On the Common Support of Workflow Type and Instance Changes under Correctness Constraints

Manfred Reichert, Stefanie Rinderle, and Peter Dadam*

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

Abstract. The capability to rapidly adapt in-progress workflows (WF) is an essential requirement for any workflow system. Adaptations may concern single WF instances or a WF type as a whole. Especially for long-running business processes it is indispensable to propagate WF type changes to in-progress WF instances as well. Very challenging in this context is to correctly adapt a (potentially large) collection of WF instances, which may be in different states and to which various ad-hoc changes may have been previously applied. This paper presents a generic framework for the common support of both WF type and WF instance changes. We establish fundamental correctness principles, position formal theorems, and show how WF instances can be automatically and efficiently migrated to a modified WF schema. The adequate treatment of conflicting WF type and WF instance changes adds to the overall completeness of our approach. By offering more flexibility and adaptability the so promising WF technology will finally deliver.

1 Introduction

In real-world environments people are expected to flexibly deal with exceptions. Though they are trained to do so, this role is purely integrated with current WF technology [1]. Either ad-hoc deviations from the modeled WF schema are completely prohibited, thus requiring to bypass the WF system in exceptional situations, or they may cause severe problems [2]. Ad-hoc adaptations of single WF instances [2,3,4], however, represent only one kind of dynamic change. In order to support evolutionary changes, adaptations may have to be applied at the WF type level as well [3,5,6,7,8,9]. In principle, a WF type change can be accomplished by modifying the respective WF schema accordingly. In doing so, in-progress WF instances must not get into trouble due to the change. This could be achieved by finishing them according to the old schema whereas future instances are created from the new one. Obviously, this simple approach is only sufficient for workflows with short duration, but raises problems in conjunction with long-running processes. Therefore, very often it is desired to propagate a

* 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 et al. (Eds.): CoopIS/DOA/ODBASE 2003, LNCS 2888, pp. 407–425, 2003.

type change Δ, which transforms the actual schema S into a new one, to in-progress WF instances as well. Very challenging in this context is to correctly adapt a (potentially large) collection of WF instances, which may be in different states and to which various ad-hoc changes may have been previously applied. In the latter case, we have to deal with the problem that Δ shall be propagated to WF instances whose current execution schema does not completely correspond to the original schema S. Nevertheless, such "biased" WF instances must not be needlessly excluded from change propagation.

In our previous work, we dealt with ad-hoc changes of single WF instances and related WF graph transformations [2]. The main emphasis was put on implementation and usability issues. The objective of this paper is to develop a formal framework for both the propagation of WF type changes to already running WF instances and ad-hoc changes of single WF instances. We present different change scenarios and have a look at fundamental principles concerning the design of adaptive WF models. For this, we establish basic correctness principles and position formal theorems. Taking a simple, but powerful WF meta model, we exemplarily show how correctness can be efficiently checked and which information is needed for this. In addition, we introduce well-defined rules and procedures for migrating WF instances to a modified schema.

Section 2 sketches the WF meta model, which we use in this paper to illustrate fundamental principles of our approach. However, the presented approach is applicable in conjunction with comparable WF meta models as well. Section 3 develops our approach for dynamic WF changes, focusing on general correctness principles as well as on implementable rules for ensuring correctness when a change is applied. Section 4 deals with conflicting changes at the type and instance level, and it shows under which conditions WF type changes may be propagated to biased WF instances as well. We discuss related work in Section 5 and conclude with a summary in Section 6.

2 WF Modeling and Execution Basics

For each business process to be supported a *WF type* T has to be defined. It is represented by a *WF schema graph* S of which different versions V_1, ..., V_n may exist (reflecting the evolution of T). To simplify matters, we assume that there is only one version V_n from which new WF instances can be created. This is not really required. However, in the present paper we put the focus on formal considerations and do not deal with versioning issues in more detail.

2.1 Modeling and Execution of Workflows

A WF schema comprises a set of *activities* and defines the control and data flow between them. *Control flow* is modeled by linking activities with *control edges*, which may be optionally associated with *transition conditions*. The use of control edges must not lead to cyclic order relationships since this may cause deadlocks at runtime (see below). Depending on the defined control edges and the chosen

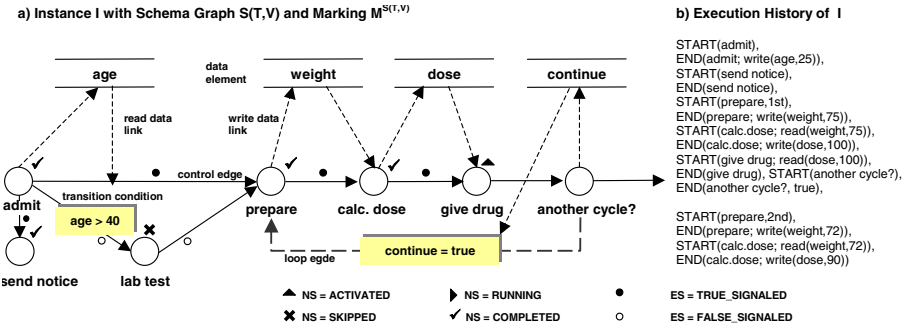


Fig. 1. WF Instance Example

transition conditions, sequences, parallel branchings, and conditional branchings can be described. For the modeling of loop backs, an additional edge type (*loop backward edge*) is provided, which allows us to distinguish between "undesired" and "desired" cycles. To simplify matters, we assume that an activity must not have more than one outgoing loop edge and that the activity nodes which constitute the loop body are well-defined (cf. Def. 1). Finally, *data flow* between activities is realized by connecting them with global *data elements*. For this, read and write *data edges* are provided. An example is depicted in Fig. 1. Formally:

Definition 1 (WF Schema Graph). *A tuple S with S = (N, D, CtrlEdges, LoopEdges, DataEdges, EC) is called a WF schema graph, if the following holds:*

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

such that

1. S_{fwd} *= (N, CtrlEdges) is an acyclic graph*
2. $\forall$ $(n_1, n_2) \in$ *LoopEdges:* $n_2 \in pred(S, n_1)$
3. $\forall$ $(n_1, n_2) \in$ *LoopEdges:* $succ(S, n_2) \subseteq L_{body}(n_1, n_2) \cup succ(S, n_1) \wedge pred(S, n_1) \subseteq L_{body}(n_1, n_2) \cup pred(S, n_2)$
4. $\forall$ $(n_1, n_2), (m_1, m_2) \in$ *LoopEdges:* $n_1 \neq m_1$

The sets pred(S,n)/succ(S,n) comprise all direct and indirect predecessors/successors of n via control edges and $L_{body}(n_1, n_2) := succ(S, n_2) \cap pred(S, n_1) \cup \{n_1, n_2\}$.

