Evaluation of Correctness Criteria for Dynamic Workflow Changes*

Stefanie Rinderle, Manfred Reichert, and Peter Dadam

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

Abstract. The capability to dynamically adapt in-progress workflows (WF) is an essential requirement for any workflow management system (WfMS). This fact has been recognized by the WF community for a long time and different approaches in the area of adaptive workflows have been developed so far. They either enable WF type changes and their propagation to in-progress WF instances or (ad-hoc) changes of single WF instances. Thus, at first glance, many of the major problems related to dynamic WF changes seem to be solved. However, this picture changes when digging deeper into the approaches and considering implementation and usability issues as well. This paper presents important criteria for the correct adaptation of running workflows and analyzes how actual approaches satisfy them. At this, we demonstrate the strengths of the different approaches and provide additional solutions to overcome current limitations. These solutions comprise comprehensive correctness criteria as well as migration rules for change realization.

1 Introduction

A rapidly changing environment and a turbulent market force any company to change their business processes ever more frequently [1]. Process changes become necessary, for example, when new laws come into effect, optimized or restructured business processes are to be implemented, exceptional situations occur, or rapid reactions to a changed market are required. Therefore, a critical challenge for the competitiveness of any enterprise is its ability to quickly react to business process changes [2,3,4].

As pointed out in [2], basically, changes can take place at two levels – the WF type and the WF instance level. Very often changes at the WF instance level are applied in an ad-hoc manner, leading to WF instances with biased execution schema when compared to their original WF schema – in the following, we denote these WF instances as *biased*. Ad-hoc changes become necessary in conjunction with real-world exceptions, e.g., a sudden circulatory collapse of a

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

W.M.P. van der Aalst et al. (Eds.): BPM 2003, LNCS 2678, pp. 41–57, 2003.

patient, and they usually affect only *single WF instances*. As opposed to this, in conjunction with schema changes at the WF type level, a collection of related instances may have to be adapted. The challenging question is how to *propagate* WF type changes to running WF instances, but without violating correctness and consistency properties set out by the used WF meta model. In other words, how can we smoothly *migrate* WF instances to a changed WF schema? Additionally, in case of concurrent changes (e.g., concurrent changes at the type and instance level), the exciting question arises how to synchronize them (e.g., how to propagate WF type changes to biased WF instances).

There is a multitude of approaches dealing with flexibility in WfMS [1,2,4,5, 6,7,8]. All of them present very interesting, but partially strongly differing ideas and solutions. Therefore, it is an important job to summarize central criteria for adaptive workflows and to compare actual approaches by using these criteria. Furthermore, we sketch suitable solutions for "still dangling" issues, e.g., related to the problem of checking compliance of WF instances with a modified schema.

At first, we summarize important criteria for different change scenarios, which are necessary to achieve a correct and consistent "post-change"-behavior.

1. **Completeness:** The WF designer must not be restricted, neither by the used WF meta model nor the offered change operations. Therefore, a WF meta model ought to provide a complete set of control and data flow constructs, e.g., allow the designer to model sequences, parallel/alternative branchings, and loops [3]. For practical purposes, at minimum, change operations for inserting and deleting activities as well as control/data dependencies between them are required. Furthermore, it must be able to combine change primitives to define complex changes, e.g., to modify the order of activities.
2. **Correctness:** The ultimate ambition of all adaptive WF meta models must be correctness of dynamic changes [1,2,4,5,6,7,8]; i.e., introducing changes to the runtime system without causing inconsistencies or errors (like deadlocks or improperly invoked activity programs). Therefore, adequate *correctness criteria* are needed. These criteria must not be too restrictive, i.e., no WF instance should be needlessly excluded from applying a dynamic change. Furthermore, it must be clear how the imposed correctness criteria can be easily and quickly checked by the WfMS. This is especially important for large-scale environments with hundreds up to thousands of WF instances.
3. **Change Realization:** Assuming that a dynamic change can be correctly propagated to a WF instance I (along the stated correctness criteria), it should be possible to automatically migrate I to the new schema. In this context, the WF instance state as well as dependent data structures (e.g., user worklists) must be correctly and efficiently adapted.

In the following, we provide a classification of actual approaches which is based on the semantics of the underlying WF meta models and on the above criteria. We point out where the strengths and weaknesses of these approaches lie. To overcome current limitations we discuss solutions which can be easily transferred to other WF models (e.g., a comprehensive correctness criterion, efficient compliance checks, and formal propositions regarding concurrent changes).

In Section 2 we give an overview of current approaches dealing with flexibility in WfMS and classify them with respect to their semantics. Section 3 summarizes and classifies correctness criteria for adaptive workflows. In Section 4 we show how the different approaches realize dynamic WF changes. Section 5 presents a critical discussion and Section 6 closes with a short summary.

2 Approaches Dealing with Flexibility

Figure 1 summarizes adaptive WF meta models which enable a flexible process support. According to [9], we classify those WF meta models according to the evaluation strategies applied for executing WF instances during runtime. The first strategy uses only one type of (control flow) token passing through each WF instance (*True-Tokens*). The other strategy is based on two types of tokens – *True-* and *False-Tokens*. True-Tokens represent activities that are to be executed next and False-Tokens describe activities which have been skipped. Approaches which solely use True-Tokens (cf. Fig. 2) have a *True-Semantics* and include, for example, Petri-Net-based formalisms [2,5,10,1]. Approaches which, in addition, use False-Tokens to represent skipped activities or skipped execution branches can be found in the area of graph-based WF meta models [8,7]. They can be further divided according to the way they represent the True- and False-Tokens. One possibility is to gain these tokens (and therefore the state of running instances) from *execution histories* [4], which log events like start and completion of activity executions (cf. Fig. 3). Alternatively, special (*model-inherent*) markings of activities and/or control edges, which represent a consolidated view on the history logs, can be used [6,7,8] (cf. Fig. 4).

