Accelerating Workflows with Fixed Date Constraints

Martin Bierbaumer, Johann Eder, and Horst Pichler

Institute for Informatics-Systems, University of Klagenfurt, Austria
`bierbaumer@uni-klu.ac.at, eder@uni-klu.ac.at, pichler@uni-klu.ac.at`

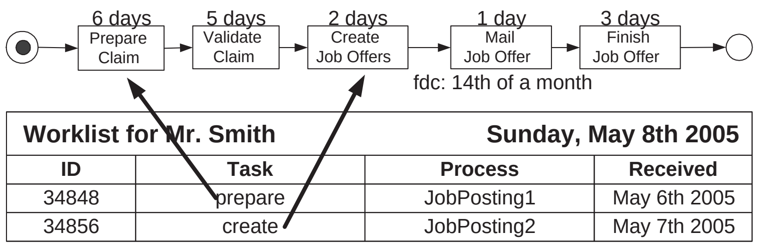
Abstract. Workflow systems execute workflows and assign work items to the work list of participants. As work lists usually hold multiple work items, participants have to decide which work item to handle next. When selecting a specific work item other work items must be postponed, which will in succession delay their appendant workflows. This may lead to disproportionately increased execution durations and turn around times if fixed-date constraints are defined on succeeding tasks. We propose a probabilistic method which assists the participant when deciding which work item to handle next, with the intention to decrease turnaround times and to avoid time-related escalations, by providing information about the delay to expect when postponing tasks.

Keywords. Workflow management systems, time plans, time management, process monitoring and tracking.

1 Introduction

Workflow systems execute workflows and assign tasks or work items to participants according to activities defined in a workflow model, i.e. an activity is assigned if all preceding activities are finished. Participants use a work list to manage the work items assigned to them. Usually, they have to decide which work item to handle next. The decision to work on a particular work item implicitly holds the decision to postpone every other work item in the work list. This may have grave effects on the execution duration of the workflows to which these postponed work items belong. Common policies for this decision problem, like first-in first-out (FIFO) or earliest-deadline-first, may be suboptimal because they do not take into account that a) the postponement of a task may not immediately delay a workflow due to eventually existing buffer times, and b) even a slight postponement may lead to a disproportionately high delay due to fixed-date constraints on succeeding tasks (e.g. 'a task must be finished until the 1st of a month').

Consider the workflow *JobPosting* to announce open positions of a big company in the local newspaper, as visualized in Fig. 1: A department initializes the process by generating a claim. At first the claim will be forwarded to the



Worklist for Mr. Smith — **Sunday, May 8th 2005**

ID	Task	Process	Received
34848	prepare	JobPosting1	May 6th 2005
34856	create	JobPosting2	May 7th 2005

Fig. 1. Scenario1: job posting workflow

personnel division where it is *prepared* for further processing. Then the claim is *validated* by the personal manager. After this, a job offer for the open position is *created*. Then the offer is *mailed* to the newspaper. Finally the job offer is *finished* (filing, notification of departments, etc.). The expected duration in days is displayed on top of each task. Additionally, as the newspapers special "Job & Career" edition appears only once a month (on each 15th) a corresponding fixed-date constraint (fdc) has been defined on the last activity *mail*, demanding that it must be finished on the 14th of a month. Currently, on Sunday May 8th 2005, Mr. Smith, employed in the companies personnel division, has two work items from two different JobPosting-processes in his work-list. For sake of simplicity, we assume that every day is a working day, Saturdays and Sundays included.

Scenario 1.a) According to the FIFO-policy, suggested by the work-list client, he starts to execute the first work item, the task *prepare* of process *JobPosting1*, which will presumably take 6 days. Although the execution of *prepare* starts immediately, the job offer will, according to the expected task durations, not appear in the May-issue, as the *mail*-task will presumably be finished on May 21st and the process will be finished at May 24th (for easier comprehension please refer to a calendar of 2005). Unfortunately, the decision to execute *prepare* first implicitly postpones the execution of his second work item, the task *create* of process *JobPosting2*, for 6 days, to May 14th. *create* can be finished on May 16th and the next step *mail* on May 17th. Thus the job offers of the second process will, presumably, also not appear in this month special issue.

Scenario 1.b) If Mr. Smith had chosen to execute the second work-item first, the *mail*-task of process *JobPosting2* could have been finished on time, until May 11th. Thus the FIFO-policy unnecessarily delays the second process for 30 days. Although this decision postpones the first work-item *prepare*, the *JobPosting1* process will not be delayed, as it will also end at May 24th (which is the same end date as in Scenario1). Due to the fixed-date constraint defined on *mail*, enough buffer time exists to compensate the postponement! The scenarios demonstrate that an intelligent and predictive selection of work-items can help to decrease the turn-around times of processes.

In this paper we introduce a method which exploits knowledge about the workflow structure and time properties, to assist workflow participants when deciding which work item to handle next, in order to decrease turnaround times and to avoid time-related escalations, by providing information about the delay to expect when selecting or postponing tasks.

The paper is structured as follows: In Section 2 we present related work. Section 3 introduces the probabilistic timed workflow model. In Section 4 we examine the relationship between delay and postponement and introduce delay tables. In Section 5 we add the notion of probabilistic delay tables in order to deal with conditional workflow execution, including a presentation of structural workflow transformations [9]. Section 6 explains how to interpret probabilistic delay tables by using delay time histograms. Finally Section 7 concludes the paper with a brief outlook on our future research topics.

2 Related Work

When dealing with time management it seems to be straightforward to apply existing and extensively examined techniques from related areas (like project management) on workflows. Unfortunately some complex issues originating from workflow modelling and instantiation complicate this intention [2, 13]. For instance workflow systems typically do not compute schedules due to the partial knowledge they have about the execution of their processes and the impossibility to know the actual flow at decision points at process instantiation time.