The status of a single WF activity is initially set to `NOT_ACTIVATED`. When all preconditions for activity execution are met (see below), it changes to `ACTIVATED`. The activity is then either started automatically or corresponding work items are inserted into user worklists. When starting execution, activity status changes to `RUNNING` and the associated application component is invoked. Finally, at successful termination, status passes to `COMPLETED`.

Execution of a newly created WF instance starts with those activities that have no incoming control edge. When completing an activity, its outgoing edges are either evaluated to `TRUE_SIGNALED` or `FALSE_SIGNALED`, depending on their transition conditions. This, in turn, leads to re-evaluation of target activities. Generally, an activity may be activated as soon as all incoming edges have been signaled and at least of them is marked with `TRUE_SIGNALED`. Consequently, if all incoming edges are marked as `FALSE_SIGNALED`, the activity cannot be executed anymore. Its status is then set to `SKIPPED`, which may lead to cascaded skipping of subsequent activities. A loop edge is evaluated whenever its source activity terminates. If the associated loop condition evaluates to true, outgoing control edges will not be evaluated, the loop edge will be signalled, and all nodes contained within the loop body will be reset to their initial state. Finally, execution of a WF instance will terminate if all activities are in one of the states `COMPLETED` or `SKIPPED`.

Each WF instance I is associated with a schema S = S(T,V), where T denotes the WF type of I and V the version of the WF schema graph to be taken for execution. (Note that other WF instances may be based on S as well). The control state of I is captured by a marking function M^S =(NS, ES). It assigns to each activity n its current status NS(n) and to each control and loop edge its marking ES(e) (cf. Fig. 1). These markings are determined according to the rules described above, 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.

Definition 2 (WF Instance). *A WF instance I is defined by a tuple (T, V, $M^{S(T,V)}$, $Val^{S(T,V)}$, $\mathcal{H}$) where*

- *T denotes the WF type of I and V the version of the schema graph $S := S(T,V) = (N, D, CtrlEdges, LoopEdges, ...)$ according to which I is executed.*
- *M^S =(NS^S, ES^S) reflects the current marking of nodes $NS^S: N \mapsto NodeStates$ and edges $ES^S: CtrlEdges \cup LoopEdges \mapsto EdgeStates$*
- *Val^S is a function on D. $Val^S(d)$ 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. It logs information about start / completion of activities. For each started activity X the values of the data elements read by X and for each completed activity Y the values of the data elements written by Y are logged (logical view).*

As described above, WF instances preserve their markings when proceeding in the flow of control. Thus M^S always reflects a consolidated view of the previous

execution of I. As we will see later, this property is very useful in connection with dynamic WF instance changes. Formally:

Lemma 1 (Preserving Instance Markings). *Let I be a WF instance with schema graph $S = (N, D, ...)$ and marking $M^S = (NS, ES)$. Further, let $x \in N$ be an arbitrary activity node with NS(x) $\in$ {*COMPLETED*, *SKIPPED*}. Then:*
$\forall\ n \in pred(S, x)$: NS(n) $\in$ {COMPLETED, SKIPPED}.

2.2 Defining and Changing Schema Graphs

Table 1 contains some primitives that can be used to define and modify schema graphs. Each primitive has a well-defined semantics and is associated with formal pre-/post-conditions, necessary to preserve (structural) correctness of the respective schema (cf. Def. 1). In this paper, we exemplarily restrict our considerations to the avoidance of deadlocks that may be caused due to cyclic order relationships (via control edges). Generally, additional constraints exist. Concerning data flow, for example, no lost updates must occur during runtime and all data elements read by an activity must always have been written by preceding activities, independently of chosen execution branches. There are other primitives (e.g., to update edge conditions), which we do not consider in the following. Finally, change primitives serve as basis for defining high-level operations (e.g., to shift an activity from its current to another position) and for deriving formal conditions for them. However, this is outside the scope of this paper.

Table 1. Examples of Basic Change Primitives

addCtrlEdge(S, n_{src}, n_{dst})	Pre: $(n_{src} \notin \text{succ}(S, n_{dst}) \cup \{n_{dst}\}) \wedge (\forall (n_1,n_2) \in \text{LoopEdges}:$ $[n_{src} \in L_{body}(n_1,n_2) \Leftrightarrow n_{src} \in L_{body}(n_1,n_2)])$ Post: CtrlEdges' = CtrlEdges $\cup \{n_{src} \rightarrow n_{dst}\}$
addActivity(S, n_{ins}, Preds, Succs)	Pre: $(\forall$ p $\in$ Preds, $\forall$ s $\in$ Succs: s $\notin$ pred(S, p)) $\wedge$ $(\forall (n_1,n_2) \in$ LoopEdges: (Preds $\cup$ Succs) $\subseteq L_{body}(n_1,n_2) \vee$ (Preds $\cup$ Succs) $\cap L_{body}(n_1,n_2) = \emptyset$ Post: N' = N $\cup \{n_{ins}\}$ CtrlEdges' = CtrlEdges $\cup$ {p $\rightarrow n_{ins}$ | p $\in$ Preds} $\cup \{n_{ins} \rightarrow$ s | s $\in$ Succs}
deleteCtrlEdge(S, n_{src}, n_{dst})	Post: CtrlEdges' = CtrlEdges $\neg \{n_{src} \rightarrow n_{dst}\}$
deleteActivity(S, n_{del})	Post: N' = N $\neg \{n_{del}\}$ CtrlEdges' = CtrlEdges $\neg$ {a $\rightarrow$ b | $n_{del} \in$ {a, b}} $\cup$ { p $\rightarrow$ s with EC(p $\rightarrow$ s) = ec | p $\rightarrow n_{del}$, $n_{del} \rightarrow$ s $\in$ CtrlEdges $\wedge$ EC($n_{del} \rightarrow$ s) = ec}

3 Dynamic Change Basics

In this section, we present issues related to dynamic change correctness in a formal and rigorous style. Due to lack of space, we restrict our considerations to changes definable by the primitives from Table 1. First of all, we do not make a

difference between changes of single instances and adaptations of a collection of instances (e.g., due to a type change). Instead we focus on fundamental issues related to dynamic instance changes. In the following, let I be an instance with schema graph S and marking M^S. Assume that S is transformed into a correct schema graph S' by applying change Δ. Two challenging issues arise:

1. Can Δ be correctly *propagated* to I, i.e., without causing errors or inconsistencies? For this case, I is said to be *compliant* with S'.
2. Assuming I is compliant with S', how can we smoothly *migrate* it to S' such that its further execution can be based on S'? Which state (marking) adaptations become necessary in this context?

We will show that these two issues are fundamental for the design of any adaptive WF model. While the first one concerns pre-conditions on the state of I, the second issue is related to post-conditions that must be satisfied after the change has been applied. In any case, we have to find an efficient solution, which enables automatic and correct compliance checks as well as instance migrations. In Section 3.1 we introduce general correctness principles which address the above issues. Based on them, for the presented WF meta model (cf. Section 2) we develop formal pre-conditions for ensuring compliance of instances with a modified schema (cf. Section 3.2). Section 3.3 shows how to efficiently determine follow-up markings of compliant WF instances when migrating them to the new schema. Section 3.4 concludes with a discussion of different change scenarios.

