A Formal Framework For Workflow Type And Instance Changes Under Correctness Constraints

Manfred Reichert, Stefanie Rinderle, Peter Dadam
University of Ulm
Faculty of Computer Science
Dept. Databases and Informationsystems D-89069 Ulm
E-Mail:{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. While changes of single instances often have to be applied in an ad-hoc manner, type changes become necessary to adapt to evolving business processes. Especially for long-running processes it is indispensable to propagate type changes to running 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 comprehensive framework for the support of both, WF type and WF instance changes. We establish general correctness principles and show how WF instances can be automatically and efficiently migrated to a modified WF schema. We point out that our approach exceeds existing adaptation models in formal foundation, completeness, and usability.

1 Introduction

Workflow management systems (WfMS) deliver a state-aware control service for process-centered applications by routing, activating, and tracking activities of workflow (WF) instances according to their pre-defined WF schema [6, 11]. Examples include the processing of customer orders, patient treatment, and trip planning. For each WF type to be supported, a corresponding WF schema has to be specified. It defines the WF activities, associates them with application components, sets out the order and conditions for their execution, and defines the data flow between them.

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 process-centered information systems. Either deviations from the modeled WF schema are completely prohibited, thus requiring to bypass the WfMS in exceptional situations, or they may cause severe inconsistencies or errors.

To be really applicable, users must be able to apply ad-hoc changes to WF instances whenever required, e.g., by dynamically inserting or deleting activities [12, 14]. Such changes must not affect robustness and stability and have to be properly integrated with respect to authorization and documentation.

The ad-hoc adaptation of single instances [9, 12, 18, 14] is only one special kind of dynamic WF change. In order to be able to react rapidly to changed business needs and to adequately support evolutionary changes, it is important that adaptations can be quickly performed at the WF type level as well [1, 10, 14, 8]. In principle, a WF type change can be accomplished by modifying the corresponding WF schema accordingly. In doing so, however, in-progress WF instances must not get into trouble due to the change. This can be achieved, for example, by finishing them according to the old WF schema whereas future WF instances are created from the new one. This simple approach, however, is only sufficient for processes with short duration, but raises problems in conjunction with long-running workflows. In this case, WF instances with same WF type but different WF schema versions may co-exist for a long time, which may lead to confusion in the processing of business cases or be even contradictory to legal regulations or business rules.

In many cases it is highly desirable to propagate a WF 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. Such "biased" instances must not be needlessly excluded from change propagation as it is the case in current approaches [1, 10, 14, 18]. To make this point clear, take a patient treatment process as an example. Even though physicians may deviate from the pre-modeled WF schema at the instance level, this must not prohibit the propagation of future type changes to these instances, on condition that they are not conflicting with current instance state and previously applied ad-hoc changes.

The objective of this paper is to develop a framework for both, WF type changes and their propagation to related WF instances (evolutionary changes) and ad-hoc changes of single WF instances. Current adaptive WF models are either too restrictive or incomplete (for a detailed discussion see Section 5), focussing only on a special kind of change [2, 16, 1, 10, 12]. In addition, many approaches ignore usability and realization issues at all [13]. We present different change scenarios and have a look at fundamental principles concerning adaptive WF models. For this, we establish general criteria for arguing about the correctness of dynamic changes. Taking a simple, but powerful WF meta model, we exemplarily show for this meta model how correctness can be efficiently checked and which information is needed for this. In addition, we introduce well-defined rules and procedures for migrating single instances or collections of instances to a modified schema. Note that this does not only require schema transformations, but also adaptations of instance states and worklist structures.

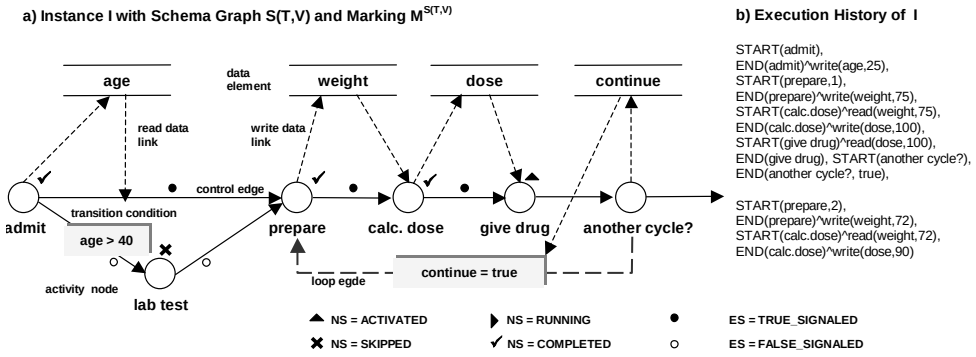

Figure 1: WF Instance Example

Section 2 sketches the WF meta model, which we use in this paper to illustrate fundamental principles of our approach. Since there is no generally accepted WF meta model, we orientate ourselves on commonly used modeling concepts. However, it is important to mention that the presented approach is applicable to several other WF description formalisms as well. Section 3 develops our framework for dynamic WF changes, focussing on general correctness principles as well as on implementable rules for ensuring correctess when a change is applied. Section 4 deals with conflicting changes at the type and instance level, and it shows under which conditions type changes may be propagated to biased WF instances as well. We discuss related work in Section 5 and conclude with a short summary in Section 6.

2 Workflow Modeling Basics

For each business process to be supported a corresponding *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.[1] However, in-progress WF instances based on older WF schema versions may exist as well.