Several publications introduced methods which cope with these workflow specific problems, e.g. [4, 5, 7, 11, 12]. They propose the calculation of time plans which define execution intervals for each activity in a workflow, such that deadlines will not be violated. The concepts presented are inspired by PERT and CPM charts which are commonly used in project management. But all of them suffer from the uncertainty about time and branching information which arise when tasks are executed conditionally. Some newer publications already utilize stochastic information which are used for performance analysis [1, 10] or scheduling issues [3]. We dealt with this problem by introducing probabilistic time management as described in [6].

How to handle different types of time constraints is explained in [7, 11]. Although most of the above mentioned publications deal with similar time related problems, as far as we know, currently no publication exists which examines the relation between postponement and delay. The approach presented in this paper follows the basic motivation of [8]: to assist the workflow participant with supportive work-list tools.

3 Probabilistic Timed Workflow Model

We define a full-blocked workflow model that we use in the rest of this paper. Workflows are represented as directed acyclic graphs (see Figures 1, 3 and 5). Edges determine the execution sequence of nodes, thus a successor can start if its predecessor(s) are finished. A node is identified by a unique name and can be of type *activity* (represented by boxes), which corresponds to individual tasks of a business process, or a control node of type *start*, *end*, *and-split|join* or *or-split|join*. Conditional branches are augmented with statistically weighted branching probabilities (defined by estimations or empirically generated values from the workflow log). Routes after an *and-split* will be processed concurrently. They will be synchronized at the according *and-join*, which does not proceed until all predecessor nodes are finished. After an *or-split* only one route, depending on a run-time evaluated condition, will be selected. The corresponding *or-join* proceeds if one predecessor node is finished (the one on the route selected at the or-split). A workflow model is *full-blocked* if the graph is restricted to proper nesting of splits and joins. Therefore and-structures and or-structures may be nested, but must not overlap (according to the conformance-class declaration in [14]).

N designates the node named N and $N.type$ yields the type of the node. $M \rightarrow N$ designates an edge between two nodes M and N, and $p_{M \rightarrow N}$ yields the branching probability for edges going out from or-splits. $N.Pred$ defines the set of adjacent predecessor nodes and $N.Succ$ defines the set of adjacent successor nodes.

Additionally, *time properties* have to be defined during the modelling process of the workflow. Each node N is augmented with the expected *duration* $N.d$ in an arbitrary time unit like hours or days. Control nodes like *start* or *end* usually have a duration of 0. The workflows overall execution duration is limited by a workflow *deadline* δ, which is defined as a time value relative to the start of the workflow (note that the definition of δ is compulsory!).

Additionally the execution of a node can be constrained to certain dates by a *fixed-date constraint*. Although it is basically possible to assign a fixed-date constraint to either the start or the end of a node, we restrict our explanations to the end of a node (to avoid the need of describing analogous concepts). A fixed-date constraint $fdc(N, F)$ expresses that the end of node N can only occur on certain dates specified by the *fixed-date object* F, where $F.valid(Date)$ returns true if the arbitrary date is valid for F. $F.next(Date)$ and $F.prev(Date)$ return the next and previous valid dates after $Date$. Assume that a fixed-date object f may for example be defined as a list of valid dates $f =$ *(12th of March, 17th of June, 25th of September)* or as an expression like $f =$ *every 3rd sunday starting with 6th of September.*

4 Calculation of Delay

4.1 Earliest Possible End

Based on the workflow structure and time properties, it is possible to calculate the *earliest possible end (EPE)* for each node, relative to the start time of the workflow. It depicts the earliest possible point in time an activity may end, defined by the sum of durations of preceding nodes.

The EPE of a node is basically the sum of durations of all which are to be executed nodes before it, starting from the current node Cur and at the current

Algorithm 1 Calculation of Earliest Possible End Times

```
1: $Cur.epe := now + Cur.d$
2: for every $N$ succeeding $Cur$ in forward top. order do
3:     $N.epe := maxPredEPE(N) + N.d$
4:     if if $fdc(N, f) \in FDC$ then
5:         $N.epe := f.next(N.epe)$
6:     end if
7: end for
```

time *now*. Additionally, as and-joins demand that every preceding node must be finished before the execution can be continued, the EPE of an and-join is determined by the maximum EPE of all adjacent preceding nodes. For this we define the operation $maxPredEPE(N)$, which yields the maximum EPE of all predecessors of and-joins or the EPE of the single predecessor of other node-types. The EPE of every node, which is a (transitive) successor of the current node Cur, is calculated as described in algorithm 1. How to deal with EPEs for or-joins is explained in section 5.

Consider the workflow from scenario 1 with *Cur* = *prepare* and *now* = $may8$ (depicting the current time 8th of May 2005) with the EPEs: *prepare.epe* = $may14$, *validate.epe* = $may19$, *create.epe* = $may21$ and *mail.epe* = $may22$. According to the fixed-date constraint defined on *mail*, the EPE must be shifted to the next valid date adhering to $fdc($*mail*,14th of a month), yielding *mail.epe* = $jun14$. Then we calculate $finish.epe = jun17$. Finally, as control nodes like *End* have a duration of 0, the earliest possible end of the workflow is *End.epe* = $jun17$.

