On Dealing with Structural Conflicts between Process Type and Instance Changes*

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 management systems must be able to support changes of single process instances as well as 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 manner. However, especially for long-running processes, it must be possible to handle the interplay between process type and instance changes as well, but without running into trouble at runtime. This paper presents an extended criterion for correctly *propagating* process type changes to both, instances which are still running according to their original schema and instances which have been individually modified. In this context, we discuss and categorize structural conflicts potentially occuring between concurrent process changes. We show that our considerations are applicable to different process meta models and present tests for quickly detecting such structural conflicts.

1 Introduction

Adaptivity in process management systems (PMS) is key to flexible enterprise information systems. Basically, changes in process-oriented applications can take place at two levels – the process type or the process instance level.

A *process type* represents a particular business process (e.g., handling of a purchase order or treatment of a patient). It is described by a *process schema* which defines a collection of activities and sets out the control as well as data flow between them. Based on such a process schema, new *process instances* can be created and executed according to the defined process logic. *Process type changes* become necessary, for example, to adapt the process-oriented information system to optimized business processes or to new laws. They are handled by (structurally) modifying the respective process schema, which leads to a new schema version of the respective type. Particularly for long-running processes (e.g., handling of leasing contracts or medical treatments) it is desired to *propagate* a process type change to already running process instances as well. Process instances for which this is possible are *compliant* with the new schema and can therefore be *migrated* to it. As opposed to process type changes, *changes of*

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

J. Desel, B. Pernici, and M. Weske (Eds.): BPM 2004, LNCS 3080, pp. 274–289, 2004.

single process instances (e.g., to insert or skip an activity) are often carried out in an ad-hoc manner in order to deal with an exceptional situation. Adapting a single process instance during runtime, in turn, results in an instance-specific schema (also called *instance execution schema* in the following), which differs from the original schema this instance was created from. In the following, we denote such individually modified process instances as *biased*.

In the literature [1,2,3,4,5,6,7,8,9,10] process type and instance changes have been an important research topic for several years. However, there are only few adaptive PMS which support both kinds of changes in one system [10,11]. All of them have in common that once an instance has been individually modified (i.e., it possesses an instance-specific process schema), it cannot longer benefit from process type changes; i.e., changes of the schema they were originally created from. In WASA_2 [10], for example, an instance change is carried out by deriving a new schema version to which the instance is migrated. In the sequence, this instance is excluded from further adaptations of its original schema version at the process type level. However, doing so is not sufficient in many cases, especially in connection with long-running processes. Therefore, it must be possible to propagate process schema changes at the type level to such biased instances as well.

This paper focuses on the interplay of process type and instance changes under appropriate correctness constraints. Such constraints are necessary since an uncontrolled propagation of process type changes to biased instances may raise severe errors at runtime. A first contribution is to present a **correctness criterion** for propagating process type changes to both unbiased and biased process instances. This criterion is independent of the used process meta model. Furthermore, it excludes *state-related*, *structural*, and *semantical* conflicts between concurrent process type and instance changes. A second contribution deals with **structural correctness** of concurrent process type and instance changes. A simple example for such a structural conflict is depicted in Fig. 1. Here, propagating the process type change (cf. Fig. 1a) to biased instance I in an uncontrolled manner would lead to a deadlock causing cycle in the resulting process instance schema (cf. Fig. 1b). A naive solution to overcome this undesired behavior would be to simulate the process type change on each instance-specific schema (i.e., to materialize the resulting instance schema) and then to verify control and data flow correctness. Doing so may become very critical regarding performance, especially in conjunction with a large number of biased instances. An alternative solution is to check for process schema and instance changes whether they are in conflict or not. Our ambition is to exclude structural conflicts for as much (biased) instances as possible by the use of simple and easy to check tests. In any case expensive control and data flow analyses shall be avoided to a large degree. This paper presents appropriate tests to detect control flow as well as data flow conflicts between process schema and instance changes.

In Section 2, a general correctness criterion handling both process type and instance changes is introduced. Section 3 provides necessary background information for our concrete solution approach. In Section 4 we present structural conflict tests which are illustrated by an example in Section 5. Section 6 discusses related work and Section 7 closes with a summary of the presented results.

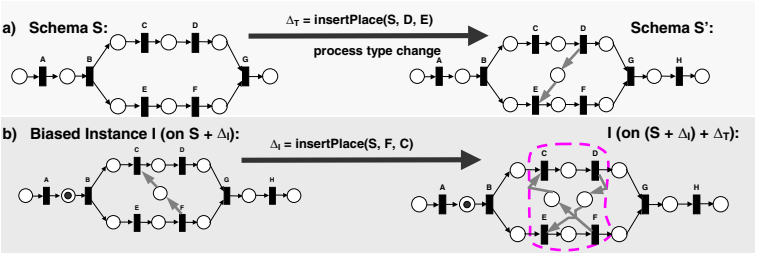

Fig. 1. Structural Conflict in Petri Nets (Deadlock)

2 A General Correctness Criterion for Process Type and Process Instance Changes

In this section we present a criterion for correctly propagating process type changes to both unbiased and biased process instances (cf. Axiom 1).

