Time Constraints in Workflow Systems

Johann Eder*, Euthimios Panagos, and Michael Rabinovich

AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
eder@acm.org, {thimios, misha}@research.att.com

Abstract. Time management is a critical component of workflow-based process management. Important aspects of time management include planning of workflow process execution in time, estimating workflow execution duration, avoiding deadline violations, and satisfying all external time constraints such as fixed-date constraints and upper and lower bounds for time intervals between activities. In this paper, we present a framework for computing activity deadlines so that the overall process deadline is met and all external time constraints are satisfied.

1 Introduction

Dealing with time and time constraints is crucial in designing and managing business processes. Consequently, time management should be part of the core management functionality provided by workflow systems to control the lifecycle of processes. At build-time, when workflow schemas are developed and defined, workflow modelers need means to represent time-related aspects of business processes (activity durations, time constraints between activities, *etc.*) and check their feasibility (i.e., timing constraints do not contradict each other). At run-time, when workflow instances are instantiated and executed, process managers need pro-active mechanisms for receiving notifications of possible time constraint violations. Workflow participants need information about urgencies of the tasks assigned to them to manage their personal work lists. If a time constraint is violated, the workflow system should be able to trigger exception handling to regain a consistent state of the workflow instance. Business process re-engineers need information about the actual time consumption of workflow executions to improve business processes. Controllers and quality managers need information about activity start times and execution durations.

At present, support for time management in workflow systems is limited to process simulations (to identify process bottlenecks, analyze activity execution durations, *etc.*), assignment of activity deadlines, and triggering of process-specific exception-handling activities (called *escalations*) when deadlines are missed at run time [10, 8, 7, 18, 2, 3, 17]. Furthermore, few research activities about workflow and time management exist in the literature. A comparison with these efforts is presented in Section 7.

Our contributions in this paper include the formulation of richer modeling primitives for expressing time constraints, and the development of techniques for checking

* On leave from the University of Klagenfurt, Austria

satisfiability of time constraints at process build and instantiation time and enforcing these constraints at run time. The proposed primitives include upper and lower bounds for time intervals between workflow activities, and binding activity execution to certain fixed dates (e.g., first day of the month). Our technique for processing time constraints computes internal activity deadlines in a way that externally given deadlines are met and no time constraints are violated.

In particular, at build time, we check whether for a given workflow schema there exists an execution schedule that does not violate any time constraints. The result is a *timed activity graph* that includes deadline ranges for each activity. At process instantiation time, we modify the the timed activity graph to include the deadlines and date characteristics given when the workflow is started. At run time, we dynamically recompute the timed graph for the remaining activities to monitor satisfiability of the remaining time constraints, given the activity completion times and execution paths taken in the already-executed portion of a workflow instance.

The remainder of the paper is organized as follows. Section 2 describes our workflow model and discusses time constraints. Section 3 presents the workflow representation we assume in this paper. Section 4 presents the calculations that take place during build time. Section 5 shows how these calculations are adjusted at process instantiation to take into account actual date constraints. Section 6 covers run time issues. Section 7 offers a comparison with related work and, finally, Section 8 concludes our presentation.

2 Workflow Model and Time Constraints

A workflow is a collection of *activities, agents,* and *dependencies* between activities. Activities correspond to individual steps in a business process. Agents are responsible for the enactment of activities, and they may be software systems (e.g., database application programs) or humans (e.g., customer representatives). Dependencies determine the execution sequence of activities and the data flow between these activities. Consequently, a workflow can be represented by a workflow graph, where nodes correspond to activities and edges correspond to dependencies between activities.

Here, we assume that execution dependencies between activities form an acyclic directed graph. We should note that we do not propose a new workflow model. Rather, we describe a generic workflow representation for presenting our work. In particular, we assume that workflows are *well structured*. A well-structured workflow consists of m sequential activities, $T_1 \dots T_m$. Each activity T_i is either a primitive activity, which is not decomposed any further, or a composite activity, which consists of n_i parallel conditional or unconditional sub-activities ${T_i}^1, \dots, {T_i}^{n_i}$. Each sub-activity may be, again, primitive or composite. Typically, well structured workflows are generated by workflow languages that provide the usual control structures and adhere to a structured programming style of workflow definitions (e.g., Panta Rhei [4]).

In addition, we assume that each activity has a duration assigned to it. For simplicity, we assume that activity durations are deterministic. Time is expressed in some basic time units, at build-time relative to the start of the workflow, at run-time in some calendar-time. Some time constraints follow implicitly from control dependencies and activity durations of a workflow schema. They arise from the fact that an activity can only start when its predecessor activities have finished. We call such constraints the *structural time constraints* since they reflect the control structure of the workflow.