There are two special cases to consider: a) As f may define a finite set of fixed dates, it may yield no valid next date at *f.next(N.epe)*. Then the user or an administrator must be informed that a future deadline or fixed-date violation will (presumably) occur, and a further delay due to a postponement of Cur has to be avoided. b) Due to and-structures many activities may currently be executed in parallel. Thus more than one current activity Cur may exist. In this case all current activities must be initialized as in line (1). The algorithm still works, as due to the full-blocked structure of the workflow graph all parallel routes will be joined at a succeeding and-join. Additionally, some participants may already have started to execute their current tasks. In this case the duration of the node must be adapted by applying *Cur.d* = *Cur.d* - *(now* - *Cur.start*), where *start* denotes the actual start time of the nodes execution.

4.2 Delay Tables

A *postponement* designates the shift of the execution of an activity, which is ready to be executed by a participant, to a later point in time. A *delay* is generated if due to a postponement the earliest possible end of the workflow (= *End.epe*) would be increased.

In our running example with $now = may8$ and $Cur = prepare$ we additionally define an overall workflow deadline of $\delta = 90$, therefore the workflow (or the node End) must be finished until $now + \delta = aug6$. The examination of dependencies between delay and postponement must start at the last node. One can see, that the workflow can not end before $End.epe = jun17$ and must not end after $now + \delta = aug6$. Thus, the following can be stated (for End): if it is not postponed it will not be delayed and end at $jun17$, if it is postponed by 1 day it will be delayed by 1 day and end at $jun18$, and so on. Finally we can state, that if it is postponed by 50 days it will be delayed by 50 days and end at $aug6$. A further postponement will lead to a violation of the workflow deadline $now + \delta = aug6$.

Based on this knowledge it is possible to define a relation between a *delay deadline (d)* and a *delay time (t)*, where t is the delay of the workflow to expect, if the end of the node is postponed to d. This relation is captured in the *delay table (DT)*. To query delay tables we introduce the operation *delay time selection*, which yields the delay time for a given delay deadline.

Definition 1 (Delay table, DT) *A* delay table $N.dt$ *for a node N is a set of tuples (d, t) describing the delay time t to expect if N ends at the delay deadline d. Furthermore a delay table is called* valid *if each t and each d is unique in the set. The operation $y = selectT(dt, x)$ applied on a valid delay table selects the delay time y for a delay deadline x, such that:*

$$y = \begin{cases} z \text{ if } \forall (d,t) \in dt, \text{ where } x \geq d\text{: } z \leq t \\ \infty \text{ if } \forall (d,t) \in dt\text{: } x > d \end{cases}$$

The valid DT of End is $End.dt$={*(jun17,0),(jun18,1),(jun19,2), +++,(aug6,50)*}. The $+++$ between two tuples (d_1, t_1),$+++$,(d_n, t_n) symbolize that the set holds additional tuples (d_i, t_i), such that $d_i = d_{i-1} + 1$ and $t_i = t_{i-1} + 1$ for $1 > i > n$. The operation *selectT* yields the delay time for a given delay deadline. If no tuple with the requested delay deadline exists, the delay time of the tuple with the next greater delay deadline is selected: e.g. *selectT(End.dt, jun21) = 3* states that the workflow will be delayed by 3 days if the node End ends at $jun21$. The explicit definition of a result of ∞ is necessary as a point in time greater then the maximum delay time in the DT may be selected. If postponed to that point in time a workflow deadline violation would occur: e.g. *selectT(End.dt, sep12)* $= \infty$ states that if it ends at $sep21$ the workflow deadline will be violated.

As the relation between d and t in $End.dt$ is linear, we call it a *linear DT*. To calculate the always linear DT of an end node we use the following operation: $End.dt = linearDT(End.epe, now + \delta)$.

Definition 2 (Calculate linear DT) *A linear DT dt, delimited by a lower bound lb and an upper bound ub, is generated as follows:* $dt = linearDT(lb, ub) = \{(d,t) \mid \forall d : lb \leq d \leq ub, t = d - lb\}$, *where* $|dt| = ub - lb + 1$.

For nodes, which are to be executed in a sequence, the DT of a predecessor node is calculated by subtracting the duration of the successor from the DT of the successor:

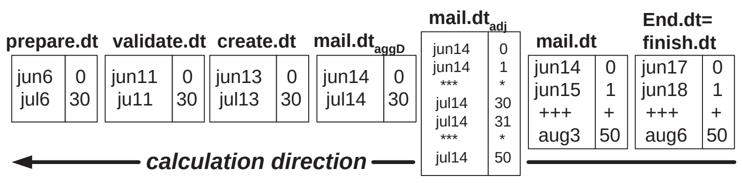


Fig. 2. Delay tables for Scenario1

Definition 3 (Subtract scalar from DT) *The scalar k is subtracted from a DT dt_1 resulting in a DT dt_2 as follows: $dt_2 = dt_1 - k = \{(d-k, t) \mid (d,t) \in dt_1\}$.*

Therefore *finish.dt = End.dt - 0* and *mail.dt = finish.dt - finish.d* (see also Fig. 2). As a fixed-date constraint *fdc(mail,f)* exists, the end of *mail* may only occur on valid dates, defined by the fixed-date object *f=14th of a month*. Thus the delay deadlines must be adjusted using the *previous* function of the fixed-date object.

Definition 4 (Adjust delay table to fixed-date object) *A delay table dt is adjusted to a fixed date object f, resulting in a delay table dt', as follows: $dt' = adjust(dt, f) = \{(f.prev(d), t) \mid (d, t) \in dt\}$.*

The operation yields the $mail.dt_{adj}$ = *(adjust(mail.dt,f))* = {*(jun14,0),(jun14,1), *** ,(july14,30),(july14,31), *** , (july14,50)*}. The ***** between two tuples (d_1, t_1),*****,(d_n, t_n) symbolize that the set holds additional tuples (d_i, t_i), such that $d_i = d_1$ and $t_i = t_{i-1} + 1$ for $1 > i > n$. As one can see this operation results in an invalid DT with multiple identical delay deadlines (but different delay times). In order to receive a valid DT we have to remove all tuples with identical delay deadlines, but the one with smallest delay time. To achieve this the operation *delay deadline aggregation* must be applied on $mail.dt_{adj}$.

