RZ 3896 (#ZUR1603-057) 03/21/2016
Computer Sciences/Mathematics 22 pages

Research Report

The Complexity of Deadline Analysis for Workflow Graphs with Multiple Resources

Mirela Botezatu[1,2], Hagen Völzer[1] and Lothar Thiele[2]

[1]IBM Research – Zurich
8803 Rüschlikon
Switzerland

[2]ETH Zurich
Switzerland

The Complexity of Deadline Analysis for Workflow Graphs with Multiple Resources

Mirela Botezatu[1,2], Hagen Völzer[1], and Lothar Thiele[2]

[1] IBM Research – Zurich, Switzerland
[2] ETH – Zurich, Switzerland

Abstract. We study whether the executions of a time-annotated sound workflow graph (WFG) meet a given deadline when an unbounded number of resources (i.e., executing agents) is available. We present polynomial-time algorithms and NP-hardness results for different cases. In particular, we show that it can be decided in polynomial time whether some executions of a sound workflow graph meet the deadline. For acyclic sound workflow graphs, it can be decided in linear time whether some or all executions meet the deadline. Furthermore, we show that it is NP-hard to compute the expected duration of a sound workflow graph for unbounded resources, which is contrasting the earlier result that the expected duration of a workflow graph executed by a single resource can be computed in cubic time. We also propose an algorithm for computing the maximum concurrency of the workflow graph, which helps to determine the optimal number of resources needed to execute the workflow graph.

1 Introduction

A workflow graph can capture the main control flow of processes modeled in languages such as BPMN, UML-Activity Diagrams, and Event Process Chains, cf. [11]. That is, the core routing constructs of these languages can be mapped to the routing constructs of workflow graphs, which are alternative choice and merge, and concurrent fork and join. Fig. 1 shows an example of a workflow graph modeling a ticket resolution work-

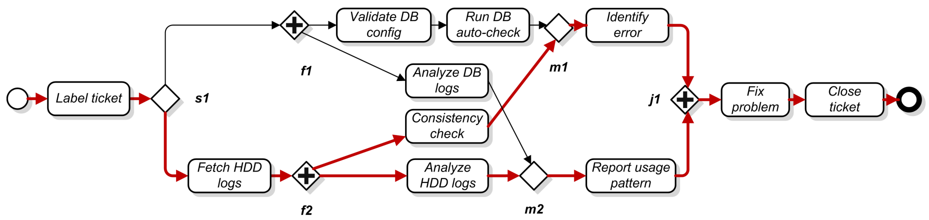

Fig. 1: An example of a workflow graph and one of its executions (red)

flow. After a task to categorize the ticket ("Label ticket"), there is a choice $s1$ whether the ticket documents a database issue (DB) or a disk issue (HDD). Following the case of HDD, there is a preliminary step to fetch the disk logs followed by a fork $f2$ that spawns two concurrent threads. One thread follows "Consistency check", the other thread follows "Analyze HDD logs". Then each thread is merged with the corresponding thread of the case DB through the merge gateways $m1$ and $m2$. After merging, there are some additional tasks “Identify error” and “Report usage pattern”, before the threads are synchronized at the join $j1$. Finally there are some wrap-up tasks, common to both cases.

A workflow graph is equivalent to a two-terminal free-choice Petri net i.e., a connected net with a unique source and sink, which is also called a *free-choice workflow net* [6]. A workflow graph can be seen as a compact representation of the corresponding free-choice net. Therefore, the theory of free-choice Petri nets directly applies to workflow graphs.

A workflow graph may contain a local deadlock or exhibit *lack of synchronization*. The latter corresponds to unsafeness in Petri nets. The absence of local deadlock and lack of synchronization has been termed *soundness*, which can be decided in cubic time by help of the rank theorem for free-choice Petri nets [5], also cf. [1].

In this paper, we analyze whether the executions of a sound workflow graph meet a given deadline, where tasks, or, equivalently, edges are annotated with execution times. We are not aware of any similar work for the model class we investigate. In our previous work [3], we considered the case where the workflow graph is executed by a single resource (i.e., executing agent). In this work, we provide results for the case where the workflow graph is executed by an unbounded number of resources. We also discuss the case of a fixed number $n > 1$ of resources in Section 6.

General workflow graphs can of course be analyzed for timing behavior in terms of their reachability graph, and there are various techniques and tools that support this [9, 10, 18]. This holds also for non-Petri-net like models, e.g., timed automata where the minimum cost reachability problem is addressed through exponential branch-and-bound based algorithms [13]. Since the construction of the reachability graph incurs an exponential blowup, these techniques do not run in polynomial time in the size of the workflow graph. In this paper, we show that some deadline analysis problems for workflow graphs can nevertheless be solved in polynomial time.

	1. All executions	2. Some execution	3. Probability of transgression	4. Expected duration	5. Min. nr. resources
A. Sound WFG	NP-hard	$\boldsymbol{O(\lvert V\rvert\lvert E\rvert)}$	NP-hard	**NP-hard**	**open***
B. Acyclic Sound WFG	$\boldsymbol{O(\lvert V\rvert + \lvert E\rvert)}$	$\boldsymbol{O(\lvert V\rvert + \lvert E\rvert)}$	NP-hard	**NP-hard**	**open***
C. Regular WFG	$O(\lvert V\rvert + \lvert E\rvert)$	$O(\lvert V\rvert + \lvert E\rvert)$	NP-hard	**NP-hard**	$O(\lvert V\rvert + \lvert E\rvert)$

Table 1: Overview of results; new contributions in bold, * we give a heuristic for this in Sect. 5

Table 1 shows the results for deadline analysis of sound workflow graphs with unbounded resources, where our new contributions in this paper are written in bold.

First, we ask whether all executions of a workflow graph finish before a given deadline. This is a question that arises when the choices made in the process at runtime are not under our control. This corresponds to Column 1 in Table 1. For the general case (Cell A.1), loops in the graph are constrained by a termination order. The complexity result for this case follows directly from Theorem 2 in our previous paper [3]. For acyclic workflow graphs, this question can be answered in linear time (Cell B.1) and we provide an algorithm for this in Section 3. For *regular graphs*, which are workflow graphs that can be generated by a regular expression, i.e., every split corresponds to a join of the same logic (see Fig. 3 for an example), the solutions consist of simple recursive algorithms that run in linear time (Cell C.1).

Next, we assume we have control over the choices made in the process at runtime. Therefore, we ask the question whether there exists an instantiation of the process – an execution – that meets a given deadline. This corresponds to Column 2 in Table 1. In particular, as one of our main contributions, we show that for general sound workflow graphs, finding the minimum duration over all executions can be solved in polynomial time (Cell A.2). When restricting to acyclic workflow graphs (Cell B.2, similarly as for Cell B.1), the problem can be solved in linear time. As above, for regular graphs, the minimum duration of an execution can be computed recursively in linear time.

Suppose not all executions meet a given deadline but only some. We can then ask whether the probability of a deadline transgression exceeds a given threshold - Column 3 in Table 1. Results carry over from our previous work [3] where we have proven that computing whether the probability of an execution with a single resource terminating before the deadline exceeds a given threshold is NP-hard (Cells A.3, B.3 and C.3).

Also in the probabilistic framework, another valuable information is the expected duration of an execution of a given workflow graph. The results related to this question map to Column 4 in Table 1. We show that computing the expected duration is NP-hard even for regular graphs. This is in contrast to the execution with a single resource where, the expected duration can be computed in cubic time for general sound workflow graphs [3].

Finally, we ask what is the optimal number of resources for the workflow graph where optimal means the minimum number k of resources such that each execution achieves its minimal execution time under k resources (Column 5 in Table 1). We propose an algorithm for computing the maximum concurrency of a workflow graph in Section 5, which is an upper bound for the optimal number of resources.

2 Preliminaries

In this section, we define the necessary fundamental notions, which include workflow graphs and their semantics.

A *weighted, directed multi-graph* $G = (V, E, c, w)$ consists of a set of nodes V, a set of edges E, a mapping $c : E \rightarrow V \times V$ that maps each edge to an ordered pair of nodes and a mapping $w : E \rightarrow \mathbb{N}$ that maps each edge to a nonnegative integer, called its *weight* or *duration*. For each edge e with $c(e) = (v, z)$, we assume $v \neq z$ for simplicity throughout the paper.

