Distributed and Parallel Databases, 16, 91–116, 2004

Flexible Support of Team Processes by Adaptive Workflow Systems*

STEFANIE RINDERLE rinderle@informatik.uni-ulm.de
MANFRED REICHERT reichert@informatik.uni-ulm.de
PETER DADAM dadam@informatik.uni-ulm.de
University of Ulm, Computer Science Faculty, Dept. Databases and Information Systems, 89069 Ulm, Germany

Recommended by: Ahmed Elmagarmid

Abstract. Process-oriented support of collaborative work is an important challenge today. At first glance, *Workflow Management Systems* (*WfMS*) seem to be very suitable tools for realizing team-work processes. However, such processes have to be frequently adapted, e.g., due to process optimizations or when process goals change. Unfortunately, runtime adaptability still seems to be an unsolvable problem for almost all existing WfMS. Usually, process changes can be accomplished by modifying a corresponding (graphical) workflow (WF) schema. Especially for long-running processes, however, it is extremely important that such changes can be propagated to already running WF instances as well, but without causing inconsistencies and errors. The paper presents a general and comprehensive correctness criterion for ensuring compliance of in-progress WF instances with a modified WF schema. For different kinds of WF schema changes, it is precisely stated, which rules and which information are needed at mininum for satisfying this criterion.

Keywords: workflow management, adaptive systems, schema evolution, compliance checks

1. Introduction

Computer supported team work has become more and more important during the last years since humans and machines can share their power and spirit. The various software systems to support collaborative work can be summarized as *Computer Supported Cooperative Work* (*CSCW*) systems. CSCW systems can be classified, for example, according to the degree of distribution of *time* and *place* the team members work at (cf. Table 1).

One of the most powerful technologies within this classification framework is offered by *Workflow Management Systems* (*WfMS*). WfMS support team members working on a complex task at distributed places and at different points in time. Furthermore, they offer a promising technology for process-oriented coordination of (distributed) team work [13], i.e., they allow to organize team work in a process-oriented manner and across organizational boundaries. For each workflow (WF) type to be supported (e.g., concerning the treatment of patients in a hospital or the design of a sales promotion), a corresponding *WF schema S* has to be defined. It comprises a set of activities with associated application components and

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

Table 1. Classification of CSCW [24].

	Time of interaction	
Presence of team members	*Same time*	*Different time*
Same place	Meeting support	Team room/shift work
Different places	Desktop conferences multicast seminar	E-Mail, collaborative editor Workflow-Management

with explicitly defined control and data flow between them. At run-time, new *WF instances* $I_1, \ldots, I_n$ can be created from this WF schema and be executed according to the defined process logic.

1.1. Problem description

Team-work processes may be of complex structure and long duration. As an example take engineering processes or therapeutical treatments which may last for several months (up to years). Therefore *changes* may take place very often. Consequently, the team-work processes have to be rapidly adapted [1]. However, today's WfMS lack almost totally of supporting adaptive processes. Either they only allow changes at the WF schema level without taking running WF instances into account[1] (e.g., MQ Series Workflow and Vitria BusinessWare) or they *propagate* such WF schema changes to running WF instances without any consistency checks (e.g., Staffware). However, doing so very often leads to heavy-weight consequences like deadlocks or program crashes due to the invocation of activity components with missing input data. Although this fundamental problem has been recognized in the WF literature (e.g. [5, 10, 12, 21]), the suggested solutions are either too restrictive or not applicable in practice (cf. Section 6). Thus, when applying today's WF technology we lose just the flexibility which is indispensable for team-work needs. To overcome this limitation, basically, we must efficiently support WF schema modifications and their propagation to running WF instances.

A WF schema change can be propagated to a WF instance if this is not "contradictory" to its previous execution and would not cause errors or inconsistencies. Then this instance is said to be *compliant* with the modified schema. A straightforward solution would be to try to replay *all* events that have taken place during the execution of this WF instance so far on the changed WF schema as well. If this is possible, compliance can be guaranteed. Otherwise, the change is in conflict with previous instance execution. Apart from the fact that replaying all execution events may cause a performace penalty at the presence of a large number of WF instances this approach works well as long as no loops have to be considered. However, it is far too restrictive in conjunction with cyclic process structures (which are very typical for team-work processes). More precisely, changes that may be applied in the current state of a WF instance may be "contradictory" to previous loop iterations since the respective execution history has already logged them without taking the change into account. Therefore replaying this history on the changed WF schema is doomed to failure. To prohibit those potentially long-running instances from migrating to the new WF schema, however, is out

of touch with reality. Furthermore, using the whole information about previous execution events is very expensive. Note that there are real-world applications with hundreds up to thousands of WF instances of a given WF type. Each of them comprises extensive execution history data (see e.g. [15]) of which much is not necessary for checking compliance.

1.2. Contribution

The paper presents a comprehensive and theoretically sound approach for compliance checking in connection with WF schema changes. Comprehensive means that we do not needlessly exclude WF instances from their migration to a changed schema (as described above). In this context, a formal underpinning is indispensable to enable the WfMS to automatically decide whether a given WF instance is compliant with the new WF schema or not; i.e., whether it can be smoothly migrated to it. In addition, it must be clear which information is needed at minimum for compliance checking. In most approaches from research, however, this is not precisely stated, hence leading to either (over) restrictive solutions or to "implementation holes" later on (for a detailed discussion see Section 6). Other approaches, in turn, assume that all history data of a WF instance must be taken for checking compliance [5, 12, 21] which is, in general, too expensive as described. In this paper, we proceed in two major steps and make the following contributions:

- We first define a comprehensive compliance property which is independent from the used WF meta model and its underlying operational semantics. Furthermore, it abolishes the restrictions of existing criteria for dynamic change correctness, especially in conjunction with loops and data flow.
- We precisely state under which conditions compliance of a WF instance with a changed WF schema can be guaranteed. These conditions depend on the current state of WF instances as well as on the kind of change operation applied. In any case, the needed information is shrunken to a minimum size that way.
- By positioning formal theorems for compliance checking we establish the basis for a complete solution avoiding "implementation holes" later on.

In previous publications concerning dynamic WF changes, we focused on ad-hoc changes of individual WF instances and on related issues ($\text{ADEPT}_{\text{flex}}$ [16]). They include the provision of high-level change operations for users (e.g., to insert a new step between two sets of activities or to shift steps), related graph transformation and reduction rules, strategies for adapting data flow when a user deviates from the pre-modeled flow of control, and the undoing of temporary changes when loop backs occur. In this paper, we contribute on orthogonal issues. We develop the formal underpinning of our current work on WF schema evolution, focussing on issues related to efficient compliance checks. In Section 2 we present a generic and comprehensive compliance property which abolishes the restrictions of present approaches. Sections 3 and 4 provide simple compliance checks for control as well as data flow changes. We summarize further relevant issues in Section 5 and discuss related work in Section 6. Finally, we sketch the main contributions presented in this paper in Section 7.

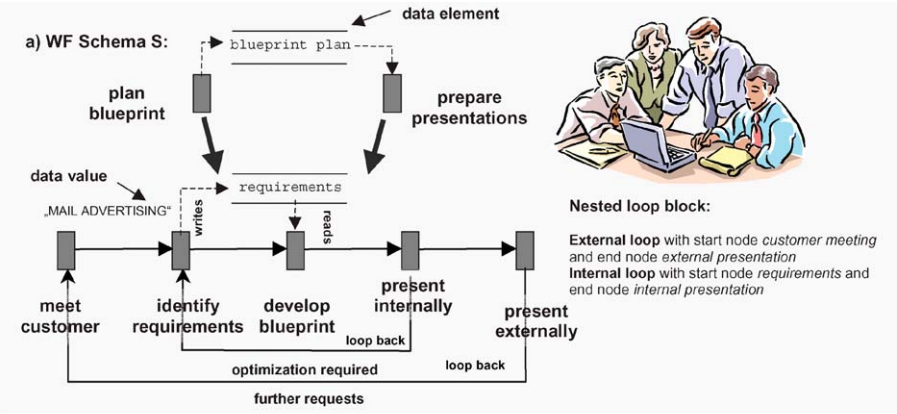
2. A comprehensive compliance property

In the following, we develop a universally valid correctness criterion for deciding whether running WF instances are compliant with a changed WF schema or not. Universally means that this criterion is independent of the underlying WF meta model. In this section, therefore, we use terms like *WF schema* and *WF instance* only in an informal manner (as described in the introduction part) to maintain the universally valid character of the presented criterion (formal notions of how a WF schema or a WF instance is defined in our approach can be found in Section 3).

To enable the WfMS to decide whether a particular WF instance can be correctly migrated to a changed WF schema or not, we need appropriate rules. In addition, it is important that these rules can be efficiently checked. Obviously, information about the execution performed so far is needed for this purpose. Many WfMS log this information within an *execution history*, which is kept for each WF instance. This history is also required, for example, when tracking the execution of a WF instance or when (partially) rolling back WF instances in case of failures [13].

A straightforward, but restrictive approach, which has been used by several groups (e.g. [5, 21]), would be to check whether the complete execution history of a WF instance could have been also produced when executing the WF instance based on the changed WF schema (*restrictive compliance property*). First, this might cause a performance penalty due to the possibly large volume of history data (see e.g. [15]) which has to be scanned. Second, following this approach, WF instances might be excluded from migrating to the changed schema, though this would not lead to inconsistencies or errors in the sequel.