3.1 Dynamic Change Correctness

To illustrate potential problems that may result from the uncontrolled migration of WF instances consider schema graph S from Fig. 2a. Let us assume that S is correctly transformed into S' by inserting two activities and a data dependency between them (cf. Fig. 2a, 2b). Assume that this change shall be applied to the instances from Fig. 2c) (currently based on S) but without performing any compliance check. Concerning I_1 no problem would occur, since its execution has not yet entered the change region. Uncontrolled migration of I_2, however, would cause malfunctions: Firstly, an inconsistent marking would result (cf. Lemma 1), thus leading to an undefined execution state. Secondly, activity `give drug` may be invoked though the data element `allergyData` read by this activity may not have been written. Concerning I_3 migration would be possible. However, when migrating I_3 to S', activation of activity `prepare` has to be undone and corresponding work items must be removed from worklists. Additionally, the newly inserted activity `test` must be activated. This example demonstrates that applicability of a dynamic change depends on current instance state as well as on applied change primitives. Furthermore, when migrating compliant instances, markings and worklist structures must be correctly adapted.

Comparable with serializability in DBMS, we need general principles which allow us to argue about the correctness of dynamic changes. In more detail, we require a formal criterion for deciding whether a given WF instance can be smoothly migrated to the modified schema or not. In addition, we must be able

to determine correct new markings resulting from such a migration. One of our design goals is to define these correctness criteria independently of the operational semantics of the used WF meta model and the offered change operations. This allows us to apply them in different scope and to different WF meta models, thus providing a good basis for reasoning about the correctness of rules and methods for checking compliance and for migrating compliant instances.

Intuitively, instance I is compliant with the modified schema S' if I could have been executed according to S' as well and would have produced the same effects on data elements [3,10]. Trivially, this will be always the case if I has not yet entered the region affected by the change. Generally, we need information about previous execution to decide this property and to determine correct follow-up markings for compliant instances. To derive such a general compliance principle, at the logical level we make use of the execution history usually kept for each WF instance (cf. Fig. 1 and 2). We assume that this history logs events related to the start and termination of activity executions (cf. Def. 2).

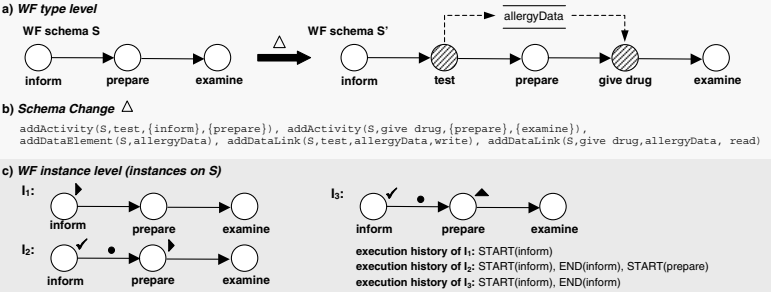


Fig. 2. Schema Graph and Related Instances

Obviously, instance I with history $\mathcal{H}$ is compliant with S' and can migrate to S' if $\mathcal{H}$ could have been produced on S' as well. We then obtain a correct new marking by "replaying" all events from $\mathcal{H}$ on S' in sequential order. Taking our example from Fig. 2 this property holds for I_1 and I_3 but does not apply to I_2. When replaying $\mathcal{H}_3$ on S' we obtain node markings as sketched above.

The described criterion is still too restrictive to serve as general correctness principle. Concerning loop changes it may needlessly exclude instances from migrations. As an example take instance I from Fig. 1 where the depicted loop is in its 2^{nd} iteration. Assume that schema S is modified by applying change `addActivity(S, perform test, {prepare}, {give drug})`. Taking the above criterion this change would not be allowed since previous loop iteration of I is not compliant with the new schema; i.e., $\mathcal{H}$ cannot be produced based on the modified schema. However, excluding such instances from migrations very often is not in accordance with practice. To overcome this restrictiveness we relax the above criterion by (logically) discarding those history

entries produced within another loop iteration than the last (completed loops) or the current one (running loops). We denote this reduced view $\mathcal{H}$ as $\text{red}_{\mathcal{H}}$. Based on this we now define a general correctness principle for dynamic WF changes:

Axiom 1 (Dynamic Change Correctness) *Let $I = (T, V, M^S, Val^S, \mathcal{H})$ be a WF instance with correct schema graph $S = S(T,V)$ and marking M^S. Assume that S is transformed into a correct schema graph S' by applying change Δ. Then:*

1. *Δ can be correctly propagated to I iff $\text{red}_{\mathcal{H}}$ can be produced on S' as well (For this case, I is said to be compliant with S').*
2. *Assume I is compliant with S'. When propagating Δ to I the correct marking $M^{S'}$ of I on S' can be obtained by replaying $\text{red}_{\mathcal{H}}$ on S'.*

These two basic properties satisfy our design goals since they do not depend on the operational semantics of the used WF meta model and the changes applied. Axiom 1 can therefore serve as fundamental correctness principle for adaptive workflow. Furthermore, it does not needlessly exclude instances from migrations on condition their execution will not get into trouble due to the change. Altogether Axiom 1 provides a good basis for arguing about dynamic change correctness. However, it would certainly be no good idea to guarantee compliance and to determine new markings of compliant instances by accessing the whole execution history and trying to replay it on the modified schema since this may cause a performance penalty. Note that histories comprise voluminous data and are usually not kept in primary storage. In the following we present optimized rules and procedures for ensuring correctness according to Axiom 1.

3.2 Rules for Checking Compliance

For the WF meta model from Section 2 and related change primitives we exemplarily show under which conditions compliance (cf. Axiom 1,1) can be guaranteed. Our basic design principles have been as follows:

1. We consider change semantics and context in order to derive precise compliance rules and to state which information is needed for checking them.
2. We make use of dynamic properties of the WF model. Particularly, the derivation of compliance rules benefits from activity markings which already provide a consolidated view on the (reduced) history of a WF instance.

We omit unnecessary details and focus on compliance rules for selected primitives from Table 1. Based on them, one can easily develop high-level change operations and related compliance rules. Since the latter can be derived by merging compliance conditions of the change primitives applied and by discarding unnecessary expressions (e.g., conditions on nodes not present in the original schema graph), we do not further consider complex change operations in this paper.