Axiom 1 (Propagating Type Changes To Biased Instances) *Let T be a process type with actual schema version S. Assume that a new (correct) schema version S' is derived from S by applying type change Δ_T to it. Then: Δ_T may be propagated to instance I (with type T and current instance execution schema $S_I := S + \Delta_I$) :⇔*

1. **Structural Correctness:** *$S_I^* := (S + \Delta_I) + \Delta_T$ is a correct schema according to the structural correctness constraints set out by the used process meta model; i.e., Δ_T can be correctly applied to $S_I = (S + \Delta_I)$.*
2. **State-Related Correctness:** *I is compliant with S_I^* (cf. 2); i.e., the execution history $\mathcal{H}_{red}$ of I can be produced on S_I^* as well.*[1]
3. **Semantical Correctness:** *Δ_T and Δ_I are semantically conflict-free.*

Axiom 1 is valid for unbiased as well as for biased process instances. More precisely, it handles unbiased instances as a special case; i.e., for an unbiased instance we obtain $S_I^* = S'$ whereby S' is correct according to the assumption of Axiom 1 and Δ_T and Δ_I are semantically conflict-free. Consequently, only state-related correctness has to be checked what exactly corresponds to the well-known compliance criterion [3,9,12]; i.e., an unbiased instance I is compliant with a changed schema S' if its previous execution trace on S is also a possible execution trace on S' (cf. Requirement 2 in Axiom 1).

Regarding Requirement 1 of Axiom 1 we first have to ensure that Δ_T is actually applicable to the instance-specific execution schema $S_I := S + \Delta_I$.

[1] An execution history $\mathcal{H}$ of instance I on S usually logs all start and end events generated during the execution of I. The reduced execution history $\mathcal{H}_{red}$ is determined by logically discarding all entries from $\mathcal{H}$ which have been produced by another than the actual loop iteration for each loop contained in S. The reduction from $\mathcal{H}$ to $\mathcal{H}_{red}$ is necessary in order to avoid restrictiveness in conjuntion with loops [12]).

Therefore, generally, all "pre-conditions" of Δ_T on S_I must be fulfilled. How these change pre-conditions exactly look like depends on the respective change operations. However, a common claim for all kinds of change operations is that all schema objects manipulated by Δ_T should be present in S_I. An example for a process type change Δ_T not applicable to S_I is depicted in Fig. 2: Δ_T deletes activity D which has already been deleted at the instance level. Intuitively, Δ_T cannot be applied to the instance-specific schema S_I.

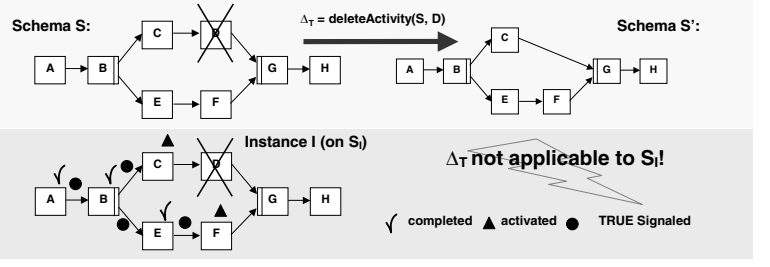

Fig. 2. Process Type Change Not Applicable to Instance-Specific Schema

If, in contrast, Δ_T is applicable to S_I, target schema $S_I^* := (S + \Delta_I) + \Delta_T$ can be produced. However, the resulting instance-specific schema S_I^* may still contain control and data flow errors (like deadlock-causing cycles or missing input data). We therefore must analyze S_I^* with respect to its structural correctness properties (e.g., absence of cycles except loop bodies) by the corresponding "post-conditions" of Δ_T on S_I. The core problem addressed in this paper is how to (efficiently) ensure that S_I^* does not contain any control or data flow errors, i.e., to (efficiently) ensure that there are no structural conflicts between process schema and instance changes (cf. Requirement 2 of Axiom 1). Obviously, an appropriate approach for this problem has to work for a large number of biased process instances as well. As mentioned in the introduction, a naive solution would be to first materialize schema $S_I^* := (S + \Delta_I) + \Delta_T$ for <u>each</u> (biased) instance I and then to apply respective correctness checks on S_I^*. However, this may result in a serious performance problem caused by the expensive materialization of S_I^* on the one hand and the subsequent complex control and data flow correctness checks on S_I^* on the other hand. Again note that these two steps would have to be applied to each biased instance to be migrated.

Therefore, in this paper, we show how expensive correctness tests (based on materialized schemes S_I^* for each biased instance I) can be avoided. The key idea behind is to detect potential control and data flow errors in $S_I^* := S + (\Delta_I) + \Delta_T$ solely based on the applied changes Δ_T and Δ_I, and the original schema S. More precisely, we elaborate quickly checkable conflict tests by exploiting the semantics of the applied changes Δ_T and Δ_I. Respective conflict tests either yield that there would be definitely no control or data flow error in schema S_I^* or

they indicate that a possible structural conflict between Δ_T and Δ_I (potentially leading to such an error) may occur.