In addition, *explicit time constraints* can be specified by workflow designers. These constraints are derived from organizational rules, laws, commitments, and so on. Such explicit constraints are either temporal relations between events or bindings of events to certain sets of calendar dates. In workflow systems, events correspond to start and end of activities. For temporal relationships between events, the following constraints exist:

Lower Bound Constraint: The duration between events A and B must be greater than or equal to δ. We write $lbc(A, B, \delta)$ to express that δ is a lower bound for the time-interval between source event A and destination event B.

Upper Bound Constraint: The distance between events A and B must be smaller than or equal to δ. We write $ubc(A, B, \delta)$ to express that δ is an upper bound for the time-interval between source event A and destination event B.

An example of lower-bound constraint includes a legal workflow with activities of serving a warning and closing a business, with the requirement that a certain time period passes between serving the warning and closing the business. Another example is that the invitation for a meeting has to be mailed to the participants at least one week before the meeting. Upper-bound constraints are even more common. The requirement that a final patent filing is done within a certain time period after the preliminary filing, or time limits for responses to business letters, or guaranteed reaction times after the report of a hardware malfunction provide typical examples of upper-bound constraints.

To express constraints that bind events to sets of particular calendar dates, we first need to provide an abstraction that generalizes a, typically infinite, set of dates such as "every other Monday" or "every fifth workday of a month". Examples of such constraints include: vacant positions are announced at the first Wednesday of each month; loans above USD 1M are approved during scheduled meetings of the board of directors; inventory checks have to be finished on December 31st.

Fixed-Date Type: A fixed-date (type) is a data type F with the following methods: $F.valid(D)$ returns true if the arbitrary date D is valid for F; $F.next(D)$ and $F.prev(D)$ return, respectively, the next and previous valid dates after D; $F.period$ returns the maximum distance between valid dates; and $F.dist(F')$ returns the maximum distance between valid dates of F and F', (with $F.period$ as default value).

Fixed-Date Constraint: Event B can only occur on certain (fixed) dates. We write $fdc(B, T)$, where T is a *fixed-date*, to express the fact that B can only occur on dates which are valid for T.

In the remainder of the paper, we assume that at most one fixed-date constraint can be associated with an activity.

3 Workflow Representation

Our techniques for time constraint management are based on the notion of the *timed activity graph*. This graph is essentially the same as the workflow graph where each

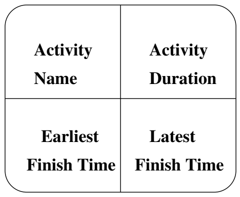

Fig. 1. Activity node of a timed workflow graph

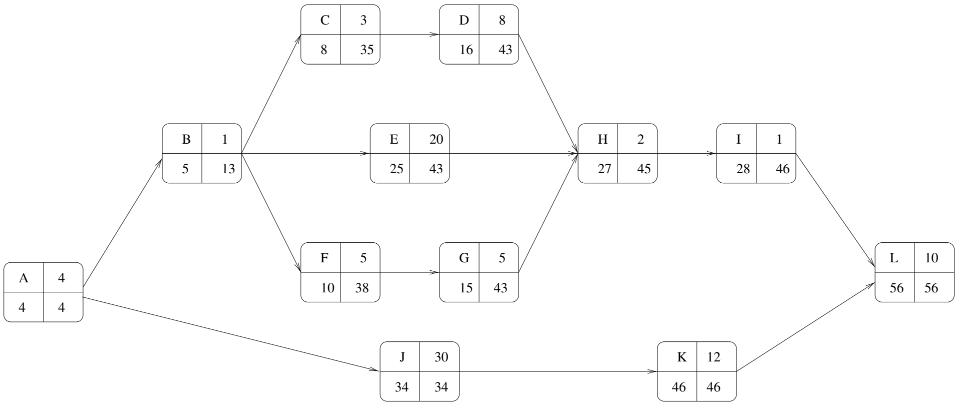

Fig. 2. Example timed workflow graph

activity node n is augmented with two values that represent termination time points for activity executions[1].

- $n.E$: the earliest point in time n can finish execution.
- $n.L$: the latest point in time n has to finish in order to meet the deadline of the entire workflow.

Figure 1 shows the representation of an activity node in the timed workflow graph. Without explicit time constraints, E and L values can be computed using the Critical Path Method (CPM) [14], a well known project planning method that is frequently used in project management software. CPM assumes that activity durations are deterministic. We are aware that this assumption does not hold for many workflows, and that for these workflows a technique dealing with a probability distribution of activity durations like the Project Evaluation and Review Technique (PERT) [14] would be more appropriate. However, we chose CPM because it allows us to present the concept more clearly without the math involved with probability distributions.