2.1 Modeling and Execution of Workflows

Informally, a WF schema comprises a set of activities and defines the control and data flow between them. The control flow is modeled by linking activities with control edges, which may optionally be 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 transition conditions, sequences, parallel branchings,

[1] This is not really required. However, in the present paper we put the focus on correctness issues and do not deal with versioning issues in more detail.

and conditional branchings can be described.[2] 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, the 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:* $n_1 \neq m_1$

Remark: The sets pred(S, n)/succ(S, n) comprise all direct and indirect predecessors / successors of activity n via control edges (transitive closure), and L_{body} *comprises the nodes of a loop body with* $L_{body}(n_1, n_2)$*:=* *succ(S,* n_2*)* $\cap$ *pred(S,* n_1*)* $\cup \{n_1, n_2\}$.

Concerning the execution of a single activity, its status is initially set to `NOT_ACTIVATED`. When all pre-conditions are met (see below), activity status changes to `ACTIVATED`. Depending on the kind of activity, it is then automatically started or corresponding work items are inserted into worklists. When starting execution, activity status changes to `RUNNING` and the associated

[2] Up to this point the described meta model is similar to the one used in MQSeries Workflow [11].

application component is invoked. Finally, at successful termination, activity status passes to `COMPLETED`.

The execution of a newly created WF instance always starts with those activities which have no incoming control edge.[3] When an actvity is completed, its outgoing control edges are either evaluated to `TRUE_SIGNALED` or `FALSE_SIGNALED`, depending on their transition conditions. This, in turn, leads to the re-evaluation of target activities. We assume that an activity may be activated as soon as all incoming control edges have been signaled and at least one of them is marked with `TRUE_SIGNALED`. Consequently, if all incoming control edges are marked as `FALSE_SIGNALED`, the activity cannot be executed anymore and its status is set to `SKIPPED`. This, in turn, may lead to the 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 be signalled, and all nodes contained within the loop body be reset in their state. Finally, the 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 flow state of I is captured by a marking function M^S = (NS, ES). It assigns to each activity n its current status NS(n) (see above) and to each control or 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). Thus M^S reflects a consolidated view of the previous execution of I. 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, $M^{S(T,V)}$, $Val^{S(T,V)}$, $\mathcal{A}$) where*

- *T denotes the type of I and V the version of the schema graph version 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).*
- *$\mathcal{A} = < e_0, \ldots, e_k >$ 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.*

[3] Due to the absence of cycles (except via loop backward edges), at least one such activity exists.

As described, WF instances preserve their markings when proceeding in the flow of control. Thus, a WF instance marking provides a consolidated view on the previous execution of the respective workflow. As we will see later, this property is very useful in connection with dynamic WF changes. Formally:

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

2.2 Defining and Changing Schema Graphs

Table 1 contains some change primitives that can be used to define and modify WF schema graphs. Each primitive has a well-defined semantics and is associated with formal pre- and post-conditions, necessary to preserve the (structural) correctness of the respective WF schema (cf. Def. 1). In this paper, we restrict our considerations to the avoidance of deadlocks that may be caused due to "undesired" cyclic order relationships (via control edges). Generally, additional constraints exist, which must be considered as well. Concerning data flow, for example, it is required that no lost updates occur during runtime or that all data elements read by an activity will always be written by preceding activities, independently of the chosen execution branches. There are other changes primitives (e.g., to update edge conditions), which we do not further consider in this paper. Change primitives also 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. This, however, is outside the scope of this paper.

3 Dynamic Change Basics

In this section, we develop our framework for dynamic WF changes. For illustration purposes we restrict our considerations to schema 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 propagation of a WF type change). Instead we focus on fundamental issues related to dynamic instance changes.

In the following, let I be a WF instance with WF schema graph S and marking M^S. Assume that S is transformed into a correct WF schema graph S' by applying the change Δ. Then 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 instance I is compliant with S', how can we smootly *migrate* I to S' such that its further execution can be based on S' ? Which state/marking adaptations become necessary in this context?

Table 1: Examples of Basic Change Primitives

addCtrlEdge(S, n_{src}, n_{dst})	Pre: $(n_{src} \notin \text{succ(S, } n_{dst}) \cup \{n_{dst}\}) \wedge (\forall\ (\text{n}_1,\text{n}_2) \in \text{LoopEdges}:$ $[n_{src} \in \text{L}_{body}(\text{n}_1,\text{n}_2) \Leftrightarrow n_{src} \in \text{L}_{body}(\text{n}_1,\text{n}_2)])$ Post: CtrlEdges' = CtrlEdges $\cup \{n_{src} \rightarrow n_{dst}\}$
addActivity(S, n_{ins}, Preds, Succs)	Pre: $(\forall\ \text{p} \in \text{Preds}, \forall\ \text{s} \in \text{Succs}: \text{s} \notin \text{pred(S, p)}) \wedge$ $(\forall\ (\text{n}_1,\text{n}_2) \in \text{LoopEdges}:$ $(\text{Preds} \cup \text{Succs}) \subseteq \text{L}_{body}(\text{n}_1,\text{n}_2) \vee$ $(\text{Preds} \cup \text{Succs}) \cap \text{L}_{body}(\text{n}_1,\text{n}_2) = \emptyset$ Post: N' = N $\cup \{n_{ins}\}$ CtrlEdges' = CtrlEdges $\cup \{\text{p} \rightarrow n_{ins} \mid \text{p} \in \text{Preds}\}$ $\cup \{n_{ins} \rightarrow \text{s} \mid \text{s} \in \text{Succs}\}$
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 $\neg \{\text{a} \rightarrow \text{b} \mid n_{del} \in \{\text{a, b}\}\} \cup$ $\{\ \text{p} \rightarrow \text{s with EC(p} \rightarrow \text{s)} = \text{ec} \mid$ $\text{p} \rightarrow n_{del}, n_{del} \rightarrow \text{s} \in \text{CtrlEdges} \wedge \text{EC}(n_{del} \rightarrow \text{s}) = \text{ec}\}$

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 WF instance migrations.