Generally, the restrictive compliance property leads to problems when WF schema changes affect loop structures. As an example take figure 1 with WF schema *S*, initially consisting of a nested loop block with one external and one internal loop (including 3 activities and one data dependency). Assume that new activities `plan blueprint` and `prepare presentations` (with one data dependency between them) shall be added to WF schema *S*. This change can be easily accomplished in a correct and consistent manner at the WF schema level. But how to treat in-progress WF instances (with schema S) when applying the change? Assume that `Instance 1` is described by the execution history shown in figure 1(b). Following the restrictive approach, the intended change could not be propagated to `Instance 1` of schema *S* since no history entries for `plan blueprint` and `prepare presentations` have been written within the first (completed) iteration of the external and the internal loop. Hence, `Instance 1` is not compliant with the new schema when taking the restrictive compliance criterion. Only WF instances, which are in the first iteration of both—the internal and the external loop—could be adequately treated in this case. From a practical viewpoint, however, in most cases it will be too restrictive to prohibit change propagation for in-progress or future loop iterations only because their previous execution is not compliant with the new schema. Think of, for example, medical treatment cycles running for months or years. Any WfMS which does not allow propagating schema changes (e.g., due to the development of a new drug) to running instances (e.g., related to patients expecting an optimal treatment) would not be accepted by a medical team at all.



b) Execution History of Instance 1:

<START(meet customer, 1st it), END(meet customer), START(identify requirements, 1st it), END(identify requirements, write(requirements, "MAIL ADVERTISING"), START(develop blueprint, read(requirements, "MAIL ADVERTISING"), END(develop requirements), START(present internally), END(present internally, optimization required = FALSE), START(present externally), END(present externally, further requests = TRUE), START(meet customer, 2nd it), END(meet customer), START(identify requirements, 2nd it), END(identify requirements, write(requirements, "TV advertising")>

c) Execution History of Instance 2:

<START(meet customer, 1st it), END(meet customer), START(identify requirements, 1st it), END(identify requirements, write(requirements, "RADIO ADVERTISING"), START(develop blueprint, read(requirements, "RADIO ADVERTISING"))>

d) Reduced Execution History of Instance 1:

<START(meet customer, 2nd it), END(meet customer), START(identify requirements, 2nd it), END(identify requirements, write(requirements, "TV advertising")>

Figure 1. Team-oriented creation of a marketing concept (Example).

Additionally, the restrictive compliance property is not always suitable when considering data flow changes as well. As an example consider `Instance 2` with the execution history shown in figure 1(c). Activity `develop blueprint` has been already started and therefore has read data element `requirements`. Assume that the read data link of activity `develop blueprint` is re-mapped from `requirements` to another data element. For this instance, the activity component associated with activity `develop blueprint` is run with `requirements` as input data element though the respective data link is not present any longer in the new schema.

In summary, the support of loops is indispensable for any WfMS. To enable the WfMS to invoke arbitrary application components, it is also important to adequately handle data flow and data flow changes. The challenge is to define a compliance property, which embraces these aspects in a uniform manner as well. The key to solution with respect to loops is to be able to differentiate between completed and future executions of loop iterations. From a formal point of view there are two possibilities. One approach is to logically treat loop structures as being equivalent to respective linear sequences. Doing so allows to apply the restrictive compliance property (with full history information). The other approach is to maintain the loop construct but to restrict the evaluation to the relevant parts of the execution history. We have adopted the second approach

since it facilitates the handling of nested loops and of loops with an unknown number of iterations.

Definition 1 (Reduced Execution History $\mathcal{H}_{\text{red}}$). Let I be a WF instance with complete execution history $\mathcal{H} = \langle e_0, \ldots, e_k \rangle$, where $e_0, \ldots, e_k$ denote start and end events of all activity executions related to I. (In conjunction with loop executions there may be several entries for one activity.)—The reduced execution history $\mathcal{H}_{\text{red}}$ is obtained as follows: In the absence of loops $\mathcal{H}_{\text{red}}$ is identical to $\mathcal{H}$. Otherwise, it is derived from $\mathcal{H}$ by discarding all history entries related to other loop iterations than the last one (completed loop) or the actual iteration (running loop).

Figure 1(d) depicts the reduced execution history derived from the execution history shown in figure 1(b). From this example we can also see how Definition 1 works in conjunction with nested loops. Taking Definition 1 we now present a comprehensive compliance property for WF schema evolution. According to this property, a WF instance is compliant with a changed schema iff the reduced execution history can be produced on the modified schema as well. In the following, we assume that a correct WF schema is always transformed into another correct WF schema. Thereby, the correctness of a WF schema depends on the underlying WF meta model. Examples are constraints like the acyclic graph structure of activity nets (as for example used in MQ Series Workflow) or the bipartite net structure of Petri Nets. However, a discussion of WF schema correctness is outside the scope of this paper.

Axiom 1 (Comprehensive Compliance Property). Let I be a WF instance on WF schema S with execution history $\mathcal{H}$ and reduced execution history $\mathcal{H}_{\text{red}}$. Assume further that a change Δ transforms the WF schema S into the correct WF schema S'. Then I is said to be compliant with S' iff

- $\mathcal{H}_{\text{red}}$ can be replayed on S' as well, i.e., $\mathcal{H}_{\text{red}}$ could have been produced by a WF instance running according to S' as well.
- each started or finished activity (of the respective WF instance) would have read and each finished activity would have written the same data element values also on the new schema.

Axiom 1 is valid for all WF execution models which store information about previous execution of WF instances. Examples include activity nets as used by MQ Series Workflow, *Well-Structured-Marking Nets* (*WSM-Nets*) as used in our approach (cf. Section 3), and the WF meta models applied in BREEZE [21] and WASA_2 [25]. Approaches only maintaining state information about currently activated or running activities (e.g., Petri Nets) are discussed in Section 6.

We have now introduced a universally valid correctness criterion for ensuring compliance of WF instances with a changed WF schema, which is fundamental for any adaptive WfMS. The challenging question is how to quickly decide Axiom 1 without need for taking the (whole) extensive history information into account.[2] One approach which

is worth to follow is to design the WF execution model (including its formal and operational semantics) in such a way that efficient compliance checks avoiding access to the complete execution history become possible. For this, at the WF instance level we use a sophisticated marking approach where activity markings represent a consolidated and compact view on the execution history of a particular WF instance. In addition, when checking compliance we exploit the semantics of the applied change operations. Due to lack of space, in this paper we discuss relevant issues along our WF meta model. The presented concepts, however, are not restricted to it. Basic design principles and the achieved compliance criteria can be transferred to other WF meta models (see above) as well.

3. Checking compliance with control flow changes

In this section we provide easy to check state conditions for WF instances which allow the WfMS to ensure compliance according to Axiom 1. Before stating these rules in Section 3.2 we present necessary background information in Section 3.1. We give an (informal) overview about the WF meta model [16] assumed in this paper, the so called *Well-Structured-Marking Nets* (*WSM-Nets*). Section 3.3 concludes with a discussion on how to deal with non-compliant WF instances.

3.1. Control flow basics

WSM-Nets allow to model all relevant WF aspects, like control and data flow, work assignments, or time constraints [4, 16].

Control flow modeling. The flow of control is internally represented by attributed WF graphs with distinguishable node and edge types. As shown in earlier publications [16], this eases efficient correctness analysis (e.g., to ensure the absence of "undesired" cycles causing deadlocks) as well as the interpretative execution of WF models. For this, we use a block concept, for which control blocks (sequences, branchings, loops) can be nested but must not overlap (see figure 2). To increase expressiveness, in addition, synchronization edges (SyncE) can be used to define "wait-for" relations between parallel nodes. In figure 2, for example, the target node `L` of the sync edge `D` $\rightarrow$ `L` may be only activated if `K` has been finished and if `D` has been either completed or the branch containing `D` is not selected for execution (i.e., `D` has been skipped). Formally, a control flow schema `S` is defined as follows:

Definition 2 (Control Flow Schema (WSM-Net)). A tuple $S = (N, D, \text{NT}, \text{CtrlE}, \text{SyncE}, \text{LoopE}, \text{DP}, \text{EC})$ is called a (correct) control flow schema if the following holds:

- N is a set of activities and D a set of process data elements
- NT: $N \mapsto \{$`StartFlow, EndFlow, Activity, AndSplit, AndJoin, XOrSplit, XOrJoin, StartLoop, EndLoop`$\}$
  NT assigns to each node of the WSM-Net a respective node type.

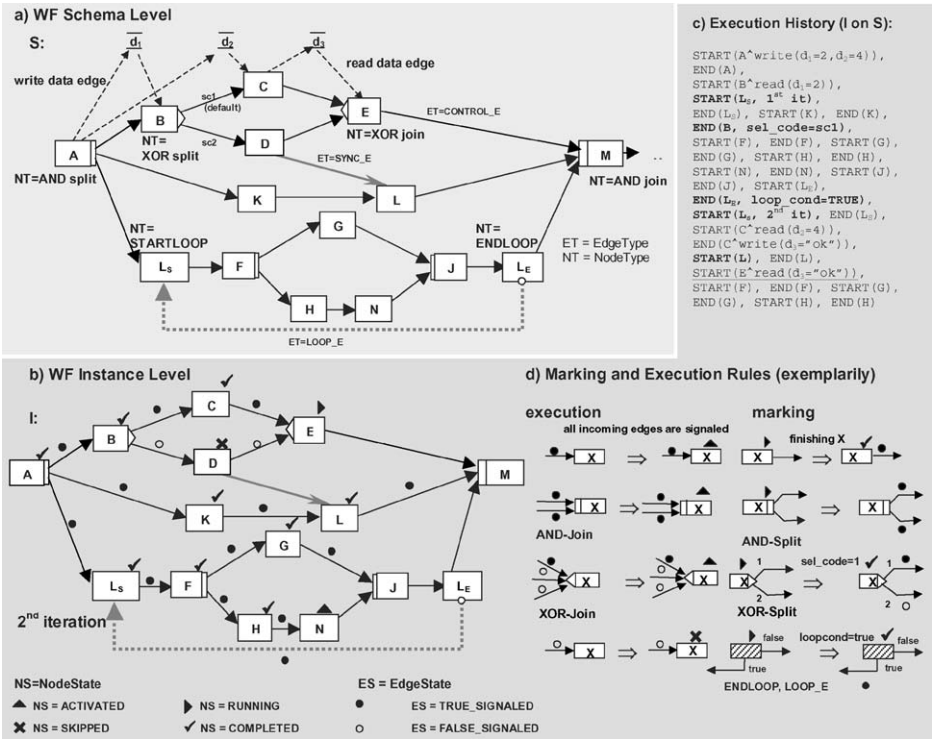


Figure 2. Modeling and execution of workflows using WSM-Nets.