Figure 2 shows the timed workflow graph we use in the rest of the paper. The interpretation of E- and L-values is as follows. The earliest point in time for activity F

[1] Since activity durations are assumed to be deterministic, we do not need to represent activity start points. These time points can be computed by subtracting activity durations from activity termination times.

to terminate is 10 time units after the start of the workflow. If F is finished 38 time units after the start of the workflow, the duration of the entire workflow is not extended. Activity L is the last activity of the workflow, and the earliest and latest completion times are the same, 56. This also means that the entire workflow has a duration of 56 time units. The distance between the E-value and the L-value of an activity is called its buffer time. In our example, activity F has a buffer of 28 time units. This buffer, however, is not exclusively available to one activity, but it might be shared with other activities. In our example, the buffer of F is shared with B, G, H, and I. If B uses some buffer-time, then the buffer of F is reduced.

Computing the timed workflow graph delivers the duration of the entire workflow, and deadlines for all activities such that the termination of the entire workflow is not delayed. Incorporating explicit time constraints into the timed activity graph is explained in detail later. For simplicity, we only consider constraints for end events of activities. Therefore, we will use a shortcut and say that an activity (meaning "the end event of the activity") participates in a constraint. The following additional properties are used for representing workflow activities: $n.d$ represents the activity duration; $n.pos$ represents whether the activity n is a start, end, or internal node of the workflow; $n.pred$ represents the predecessors and $n.succ$ the successor activities of n; $n.deadline$ holds the externally assigned deadline of n; $n.tt$ the actual termination time of an activity instance.

For an upper- or lowerbound-constraint c we represent the source activity with $c.s$, the destination activity with $c.d$ and the bound with $c.\delta$. For a fixed-date constraint f, we write $f.a$ for the activity on which f is posed and $f.T$ for the fixed-date.

Since we assume well structured workflows, in the remainder of the paper we assume that for all upper and lower bound constraints the source node is before the destination node according to the ordering implied by the workflow graph.

4 Build-Time Calculations

At build time, our goal is to check if the set of time constraints is satisfiable, i.e., that it is possible to find a workflow execution that satisfies all timing constraints. We start from the original workflow graph and construct a timed workflow graph such that an execution exists that satisfies all constraints. Initially, all fixed-date constraints are transformed into lower-bound constraints. Then, the E- and L-values of all activity nodes in the timed graph are computed from activity durations and lower-bound constraints, using a straightforward modification of the CPM method. Finally, upper-bound constraints are incorporated into the timed graph. The resulting timed graph has at least two (possibly not distinct) valid executions. These executions are obtained if all activities complete at their E-values or all activities complete at their L-values. There may be other valid combinations of activity completion times within (E, L) ranges. We say that a timed graph *satisfies* a constraint if the executions in which all activities complete at their E- or L-values are valid with respect to this constraint.

4.1 Fixed-Date Constraints

The conversion of fixed-date constraints into lower-bound constraints is done using worst-case estimates. This is because at build time we do not have calendar value(s) for the start of the workflow and, thus, we can only use information about the duration between two valid time points for a fixed-date object. At process-instantiation time we will have more information concerning the actual delays due to fixed-date constraints.

Consider a fixed-date constraint $fdc(a,T)$. Assume that activities start instantaneously after all their predecessors finish. In the worst case, activity a may finish at $T.period + a.d$ after its last predecessor activity finishes. Indeed, let t_1 and t_2 be valid dates in T with the maximum time-interval between them. i.e., $t_2 - t_1 = T.period$, and let b be the last predecessor activity to finish. The time-interval between end-events of b and a is the longest if b finishes just after time $t_1 - a.d$, because then a cannot start immediately (it would then not finish at valid date t_1), and would have to wait until time $t_2 - a.d$ before starting. In this case, the distance between b and a is $\delta = (t_2 - t_1) + a.d$ = $T.period + a.d$, assuming b itself does not have a fixed-date constraint associated with it. If b has a fixed-date constraint $fdc(b, T')$, one can use similar reasoning to obtain $\delta = T.dist(T')$ if $a.d \leq T.dist(T')$ and $\delta = a.d + T.period$ otherwise.

To guarantee the satisfiability of all time constraints at build-time, without knowing the start date of the process, the timed graph must allow the distance of at least δ between a and all it's predecessors, where δ is computed for each predecessor as shown above. Consequently, the fixed-date constraint $fdc(a,T)$ is replaced by a lower-bound constraint $lbc(b, a, \delta)$ for every predecessor b of activity a.