2.1 Approaches with True-Semantics

In [2], a WF schema is represented by a *WF Net* which is a labeled place/transition net $N = (P, T, F, l)$ (cf. Fig. 2). Thereby, P denotes the set of places, T the set of transitions, $F \subseteq (T \times P) \cup (P \times T)$ the set of directed arcs, and l the labeling function, which assigns a label to each transition. The dynamic behavior of a WF instance is described by a *marked WF net* (N, s) with marking (function) s and associated marking rules. The authors abstract from data flow issues, WF attributes and WF resources and consider only one WF instance at a time.

The approach presented in [5,14,10] is based on *Flow Nets*, which are closely related to WF nets. In Chautauqua [14], Flow Nets are generalized to *Information Control Networks (ICN)*. They allow the enactment of a new WF instance by creating an instance specific token, which represents a data form of the enacted ICN. In doing so, data flow is carried out by passing the token through the ICN. WF instances (with same WF type) are distinguished by the use of coloured tokens and are controlled by the same ICN. A meta language to support dynamic evolution of processes is presented in [10]. In the following, however,

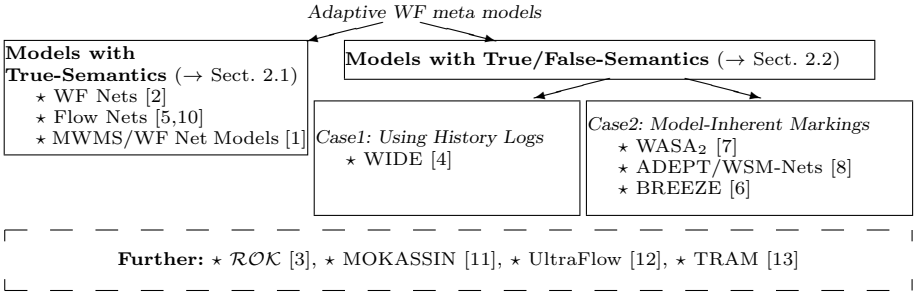

Fig. 1. Selected Approaches Dealing With Flexibility Issues

we focus on the formalisms of Flow Nets as described above. An example for the above approaches is depicted in Fig. 2a.

Another interesting approach is presented by *MWMS* [1]. The authors use *Net Models (NM)*, which are marked, acyclic Free-Choice Petri Nets. Data flow issues are not taken into account. A NM Σ can be mapped to a *Sequential Model (SM)* $\mathcal{A}$ which represents the global states and state transitions of Σ. Thus, $\mathcal{A}$ is comparable to the *reachability graph* of Petri Net Σ.

2.2 Approaches with True/False-Semantics

Case 1: Approaches based on History Logs
The most famous example and also one of the first approaches dealing with dynamic WF changes was offered by *WIDE* [4], which uses a graph-based WF meta model. The modeling of sequential, parallel, alternative, and iterative activity executions is possible. Furthermore, there is a set of global process variables associated with each WF schema S. A WF instance I on WF schema S can be described by S and by its execution history $\mathcal{H} = (\langle \epsilon^S_{I,0}, \mu^S_{I,0} \rangle, \ldots, \langle \epsilon^S_{I,i}, \mu^S_{I,i} \rangle)$, where $\epsilon^S_{I,k}$ denotes the k^{th} completion of a task execution in I and $\mu^S_{I,i}$ denotes related write operations on WF variables performed by $\epsilon^S_{I,k}$.

In WIDE, WF schemata can be described either graphically or by using predecessor and successor functions. Fig. 3 shows the latter variant.

Case 2: Approaches Using Model Inherent Markings
WASA$_2$ [7] introduces an object-oriented WF meta model. It comprises one generic class *Workflow* of which *WF schema* and *WF instance* are instances. Workflows are modeled by using a graph-based WF language comparable to activity nets applied in IBM MQ Series Workflow. In more detail, a *WF schema* $S = (V_S, C_S, D_S)$ is a tuple with sets of activity nodes V_S, control connectors C_S, and data connectors D_S. Similarly, a *WF instance I* can be described. The flow of data is modeled by data connectors which map output and input parameters of subsequent activities. A WF schema S is correct iff all input parameters are correctly supplied by a type-conform output parameter and the graph structure

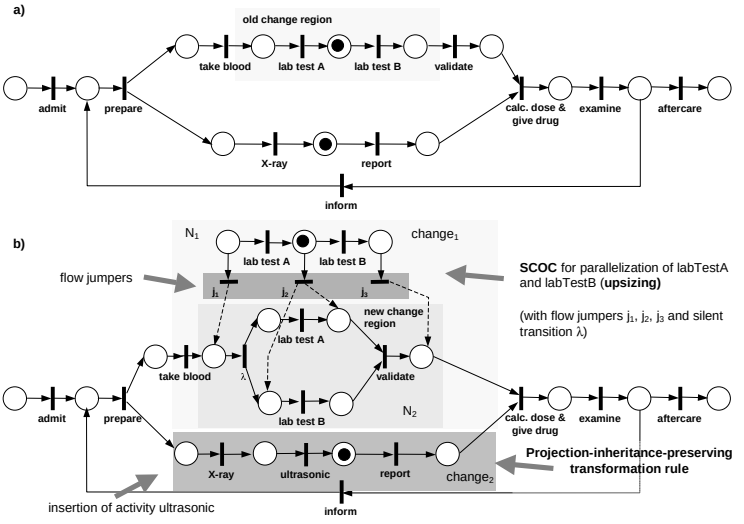

Fig. 2. A Petri-Net-Based Workflow With Changes ([2,5,10])

is acyclic (i.e., no deadlocks will occur). The state of a WF instance is denoted by the marking of the instance nodes (model-inherent).