– CtrlE $\subset N \times N$ is a precedence relation representing "normal" control dependencies between sequential activities
– SyncE $\subset N \times N$ is a precedence relation between activities of parallel branches
– LoopE $\subset N \times N$ is a set of loop backward edges
– DP: $N \mapsto D \cup \{\texttt{UNDEFINED}\}$

  For an XOR-split n, DP(n) corresponds to the global process data element indicating the branch to be selected. For nodes m with NT(m) $\neq$ `XOrSplit` we obtain DP(m) = `UNDEFINED`.
– EC: CtrlE $\mapsto$ EdgeCode $\cup \{\texttt{UNDEFINED}\}$

  EC(e) assigns a *selection code* to the outgoing control edges of an XOR-Split.

Informally, a control flow schema is correct iff

- $S_{\text{fwd}} = (N, \text{CtrlE}, \text{SyncE})$ is an acyclic graph, i.e., the use of control and sync edges must not cause undesired cycles leading to deadlocks (for details see [16]),
- for each split (loop start) node there is a unique join (loop end) node, and
- S is structured following a block concept; control blocks (sequences, branchings, loops) can be nested but must not overlap.

The described WSM-Nets are somewhat comparable to BPEL4WS (Business Process Execution Language for Web Services) [6], but with a more restricted use of links (called sync edges in our approach). The use of sync edges is combined with a precise formal and operational semantics and therefore enables consistency checks at buildtime as well as at runtime.

Workflow execution. Based on a given WF schema S, new WF instances can be created and started. Similar to firing rules in Petri Nets, the marking of a WF instance is determined by well defined marking and execution rules (cf. figure 2(d)). As opposed to Petri Nets, logically, for each WF instance its own marking is maintained based on the related WF schema. Markings can be considered as a very compact and space-efficient representation of the reduced execution history $\mathcal{H}_{\text{red}}$ (cf. Definition 1). In addition, except loop backs, the markings of already passed regions are maintained (cf. figure 2(b)), which is very useful for compliance checking as we show in the following. Furthermore, activity nodes of non-selected execution branches are marked as SKIPPED.

For each activity, its status is initially set to NOT_ACTIVATED. It is changed to ACTIVATED when all preconditions for its execution are met (cf. figure 2(d)). If so, the activity is released as a work task and inserted into user worklists. When selecting this activity for execution its status changes to RUNNING. The corresponding work items are then removed from other user worklists and an application component associated with this activity is started. At successful termination, activity status passes to COMPLETED. Otherwise, if the scheduler recognizes that this activity cannot be selected for execution any longer, its status will change to SKIPPED (e.g., activity D of instance I in figure 2(b)). Edges are initially marked with NOT_SIGNALED. During WF execution their status either changes to TRUE_SIGNALED or FALSE_SIGNALED. Finally, if a loop condition evaluates to true, the marking of the corresponding edge (with type LOOP_E) is changed to TRUE_SIGNALED (cf. figure 2(d)) and the markings of all activities/edges of the loop body are reset to their initial state. Otherwise the loop is left whereas the actual markings of the loop body remain. Formally, a WF instance I is defined as follows:

Definition 3 (WF Instance Based on a WSM-Net). A WF instance I is defined by a tuple $(S, M^S, \text{Val}^S, \mathcal{H})$ where

- $S = (N, D, \text{NT}, \text{CtrlE}, \text{SyncE}, \ldots)$ denotes the WF schema the execution of I is based on.
- $\text{M}^S = (\text{NS}^S, \text{ES}^S)$ describes node and edge markings of I:

  NS^S: $N \mapsto \{$`NotActivated, Activated, Running, Completed, Skipped`$\}$
  ES^S: $(\text{CtrlE} \cup \text{SyncE} \cup \text{LoopE}) \mapsto \{$`NotSignaled, TrueSignaled, FalseSignaled`$\}$

- Val^S is a function on D. It reflects for each data element $d \in D$ either its current value or the value UNDEFINED (if d has not been written yet).
- $\mathcal{H} = \langle e_0, \ldots, e_k \rangle$ is the execution history of I whereby $e_0, \ldots, e_k$ denote the start and end events of activity executions. For each started activity X the values of data elements

read by X and for each completed activity Y the values of data elements written by Y are logged.

3.2. *Checking compliance with control flow changes*

The ability to check compliance efficiently is indispensable for the flexible and efficient support of team ware processes by a WfMS. Regarding existing approaches, it remains pretty vague how compliance can be decided in conjunction with a multitude of running WF instances. Thus, we present formal and precise conditions for checking the logical compliance property (cf. Axiom 1) when new activities, control edges, or sync edges are inserted into a WF schema with related WF instance(s). (Note that the addition of a new activity node is always accompanied by the insertion of associated control or sync edges, which embed this activity into the WF schema context.) Due to a better structuring and understanding of this paper we focus on data flow issues later on (cf. Section 4).

Theorem 1 (*Insertion of Activities/Control Edges/Sync Edges*)**.** *Let* $S = (N, D, NT, CtrlE, SyncE, LoopE, DP, EC)$ *be a correct WF Schema and* I *be a WF instance on* S *with reduced execution history* $\mathcal{H}_{red}$ *and with marking* $M^S = (NS, ES)$*. Assume further that change operation* Δ *transforms* S *into a correct WF schema* $S' = (N', D', NT', CtrlE', SyncE', LoopE', DP', EC')$*.*

(a) Δ *inserts an activity* n_{insert} (*with associated control and sync edges*) *into* S*. Then*:

I *is compliant with* $S' \Leftrightarrow$
$\forall\, n \in \{x \in N \mid n_{insert} \rightarrow x \in (CtrlE' \cup SyncE')\}$:
$NS(n) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED, SKIPPED}\} \vee$
n_{insert} *is inserted into an already skipped branch of an XOR-branching*

(b) Δ *inserts a control edge* $\mathtt{n}_{src} \rightarrow \mathtt{n}_{dest}$ *into* S*. Then*:

I *is compliant with* $S' \Leftrightarrow NS(n_{dest}) \in \{NOT\_\texttt{ACTIVATED, ACTIVATED, SKIPPED}\}$

(c) Δ *inserts a sync edge* $\mathtt{n}_{\mathrm{src}} \rightarrow \mathtt{n}_{dest}$ *into* S (n_{src} *and* n_{dest} *ordered parallel so far*)*. Then*:

I *is compliant with* $S' \Leftrightarrow$
$[NS(n_{dest}) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED, SKIPPED}\}] \vee$
$[NS(n_{src}) = \texttt{COMPLETED} \wedge NS(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\}$ *with*
$\exists e_i = \texttt{END}(n_{src})\ e_j = \texttt{START}(n_{dest}) \in \mathcal{H}_{red} \wedge i < j))] \vee$
$[NS(n_{src}) = \texttt{SKIPPED} \wedge NS(n_{dest}) \in \{\texttt{RUNNING, COMPLETED}\})$ *with*
$\forall\, n \in N_{critical}$ *with* $NS(n) \neq \texttt{SKIPPED}$:
$\exists e_i = \texttt{START}(n_{dest}),\ e_j = \texttt{END}(\mathtt{n}) \in \mathcal{H}_{red}$ *with* $j < i$),
where $N_{critical} = (c\_pred^*(S, \mathtt{n}_{\mathrm{src}}) \neg c\_pred^*(S, n_{dest}))$
and $c\_pred^*(S, n)$) *denotes all direct/indirect predecessors of* n *in* S
concerning control edges]

A formal proof of this theorem is given in the Appendix. Informally, for adding activities, compliance can be always guaranteed if all (direct) successors of the newly inserted activity $\mathtt{n}_{\text{insert}}$ are actually marked with ACTIVATED or NOT_ACTIVATED. In this case they have not yet written any entry into the execution history. Interestingly, the same applies when inserting activities into already skipped branches.

Concerning the insertion of a single control or sync edge, compliance can be always ensured if the target node of the respective edge has not been started yet. This is a sufficient condition for guaranteeing compliance, but it is not always necessary. In a few cases additional information from the reduced execution history may be required to ensure compliance. As an example take WF schema S from figure 2(a). Assume that sync edge D $\rightarrow$ K is inserted into S. Regarding WF instance I (cf. figure 2(b)) we see that the source node D is skipped and the target node K is completed. According to Theorem 1c, in this situation, I is only compliant with the new schema iff B has written its end entry before the start entry of K into the execution history ($N_{\text{critical}} = \{\mathtt{B}\} \wedge \text{NS}(\mathtt{B}) \neq \mathtt{SKIPPED}$). Considering the (execution) history from figure 2(c), this constellation is obviously not given. Consequently, the insertion of sync edge D $\rightarrow$ K cannot be propagated to I.

Intuitively, delete operations are also very important for practical purposes, e.g., activities may have to be skipped (and therefore the associated control and sync edges embedding the respective activity into the workflow context be deleted). Thus we provide Theorem 2 which summarizes the compliance conditions for delete operations:

Theorem 2 (*Deletion of Activities/Control Edges/Sync Edges*)**.** *Let $S = (N, D, \ldots)$ be a correct WF Schema and I be a WF instance on S with reduced execution history $\mathcal{H}_{\text{red}}$ and marking $M^S = (NS, ES)$. Assume further that change operation Δ transforms S into a correct WF schema $S' = (N', D', \ldots)$.*

(a) Δ *deletes an activity* n_{insert} *from S* (*including the re-linking of control edges*)*. Then*:

I is compliant with $S' \Leftrightarrow$
$NS(n_{delete}) \in \{\mathtt{NOT\_ACTIVATED}, \mathtt{ACTIVATED}, \mathtt{SKIPPED}\}$

(b) Δ *deletes a control or sync edge* $\mathtt{n}_{\text{src}} \rightarrow \mathtt{n}_{\text{dest}}$ *from S. Then*:

I is compliant with S'

For delete operations compliance checks can be always performed solely on basis of activity markings. Intuitively, only those activities of a WF instance I can be dynamically deleted which have not yet written any entry into the execution history. This is the case if the node marking of the activity to be deleted is NOT_ACTIVATED, ACTIVATED, or SKIPPED. Concerning control or sync edges their deletion is uncritical with respect to compliance of WF instances with the resulting WF schema. Note that order relations between the source and end activity nodes of deleted edges are abolished. Therefore the previous execution can be replayed on the changed schema.

