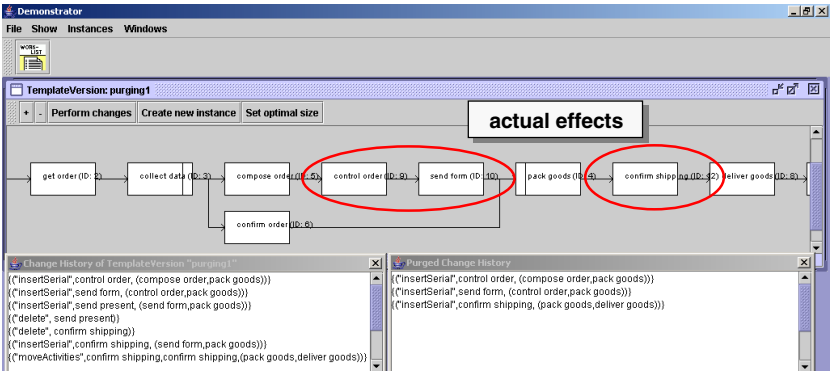

Fig. 8. Purging A Change Log (Prototype)

several other notions of equivalence between process schemes [13]. In [22], for example, v. Glabbeek and Goltz provide a very nice classification of semantic equivalences based on the basic notions of *bisimulation* and *trace equivalence*. Another approach to provide semantic equivalence of process schemes is given [23]. This work offers interesting methods to maintain the semantical meaning of a process schema before and after the change by applying semantics–preserving transformations.

6 Summary

In this paper, we have established a formal framework for dealing with concurrent process changes. An important application of this results is the propagation of process type changes to biased process instances. Based on the particular degree of overlap between process type and instance change we have to choose different migration strategies. To be able to decide to which degree process changes overlap we have presented an advanced approach which comprises structural aspects as well as operational solutions like purging change transactions.

We have implemented the presented concepts within a proof–of–concept prototype. Within this prototype migration of unbiased process instances as well as migration of biased instances with disjoint bias can be correctly and efficiently carried out. Furthermore, it can be precisely determined to which degree process type and instance changes overlap. Alg. 1 for purging change logs has been implemented. Fig. 8 depicts the example change log from Fig. 7 and the resulting purged change log.

References

1. v.d. Aalst, W., van Hee, K.: Workflow Management. MIT Press (2002)
2. v.d. Aalst, W., Basten, T.: Inheritance of workflows: An approach to tackling problems related to change. Theoret. Comp. Science **270** (2002) 125–203

3. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data and Knowledge Engineering **24** (1998) 211–238
4. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proc. COOCS'95, Milpitas, CA (1995) 10–21
5. Sadiq, S., Marjanovic, O., Orlowska, M.: Managing change and time in dynamic workflow processes. IJCIS **9** (2000) 93–116
6. Reichert, M., Dadam, P.: ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. JIIS **10** (1998) 93–129
7. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: Proc. HICSS-34. (2001)
8. Rinderle, S., Reichert, M., Dadam, P.: On dealing with structural conflicts between process type and instance changes. In: Proc. BPM'04. LNCS, Potsdam (2004)
9. Reichert, M., Rinderle, S., Dadam, P.: On the common support of workflow type and instance changes under correctness constraints. In: Proc. CoopIS '03, Catania, Italy (2003) 407–425
10. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems – a survey. Data and Knowledge Engineering, Special Issue on Advances in Business Process Management **50** (2004) 9–34
11. Kochut, K., Arnold, J., Sheth, A., Miller, J., Kraemer, E., Arpinar, B., Cardoso, J.: IntelliGEN: A distributed workflow system for discovering protein-protein interactions. Distributed and Parallel Databases **13** (2003) 43–72
12. v.d. Aalst, W., Basten, T.: Identifying commonalities and differences in object life cycles using behavorial inheritance. In: Proc. ICATPN '01, Newcastle, UK (2001) 32 – 52
13. Kiepuszewski, B.: Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows. PhD thesis, Queensland University of Technology, Brisbane (2002) (available via *http://www.tm.tue.nl/it/research/patterns)*.
14. Rinderle, S., Reichert, M., Dadam, P.: Flexible support of team processes by adaptive workflow systems. Distributed and Parallel Databases **16** (2004) 91–116
15. Rinderle, S., Reichert, M., Dadam, P.: On dealing with semantically conflicting business process changes. Technical Report UIB-2003-04, University of Ulm (2003)
16. Guth, V., Oberweis, A.: Delta analysis of petri net based models for business processes. In: Proc. Applied Informatics. (1997) 23–32
17. Rinderle, S.: Schema Evolution In Process Management Systems. PhD thesis, University of Ulm (2004) (to appear).
18. Badrinath, B., Ramamritham, K.: Semantics-based concurrency control: Beyond commutativity. ACM Transactions on Database Systems **17** (1992) 163–199
19. Wäsch, J., Klas, W.: History merging as a mechanism for concurrency control in cooperative environments. In: Proc. RIDE'96, New Orleans (1996) 76–85
20. Basten, T.: Branching bisimilarity is an equivalence indeed! Information Processing Letters **58** (1996) 141–147
21. Verbeek, E.: Verification of WF–Nets. PhD thesis, Technical University of Eindhoven (2004)
22. v. Glabbeek, R., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Informatica **37** (2001) 229–327
23. Frank, H., Eder, J.: Equivalence transformations on statecharts. In: Proc. SEKE'00, Chicago (2000) 150–158