In Section 3.1 we introduce general correctness principles for dynamic instance changes 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 WF instances with a modified WF schema (cf. Section 3.2). Section 3.3 shows how correct follow-up markings of compliant instances can be automatically determined when migrating them to the new schema. Section 3.4 concludes with a discussion of different change scenarios (evolutionary changes, ad-hoc changes) and their realization within the developed framework.

3.1 Dynamic Change Correctness

To illustrate potential problems that may result from the uncontrolled migration of WF instances, consider the WF 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. 2c). Assume that this change is to be applied to the WF instances shown in Fig. 2 b) (currently

based on WF schema S), but without performing additional compliance checks. Concerning I_1 no problem would occur, since its execution has not yet entered the change region. As opposed to this, uncontrolled migration of I_2 to the modified schema graph would cause severe problems. Firstly, an inconsistent marking would result (cf. Lemma 1), thus leading to an undefined execution behavior. 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', first of all, activation of activity `prepare` has to be undone and corresponding work items have to be removed from user worklists. In addition, the newly inserted activity `test` must be activated. This simple example demonstrates that the applicability of a dynamic WF change depends on the current instance state as well as on the change primitives applied. In addition, when migrating a WF instance to the modified WF schema, markings as well as related worklist structures have to be correctly adapted.

To migrate WF instances in a correct and reliable manner, appropriate rules are required. Comparable with serializability in DBMS, we need general principles which allow us to argue about correctness of dynamic WF changes. In more detail, we require a formal criterion for deciding whether a given WF instance can be smoothly migrated to the modified WF schema or not. In addition, we must be able to determine correct new markings resulting from such "on-the-fly" migrations. One of our design goals is to define these correctness criteria independently of the operational semantics of the underlying WF meta model and the semantics of 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 compliance checking and for migrating compliant WF instances.

Intuitively, an instance I is compliant with the modified WF schema graph S', if I could have been executed according to S' as well and would have produced the same effects on data elements [1, 14]. Trivially, this will be always the case, if I has not yet entered the graph region affected by the change. Generally, we need information about previous execution to decide this property and to determine correct follow-up markings in case of compliance. For deriving such a general compliance principle, at the logical level we make use of the execution history which is usually kept for each WF instance (cf. Fig. 1 and Fig. 2). We assume that this history logs events related to the start and termination of activity executions (cf. Def. 2).

Obviously, an instance I with execution history $\mathcal{A}$ will always be compliant with S' (and can therefore be migrated to S'), if $\mathcal{A}$ could have been produced on S' as well. We then obtain a correct follow-up marking by "replaying" all events from $\mathcal{A}$ on S' in sequential order. Taking our example from Fig. 2, this property holds for I_1 and I_3, but it does not apply to I_2. Furthermore, when replaying $\mathcal{A}_3$ on S' we obtain the node markings as sketched above.

The described criterion is still too restrictive to serve as a general correctness principle. Concerning changes of loop structures, it may needlessly exclude instances from migrations though this would not lead to problems. As an example take instance I from Fig. 1, where the depicted loop execution is in its 2^{nd} iteration. Assume that WF schema S is modified by applying operation `addActivity(S, perform test, {prepare}, {give drug})`. Taking the

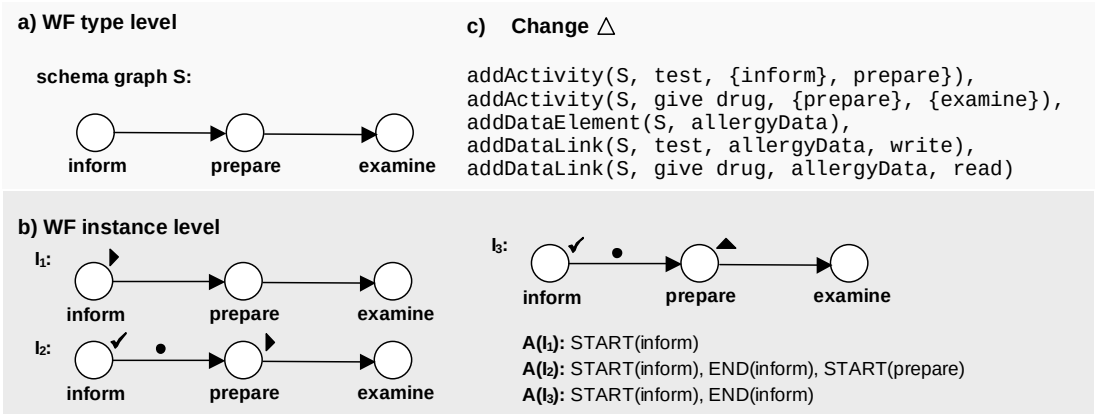

Figure 2: WF Schema Graph and Related WF Instances

above criterion this change would not be allowed since the previous loop iteration of I is not compliant with the new WF schema; i.e., history $\mathcal{A}$ cannot be produced based on the modified WF schema graph. Unfortunately, excluding such WF instances from migrations very often will be not in accordance with practice and will therefore be not accepted by users. To overcome this restrictiveness, we relax the above criterion by (logically) discarding those history entries, which were produced within another loop iteration than the last (completed loops) or the current one (running loops).[4] We denote this reduced view on the execution history $\mathcal{A}$ as $\text{red}_{\mathcal{A}}$. 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, $\mathcal{A}$) 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 $\text{red}_{\mathcal{A}}$ 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 $\text{red}_{\mathcal{A}}$ on S'.*

These two basic properties satisfy our main design goals, since they are independent of the operational semantics of the used WF meta model and the change operations applied. Axiom 1 can therefore serve as fundamental correctness principle for adaptive WfMS. Furthermore, it will not needlessly exclude WF instances from migrations, on condition that their further execution does not get into trouble due to the change.