How to check state-related correctness (cf. Requirement 2 of Axiom 1) has been described in another paper of our group [12]. We have developed a set of compliance rules which can be used for checking state-related compliance of unbiased as well as of biased process instances with a modified schema. More precisely, these compliance rules define efficiently checkable conditions on activity node markings for each applicable change operation.

A semantical conflict (cf. Requirement 3 of Axiom 1) may occur, for example, if Δ_T inserts activity *"give drug A"* at process type level and Δ_I inserts activity *"give drug B"* at instance level and there is a medical incompatibility between drugs *A* and *B*. Consequently, executing instance I on target schema S_I^* would lead to a medication with incompatible drugs. The detection of this semantical conflict requires additional information about the changes. Due to lack of space we obstain from further details about semantical issues in this paper.

3 Fundamentals

In order to be able to precisely define structural conflicts tests for concurrent process type and instance changes we need a formal process meta model. In this paper, we exemplarily use WSM-Nets (as for example applied in ADEPT [13]) and the change operations based on them for this purpose. However, similar conflict tests can be developed for other process meta models as well.

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 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*

Thus, a process schema is represented by attributed serial-parallel graphs with additional sync links. 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 for which a mandatory input parameter is linked to a data element $d \in D$ it has to be ensured that d will be always written at runtime independently of which execution path will be chosen.
7. Parallel write accesses on data elements (and consequently lost updates) have to be avoided.

Taking a correct WSM Net S new instances can be created and started. Logically, each process instance I is associated with an instance-specific schema $S_I := S + \Delta_I$ (for unbiased instances $S_I = S$ holds). The control state of I is captured by a marking function M^{S_I} =(NS^{S_I}, ES^{S_I}). It assigns to each activity n its current status $NS(n)$ and to each control and loop edge its marking $ES(e)$. These markings are determined according to well defined marking rules [8], 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* $(S, \Delta_I, M^{S_I}, Val^{S_I}, \mathcal{H})$ *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. Schema* $S_I := S + \Delta_I$*, which results from the application of* Δ_I *to S, is called the instance execution schema of I.*
- M^{S_I} *=* (NS^{S_I}, ES^{S_I}) *describes node and edge markings of I:*
  NS^{S_I}*:* $N \mapsto$ {`NotActivated, Activated, Running, Completed, Skipped`}
  ES^{S_I}*:* $(CtrlE \cup SyncE \cup LoopE) \mapsto$
  {`NotSignaled, TrueSignaled, FalseSignaled`}
- Val^{S_I} *is a function on D. It reflects for each data element* $d \in D$ *either its current value or the value* `UNDEFINED` *(if d has not been written yet).*
- $\mathcal{H} = < e_0, \ldots, e_k >$ *is the execution history of I.* $e_0, \ldots, e_k$ *denote the start and end events of activity executions. For each started activity X the values of data elements read by X and for each completed activity Y the values of data elements written by Y are logged.*

Activities marked as `Activated` are ready to fire and can then be worked on, i.e., their status changes to `Running`. As an example take instance I in Fig. 3: Activity A is completed whereas activity B is activated. Activities with marking `Skipped` cannot longer be selected for execution.

Table 1 presents a selection of *high-level operations* which can be used to define or change WSM-Nets. We distinguish between *basic* and *high-level change*

operations. Examples for basic change operations are insertion/deletion of activity nodes or control edges. Such basic change operations can be applied to a process schema; afterwards structural correctness of the resulting schema has to be checked. In contrast, high-level change operations include formal pre- and post-conditions and automatically perform the necessary schema transformations such that schema correctness can be ensured. An example is process type change Δ_T in Fig. 3: Δ_T serially inserts activity X into S by automatically embedding X between activities D and G. (For more complex examples see [8]).

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

Change Operation Δ Applied to Schema S	**Effects on Schema S**
	Additive Change Operations
serialInsert(S, X, A, B)	serial insertion of activity X between activities A and B
parallelInsert(S, X, CtrlBlock)	insertion of activity X parallel to control block CtrlBlock
insertSyncEdge(S, src, dest)	insertion of sync edge linking two parallel nodes src and dest
	Subtractive Change Operations
deleteActivity(S, X)	deletes activity X from schema S
deleteSyncEdge(S, edge)	deletes edge $\in$ SyncE from schema S
	Order-Changing Operations
serialMoveActivity(S, X, A, B)	moves activity X from current position to position between activities A and B
parallelMoveActivity(S, X, CtrlBlock)	moves activity X to position parallel to control block CtrlBlock
	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 $\in$ CtrlE $\cup$ 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) to S (mode $\in$ {read, write})
deleteDataEdge(S, dL))	deletes data edge dL from S

4 Structural Conflict Tests

In this section, we provide simple but effective tests for detecting potential conflicts between concurrently applied control and/or data flow changes. In particular, respective tests can be used in connection with the common support of process type and process instance changes.