Another approach with model-inherent markings is based on *Well-Structured Marking-Nets (WSM-Nets)* as applied in our *ADEPT* WfMS [8], for example. WSM-Nets are serial-parallel graphs with distinguishable node and edge types, where loops and branchings are modeled in a block-oriented fashion (block structure). This structure is relaxed by offering *sync edges*, which allow to define precedence relations between activities of parallel branches. We first provide two self-explanatory definitions for WSM-Nets (cf. Def. 1) and for WF instances (cf. Def. 2) based on them.

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

- *N is a set of activities and D a set of process data elements*
- *NT: $N \mapsto$* {`StartFlow, EndFlow, Activity, AndSplit, AndJoin, XOrSplit, XOrJoin, StartLoop, EndLoop`}
- *$CtrlE \subset N \times N$ is a precedence relation*
- *$SyncE \subset N \times N$ is a precedence relation between activities of parallel executed branches*
- *$LoopE \subset N \times N$ is a set of loop backward edges*
- *$DataE \subseteq N \times D \times$ {read, write} is a set of read/write data links between activities and data elements*
- *EC: $CtrlE \cup SyncE \cup LoopE \mapsto$* `Predicates`*$(D)$ where* `Predicates`*(D) denotes the set of all valid transition conditions on data elements from D.*

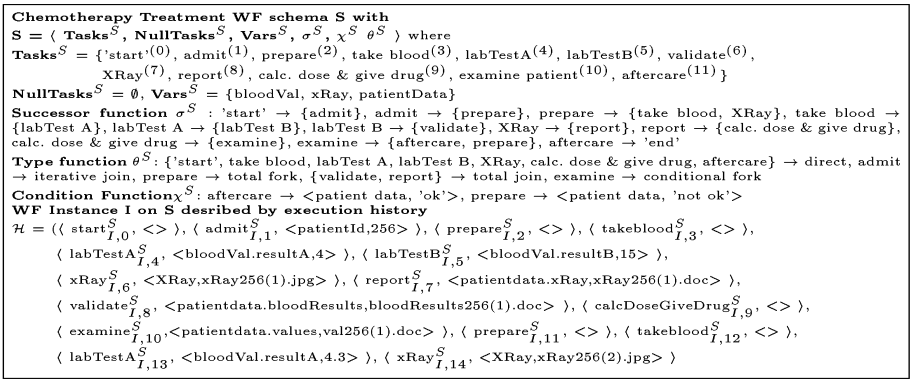

Fig. 3. WF Instance Including History Logs in WIDE

A WSM-Net is correct iff
• S_{fwd} = (N, CtrlE, SyncE) is an acyclic graph, i.e., the use of sync edges must not cause undesired cycles leading to deadlocks (for details see [8]),
• for each split (loop start) node there is a unique join (loop end) node, and
• S is structured following a block concept, for which control blocks (sequences, branchings, loops) can be nested but must not overlap.

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

- *S = (N, D, NT, CtrlE, SyncE, ...) denotes the WSM-Net the execution of I is based on.*
- *M^S = (NS^S, ES^S) describes node and edge markings of I:*
  NS^S: $N \mapsto$ {`NotActivated, Activated, Running, Completed, Skipped`}
  ES^S: (CtrlE ∪ SyncE ∪ LoopE) $\mapsto$ {`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).*
- *$\mathcal{H} = < e_0, \ldots, e_k >$ is the execution history of I. $e_0, \ldots, e_k$ denote the start and end events of activity executions. For each started activity X the values of data elements read by X and for each completed activity Y the values of data elements written by Y are logged.*

Activities marked as `Activated` are ready to fire and can then be worked on, i.e., their status changes to `Running`. Activities with marking `Skipped` cannot longer be selected for execution. An example of a WF instance based on a WSM-Net is shown in Fig. 4. Another approach using WF graphs similar to WSM-Nets but without loops is offered by *BREEZE* [6].

3 Correctness Criteria for Dynamic WF Changes

We first focus on WF schema changes at the type level and their propagation to running WF instances. Regarding correctness, however, it is not important

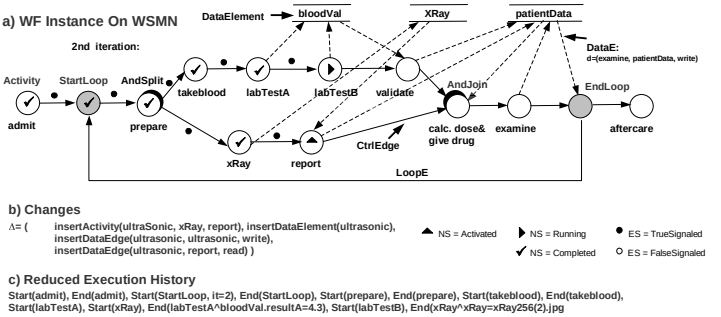

Fig. 4. A (Clinical) WF Instance Based On WSM-Nets

whether a WF type (and therefore a collection of running instances) or a single WF instance is affected by a change. At the end of this section we provide a correctness criterion to handle concurrent changes at the type and the instance level (i.e., to correctly propagate WF type changes to biased WF instances).