[4] i.e., for each loop construct we omit unnecessary history entries. In doing so, we also consider nested loops.

Altogether, Axiom 1 provides a good basis for arguing about the correctness of dynamic WF changes and related change procedures. However, it would be certainly no good idea to guarantee compliance and to determine correct follow-up markings of WF instances by accessing the whole (reduced) execution history and by trying to "replay" its entries on the modified schema graph. Note that histories may comprise voluminous data such that they are usually not kept in primary storage. In particular, if a large number of instances is to be migrated to the new schema, this approach causes heavy performance losses. In the following, we present optimized rules and procedures which ensure correctness according to Axiom 1.

3.2 Rules for Checking Compliance

For the WF meta model from Section 2 and the related change primitives, we exemplarily show under which conditions compliance (cf. Axiom 1,1) can be guaranteed.[5] 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 the dynamic properties of the described WF model. In particular, derivation of compliance rules benefits from the used marking approach since activity markings already provide a consolidated view on the (reduced) execution history of a WF instance (cf. Lemma 1).

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 applied change primitives and by discarding unnecessary expressions (e.g., conditions on nodes not present in the original schema graph), we do not further consider high-level operations in this paper.

Let I be a WF instance with WF schema S, consistent marking M^S, and execution history $\mathcal{A}$. Assume that S is transformed into a correct WF schema S' by applying one of the change primitives from Table 1. The challenging question is, under which conditions we can ensure compliance of I with S' (cf. Axiom 1,1) and which information is needed for this. Table 2 summarizes well-founded compliance conditions for selected change primitives. Based on them we can state the following theorem:

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 $\mathcal{A}$. 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, $\mathcal{A}$) = `TRUE` *(cf. Table 2).*

[5] As mentioned, similar considerations can be applied to comparable WF meta models as well.

Table 2: Examples of Compliance Rules

Change Operation Δ ...	... and Related Compliance Condition *Compliant(S, Δ, NS, ES, Val^S, $\mathcal{A}$)*
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}) $\in \mathcal{A}$, $\Rightarrow i < j$)) $\vee$ (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}) $\in \mathcal{A}$, $\Rightarrow j < i$)) $\vee$ (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}) $\in \mathcal{A}$, $\Rightarrow j > i$) })) $\vee$ (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) $\in \mathcal{A}$, $\Rightarrow j > i$)) 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

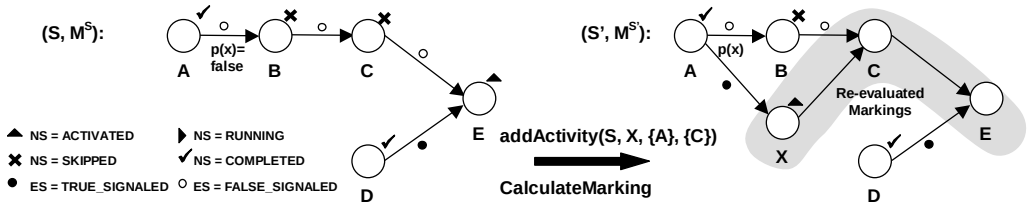

Figure 3: Insertion before a Skipped Node

In this paper, we omit formal proofs. Instead we exemplarily describe compliance conditions related to the primitives `addActivity` and `addCtrlEdge`.

Regarding the insertion of an activity n_{ins} between two node sets *Preds* and *Succs*, compliance can be always 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 a history entry into $\mathcal{A}$. Furthermore, compliance can be ensured, if all nodes from *Preds* are marked as `SKIPPED`. Then n_{ins} will be skipped as well, i.e., its insertion will have 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 for an example).

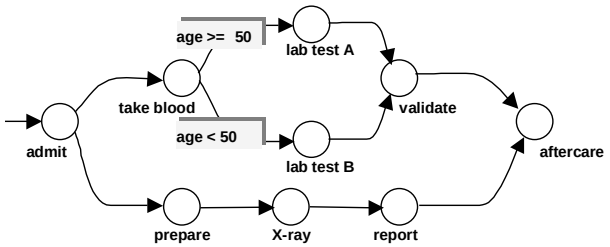

Figure 4: Patient Treatment Workflow

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 the dynamic insertion of a control edge $n_{src} \rightarrow n_{dst}$ will even be allowed, if n_{dst} has been already started or completed. As an example take the WF schema graph S from Fig. 4. Assume that an additional control edge is to be inserted between the activities `test B` and `X-ray`. Concerning WF instances, for which both activities are completed, this insertion would be (only) allowed if `test B` had written its end entry into $\mathcal{A}$ 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 activity `take blood` had been completed before activity `X-ray` started ($N_1 = \{$`take blood`$\}$). Note that transitive order relationships have to be taken into consideration as well when checking compliance.

Compliance conditions related to the deletion of activity nodes and control edges as well as to changes of the data flow schema are summarized in Table 2. In a similar way we can derive compliance rules for other changes primitives, e.g., the insertion or deletion of loop edges or the update of edge transition conditions. Altogether, we can state that compliance of WF instances with a WF schema graph – as postulated by Axiom 1,1 – can be checked on basis of current activity markings; i.e., we usually do not have to check the producibility of whole execution histories on the modified schema. Nevertheless, Axiom 1,1 serves as the formal basis for deriving and veryfying compliance rules.

3.3 How To Adapt WF Instance States?

We have described how the compliance property set out by Axiom 1,1 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 want to show how compliant WF instances can be migrated to the changed WF schema.