4.2 Lower-Bound Constraints

The construction of the timed workflow graph that includes structural and lower-bound constraints is presented below. We should note that due to the way we carry out the computations, the activities in the resulting graph satisfy all lower-bound constraints.

Forward Calculations

```
  for all activities a with a.pos = start
      a.E := a.d
  endfor
  for all activities a with a.pos ≠ start
      in a topological order
      a.E := max({b.E + b.d | b ∈ a.pred},
                 {m.s.E + m.δ | m = lbc(s,a,δ)})
  endfor
```

Backward Calculations

```
  for all activities a with a.pos = end
      a.L := a.E
  endfor
  for all activities a with a.pos ≠ end
      in a reverse topological order
      a.L := min({s.L - s.d| s ∈ a.succ},
                 {m.d.L - m.δ | m = lbc(a,d,δ)})
  endfor
```

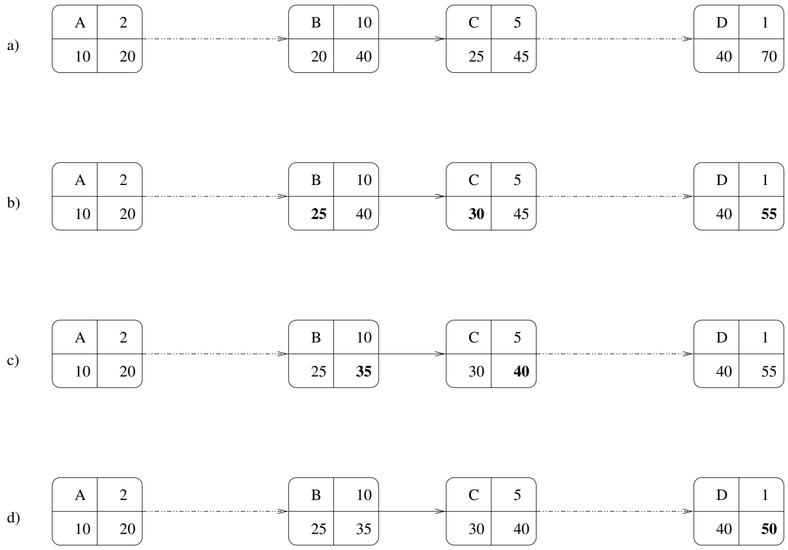

Fig. 3. Incorporating upper-bound constraints

4.3 Upper-Bound Constraints

In the timed workflow graph constructed during the previous step, an upper-bound constraint $ubc(s, d, \delta)$ is violated when either $s.E + \delta < d.E$ or $s.L + \delta < d.L$. In this case, we can use the buffer times of s and d to increase the E-value of s and decrease the L-value of the d in an attempt to satisfy the constraint. However, buffer time availability is a necessary but not sufficient condition for satisfying an upper-bound constraint. For example, in the workflow graph shown in Figure 2, the upper-bound constraint $ubc(B, C, 8)$ cannot be satisfied because the value of $C.E$ is always increased by the same amount as $B.E$.

A necessary condition for constraint $ubc(s, d, \delta)$ to be satisfiable is that the *distance* between s and d is less than δ. The distance is the sum of the durations of the activities on the longest path between s and d, and it can be computed by using the forward or backward calculations presented in the previous section, with s the starting node and d the ending node. We should note that for well structured workflows this distance does not change when the deadline of the entire workflow is relaxed and, therefore, more buffer becomes available for each activity. Consequently, extending the deadline of the whole workflow does not help us in satisfying violated upper-bound constraints.

One can show that if there is a node n between s and d with less buffer than both of them, then the buffer of s and the buffer of d can be reduced without influencing the E-values or L-values of the other node by: $min(buffer(s) - buffer(n), buffer(d) - buffer(n))$. Instead of finding the "safe" value by which the buffer of one end-point of the constraint can be reduced without affecting E and L values of the other end-point, we follow a more constructive approach.

If an upper-bound constraint is violated, we set the E-value of the source node to the value of $d.E - \delta$. If this value is greater than $s.L$, the upper-bound constraint is violated. Otherwise, we recompute the E-values of the timed graph starting at s and if the E-value of d does not change, the constraint is satisfied. In a similar way, we decrease the L-value of the destination node by δ and if this value is not less than $d.E$, we recompute the L-values of all predecessors of d. If the L-value of s does not change, the constraint is satisfied.

