# Model-Driven Design of Performance Requirements

Antonio García-Domínguez, Inmaculada Medina-Bulo
Dept. of Computer Languages and Systems
University of Cádiz
Cádiz, Spain
Email: {antonio.garciadominguez, inmaculada.medina}@uca.es

Mariano Marcos-Bárcena
Dept. of Mechanical Engineering and Industrial Design
University of Cádiz
Cádiz, Spain
Email: mariano.marcos@uca.es

***Abstract*—**

**Obtaining the expected performance of a workflow is much simpler if the requirements for each of its tasks are well defined. However, most of the time, not all tasks have well-defined requirements, and these must be derived by hand. This can be an error-prone and time consuming process for complex workflows. In this work, we present an algorithm which can derive a time limit for each task in a workflow, using the available task and workflow expectations. The algorithm assigns the minimum time required by each task and distributes the slack according to the weights set by the user, while checking that the task and workflow expectations are consistent with each other. The algorithm avoids having to evaluate every path in the workflow by building its results incrementally over each edge. We have implemented the algorithm in a model handling language and tested it against a naive exhaustive algorithm which evaluates all paths. Our incremental algorithm reports equivalent results in much less time than the exhaustive algorithm.**

***Index Terms*—performance engineering, performance analysis, service level agreement, UML activity diagrams, workflows.**

## I. Introduction

High-quality software is expected to work, and to work well. This means that, in addition to correctly implementing the expected functionality, it also needs to meet a variety of non-functional requirements, such as usability, security, or efficiency. However, performance aspects tend to be ignored until the functionality is done, when it is more expensive and difficult to revise the design of the system [1].

In this context, workflow-based languages have started to become commonplace in some contexts, such as in business process automation. Some examples include BPMN 2.0 [2], WS-BPEL 2.0 [3] or jPDL [4]. These languages are more powerful than traditional flowcharts and UML activity diagrams, and can be executed by process engines. Tasks can be reused and recombined over several workflows.

Workflows are not without their problems, however. A client might impose a certain Service Level Agreement (SLA) on a critical workflow. Some tasks in the workflow might call external systems from third-party information providers. We need to ensure we can meet the SLA from our client, while we do not ask for too much or too little performance from the external providers. If the SLA requested by the client was unfeasible, we should be able to report this to the user as soon as possible.

Ideally, we would have collected extensive performance metrics on the workflow and its tasks and used them to base our decisions. However, if the workflow and its tasks had not been developed yet, we would not have historical data. We would have to rely on educated guesses on the performance required by the tasks in order to obtain the desired performance from the workflow. These guesses would have to be consistent with the structure of the workflow, and would have to be kept up to date as the workflow was revised. Doing this by hand would be a time-consuming and error-prone process. Nevertheless, having concrete performance requirements for each task would make it easier to agree on SLAs and detect performance degradations during monitoring or periodic testing.

In this work, we define two algorithms for the same goal: inferring time limits from the tasks (vertexes) in a workflow (graph), using a global time limit set by the user and some local annotations. The annotations are placed by the user according to an expected peak or average workload. The first algorithm is simpler but less efficient, trying all paths in the graph. It is an extended and generalised version of one of the algorithms presented in [5]. The second algorithm obtains equivalent results in less time by producing its results incrementally, discarding uninteresting paths. The algorithms have been implemented as Eclipse plug-ins and are freely available from the SODM+T project website [6].

The structure of this work is as follows: in Section II, we define some key terms and specify the inputs and outputs of both algorithms. In Section III we describe the running example with which we illustrate both algorithms. Section IV is dedicated to the simpler, exhaustive algorithm, while Section V describes the incremental algorithm. The algorithms are analysed and compared in Section VI. Finally, Section VII discusses related works and Section VIII offers some conclusions and future lines of work.

## II. Design requirements and concepts

Before describing the exhaustive and the incremental algorithms, we need to describe their inputs and outputs and define some useful terms.