A *workflow graph* $\Gamma = (V, E, c, l, w)$, is a weighted multi-graph $G = (V, E, c, w)$ with distinct and unique source and sink nodes, denoted v_{source} and v_{sink}, respectively, equipped with an additional mapping $l : V \setminus \{v_{source}, v_{sink}\} \rightarrow \{\text{XOR}, \text{AND}\}$ that associates a *branching logic* with every node, except for the source and the sink. Furthermore, we assume that every node is on a path from the source to the sink, that the source has a unique outgoing edge, called the *source edge* (e_{source}), and that the sink has a unique incoming edge, called the *sink edge* (e_{sink}). For each node v, we define the *pre-set* of v, ${}^{\bullet}v = \{e \in E \mid \exists\, z \in V : c(e) = (z, v)\}$ and the post-set of v,

$v^\bullet = \{e \in E \mid \exists\, z \in V : c(e) = (v, z)\}$. A node with a single incoming edge and multiple outgoing edges is called a *split*. A node with multiple incoming edges and a single outgoing edge is called a *join*. We don't allow nodes that have multiple incoming edges as well as multiple outgoing edges. Note that this is not restrictive as such a node can be converted into a join followed by a split without changing the semantics.

Fig. 1 shows a workflow graph in BPMN notation: An XOR gateway is depicted as a diamond, an AND gateway as a diamond decorated with a plus sign. Source and sink are depicted as circles. A node that is neither a join, split, nor source or sink is usually called a *task*. A task is shown as a rounded rectangle in Fig. 1. It is natural to assign durations to tasks. Tasks are executed by *resources*: non-preemptive, identical agents, and we assume an unbounded number of these. We will henceforth omit tasks for simplicity and annotate each edge with a duration $w(e)$ as formalized above.

Let A be a set. A *multi-set* over A is a mapping $m : A \to \mathbb{N}$. For two multi-sets m_1, m_2, and each $x \in A$, we have: $(m_1 + m_2)(x) = m_1(x) + m_2(x)$ and $(m_1 - m_2)(x) = m_1(x) - m_2(x)$.

A *marking* $m : E \to \mathbb{N}$ of a workflow graph is a multi-set over E. If $m(e) = i$, we say that there are *i tokens* on edge e. The marking with exactly one token on the source edge and no token elsewhere is called the *initial marking*, denoted by m_s. The marking with exactly one token on the sink edge and no token elsewhere is called the *final marking* of the workflow graph, denoted by m_f.

The *semantics* of workflow graphs is defined as a token game as it is in Petri nets. A comprehensive analysis of the relationship between workflow graphs and free-choice workflow nets (a subclass of Petri nets) can be found in [6]. The execution of a node with an AND-logic removes one token from each of its incoming edges and adds one token to each of the outgoing edges. The execution of a node with a XOR-logic removes non-deterministically a token from one of its incoming edges that has a token, then non-deterministically adds one token to one of the outgoing edges. Although we omit tasks, we allow nodes with just one incoming and one outgoing edge for technical reasons. For such nodes, XOR- and AND-logic behave the same.

A triple $T = (E_1, v, E_2)$ is called a *transition* of Γ if $v \in V$, $E_1 \subseteq {}^\bullet v$, and $E_2 \subseteq v^\bullet$. A transition (E_1, v, E_2) is *enabled* in a marking m if for each edge $e \in E_1$ we have $m(e) > 0$ and any of the following propositions:

- $l(v) = \text{AND}$, $E_1 = {}^\bullet v$, and $E_2 = v^\bullet$, or
- $l(v) = \text{XOR}$, there exists an edge e such that $E_1 = \{e\}$, and there exists an edge e' such that $E_2 = \{e'\}$.

We will use ${}^\bullet T$ to denote E_1 and $T^\bullet$ to denote E_2.

A transition T can be executed in a marking m if T is enabled in m. When T is executed in m, a marking m' results such that $m' = m - E_1 + E_2$. We write $m \to m'$ if there exists a transition T, enabled in a marking m and its execution results in a marking m'. We write $m \xrightarrow{T} m'$ when the transition T is enabled in a marking m and its execution results in the marking m'. We use $\xrightarrow{*}$ to denote the transitive and reflexive closure of $\to$. We say m' is *reachable from a marking m* if $m \xrightarrow{*} m'$. We say m' is a *reachable marking* of Γ if $m_s \xrightarrow{*} m'$.

An *execution* of Γ is an alternate sequence $\sigma = \langle m_s, T_0, m_1, T_1, \cdots \rangle$ of markings m_i of Γ and transitions T_i such that $m_i \xrightarrow{T_i} m_{i+1}$, for each $i \geq 0$. We will be using also the shorter notation $\sigma = \langle m_s, m_1, \cdots \rangle$ to denote an execution.

An execution σ is *maximal* if either σ is of infinite length or σ ends in a marking from which no other marking can be reached.

We say an edge e is *taken* at i if $\exists\, T_i$ such that $e \in T_i^{\bullet}$.

A maximal execution is *fair* if for each XOR-split v, that is executed infinitely often in σ, each edge $e \in v^{\bullet}$ is taken infinitely often in σ.

If $\sigma = \langle m_0, T_0, m_1, T_1, \cdots, T_n, m_{n+1} \rangle$ is an execution, then $\tau_\sigma = \langle T_0, T_1, \cdots, T_n \rangle$ is a *transition sequence* leading from m_0 to m_{n+1} and we write $m_0 \xrightarrow{\tau_\sigma} m_{n+1}$.

A reachable marking m is a *local deadlock* if m has a token on an incoming edge e of an AND-join such that each marking reachable from m also contains a token on e. A reachable marking m is *unsafe* or exhibits *lack of synchronization* if one edge has more than one token in m. A workflow graph is said to be *sound* if it has no *local deadlock* and no unsafe reachable marking. Soundness guarantees that every fair execution terminates in the final marking of Γ. Soundness has various equivalent characterizations and can be decided in polynomial time [5, 1].

We now equip each token with an integer-valued clock initialized to zero. Then the *state* of the workflow graph is given by the tuple (m, c) where m is the marking and $c : m \to \mathbb{N}$ (note that for safe workflow graphs, $m : E \to \{0, 1\}$, hence m is a subset of E). We carry over the token-game semantics for clocks and we set $(m, c) \xrightarrow{T} (m', c')$, when $m \xrightarrow{T} m'$ and $c'(e) = c(e)$ for $e \in m' \setminus T^{\bullet}$ and $c'(e) = w(e) + \max\{\, c(e') \mid e' \in {}^{\bullet}T\}$ for $e \in T^{\bullet}$.

In the initial marking, the state of the workflow graph is given by (m_s, c_s), where $c_s(e_{source}) = w(e_{source})$. Similarly, in the final marking, the state of the workflow graph is given by (m_f, c_f). We then define the *duration of an execution* σ as $c_f(e_{sink})$, where σ ends in the final marking m_f.

Let Γ be a WFG; Γ is *sequential* if it contains no AND-split and no -join. It is *acyclic* if the underlying graph has no cycles. A *regular* workflow graph is a workflow graph that can be generated from a regular expression as follows. Let ϵ be a constant symbolizing an edge and X, Y variables for workflow graphs. Then a regular workflow graph expression is the smallest set such that ϵ is a regular workflow graph, and if X and Y are regular workflow graphs, then $X\ ;\ Y$, X AND Y, X XOR Y, and X LOOP Y are also regular workflow graphs. From each regular workflow graph expression, we can generate a workflow graph, where each expression type corresponds to one of the graph fragment patterns shown in Fig. 2 and composition is done by replacing an edge labeled with a variable by another pattern. For example, the expression $((\epsilon;\epsilon)$ *AND*

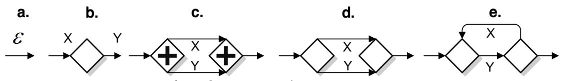

Fig. 2: Regular patterns

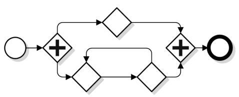
Fig. 3: Regular graph

$(\epsilon$ *LOOP* $\epsilon))$ generates the graph shown in Fig. 3. Note that the loop construct has two loop bodies. It can be viewed as a combination of a while and a repeat loop, one loop body before the loop condition one after it. It can be decided in linear time whether a workflow graph is a regular workflow graph using graph parsing techniques [14].

3 Workflow graphs with nondeterministic choice

In this section, we present our first main contribution, a polynomial time algorithm that computes the minimum execution time of a workflow graph, which can be used to determine whether some fair execution of a time annotated workflow graph with an unbounded number of resources meets the deadline.

3.1 The minimum duration of a workflow graph

We start by presenting several preliminary notions that are necessary for the algorithm. We introduce the *accumulated cost associated with an edge* in a fair execution. Based on our definition of the accumulated cost associated with an edge, the cost accumulated on the *source* edge represents the cost of a fair execution. Next, we present an algorithm to compute the *minimum* cost accumulated on the *source* edge, this equals the *minimum* cost of a fair execution of a given workflow graph and prove its correctness.

In the following, let Γ be a sound workflow graph.