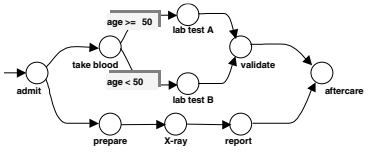
Let I be an instance with schema S, marking M^S, and history $\mathcal{H}$. Assume that S is transformed into correct schema S' by applying one of the primitives from Table 1. The challenge is to find conditions under which we can ensure compliance of I with S' (cf. Axiom 1,1). Table 2 summarizes some well-founded compliance conditions. Based on them we can state the following theorem:

Table 2. Examples of Compliance Rules

Change Operation Δ ...	... and Related Compliance Condition *Compliant(S, Δ, NS, ES, Val^S, $\mathcal{H}$)*
addActivity(S,n_{ins},Preds,Succs)	($\forall$ n $\in$ Preds: NS(n) = SKIPPED) $\vee$ ($\forall$ n $\in$ Succs: NS(n) $\in$ {NOT_ACTIVATED, ACTIVATED} $\vee$ (NS(n) = SKIPPED $\wedge$ $\forall$ m $\in$ succ(S,n): NS(m) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED})
addCtrlEdge(S, n_{src}, n_{dst}) (with EC($n_{src} \rightarrow n_{dst}$) = =True)	NS(n_{dst}) $\in$ {NOT_ACTIVATED, ACTIVATED} $\vee$ (NS(n_{dst}) $\in$ {RUNNING, COMPLETED} $\wedge$ NS(n_{src}) $\in$ {COMPLETED} $\wedge$ (e_i = END(n_{src}), e_j = START(n_{dst}) $\in \mathcal{H}$, $\Rightarrow i < j$)) $\vee$ (NS(n_{dst}) $\in$ {RUNNING, COMPLETED} $\wedge$ NS(n_{src}) = SKIPPED $\wedge$ ($\forall$ n $\in N_1$ with NS(n) = COMPLETED, e_j = END(n), e_i = START(n_{dst}) $\in \mathcal{H}$, $\Rightarrow j < i$)) $\vee$ (NS(n_{dst}) = SKIPPED $\wedge$ NS(n_{src}) $\in$ {NOT_ACTIVATED, ACTIVATED, RUNNING, COMPLETED} $\wedge$ ($\forall$ n $\in N_2$: NS(n) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED})) $\vee$ (NS(n_{dst}) = SKIPPED $\wedge$ NS(n_{src}) = COMPLETED $\wedge$ ($\forall$ n $\in N_2$ with NS(n) $\in$ {RUNNING, COMPLETED}: (e_j = START(n), e_i = END(n_{src}) $\in \mathcal{H}$, $\Rightarrow j > i$) })) $\vee$ (NS(n_{dst}) = NS(n_{src}) = SKIPPED $\wedge$ ($\forall$ s $\in N_2$ with NS(s) $\in$ {RUNNING, COMPLETED}, $\forall$ p $\in N_1$ with NS(p) = COMPLETED: e_i = END(p), e_j = START(s) $\in \mathcal{H}$, $\Rightarrow j > i$)) where N_1:= pred(S, n_{src})$\neg$ pred(S, n_{dst}) $\cup$ {n_{src}}, N_2:= succ(S, n_{dst})$\neg$ succ(S, n_{src}) $\cup$ {n_{dst}}
deleteActivity(S,n_{del})	NS(n_{del})) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}
deleteCtrlEdge(S,n_{src},n_{dst})	(NS(n_{dst}) $\in$ NOT_ACTIVATED, ACTIVATED) $\vee$ (ES($n_{src} \rightarrow n_{dst}$) = FALSE_SIGNALED $\wedge$ (($\exists$ n $\rightarrow n_{dst} \in$ CtrlEdges, n $\neq n_{src}$) $\vee$ ($\forall$ n $\in$ succ(S, n_{dst}): NS(n) $\notin$ {RUNNING, COMPLETED})) $\vee$ (ES($n_{src} \rightarrow n_{dst}$) = TRUE_SIGNALED $\wedge$ (($\nexists$ n $\rightarrow n_{dst} \in$ CtrlEdges, n $\neq n_{src}$) $\vee$ ($\exists$ e = n $\rightarrow n_{dst} \in$ CtrlEdges, n $\neq n_{src}$ with ES(e) $\neq$ FALSE_SIGNALED))
addDataElement(S,d)	no condition
deleteDataElement(S,d)	val(d) = UNDEFINED
addDataLink(S,n,d,read)	NS(n) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}
addDataLink(S,n,d,write)	NS(n) $\neq$ COMPLETED

Theorem 1 (Compliance Rules). *Let I be a WF instance with schema graph S, marking M^S = (NS, ES), data values Val^S, and execution history $\mathcal{H}$. Assume that S is transformed into a correct schema graph S' by applying change operation Δ to it. Then: I is compliant with S' (according to Axiom 1,1) $\Leftrightarrow$ Compliant(S, Δ, NS, ES, Val^S, $\mathcal{H}$) = TRUE (cf. Table 2).*

Due to lack of space we omit formal proofs. Instead we exemplarily describe compliance conditions for the primitives `addActivity` and `addCtrlEdge`. Regarding insertion of an activity n_{ins} between two node sets *Preds* and *Succs* compliance can be guaranteed if all nodes from *Succs* actually possess one of the markings `ACTIVATED` or `NOT_ACTIVATED`. In this case, none of the successors of n_{ins} has yet written any entry into $\mathcal{H}$. Furthermore, compliance can be ensured if all nodes from *Preds* are marked as `SKIPPED`. Then n_{ins} is skipped as well, i.e., its insertion has no effect on compliance. Finally, n_{ins} may be inserted as predecessor of a skipped node provided that none of the successors of this node has a marking other than `ACTIVATED`, `NOT_ACTIVATED`, or `SKIPPED` (see Fig. 3).



The addition of a control edge $n_{src} \rightarrow n_{dst}$ will always be possible if NS(n_{dst}) $\in$ {`ACTIVATED`, `NOT_ACTIVATED`} applies. In case n_{dst} is marked as `SKIPPED` compliance can be guaranteed if all successors of n_{dst} (less the successors of n_{src}) possess one of the markings `ACTIVATED`, `NOT_ACTIVATED`, or `SKIPPED`.

Under certain conditions dynamic insertion of a control edge $n_{src} \rightarrow n_{dst}$ is even allowed if n_{dst} has been already started or completed. As an example

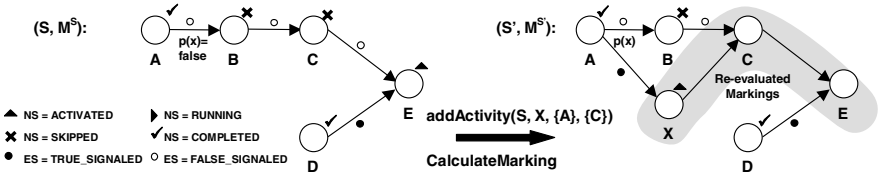


Fig. 3. Insertion before a Skipped Node

take the process schema (medical workflow) from the previous figure. Assume that an additional control edge is inserted between activities `test B` and `X-ray`. Concerning instances for which both activities are completed insertion is (only) allowed if `test B` had written its end entry into $\mathcal{H}$ before the start entry of `X-ray` was logged. As a second example, consider an instance I where `test B` is marked as `SKIPPED` and `X-ray` as `COMPLETED`. Taking this marking I would be only compliant with S' if `take blood` had been completed before `X-ray` started ($N_1 = \{$`take blood`$\}$).