Both algorithms take the following inputs:

- A directed acyclic graph (DAG) $G = (V, E)$, where $V$ is the set of all *vertexes* and $E \subseteq V \times V$ contains all *edges*. The graph must have one vertex with no incoming edges

(*source*, $s$), and one or more vertexes with no outgoing edges (*sinks*). All vertexes must be reachable from $s$, i.e. the graph must be weakly connected. If there is an edge from $A$ to $B$, it means that $B$ *runs after* $A$.
For a vertex $v$, let $o(v)$ be its outgoing edges and $i(v)$ its incoming edges. For an edge $e$, let $s(e)$ be its source vertex and $g(e)$ be its target vertex.

- $L > 0$, the global time limit which must be met by all paths from the source $s$ to a sink of $G$.
- $c : V \to C(L)$, a function which maps each vertex to an element of $C(L) = \{(m, w) \mid 0 \leq m \leq L, w \geq 0\}$, the set of all valid constraints when the global time limit is equal to $L$. For a vertex $v$, its constraint $c(v) = (m(v), w(v)) \in C(L)$ is an ordered pair where $m(v)$ is the minimum time limit which should be assigned to the vertex $v$, and $w(v)$ is the weight which should be used to distribute the *slack* (defined later).
  Using $c(v)$, we can derive a constraint for each path $p$ in the graph, $c(p) = (m(p), w(p)) \in C(L)$, with $m(p) = \sum_{v \in p} m(v)$ and $w(p) = \sum_{v \in p} w(v)$. Therefore, the constraint of $p_a + p_b$, the concatenation of the paths $p_a$ and $p_b$, is equal to $c(p_a + p_b) = (m(p_a) + m(p_b), w(p_a) + w(p_b))$.

If successful, the algorithms produce a function $t : V \to [0, L]$, which assigns a time limit to each vertex. Let $P_S(v)$ represent the set of all possible paths in $G$ from the vertex $v$ to a sink. $t$ must meet the following constraints:

- $\forall v \in V \; t(v) \geq m(v)$: the time limit assigned to each vertex must be greater or equal than the minimum value set by the user.
- $\forall p \in P_S(s) \; \sum_{v \in p} t(v) \leq L$: the sum of the time limits over every path must meet the global time limit.

Finally, the algorithms must validate that the global and local constraints are consistent with each other. To do this, we need to define several concepts. If a vertex $v$ *receives* a certain amount of time, $0 \leq r(v) \leq L$, each path $p \in P_S(v)$ (starting at $v$ and ending at a sink) is also said to *receive* $r(p) = r(v)$ seconds. This is the amount of time that must be distributed over the vertexes in $p$, taking into account their individual constraints. In general, $r(v)$ is not known *a priori* except for the source $s$: by definition, the source of the graph receives all the available time, $L$, so $r(s) = L$.

If the global and local constraints are consistent, then $r(p) \geq m(p)$ for every path: the time received can fulfil the minimum time constraints of each of its vertexes. The amount in which $r(p)$ exceeds $m(p)$, $s(p) = r(p) - m(p)$, is known as the *slack* of the path $p$. Generally, the slack is proportionally distributed over the vertexes in $p$ according to the weight of each vertex: the *slack per unit of weight* assigned to each vertex is $S_w(p) = s(p)/w(p)$. When $w(p) = 0$, we will assume that $S_w(p) = 0$: since all vertexes in $p$ have a zero weight, no slack can be distributed.

The algorithms must ensure that $s(p) \geq 0$ for every path $p$, so all paths meet the minimum time constraints, and that $w(p) > 0 \Rightarrow s(p) > 0$, so every path $p$ with a non-zero weight has some slack to distribute. In case these conditions are not met, the situation should be reported to the user and execution should be aborted.