As pointed out, one problem to be solved is the efficient and correct adaptation of activity and edge markings. According to Axiom 1,2 the state of a migrated WF instance must be the

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, $\mathbf{n}_{src}$, $\mathbf{n}_{dst}$)	$N_{check}(op)$:= {n_{dst}}, $E_{check}(op)$:= {n_{src} → n_{dst}}
op = deleteCtrlEdge(S, $\mathbf{n}_{src}$, $\mathbf{n}_{dst}$)	$N_{check}(op)$ = {n_{dst}}

same as it can be obtained when replaying the (reduced) execution history on the new schema. How extensive marking adaptations turn out for a WF instance I depends on the kind and the scope of the change. Except initialization of newly inserted nodes and edges, no adaptations will become necessary if the execution of I has not yet entered the change region. In other cases extensive state adaptations may be required. An activity `X`, for example, may have to be deactivated if new 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 are already in a final state. As shown in Fig. 3 it may even become necessary to undo the skipping of nodes when inserting an activity.

We now describe how such state markings can be automatically and efficiently adapted when migrating compliant WF instances. Our approach makes use of the semantics of the applied change operations as well as of the execution properties of the WF meta model presented in Section 2. 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 calculates consistent follow-up markings for compliant instances.

Table 3 shows the initial sets of nodes and edges whose markings must be evaluated when the respective change operation is applied – for operation *op* we denote these sets as $N_{check}(op)$ and $E_{check}(op)$ respectively. Depending on the result of the evaluation, the 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 `NOT_ACTIVATED`. (Note that an initial evaluation of n_{ins} only becomes necessary if $Preds = \emptyset$ holds.) Secondly, all successors of n_{ins} must be reevaluated 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

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` since for this case n_{dst} may have to be deactivated.

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) $\cup$ $E_{check}(op_i)$; $N_{check}(\Delta)$:= N_{check} (Δ) $\cup$ $N_{check}(op_i)$;
done
$E_{check}(\Delta)$:= $E_{check}(\Delta)$ $\cap$ E'; $N_{check}(\Delta)$:= $N_{check}(\Delta)$ $\cap$ N';

Concerning a complex change $\Delta = op_1, \ldots, op_n$ the total sets $N_{check}(\Delta)$ and $E_{check}(\Delta)$ of nodes and edges to be (initially) evaluated can be determined by the use of Algorithm 1. In principle, we obtain these sets 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 which are not present in the resulting WF schema graph anymore. This is considered by Algorithm 1.

Let I be a WF instance with WF schema S and marking M^S. Assume that S is transformed into a correct WF schema S' by applying change $\Delta = op_1, \ldots, op_n$. Assume further that I is compliant with S'. Algorithm 2 then determines the correct follow-up marking $M^{S'}$ of instance I when migrating it to S'. Basic to this are the marking and execution rules of our WF meta model. Algorithm 2 starts with the sets $N_{check}(\Delta)$ and $E_{check}(\Delta)$ as input. If the markings of respective nodes or edges are adapted during the execution of Algorithm 2, context nodes and edges will be re-evaluated as well, etc. By means of Algorithm 1 and 2, the total expenditure for state adaptations can be significantly reduced when compared to the complete re-evaluation of all node and edge markings or the complete replay of all history events on the new WF schema. Nevertheless, our marking procedure guarantees correctness according to Axiom 1,2. Formally:

Theorem 2 (Optimized Marking Adaptations) *Let I = (T, V, M^S, Val^S, $\mathcal{A}$) 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 that 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 $\mathcal{A}$ on S'.*

While Algorithm 1 has to be carried out only once at change definition time, Algorithm 2 must be applied for each WF 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 WF instance is compliant with the modified WF schema. With Algorithm 1 we obtain N_{check} = {C} and E_{check} = {A $\rightarrow$ X}. Furthermore, when running Algorithm 2 with these sets as input, the newly inserted activity X will be activated whereas the skipping of C as well as the 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 sequentially so far) and its propagation to compliant WF instances is depicted in Fig. 5. Note that both, necessary compliance checks