4.1 On Detecting Control Flow Conflicts

A serious problem which may arise from the uncontrolled propagation of a process type change Δ_T to a biased instance (on instance-specific schema $S_I := S + \Delta_I$) is the occurence of deadlock-causing cycles (for an example

see Fig. 1). As mentioned before, a naive solution would be to first materialize the target schema $S_I^* := (S + \Delta_I) + \Delta_T$ and then to carry out respective cycle checks on S_I^*. Since these materialization and validation steps would have to be applied for each biased instance I, this approach would cause severe performance problems. Thus, our ambition is to perform an appropriate deadlock test based on information given by the process type and instance changes themselves and the original process schema S. A first version of a deadlock tests satisfying these claims is given in Proposition 1 [13]:

Proposition 1 (Basic Deadlock Prevention). *Let S be a WSM-Net and I be a biased instance with starting schema S and execution schema $S_I := S + \Delta_I = (N_I, D_I, NT_I, CtrlE_I, SyncE_I, \ldots)$. Assume that type change Δ_T transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...). Then: $S_I^* = (S + \Delta_I) + \Delta_T$ does not contain deadlock-causing cycles if the following condition holds:*

$\forall\ (s_1, d_1) \in \mathcal{AS}(S, \Delta_T), \forall\ (s_2, d_2)\ \mathcal{AS}(S, \Delta_I)$:

$$d_1 \notin (pred^*(S, s_2) \cup \{s_2\}) \vee d_2 \notin (pred^*(S, s_1) \cup \{s_1\})\ (\Psi)$$

whereas

- $\mathcal{AS}(S, \Delta_T) := SyncE' \setminus SyncE$
- $\mathcal{AS}(S, \Delta_I) := SyncE_I \setminus SyncE$
- *$pred^*(S, n)$ denotes all direct and indirect predecessors of activity n when considering both control and sync edges of S.*

By simply applying condition Ψ from Proposition 1 we can exclude deadlocks when propagating a type change to a biased instance. Note that condition Ψ is based on the original process schema S. Consequently, an easy conflict test can be derived which avoids the materialization of any other schema (S_I or S_I^*).

In general, the quality of a conflict test can be measured according to how efficiently it can be applied to concurrent process type and instance changes. Another important quality factor is the number of "uncritical" instances I for which conflicts between process type and instance changes can be definitely excluded. The deadlock test derived from Proposition 1 is a "good" test with respect to efficiency. However, it still scores lower regarding the second quality factor. Reason is that for particular instance changes Δ_I this test indicates conflicts with type change Δ_T although the target schema $S_I^* := (S + \Delta_I) + \Delta_T$ will not contain any deadlock causing cycle. An example is depicted in Fig. 3: Instance change Δ_I inserts a sync edge between activities C and F already contained in S whereas type change Δ_T inserts a sync edge between also newly inserted activites X and Y. From the applied changes we derive $\mathcal{AS}(\mathrm{S}, \Delta_T) = \{(\mathrm{X}, \mathrm{Y})\}$ and $\mathcal{AS}(\mathrm{S}, \Delta_I) = \{(\mathrm{C}, \mathrm{F})\}$ (cf. Proposition 1). The expression yielding from applying condition Ψ from Proposition 1 to these sets cannot be evaluated due to the absence of activites X and Y in S. Consequently, the respective conflict test is unable to exclude the occurence of a deadlock-causing cycle in S although in fact there is none.

At first glance it seems that we must materialize and validate target schema S_I^* in order to overcome this problem. This approach, however, offends against the efficiency quality factor. Fortunately, there is another solution avoiding materialization of S_I^* and excluding deadlock conflicts for "uncritical" instances.

Consider again the example given in Fig. 3: Here we cannot evaluate condition Ψ based on sync edge (X, Y) since its source and destination activities have been newly inserted by Δ_T as well. However, the insertion of sync edge (X, Y) does not only set out the direct order relation "X before Y" but also, for example, the transitive order relation "D before E". Since D and E are present in S we are able to verify condition Ψ for a respective sync edge (D, E). Based on this consideration we try to virtually re-link the actual sync edge (X, Y) to the virtual sync edge (D, E). The challenge is to determine the virtual sync edge(s) based on which condition Ψ can be evaluated on S. Then solely based on S we can determine whether S_I^* will contain a deadlock-causing cycle or not. From Δ_T we know which activities have been inserted and into which context they have been embedded (*insertion context*). For serial insertion of activities, for example, the insertion context includes the direct predecessor and successor of the newly inserted activity. For the newly inserted activity X in Fig. 3, for example, insertion context (D, G) includes the direct predecessor D of X in S' and for the newly inserted activity Y its insertion context (B, E) includes the direct successor E of Y in S'. Altogether, this is the information we need for determining the virtual sync edges between activities present in S. In our example (cf. Fig. 3) we get the virtual sync egde (D, E) instead of (X, Y).