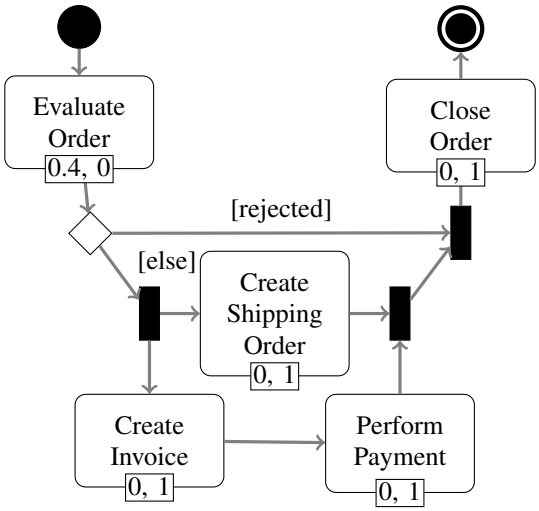



Fig. 1: Running example: original UML activity diagram

## III. Running example

In order to illustrate the algorithms, we will use the simple workflow in Figure 1 as a running example throughout this work. The algorithms are tested with more complex models in Section VI. It is a simple UML activity diagram describing how to process a specific order. Execution starts from the initial node, and proceeds as follows:

1) The order is evaluated and either accepted or rejected.
2) If rejected, close the order: we are done.
3) Otherwise, fork into two execution branches:
   a) Create the shipping order and send it to the shipping partner.
   b) Create the invoice, send it to the customer and receive the payment.
4) Once these two branches are done, close the order.

The start, finish, decision, fork, join and merge nodes in Figure 1 cannot have time limits: they represent control structures, rather than actual tasks. These elements are implicitly annotated with the $(0, 0)$ constraint: the algorithms effectively ignore them. Therefore, we can simplify the graph to that in Figure 2, from which the same time limits are inferred. Each node is annotated with its minimum time limit $m$, its weight for distributing the slack $w$, and the (unknown for now) time limit $t$.

It is interesting to note that the running example has two kinds of constraints on its vertexes: while "Evaluate Order" has a non-zero minimum time limit and zero weight, the rest have no minimum time limit and a non-zero weight. Therefore, the time limit for "Evaluate Order" is fixed by the user, while the other time limits depend entirely on how much slack is left. Since all the vertexes with non-zero weights have the same weight, the slack will be uniformly distributed among them in absence of other constraints.

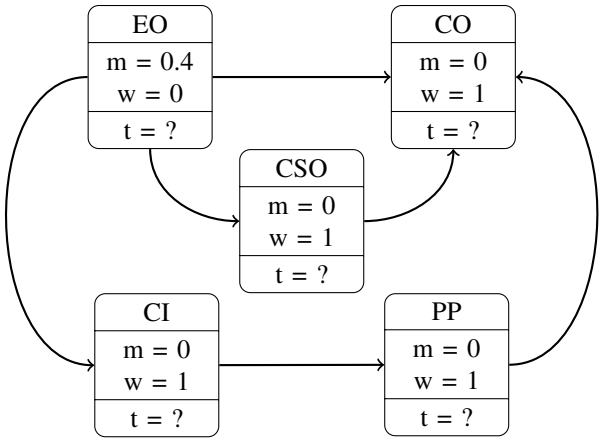



Fig. 2: Running example: simplified graph

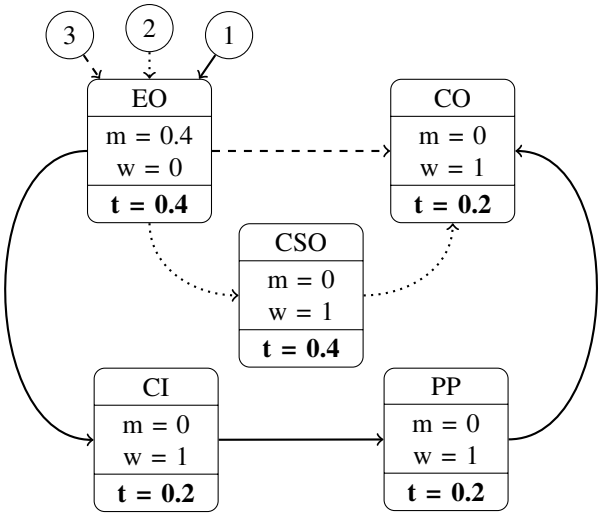



Fig. 3: Results of the exhaustive algorithm on the running example, with $L = 1$ second