To facilitate the computation of the cost accumulated on an edge in a fair execution σ, we express the execution as the sequence of edges that get marked in σ. To introduce an unambiguous representation, we use $\tau_\sigma = \langle T_0, T_1, \cdots, T_n \rangle$, the transition sequence that corresponds to σ. The sequence of edges that get marked in σ is given by $\langle e_{source}, T_0^\bullet, T_1^\bullet, \cdots, T_n^\bullet \rangle$, where each set $T_k^\bullet$ such that $|T_k^\bullet| > 1$ is ordered in a fixed predefined order (e.g., alphabetic). We use the notation $\sigma = \langle e_{source}, \cdots, e_i, \cdots, e_{sink} \rangle$. Since we are interested in fair executions (and we assume soundness), the sequence of edges is finite and ends with e_{sink}.

Having the sequence of edges that are marked in σ, we traverse the sequence backwards, from the last to the first edge in the sequence and *update* the cost of an edge $e \in {}^\bullet v$ at position i in the sequence based on the cost already computed for the edges in the sequence that belong to $v^\bullet$.

As an example, consider the workflow graph in Fig. 4. In Fig. 4, edges are labeled (e.g. $e8$; 2) with an edge name ($e8$) and a duration (2). Fig. 5 represents the workflow graph restricted to the elements that are contained in the fair execution with minimum duration, i.e., it is a representation of the minimum duration execution. Each edge in Fig. 5 is labeled with the accumulated cost for reaching the sink in that execution.

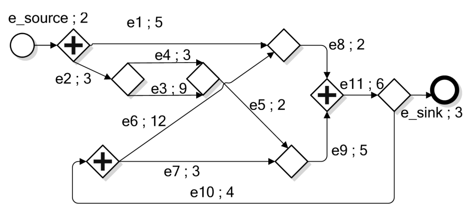

Fig. 4: Workflow graph with edge weights

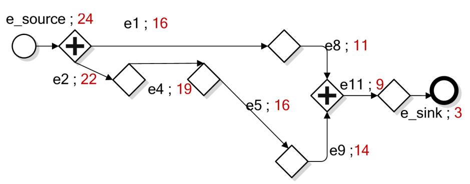

Fig. 5: Minimum duration execution and the accumulated costs

For e_{11}, the accumulated cost to reach the sink is: $w(e_{11})$ to which we add the cost of e_{sink} therefore, $6 + 3 = 9$. Based on our update rule for AND-join nodes, the cost associated to e_9 becomes $w(e_9)$ plus the cost of e_{11} and we obtain 14 and the cost associated to e_8 becomes $w(e_8)$ plus the cost of e_{11}, and we get 2+9=11. For edges e_5 and e_1, we update the cost by adding the edge weight to the accumulated cost on the outgoing edge of the XOR-split, and we obtain costs 16 (11+5) for $e1$ and 16 (14+2) for e_5. We apply the same rule for e_4 and we obtain an accumulated cost of 19 (16+3) and subsequently also for e_2 and we obtain 22 (19+3). Now we can compute the cost of the execution. Note that the AND-join we are about to process spawns two threads. The cost of the execution is decided by the longest thread (in terms of duration). Therefore, we update the cost accumulated on e_{source} to be equal to $w(e_{source}) + max(16, 22)$ which equals 24 and this equals the cost of the execution.

Now we present formally how to compute the accumulated cost associated with an edge for a given execution. Let e^i be the edge at position i in the sequence of edges that get marked in the execution.

Since we update based on the edges in $v^\bullet$, for the XOR nodes, we define a function $next_\sigma(e^i)$ such that for the edge at position i, $e^i \in {}^\bullet v$, it returns the edge in $v^\bullet$ that get marked next after e^i gets marked.

For each position i in the sequence of edges that get marked in the execution, starting from the last index, we update the cost of the edge e^i, which we denote by $d_\sigma(e^i)$:

$$d_\sigma(e^i) = \begin{cases} w(e^i) & \text{if } e^i = e_{sink} \\ w(e^i) + d_\sigma(next_\sigma(e^i)) & \text{if } l(v) = \text{XOR} \\ w(e^i) + \max\{d_\sigma(e') \mid e' \in v^\bullet\} & \text{if } l(v) = \text{AND and } |v^\bullet| > 1 \\ w(e^i) + d_\sigma(e') & \text{if } l(v) = \text{AND and } \{e'\} = v^\bullet \end{cases}$$

Note that this procedure may update the cost of an edge e multiple times in case the execution is cyclic, i.e. executes an edge multiple times. As the final accumulated cost associated with the edge e in σ, we take the value of $d_\sigma(e)$ after the last update.

Since e_{source} is always the first edge in the sequence of edges that get marked in a fair execution, it follows that $e^0 = e_{source}$ and $d_\sigma(e^0) = d_\sigma(e_{source})$. Since the computation of $d_\sigma(e)$ follows the semantics of workflow graphs, it is easy to see that $d_\sigma(e_{source}) = c(\sigma)$, the duration of the execution σ.

The algorithm for computing the minimum duration of a fair execution of a workflow graph with an unbounded number of resources, Algorithm 1, is given below. It works on a weighted workflow graph, and for each node v, and each edge $e \in {}^\bullet v$, it updates a value $\delta(e)$ that represents the currently known *minimum cost* to reach the sink from e based on relaxation rules specific to each node type (see Algorithm 3). All edge costs are updated at most $|V|$ times for a cyclic workflow graph (see Algorithm 1) and only once for an acyclic workflow graph (see Algorithm 2). Upon termination of our algorithm, the value associated to e_{source}, $\delta(e_{source})$, represents the duration of the minimum duration execution.

The outer loop of the algorithm is similar to the Bellman-Ford algorithm [2] for sequential graphs, but the parallel constructs entail a different relaxation procedure to

Algorithm 1 Minimum duration

```
function WFGMin( V, E)
    for $e \in E \setminus \{e_{sink}\}$ do
        $\delta(e) \leftarrow \infty$
    end for
    $\delta(e_{sink}) \leftarrow w(e_{sink})$
    for $i = 1 : |V|$ do
        for all $e \in E$ do
            $u, v \leftarrow$ nodes s.t. $e = c(u, v)$
            Relax(e,v)
        end for
    end for
end function
```

Algorithm 2 Min duration, acyclic

```
function AcyclicWFGMin(V, E)
    for $e \in E \setminus \{e_{sink}\}$ do
        $\delta(e) \leftarrow \infty$
    end for
    $\delta(e_{sink}) \leftarrow w(e_{sink})$
    TopologicalSort($\Gamma$)
    while $V \neq \emptyset$ do
        Select $v \in V$ s.t. $v$ is maximal with respect to the topological sort
        $V \leftarrow \{V \setminus v\}$
        for all $e \in {}^\bullet v$ do
            Relax(e,v)
        end for
    end while
end function
```

Algorithm 3 Relaxation of an edge $e \in {}^\bullet v$

```
function Relax(e,v)
    if $l(v)$ = XOR and $\{e'\} = v^\bullet$ then
        if $\delta(e) > w(e) + \delta(e')$ then
            $\delta(e) \leftarrow w(e) + \delta(e')$
        end if
    end if
    if $l(v)$ = XOR and $|v^\bullet| > 1$ then
        if $\delta(e) > w(e) + min_{e' \in v^\bullet}(\delta(e'))$ then
            $\delta(e) \leftarrow w(e) + min_{e' \in v^\bullet}(\delta(e'))$
        end if
    end if
    if $l(v)$ = AND and $\{e'\} = v^\bullet$ then
        if $\delta(e) > w(e) + \delta(e')$ then
            $\delta(e) \leftarrow w(e) + \delta(e')$
        end if
    end if
    if $l(v)$= AND and $|v^\bullet| > 1$ then
        if $\delta(e) > w(e) + \max\{\delta(e') \mid e' \in v^\bullet\}$ then
            $\delta(e) \leftarrow w(e) + \max\{\delta(e') \mid e' \in v^\bullet\}$
        end if
    end if
end function
```

reflect the semantics of sound workflow graphs. In addition, the correctness proofs are more complex due to the characteristics of workflow graphs.

Next, we will show the correctness of the algorithm. For this we introduce the definition of the *minimum cost* that can be accumulated on an edge. This is necessary for the proofs, as we will demonstrate that the algorithm computes the minimum cost accumulated on the source edge.

Let e be an edge of Γ and v a node of Γ such that $e \in {}^\bullet v$. We define the *edge enabling marking* m_e, as the reachable marking for which $m_e(e) = 1$, v is enabled in m_e and no other node is enabled in m_e. It has been shown [8] that for a sound workflow graph, the edge enabling marking is unique.

We define $d^*(e)$, *the minimum cost downstream from* e, as follows:

$$d^*(e) = \min\{d_\sigma(e) \mid \sigma \text{ is a fair execution that starts in } m_e \}. \quad (1)$$

Because Γ is sound, note that since m_e is a reachable marking, it holds that $m_e \overset{*}{\rightarrow} m_f$.

Since $d_\sigma(e_{source})$ represents the cost of a fair execution σ, $d^*(e_{source})$ represents the duration of the minimum duration execution.