In the following, we assume that a WF schema S is always correctly transformed into another schema S' by applying change Δ. What this exactly means depends on the structural and dynamic correctness properties set out by the used WF meta model. We focus on the discussion how the approaches from Section 2 decide whether an instance I can be correctly migrated to a changed schema S' or not. It is remarkable that all discussed solutions are based on formal correctness criteria. While some of these approaches precisely state how to ensure correctness in conjunction with dynamic WF changes, others do not address this point in detail.

We further distinguish between approaches founding their correctness criteria on *graph equivalence* – *WF Nets* [2], *MWMS* [1], and *WASA*$_2$ [7] – and approaches with correctness criteria based on *execution equivalence* – *Flow Nets* [5,10], *WIDE Nets* [4],

and *ADEPT WSM-Nets* [8]. The core idea of graph equivalence is to map the WF instance graph of I to the changed WF schema S'. Depending on the "degree of coverage" it can be decided whether Δ is applicable to I as well. Execution equivalence focuses on the work done by I so far. If this work could have been achieved on S' as well, I can be smoothly migrated to S'. Note that both approaches are closely related to each other.

3.1 Approaches Based on Graph Equivalence

WF Nets: A first representative based on graph equivalence (see above) is described in [2]. The authors use *branching bisimilarity* as an equivalence relation on marked, labeled P/T-Nets (cf. Section 2). It specifies under which conditions two different marked, labeled P/T-Nets S and S' have the same (observable) behavior (notation: $S \sim_b S' \iff \exists$ branching bisimulation $\mathcal{R}$ such that $S\mathcal{R}S'$).

Informally, a marked, labeled P/T-Net must be able to simulate each action of an equivalent marked net.

Correctness-Criterion 1 (Branching Bisimilarity) *Let S be a marked, labeled WF Net and Δ be a change which transforms S into another marked, labeled WF Net S'. Then: Δ can be carried out correctly iff $S \sim_b S'$.*

Generally, it is difficult to ensure Criterion 1 for arbitrary changes. Therefore the authors restrict the set of possible change operations to those which preserve special inheritance relations between the old and the new net (e.g., insertion of activity `ultrasonic` in Fig. 2b). If these inheritance relations hold after applying a change, branching bisimilarity between both nets can be ensured. Inheritance-ensuring change operations are:

- adding sequences as well as parallel, alternative and iterative branches (direction of inheritance from class to subclass) and
- removing sequences as well as parallel, alternative and iterative branches (direction of inheritance from class to superclass).

Unfortunately, branching bisimilarity cannot be automatically ensured for other change operations [2]. The reason for this is a phenomenon called *dynamic change bug*. Examples for non-supported operations are order-changing operations like parallelizing transitions `labTestA` and `labTestB` in Fig. 2a. Excluding those change operations, however, leads to serious problems since related forward and backward jumps have to be frequently applied in practice.

In *MWMS* [1] a set of change operations (parallelization, sequentialization and swapping of activities) is proposed obeying special constraints (summarized by the *Minimal Critical Specification (MCS)* for the underlying SM). Intuitively, only such change operations can be carried out which maintain the given set of activities and which only change their order relations. For these changes the following correctness criterion is provided:

Correctness-Criterion 2 (Safe States) *Let $\mathcal{A} = (S, E, T, s_{in})$ be a SM and Δ be a change operation (within the respective MCS). Δ transforms $\mathcal{A}$ into another SM $\mathcal{A}' = (S', E', T', s'_{in})$. Then an instance I on $\mathcal{A}$ can be migrated to $\mathcal{A}'$ iff I is not in an unsafe state. A state of $\mathcal{A}$ is unsafe iff there is no corresponding state in S'*

Another approach using graph equivalence is offered by $WASA_2$ [7]. The author does not explicitly state which changes can be applied. Exemplarily, operations like adding and deleting activities or changing activity orders are provided. The decision whether a change Δ can be applied to an in-progress WF instance or not is based on a *valid mapping* of the *purged WF instance graph* to the WF schema graph. Thereby, the purged WF instance graph is gained by deleting all activities which have not been started yet, and by removing all associated control and data connectors (cf. Fig. 5b). A mapping $m : V_I \mapsto V_S$ between a WF instance $I = (V_I, C_I, D_I)$ and a WF schema $S' = (V_{S'}, C_{S'}, D_{S'})$ is defined as follows:

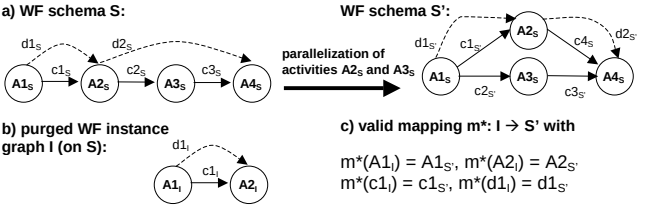

Fig. 5. Valid mapping in WASA$_2$

$$\forall j' \in V_I : \exists j \in V_S \text{ with } [\text{m(j')} = \text{j} \Rightarrow \text{SchemaOf(j')} = \text{j (i.e., j' is based on j)}]$$
$$\wedge\ [\text{m(j')} = \text{m(k')} \Rightarrow j' = k' \forall j', k' \in V_I]$$

With this, the following correctness criterion based on *valid mappings* between WF instance graph and WF schema graph can be stated:

Correctness-Criterion 3 (Valid Mapping) *Let $I = (V_I, C_I, D_I)$ be a purged WF instance graph derived from WF schema S. Then: Change Δ can be correctly applied to I as well iff $\exists$ a valid mapping m*: $V_I \mapsto V'_S$. A mapping m* is valid if all control connectors between two instance objects i, $j \in V_I$ have counterparts i', $j' \in V_{S'}$ with SchemaOf(i) = i' and SchemaOf(j) = j' and $\forall d \in D_I \exists\ d' \in D_{S'}$ (and vice versa).*

Intuitively, an instance I can be migrated to a changed WF schema S' if each completed activity of I is also contained in S' and all control and data dependencies existing in I have counterparts in S' (cf. Fig. 5c).