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 condition 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 = ∅;
```

and marking adaptations are automatically performed in our approach. Moreover, access to the (complete) execution history is avoided as far as possible. – The "dynamic change bug" as discussed in the WF literature (e.g. [2, 15]) is therefore not present in our framework.

3.4 Realizing Workflow Schema Evolution and Ad-hoc Changes

The presented correctness principles, compliance rules, and migration procedures are independent from whether a single WF instance or a collection of WF instances has to be adapted. Ad-hoc changes of single WF instance may become necessary, for example, to evolve the structure of a particular business case during runtime or to deal with an exceptional situation. As opposed to this, a large number of WF instances may have to be adapted, if a WF type change shall be propagated to related WF instances.

WF Schema Evolution and Change Propagation: First of all, our framework allows the designer to restrict the set of migratable WF instances by specifying appropriate selection predicates (based on WF attributes). Then for each selected WF instance I, the WfMS checks whether it is compliant with the modified schema version or not. If the former is the case, I is re-linked to the new schema S' and its further execution is based on S' (cf. Fig. 6 b). Among other things this includes the adaptation of markings and related data structures (e.g., worklists) as described above. As opposed to this, non-compliant WF instances may be finished according to the old WF schema version or be rolled back to a compliant state to enable their migration.

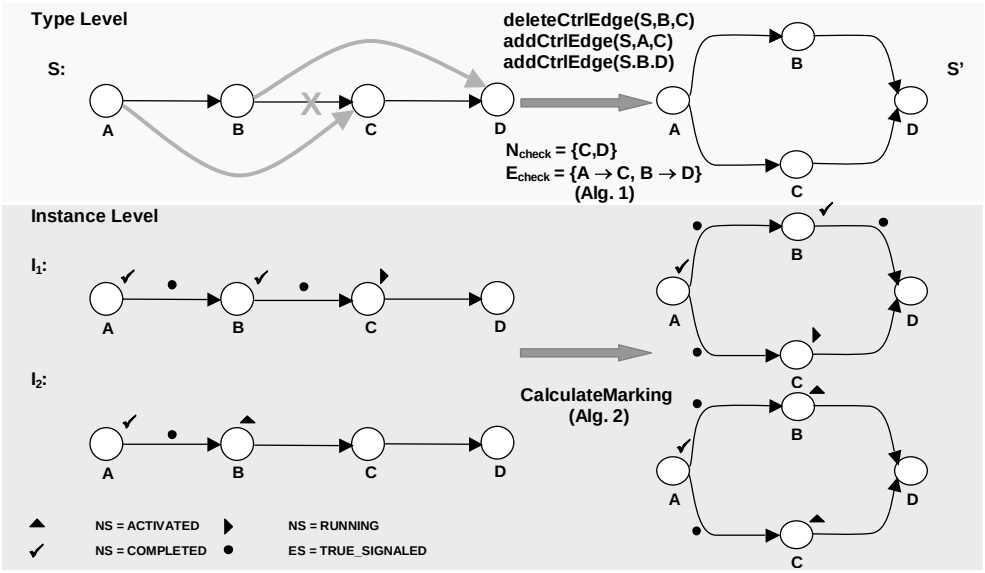

Figure 5: Instance Migrations Due To Type Change

In connection with loops, such a compliant state may be also reached when a loop enters its next iteration. A discussion of this special case and the support of delayed WF instance migrations, however, is outside the scope of this paper.

(Ad-hoc Changes of Single WF Instances: An ad-hoc change of a WF 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 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 WF schema graph 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, ..., 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. 6 a). How biased WF instances are "physically" represented, whether S_I is materialized or only Δ_I is stored in the WfMS database, and other implementation issues are outside the scope of this paper. In any case, a biased WF instance always keeps the reference to its original WF schema. As we will see in the next section, under certain conditions this allows us to propagate WF type changes to biased WF instances as well.

Definition 3 (Biased Instance) *A biased instance I is described by a tuple (T, V, Δ_I, $M^{S+\Delta_I}$, $Val^{S+\Delta_I}$, $\mathcal{A}$), 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, ..., 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.*

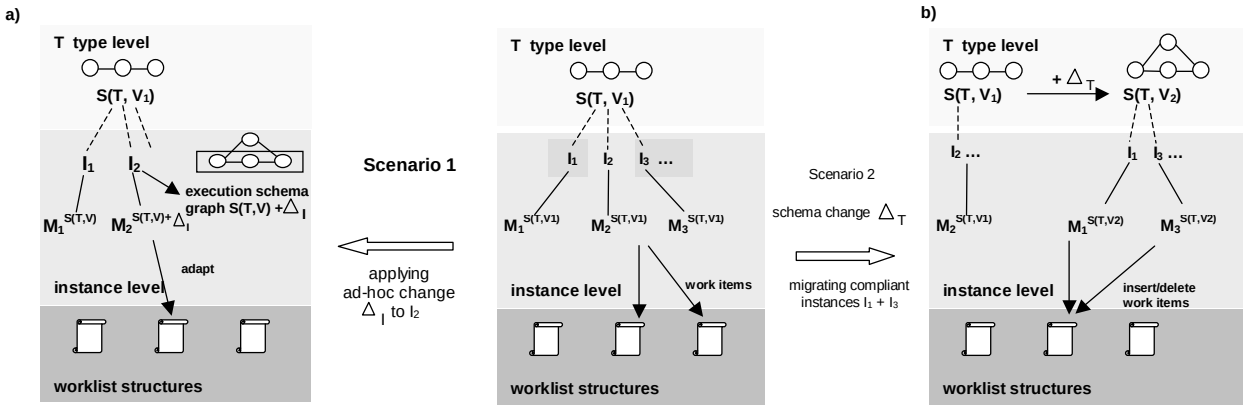

Figure 6: Managing Type and Instance Changes

Trivially, the execution schema S_I of an unbiased instance I (with $\Delta_I = \emptyset$) corresponds to its original schema S.

4 Conflicting Type and Instance Changes

As motivated in Section 1, biased WF instances must not be needlessly excluded from adapting to a WF type change. Generally, for each WF type T it must be possible to propagate a related schema change Δ_T to biased WF instances (with type T) as well. Otherwise, the WfMS will not have the required flexibility to adequately support long-running processes. 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 (with WF type T) which was created from WF 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 further that a new WF schema version S' = S(T, V_{n+1}) is derived from S by applying type change Δ_T (= $op_T^1, \ldots, op_T^m$) to it (S' = S + Δ_T). Then the following issues arise:

1. May Δ_T be propagated to I as well though the current execution schema S_I= S + Δ_I of I differs from 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" WF instances (cf. Section 3) we introduce a general criterion that allows us to argue about these two issues. Obviously, when propagating a WF type change Δ_T to a biased WF instance I, we must not only consider its current state (i.e., its

marking M^{S_I}) but we also have to cope with structural and semantic conflicts that may exist between the concurrent changes Δ_I and Δ_T (Note that both, Δ_I and Δ_T have been based on S). In the following, we put the emphasis on structural conflicts.

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 WF 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, Δ_I, ...) :⇔

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

According to Axiom 2, a WF type change Δ_T may be propagated to a biased instance I if Δ_T is applicable to the execution schema of I as well and if the change does not conflict with the the previous execution of I. That means, the resulting WF schema S* = S_I + Δ_T must satisfy the correctness properties set out by the WF meta model from Section 2. In addition, I must be compliant with S* according to Axiom 1.