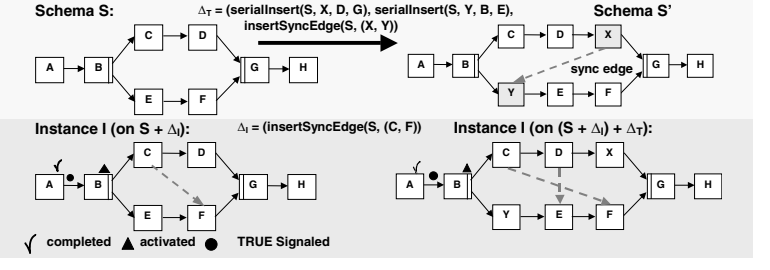

Fig. 3. Insertion of Sync Edges on Process Type and Instance Level

Thus, the idea behind is to first transfer the order relations set out by the newly inserted sync edges to starting schema S by applying "virtual" graph reduction rules and then to apply condition Ψ of Proposition 1 to the reduced graph. The respective graph reduction approach applicable in connection with the composed insertion of activities and sync edges is given in Algorithm 1:

Algorithm 1 (Graph Reduction Rules (Deadlock Prevention)) *Let S = (N, D, NT, CtrlE, SyncE, ...) be a WSM-Net and Δ be a type change which transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...). Let further*

- $\mathcal{AS}(S, \Delta) := SyncE' \setminus SyncE$ *and*
- $\mathcal{AA}(S, \Delta) := \{(X, (src, dest)) \mid X \in N', src, dest \in N, X$ *serially inserted between src and dest by* $\Delta\}$.

```
GraphRed(N, AS(S, Δ), AA(S, Δ)) ⟶ (AS_red(S, Δ))

AS_red(S, Δ):=∅
forall (src, dest) ∈ AS(S, Δ) do
   while src ∉ N do
      find (src, (pSrc, sSrc)) ∈ AA(S, Δ);
      src := pSrc;
   done
   while dest ∉ N do
      find (dest, (pDest, sDest)) ∈ AA(S, Δ);
      dest := sDest;
   done
   AS_red(S, Δ) := AS_red(S, Δ) ∪ {(src, dest)}
done
```

Algorithm 1 works by replacing the source (destination) nodes of the newly inserted sync edges by their direct predecessors (successors) if these nodes have not been present in the original schema S. If several activities are inserted in a row Algorithm 1 iteratively replaces them by their direct predecessors/successors until we find an adequate predecessor/successor also present in S. In the following Prop. 2, condition Ψ of Prop. 1 is applied based on the graph reduction of Algorithm 1. A deadlock test derived from this proposition fulfills both desired quality factors: It is efficiently applicable based on original schema S and it does not indicate deadlocks for target schema S_I^* if S_I^* is actually deadlock-free.

Proposition 2 (Deadlock Prevention (2)). *Let the assumption be as in Proposition 1. Let further $\mathcal{AS}_{red}(S, \Delta_T)$ and $\mathcal{AS}_{red}(S, \Delta_I)$ be the sync edge reductions after applying Algorithm 1.*
Then: $S_I^* = (S + \Delta_I) + \Delta_S$ does not contain deadlock-causing cycles iff the following condition holds:

$$\forall\ (s_1, d_1) \in \mathcal{AS}_{red}(S, \Delta_T),\ \forall\ (s_2, d_2) \in \mathcal{AS}_{red}(S, \Delta_I):$$
$$d_1 \notin (pred^*(S, s_2) \cup \{s_2\}) \vee d_2 \notin (pred^*(S, s_1) \cup \{s_1\})\ (\Psi)$$

As already mentioned, the reduction rules of Algorithm 1 are necessary in order to transfer the order relations set out by the newly inserted sync edges to the original schema S. As decribed in Proposition 2, we apply Algorithm 1 to the sync edges and activities newly inserted by Δ_T and Δ_I. Based on the resulting sets $\mathcal{AS}_{red}(\mathrm{S}, \Delta_T)$ and $\mathcal{AS}_{red}(\mathrm{S}, \Delta_I)$ condition Ψ from Proposition 1 can be applied to S. Doing so saves us from expensive checks on S_I^*.

In general, there are further constraints set out by the particular process meta model. In block-structured meta models like BPEL4WS [14] or ADEPT [8], for example, it is forbidden that sync links cross the boundaries of loop blocks. However, uncontrolled propagation complex process type changes to biased instances may result in such undesired sync links. Therefore we have made formal propositions for respective cases as well from which quick conflict tests can be derived. Due to lack of space we obstain from further details here.

4.2 On Detecting Data Flow Conflicts

An uncontrolled propagation of process type change Δ_T to biased instance I on $S_I = S + \Delta_I$ may not only cause control flow errors as described above but also severe data flow problems. The detection of data flow conflicts based on the

materialization of schema $S_I^* := (S + \Delta_I) + \Delta_T$ has at least the same complexity as respective control flow checks. Our data flow constraints from Def. 1 forbid activities with missing input data and lost updates on data elements. Respective problems may occur for S_I^* if both instance and type change delete write data links on the same data element read by other activities in the sequel. An example is depicted in Fig. 4 where Δ_T and Δ_I delete write data links related to the same data element d_1 which causes missing input data of activity G in $S^*_{incorrect}$.

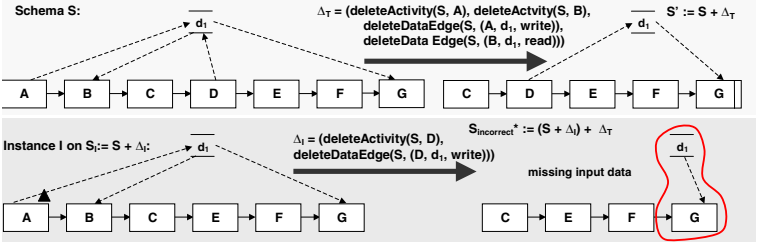

Fig. 4. Deleting All Necessary Write Accesses on Instance Data (Example)

Therefore, in the following we provide a formal proposition to exclude data flow errors for S_I^* for a magnitude of instances solely on basis of Δ_T and Δ_I.