While the above can be used for individual upper-bound constraints, it is not enough for handling multiple upper-bound constraints. Figure 3 shows an example that demonstrates this problem. Figure 3a) shows the starting timed graph. Figure 3b) shows the timed graph after the integration of $ubc(B, D, 15)$. When $ubc(A, C, 20)$ is integrated, $ubc(B, D, 15)$ is violated at the L-values, as shown in Figure 3c). Finally, Figure 3d) shows the successful integration of both constraints.

We address this problem by checking whether an already incorporated upper-bound constraint is violated when new upper-bound constraints are incorporated into the timed graph. The following unoptimized algorithm summarizes this procedure. In this algorithm, the re-computation of the timed graph involves the forward and backward computations presented in the previous section.

```
repeat
    error := false
    for each m = ubc(s,d,δ)
          if m.s.E + m.δ < m.d.E          (* violation at E *)
                if m.s.L > m.d.E - m.δ    (* slack  at m.s  *)
                      m.s.E := m.d.E - m.δ
                      recompute timed graph
                      if m.d.E changes
                            error := true
                      endif
                else
                      error := true
                endif
          endif
          if m.s.L + m.δ < m.d.L          (* violation at L *)
                if m.d.E < m.s.L + m.δ    (* slack  at m.d  *)
                      m.d.L := m.s.L + m.δ
                      recompute timed graph
                      if m.s.L changes
                            error := true
                      endif
                else
                      error := true
                endif
          endif
    endfor
until error = true or nothing changed
```

This algorithm for the incorporation of upper-bound constraints has the following properties:

1. *Termination:* The algorithm terminates.
   At each loop there is at least one node `x` for which `x.E` is increased or `x.L` is decreased by at least one unit. Since there is a finite number of nodes, and the E- and L-values are bound, the algorithm must terminate.

2. *Admissibility:* A solution is found if there exists a timed graph satisfying all constraints.
   For an upper-bound constraint $m(s, d, \delta)$, $m.d.E - m.\delta$ is less than or equal to $m.s.E$ and $m.s.L + m.\delta$ is greater than or equal to $m.d.L$ for any timed graph satisfying the constraints. Since we set $m.s.E$ and $m.d.L$ to these values and, moreover, we know that the algorithm terminates, we can conclude that the algorithm will compute a solution, if one exists.
3. *Generality:* The algorithm finds the most general solution, if one exists.
   Let G and G' be timed graphs which differ only in the E- and L-values. We call G more general than G', if for every activity a the following condition holds: $a_G.E \leq a_{G'}.E$ and $a_G.L \geq a_{G'}.L$. Following the discussion of admissibility, it is easy to see that the timed graph generated by the algorithm above is more general than any other timed graph satisfying the constraints.
4. *Complexity:* The worst-case complexity of the algorithm is $O(m * d * n)$, where m is the number of upper-bound constraints, d is the largest buffer, and n is the number of activities.
   We can give an upper bound for number of iterations of this algorithm as follows: in each iteration there is at least one E-value increased or one L-value decreased at least by one unit. If there are m upper-bound constraints, and d is the largest buffer, then the number of iterations is $m * d * 2$ in the worst case. The recalculation is linear with the number of nodes.

5 Calculations at Process Instantiation Time

At process instantiation time, an actual calendar is used in order to transform all time information which was computed relative to the start of the workflow to absolute time points. It is also possible at this procedure to set the $a.deadline$ value for an activity a, and increase or decrease the buffers computed at build time. Based on the calculations performed at build time, a deadline for an activity a is valid if it is greater than or equal to $a.E$. Fixed-date constraints are also resolved at process instantiation time, since they rely on absolute time points. (We used worst case estimates for these constraints during build time).

The computations that take place at process instantiation time are presented below, assuming that the variable *start* corresponds to the start-time of the workflow instance.

Forward Calculations

```
for all activities a with a.pos = start
    a.E := start + a.d
endfor
for all activities a with a.pos ≠ start
    in a topological order
    a.E := max({b.E + b.d | b ∈ a.pred},
               {m.s + m.δ | m = lbc(s,a,δ)})
    if there exists dc = fdc(a,T)
         a.E := dc.T.next(a.E)
    endif
endfor
```

Backward Calculations

```
for all activities a with a.pos = end
    if a.deadline < a.E
          raise exception
    else
          a.L := a.deadline
endfor
for all activities a with a.pos ≠ end
    in a reverse topological order
    a.L := min({s.L - s.d| s ∈ a.succ},
               {m.d - m.δ | m = lbc(a,d,δ)})
    if exists a.deadline and a.deadline < a.L
          a.L := a.deadline
    endif
    if there exists dc = fdc(a,T)
          a.L := dc.T.prev(a.L)
    endif
    if a.L < a.E
          raise exception
    endif
endfor
```