As an example take the WF schema S = S(T,V) from Fig. 7. Assume that WF type change Δ_T^1 = addCtrlEdge(S, E, D) is applied to S. Then condition 1 of Axiom 2 is not satisfied with respect to I since the resulting WF schema graph S_I + Δ_T^1 would contain a cycle leading to a deadlock. Due to this unresolvable conflict between WF instance and WF type change, Δ_T^1 cannot be propagated to I. As opposed to this, WF type change Δ_T^2 = [addActivity(S, Y, {D}, {E})], for example, may be propagated to I since the conditions defined by Axiom 2 are met. As a last example, take change 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 for instance I, since X (which was inserted by a previous instance change) would then read data element *d* with undefined value.

4.2 Checking Correctness

The challenging question is how to efficiently verify the conditions set out by Axiom 2. A naive solution would be to first generate the WF schema graph S_I + Δ_T and then to check whether it satisfies the required structural and dynamic properties. Generally this would be too expensive, in particular if different WF aspects (control flow, data flow, work assignements, etc.) are concerned or Δ_T is to be propagated to a large collection of WF instances. Instead we must define appropriate and efficient rules for excluding potential conflicts between WF instance and WF type changes for as many WF instances as possible. Concerning the absence of cycles and deadlocks, for example, the following conflict rule can be used:

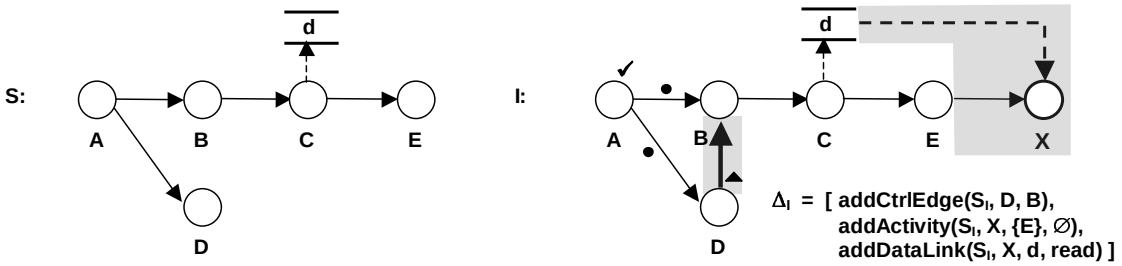

Figure 7: Original Schema and Biased Instance

Lemma 2 (Deadlock Prevention) *Let T be a WF type with actual schema version S = S(T, V_n) and I = (T, V_n, Δ_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 + Δ_I) + Δ_T does not contain cycles (except loop backs) and is therefore deadlock-free if the following condition holds:*

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

$d_1 \notin$ `pred`*(S,* s_2*)* $\vee$ $d_2 \notin$ `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, for example, the condition defined by this lemma will not be 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 WF schema graph S and can 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 (more expensive) checks.

4.3 Propagating WF Type Changes to Biased Instances

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

Theorem 3 (Commutativity of WF Type and WF 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) + $\Delta_T \equiv$ (S + Δ_T) + Δ_I (= S' + Δ_I).

According to this theorem, WF type and WF instance changes are commutative provided that the specified conditions are met. In particular, this property allows us to treat WF type changes and the related change propagation similar to the unbiased case (cf. Section 3.4). More precisely, a WF type change can be propagated to a biased WF instance I by re-linking this instance to the new WF schema graph S' (cf. Fig. 8) and by re-calculating marking $M^{S'_I}$ for the resulting WF 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 adavantages with respect to the management of WF schema graph versions and the propagation of future type changes. – In summary, the presented concepts offer an important contribution for the design of adaptive WfMS.

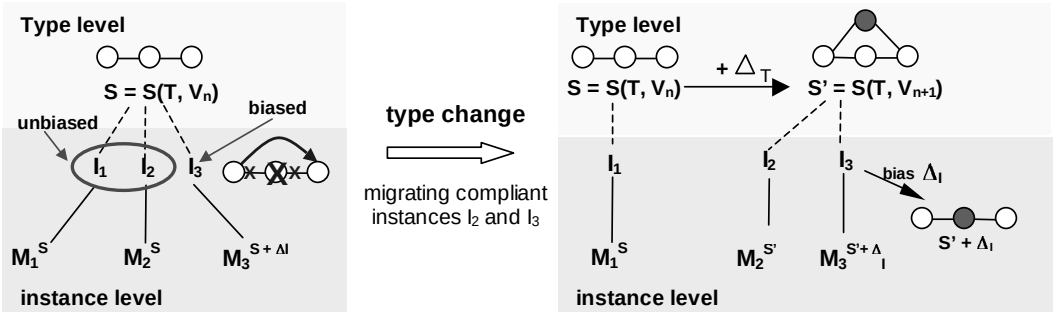

Figure 8: WF Type Change and Propagation To Biased and Unbiased WF Instances

5 Related Work

One of the first frameworks for WF type changes has been presented in [2, 3]. For WF modeling and WF execution high-level Petri Nets are used. The execution of WF instances with the same WF type is based on the same net and on coloured markings. A WF schema modification is carried out by substituting sub nets. Another Petri Net-based approach is presented in [15, 16]. Dynamic change correctness is ensured by prohibiting WF instances from migrating to the modified net if they have already reached modification hit regions. Both approaches consider the adaptation of markings as a very complex problem (the so called *dynamic change bug*). To solve it, [3] suggests that the WF designer herself or himself has to adapt the marking for every instance. As opposed to this, [16] proposes the definition of a function, which maps markings of the old to markings of the new net. However, only special change operations are

considered by this approach. Apart from this, many of the issues discussed in this paper (e.g., efficient compliance checks, efficient adaptation of WF instance markings, etc.) have not been addressed.

Several approaches use graph-based meta models to cope with WF schema evolution [1, 10, 14]. WIDE [1] offers a history-based compliance criterion to guarantee correctness when migrating WF instances to the new WF schema. TRAM [10] focuses on WF schema versioning concepts. To efficiently manage WF instance migrations the definition of so-called migration conditions is proposed for every change operation. Based on them, it can be decided whether a WF instance can migrate to the new WF schema version or not. BREEZE [14] also uses a compliance criterion but focuses on the handling of non-compliant WF instances. In summary, all these approaches are too restrictive in conjunction with loops. Neither a special treatment of data flows nor suggestions about how to adapt markings when propagating WF schema changes are offered. Finally, all approaches focus on WF schema evolution and do not consider ad-hoc changes.