To our knowledge no statements have been published so far, how Criterion 3 can be (efficiently) checked. However, an implementation of WASA$_2$ exists [7].

3.2 Approaches Based on Execution Equivalence

Flow Nets: A first approach based on execution equivalence has been presented in [5,10] (details of the used Flow Nets have been given in Section 2.1). In [5], changes of a Flow Net S (with True-Semantics) are carried out by substituting the marked sub-net $\mathcal{N}_1$ of S, which is affected by Δ, by another marked sub-net $\mathcal{N}_2$, which reflects the modifications set out by Δ. Thereby, $\mathcal{N}_1$ is referred to as the *old change region* and $\mathcal{N}_2$ as *new change region*. As the authors point out, the selection of the change regions cannot be fixed. Roughly, the old change region is defined as the smallest marked sub-net containing all activities affected by Δ. Assume, for example, that in Fig. 2a) change operation Δ is to parallelize the so far sequentially ordered activities `LabTestA` and `LabTestB`. Then $\mathcal{N}_1$ is the sub-net containing the affected activities `LabTestA` and `LabTestB`. To be able to decide whether Δ can be correctly propagated to an instance or not the authors introduce the *pre-change (firing) sequence* ω to conceptualize the work done by the WF instance so far. Thereby, ω denotes all transition firings previous to the introduction of Δ. Based on this, the following correctness criterion can be formulated:

Correctness-Criterion 4 (Pre-Change sequence ω) *Let Δ be a change, which transforms Flow Net S into Flow Net S'. Let further be I an instance on S with pre-change sequence ω. Then I can be migrated to S' iff ω can be continued on S' as well.*

In order to check whether Criterion 4 is met, the authors present two kinds of change operations and a special change class, the so called *Synthetic Cut-Over Change (SCOC)*. Applying SCOC, the old change region $\mathcal{N}_1$ is maintained in S' together with $\mathcal{N}_2$ (for an example see Fig. 2b); i.e., S' contains two versions of the modified subnet. How this "fusion" of old and new change region is carried out depends on the applied change. In [5] two change scenarios – *Upsizing* and *Downsizing* – are introduced. Upsizing means that $\mathcal{N}_2$ can "do more" than $\mathcal{N}_1$, i.e., the set of all valid firing sequences on $\mathcal{N}_1$ is a subset of all valid firing sequences on $\mathcal{N}_2$. Downsizing is the dual counterpart of upsizing, i.e., $\mathcal{N}_2$ can "do less" than $\mathcal{N}_1$. For example, Fig. 2b shows an upsizing. In this case, the SCOC can be constructed by sticking $\mathcal{N}_1$ and $\mathcal{N}_2$ together over *flow-jumpers* (cf. Fig. 2b). Flow-jumpers are transitions, which map each marking of $\mathcal{N}_1$ to a marking of $\mathcal{N}_2$. This way of constructing the SCOC in conjunction with upsizing operations is correct regarding Criterion 4. In the other case – downsizing – the SCOC is constructed by merging $\mathcal{N}_1$ and $\mathcal{N}_2$ over one output place, i.e., instances with tokens in $\mathcal{N}_1$ are further executed according to the old net. Trivially, this restrictive approach is also correct regarding Criterion 4. Other important change operations, like the insertion of new activities, are not discussed. Very interesting is that upsizing and downsizing are excluded by [2] since these changes lead to the dynamic change bug (cf. Section 3.1).

A widely-used correctness property is the *compliance criterion* introduced by *WIDE* [4]. Intuitively, change Δ of WF schema S can be correctly propagated to a WF instance I iff the execution of I, taken place so far, can be "simulated" on the modified WF schema S' as well. Note that Criterion 5 is similar to Criterion 4 at first glance. But Criterion 4 is only based on a snapshot of the WF execution whereas Criterion 5 takes the whole WF execution into account. Since the authors work with a history-based execution model, compliance is based on *replaying the execution history $\mathcal{H}$* of WF instance I on the changed WF schema S'. Formally:

Correctness-Criterion 5 (Intuitive Compliance Criterion) *Let S be a WF schema and I be a WF instance on S with execution history $\mathcal{H}$. Let further S be transformed into another schema S' by change operation Δ. Then: I is compliant with S' iff $\mathcal{H}$ can be produced on S' as well.*

Assume that in WF schema S in Fig. 3 task `aftercare` is to be deleted. Referring to execution history $\mathcal{H}$ of WF instance I (cf. Fig. 3) the intended deletion is possible since $\mathcal{H}$ contains no entry related to `aftercare` and can therefore be (re-)produced on the changed WF schema as well. As opposed to this, the insertion of a new task `ultrasonic` between tasks `xRay` and `report` is not possible regarding Criterion 5. The reason is that activity `ultrasonic` has not written

any entries into $\mathcal{H}$ during the first loop iteration (note that the loop is actually in its 2^{nd} iteration). Inserting `ultrasonic` in the actual loop iteration, however, would cause no inconsistencies or errors at runtime (irrespective of (rare) roll-back operations into former loop iterations which become more expensive). Consequently, Criterion 5 is too restrictive, especially in conjunction with iterative, long-running workflows. Since in [4] no further information about how to check Criterion 5 is given, we assume that compliance is ensured by trying to replay the whole execution history on the changed WF schema. Doing so causes a big overhead due to the possibly extensive volume of the history (caused by information like user assignments or time stamps).