Incorporation of Upper-Bound Constraints: **incorporate()**

```
repeat
    error := false
    ok    := true
    for each m = ubc(s,d,δ)
          if m.s.E + m.δ < m.d.E          (* violation at E *)
                m.s.E := m.d.E - m.δ
                ok    := false
                if there exists dc = fdc(m.s,T)
                      m.s.E := dc.T.next(m.s.E)
                endif
                if m.s.E > m.s.L
                      error := true
                endif
          endif
          if m.s.L + m.δ < m.d.L          (* violation at L *)
                 m.d.L := m.s.L + m.δ
                 ok    := false
                 if there exists dc = fdc(m.d,T)
                       m.s.L := dc.T.prev(m.d.L)
                 endif
                 if m.d.E > m.d.L
                       error := true
                 endif
          endif
    endfor
    if ok = false and error = false
          error := recompute();
    endif
until error = true or ok = true
```

Timed Graph Re-computation: **recompute()**

```
for all activities a in topological order
    a.E := max({b.E + b.d | b ∈ a.pred},
               {m.s + m.δ | m = lbc(s,a,δ)}, a.E)
    if there exists dc = fdc(a,T)
```

```
        a.E := dc.T.next(a.E)
    endif
    if a.L < a.E
        return false
    endif
endfor
for all activities a in reverse topological order
    a.L := min({s.L - s.d | s ∈ a.succ},
               {m.d - m.δ | m = lbc(a,d,δ)}, a.L)
    if there exists dc = fdc(a,T)
        a.L := dc.T.prev(a.L)
    endif
    if a.L < a.E
        return false
    endif
endfor
return true
```

There is a possibility of optimizing the re-computation procedure by starting at the first node where an E-value was changed. However, there is additional overhead associated with this. Finally, the algorithm for incorporating upper-bound constraints into the timed graph has the same properties as the corresponding one presented in Section 4.

6 Time Management at Run-Time

6.1 General computations

During the execution of a given workflow instance, we have to ensure that deadlines are not missed and any time constraints attached to activities are not violated. In order to achieve this, we may have to delay the execution of some of the activities that are either sources of upper-bound constraints or destinations of lower-bound constraints. Figure 4 shows a workflow segment having the upper-bound constraints $ubc(C, I, 18)$ and $ubc(G, H, 7)$. In this example, if F ends at 10 and C ends at 25 and, thus, D will end at 33 and H at 35, G must not start before 28 because the upper-bound constraint will be violated.

Even when we can immediately start the execution of an activity that is the source of some upper-bound constraint, it can be advantageous to delay its enactment so that the remaining activities have more buffer. In the example of Figure 4, if C starts at 7, it will finish at 10 and the buffer for all other activities is reduced. In particular, E, H, and I will have no buffer available since they have to finish at their E-values to satisfy the upper-bound constraints.

Selecting an optimal delay value for an activity is part of on-going work. Furthermore, existing work [11–13] can be used for distributing available buffer and slack times to activities and avoid time exceptions – `assign-deadline()` corresponds to this in the algorithm presented below. Buffer distribution addresses the distribution of extra buffer time that results from the assignment of an overall workflow deadline that is greater than the L-values of all activities with no successors. Slack distribution addresses the distribution of slack time that becomes available when activities finish before their L-values.