Proposition 3 (Avoiding Missing Input Data and Lost Updates). *Let S be a WSM-Net and I be a biased instance with starting schema S and execution schema $S_I := S + \Delta_I = (N_I, D_I, NT_I, CtrlE_I, SyncE_I, \ldots)$. Assume that type change Δ_T transforms S into a correct schema S' = (N', D', NT', CtrlE', SyncE', ...). Then: Propagating Δ_T to I neither results in missing input data nor in lost updates if*

$$\forall\ mDL_1 = (d_1,\ mode_1,\ [\text{"add"}|\text{"delete"}]) \in \mathcal{AD}(S,\ \Delta_T) \cup \mathcal{DD}(S,\ \Delta_T),$$
$$\forall\ mDL_2 = (d_2,\ mode_2,\ [\text{"add"}|\text{"delete"}]) \in \in \mathcal{AD}(S,\ \Delta_I) \cup \mathcal{DD}(S,\ \Delta_I)$$
$$\text{with } mode_i \in \{read,\ write\}\ (i = 1,\ 2):$$
$$d_1 \neq d_2 \vee$$
$$mode1 = mode2 = read \vee$$
$$mDL_1 = (d_1,\ \text{"read"},\ \text{"delete"}) \vee mDL_2 = (d_2,\ read,\ \text{"delete"})\ (\clubsuit)$$

whereas

- $\mathcal{AD}(S,\ \Delta_T) := \{(d, mode, \text{"add"}) \in DataE' \setminus DataE,\ mode \in \{read,\ write\}\}$
- $\mathcal{DD}(S,\ \Delta_T) := \{(d, mode, \text{"delete"}) \in DataE \setminus DataE',\ mode \in \{read,\ write\}\}$
- $\mathcal{AD}(S,\ \Delta_I) := \{(d, mode, \text{"add"}) \in DataE_I \setminus DataE,\ mode \in \{read,\ write\}\}$
- $\mathcal{DD}(S,\ \Delta_I) := \{(d, mode, \text{"delete"}) \in DataE \setminus DataE_I,\ mode \in \{read,\ write\}\}$

In Fig. 5, Δ_T deletes activities B and F together with data edges *(B, d_2, write)* and *(F, d_2, read)*. At the instance level, Δ_I serially inserts activity Y between activities D and E with a read data link connected to data element d_2 ($\Delta_I = \{addDataEdge(S,\ (Y,\ d_2,\ read))\}$). Obviously, propagating Δ_T to S_I

leads to the problem of missing input data regarding the newly inserted activity Y. Condition ♣ from Proposition 3 indicates this conflict since both type and instance change, work on the same data element d_2 by deleting write data links and inserting new read data links for this data element. Such critical instances can be easily detected by a test derived from Proposition 3. Note that otherwise expensive data flow analyses on S_I^* would become necessary.

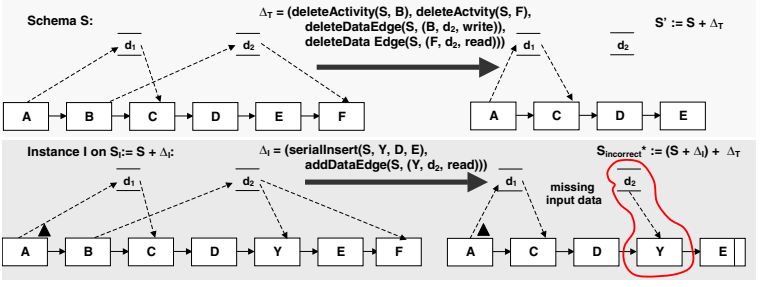

Fig. 5. Deleting Write Accesses on Data Read by Newly Inserted Activity (Example)

For a few special cases, a conflict test derived from Propostion 3 may identify potential conflicts which would not lead to a violation of the data flow constraints set out in Def. 1. Nevertheless, the presented propositions and the respective tests are very helpful to quickly and efficiently detect conflicts between concurrent data flow changes at the type and instance level.

4.3 Structural Conflicts for Seleted Process Meta Models

Some of the potential conflicts between concurrent process type and instance changes as introduced in Section 4.1 are present for other process meta models as well. One example is the deadlock-causing cycle contained in a Petri Net after the uncontrolled insertion of new order relations by process and instance changes (cf. Fig. 1). Of course a conflict test derived from Proposition 2 may be easily transferred to Activity Nets as used by WebSphere MQWorkflow [15] as well.