Compliance conditions for the deletion of activities and control edges as well as for data flow changes are summarized in Table 2. In a similar way we can derive compliance rules for other primitives, e.g., insertion or deletion of loop edges or update of transition conditions. Altogether we can state that compliance – as postulated by Axiom 1,1 – can be checked on basis of current activity markings; i.e., we usually must not explicitly check the producibility of whole execution histories on the modified schema.

3.3 How to Correctly Adapt Workflow Instance Markings?

We have described how compliance can be ensured and which information is needed. Our main goal was to prevent access to the whole execution history. By holding this maxim we now show how compliant instances can be migrated to the changed schema. One problem to be solved is the efficient and correct adaptation of activity markings. According to Axiom 1,2 the marking of a migrated instance must be the same as it could be obtained when replaying the respective (reduced) history on the new schema. How extensive marking adaptations turn out for instance I depends on the kind and scope of the change. Except for initialization of newly inserted nodes and edges, no adaptations will become necessary if execution of I has not yet entered the change region. In other cases extensive marking adaptations may be required. An activity `X`, for example, may have to be deactivated if control edges are inserted with `X` as target activity. Conversely, a newly added activity will have to be immediately activated or skipped if all predecessors possess a final marking. As shown in Fig. 3 it may even become necessary to undo the skipping of nodes when inserting an activity.

We now describe how markings can be automatically and efficiently adapted when migrating compliant instances. Initially, we can restrict marking evaluations to those nodes and edges, which constitute the context of a change region. We sketch how these sets can be determined for selected change primitives as well

as for complex changes. Based on this, we present an algorithm which correctly calculates new markings for compliant instances.

Table 3. Node and Edge Sets to be Evaluated

op = addActivity(S, n_{ins}, Preds, Succs)	$N_{check}(op)$:= Succs (∪ $\{n_{ins}\}$ if Preds = ∅) $E_{check}(op)$:= {p → n_{ins} ∈ CtrlEdges' | p ∈ Preds}
op = deleteActivity(S, n_{del})	$N_{check}(op)$:= {n ∈ N | n_{del} → n ∈ CtrlEdges } $E_{check}(op)$:= ∅
op = addCtrlEdge(S, n_{src}, n_{dst})	$N_{check}(op)$:= {n_{dst}}, $E_{check}(op)$:= {n_{src} → n_{dst}}
op = deleteCtrlEdge(S, n_{src}, n_{dst})	$N_{check}(op)$ = {n_{dst}}

Table 3 shows node and edge sets whose markings must be initially evaluated when the respective change operation *op* is applied – we denote these sets as $N_{check}(op)$ and $E_{check}(op)$ respectively. Depending on the evaluation result, inspection of additional nodes and edges may become necessary. As a first example, take the dynamic insertion of an activity n_{ins}. Firstly, all incoming control edges of n_{ins} must be evaluated. Depending on this, n_{ins} either has to be activated, skipped, or left in its initial state. (Note that an initial evaluation of n_{ins} only becomes necessary if $Preds = \emptyset$ holds.) Secondly, all successors of n_{ins} must be re-evaluated as well. Due to the insertion of n_{ins}, activation or skipping of these activities may have to be undone. Regarding the insertion of a control edge, the marking of both, the newly added edge and its target node n_{dst} have to be re-evaluated, i.e., we obtain $N_{check}(op) = \{n_{dst}\}$ and $E_{check}(op) := \{n_{src} \rightarrow n_{dst}\}$. The latter will be also required if the evaluation of the edge marking results in `NOT_SIGNALED` (for this case n_{dst} may have to be deactivated).

Concerning a complex change $\Delta = op_1, \ldots, op_n$ the total node and edge sets $N_{check}(\Delta)$ and $E_{check}(\Delta)$ to be (initially) evaluated can be determined with Algorithm 1. In principle, we obtain them by unifying the corresponding sets of the applied change operations. However, since these operations can be based on each other, there may be temporarily generated nodes or edges not present in the resulting schema graph anymore. This is considered by Algorithm 1.

Algorithm 1: CalcEvalSet(S, S', $\Delta = op_1, \ldots, op_n$) $\longrightarrow$ $N_{check}(\Delta)$, $E_{check}(\Delta)$

$E_{check}(\Delta) := \emptyset$; $N_{check}(\Delta) := \emptyset$;
for `i:=1 to n` **do**
  $E_{check}(\Delta)$:= E_{check} (Δ) ∪ $E_{check}(op_i)$; $N_{check}(\Delta)$:= N_{check} (Δ) ∪ $N_{check}(op_i)$;
done
$E_{check}(\Delta)$:= $E_{check}(\Delta)$ ∩ E'; $N_{check}(\Delta)$:= $N_{check}(\Delta)$ ∩ N';

Let I be an instance with schema S and marking M^S. Assume S is transformed into a correct schema S' by applying change $\Delta = op_1, \ldots, op_n$. Assume

Algorithm 2: CalcMarking(S, S', (NS, ES),$N_{check}(\Delta)$, $E_{check}(\Delta)$) $\longrightarrow$ (NS', ES')

```
N_check := N_check(Δ); E_check := E_check(Δ);
forall e ∈ E' ∩ E do ES'(e) = ES(e) done;
forall e ∈ E' ¬ E do ES'(e) = NOT_SIGNALED done
forall n ∈ N' ∩ N do NS'(n) = NS(n) done;
forall n ∈ N' ¬ N do NS'(n) = NOT_ACTIVATED done

repeat
   while E_check ≠ ∅ do
      fetch an edge e = n_src → n_dst from E_check;
      determine marking newES of e according to marking of n_src and transition cond. EC'(e)
      if ES'(e) ≠ newES then
          ES'(e) := newES , N_check := N_check ∪ {n_dst}
      endif
   done
   while N_check ≠ ∅ do
      fetch a node n from N_check;
      determine marking newNS of n according to markings of incoming control edges of n.
          if NS'(n) ≠ newNS then
              if newNS = SKIPPED or NS'(n) = SKIPPED then
                  E_check := E_check ∪ {e = n_src → n_dst ∈ E' | n_src = n }
          endif
          NS'(n) := newNS
      endif
   done
until E_check = ∅ and N_check = ∅;
```

further that I is compliant with S'. Algorithm 2 then correctly determines new marking $M^{S'}$ of I on S'. Basic to this are the marking and execution rules of our WF meta model. Algorithm 2 starts with sets $N_{check}(\Delta)$ and $E_{check}(\Delta)$ as input. If the markings of respective nodes or edges are adapted during execution of Algorithm 2, context nodes and edges will be re-evaluated as well, etc. By means of Algorithms 1 and 2, total expenditure for marking adaptations can be significantly reduced when compared to the re-evaluation of all node and edge markings or the complete replay of all history events on the new schema. Nevertheless, our approach guarantees correctness according to Axiom 1,2. Formally:

Theorem 2 (Optimized Marking Adaptations). *Let I = (T, V, M^S, Val^S, $\mathcal{H}$) be an instance with schema S = S(T,V) and marking M^S. Assume that change Δ transforms S into a correct schema S' and I is compliant with S'. Then: With CalculateMarking(S, S', M^S, $N_{check}(\Delta)$, $E_{check}(\Delta)$) (cf. Alg. 2) we obtain the correct marking $M^{S'}$ of I (cf. Axiom 1,2) when migrating it to S'; i.e., we obtain the same marking as it would result when replaying $\mathcal{H}$ on S'.*

While Algorithm 1 has to be carried out only once at change definition time, Algorithm 2 must be applied for each instance to be migrated. The complexity of Algorithm 2 can be estimated by $O(n)$ (where n corresponds to the number of activities of schema S'). Additionally, for each WF instance complexity $O(n)$ arises from the described compliance checks.

As a first example, take the activity insertion from Fig. 3. As already shown, the depicted instance is compliant with the modified schema. With Algorithm 1 we obtain $N_{check} = \{C\}$ and $E_{check} = \{A \rightarrow X\}$. Furthermore, when running Al-

gorithm 2 with these sets as input, the newly inserted activity X will be activated whereas skipping of C and activation of E will be undone. A second example, which shows a change at the type level (parallel ordering of activities that have been executed serially so far) and its propagation to compliant instances is depicted in Fig. 4. Note that both, necessary checks and marking adaptations can be completely automated in our approach. Thus the "dynamic change bug" as discussed in WF literature (e.g. [9,11]) is not present in our approach.

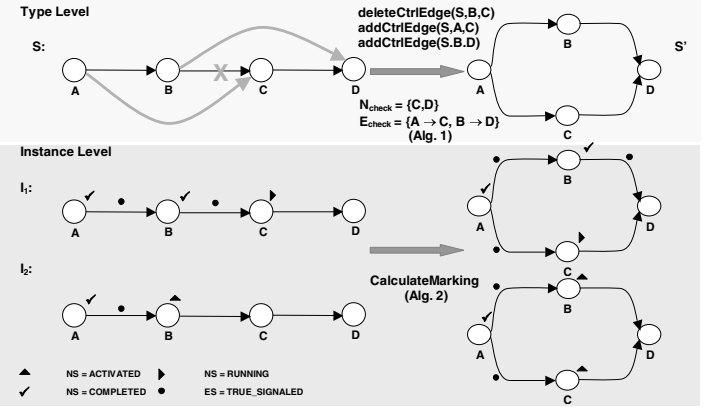


Fig. 4. Instance Migrations Due To Type Change

3.4 Realizing Workflow Type and Workflow Instance Changes

The presented correctness principles, compliance rules, and migration procedures are applicable for both WF schema evolution (incl. change propagation to running instances) and ad-hoc changes of single instances.

WF schema evolution: First of all, we allow designers to restrict the set of migratable instances by specifying appropriate selection predicates (based on WF attributes). For each selected instance I the WfMS checks whether it is compliant with the modified schema or not. In the former case I is re-linked to the new schema S' and its further execution is based on S' (cf. Fig. 5 b). Among other things this includes adaptation of markings and related data structures as described. Non-compliant instances may be finished according to the old schema version or be rolled back to a compliant state to enable their migration. In connection with loops such a compliant state may be reachable when a loop enters its next iteration. A discussion of this special case and the support of delayed instance migrations, however, is outside the scope of this paper.

(Ad-hoc) changes: An ad-hoc change of instance I may become necessary, for example, to deal with exceptional situations. For change definition, high-level operations are offered to users (e.g., to jump forward in the flow or to shift activities) which are based on the described primitives. All runtime deviations are

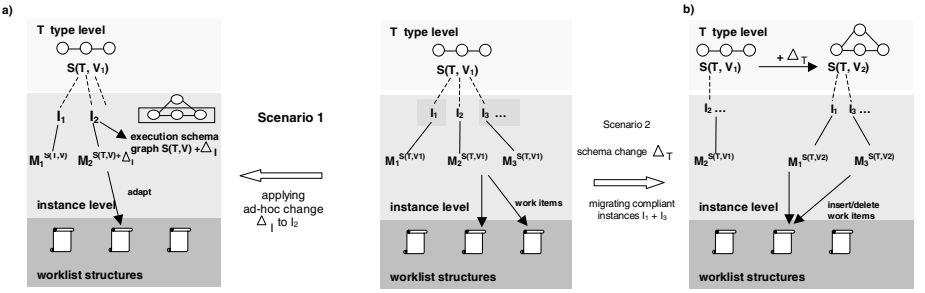


Fig. 5. Managing Type and Instance Changes

properly integrated with respect to authorization and are logged in the change history of I. Obviously, this results in an instance-specific execution schema $S_I = S + \Delta_I$ which differs from the original schema S (cf. Def. 3) – Δ_I is called the *bias* of I (with respect to S) and describes the set of instance-specific changes $op_I^1, \ldots, op_I^n$ that have been applied to I so far. Execution of I as well as future change definitions are logically based on S_I (cf. Fig. 5 a).

Definition 3 (Biased Instance). *A biased instance I is described by a tuple $(T, V, \Delta_I, M^{S+\Delta_I}, Val^{S+\Delta_I}, \mathcal{H})$, where $S = S(T,V)$ corresponds to the schema version from which I was created and Δ_I comprises instance-specific changes $op_I^1, \ldots, op_I^n$ 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 execution schema of I.*

Trivially, the execution schema S_I of an unbiased instance I (with $\Delta_I = \emptyset$) corresponds to its original schema S. A biased instance always keeps the reference to its original schema. As we will see in the next section, under certain conditions this allows us to propagate type changes to biased instances as well. How biased instances are "physically" represented, whether S_I is materialized or only Δ_I is stored and other implementation issues are outside the scope of this paper.

4 Conflicting Type and Instance Changes

Biased WF instances must not be needlessly excluded from adapting to a WF type change. As an example take a patient treatment process. Even though physicians may have deviated from the original WF schema S at the instance level (e.g., by inserting or skipping activities) this must not prohibit the propagation of future WF type changes to these instances on condition that they are not conflicting with current instance state and previously applied ad-hoc changes. In this section, we sketch what is needed and which issues arise in this context.