Definition 5 (Delay deadline aggregation) *The operation $aggD(dt_1)$ yields an aggregated delay table dt_2, such that $dt_2 = aggD(dt_1) = \{(x, minT(dt_1, x)) \mid (x, t) \in dt_1\}$; where $y = minT(dt_1, x)$ selects the minimum delay time y for a delay deadline x of the delay table dt_1, such that $\forall(x, t_1) \in dt_1$: $y \leq t_1$ must hold.*

Now we apply $aggD$ on the intermediary DT $mail.dt_{adj}$: $mail.dt_{aggD}$ = $aggD(mail.dt_{adj})$ = {*(jun14,0),(jul14,30)*}, which is interpreted as: if *mail* ends at $jun14$ or before, the delay will be 0; if it ends after $jun14$ and before or exactly at $july14$ the delay will be 30. And the next possible end, according to the fixed-date constraint, is $aug14$. But if *mail* would end at $aug14$, the workflow deadline would be violated. Thus the last allowed delay deadline (or end time) is *jul14*. The operation which combines $aggD$ and $adjust$ is defined as follows:

Definition 6 (Apply fixed-date object on delay table) *The fixed-date object f is applied on a delay table dt_1 yielding a delay table dt_2, such that $dt_2 = applyFDC(dt_1, f) = aggD(adjust(dt_1, f))$.*

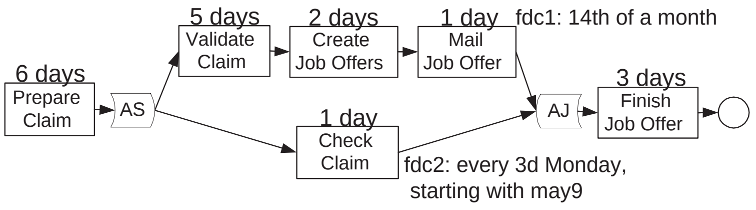


Fig. 3. Scenario2: and-structure

Afterwards we proceed by subsequently subtracting the durations *create.d=2* and *validate.d=5*, finally yielding *prepare.dt=*{*(jun6,0),(jul6,30)*}. The DT of the current activity *prepare* can be used to answer participant-questions like "What is the expected delay if the end of *prepare* is postponed to *jun25*?". The answer is *selectT(prepare.dt, jun25) = 30*: "The delay will be presumably be 30 days!". And for *selectT(prepare.dt, aug8)* $= \infty$ the answer would be: "A deadline violation will occur!". The DT may also be used to inform the participant (Mr. Smith) that he may postpone the end of *prepare* to *jun*6 without delaying the workflow, which is defined by the tuple with the minimum delay deadline or the tuple with a dely time of 0 respectively.

4.3 Calculation of DTs for And-Splits

We adapt the example, as visualized in Fig. 3: The company decides that the management's staff unit must be informed about every claim. Additionally a fixed-date constraint is attached to the new activity *check* (which takes 1 day), as the staff unit team only meets every 3d monday (starting with may 9th) to discuss and check new claims. Assume that the semantics of *finish* is now: finally the job offer and the claim are finished together (filing, notification of departments, etc.). Again *Cur=prepare, now=may8* and δ*=90*. At first the EPEs must be calculated according to the algorithm presented in Section 4.1: *Start.epe=may8, prepare.epe=may14, AS.epe=may14, validate.epe=may19, create.epe=may21, mail.epe=jun14, check=may30, AJ.epe=jun14, finish.epe=jun17* and *End.epe=jun17*. The EPE of the new activity *check* has been adjusted to the next valid date *may*30 (= *may*9 + 3weeks) according to its fixed-date constraint. And the EPE of the and-split AJ is equal to the maximum EPE of its predecessors (*mail* and *check*), which is *mail.epe*.

The calculation of DTs starts with the initialization of the last node *End.dt = linearDT(End.epe,now +* δ*)*, followed by the subsequent calculation of $finish.dt$ and *AJ.dt* (see also Fig.4). The DTs of nodes which are predecessors of an and-split are calculated like nodes in a sequence: *mail.dt = AJ.dt – AJ.d* and *check.dt = AJ.dt – AJ.d*. Since *AJ.d=0*, their DTs are equal to *AJ.dt*. For the upper parallel route we refer to the calculations of Scenario1, as the DTs are equal. For the single node *check* of the second route we have to apply its fixed-date

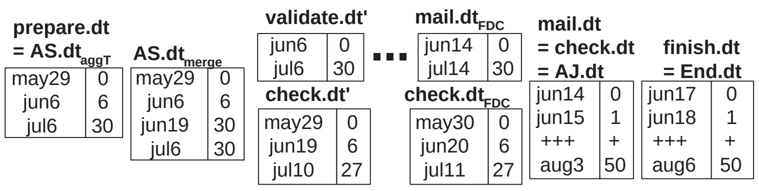


Fig. 4. Delay tables for Scenario2

constraint *applyFDC(check.pdt,every 3d monday starting with may9))*, resulting in $check.dt_{fdc}$ (to save some space we did not show the intermediate results $check.dt_{aggD}$ and $check.dt_{adjust}$).