Lemma 1 *Let e be an edge and v a node such that $e \in {}^\bullet v$. We always have $\delta(e) \geq d^*(e)$.*

The proof of Lemma 1 is presented in the Appendix.

Lemma 2 *Let e be an edge. Let σ be a fair execution such that $d_\sigma(e) = d^*(e)$. Let $S = \langle e_{i-1}, \cdots, e_{sink} \rangle$ be the sequence edges that get marked after e gets marked for the last time in σ. Each sequence of calls of $Relax(e, v)$ that has the property that edges $e_{sink}, \cdots, e_{i-1}, e$ have been relaxed in this order, after the sequence of calls to $Relax(e, v)$ we have $\delta(e) = d^*(e)$.*

The proof of Lemma 2 is presented in the Appendix.

Definition 1 *A fair execution σ of Γ, is a* loop-free execution *if no node is executed more than once in σ, and therefore no edge is marked more than once in σ.*

Lemma 3 *Some fair execution of Γ with minimum duration is loop-free.*

The proof of Lemma 3 is presented in the Appendix.

For a workflow graph Γ and a fair, loop-free execution σ of Γ, we define Γ_σ as the workflow graph Γ restricted to σ such that it contains only the nodes of Γ that are executed in σ and the edges of Γ such that $\sigma(e) = 1$. For a fair, loop-free execution σ of Γ, it follows that Γ_σ is an acyclic workflow graph.

The elements of an acyclic workflow graph are in a partial order defined by the flow of the graph: Let $G = (V, E, c)$ be an acyclic multi-graph. If x_1, x_2 are two elements in $V \cup E$ such that there is a path from x_1 to x_2, then we say that x_1 precedes x_2, denoted $x_1 \leq x_2$, and x_2 follows x_1.

Lemma 4 *For a sound workflow graph, after running the Algorithm 1, it holds that $\delta(e_{source}) = d^*(e_{source})$.*

Proof: Lemma 3 states that some fair execution of Γ, with minimum duration, is loop-free (i). Recall that for a given fair execution σ, $d_\sigma(e_{source})$ represents the duration of execution of σ (ii). From (i) and (ii) it follows that some execution that minimizes $d_\sigma(e_{source})$ is loop-free (iii).

Note that m_s is the edge enabling marking for e_{source}.

Using (iii) and the definition for $d^*(e)$ instantiated to e_{source}, we obtain:

$d^*(e_{source}) = \min\{d_\sigma(e_{source}) \mid \sigma$ is a fair execution that starts in m_s $\}$. It follows that some σ^* for which $d_{\sigma^*}(e_{source}) = d^*(e_{source})$, is a fair, loop-free execution.

Since σ^* is loop-free, it means that at most $|V|$ nodes are executed in σ^*. In each complete relaxation step (one iteration of the loop in line 6 in Algorithm 1), we relax all the edges. Therefore, at the $|V|$-th iteration we have relaxed all the edges, in decreasing order with respect to the partial order on the edges of Γ_{σ^*}. It means that at the $|V|$-th iteration, we will have relaxed all the edges that get marked after e gets marked in σ^*. Therefore, from Lemma 2, $\delta(e) = d^*(e)$.

Therefore, we computed the duration of the minimum duration execution of the workflow graph, which is $d^*(e_{source})$.

For Algorithm 1, the initialization of the edge costs takes $O(|V|)$ time and each of the $|V|$ iterations over the edges of the workflow graph is performed in $O(|E|)$ time. The cost update is performed in constant time. Hence, we have proven the following:

Theorem 1 *The minimum duration execution of a sound workflow graph with unbounded number of resources can be computed in time $O(|V||E|)$.*

3.2 Regular and acyclic workflow graphs

In the following, we briefly present the ideas for computing the maximum duration of execution for regular and acyclic workflow graphs.

As presented in [3], for a regular workflow graph with a structured cycle, i.e., a while or repeat loop, or more general, of the form X LOOP Y, the computation of the maximum duration requires the specification of the maximal number of iterations for each loop. If we assume that the backedge of each loop (i.e., edge "x" in Fig. 2) of the regular graph is annotated with a positive integer k that represents the maximum number of times the backedge can be traversed, then the maximum duration of X LOOP Y is $(k+1) \cdot d_X + k \cdot d_Y$ where d_X denotes the maximum duration of the loop body X, and d_Y represents the maximal duration associated to reentering the loop. For computing the minimum duration we take $k = 0$ and the minimum duration of the loop body. We still obtain the minimum/maximum duration of such an annotated regular workflow graph in linear time (Cell C.1, C.2 of Table 1).

For acyclic workflow graphs, we can use the algorithm for the cyclic case but without the need to perform $|V|$ iterations. Instead we exploit the fact that the elements of an acyclic workflow graph are in a partial order defined by the flow of the graph. Therefore, in order to make sure that the edges are relaxed respecting the partial order, first, the graph is sorted topologically - $O(|V| + |E|)$. Secondly, the edges are relaxed in descending order with respect to the topological sorting in $O(|E|)$ time. The algorithm that formalizes this idea is Algorithm 2.

Theorem 2 *The minimum duration execution of a sound acyclic workflow graph with unbounded number of resources can be computed in time $O(|V| + |E|)$.*

Note that, in the acyclic case, for computing the maximum duration execution, one only needs to select the maximum instead of the minimum in the *Relax*(e, v) procedure when $l(v) = \text{XOR}$ and $|v^\bullet| > 1$.

4 Workflow graphs with probabilistic choice

If not all fair executions of a workflow graph meet the deadline, we could ask whether at least a large portion of the fair executions does. We approach this question by assuming that decisions are resolved through a coin flip, i.e., each XOR-node v is assigned a distribution $\mu : v^\bullet \to [0,1]$ such that $\mu(e) > 0$ for each $e \in v^\bullet$ and $\sum_{e \in v^\bullet} = 1$. Although some fair executions may not terminate, their probability[3] is zero. We can then take the duration of an execution as a random variable and ask whether the probability of an execution terminating before the deadline exceeds a given threshold.

4.1 Expected duration

In the following, we will present our result for the complexity of computing the expected duration of a workflow graph.

Theorem 3 *Given a regular, acyclic probabilistic workflow graph Γ, computing the expected duration of Γ executed by an unbounded set of resources is NP-hard.*

The proof consists of a reduction from the *subset sum problem* which is the problem: given a set $D = \{d_1, \cdots, d_n\}$ of integers and an integer S, to determine whether any non-empty subset $D' \subseteq D$ sums up to exactly S. This problem is known to be NP-hard. This is equivalent to solving a problem where all the values $d_1, \cdots, d_n, S$ are multiples of 4 (this statement will be used in the proof later on). For the proof, we use the class of (regular, acyclic) probabilistic workflow graphs $\Gamma_{\epsilon,n}$ in Fig. 6, where each decision outcome has probability 0.5.

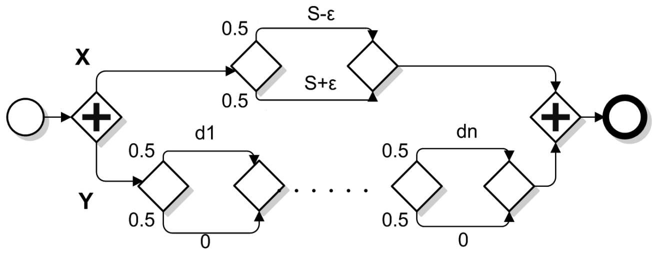

Fig. 6: A probabilistic workflow graph

Proof. Let X, Y be random variables that denote the duration of each of the two parallel flows of $\Gamma_{\epsilon,n}$. The expected duration of the workflow graph $\Gamma_{\epsilon,n}$ is:

$$\mathbb{E}(\Gamma_{\epsilon,n}) = \mathbb{E}(max(X, Y))$$

$$\mathbb{E}(\Gamma_{\epsilon,n}) = \tfrac{1}{2}\mathbb{E}(max(S - \epsilon, Y)) + \tfrac{1}{2}\mathbb{E}(max(S + \epsilon, Y))$$

Let f be the probability distribution of Y. We rewrite the terms of $\mathbb{E}(\Gamma_{\epsilon,n})$ as follows:

$$\mathbb{E}(max(Y, S - \epsilon)) = (S - \epsilon)\Pr(Y \leq S - \epsilon) + \sum_{y>S-\epsilon} yf(y) \tag{1}$$

[3] We do not explicitly construct the probability space here on which the development of this chapter is formally based on. As workflow graphs contain concurrency, we need to consider maximal partial-order executions to obtain a single probability space and to avoid the notion of an adversary as in Markov decision processes. Note that a probabilistic workflow graph does not contain real non-determinism, just concurrency. The construction of such a probability space is provided elsewhere [16, 17], e.g. for Petri nets and in fact rests on the assumption that the Petri net is free-choice. In this paper, we are only concerned with the duration of an execution, which is independent of the interleaving, i.e., the ordering of concurrent events.

$$\mathbb{E}(max(Y, S+\epsilon)) = (S+\epsilon)\Pr(Y \leq S+\epsilon) + \sum_{y>S+\epsilon} yf(y) \tag{2}$$

By using equations (1), (2) we obtain the following expression for $\mathbb{E}(\Gamma_{\epsilon,n})$:

$$\mathbb{E}(\Gamma_{\epsilon,n}) = \tfrac{1}{2}\Big[(S+\epsilon)\Pr(Y \leq S+\epsilon) + \sum_{y>S+\epsilon} yf(y) + (S-\epsilon)\Pr(Y \leq S-\epsilon) + \sum_{y>S-\epsilon} yf(y)\Big].$$

Let us choose $\epsilon > 0$ such that no subset of $\{d_1, \cdots, d_n\}$ has sum in $[S-\epsilon, S)$ nor in $(S, S+\epsilon]$. Note that the sum, can still potentially equal exactly S. Such ϵ is easy to find. It is enough to choose $\epsilon = 2$ as all the numbers $d1, \cdots, d_n, S$ are multiples of 4.