Object oriented approaches are offered by [7, 17]. In MOKASSIN [7, 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, but without discussing issues related to compliance checks. Apart from this, correctness issues are completely ignored. Another versioning concept is offered by $WASA_2$ [17, 18], which proposes a mapping between the modified WF schema and the subworkflows resulting from the corresponding WF instances to allow efficient compliance checks. However, data flow changes or type changes in conjunction with loops have not been addressed in detail.

ULTRAFlow [4] presents a rule-based approach. Changes are mainly realised by modifying the implementation and meta data of activities. Special synchronisation methods guaranteeing consistent access of WF instances on modified specifications are provided. However, on ULTRAFlow, important change operations like the deletion of activities or the modification of data flows are not treated at all.

Finally, in [5] a statechart-based approach focusing on WF schema modifications, is presented, which enables semantic preserving schema changes.

6 Summary and Outlook

Long regarded as technology for the automation of well-structured, repetitive processes, showing only little variations in their possible excecution sequences, WF management is in the throes of transformation as more and more non-traditional applications require comprehensive process support. In many domains, like hospitals, engineering environments, bioinformatics, or offices, however, process-centered applications will not be accepted whenever rigidity comes with them. Creating WF-based applications without a vision for adaptive workflow is therefore shortsighted and expensive. Indeed, insufficient flexibility and adaptability have been primary reasons why

workflow 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 as well as the flexible support of ad-hoc adaptations will be key ingredients in the next generation of WfMS, ultimatively resulting in highly adaptive process-oriented applications.

We have implemented the fundamentals concerning dynamic WF changes in a proof-of-concept prototype, which is even based on a more powerful WF meta model than the one assumed in this paper. Currently, the incorporation of change propagation facilities is on its way. There are many other challenging issues related to adaptive workflow, which have to be better understood, before we come to a complete solution. Dynamic adaptations, for example, may also concern other components of process-centered information systems, like the organizational database [6], security constraints, actor and resource assignments, activity programs, or temporal constraints [14]. Apart from this, we consider it as very important to incorporate more semantics when checking compliance of WF instances with a modified schema.

References

[1] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data and Knowledge Engineering*, 24(3):211–238, 1998.

[2] C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proc. Int'l Conf. on Org. Comp. Sys.*, pages 10–21, Milpitas, CA, August 1995.

[3] C.A. Ellis and C. Maltzahn. The Chautauqua workflow system. In *Proc. 30th Int'l Conf. on System Science*, Maui, 1997.

[4] A. Fent, H. Reiter, and B. Freitag. Design for change: Evolving workflow specifications in ULTRAflow. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAISE '02)*, pages 516–534, May 2002.

[5] H. Frank and J. Eder. Equivalence transformations on statecharts. In *Proc. 12th Int'l Conf. on Software and Knowledge Eng.*, pages 150–158, Chicago, July 2000.

[6] S. Jablonski and C. Bussler. *Workflow Management: Concepts, Architecture and Implementation*. Int. Thompson Publ., 1995.

[7] G. Joeris. Defining flexible workflow execution behaviors. In *Proc. GI-Workshop, Enterprise-wide and Cross-enterprise Workflow-Management (Informatik '99)*, pages 49–55, October 1999.

[8] G. Joeris and O. Herzog. Managing evolving workflow specifications. In *Proc. Int'l Conf. on Coop. Inf. Syst. (CoopIS '98)*, pages 310–321, New York, August 1998.

[9] K. Kochut, J. Arnold, A. Sheth, J. Miller, E. Kraemer, B. Arpinar, and J. Cardoso. IntelliGEN: A distributed workflow system for discovering protein-protein interactions. *Distributed and Parallel Databases*, 13:43–72, 2003.

[10] M. Kradolfer and A. Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Proc. CoopIS '99*, pages 104–114, Edinburgh, September 1999.

[11] F. Leymann and D. Roller. *Production Workflow.* Prentice Hall, 2000.

[12] M. Reichert and P. Dadam. ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. *Journal of Intelligent Inf. Syst.*, 10(2):93–129, 1998.

[13] S. Rinderle, M. Reichert, and P. Dadam. Evaluation of correctness criteria for dynamic workflow changes. In *Proc. Int'l Conf. on Business Process Management (BPM '03)*, Eindhoven, The Netherlands, June 2003.

[14] S. Sadiq, O. Marjanovic, and M. Orlowska. Managing change and time in dynamic workflow processes. *The Int'l Journal of Coop. Inf. Syst.*, 9(1&2), 2000.

[15] W.M.P. van der Aalst. Exterminating the dynamic change bug: A concrete approach to support worfklow change. *Information Systems Frontiers*, 3(3):297–317, 2001.

[16] W.M.P van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.

[17] M. Weske. Flexible modeling and execution of workflow activities. In *Proc. 31st Int'l Conf. on System Sciences*, pages 713–722, Hawaii, 1998.

[18] M. Weske. Adaptive workflows based on flexible assignment of workflow schemes and workflow instances. In *Proc. GI-Workshop Enterprise-wide and Cross-enterprise Workflow-Management (Informatik '99)*, pages 42–48, Paderborn, October 1999.