Before combining DTs of and-split successors, their durations must be subtracted (resulting in an intermediate dt'): *validate.dt'*={*(jun6,0),(jul6,30)*} and *check.dt'* = {*(may29,0),(jun19,6),(jul10,27)*}. To combine two DTs the operation *delay table conjunction*, which is a combination of the operations *merge* and *aggT*, has to be applied.

Definition 7 (Merging two delay tables) *The operation* merge(dt_1,dt_2) *applied on two delay tables* dt_1 *and* dt_2*, results in a delay table* dt *as follows:* dt = merge(dt_1, dt_2) = {(d,t) | d ∈ D, selectT(dt_1,d) ≠ ∞, selectT(dt_2,d) ≠ ∞, t = max(selectT(dt_1,d),selectT(dt_2,d))}*, where* D = {d_1 | (d_1,t_1) ∈ dt_1 } ∩ {d_2 | (d_2,t_2) ∈ dt_2}.

The merge-operation *AS.dt = merge(validate.dt',check.dt'* selects the greatest delay time (as all routes are processed in parallel) for every possible delay deadline, which is accomplished as follows: first a set D which holds all delay deadlines of both sets is generated, which is for our example *D*={*may29,jun6,jun19,jul6,jul10*}. Then for every delay deadline in D the *selectT*-operation is applied on both DTs and the maximum of the results is selected, which is 0 for $may29$, 6 for $jun6$, 30 for $jun19$ and 30 for $jul6$: $AS.dt_{merge}$ = {*(may29,0),(jun6,6),(jun19,30), (jul6,30)*}. The delay deadline $jul10$ has been filtered out by the condition *maxT(*dt_1*,d)* ≠ ∞ as *maxT(validate.dt,jul10)* = ∞. The intermediate DT is invalid as it holds two tuples with equal delay times. To receive a valid DT it is necessary to remove the tuple with the earlier delay deadline by applying the delay time aggregation.

Definition 8 (Delay time aggregation) *The operation* $aggT(dt_1)$ *yields an aggregated delay table* dt_2*, such that* $dt_2 = aggT(dt_1) = \{(maxD(dt_1, x), t) \mid (x, t) \in dt_1\}$*; where* $y = maxD(dt_1, x)$ *selects the maximum delay deadline* y *for a delay time* x *of the delay table* dt_1*, such that* $\forall (d_1, x) \in dt_1 : y \geq d_1$.

According to this $AS.dt_{aggT} = aggT(AJ.dt_{merge})$ = {*(may29,0),(jun9,6), (jul9,30)*}. And finally, due to *AS.d = 0*, the DT of the current activity *prepare* is *prepare.dt* = $AS.dt_{aggT}$ *- 0*. Based on the operations *merge* and *aggT* we define the *delay table conjunction.*

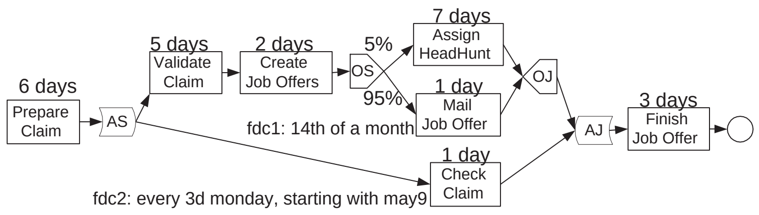


Fig. 5. Scenario3: or-structure

Definition 9 (Delay table conjunction) *The delay table conjunction* $dt = dt_1 \wedge dt_2$ *applied on two delay tables* d_1 *and* d_2 *yields a delay table* dt*, such that* $dt = aggT(merge(dt_1, dt_2))$.

Propositions: The conjunction is commutative and associative as $dt_A \wedge dt_B = dt_B \wedge dt_A$ and $(dt_A \wedge (dt_B \wedge dt_C)) = ((dt_A \wedge dt_B) \wedge dt_C)$. Thus it can be extended to any number of DTs: $dt_1 \wedge ... \wedge dt_n = \bigwedge_{i=1}^{n} dt_i$.

5 Calculation of Probabilistic Delay

One fundamental core problem has not been addressed so far: in workflows with or-splits multiple different execution routes (also called *instance types*) are possible. Consider the workflow in Fig. 5: The company decides to assign a head hunter for open high-management positions, instead of advertising in the newspaper. Selection of and negotiation with a head hunter takes about 7 days. According to past experiences about 5% of all job offers address high-management positions. No fixed-date constraint is assigned to this task. In this graph two instance types can be identified, as two different execution sequences of activities and nodes are possible: the first one via activity *assign* and the second one via activity *mail*. Note that different instance types are only produced if the graph contains or-structures (after and-splits all nodes will be executed unconditionally).

5.1 Unfolding the Graph

When calculating the EPEs a problem arises at or-joins: *OJ.epe* can not be determined unambiguously, as the EPE depends on the path executed prior to OJ. Which one this will be, is unknown at the current node Cur. The only thing we can tell is, that it is likely that the route via *assign* will be executed with a probability of 5% and the route via *mail* with a probability of 95%. As the algorithm to calculate EPEs demands exactly one EPE per node, we developed the following solution.

At first we *unfold* the workflow, in order to split up different instance types. [9] developed a *backward unfolding procedure*, which successively applies basic

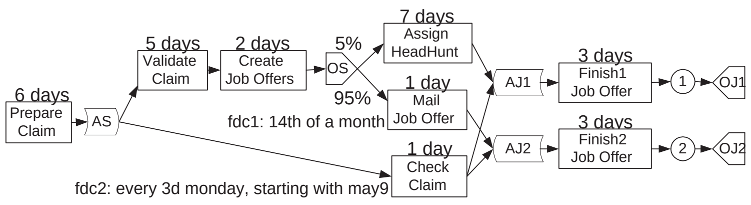


Fig. 6. Unfolded workflow graph

transformation operations on an originally block-structured graph. A transformation operation results in a new workflow graph, which is semantically equal to the old one, based on the notion of equivalence of tasks and identical execution order. The result of the backward unfolding procedure is a graph where every or-join has been moved to the end.