We will show that $\mathbb{E}(\Gamma_{\epsilon,n}) = \mathbb{E}(\Gamma_{\frac{\epsilon}{2}})$ if there is no non-empty subset of $\{d_1, \cdots, d_n\}$ that sums up to exactly S (i), and $\mathbb{E}(\Gamma_{\epsilon,n}) \neq \mathbb{E}(\Gamma_{\frac{\epsilon}{2}})$ otherwise (ii). If we can compute the expected duration of a workflow graph with unbounded resources in polynomial time we can solve the subset sum problem in polynomial time. Note that both ϵ and $\epsilon/2$ are integers, so we are always considering workflow graphs with integer weights.

(i) There is no non-empty subset of $\{d_1, \cdots, d_n\}$ that sums up to exactly S

.

In this case, it holds that $\Pr(Y \leq S-\epsilon) = \Pr(Y \leq S+\epsilon)$. Therefore we update the equation for $\mathbb{E}(\Gamma_{\epsilon,n})$):

$$\mathbb{E}(\Gamma_{\epsilon,n}) = \tfrac{1}{2}\Big[\Pr(Y \leq S+\epsilon)(S+\epsilon+S-\epsilon) + \sum_{y>S-\epsilon} yf(y) + \sum_{y>S+\epsilon} yf(y)\Big].$$

$\mathbb{E}(\Gamma_{\epsilon,n}) = \frac{1}{2}\Big[2S\Pr(Y \leq S+\epsilon) + \sum_{y>S-\epsilon} yf(y) + \sum_{y>S+\epsilon} yf(y)\Big]$. One can easily observe that $\mathbb{E}(\Gamma_{\epsilon,n}) = \mathbb{E}(\Gamma_{\frac{\epsilon}{2}})$.

(ii) There exists a non-empty subset of $\{d_1, \cdots, d_n\}$ that sums up to exactly S

.

In this case, $\Pr(Y \leq S-\epsilon) \neq \Pr(Y \leq S+\epsilon)$. Therefore,

$$\mathbb{E}(\Gamma_{\epsilon,n}) = \tfrac{1}{2}\Big[(S+\epsilon)\Pr(Y \leq S+\epsilon) + (S-\epsilon)\Pr(Y \leq S-\epsilon) + \sum_{y>S-\epsilon} yf(y) + \sum_{y>S+\epsilon} yf(y)\Big].$$

$$\mathbb{E}(\Gamma_{\epsilon,n}) = \tfrac{1}{2}\Big[(S+\epsilon)(\Pr(Y \leq S-\epsilon) + \Pr(Y = S)) + (S-\epsilon)\Pr(Y \leq S-\epsilon) + \sum_{y>S-\epsilon} yf(y) + \sum_{y>S+\epsilon} yf(y)\Big].$$

$$\mathbb{E}(\Gamma_{\epsilon,n}) = \tfrac{1}{2}\Big[\underbrace{2S\Pr(Y \leq S-\epsilon)}_{T_1} + \underbrace{(S+\epsilon)\Pr(Y = S)}_{T_2} + \underbrace{\sum_{y>S-\epsilon} yf(y) + \sum_{y>S+\epsilon} yf(y)}_{T_3}\Big].$$

Please note that term T_2 has different value for $\mathbb{E}(\Gamma_{\epsilon,n})$ and $\mathbb{E}(\Gamma_{\frac{\epsilon}{2}})$, while T_1 and T_3 have the same value for $\mathbb{E}(\Gamma_{\epsilon,n})$ and $\mathbb{E}(\Gamma_{\frac{\epsilon}{2}})$. Therefore, $\mathbb{E}(\Gamma_{\epsilon,n}) \neq \mathbb{E}(\Gamma_{\frac{\epsilon}{2}})$.

5 Minimum number of resources

In this section, we compute the maximum *degree of concurrency* of Γ, i.e., the maximum number of tokens that can exist in the graph in a reachable marking. This can help in answering a natural question that arises in the quantitative timing analysis of a business process. What is the minimum number k^* of resources one needs, such that each execution achieves its minimal execution time? This means there does not exist any execution for which the duration could be decreased by having more than k^* resources. The maximum number of tokens that can exist in the graph is an upper bound for k^*. There are cases where tighter bounds can be given, as illustrated in Figure 7 where the maximum number of tokens is 3, obtained in the marking that marks edges e_2, e_3 and e_4 but 2 resources would suffice for reaching the minimum duration, i.e., 15.

Before presenting the algorithm we need to introduce one more subclass of workflow graphs.

A workflow graph Γ is a *marked graph* if any node $v \in \Gamma \setminus \{v_{source}, v_{sink}\}$ is either an AND-node or an XOR-node with a single incoming and a single single outgoing edge.

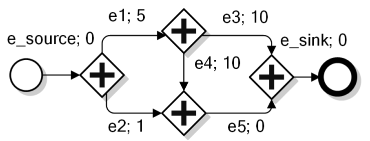

Fig. 7: Tighter bound example

There is an EXPTIME algorithm for computing the maximum degree of concurrency for general worklfow graphs. It is based on computing the reachability graph of Γ, which is the transition relation $\rightarrow$ restricted to its reachable markings. Note that for sound workflow graphs the reachability graph is finite, but exponential in the size of Γ. Each reachable marking is visited to compute the maximum concurrency degree.

However, efficient algorithms are known for subclasses such as marked graphs, regular or sequential worklfow graphs. Therefore, we propose to leverage this fact and tackle the problem through a *divide and conquer* strategy. This approach has the potential of speeding up the computation of the maximum degree of concurrency of a work in practice.

In order to divide the problem into smaller parts, we compute the Refined Process Structure Tree (RPST) [14] of the workflow graph. The RPST represents a decomposition of a workflow graph into a hierarchy of sub-workflows that are subgraphs with a single entry and a single exit of control called *fragments* (see e.g., Figure 8 (a)). The decomposition results in a parse tree which reflects the containment relationship of the fragments.

The algorithm for computing the maximum degree of concurrency works as follows: (1) divide the problem of computing the maximum degree of concurrency to subproblems by decomposing the workflow graph into its fragments. These fragments are labeled with their corresponding subclass (e.g., marked graph). (2) conquer the problem by computing the maximum degree of concurrency of the workflow graph based on the maximum degree of concurrency computed for its fragments. Note that we omit here trivial fragments consisting of a single edge.

The complexity of the algorithm depends on the subclass of the fragment f, as follows. It runs in linear time for sequential workflow graphs, where the returned value is the maximum weight of an edge of this fragment. Similarly it runs in linear time for regular fragments. For regular fragments modeling concurrency (cf. Figure 2.c) the maximum degree of concurrency is the *sum* of the weights of the edges. For regular fragments modeling choice (cf. Figure 2.b,d,e) the maximum degree of concurrency is the *maximum* of the weights of the edges. The computation of the concurrency degree runs in polynomial time for marked graphs and in exponential time for the *complex fragments* – the fragments which are not regular nor marked-graphs nor state-machines.

An example for how our algorithm works is provided in Figure 8. In Figure 8, edge weights represent the concurrency degree. The root fragment is *Sequence Fragment 2* and the tree has one leaf – the *Marked Graph Fragment*, Figure 8 (a). After computing the concurrency degree of the marked graph (value 3) the work is updated as shown in Figure 8 (b). In the next iteration, the wor is reduced to the e wo composed of a regular fragment contained in a sequence fragment, as shown in Figure 8 (c). The concurrency degree of the regular fragment is computed (we obtain 3+1=4), we update the he w and we are left with a sequence fragment Figure 8 (d). The maximum degree of concurrency of the the is the concurrency degree of this fragment (4).