WSM-Nets: In *ADEPT* [8], we focus on finding a correctness criterion which works in conjunction with loops as well as other orthogonal aspects (e.g., data flow). 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 possible approaches. One approach is to logically treat loop structures as being equivalent to respective linear sequences. The other approach is to maintain the loop construct but to restrict the evaluation to the relevant parts of the execution history (cf. Def. 3). We adope the second approach since it facilitates the treatment of nested loops, provides a good basis for implementation, and leads to "smart" proofs.

Definition 3 (Reduced Execution History $\mathcal{H}_{red}$). *Let I be a WF instance with execution history $\mathcal{H}$. The reduced execution history $\mathcal{H}_{red}$ is obtained as follows: In the absence of loops $\mathcal{H}_{red}$ 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). (Note that $\mathcal{H}_{red}$ can be easily produced in conjunction with nested loops as well.)*

As an example take Fig. 4c, which shows the reduced execution history for the instance from Fig. 4a. Taking Def. 3 we now present a comprehensive compliance criterion for WF schema evolution. According to this property, an instance is compliant with a changed schema iff the reduced execution history can be produced on the modified schema as well.

Correctness-Criterion 6 (Comprehensive Compliance Criterion) *Let I be a WF instance on WF schema S with execution history $\mathcal{H}$ and reduced execution history $\mathcal{H}_{red}$. Assume further that a change Δ transforms S into the correct WF schema S'. Then I is said to be compliant with S' iff $\mathcal{H}_{red}$ can be produced on S' as well.*

Again, the challenging question is how to efficiently check the comprehensive compliance criterion. We present easily and quickly checkable marking conditions for each kind of change on WSM-Nets [8] (additive, subtractive, order-changing, and complex operations). Due to lack of space, we exemplarily summarize these conditions for additive change operations in Theorem 1.

Theorem 1 (Additive Change Operations On WSM-Nets). *Let S = (N, D, ...) be a correct WSM-Net and I be a WF instance on S with execution history*

$\mathcal{H}_{red}$. *Assume further that change* Δ *transforms S into correct WSM-Net S' = (N', D', ...).*
(a) Δ *inserts an activity* n_{insert} *(with associated control and sync edges) into S. Then:*
I is compliant with S' $\Leftrightarrow$
$\forall\ n \in \{x \in N \mid n_{insert} \rightarrow x \in E'\}$: *NS(n)* $\in$ {`NotActivated, 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}) $\in$ {`NotActivated, Activated, Skipped`}
(c) Δ *inserts a sync edge* $\mathtt{n}_{src} \rightarrow \mathtt{n}_{dest}$ *into S (*n_{src} *and* n_{dest} *ordered parallel so far). Then:*
I is compliant with S' $\Leftrightarrow$
[*NS*(n_{dest}) $\in$ {`NotActivated, Activated, Skipped`}] $\vee$
[*NS*(n_{src}) = `Completed` $\wedge$ *NS*(n_{dest}) $\in$ {`Running, Completed`} *with*
$\exists e_i = \mathtt{END}(n_{src}),\ e_j = \mathtt{START}(n_{dest}) \in \mathcal{H}_{red} \wedge i < j))$] $\vee$
[*NS*(n_{src}) = `Skipped` $\wedge$ *NS*(n_{dest}) $\in$
{`Running, Completed`}) *with*
$\forall\ n \in N_{critical}$ *with NS(n)* $\neq$ `Skipped`*:*
$\exists e_i = \mathtt{START}(\mathtt{n}_{dest}),\ e_j = \mathtt{END}(\mathtt{n}) \in \mathcal{H}_{red}$ *with* $j < i$*),*
where $N_{critical}$ = (*c_pred** *(S,* n_{src}) $\neg$ *c_pred** *(S,* n_{dest}))
and c_pred* *(S, n)) denotes all direct/indirect predecessors of n in S concerning*
edges $\in$ *CtrlE*]

For additive change operations, Theorem 1 presents precise conditions for efficient compliance checks with an estimated complexity of $O(n)$. These conditions base on a consolidated view of the reduced execution history $\mathcal{H}_{red}$. As an example take the insertion of activity `ultrasonic` as defined by change Δ in Fig. 4b. According to Theorem 1(a) it is only necessary to determine the marking of the successors of `ultrasonic` in S'. In our example, activity `report` is marked as `Activated` such that Δ can be applied to the WF instance depicted in Fig. 4a. To show their efficiency we have implemented several simulations checking our compliance conditions.

Besides, we explicitly deal with compliance issues in conjunction with data flow changes. We shortly summarize the basic ideas: Data elements can be always inserted, but must not be deleted if there was a read or write access on them. Read data edges $e_{read} = (n, d, read)$ on data element d can only be inserted or deleted iff activity n is marked as `NotActivated, Activated` or `Skipped`. Write data edges $e_{write} = (n, d, write)$ on data element d can only be inserted or deleted iff activity n has not been completed yet.

3.3 Concurrent Type and Instance Changes

Finally, we want to give an idea how the propagation of WF schema changes to biased WF instances can be managed correctly in the context of WSM-Nets [8].