Fig. 6 shows the unfolded version of the graph from Fig. 5. Three basic transformation operations have been applied to move the or-join to the end of the workflow: first the or-join has been moved over the and-join, which resulted in a duplication of nodes and edges succeeding the or-join. The second and third operation moved the node over the activity $finish$ and end node towards the end of the workflow. It is important to notice that the possible order of node-execution is equal to the one in the original graph. A backward unfolded graph will have as many end-nodes as possible routes through the graph exist, which is one for each instance type. Furthermore, as synchronizing or-joins have been moved to the end, it enables us to calculate exactly one earliest possible end time for each node, by applying the algorithm of section 4.1. For further details on graph transformations see [9].

5.2 Calculation of Execution Probabilities

As mentioned above, the calculation of DTs starts at the end node of a graph, but in the unfolded graph multiple end nodes have to be considered. Therefore the initial DTs of end-nodes must be weighted according to their execution probability, in relation to the current activity Cur. This can be accomplished by calculating the execution probability dependent on predecessor execution probabilities and the type of nodes. For this we have to traverse the graph in a forward topological order starting with the current node (see algorithm 2). As the current node will be executed in any case, its execution probability is initialized with $Cur.x = 1$. The calculation finally yields $End1.x = 0.05$ and $End2.x = 0.95$, which means, that starting from the current node Cur, the end-node $End1$ will be executed with a probability of 5% and the end-node $End2$ with a probability of 95%. The sum of probabilities of all end nodes must be 1.

Algorithm 2 Calculation of execution probabilities

```
 1: $Cur.x = 1$
 2: for every successor $N$ of $Cur$ in forward top. order do
 3:     if $P \in N.Pred$ and $P.t$ = or-split then
 4:         $N.x = P.x * p_{P \rightarrow N}$
 5:     else if $N.type$ = and-join then
 6:         $N.x = minimum\ of\ all\ P.x,\ P \in Pred_N$
 7:     else
 8:         $N.x = P.x, P \in Pred_N$
 9:     end if
10: end for
```

5.3 Probabilistic Delay Tables

With the unfolded graph and execution probabilities for end-nodes it is possible to calculate a set of *probabilistic delay tables (PDTs)* for the current node, which can be queried in several ways. Each node N of an unfolded graph can hold multiple weighted DTs, one for each end-node that is reachable from N (or each possible instance type starting from N), which are stored in the set $N.PDT$.

Definition 10 (Set of Probabilistic Delay Tables, PDT) *The set of* probabilistic delay tables *$N.PDT$ of a node N is a set of tuples (e, p, dt), where dt is a delay table, e is the end-node from which dt originated and p is the execution-probability of the end-node.*

For the PDT-calculation algorithm the operations defined on DTs must be adapted to set-based PDT-operations. Details will be explained in the subsequent section, along with the calculation of PTDs for our running example:

Definition 11 Operations on PDTs

- *Subtract scalar: PDT - k = $\{$(e,p,dt-k) $|$ (e,p,dt) $\in$ PDT$\}$*
- *Apply fixed-date constraint: applyFDC(PDT,f) = $\{$(e,p,applyFDC(dt)) $|$ (e,p,dt) $\in$ PDT$\}$*
- *Conjunction: $PDT_1 \wedge PDT_2$ = $\{$(e,p,$dt_1 \wedge dt_2$) $|$ (e,p,dt_1) $\in$ PDT_1, (e,p,dt_2) $\in$ $PDT_2\}$.*

Propositions: The conjunction is commutative and associative as $PDT_A \wedge PDT_B = PDT_B \wedge PDT_A$ and $(PDT_A \wedge (PDT_B \wedge PDT_C)) = ((PDT_A \wedge PDT_B) \wedge PDT_C)$. Thus it can be extended to any number of PDTs: $PDT_1 \wedge ... \wedge PDT_n = \bigwedge_{i=1}^{n} PDT_i$.

5.4 Calculation of PDTs

Before calculating the PDTs, the graph must be unfolded and or-joins must be deleted, followed by the forward calculation of execution probabilities and EPEs, resulting in: *End1.epe=jun2, End2.epe=jun17, End1.x=0.05* and *End2.x=0.95.*

Algorithm 3 PDT calculation

```
1: for all nodes $N$ in a backward topological order do
2:   if $N.type = end$ then
3:     $N.PDT=\{(N, N.x, linearDT(N.epe, now + \delta))\}$
4:   else if $N.type = and\text{-}split$ then
5:     $N.PDT = \bigwedge S.PDT', \forall S \in N.Succ$
6:   else
7:     $N.PDT = \bigcup S.PDT', \forall S \in N.Succ$
8:   end if
9:   if $\exists fdc(N, f) \in FDC$ then
10:    $N.PDT' = applyFDC(N.PDT, f) - N.d$
11:  else
12:    $N.PDT' = N.PDT - N.d$
13:  end if
14:  if $N = Cur$ then
15:    return
16:  end if
17: end for
```

The calculation of $PDTs$ is again performed in a backward topological order, where the PDT of each node is determined according to its node type (see algorithm 3 and Fig. 7). Note, that we use the intermediary $N.PDT'$ for each node N, which contains the PDT after fixed-date adjustment and subtraction of node duration .