Order changes are an example for complex change operations which can be simply built by serially applying one or more basic operations (i.e., insertion/deletion of control or sync edges). Figure 3 shows such an order changing operation, namely swapping of two activities B and C. The comprehensive compliance property can be always ensured in conjunction

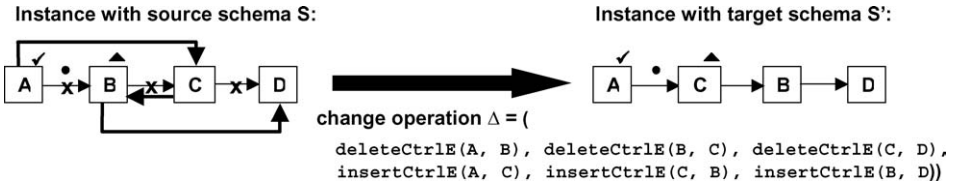


Figure 3. Complex change operation: Shifting an activity.

with such complex operations if the respective compliance conditions are fulfilled for each applied basic operation. Further optimizations are conceivable with respect to checking compliance for complex changes, but are outside the scope of this paper.

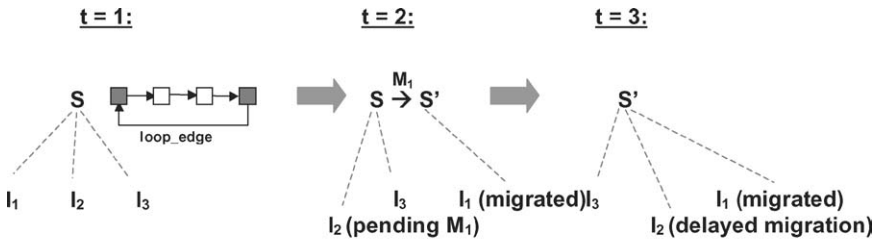
3.3. Never-more-compliant and re-compliant instances

Generally, applying the compliance property will lead to a set of WF instances, which do not fulfill this property and thus—at first glance—cannot be migrated. This includes WF instances, which can never be migrated ("*never-more-compliant instances*") and others, which only fail because the current execution of a loop iteration has proceeded too far. The latter WF instances become a candidate for migration when the loop enters its next iteration ("*re-compliant instances*").

Normally, *never-more-compliant instances* will never reach a state again in which they are compliant with the modified schema. The easiest way would be to finish these WF instances according to their old WF schema which requires appropriate versioning concepts [10, 12]. Alternatively, we can put these WF instances (or some of them) back to a compliant state by partial rollback [5, 21]. But on the one hand, only activities can be rolled back which support cancelation or compensation activities. On the other hand, rollback of processes is often out of touch with reality, in particular concerning teamware processes (e.g., patient treatment). Up to now, only in [7] the authors have recognized that in the case of loop backs WF instances may become compliant with the changed WF schema again (*re-compliant instances*).

Re-compliant instances. In particular, the marking of a loop is reset if a loop back takes place such that Axiom 1 will be satisfied with delay. Thus, WF instances which are not compliant according to their actual loop iteration may become re-compliant when another loop iteration takes place and therefore can be migrated to the new schema with delay (*delayed migration*). As shown in figure 4, re-compliant instances can be held as "pending to migration" until the loop condition is evaluated.

The treatment of re-compliant instances, which is especially important in conjuntion with long-running processes, is not as trivial as it looks like at first glance. At first, if an instance contains nested loops there can be several events (loop backs) to trigger the execution of a previously delayed migration. Furthermore, the interesting question remains how to deal with pending instances if further schema changes take place. Due to lack of space we abstain from further discussion of this point.



Event	Condition	Action
Loop_Back	ES(loop_edge) = TRUE_SIGNALED	migrate I_2 to S'

Figure 4. Principle of delayed migration.

4. Checking compliance with data flow changes

As outlined in the introduction, the proper handling of data flows and data flow changes is essential for WfMS, which shall be broadly applicable. However, with few execptions (e.g. [12]), data flow changes and their influence on running WF instances have been factored out by existing approaches so far. In particular, in some approaches (e.g., WF models based in Petri Nets), the flow of data can be only modeled in an implicit way or mixed with control flow specification. Doing so aggravates any check of compliance in conjunction with data flow changes.

In Section 4.2, we discuss how Axiom 1 can be ensured in conjunction with data flow changes. To provide a basis for discussion, in Section 4.1, we first summarize the necessary background information about data flow modeling in our approach.

4.1. Data flow basics

The *data flow* between activities is modeled by connecting *input/output parameters* of WF activities with global variables (*data elements*). Thereby each activity *input parameter* is mapped to exactly one data element by a *read data edge* and each activity *output parameter* is connected to a data element by a *write data edge*. An example is shown in figure 2(a). Activity A writes *data element* d_1 which is then read by activity B. For the modeling of such a data flow schema (*DF schema*) a number of correctness properties must be met. The most important one is that for each activity the data flow ensures that all mandatory input parameters will be supplied at runtime.

At runtime, different versions of a data object may be stored for a data element. For each write access, always a new version is created, i.e., data objects are not physically overwritten. Holding different versions is important for the context-dependent reading of data elements as well as for rollback operations in case of failures. To simplify matters, we assume that the data element values are logged within the execution history, i.e., for each started activity X the values of the data objects read by X and for each completed activity

Y the values of the data objects written by Y are stored together with the respective history entry (cf. figure 2(c)).

4.2. Checking compliance for data flow changes

Changes of a DF schema may become necessary in conjunction with control flow schema changes (e.g., removing associated data edges of an activity to be deleted) or may have to be applied independently in order to re-link data edges or data elements (e.g., if errors in the modeled data flow have to be corrected). To modify DF schemes, our approach offers operations for adding and deleting data elements as well as data edges.

Taking the compliance property from Axiom 1, all conditions set out for control flow changes (cf. Theorems 1 and 2) must be further fulfilled. Additionally, it is required that each started or finished activity (of the respective WF instance) would have read and each finished activity would have written the same data element values also on the new schema. The compliance of a WF instance in case of DF schema changes can be easily checked based on the following conditions.

Theorem 3 (*Data Flow Changes*)**.** *Let $S = (N, D, \ldots)$ be a correct WF Schema with DF schema DFS and let I be a WF instance on S with reduced execution history $\mathcal{H}_{\text{red}}$ and marking $M^S = (NS, ES)$. Assume that Δ transforms S into a correct WF schema $S' = (N', D', \ldots)$ with DF schema DFS′.*

(a) *Δ inserts a data element d into DFS. Then I is compliant with S'.*

(b) *Δ deletes a data element d from DFS. Then*:

I *is compliant with* $S' \Leftrightarrow$
No read or write access on d by an activity with state `RUNNING` *or* `COMPLETED`

(c) *Δ inserts or deletes a read edge $d \rightarrow n$. Then*:

I *is compliant with* $S' \Leftrightarrow NS(n) \in \{$`NOT_ACTIVATED, ACTIVATED, SKIPPED`$\}$

(d) *Δ inserts or deletes a write edge $n \rightarrow d$. Then*:

I *is compliant with* $S' \Leftrightarrow NS(n) \neq$ `COMPLETED`

As already mentioned, data flow adaptations also become necessary in conjunction with the insertion and deletion of activities. In this case, the conditions of Theorem 3 are already met if the state conditions of the according node insertion or deletion operations are fulfilled (cf. Theorems 1 and 2). Concerning data flow changes, again the conditions for using complex operations arise from the aggregation of the conditions of basic change operations.

5. Further issues and proof-of-concept prototype

The results presented in this paper are embedded in a major project on adaptive WF management [16, 17]. We do not only focus on efficiently checking compliance of running

WF instance with a changed WF schema but work on further important issues related to evolutionary processes as well. The first important question is how to adapt WF instance markings after their migration to the changed WF schema. In [18] we present an efficient algorithm for these marking adaptations with linear complexity.

Especially important for team-oriented processes is the interplay of WF schema modification (and their propagation to a potentially large collection of in-progress WF instances) and ad-hoc changes of single WF instances. Think of, for example, team processes where the related WF schema has to be adapted to a new law, but single WF instances have already been changed by team members (e.g., due to exceptional situations). The challenging question, arising in this context, is whether the WF schema changes can be correctly propagated to the individually modified WF instances as well and how to efficiently check this [11]. Intuitively, checking compliance no longer depends just on state conditions for individually modified instances. Moreover, structural and semantical conflicts between the WF schema and the WF instance change have to be taken into account as well [18–20].

We have implemented the presented results in a powerful proof-of-concept prototype. To avoid misunderstandings this prototype is different from the ADEPT WfMS [17]. Due to lack of space we omit a discussion of implementation details (e.g., handling of concurrent changes, caching of WF templates and instances, clustering of WF instances to improve performance of instance migrations etc.). Some illustrative screens of the prototype are shown in figures 5 and 6. In figure 5, we start with the WF schema `medical treatment`

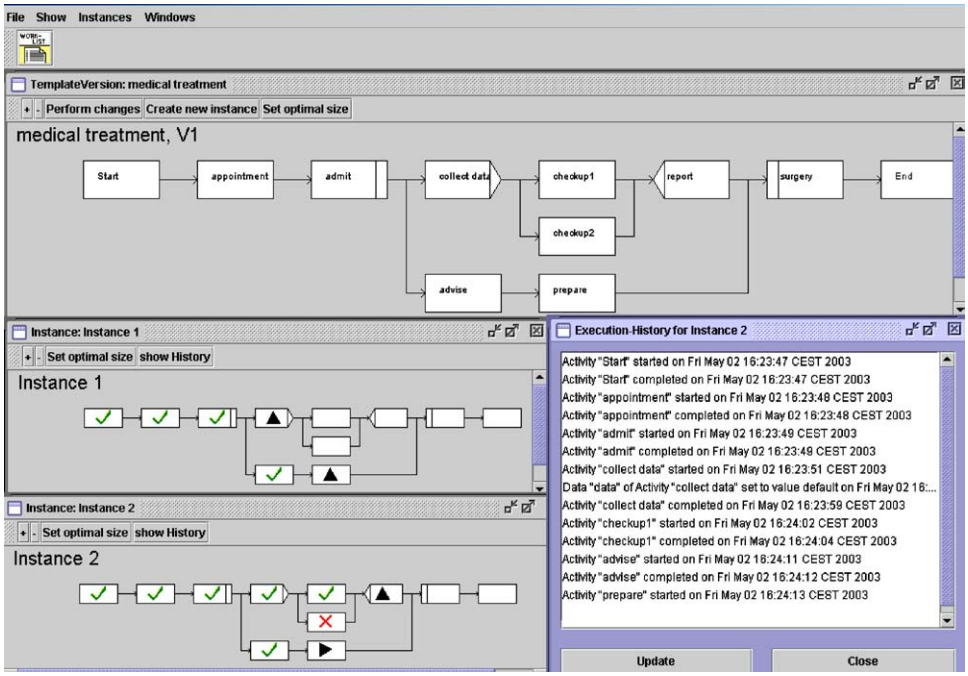


Figure 5. WF schema and two related WF instances: Pre-change.