But it should be clear that the following conclusions are applicable to other WF meta models as well. To meet a formal point of view, we first give a definition of a biased WF instance (compare Def. 1 and 2).

Definition 4 (Biased WF Instance). *A biased instance I is described by a tuple (S, Δ_I, $M^{S+\Delta_I}$, $Val^{S+\Delta_I}$, $\mathcal{H}$), where S denotes the WSM-Net from which I was created and Δ_I comprises ad-hoc changes op_I^1, ..., op_I^n that have been applied to I so far. WSM-Net $S_I := S + \Delta_I$, which results from the application of Δ_I to S, is called execution schema of I.*

Comparable to the already discussed correctness criteria we introduce a general criterion that allows us to argue about both – propagation of WF schema changes on "normal" (unbiased) and on biased WF instances. Obviously, when propagating a WF schema change Δ_S to a biased WF instance I we must not only consider its current state (i.e., marking $M^{S+\Delta_I}$) but we also have to cope with structural and semantic conflicts that may exist between the concurrent changes Δ_I and Δ_S. (Note that both, Δ_I and Δ_S have been based on S.) Due to lack of space we only consider structural conflicts in the following.

Correctness-Criterion 7 (Concurrent Changes) *Let S be a correct WSM-Net and I = (S, Δ_I, $M^{S+\Delta_I}$, ...) be a biased WF instance that was created from S. Let further Δ_S be a change operation, which transforms S into another correct WSM-Net S'. Then: Δ_S may be propagated to biased WF instance I :⇔*

1. *$S^* = (S + \Delta_I) + \Delta_S$ is a correct WSM-Net, i.e., Δ_S can be correctly applied to the execution schema $S_I = (S + \Delta_I)$.*
2. *I is compliant with S^*; i.e., the reduced execution history $\mathcal{H}_{red}$ can be produced on S^* as well. The marking M^{S^*} resulting from this is considered as a correct marking.*

Again, the challenging question is how to efficiently verify the conditions set out by Criterion 7. A naive solution would be to first generate the WSM-Net $S_I + \Delta_S$ 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 assignments, etc.) are concerned or Δ_S is to be propagated to a large collection of instances. Instead we must define appropriate and efficient rules for excluding potential conflicts (e.g., undesired cycles and deadlocks) between instance and type changes for as many instances as possible. Due to lack of space we abstain from further details.

4 Change Realization

We have now reached the stage of checking compliance of WF instances with a changed WF schema. This analysis leads to two instance categories – compliant and non-compliant WF instances [4,6]. We first discuss how the different approaches concretely carry out the migration of compliant WF instances

(*instance adaptations*). Then a short overview about approaches dealing with non-compliant WF instances is presented.

Approaches With True-Semantics: For Petri-Net based approaches, instance migration means to find a suitable marking on the changed net.

WF Nets: In [2], for each imposed change operation *transfer rules* are defined, which automatically adapt net markings. Concerning the insertion of sequences and alternative branches, the respective transfer rule maps marking s of the old net S to the identical marking on the new net S' (transfer rule is identity function $id : (S, s) \mapsto (S', s)$). As an example take the markings of the net in Fig. 2 before and after insertion of transition `ultrasonic`. For other change operations the insertion of additional tokens becomes necessary. Examples are changes like the insertion of new parallel branches or the deletion of alternative branches and sequences which contain tokens. Due to lack of space we abstain from discussing further transfer rules.

Flow Nets: For the change operations provided by [5,10], trivially, markings are adapted by constructing the SCOC (cf. Section 3.2). Note that the schema resulting from a SCOC always contains the marking of the old net.

Approaches With True/False-Semantics: To our knowledge neither $WASA_2$ [7] nor *WIDE* [4] provide detailed information about marking adaptations. In WIDE, however, follow-up markings may result from the replay of the execution history. As mentioned in Section 3.2, doing so is very expensive since execution histories often contain extensive data.

WSM-Nets: Our *ADEPT* approach is somewhat different regarding marking adaptations of compliant instances. To keep these adaptations efficient, we restrict them to those nodes and edges of the respective execution schema S_I, which constitute the context of the change region. Therefore, for each change operation *op* initial sets of nodes end edges to be re-evaluated are determined. Depending on the result of the evaluation the inspection of additional nodes and edges may become necessary. In addition, we benefit from well-defined marking rules as well as the way markings are represented (preserving markings of passed regions, True/False semantics). As an example take change Δ in Fig. 4. In the course of the following adaptation, `ultrasonic` has to be marked as `Activated` and marking of `report` is re-evaluated to `NotActivated`. Finally, an algorithm has been formulated, which evaluates instance markings with an estimated complexity of $O(n)$.

Dealing With Non-Compliant WF Instances: There are several approaches dealing with (temporarily) non-compliant instances [5,6]. *BREEZE* [6] provides a special graph construct which consists of compensation activities. With this, non-compliant instances are partially rolled back into a compliant state. The first approach which gives an idea of *delayed migration* is presented in the area of *Flow Nets* [5]. As an example consider Fig. 2. Even if the given instance passes through the old change region, a delayed migration to the new change region is possible when another loop iteration takes place. We have adopted this concept and suggest to keep such (temporary) non compliant instances *pending to migrate*.

5 Discussion

WF Nets: In [2], a wide range of change operations is covered and provided with correctness criteria and (automatic) transfer rules for adapting markings. In case of selected change operations (e.g., adding new parallel branches) new tokens can be automatically created when migrating instances. Though the authors completely abstract from data flow, for practical purposes it is necessary that tokens carry data flow information as well. Therefore, the semantics of newly inserted tokens at runtime is not always clear.