The algorithm starts with the initialization of all end-nodes End_1 and End_2 as defined in line 2 of the algorithm: $End_i.PDT = \{ (End_i, p_i, linearDT(End_i.epe, now + \delta))\}$. The PDT of an end-node contains exactly one tuple with a reference to the originating end-node End_i (in this case the node itself), the execution probability of this end-node $End_i.x$, and the initial linear delay table, calculated as $linearDT(End_i.epe, now + \delta)$. Since no fixed-date constraint on End_i exists (line 9), the intermediary PDT is calculated as $End_i.PDT' = End_i.PDT - End_i.d$ (line 12). Due to $End_i.d = 0$, this results in $End_i.PDT' = End_i.PDT$.

In the next iteration the PDT of the preceding node is determined. Since $finish_i$ is no end-node and no and-split, the PDT is calculated as union of all successor-PDT' (line 7). As $finish_i$ has only one successor, the PDTs are equal: $finish_i.PDT = End_i.PDT'$. Then the duration of $finish_i.d = 3$ is subtracted, yielding $finish_i.PDT'$ (line 12).

The calculation of PDTs for nodes between $finish_i$ and OS are calculated analogously, as only one possible route to an end-node has to be considered and therefore the *union*-operation always results in a set with one tuple. At *mail* the fixed-date constraint has to be applied before subtracting the duration (line 10), yielding the same DT as in scenario 2.

So far, each determined PDT holds exactly one tuple: e.g. $assign.PDT' = \{(e,p,dt)\}$ states that the delay information stored in dt originates from the end-node $e=End_1$, which has an execution probability of $p=0.05$. The or-split OS has two successors, therefore $check.PDT = assign.PDT' \cup mail.PDT'$ (line 7). This

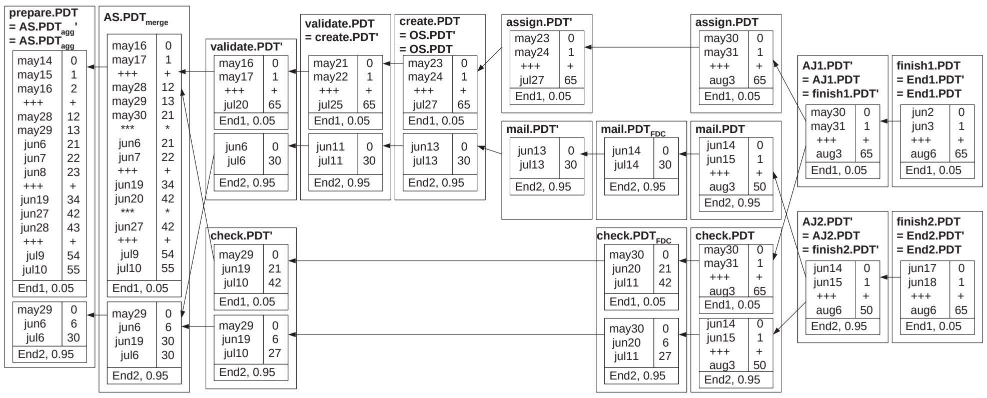

prepare.PDT = AS.PDT_agg' = AS.PDT_agg
AS.PDT_merge
validate.PDT'
validate.PDT = create.PDT'
create.PDT = OS.PDT' = OS.PDT
assign.PDT'
assign.PDT
AJ1.PDT' = AJ1.PDT = finish1.PDT'
finish1.PDT = End1.PDT' = End1.PDT
mail.PDT'
mail.PDT_FDC
mail.PDT
check.PDT'
check.PDT_FDC
check.PDT
AJ2.PDT' = AJ2.PDT = finish2.PDT'
finish2.PDT = End2.PDT' = End2.PDT
End1, 0.05
End2, 0.95

Fig. 7. Calculation of probabilistic delay tables for Scenario3

results in a set with two tuples holding information about two different delay tables which originate from two different end-nodes (with different execution probabilities). Thus we can state, that the number of tuples in the PDT of a node is equal to the number of instance types containing this node. The same can analogously be stated for the node *check*, as it also has two successors leading to two different end-nodes. We proceed with the calculation of *validate.PDT'* (subtract of duration) and *check.PDT'* (apply fdc and subtract duration)

At the and-split a PDT-conjunction has to be applied on successor-PDTs: *validate.PDT'* $\wedge$ *check.PDT'*. The PDT-conjunction basically applies a regular DT-conjunction on all DTs with identical end-node references (they belong to the same instance type)! Therefore it consists of two steps: 1) first it merges all DTs, which originate from the same end-node, into one DT_{merge} (see PDT_{merge} in Fig. 7). 2) Subsequently, to remove duplicate delay times, it aggregates each intermediate DT_{merge} by applying an aggregation on each DT. Note that PDT_{merge} and PDT_{agg} are displayed for illustration issues only, as according to its definition the PDT-conjunction does not produce intermediate results. Finally, the PDT for the current activity *prepare* is calculated.

6 Interpretation and Application

PDTs are used to generate information about the delay to expect if a workflow participant postpones the end of his current task to a given date. The answer to the question "What is the expected delay, if the end of *prepare* is postponed to $june1$?" can no longer yield an ambiguous result, as the PDT of prepare holds two DTs with different weights. Therefore we introduce *(cumulated) delay time histograms.*

Definition 12 (Delay Time Histograms) *The cumulated delay time histogram* N.CDTH$_x$ *for a node* N *and a date* x, *based on the delay time histogram* N.DTH$_x$

$= \{(\Sigma\mathrm{p},\mathrm{selectT}(\mathrm{dt},\mathrm{x})) \mid (\mathrm{e},\mathrm{p},\mathrm{dt}) \in \mathrm{N.PDT}\}$*, is a set of tuples* $(\mathrm{c}_i,\mathrm{t}_i)$ *such that* $\mathrm{c}_i = \sum_{t_j \leq t_i} \mathrm{p}_j$, where $(\mathrm{p}_i,\mathrm{t}_i) \in \mathrm{I}$.