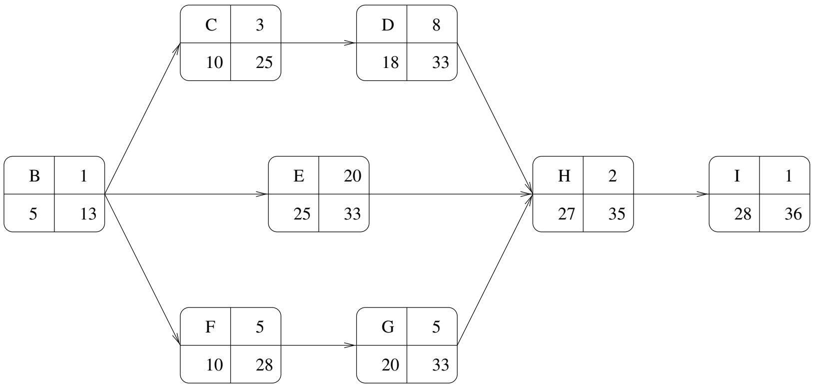

Fig. 4. Workflow segment with $ubc(C, I, 18)$ and $ubc(G, H, 7)$

The algorithm presented below assumes that activities finish within the interval defined by their E- and L-values (this implies that the termination of an activity should be delayed until, at least, its E-value. Allowing activities to finish before their E-values is subject of on-going work). When activity a finishes in the interval $(a.E, a.L)$, we may have to recompute the timed graph and re-incorporate upper-bound constraints before modifying the L-values of the ready activities according to the buffer and slack distribution algorithm. In addition, the re-computation of the timed graph should use the L-values of any active activities for computing E-values in order to avoid upper-bound constraint violations.

```
if a.tt ≤ a.L        (* a.tt = actual termination time *)
    a.L  := a.tt
    a.E  := a.tt
    done := false
    if a is the source of a lower-bound constraint
          recompute()
          incorporate()
          done := true
    endif
    for each b in a.succ that is ready for execution
          if b is source of an upper-bound constraint
                if done = false
                      recompute()
                      incorporate()
                endif
                b.L := assign-deadline(b)
                recompute()
                incorporate()
                done := true
          else
                b.L  := assign-deadline(b)
                done := false
          endif
          launch b for execution
    endfor
else
    invoke escalation process, deadline was missed
endif
```

6.2 Schedules

The execution of a workflow instance according to the procedures above requires re-computation of the timed graph after the completion of an activity that is the source of a lower-bound constraint or has a successor that is the source of an upper-bound constraint. We can avoid these re-computations by sacrificing some flexibility in the timed graph. Recall that the timed graph specifies ranges for activity completion times such that there *exists* a combination of activity completion times that satisfies all timing constraints and in which each completion time is within the range of its activity. Run-time re-computation was required because, once completion time for finished activities has been observed, not all completion times within the ranges of the remaining activities continue to be valid.

We define a *schedule* to be a (more restrictive) timed graph in which *any* combination of activity completion times within $[E, L]$ ranges satisfies all timing constraints. In other words, given a schedule, no violations of time constraints occur as long as each activity a finishes at time within the interval $[a.E, a.L]$. Consequently, as long as activities finish within their ranges, no timed graph re-computation is needed. Only when an activity finishes outside its range must the schedule for the remaining activities be recomputed.

It follows directly from the schedule definition that, for every upper-bound constraint $ubc(s, d, \delta)$, $s.E + \delta \geq d.L$ and for every lower-bound constraint $lbc(s, d, \delta)$, $s.L + \delta \leq d.E$; the reverse is also true, i.e., the timed graph that satisfies these properties is a schedule (compare this property with the two inequalities that are incorporated into the timed graph in Section 4). From the way we compute E- and L-values for the activities in a timed workflow graph, the E- and L-values already qualify as schedules. Consequently, when every workflow activity finishes execution at its E-value, there is no need to check for time constraint violations. The same is true when activities finish execution at L-values.

The development of algorithms for computing schedules with various characteristics is subject of ongoing research.

7 Related Work

The area of handling time-related issues and detecting potential problems at build, instantiation, and run time has not received adequate attention in the workflow literature. Existing workflow systems offer some limited abilities to handle time. For example, they support the assignment of execution durations, and deadlines to business processes and their activities, and they monitor whether deadlines are met.

In [9], an ontology of time is developed for identifying time structures in workflow management systems. They propose the usage of an Event Condition Action (ECA) model of an active database management system (DBMS) to represent time aspects within a workflow environment. They also discuss special scheduling aspects and basic time-failures. We used parts of their definitions as basis of our concept.

In [11–13], the authors propose to use static data (e.g., escalation costs), statistical data (e.g., average activity execution time and probability of executing a conditional activity), and run-time information (e.g., agent work-list length) to adjust activity deadlines and estimate the remaining execution time for workflow instances. However, this work can be used only at run-time, and it does not address explicit time constraints.

[1] proposes the integration of workflow systems with project management tools to provide the functionality necessary for time management. However, these project management tools do not allow the modeling of explicit time constraints and, therefore, have no means for their resolution.

In [15], the authors present an extension to the net-diagram technique PERT to compute internal activity deadlines in the presence of sequential, alternative, and concurrent executions of activities. Under this technique, business analysts provide estimates of the best, worst, and median execution times for activities, and the β-distribution is used to compute activity execution times as well as shortest and longest process execution times. Having done that, time constraints are checked at build time and escalations are monitored at run-time. Our work extends this work by providing a technique for handling both structural and explicit time constraints at process build and instantiation times, and enforcing these constraints at run-time.

In [6, 16], the notion of explicit time constraints is introduced. Nevertheless, this work focused more on the formulation of time constraints in workflow definitions, the enforcement of time constraints through monitoring of time constraints at run-time and the escalation of time failures within workflow transactions [5]. Our work follows the work described in [6, 16] and extends it with the incorporation of explicit time constraints into workflow schedules.