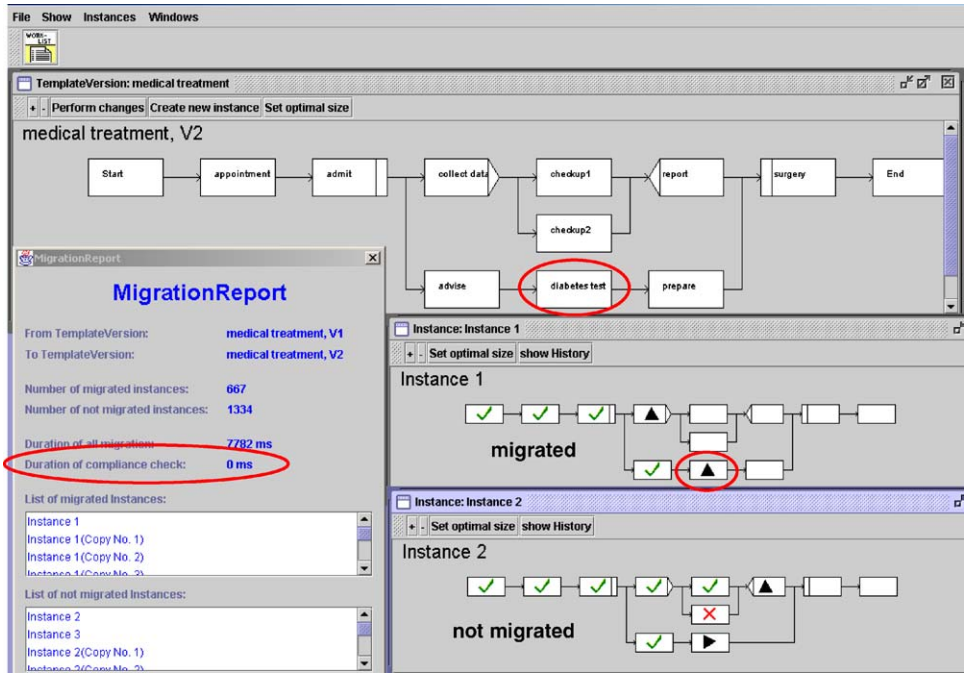


Figure 6. WF schema and two related WF instances: Post-change.

in its first version `V1`. Figure 5 also shows two related WF instances `Instance 1` and `Instance 2` (out of altogether 2000 WF instances running according to the schema `medical treatment, V1`) and the execution history of `Instance 2`.

Our prototype includes a WF editor which allows to create new WF schemes and to correctly change existing ones. Each time a modified WF schema is released, a new schema version is generated and stored in the repository. First of all, we allow designers to restrict the set of migratable instances by specifying appropriate selection predicates (based on WF attributes). For the selected instances the system automatically checks compliance using the compliance rules presented in Theorems 1–3. At this point it is important to mention that using these formal statements (and the respective proofs) has helped us to come to a complete solution without implementation holes. Afterwards, all compliant instances are migrated to the new WF schema version by correctly adapting their markings and related data structures (e.g., user worklists). The results of such a migration process are summarized in a `Migration Report` (see figure 6 for an example). Figure 6 shows the WF schema version `V2` resulting from a change of the WF schema `medical treatment (V1)` depicted in figure 5, namely the insertion of a new activity `diabetes test`. Figure 6 also shows the two instances from figure 5 after change propagation: `Instance 1` has been compliant with WF schema version `V2` and has therefore been migrated to `V2`, whereas `Instance 2` remains unchanged since it is not compliant with `V2` (cf. figure 6).

From the `Migration Report` shown in figure 6 it can be seen that the necessary compliance checks only took a very little fraction of time (when compared to the approaches replaying the whole execution history). Therefore, implementing this proof-of-concept prototype affirms that the proposed compliance checks (cf. Theorems 1 and 2) are very quick for complex WF graphs as well as for a large number of active instances. As mentioned above the set of WF instances for which compliance has to be decided can be shrunken by user defined constraints (e.g., "migrate only those WF instances that have been started after Dec, 31th 2002").

6. Related work

Obviously, there are similarities between schema changes in WfMS [5, 19, 21, 23] and in DBMS [2]. The underlying problems are similar if considerations are restricted to the mapping of schema elements (activity nodes, control/data flow edges) from the old to the new schema. WF schema evolution, however, also raises orthogonal issues. If changes at the WF schema level shall be applied at the WF instance level as well, one has to consider that WF instances may be in a different state when a change propagation takes place. Depending on their current state and on the applied change operations, a migration to the new schema may then be possible or not. For deciding which instances are compliant with the new schema and which can therefore be smoothly migrated to it, theoretically sound and efficient solutions are required.

Regarding related work on WF schema evolution [5, 7, 9, 19, 21, 23], we distinguish between *history* and *snapshot based approaches*. The latter only consider currently activated or running activities without maintaining information about their previous execution (e.g., Petri Nets). A survey on correctness criteria for dynamic WF changes and a formal comparison of respective approaches can be found in a previous publication of our group [19].

History based approaches. WIDE [5] offers a complete and minimal set of basic operations to transform a correct schema S into another correct schema S'. To migrate WF instances to S', for the first time, the restrictive compliance property as discussed in Section 2 has been suggested. TRAMs [12] focuses on WF schema versioning concepts. To efficiently manage an instance migration the authors propose the definition of so called migration conditions for each change operation which are somewhat similar to the presented compliance rules. With these conditions it can be decided whether an instance can smoothly migrate to the new WF schema version or not. Recent results concerning WF schema evolution come from the BREEZE project [21], which uses a model and change operations similar to our approach [16]. BREEZE uses compliance as a correctness criterion as well but focuses on the question how to deal with non-compliant WF instances. In summary, all these approaches are too restrictive in conjunction with loops since they are based on the restrictive compliance property. Furthermore, compliance in connection with data flow schema changes has not been considered in detail. Finally, the authors do not show how their suggested compliance property can be (formally) checked, which is important when incorporating compliance checks into a WF engine implementation.

Object oriented approaches are offered by Joeris and Herzog [10] and Weske [25]. In MOKASSIN [10] changes are carried out by encapsulating change primitives within WF instances. Consequently, WF instances or users are themselves responsible for preserving consistency. The restrictive compliance property is considered as being too restrictive. Instead, a more granular version concept is proposed, but without discussing issues related to (efficient) compliance checks. Another versioning approach has been presented by $WASA_2$ [25], which proposes a mapping between the modified WF schema and the sub-workflows resulting from the corresponding instances to allow efficient compliance checks. However, data flow changes have not been treated in detail and formal considerations are not given.

Snapshot based approaches. Petri Net based approaches [7, 8, 22, 23] fight with several approach-inherent problems: Generally, they often lack a clear seperation between control and data flow tokens, which complicates (dynamic) net changes. In [7], both, the WF schema and the WF instances are captured in one Petri net based on coloured markings. (To avoid misunderstandings, in our approach, multiple WF instances may be related to the same WF schema. As opposed to Petri Nets, however, each WF instance has its own marking defined on that schema.) A schema modification is carried out by substituting marked sub nets, whereas precise or formal conditions for checking compliance of WF instances with the new net are missing. Another serious problem arises from the fact that markings of previously passed regions are not preserved and "skipped" regions are not marked at all. Therefore the "challenging" question is how to adapt instance markings after propagating a schema change without knowledge of their previous execution. In [8] the WF designer has to manually adapt the markings for each WF instance. In addition, complex reachability analyses become necessary to check consistency of net markings after a change. In contrast, the compliance conditions proposed in this paper and the respective marking adaptation algorithms (cf. [18]) are of linear complexity.

Recent approaches wrestle with that problem as well. In [22] the authors propose that WF schema modifications shall not be propagated to WF instances which are executed on modified regions. The adaptation of markings is seen as a very complex problem (the so called dynamic change bug) [23]. To fix this bug the authors suggest that the modeler has to specify a mapping between the markings of the old net an the new net which has to be applicable for every running instance [23]. Besides, Petri Nets suffer from the implicit modeling of cycles. Thus the distinction between desired cycles and deadlocks is a NP-hard problem.

7. Summary and outlook

Applications which aim at the support of complex, long-running team processes need adaptive workflow to be able to react rapidly to process changes. Therefore, flexible WfMS will be a key technology in this context. We have been engaged in several application projects dealing with patient treatment and product development, for instance, and have gained deep insights into team processes and collaborative real-world scenarios. One important conclusion we have drawn from these projects is that by offering more flexibility and adaptability the so promising WF technology will finally deliver in many collaborative scenarios as

well. In this paper we have elaborated a comprehensive and formal foundation for checking compliance of WF instances with a (modified) WF schema. The compliance criteria embrace WF schemes with (nested) loops and with explicitly defined data flows. One very important aspect of this work is its formal style and rigour. We have positioned axioms and theorems which are fundamental for the design of any adaptive workflow model. The handling of control as well as data flow changes and the provision of a proof-of-concept prototype add to the overall completeness of our approach. The solution has been described using WSM Nets but may be easily applied to other WF models with similar properties (e.g. [3, 14]).