The DTH contains one tuple with delay time and probability for each DT in the PDT, selected for a given delay deadline *x*: *prepare.DTH*$_{jun1}$ = {*(0.05,21), (0.95,6)*}. The expression *Σp* aggregates tuples with equal delay times by adding up their probabilities. For the CDTH the probability values of the DTH are cumulated in increasing order of delay times: *prepare.CDTH*$_{jun1}$ = {*(0.95,6), (1.0,21)*} . It is interpreted as "With a probability of 95% the maximum delay will be 6 days, and with a probability of 100% the maximum delay will be 21 days!". For huge PDTs containing hundreds of weighted DTs, answers like the ones above will be of no use for the participant. Therefore we introduce the selection for a defined minimum probability.

Definition 13 (Selection of probabilistic delay time) t = selectPT(N,x,c) *selects the maximum delay time* t *for a delay deadline* x *and a given minimum probability* c*, such that* $\forall(\mathrm{c}_i,\mathrm{t}_i) \in \mathrm{N.CDTH}_x$*, where* $\mathrm{c}_i \leq \mathrm{c}$*:* $\mathrm{t}_i \geq \mathrm{t}$.

Thus *selectPT(prepare,jun1,0.9) = 6*: "The maximum delay will be 6 days (with a 90% certainty)!". As the workflow participant should not be confused with probabilities, the minimum certainty should be configured as a fixed work-list parameter. The selectPT-operation can now be used to generate a presorted work list, which proposes an execution order of work items where the overall delay to expect is significantly reduced, compared to for instance a FIFO-strategy. The examination of these algorithms is subject of ongoing work. Nevertheless a necessary prerequisite is, that the workflow system features a time management component, which performs the above explained calculation algorithms based on empirical data from the workflow log. Additionally the work-list must be enabled to invoke this algorithms on demand and to represent the results accordingly.

7 Conclusions and Future Work

We proposed a method which aims at assisting workflow-participants in their decisions to select work-items to be executed next. This method is based on delay tables, containing probabilistic information about the delay to expect when the execution of a task is postponed to a certain date. They are calculated by utilizing the structure of the workflow, augmented with empirical information about expected execution-durations and branching-probabilities. A suggested execution sequence will decrease turnaround times, avoid time-related escalations and subsequently save costs.

Currently, we investigate how to include duration-distributions for activities, as scalar duration values are to imprecise in a realistic administrative workflow scenario. Another research objective is to cope with the complexity explosion due to unfold-operation, by adapting the algorithm in order to work with still folded (and non-full-blocked) graphs. Furthermore we proceed with our investigations on optimization algorithms for presorted work lists. The integration of probabilistic time management into workflow environments is subject of ongoing research.

References

1. W. M. P. van der Aalst and H. A. Reijers. Analysis of discrete-time stochastic petrinets. In *Statistica Neerlandica, Journal of the Netherlands Society for Statistics and Operations Research*, Volume 58 Issue 2, 2003.
2. C. Bussler. Workflow Instance Scheduling with Project Management Tools. In *9th Workshop DEXA'98*. IEEE Computer Society Press, 1998.
3. G. Baggio and J. Wainer and C. A. Ellis. Applying Scheduling Techniques to Minimize the Number of Late Jobs in Workflow Systems. In *Proc. of the 2004 ACM Symposium on Applied Computing (SAC)*. ACM Press, 2004.
4. C. Bettini, X. X. Wang, and S. Jajodia. Temporal reasoning in workflow systems. *Distributed and Parallel Databases*, 11(3), May 2002.
5. C. Combi and G. Pozzi. Temporal conceptual modelling of workflows. In *Proc. of the Int. Conf. on Conceptual Modeling (ER 2003)*. LNCS 2813. Springer, 2003.
6. J. Eder and H. Pichler. Duration Histograms for Workflow Systems In Proc. of the Conf. on Engineering Information Systems in the Internet Context 2002, Kluwer Academic Publishers, 2002.
7. J. Eder and E. Panagos. Managing Time in Workflow Systems. In *Workflow Handbook 2001*. Future Strategies INC. in association with Workflow Management Coalition, 2000.
8. J. Eder, W. Gruber, M. Ninaus, and H. Pichler Personal Scheduling for Workflow Systems. LNCS 2678, Springer Verlag, 2003.
9. W. Gruber Modeling and Transformation of Workflows with Temporal Constraints. Akademische Verlagsgesellschaft, Berlin, 2004.
10. M. Gillmann, G. Weikum, and W. Wonner. Workflow management with service quality guarantees. In *Proc. of the 2002 ACM SIGMOD Int. Conf. on Management of Data*. ACM Press, 2002.
11. O. Marjanovic, M. Orlowska. On modeling and verification of temporal constraints in production workflows. *Knowledge and Information Syst.*, 1(2), 1999.
12. E. Panagos and M. Rabinovich. Predictive workflow management. In *Proc. of the 3rd Int. Workshop on Next Generation Information Technologies and Systems*, Neve Ilan, Israel, June 1997.
13. S.Sadiq, O. Marjanovic, and M. E. Orlowska. Managing Change and Time in Dynamic Workflow Processes. In *International Journal of Cooperative Information Systems*. Vol. 9, Nos. 1 & 2. March - June 2000.
14. Interface 1: Process Definition Interchange - Process Model. *Workflow Management Coalition Specification*. Document number TC00-1016-P.