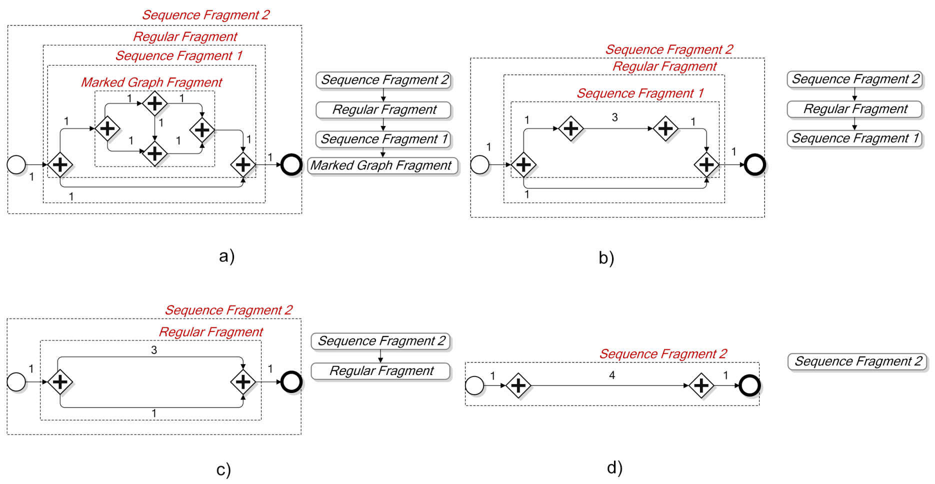

Fig. 8: An example of a work decomposition into its fragments and its corresponding RPST (a), the wor and RPST after computing the concurrency degree for the *Marked Graph Fragment* (b) the e wo and RPST after computing the concurrency degree for *Sequence Fragment 1* (c) the he w and RPST before the algorithm ends (d)

In the following, we present the approaches for computing the maximum degree of concurrency for marked graphs and for complex fragments.

Let $\boldsymbol{w}$ denote a $|E| \times 1$ column vector representing the degree of concurrency associated with each edge of Γ. Finding the maximum degree of concurrency of a marked graph Γ, $deg(\Gamma)$, can be formulated as:

$$deg(\Gamma) = max\{m \cdot \boldsymbol{w} \mid m \text{ is a marking reachable from } m_0\} \quad (1)$$

The solution we propose for computing deg(Γ) in a marked graph is identical to the computation of the maximum weighted sum of tokens in [12]. In [12] the author formulates this problem as an integer programming (IP) problem with integer data and totally unimodular constraint matrix. Note that any IP problem with with integer data and totally unimodular constraint matrix is solvable in polynomial time.

The complexity of the algorithm is therefore dominated by complex fragments, for which we resort to the EXPTIME algorithm. Since complex fragments are rare in practice, this approach can be efficient in computing the maximum degree of concurrency. In a previous study documented in [15] on 645 industrial business process which were translated to workflow graphs, only about 4% of the total of their corresponding fragments were complex with an average number of edges between 21 and 32.

6 Conclusion

We presented new results on the deadline analysis of workflow graphs with an unbounded number of resources.

The same questions can be asked in settings with a fixed number $n > 1$ of resources. This constraint leads to problems that can not be solved in polynomial time. The probability of deadline transgression and the expected duration remain NP-hard which is easy to see from our justifications in the current work. For the maximum duration – the worst case execution time is attained when we require all the tasks to be executed by a single resource, which we have studied in [3]. What is different, is the fact that computing the minimum duration of execution becomes NP-hard for a fixed number $n > 1$ of resources. For example, for a simple workflow graph as the one in Figure 9, let's assume we need to complete n tasks $T_1, \cdots, T_n$ and we have k identical agents to solve them. Finding an assignment of the tasks to the agents such that the duration of execution (*makespan*) is minimized is NP-hard as one can reduce 2-PARTITION to finding the minimum duration when there are exactly two resources available [7].

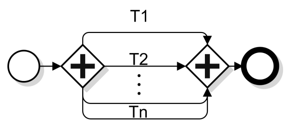

Fig. 9: Regular WFG with n parallel threads

In future work, we would like to investigate further whether computing the maximum degree of concurrency of a WFG is NP-hard or a polynomial time algorithm exists.

References

1. W. M. P. Aalst, A. Hirnschall, and H. M. W. Verbeek. *Advanced Information Systems Engineering: 14th International Conference, CAiSE 2002 Toronto, Canada, May 27–31, 2002 Proceedings*, chapter An Alternative Way to Analyze Workflow Graphs, pages 535–552. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
2. R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.
3. M. Botezatu, H. Völzer, and L. Thiele. The complexity of deadline analysis for workflow graphs with a single resource. In *Proceedings of the 20th IEEE ICECCS Conference*, December 2015.

4. S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.
5. J. Desel and J Esparza. *Free Choice Petri Nets*. Cambridge University Press, New York, NY, USA, 1995.
6. C. Favre, D. Fahland, and H. Völzer. The relationship between workflow graphs and free-choice workflow nets. *Inf. Syst.*, 47:197–219, 2015.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
8. B. Gaujal, S. Haar, and J. Mairesse. Blocking a transition in a free choice net and what it tells about its throughput. *Journal of Computer and System Sciences*, 66(3):515 – 548, 2003.
9. H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 102–111, Dec 1989.
10. M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 585–591, Berlin, Heidelberg, 2011. Springer-Verlag.
11. H. Mili, G. Tremblay, G. Jaoude, É. Lefebvre, L. Elabed, and G. El Boussaidi. Business process modeling languages: Sorting through the alphabet soup. *ACM Comput. Surv.*, 43(1):4:1–4:56, December 2010.
12. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.
13. L. Popova-Zeugmann and M. Heiner. Worst-case analysis of concurrent systems with duration interval petri nets. In *BTU COTTBUS*, pages 162–179, 1996.
14. J. Vanhatalo, H. Völzer, and J. Koehler. The refined process structure tree. *Data Knowl. Eng.*, 68(9):793–818, 2009.
15. J. Vanhatalo, H. Völzer, and F. Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In Bernd Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer Berlin Heidelberg, 2007.
16. D. Varacca, H. Völzer, and Glynn Winskel. Probabilistic event structures and domains. *Theor. Comput. Sci.*, 358(2-3):173–199, 2006.
17. H. Völzer. Randomized non-sequential processes. In *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, pages 184–201, 2001.
18. M. Wan and G. Ciardo. Symbolic reachability analysis of integer timed petri nets. In *SOFSEM 2009: Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 595–608. Springer Berlin Heidelberg, 2009.
19. C.-Q. Yang and B.P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 366–373, Jun 1988.

Appendix

Preliminaries for proofs of different lemmas

Several of the lemmas we will present in the following, are originally phrased in the context of free choice Petri Nets. Due to the equivalence between workflow graphs and free choice Petri Nets stated in [6], the results hold for workflow graphs as well.

The *Parikh vector* of a transition sequence τ, written $\vec{\tau}$, maps every transition T to the number of occurrences of T in τ. More formally, it is the multi-set of transitions such that $\vec{\tau}\ (T) = k$ if T appears exactly k times in τ.

We say a transition T is included in an execution σ, denoted $T \in \sigma$, if $\vec{\tau_\sigma}\ [T] > 0$. We write $\vec{\sigma}$ instead of $\vec{\tau_\sigma}$, henceforth, for convenience.

For two vectors $\vec{\tau_1}$ and $\vec{\tau_2}$ we set $\vec{\tau_1} \leq \vec{\tau_2}$ if $\forall\ T$, $\vec{\tau_1}\ [T] \leq \vec{\tau_2}\ [T]$.

Two transition sequences τ_1 and τ_2 of Γ are *permutations* of each other if $\vec{\tau_1} = \vec{\tau_2}$.

Let *Trans*(v) denote the set of transitions of a node v. *Trans*$(v) = \{T \mid T = (E_1, v, E_2)\}$.

The *incidence matrix* $\mathbf{N}$ of a workflow graph is a matrix whose rows represent the edges of the workflow graph, and the columns represent the transitions of the workflow graph. The entry $\mathbf{N}(i, j)$ corresponds to the change of the marking of the edge i caused by the occurrence of the transition $j = (E_1, v, E_2)$.

$$\mathbf{N}(i, j) = \begin{cases} -1 & \text{if } i \in E_1 \setminus E_2 \\ 1 & \text{if } i \in E_2 \setminus E_1 \\ 0 & \text{otherwise} \end{cases}$$

We will use the following marking equation lemma [5]:

Lemma 5 (Marking equation lemma) *For every finite transition sequence τ, of a workflow graph with the incidence matrix $\mathbf{N}$, with $m \xrightarrow{\tau} m'$, the following equation holds:*

$$m' = m + \mathbf{N} \cdot \vec{\tau}$$

Let τ_2 be a permutation of the transition sequence τ_1. If $m_0 \xrightarrow{\tau_1} m$ and $m_0 \xrightarrow{\tau_2} m'$, then it follows from the marking equation lemma that $m = m'$.