For other process meta models additional conflicts between process type and instance changes may occur. In the following, we exemplarily consider Activity Nets [15]. One reasonable control flow constraint for this process meta model may be to require the absence of isolated activity nodes in order to ensure clearly defined process start and end states. An example for an Activity Net containing an isolated activity node after an uncontrolled application of concurrent process type and instance changes is depicted in Fig. 6: Δ_T deletes control link (C, E) whereas Δ_I has already deleted control link (E, I). The uncontrolled propagation of Δ_T to instance-specific schema S_I leads to target schema $(S + \Delta_I) + \Delta_T$ containing isolated activity node E. Consequently, it would be a good idea to find an appropriate formal proposition setting out conditions to detect isolated

activity nodes based on the applied change operations. Based on this an efficient conflict test could be derived.

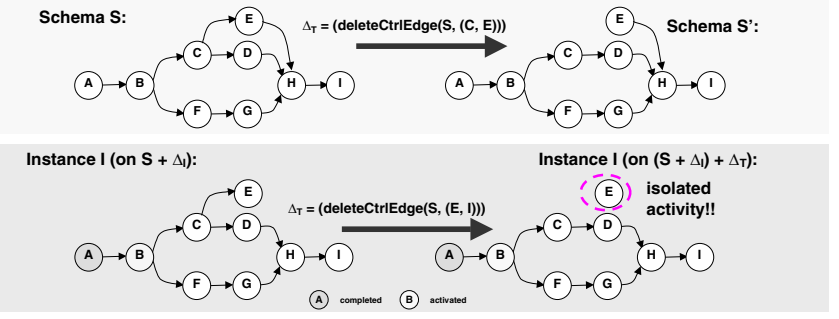

Fig. 6. Changes Causing Isolated Activity Nodes in Activity Nets

Interestingly, the test detecting isolated activities does not depend on the underlying process meta model but on the kind of applied change operation. As we have discussed in Section 3, generally, there are two levels for defining change operations. On the basis level change primitives may be carried out without special pre- and post-conditions. This is the reason why we get an isolated activity node in Fig. 6. When applying change operations on a higher semantical level, as for example defined for WSM-Nets (cf. Table 1) this specific problem is prohibited. Another problem arising in connection with basis change primitives is present for block-structured process meta models, like BPEL4WS [14] and ADEPT, namely the violation ot the block structure.

5 Illustrating Example

To summarize the results presented in this paper and to show the whole migration process followed in our approach we provide an illustrating example. In Fig. 7 process type change Δ_T transforms schema S into new schema version S' by serially inserting activites X and Y and by connecting them via a sync link (X, Y). Furthermore Δ_T deletes activities F and H together with their respective data links dL_6 and dL_7. Based on original schema S two instances have been started. Instance I_1 is biased and therefore runs according to its instance-specific schema S_{I_1} whereas I_2 is an unbiased instance still running according to S. If now type change Δ_T shall be propagated to these instances we have to check structural as well as state-related compliance for the running instances. Instance change Δ_{I_1} has serially inserted two activites U and T and sync link (T, U) between them. At first, the deadlock test derived from Proposition 2 is carried out to detect whether target schema S_1^* will contain a deadlock-causing cycle or not. After applying the graph reduction rules of Algorithm 1 we obtain that S_1^* will actually contain a deadlock-causing cycle and therefore I_1 cannot

migrate to S' (and remains running according to S). For unbiased Instance I_2 we only have to check state-related compliance as described in Section 2. Since the previous execution of I_2 can be replayed on S', I_2 is compliant with S' and therefore migrates to S' by applying appropriate marking adaptation rules [13].

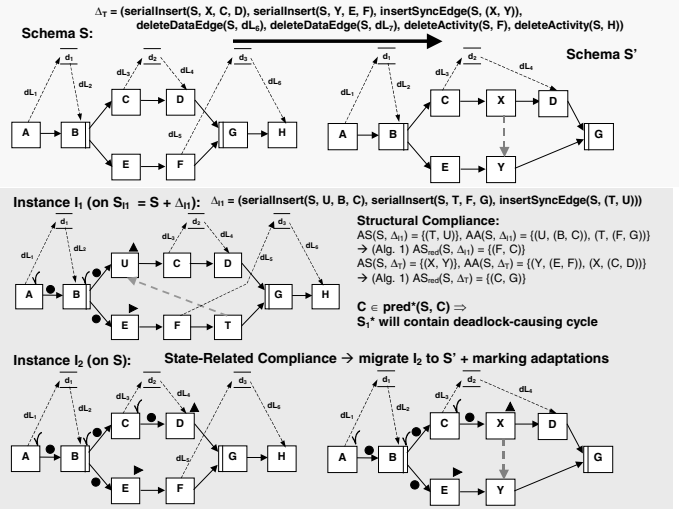

Fig. 7. Process Type Change and Instance Migration

6 Related Work

Today's workflow technology is rather weak with respect to dynamic process changes [16]. Particularly, it is unsuited for supporting long-running processes. As a consequence, process descriptions are often split into a series of smaller, short-running process fragments that are maintained as separate schemes and correlated through application data at runtime. Such a fragmented representation, however, does not provide a natural view of the process and is also unfavorable in other respects. In particular, it does not abolish the need for dynamic instance migrations (even if techniques such as late binding are applied).