Let $I = (T, V_n, \Delta_I, ...)$ be a biased WF instance (cf. Def. 3) which was created from schema version $S = S(T, V_n)$ and to which instance-specific changes $op_I^1, \ldots, op_I^n$ – described by bias Δ_I – have been applied so far. Assume that a new schema version $S' = S(T, V_{n+1})$ is derived from S by applying WF type change Δ_T ($= op_T^1, \ldots, op_T^m$) to it ($S' = S + \Delta_T$). Then the following issues arise:

1. May Δ_T be propagated to I as well though the current execution schema $S_I = S + \Delta_I$ of I differs from the original schema S?
2. If change propagation is possible how can it be efficiently and correctly accomplished? Which execution schema S_I' (and marking $M^{S'_I}$) must result?

4.1 Correctness Issues

Comparable to the migration of unbiased instances (cf. Section 3) we introduce a general criterion that allows us to argue about the two issues described above. Obviously, when propagating a WF type change Δ_T to a biased WF instance I we do not only have to consider its current marking M^{S_I} but must also deal with structural and semantical conflicts that may exist between the "concurrent" changes Δ_I and Δ_T (Note that Δ_I as well as Δ_T have been based on S). In this paper we restrict our considerations to structural conflicts. A comprehensive treatment of semantically conflicting changes is given in [12].

Axiom 2 (Propagating Type Changes To Biased Instances) *Let T be a WF type with actual schema version $S = S(T, V_n)$. Assume that a new schema version $S' = S(T, V_{n+1})$ is derived by applying type change Δ_T to S. Then:*
Δ_T may be propagated to WF instance $I = (T, V_n, \Delta_I, \ldots) :\Leftrightarrow$

1. *$S^* = (S + \Delta_I) + \Delta_T$ is a correct schema graph, i.e., Δ_T can be correctly applied to the execution schema $S_I = (S + \Delta_I)$.*
2. *I is compliant with S^*; i.e., the reduced execution history $red_{\mathcal{H}}$ (cf. Section 3.1) can be produced on S^* as well. The marking M^{S^*} resulting from this is considered as a correct state.*

According to Axiom 2 type change Δ_T may be propagated to a biased instance I if Δ_T can be correctly applied to the execution schema of I and does not conflict with its current marking. The resulting schema $S^* = S_I + \Delta_T$ must therefore satisfy the correctness properties of the used WF meta model (cf. Section 2). In addition, I must be compliant with S* according to Axiom 1. As an example take schema S = S(T,V) from Fig. 6. Assume that type change Δ_T^1 = [`addCtrlEdge(S,E,D)`] is applied to S. Then condition 1 of Axiom 2 is not satisfied since the resulting schema $S_I + \Delta_T^1$ contains a deadlock-causing cycle. Δ_T^1 must therefore be not propagated to I. As opposed to this, type change Δ_T^2 = [`addActivity(S,Y,{D},{E})`] may be propagated to I since the conditions defined by Axiom 2 are met. As a last example take Δ_T^3 = [`deleteDataLink(S,C,d,write)`, `deleteActivity(S,C)`]. It is quite evident that propagation of Δ_T^3 to I would result in an incorrect data flow schema since X (which was inserted by a previous instance change) would read data element d with undefined value.

4.2 Checking Correctness

The challenge is to efficiently verify the conditions from Axiom 2. A naive solution would be to first generate schema $S_I + \Delta_T$ and then to check whether

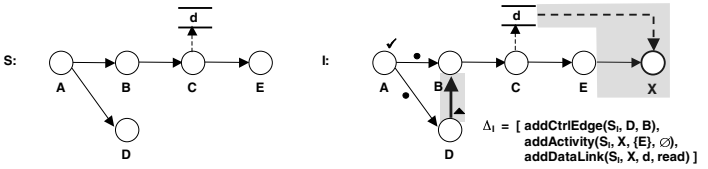


Fig. 6. Original Schema and Biased Instance

it satisfies the required structural and dynamic properties. Generally this would be too expensive, in particular if different WF aspects (control flow, data flow, etc.) are concerned or Δ_T shall be propagated to a large number of instances. Instead we must define appropriate rules for excluding conflicts between type and instance changes for as many cases as possible. Concerning the absence of deadlock-causing cycles, for example, the following conflict rule can be used:

Lemma 2 (Deadlock Prevention). *Let T be a WF type with actual schema version $S = S(T, V_n)$ and $I = (T, V_n, \Delta_I, ...)$ be a biased instance with execution schema $S_I = S + \Delta_I$. Assume that type change Δ_T transforms S into a correct schema $S' = S(T, V_{n+1})$. Then: $S^* = (S + \Delta_I) + \Delta_T$ does not contain deadlock-causing cycles if the following condition holds:*

$\forall\ s_1 \rightarrow d_1 \in AddedCtrlEdges_{\Delta_T}, \forall\ s_2 \rightarrow d_2 \in AddedCtrlEdges_{\Delta_I}$:

$d_1 \notin \texttt{pred}(S, s_2) \vee d_2 \notin \texttt{pred}(S, s_1)$

($AddedCtrlEdges_{\Delta}$ denotes the set of control edges inserted by change primitives addActivity and addCtrlEdge from Δ.)

Taking change Δ_T^1 from Section 4.1 the condition of this lemma is not satisfied. Concerning Δ_T^2, however, deadlocks can be excluded. Though the condition set out by Lemma 2 will not always be necessary, it is sufficient to exclude potential deadlocks. In particular, the related checks can be based on the original schema graph S and be accomplished by simple graph algorithms (with complexity O(n)). Generally, for each change operation we have to define corresponding conflict rules. Concerning data flow changes, for example, we can exclude potential conflicts by ensuring that the data element sets for which Δ_I and Δ_T have inserted or deleted data edges are disjoint. In our example from Section 4.1, Δ_I has inserted a read data link with source d and Δ_T^3 removed a write data link with target d. Thus a potential conflict exists, which requires additional checks.

4.3 Propagating Type Changes to Biased Instances

Assume that schema S is correctly transformed into a new schema S' by applying type change Δ_T to it. Assume further that $I = (T, V_n, \Delta_I, ...)$ is an instance to which Δ_T can be correctly propagated according to Axiom 2. Then the question arises how we can migrate this biased instance to the new schema version of type T. Due to lack of space we only consider changes based on the primitives from Table 1. For them the following theorem applies:

Theorem 3 (Commutativity of Type and Instance Changes). *Let T be a WF type with actual schema graph version S = S(T, V_n) and I = (T, V_n, Δ_I, ...) be a biased instance (with type T). Assume that type change Δ_T transforms S into S' = S(T,V_{n+1}) and Δ_T can be correctly propagated to I (according to Axiom 2). Then: Δ_T and Δ_I are commutative, i.e., Δ_I can be correctly applied to S' as well and (S + Δ_I) + Δ_T ≡ (S + Δ_T) + Δ_I (= S' + Δ_I).*

According to this theorem, type and instance changes are commutative provided that the specified conditions are met. In particular, this property allows us to treat type changes and related change propagation similar to the unbiased case (cf. Section 3.4). More precisely, a type change can be propagated to a biased instance I by re-linking this instance to the new schema S' (cf. Fig. 7) and by re-calculating marking $M^{S'_I}$ for the resulting execution schema S_I'. Note that S_I' can be simply derived by applying bias Δ_I to S'. Though at first glance it seems to make no significant difference whether we apply Δ_T to S_I or Δ_I to S' the latter variant offers several advantages with respect to the management of schema versions and propagation of future type changes. Due to lack of space we abstain from further details.