Proof of Lemma 1

Lemma 1 *Let e be an edge and v a node such that $e \in {}^\bullet v$. We always have $\delta(e) \geq d^*(e)$.*

We prove the lemma by induction on k, the number of calls of *Relax*(e, v).

Base case: $k = 0$: $\delta(e_{sink}) = w(e_{sink})$, therefore clearly, $\delta(e_{sink}) = d^*(e_{sink})$, and for all $e \in E \setminus \{e_{sink}\}$, $\delta(e) = \infty$, and therefore $\delta(e) > d^*(e)$.

Induction step: Suppose that after the k-th call of *Relax*(e, v) we have $\delta(e) \geq d^*(e)$ for all e. At the $(k + 1)$-th call of *Relax*(e, v), only $\delta(e)$ may get updated.

We will show that from the definition of $d^*(e)$ and from the induction hypothesis, it follows that $\delta(e) \geq d^*(e)$, for each of the relaxation cases.

For the case $l(v)$ = XOR and $|v^\bullet| > 1$ the proof relies on Lemma 1.1, and for the case $l(v)$ = AND and $|v^\bullet| > 1$ the proof relies on Lemma 1.2.

We will present the reasoning for one of the relaxation cases, as the justification for the remaining ones is similar.

Let $l(v)$ = XOR and $|v^\bullet| > 1$. Let ${}^\bullet v = \{e\}$ Before the $(k+1)$-th relaxation step, it holds that $\delta(f) \geq d^*(f)\ \forall f \in v^\bullet$ (due to the induction hypothesis). After the $(k+1)$-th relaxation step, $\delta(e)$ gets updated, such that $\delta(e)$ becomes $w(e) + min\{\delta(f) \mid f \in v^\bullet\}$, as presented in Algorithm 3. Therefore, due to the induction hypothesis, $\delta(e) \geq w(e) + min\{d^*(f) \mid f \in v^\bullet\}$ (i).

From Lemma 1.1 we have that $d^*(e) = w(e) + min\{d^*(f) \mid f \in v^\bullet\}$ (ii).

From (i) and (ii) it follows that $\delta(e) \geq d^*(e)$.

Note that at each relaxation step we can only decrease the value of $\delta(e)$. Once $\delta(e) = d^*(e)$, it doesn't change (it can not decrease further) as otherwise it would contradict the claim that $\delta(e) \geq d^*(e)$.

Lemma 1.1 *Let v be a node such that $l(v)$ = XOR and $v^\bullet > 1$. Let $\{e\} = {}^\bullet v$. We have $d^*(e) = w(e) + min\{d^*(f) \mid f \in v^\bullet\}$.*
Proof:

From the definition of $d_\sigma(e)$ we have:$d_\sigma(e) = w(e) + d_\sigma(next_\sigma(e))$ (such that this is the last update of $d_\sigma(e)$). Also by definition, we have: $d^*(e) = min\{d_\sigma(e) \mid \sigma$ is a fair execution that starts in $m_e\}$. Therefore, $d^*(e) = min\{w(e) + d_\sigma(next_\sigma(e)) \mid \sigma$ is a fair execution that starts in $m_e\}$.
$d^*(e) = w(e) + min\{d_\sigma(next_\sigma(e)) \mid \sigma$ is a fair execution that starts in $m_e\}$.

Note that σ is an execution like: $\langle e\ f \cdots \rangle$ where $f \in v^\bullet$. Let σ_f be the execution when $f = next_\sigma(e)$.

We can re-write the definition of $d^*(e)$:
$d^*(e) = w(e) + min_{f \in v^\bullet} min\{d_\sigma(f) \mid \sigma$ is a fair execution that starts in m_e $\}$.

Recall that σ started in m_e. Upon executing v, we obtain the marking m_f. Note that since m_e is an edge enabling marking, m_f is also an edge enabling marking. This can be easily verified by analyzing the different possibilities for $l(v')$ where $f \in {}^\bullet v'$. We obtain: $d^*(e) = w(e) + min_{f \in v^\bullet} min\{d_\sigma(f) \mid \sigma$ is a fair execution that starts in m_f $\}$. From this it follows that $d^*(e) = w(e) + min_{f \in v^\bullet} d^*(f)$.

Lemma 1.2 *Let v be a node such that $l(v)$ = AND and $v^\bullet > 1$. Let $\{e\} = {}^\bullet v$. We have $d^*(e) = w(e) + max\{d^*(f) \mid f \in v^\bullet\}$.*

Our proof relies on some notions we would like to fix here:

- From Lemma 3, it follows that if $\sigma^* = argmin_\sigma\{d_\sigma(e) \mid \sigma$ is a fair execution that starts in $m_e\}$, then σ^* is a loop-free execution.
- For a loop-free execution σ, and an unbounded number of resources, the duration of the execution is equal to the longest path in Γ_σ - the *critical path* [19].

Proof:

I. We prove that $d^*(e) \geq w(e) + max_{f \in v^\bullet} d^*(f)$.

From the definition of $d_\sigma(e)$ we have: $d_\sigma(e) = w(e) + max\{d_\sigma(f) \mid f \in v^\bullet\}$. Also by definition, we have: $d^*(e) = min\{d_\sigma(e) \mid \sigma$ is a fair execution that starts in $m_e\}$.

Therefore, $d^*(e) = min\{w(e) + max\{d_\sigma(f) \mid f \in v^\bullet\} \mid \sigma$ is a fair execution that starts in m_e $\}$.

$d^*(e) = w(e) + min\{max_{f\in v^\bullet} d_\sigma(f) \mid \sigma$ is a fair execution that starts in $m_e\}$.

We use the max-min inequality [4] which says that, for any function $f : Z \times W \to \mathbb{R}$ it holds that:

$$min_{w\in W} max_{z\in Z} f(z, w) \geq max_{z\in Z} min_{w\in W} f(z, w) \tag{1}$$

We obtain that:

$min\{max_{f\in v^\bullet} d_\sigma(f) \mid \sigma$ is a fair execution that starts in $m_e\} \geq max_{f\in v^\bullet} min\{d_\sigma(f) \mid \sigma$ is a fair execution that starts in $m_e\}$. Therefore,

$$d^*(e) \geq w(e) + max_{f\in v^\bullet} min\{d_\sigma(f) \mid \sigma \text{ is a fair execution that starts in } m_e\} \tag{2}$$

Note that since we first take minimum in the right hand side of the inequality, in $min\{d_\sigma(f) \mid \sigma$ is a fair execution that starts in $m_e\}$, due to Lemma 3 we can restrict to loop-free executions (some execution that minimizes $d_\sigma(f)$ is loop-free).

Note that $\forall\sigma$ starting in m_e, $\exists\sigma'$ starting in m_e such that $\sigma' = \langle m_e, \cdots, m_f, \cdots\rangle$, such that m_f is the edge enabling marking for $f \in v^\bullet$ and $\vec{\sigma}=\vec{\sigma'}$.

From Lemma 6, the edge enabling marking m_f can be reached without firing any transition in $Trans(v_f)$ where $f \in {}^\bullet v_f$.

$\sigma' = \underbrace{\langle m_e, \cdots\rangle}_{\sigma^{Pref}} \underbrace{\langle m_f, \cdots\rangle}_{\sigma^{Suf}}$.

We have that $d_{\sigma'}(f) = d_{\sigma^{Suf}}(f)$ (i).

We also have that $d_{\sigma'}(f) = d_\sigma(f)$ (ii). The reason for this is the fact that from $\vec{\sigma}=\vec{\sigma'}$ and from the acyclicity of σ and σ' it follows that $\Gamma_\sigma = \Gamma_{\sigma'}$. It is clear that $d_\sigma(f) = d_{\sigma'}(f)$ both represent the longest path starting from f in the same acyclic workflow graph.

From (i) and (ii) it follows that $d_{\sigma^{Suf}}(f) = d_\sigma(f)$. Therefore, $min\{d_\sigma(f) \mid \sigma$ is a fair execution that starts in $m_e\} = min\{d_\sigma(f) \mid \sigma$ is a fair execution that starts in $m_f\}$. Plugging this into Equation 2 we have that: $d^*(e) \geq w(e) + max_{f\in v^\bullet} min\{d_\sigma(f) \mid \sigma$ is a fair execution that starts in $m_f\}$, and therefore $d^*(e) \geq w(e) + max_{f\in v^\bullet} d^*(f)$.

II. We prove that $d^*(e) \leq w(e) + max_{f\in v^\bullet} d^*(f)$.

We construct an execution σ'' such that $max_{f\in v^\bullet}(d_{\sigma''}(f)) \leq max_{f\in v^\bullet} d^*(f)$.

Let $\sigma^*_f = argmin\{d_\sigma(f) \mid \sigma$ is a fair execution that starts in $m_f\}$.

Case 1: σ^*_f agree on all choices $\forall f \in v^\bullet$. Then σ'' is any of the σ^*_f.