Flow Nets: [5,10] introduce a special class of changes, the described SCOC (cf. Section 3.2). Trivially, applying SCOC changes, marking adaptations are always correctly performed since the old change region is completely contained in the new net. For a special kind of changes – upsizing – the states of the old change region are mapped to states of the new change region by flow-jumpers (cf. Fig. 2b). [10] suggest to determine these flow jumpers manually which implies a very experienced WF designer. A very nice idea is offered by delayed migrations, i.e., the possibility of (temporarily) non-compliant instances to later migrate to the changes schema, e.g., when a loop back takes place. This approach gets even more complex when data flow issues are to be taken into account as well.

In *MWMS* [1] a special class of change operations is offered which provides correct migration of instances in safe states. Both the imposed WF model and the offered change operations are strongly restricted. To our knowledge, there is no detailed discussion about how to check the provided correctness criterion and how to adapt markings after instance migrations.

WASA$_2$ [7] suggests valid mappings between the purged WF instance graph and the changed WF schema in order to preserve correctness. Though [7] presents an implementation there is no detailed conceptualization of the mentioned valid mappings. In WASA$_2$ the loop problem is not present since only acyclic WF graphs are allowed. However, WASA$_2$ is one of the few approaches, which benefits from a concrete implementation of a powerful WF engine.

WIDE [4] has offered a cornerstone for many other approaches – the intuitive compliance criterion. Unfortunately, this criterion suffers from its restrictions concerning loops (cf. Section 3). Furthermore, it is not clear how the given criterion can be checked and implemented. If we had assumed that replaying the whole execution history is necessary this could not be efficiently realized. Generally, execution history logs are not captured in primary storage and contain extensive information like work assignments, time stamps, etc.

All discussed approaches – except our *ADEPT* approach – do not explicitly deal with data flow aspects. Furthermore, one of the few approaches to care about orthogonal aspects in conjunction with dynamic WF changes is offered by *BREEZE* [6]. Here, WF schema evolution in conjunction with time management is discussed. As already mentioned concurrent changes have been adressed only in the area of our *ADEPT* WSM-Nets so far.

Table 1 compares the discussed approaches along the stated criteria.

Table 1. A Comparison Of The Discussed Approaches

	WF Nets	*Flow Nets*	*MWMS*	*WASA$_2$*	*WIDE*	*ADEPT*
Completeness of						
• WF Model	−	○	−	−	+	+
• Changes	−	+	−	+	+	+
• Correctness Criteria	+	+	+	+	−	+
Checking Compliance	+	−	−	−	−	+
Change Realization	+	−	?	?	+	+
Available Implementation	?	+	?	+	?	+

6 Summary and Outlook

In this paper we have compared actual approaches dealing with adaptive workflows along fundamental criteria. Thereby the main focus lies on providing correctness criteria to decide whether a WF instance can be smoothly migrated to a changed WF schema or not. In many applications, the question how to efficiently check these criteria, how to accomplish instance migrations, how to implement the presented concepts, and how to offer change facilities to users remains unanswered. Therefore, we have presented simple state conditions for compliance checks and a nice solution to adapt instance markings after change propagation. Furthermore, we have discussed issues regarding concurrent changes. We strictly encourage other research groups to deal with this exciting problem as well and to provide implementations of their concepts within a powerful WF engine. There are many other interesting questions mainly concerning implementation of the presented concepts. Within this, questions related to change authorization, change analyses, and usability have to be carefully answered.

References

1. Agostini, A., De Michelis, G.: Improving flexibility of workflow management systems. In: Proc. BPM '2000. LNCS 1806, Springer (2000) 218–234
2. van der Aalst, W., Basten, T.: Inheritance of workflows: An approach to tackling problems related to change. Theoretical Computer Science **270** (2002) 125–203
3. Edmond, D., ter Hofstede, A.: A reflective infrastructure for workflow adaptability. Data and Knowledge Engineering **34** (2000) 271–304
4. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data and Knowledge Engineering **24** (1998) 211–238
5. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proc. Int'l Conf. on Org. Comp. Sys. (COOCS '95), Milpitas, CA (1995) 10–21
6. Sadiq, S., Marjanovic, O., Orlowska, M.: Managing change and time in dynamic workflow processes. Int'l J Coop IS **9** (2000)
7. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: Proc. 34th Hawaii Int'l Conf. on System Sciences (HICSS-34). (2001)

8. Reichert, M., Dadam, P.: ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. Journal of Intelligent Inf. Systems **10** (1998) 93–129
9. Kiepuszewski, B., ter Hofstede, A., Bussler, C.: On structured workflow modelling. In: Proc. CAiSE '00. LNCS 1789, Springer (2000) 431–445
10. Ellis, C., Keddara, K.: A workflow change is a workflow. In: Proc. BPM 2000. Volume 1806 of LNCS., Springer (2000) 516–534
11. Joeris, G., Herzog, O.: Managing evolving workflow specifications. In: Proc. Int'l Conf. on Coop. Inf. Systems (CoopIS '98), New York City (1998) 310–321
12. Fent, A., Reiter, H., Freitag, B.: Design for change: Evolving workflow specifications in ULTRAflow. In: Proc. CAiSE '02. (2002) 516–534
13. Kradolfer, M., Geppert, A.: Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In: Proc. CoopIS '99, Edinburgh (1999) 104–114
14. Ellis, C., Maltzahn, C.: The Chautauqua workflow system. In: Proc. 30th Int'l Conf. on System Science, Maui (1997)