8 Conclusions

Dealing with time and time constraints is crucial in designing and managing business processes.

In this paper, we proposed modeling primitives for expressing time constraints between activities and binding activity executions to certain fixed dates (e.g., first day of the month). Time constraints between activities include lower- and upper-bound constraints. In addition, we presented techniques for checking satisfiability of time constraints at process build and process instantiation time, and enforcing these constraints at run-time. These techniques compute internal activity deadlines in a way that externally assigned deadlines are met and all time constraints are satisfied. Thus the risk of missing an external deadline is recognized early and steps to avoid a time failure can be taken, or escalations are triggered earlier, when their costs are lower.

Our immediate work focuses on: (1) using the PERT-net technique for computing internal deadlines to express deviations from the average execution duration of activities; (2) addressing conditionally and repetitive executed activities by providing execution probabilities to estimate average duration and variance for workflow executions; (3) considering optional activities and pruning the workflow graph when such activities should be eliminated to avoid time exceptions; and (4) addressing the different duration values that could be used at build time: best, average, and worst case execution times and turn-around times, which include the time an activity spends in the work-list and the time between start and end of an activity.

References

1. C. Bussler. Workflow Instance Scheduling with Project Management Tools. In *9th Workshop on Database and Expert Systems Applications DEXA'98*, Vienna, Austria, 1998. IEEE Computer Society Press.
2. CSESystems. *Benutzerhandbuch V 4.1 Workflow*. CSE Systems, Computer & Software Engineering GmbH, Klagenfurt, Austria, 1996.
3. CSE Systems Homepage. http://www.csesys.co.at/, February 1998.
4. J. Eder, H. Groiss, and W. Liebhart. The Workflow Management System Panta Rhei. In A. Dogac et al., editor, *Advances in Workflow Management Systems and Interoperability*. Springer, Istanbul, Turkey, August 1997.
5. J. Eder and W. Liebhart. Workflow Transactions. In P. Lawrence, editor, *Workflow Handbook 1997*. John Wiley, 1997.
6. Johann Eder, Heinz Pozewaunig, and Walter Liebhart. Timing issues in workflow management systems. Technical report, Institut f"ur Informatik-Systeme, Universit"at Klagenfurt, 1997.
7. TeamWare Flow. Collaborative workflow system for the way people work. P.O. Box 780, FIN-00101, Helsinki, Finland.
8. InConcert. Technical product overview. XSoft, a division of xerox. 3400 Hillview Avenue, Palo Alto, CA 94304. http://www.xsoft.com.
9. Heinrich Jasper and Olaf Zukunft. Zeitaspekte bei der Modellierung und Ausführung von Workflows. In S. Jablonski, H. Groiss, R. Kaschek, and W. Liebhart, editors, *Geschäftsprozeßmodellierung und Workflowsysteme*, volume 2 of *Proceedings Reihe der Informatik '96*, pages 109 – 119, Escherweg 2, 26121 Oldenburg, 1996.
10. F. Leymann and D. Roller. Business process management with flowmark. In *Proceedings of the 39th IEEE Computer Society International Conference*, pages 230–233, San Francisco, California, February 1994. http://www.software.ibm.com/workgroup.
11. E. Panagos and M. Rabinovich. Escalations in workflow management systems. In *DART Workshop*, Rockville, Maryland, November 1996.
12. E. Panagos and M. Rabinovich. Predictive workflow management. In *Proceedings of the 3rd International Workshop on Next Generation Information Technologies and Systems*, Neve Ilan, ISRAEL, June 1997.
13. E. Panagos and M. Rabinovich. Reducing escalation-related costs in WFMSs. In A. Dogac et al., editor, *NATO Advanced Study Institue on Workflow Management Systems and Interoperability*. Springer, Istanbul, Turkey, August 1997.
14. Susy Philipose. *Operations Research - A Practical Approach*. Tata McGraw-Hill, New Delhi, New York, 1986.
15. H. Pozewaunig, J. Eder, and W. Liebhart. ePERT: Extending PERT for Workflow Management Systems. In *First EastEuropean Symposium on Advances in Database and Information Systems ADBIS 9́7*, St. Petersburg, Russia, Sept. 1997.
16. Heinz Pozewaunig. Behandlung von Zeit in Workflow-Managementsystemen - Modellierung und Integration. Master's thesis, University of Klagenfurt, 1996.
17. SAP Walldorf, Germany. *SAP Business Workflow©Online-Help*, 1997. Part of the SAP System.
18. Ultimus. Workflow suite. Business workflow automation. 4915 Waters Edge Dr., Suite 135, Raleigh, NC 27606. http://www.ultimus1.com.