Case 2: They don't agree on all choices and we need to construct σ''. We will iteratively modify σ^*_f, $\forall f \in v^\bullet$ without increasing $d_{\sigma^*_f}(f)$ until all of them will become identical to common σ''.

Let v' be a choice for which σ^*_f don't agree. Let $\Gamma_{\sigma^*_f}$ be a workflow graph restricted to elements contained in σ^*_f. Let also $\pi(\Gamma_{\sigma^*_f}, v')$ be the longest path in $\Gamma_{\sigma^*_f}$, starting from vertex v'. By taking $f_{min} = argmin_f\{\pi(\Gamma_{\sigma^*_f}, v')\}$, to modify each σ^*_f it is enough for each of them to take $choice(\sigma^*_f, v') = choice(\sigma^*_{f_{min}}, v')$. Due to the choice of f_{min}, such modification will not increase $d_{\sigma^*_f}(f)$ and will eventually converge to common σ'' for all f.

From the way we constructed σ'' we have:

$$w(e) + max_{f \in v^{\bullet}} d^*(f) \geq w(e) + max_{f \in v^{\bullet}} d_{\sigma''}(f) \tag{3}$$

From the definition, we have:

$$w(e) + max_{f \in v^{\bullet}} d_{\sigma''}(f) = d_{\sigma''}(e) \tag{4}$$

We have:

$$d_{\sigma''}(e) \geq d^*(e) \tag{5}$$

From Eq. 3, Eq. 4 and Eq. 5 we have $d^*(e) \leq w(e) + max_{f \in v^{\bullet}} d^*(f)$.

Proof of Lemma 2

Lemma 2 *Let e be an edge of a workflow graph Γ. Let σ be an execution such that $d_\sigma(e) = d^*(e)$. Let $S =< e_{i-1}, \cdots, e_{sink} >$ be the edges that get marked after e gets marked, in σ. Each sequence of calls of $Relax(e, v)$ that has the property that edges $e_{sink}, \cdots, e_{i-1}, e$ have been relaxed in order, after the sequence of calls to $Relax(e, v)$ we have $\delta(e) = d^*(e)$.*

We prove the lemma by induction on $k = |S|$.

Base case: k=0: $\delta(e_{sink}) = w(e_{sink})$, therefore $\delta(e_{sink}) = d^*(e_{sink})$.

Induction step: Suppose that after having relaxed $e_{sink}, \cdots, e_{k-2}, e_{k-1}$ in order, we have that $\delta(e_{k-1}) = d^*(e_{k-1})$. We will show that when we relax the edge e_k, given the definition of $d^*(e)$ and the induction hypothesis, it follows that $\delta(e_k) = d^*(e_k)$. We present the reasoning for one of the cases, as the justification for the remaining ones is similar.

Let v be a node, with $l(v) =$ XOR and $|v^{\bullet}| > 1$. Let ${}^{\bullet}v = \{e_k\}$ and $< e_{k-1}, \cdots, e_{sink} >$ are the edges that get marked after e gets marked, in σ. Assume without loss of generality that $e_{k-1} \in v^{\bullet}$.

From the induction hypothesis, we have $\delta(e_{k-1}) = d^*(e_{k-1})$. Since e_{k-1} is the edge that gets marked after e_k gets marked in σ for which $d_\sigma(e_k) = d^*(e_k)$, we have that $d^*(e_k) = w(e_k) + d^*(e_{k-1})$ (recall claim (ii) from the proof of Lemma 1).

We have demonstrated in Lemma 1 that after each call of $Relax(e, v)$ it holds that $\delta(e) \geq d^*(e)$, therefore for our case $\delta(e_k) \geq w(e_k) + d^*(e_{k-1})$ (i).

Also, after the relaxation of e_k we have that $\delta(e_k) = w(e_k) + min\{\delta(e') \mid e' \in v^{\bullet}\}$. Since $e_{k-1} \in v^{\bullet}$ it follows that $\delta(e_k) \leq w(e_k) + \delta(e_{k-1})$ and since $\delta(e_{k-1}) = d^*(e_{k-1})$, it follows $\delta(e_k) \leq w(e_k) + d^*(e_{k-1})$. (ii)

From (i) and (ii) it follows that after relaxing e_k, we have $\delta(e_k) = w(e_k) + d^*(e_{k-1})$, and therefore $\delta(e_k) = d^*(e_k)$.

Proof of Lemma 3

Lemma 3 *Some execution of Γ, with minimum duration, is loop-free.*

Let σ^* be an execution of Γ of minimum cost. Suppose σ^* is not loop-free. Then, $\sigma^* =< m_0, T_0, \cdots, m_i, T_i, \cdots, m_j, T_j, \cdots, m_f >$ such that $T_i = (E_1, v, E_2)$ and $T_j = (E'_1, v, E'_2)$ (there exists a node v such that a transition in $Trans(v)$ is executed more than once). Let τ_{σ^*} be the transition sequence corresponding to σ^*.

We construct σ' such that $\tau_{\sigma'}$ is a permutation of τ_{σ^*}. We will show that in σ', a marking is repeated which contradicts the claim that σ' (implicitly σ^*) is an execution of minimum cost of Γ.

We divide σ^* into three parts:

$\sigma^* = \underbrace{< m_0, \cdots, m_i >}_{\alpha} < T_i > \underbrace{< m_{i+1}, \cdots, m_f >}_{\beta}$

The constructed sequence σ' starts with α, followed by a permutation of the transitions contained in $\tau_{T_i\beta}$.

Let $\beta' =< m'_0, \cdots, m_k >$ be a maximal sequence such that:

1. $m'_0 = m_i$
2. $\vec{\beta'} \leq \vec{\beta}$

Claim: T is enabled in m_k implies $T \in Trans(v)$.

We prove indirectly that such a β' exists. Let $A = \vec{\beta} - \vec{\beta'} - [T_i]$. Suppose $\exists\ T'$, $T' \in Trans(v')$ enabled in m_k. We distinguish two cases:

Case 1: $\forall\ T'' \in A : T'' \notin Trans(v')$. In this case if we fire all the transitions in A we obtain m_f (due to the marking equation lemma). But m_f enables T', because of the assumption of case 1, therefore we reach a contradiction, namely that m_f is not the final marking.

Case 2: $\exists\ T'' \in A$ such that $T'' \in Trans(v')$. Then, T'' is enabled in m_k and therefore β' is not maximal and we reach a contradiction.

At this point, we have an intermediary prefix for σ', let's call it σ_{temp}, $\sigma_{temp} = \alpha\ \beta'\ T_i$. σ_{temp} does not include all the transitions in σ^*, more precisely we still have to add the transitions in $\vec{\beta} - \vec{\beta'}$.

Note that $T_j \in \vec{\beta} - \vec{\beta'} - [T_i]$. Let π denote the transition sequence obtained from τ_β after removing from it the transitions in $\tau_{\beta'}$ and T_i.

Since $m_0 \xrightarrow{\tau_\alpha} m_i \xrightarrow{\tau_{\beta'}} m_k \xrightarrow{T_i} m'_k \xrightarrow{\pi} m_f$, it means $\exists\ m_l \in \pi$ such that T_j is enabled in m_l (because T_j, $T_i \in Trans(v)$ and they were both blocked in β' and $T_j \in \vec{\pi}$. We repeat the same procedure for the sequence π as we did for the original sequence and we reach a marking m_p in which the only enabled transitions belong to $Trans(v)$ (recall that $T_j \in Trans(v)$).

It holds that, for $l(v) \in$ {XOR-split, AND-split, AND-join }, $m_k = m_p$ due to Lemma 6, stated by the authors in [8]. Since $\sigma' =< m_0, \cdots, m_k, \cdots, m_p, T_p, m_{p+1} \cdots, m_f >$ and $m_k = m_p$, one can construct a lower cost execution $\sigma'' =< m_0, \cdots, m_k, T_p, m_{p+1} \cdots,$ $m_f >$ by removing the execution in σ' that led to the repetition of the marking. This implies that σ^* can not be the execution of minimum cost and thus we reach a contradiction.

If $l(v)$ =XOR-join the results follows easily by noting that the edge $\{e\} = v^\bullet$ is marked twice and therefore the transition $T = (E_1, v, E_2)$ such that $e \in E_1$ is repeated, and thus the unique edge enabling marking of e is repeated.

Lemma 6 *If Γ is a sound workflow graph and v a node in Γ, such that $l(v) \in$ { XOR-split, AND-split, AND-join } then there exists a unique reachable marking m_v such that* *the only transitions enabled in m_v are the transitions in $Trans(v)$. Furthermore, the marking m_v can be reached from the initial marking and without firing any transitions in $Trans(v)$.*

An equivalent way of expressing the lemma above is by referring to edge enabling markings. For any edge e of Γ, there exists a unique edge enabling marking m_e.