In this paper we have concentrated on correctness criteria and their efficient evaluation in the context of WF schema evolution. How to efficiently check WF instances for compliance is one issue, how to "physically" perform the migrations (incl. correct state adaptations), how to internally represent WF instances and WF schemes, how to interact with the WF schema designer (who defines the change), how to adapt user worklists, or how to to deal with concurrent changes (and with locking issues in this context) are other important questions. Work on some of these issues is in progress [18–20]. The challenge is to elaborate solutions, which do not work only in an isolated fashion but in conjunction with each other as well.

Appendix

To prove Theorem 1 we first give some useful information. To begin with, we do not need any special treatment of loops since using the reduced execution history logically leads to a "loop-free" WF schema. Thus we have to care about acyclic WF schema graphs with sequences, AND-branchings and XOR-branchings. Furthermore, Table 2 informally summarizes certain predecessor and successor functions on WF schema graphs which are needed for the following considerations.

Finally, we need the following Lemma 1 to prove Theorem 1. It states that all predecessors of a running or completed activity $\mathtt{n}^*$ must have one of the markings COMPLETED or SKIPPED.

Table 2. Predecessor and successor functions on WF graphs.

c_succ(S,n)/c_pred(S,n)	set of all *direct* successors/predecessors of activity n considering only edges e ∈ CtrlE in WF schema S
$\mathtt{c\_succ}^*\mathtt{(S,n)}/\mathtt{c\_pred}^*\mathtt{(S,n)}$	set of all *direct* or *indirect* successors/predecessors of activity n considering only edges e ∈ CtrlE in WF schema S
succ(S,n)/pred(S,n)	set of all *direct* successors/predecessors of activity n referring to edges e ∈ (CtrlE ∪ SyncE) in WF schema S
$\mathtt{succ}^*\mathtt{(S,n)}/\mathtt{pred}^*\mathtt{(S,n)}$	set of all *direct* and *indirect* successors/predecessors of activity n referring to edges e ∈ (CtrlE ∪ SyncE) in WF schema S
	$\mathtt{succ}^*\mathtt{(S,n)} = \{n^* \in N \mid n^* \in \mathtt{succ(S}, n) \vee (\exists\, n^{**} \in \mathtt{succ(S}, n)\colon n^* \in \mathtt{succ}^*\mathtt{(S}, n^{**}))\}$

Lemma 1. *Let $S = (N, D, \ldots)$ be a correct WF schema and I a WF instance on S with marking $M^S = (NS, ES)$. Then*:

$$\forall\, n^* \in N \text{ with } NS(n^*) \in \{\texttt{RUNNING, COMPLETED, SKIPPED}\} \Rightarrow$$
$$\forall\, n \in \texttt{pred}^*(S, n^*): NS(n) \in \{\texttt{COMPLETED, SKIPPED}\}$$

Proof Sketch (Lemma 1): For arbitrary paths $w = i_0 \rightarrow \cdots \rightarrow i_k$ in S we can show by induction over the length k of w:

$$[\mathrm{NS}(i_k) \in \{\texttt{RUNNING, COMPLETED, SKIPPED}\} \Rightarrow$$
$$\mathrm{NS}(i_\mu) \in \{\texttt{COMPLETED, SKIPPED}\}\ \forall \mu = 0, \ldots, k-1]$$

Based on this, the proposition of Lemma 7 can be easily proven. □

We now have done all necessary prepartory work for proving Theorem 1.

Proof (Theorem 1):

(a) Δ inserts an activity n_{insert} (with associated control and sync edges) into S.
This proposition can be more formally described as follows:

I is compliant with $S' \Leftrightarrow B_1 \vee B_2 \vee B_3$ with
$B_1 \equiv [\forall \mathrm{n} \in \texttt{succ}(\mathrm{S}', n_{\text{insert}}):$
$\mathrm{NS}(n) \in \{\texttt{NOT\_ACTIVATED, ACTIVATED, SKIPPED}\}]$
$B_2 \equiv [\forall\, n \in \texttt{c\_pred}(\mathrm{S}',\ n_{\text{insert}}): \mathrm{NS}(n) = \texttt{SKIPPED}]$
$B_3 \equiv [n_{\text{insert}}$ is inserted into a skipped, empty branch]

(The statement "n_{insert} is inserted into an already skipped branch" corresponds to $B_2 \vee B_3 \vee$ $[\forall \mathrm{n} \in \texttt{c\_succ}(\mathrm{S}', n_{\text{insert}}): \mathrm{NS}(n) = \texttt{SKIPPED}]$ where the last term is already included by B_1.)

"$\Rightarrow$" I is compliant with $S' \Rightarrow B_1 \vee B_2 \vee B_3$

Proof by Contradiction, we show: $\neg(B_1 \vee B_2 \vee B_3) \Rightarrow I$ is not compliant with S'

Assumption: $\neg(B_1 \vee B_2 \vee B_3)$ holds
$\neg(B_1 \vee B_2 \vee B_3) \equiv \neg B_1 \wedge \neg B_2 \wedge \neg B_3$
$\equiv [\exists n^* \in \texttt{succ}(\mathrm{S}', \mathrm{n}_{\text{insert}}): \mathrm{NS}(n^*) \in \{\texttt{RUNNING, COMPLETED}\} \wedge$
$[\exists n^{**} \in \mathrm{c\_pred}(\mathrm{S}', n_{\text{insert}}): \mathrm{NS}(n^{**}) \neq \texttt{SKIPPED}] \wedge$
$[n_{\text{insert}}$ is not inserted into a skipped, empty branch]

With $\neg B_1$ and Lemma 1 we obtain NS'(n_{insert}) $\in \{\texttt{COMPLETED, SKIPPED}\}$. Consequently, the marking $\mathrm{NS}(n_{\text{insert}})$ must be SKIPPED. After re-evaluating the marking of the modified instance (cf. figure 2(e)), a newly inserted activity will be either marked as SKIPPED (insertion into a skipped branch) or as NOT_ACTIVATED or ACTIVATED.

Taking the above assumption, n_{insert} must therefore have been inserted into an already skipped branch of an XOR-branching with split node s and join node j. Because of $\neg B_3$ this branch cannot be empty. Based on this, it either follows that n_{insert} is not a direct successor of s—then $\forall\ n \in$ c_pred(S', n_{insert}): NS(n) = SKIPPED—or n_{insert} is not a direct predecessor of j—$\forall\ n \in$ c_succ(S', n_{insert}): NS(n) = SKIPPED. The first statement can not be true because of $\neg B_2$ and the latter because of $\neg B_1$. This is contradicting to our assumption. □

Let now statements C_1 and C_2 be as follows:

$C_1 \equiv$ [$\forall\ n \in$ succ(S',n_{insert}):
NS(n) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}]
$C_2 \equiv$ [n_{insert} is inserted into a skipped branch of an XOR-branching]

"$\Leftarrow$": $C_1 \vee C_2 \Rightarrow I$ is compliant with S' (according to Axiom 1)
We first prove $C_1 \Rightarrow I$ is compliant with S'.

Assumption:
$C_1 \equiv$ [$\forall\ n \in$ succ(S',n_{insert}):
NS(n) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}]
$\Rightarrow \forall\ n \in$ succ(S', n_{insert}):
$\nexists\ e_i \in \mathcal{H}_{\text{red}}$ with $e_i \in$ {START(n), END(n)}
$\Rightarrow \forall\ n \in$ succ*(S', n_{insert}):
$\nexists\ e_i \in \mathcal{H}_{\text{red}}$ with $e_i \in$ {START(n), END(n)} ($\diamond$)

That means that the history $\mathcal{H}_{\text{red}}$ contains no entry of a direct or indirect successor of n_{insert}. Furthermore, a re-evaluation of the instance marking results in

NS'(n_{insert}) $\in$ {NOT_ACTIVATED, ACTIVATED, SKIPPED}
$\Rightarrow\ \nexists\ e_i \in \mathcal{H}_{\text{red}}$ with $e_i \in$ {START(n_{insert}), END(n_{insert})} ($\diamond\diamond$)

We now show that I is compliant with S', i.e., the previous execution events $e_0, \ldots, e_k$ stored in $\mathcal{H}_{\text{red}}$ can be applied to S' in the given order. Let N_{rel} be the set of all activity nodes of N' which can be executed before n_{insert} is started (see figure 7(a)). So, N_{rel} contains all

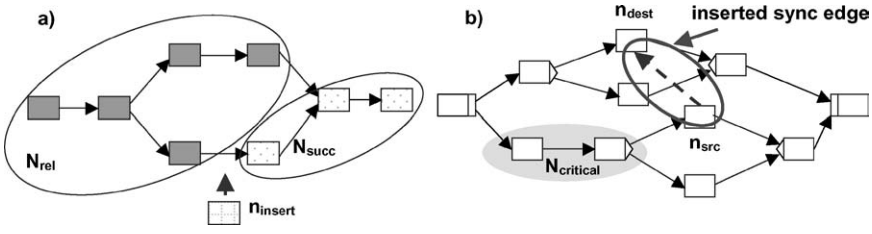


Figure 7. Important sets of a WF schema referring to n_{insert}.

activity nodes positioned before or parallel to n_{insert}. Formally:

$$N_{\text{rel}} := \text{pred}^*(S', n_{\text{insert}}) \cup \{n \in N' \mid n \notin \text{pred}^*(S', n_{\text{insert}}) \wedge n \notin \text{succ}^*(S', n_{\text{insert}})\}$$

With ($\diamond$) and ($\diamond\diamond$) it follows:

$$\forall\, e_i \in \mathcal{H}_{\text{red}} \text{ with } e_i = \texttt{START(n)} \vee e_i = \texttt{END(n)}: n \in N_{\text{rel}} \subseteq N$$

Thus all entries of $\mathcal{H}_{\text{red}}$ have been written by activity nodes which are—in principle—executable before n_{insert} referring to S'. Since the subgraph of S induced by the node set N_{rel} (cf. figure 7(a)) is not affected by the insertion and therefore remains unchanged, $e_1, \ldots, e_k$ can be carried out on this subgraph in the given order and therefore on S' as well.