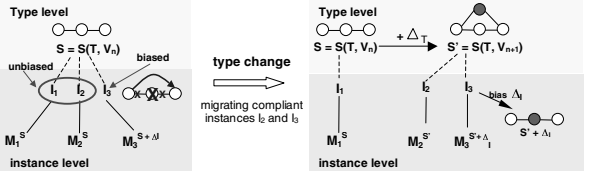


Fig. 7. WF Type Change and Propagation To Biased and Unbiased WF Instances

5 Related Work

One of the first approaches which has used a generic correctness criterion for dynamic WF changes was developed within the WIDE project [10]. WIDE applies a history-based compliance criterion to guarantee correctness when migrating instances to a modified schema. Concerning loops, however, this criterion is too restrictive. Furthermore, issues related to data flow changes, marking adaptations and conflicting changes are not considered. Similar correctness criteria have been applied in TRAMs and BREEZE. TRAMs [7] focuses on schema versioning concepts. To efficiently manage instance migrations the definition of migration conditions is proposed for every change operation. Based on them, it can be decided whether an instance can migrate to the new version or not. BREEZE [3] also uses formal compliance criteria but focuses on the handling of non-compliant WF instances. Furthermore it deals with the correct treatment of temporal constraints at the presence of dynamic changes.

Petri-Net based approaches for adaptive WF [9,11,13] must deal with the problem that actual state tokens of a net instance do not represent a view on previous instance execution as in our work. This, in turn, complicates compliance checking and marking adaptations in order to avoid the "dynamic change bug" [9]. [11] suggests the construction of a hybrid WF schema which reflects parts of both the old and the new WF schema. Additionally, the designer must explicitly specify rules for mapping tokens between old and new net versions. Actual results from the Petri Net field come from [9,14]. As correctness criterion *branching bisimularity* is proposed: An instance I can migrate to a modified schema if each action of I can be simulated on this schema as well. Unfortunately, branching bisimularity can be only ensured for special change operations and excludes, for example, order-changing operations. Apart from this, other issues addressed by our work (e.g., efficient compliance checks, treatment of loops, conflicting type and instance changes) have not been discussed in these papers.

In MOKASSIN [8] change primitives are encapsulated within WF instances. The compliance criterion is considered as being too restrictive. Instead, a more granular version concept is proposed. However, correctness issues are completely factored out by the authors. Another versioning approach has been offered by WASA_2 [4], which uses a correctness criterion based on the mapping of WF instances against WF schemes. In WASA_2, biased instances cannot be adapted to type changes anymore as opposed to our approach. Furthermore, changes in conjunction with loops have not been dealt with. Finally, ULTRAFlow [15] presents a rule-based approach. Changes are realized by modifying the implementation and meta data of activities. Special synchronization methods guaranteeing consistent access of instances on modified specifications are provided. However, ULTRAFlow totally factors out important change operations (e.g., deletion of activities) and does not consider data flow issues.

A formal treatment and comparison of correctness criteria of some of the above approaches has been given in [16].

6 Summary and Outlook

In many domains, like hospitals, engineering environments, or offices, process-centered applications will not be accepted whenever rigidity comes with them. Creating WF-based applications without a vision for adaptive WF is therefore shortsighted and expensive. Indeed, insufficient flexibility and adaptability have been primary reasons why WF technology failed in many process automation projects in the past. Both, the capability to quickly and correctly propagate WF type changes to in-progress WF instances and the support of ad-hoc adaptations will be key ingredients in the next generation of WfMS, ultimately resulting in highly adaptive process-oriented applications.

In this paper we have focused on the common support of WF type and WF instance changes and have discussed limitations of current approaches. The very important aspect of our work is its formal foundation. We have given general axioms and theorems which are fundamental for the correct handling of dynamic

WF changes. The treatment of conflicting type and instance changes adds to the completeness of our approach. Finally, there is already a powerful proof-of-concept prototype which demonstrates the feasibility of the presented concepts.

There are many other challenging issues related to adaptive workflow which must be better understood before we obtain a complete solution. First of all, we believe that usability issues constitute a field that would benefit by more intense study of the research community. Additionally, dynamic changes may also concern other components of the process-centered information system, like the organizational database, security constraints, temporal constraints [3], actor and resource assignments, or activity programs. Finally, we consider it as very important to incorporate more semantics into the dynamic change process [12].

References

1. van der Aalst, W., van Hee, K.: Workflow Management. MIT Press (2002)
2. Reichert, M., Dadam, P.: ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. Int'l J Intelligent Information Systems **10** (1998) 93–129
3. Sadiq, S., Marjanovic, O., Orlowska, M.: Managing change and time in dynamic workflow processes. Int'l J Cooperative Information Systems **9** (2000)
4. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: Proc. HICSS-34, Maui, Hawaii (2001)
5. Edmond, D., ter Hofstede, A.: A reflective infrastructure for workflow adaptability. Data and Knowlege Engineering **34** (2000) 271–304
6. 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
7. Kradolfer, M., Geppert, A.: Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In: Proc. CoopIS'99, Edinburgh (1999)
8. Joeris, G., Herzog, O.: Managing evolving workflow specifications. In: Proc. CoopIS '98, New York (1998) 310–321
9. van der Aalst, W.: Exterminating the dynamic change bug: A concrete approach to support worfklow change. Information Systems Frontiers **3** (2001) 297–317
10. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data and Knowledge Engineering **24** (1998) 211–238
11. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proc. Int'l Conf. on Org. Comp. Sys., Milpitas, CA (1995) 10–21
12. Rinderle, S., Reichert, M., Dadam, P.: On dealing with semantically conflicting business process changes. Technical Report UIB-2003-04, University of Ulm (2003)
13. Agostini, A., De Michelis, G.: Improving flexibility of workflow management systems. In: Proc. Int'l Conf. BPM'00, LNCS 1806. (2000) 218–234
14. van der Aalst, W., Basten, T.: Inheritance of workflows: An approach to tackling problems related to change. Theoretical Computer Science **270** (2002) 125–203
15. Fent, A., Reiter, H., Freitag, B.: Design for change: Evolving workflow specifications in ULTRAflow. In: Proc. CAISE '02. (2002) 516–534
16. Rinderle, S., Reichert, M., Dadam, P.: Evaluation of correctness criteria for dynamic workflow changes. In: Proc. Int'l Conf. BPM'03. (2003) 41–57