Adaptive workflows have been addressed in many research papers so far [1,2, 3,5,6,7,10]. The main focus was put on providing appropriate correctness criteria for deciding about compliance of unbiased process instances with a changed process schema. More precisely, these criteria solely aim on state-related correctness when propagating a process type change to an instance. There are only few approaches [10,11] to allow process type and changes within one PMS. However, there is no **interplay** between process type and instance changes. $WASA_2$ [10], for example, realizes changes of single process instances by deriving a new

schema version with one running instance. Consequently, individually modified instances are totally excluded from further process type optimizations.

For unbiased process instances, correctness criteria range from graph equivalence [1,2,10] to trace equivalence [3,5,6,7]:

In [1], v.d. Aalst and Basten base correctness of dynamic process change on special inheritance relations between original and changed process schema. Compliance can be ensured by checking easy conditions on these two schemes. This approach also provides transformation rules which automatically adapt instance markings on the changed schema. For correctness checking, Agostini and De Michelis [2] construct reachability graphs for the original and the changed process schema. Based on this, it can be determined whether a process instance is in a state which exists on the changed process schema as well. In contrast, Weske [10] proposes to construct the purged instance graph – a subgraph of the respective process schema consisting of already passed regions for each instance. Then it has to be analyzed if there is a valid mapping between the purged instance graph and the changed process schema.

A first approach based on trace equivalence was presented by Casati et al [3]. Here a process instance is compliant with a changed process schema if the execution history of this instance can be reproduced on the change process schema as well. Ellis et al [5] present a Petri-Net based approach. A process instance is compliant with the changed process schema if the firing sequence of this instance previous to the change can be executed on the changed process schema as well. Kradolfer [6] and Sadiq et al [7] both use the compliance criterion presented in [3]. Thereby Kradolfer [6] provides conditions based on the instance execution history and the applied change operation to check compliance whereas Sadiq et al [7] focus on the treatment of non-compliant instances and temporal aspects in conjunction with dynamic change.

The described WSM-Nets are somewhat comparable to BPEL4WS (Business Process Execution Language for Web Services) [14], but with a better understanding and formal foundation regarding the use of links (called sync links in our approach). Though there is some work on exception handling in BPEL4WS [17], dynamic change issues have been completely factored out.

A detailed discussion of all these approaches can be found in [16]. It would be very interesting to learn more about the definition of change operations and the use of their specific semantics for the different process meta models.

7 Summary and Outlook

We have introduced basic work on the challenging question of how to correctly deal with concurrent process type and instance changes. At first, a comprehensive correctness criterion has been presented including structural, state-related and semantical correctness when propagating process type changes to biased process instances. Furthermore, we have derived formal propositions for constructing efficient tests which allow us to quickly and efficiently detect potential conflicts between changes at the type and instance level. These tests are based on WSM-Nets. However, we have discussed possible conflicts between process type and instance changes for other process meta models like Petri-Nets and

Activity Nets as well. In our future work on adaptive processes, we will consider semantical conflicts between concurrent process changes and their treatment as well. Furthermore, we will fully implement both, structural as well as semantical conflict tests in our current proof-of-concept prototype for process schema evolution. Hereby, it is very important to think about implementation issues like locking of instances or the order in which structural, state-related and semantical correctness is checked. In any case, the support of process type and instance changes would benefit by a more intense study of the research community.

References

1. van der Aalst, W., Basten, T.: Inheritance of workflows: An approach to tackling problems related to change. Theoretical Computer Science **270** (2002) 125–203
2. Agostini, A., De Michelis, G.: Improving flexibility of workflow management systems. In: Proc. BPM '00. LNCS 1806, Springer (2000) 218–234
3. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data and Knowledge Engineering **24** (1998) 211–238
4. Edmond, D., ter Hofstede, A.: A reflective infrastructure for workflow adaptability. Data and Knowledge Engineering **34** (2000) 271–304
5. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proc. COOCS '95, Milpitas, CA (1995) 10–21
6. Kradolfer, M., Geppert, A.: Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In: Proc. CoopIS '99, Edinburgh (1999) 104–114
7. Sadiq, S., Marjanovic, O., Orlowska, M.: Managing change and time in dynamic workflow processes. IJCIS **9** (2000) 93–116
8. Reichert, M., Dadam, P.: ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. JIIS **10** (1998) 93–129
9. Rinderle, S., Reichert, M., Dadam, P.: Evaluation of correctness criteria for dynamic workflow changes. In: Proc. Int'l Conf. on Business Process Management (BPM '03). LNCS 2678, Eindhoven, The Netherlands, Springer (2003) 41–57
10. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: HICSS-34. (2001)
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. Rinderle, S., Reichert, M., Dadam, P.: Flexible support of team processes by adaptive workflow systems. Distributed and Parallel Databases (to appear)
13. 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
14. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.0. (2002) http://www.ibm.com/developerworks/library/ws-bpel/.
15. Leymann, F., Altenhuber, W.: Managing business processes as an information ressource. IBM Systems Journal **33** (1994) 326–348
16. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems – a survey. Data and Knowledge Engineering (to appear)
17. Curbera, F., Khalaf, R., Leymann, F., Weerawarana, S.: Execption handling in the BPEL4WS language. In: BPM'03, Eindhoven (2003) 276–290