Referring to the second part [$C_2 \Rightarrow I$ is compliant with S'] it is clear that n_{insert} is inserted into a skipped branch, i.e., we obtain NS'(n_{insert}) = SKIPPED. Therefore n_{insert} has not yet written any entry into the execution history. Consequently, the previous execution history $\mathcal{H}_{\text{red}}$ is producible on S' as well.

In the following, we first prove part (c) of Theorem 1 (insertion of sync edges into S) since part (b) (insertion of control edges) is less complex and can be proven in a similar way.

(c) Δ inserts a sync edge $\text{n}_{\text{src}} \rightarrow \text{n}_{\text{dest}}$ into S (n_{src} and n_{dest} ordered parallel so far).
First let

$$\begin{aligned}
A_1 &\equiv [\text{NS}(n_{\text{dest}}) \in \{\texttt{NOT_ACTIVATED, ACTIVATED, SKIPPED}\}]\\
A_2 &\equiv [(\text{NS}(n_{\text{src}}) = \texttt{COMPLETED} \wedge \text{NS}(n_{\text{dest}}) \in \{\texttt{RUNNING, COMPLETED}\})\\
&\quad \text{with } \exists e_i, e_j \in \mathcal{H}_{\text{red}}: i < j \wedge e_i = \texttt{END}(n_{\text{src}}), e_j = \texttt{START}(n_{\text{dest}})]\\
A_3 &\equiv [(\text{NS}(n_{\text{src}}) = \texttt{SKIPPED} \wedge \text{NS}(n_{\text{dest}}) \in \{\texttt{RUNNING, COMPLETED}\})\\
&\quad \text{with } \forall n \in N_{\text{critical}} \text{ with } \text{NS}(n) \neq \texttt{SKIPPED}:\\
&\quad\ \exists e_k, e_l \in \mathcal{H}_{\text{red}}: l < k \wedge e_k = \texttt{START}(n_{\text{dest}}), e_l = \texttt{END(n)}]\\
&\quad \text{where } N_{\text{critical}} = (\text{c_pred}^*(n_{\text{src}}) \neg \text{c_pred}^*(n_{\text{dest}})) \text{ (cf. figure 7(b))}
\end{aligned}$$

The negation of A_1, A_2 and A_3 yields

$$\begin{aligned}
\neg A_1 &\equiv [\text{NS}(n_{\text{dest}}) \in \{\texttt{RUNNING, COMPLETED}\}]\\
\neg A_2 &\equiv [\text{NS}(n_{\text{src}}) \neq \texttt{COMPLETED} \vee\\
&\quad \text{NS}(n_{\text{dest}}) \in \{\texttt{NOT_ACTIVATED, ACTIVATED, SKIPPED}\} \vee\\
&\quad \nexists e_i, e_j \in \mathcal{H}_{\text{red}}: i < j \wedge e_i = \texttt{END}(n_{\text{src}}), e_j = \texttt{START}(n_{\text{dest}})]\\
\neg A_3 &\equiv [\text{NS}(n_{\text{src}}) \neq \texttt{SKIPPED} \vee\\
&\quad \text{NS}(n_{\text{dest}}) \in \{\texttt{NOT_ACTIVATED, ACTIVATED, SKIPPED}\} \vee\\
&\quad \exists n \in N_{\text{critical}} \text{ with } \text{NS}(n) \neq \texttt{SKIPPED}:\\
&\quad \nexists e_k, e_l \in \mathcal{H}_{\text{red}}: l < k \wedge e_k = \texttt{START}(n_{\text{dest}}), e_l = \texttt{END(n)}]
\end{aligned}$$

"$\Rightarrow$": I is compliant with $S' \Rightarrow A_1 \vee A_2 \vee A_3$

Proof by contradiction, we show:

$\neg(A_1 \vee A_2 \vee A_3) \Rightarrow I$ is not compliant with S'

Assumption: $\neg(A_1 \vee A_2 \vee A_3)$ holds.

$$\begin{aligned}
&\neg(A_1 \vee A_2 \vee A_3) \equiv \neg A_1 \wedge \neg A_2 \wedge \neg A_3 \equiv (\neg A_1 \wedge \neg A_2) \wedge \neg A_3\\
&\equiv [(\mathrm{NS}(n_{\mathrm{dest}}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{\mathrm{src}}) \neq \texttt{COMPLETED})\\
&\quad \vee (\mathrm{NS}(n_{\mathrm{dest}}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge\\
&\qquad \nexists e_i, e_j \in \mathcal{H}_{\mathrm{red}}\colon i < j \wedge e_i = \texttt{END}(n_{\mathrm{src}}), e_j = \texttt{START}(n_{\mathrm{dest}}))]\\
&\quad \wedge \neg A_3\\
&\equiv [(\mathrm{NS}(n_{\mathrm{dest}}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{\mathrm{src}}) \neq \texttt{COMPLETED})\\
&\quad \vee ((\exists e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}})) \wedge\\
&\qquad ((\nexists e_i \in \mathcal{H}_{\mathrm{red}}\colon e_i = \texttt{END}(n_{\mathrm{src}})) \vee\\
&\qquad (\exists e_i \in \mathcal{H}_{\mathrm{red}}\colon e_i = \texttt{END}(n_{\mathrm{src}}) \wedge i > j)))] \wedge \neg A_3\\
&\equiv [(\mathrm{NS}(n_{\mathrm{dest}}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \mathrm{NS}(n_{\mathrm{src}}) \neq \texttt{COMPLETED})\\
&\quad \vee ((\exists e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}}) \wedge\\
&\qquad \nexists e_i \in \mathcal{H}_{\mathrm{red}}\colon e_i = \texttt{END}(n_{\mathrm{src}})) \vee\\
&\quad (\exists e_i, e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}}), e_i = \texttt{END}(n_{\mathrm{src}}) \wedge i > j))]\\
&\quad \wedge \neg A_3\\
&\equiv [(\exists e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}}) \wedge \nexists e_i \in \mathcal{H}_{\mathrm{red}}\colon e_i = \texttt{END}(n_{\mathrm{src}}))\\
&\quad \vee (\exists e_i, e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}}), e_i = \texttt{END}(n_{\mathrm{src}}) \wedge i > j)]\\
&\quad \wedge \neg A_3\\
&\equiv\colon (E_1 \vee E_2) \wedge \neg A_3 \; (\equiv (E_1 \wedge \neg A_3) \vee (E_2 \wedge \neg A_3))
\end{aligned}$$

Because of $n_{\mathrm{src}} \in$ pred(S', n_{dest}) and due to the compliance of I with S' the end entry of n_{src} cannot be situated before the start entry of n_{dest} in the execution history $\mathcal{H}_{\mathrm{red}}$; i.e., E_2 and therefore $(E_2 \wedge \neg A_3)$ cannot hold. Accordingly, $(E_1 \wedge \neg A_3)$ must hold.

$$\begin{aligned}
&(E_1 \wedge \neg A_3)\\
&\quad \equiv [\exists e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}}) \wedge \nexists e_i \in \mathcal{H}_{\mathrm{red}}\colon e_i = \texttt{END}(n_{\mathrm{src}})] \wedge\\
&\qquad [\mathrm{NS}(n_{\mathrm{src}}) \neq \texttt{SKIPPED}\\
&\qquad\quad \vee \mathrm{NS}(n_{\mathrm{dest}}) \in \{\texttt{NOT_ACTIVATED, ACTIVATED, SKIPPED}\}\\
&\qquad\quad \vee \exists n \in N_{\mathrm{critical}}, \mathrm{NS}(n) \neq \texttt{SKIPPED}\colon\\
&\qquad\qquad \nexists e_k, e_l \in \mathcal{H}_{\mathrm{red}}\colon l < k \wedge e_k = \texttt{START}(n_{\mathrm{dest}}), e_l = \texttt{END(n)}]\\
&\quad \equiv [\exists e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}}) \wedge\\
&\qquad \nexists e_i \in \mathcal{H}_{\mathrm{red}}\colon e_i = \texttt{END}(n_{\mathrm{src}})\\
&\qquad \wedge \mathrm{NS}(n_{\mathrm{src}}) \neq \texttt{SKIPPED}]\\
&\qquad \vee [(\exists e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}})\\
&\qquad\quad \wedge \nexists e_i \in \mathcal{H}_{\mathrm{red}}\colon e_i = \texttt{END}(n_{\mathrm{src}}))\\
&\qquad\quad \wedge \mathrm{NS}(n_{\mathrm{dest}}) \in \{\texttt{NOT_ACTIVATED, ACTIVATED, SKIPPED}\}) \vee\\
&\qquad\quad (\exists e_j \in \mathcal{H}_{\mathrm{red}}\colon e_j = \texttt{START}(n_{\mathrm{dest}}) \wedge\\
&\quad \nexists e_i \in \mathcal{H}_{\mathrm{red}}\colon e_i = \texttt{END}(n_{\mathrm{src}})\\
&\quad \wedge (\exists n \in N_{\mathrm{critical}}, \mathrm{NS}(n) \neq \texttt{SKIPPED}\colon\\
&\qquad \nexists e_k, e_l \in \mathcal{H}_{\mathrm{red}}\colon l < k \wedge e_k = \texttt{START}(n_{\mathrm{dest}}), e_l = \texttt{END(n)}))]
\end{aligned}$$

$$
\begin{aligned}
&\equiv [\exists e_j \in \mathcal{H}_{\text{red}}\colon e_j = \texttt{START}(n_{\text{dest}}) \\
&\quad \wedge\ \nexists e_i \in \mathcal{H}_{\text{red}}\colon e_i = \texttt{END}(n_{\text{src}}) \wedge \text{NS}(n_{\text{src}}) \neq \texttt{SKIPPED}] \\
&\quad \vee [(\exists e_j \in \mathcal{H}_{\text{red}}\colon e_j = \texttt{START}(n_{\text{dest}}) \wedge \nexists e_i \in \mathcal{H}_{\text{red}}\colon e_i = \texttt{END}(n_{\text{src}}) \\
&\qquad \wedge (\exists n \in N_{\text{critical}}, \text{NS}(n) \neq \texttt{SKIPPED}\colon \\
&\qquad\quad \nexists\, e_k, e_l \in \mathcal{H}_{\text{red}}\colon l < k \wedge e_k = \texttt{START}(n_{\text{dest}}), e_l = \texttt{END}(\texttt{n}))] \\
&\equiv\colon C_1 \vee C_2
\end{aligned}
$$

C_1 results in

$\text{NS}(n_{\text{dest}}) \in \{\texttt{RUNNING, COMPLETED}\} \wedge \text{NS}(n_{\text{src}}) \notin \{\texttt{COMPLETED, SKIPPED}\}$.
In this case I cannot be compliant with S'. Therefore C_2 must hold.

$$
\begin{aligned}
&C_2 \\
&\quad \equiv [\exists e_j \in \mathcal{H}_{\text{red}}\colon e_j = \texttt{START}(n_{\text{dest}}),\ \nexists e_i \in \mathcal{H}_{\text{red}}\colon e_i = \texttt{END}(n_{\text{src}}) \wedge \\
&\qquad (\exists n \in N_{\text{critical}} \text{ with } \text{NS}(n) \neq \texttt{SKIPPED}\colon \\
&\qquad\quad \nexists\, e_k, e_l \in \mathcal{H}_{\text{red}}\colon l < k \wedge e_k = \texttt{START}(n_{\text{dest}}), e_l = \texttt{END}(\texttt{n}))] \\
&\quad \equiv (\exists e_j \in \mathcal{H}_{\text{red}}\colon e_j = \texttt{START}(n_{\text{dest}}) \wedge \nexists e_i \in \mathcal{H}_{\text{red}}\colon e_i = \texttt{END}(n_{\text{src}})) \wedge \\
&\qquad (\exists n \in N_{\text{critical}}, \text{NS}(n) \neq \texttt{SKIPPED} \wedge \\
&\qquad\quad (\nexists e_l \in \mathcal{H}_{\text{red}}\colon e_l = \texttt{END}(\texttt{n}) \vee \\
&\qquad\qquad \exists e_l \in \mathcal{H}_{\text{red}}\colon e_l = \texttt{END}(\texttt{n}) \wedge j < l\ )) \\
&\quad \equiv [(\exists e_j \in \mathcal{H}_{\text{red}}\colon e_j = \texttt{START}(n_{\text{dest}}) \wedge \nexists e_i \in \mathcal{H}_{\text{red}}\colon e_i = \texttt{END}(n_{\text{src}})) \\
&\qquad\quad \wedge (\exists n \in N_{\text{critical}}, \text{NS}(n) \neq \texttt{SKIPPED} \\
&\qquad\qquad \wedge \nexists e_l \in \mathcal{H}_{\text{red}}\colon e_l = \texttt{END}(\texttt{n}))] \vee \\
&\qquad [(\exists e_j \in \mathcal{H}_{\text{red}}\colon e_j = \texttt{START}(n_{\text{dest}}) \wedge \nexists e_i \in \mathcal{H}_{\text{red}}\colon e_i = \texttt{END}(n_{\text{src}})) \\
&\qquad\quad \wedge (\exists n \in N_{\text{critical}}, \text{NS}(n) \neq \texttt{SKIPPED} \\
&\qquad\qquad \wedge \exists e_l \in \mathcal{H}_{\text{red}}\colon e_l = \texttt{END}(\texttt{n}) \wedge j < l\ )] \\
&\quad \equiv\colon D_1 \vee D_2
\end{aligned}
$$

Because of D_1 it follows that there is a predecessor node $n \in N_{\text{critical}}$ of n_{src} which is neither marked as COMPLETED nor as SKIPPED (see figure 7(b)). Referring to S' this node is also a predecessor of n_{dest} since S' contains the additional edge $n_{\text{src}} \rightarrow n_{\text{dest}}$. Accordingly, I cannot be compliant with S'.

D_2 yields that a predecessor node $n \in N_{\text{critical}}$ of n_{src} with $\text{NS}(n) = \texttt{COMPLETED}$ exists whose end entry is situated after the start entry of n_{dest} in the execution history $\mathcal{H}_{\text{red}}$. Since n is a predecessor of n_{dest} in S' it follows that I is not compliant with S'.

"$\Leftarrow$": $A_1 \vee A_2 \vee A_3 \Rightarrow I$ is compliant with S'
With A_1 it follows that $\mathcal{H}_{\text{red}}$ still does not contain an entry related to n_{dest}. Therefore $\mathcal{H}_{\text{red}}$ could have been produced on S' as well; i.e., I is compliant with S'. The same applies to A_2 because the end entry of n_{src} had been written into $\mathcal{H}_{\text{red}}$ before the start entry of n_{dest} was logged.

After insertion of $n_{\text{src}} \rightarrow n_{\text{dest}}$, in any case, n_{src} has to be either executed or skipped before n_{dest} is activated or skipped. In addition, other (predecessor) nodes of n_{src}, which could have been executed parallel to n_{dest} so far may now have to be executed or skipped

before n_{dest} can be marked. This node set is determined by N_{critical} (see figure 7(b)). Only if each activity node of N_{critical} has either been marked as SKIPPED or has written its end entry before the start entry of n_{dest} into $\mathcal{H}_{\text{red}}$, the execution history can be produced on the new schema S' as well. This follows directly from A_3.

Notes

1. Only allowing future WF instances to be run according to the new version of the WF schema.
2. This problem is comparable to serializability of database transactions, which is ensured by suitable synchronization methods; i.e., the defined compliance criterion (Axiom 1) can be considered as a general correctness criterion (like serializability) for which we have to find suitable checking routines.

References

1. A. Agostini and G. de Michelis, "A light workflow management system using simple process models," Int'l Journal of Collaborative Comp., vol. 9, nos. 3/4, pp. 335–363, 2000.
2. J. Andany, M. Leonard, and C. Palisser, "Management of schema evolution in databases," in Proc. Int'l Conf. VLDB '91, Barcelona, Sept. 1991, pp. 161–170.
3. S. Bassil, M. Benyoucef, R. Keller, and P. Kropf, "Addressing dynamism in e-negotiations by workflow management systems," in Proc. DEXA'2002 Workshop, Sept. 2002.
4. T. Bauer and P. Dadam, "Efficient distributed workflow management based on variable server assignments," in Proc. Int'l CAiSE '00, Stockholm, June 2000, pp. 94–109.
5. F. Casati, S. Ceri, B. Pernici, and G. Pozzi, "Workflow evolution," Data and Knowledge Engineering, vol. 24, no. 3, pp. 211–238, 1998.
6. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana, Business Process Execution Language for Web Services, Version 1.0, 2002. http://www.ibm.com/developerworks/library/ws-bpel/.
7. C.A. Ellis, K. Keddara, and G. Rozenberg, "Dynamic change within workflow systems," in Proc. Int'l ACM Conf. COOCS '95, Milpitas, CA, August 1995, pp. 10–21.
8. C.A. Ellis and C. Maltzahn, "The Chautauqua workflow system," in Proc. Int'l Conf. on System Science, Maui, 1997.
9. A. Fent, H. Reiter, and B. Freitag, "Design for change: Evolving workflow specifications in ULTRAflow," in Proc. Int'l CAISE '02, May 2002, pp. 516–534.
10. G. Joeris and O. Herzog, "Managing evolving workflow specifications," in Proc. Int'l Conf. CoopIS '98, New York City, August 1998, pp. 310–321.
11. 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, vol. 13, pp. 43–72, 2003.
12. M. Kradolfer and A. Geppert, "Dynamic workflow schema evolution based on workflow type versioning and workflow migration," in Proc. Int'l Conf. CoopIS '99, Edinburgh, Sept. 1999, pp. 104–114.
13. F. Leymann and D. Roller, Production Workflow, Prentice Hall, 2000.
14. R. Müller and E. Rahm, "Dealing with logical failures for collaborating workflows," in Proc. Int'l Conf. CoopIS '00, Eilat, 2000, pp. 210–223.
15. P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum, "Workflow history management in virtual enterprises using a light-weight workflow management system," in Proc. RIDE'99, March 1999, pp. 148–155.
16. M. Reichert and P. Dadam, "ADEPT$_{\text{flex}}$—supporting dynamic changes of workflows without losing control," Journal of Intelligent Information Systems, vol. 10, no. 2, pp. 93–129, 1998.
17. M. Reichert, S. Rinderle, and P. Dadam, "ADEPT workflow management system: Flexible support for enterprise-wide business processes (tool presentation)," in Proc. Int'l Conf. BPM '03, LNCS 2678, Eindhoven, Springer, June 2003, pp. 370–379.

18. M. Reichert, S. Rinderle, and P. Dadam, "A formal framework for workflow type and instance changes under correctness constraints," Technical Report UIB-2003-01, University of Ulm, Computer Science Faculty, April 2003.
19. S. Rinderle, M. Reichert, and P. Dadam, "Evaluation of correctness criteria for dynamic workflow changes," in Proc. Int'l Conf. BPM '03, LNCS 2678, Eindhoven, Springer, June 2003, pp. 41–57.
20. S. Rinderle, M. Reichert, and P. Dadam, "On dealing with semantically conflicting business process changes," Technical Report UIB-2003-04, University of Ulm, Computer Science Faculty, June 2003.
21. S. Sadiq, O. Marjanovic, and M. Orlowska, "Managing change and time in dynamic workflow processes," Int'l Journal of Cooperative Information Systems, vol. 9, no. 1/2, pp. 93–116, 2000.
22. W.M.P. van der Aalst, "Exterminating the dynamic change bug: A concrete approach to support worfklow change," Information Systems Frontiers, vol. 3, no. 3, pp. 297–317, 2001.
23. W.M.P van der Aalst and T. Basten, "Inheritance of workflows: An approach to tackling problems related to change," Theoretical Computer Science, vol. 270, no. 1/2, pp. 125–203, 2002.
24. M. Weber, Distributed Systems, Spektrum, Akademischer Verlag, 1998 (in German).
25. M. Weske, "Flexible modeling and execution of workflow activities," in Proc. 31st Int'l Conf. on System Sciences, Hawaii, 1998, pp. 713–722.