In addition, constraints may have both a non-zero minimum time limit and a non-zero weight. The inferred time limit would have a fixed part from the minimum time limit, and a variable part from the slack. Such constraints are uncommon for nodes, but they appear naturally in most paths. For instance, the constraint for the path $\langle \text{EO}, \text{CI}, \text{PP}, \text{CO} \rangle$ is $(0.4, 0) + (0, 1) + (0, 1) + (0, 1) = (0.4, 3)$.

## IV. Exhaustive algorithm

Before describing the incremental algorithm, we will present a simpler algorithm which considers all paths from the source to each of the sinks. This algorithm will be used as test oracle and performance baseline for the incremental algorithm. It is a generalisation of the algorithm presented in [5], taking into account the requirements in Section II.

It follows these steps:

1) Compute all the paths in $P_S(s)$. These are all the paths from the source of the graph, $s$, to each sink.
2) Sort each path $p$ in $P_S(s)$ in ascending order of $S_w(p)$, so the paths with less slack per unit of weight are first. From Section II, we know that $\forall p \in P_S(s)\ r(p) = L$.
3) Visit each path $p$ in $P_S(s)$:
   a) If $s(p) = 0$, the minimum time limits are taking up all the available time. If $w(p) > 0$, there will be vertexes which will not receive any slack as they should. In that case, the user is notified and execution is aborted.
   b) If $s(p) < 0$, the global and local constraints are not consistent. The sum of the minimum time limits in $p$ exceeds the global time limit. The user is notified and execution is aborted.
   c) Otherwise, $s(p) > 0$. Split the vertexes in $p$ into two disjoint subsets: $U$ has the *unrestricted* vertexes that do not have time limits yet, and $R$ has the *restricted* vertexes that do.
   The slack to be distributed among the unrestricted vertexes, $S$, is equal to $L - (t(R) + m(U))$, where $t(R) = \sum_{v \in R} t(v)$ and $m(U) = \sum_{v \in U} m(v)$.
   The slack per unit of weight is $S_w = S/w(U)$ if $w(U) > 0$. Otherwise, $S_w = 0$.
   Each vertex $v \in U$ is assigned the time limit:

$$t(v) = m(v) + S_w w(v) \tag{1}$$

This algorithm basically tries all paths, sorting them so each vertex will be visited first in the path from which its strictest time limit can be inferred. Once it has a time limit, it will not be changed again: instead, its time limit will be used to calculate the time limits for the vertexes in the following (and less strict) paths.

As an example, we will invoke the algorithm on the running example in Figure 2 with the global limit $L$ set to 1 second, producing the results in Figure 3. We have labelled with numbers the three distinct paths from the source (the vertex with no incoming edges) to the sink (the vertex with no outgoing edges), ordering them from most to least restrictive.

For the first path (drawn with a solid line), $\langle \text{EO}, \text{CI}, \text{PP}, \text{CO} \rangle$, we assign 0.4 seconds to EO and distribute the 0.6 seconds of slack among the other three. For the second path (drawn with a dotted line), $\langle \text{EO}, \text{CSO}, \text{CO} \rangle$, we assign the 0.4 seconds that have not been assigned yet to CSO. Finally, we do not have to do anything in the third and least restrictive path (drawn with a dashed line), as all vertexes have been annotated.

We can conclude that the algorithm is easy to understand and implement. However, it has an important flaw: it needs to visit all paths in $P_S(s)$, which grows exponentially as forks are added to the graph. This means that the algorithm is unusable except for simple graphs.

## V. Incremental algorithm

In the previous section, we described a simple exhaustive algorithm for computing the time limits. The algorithm was easy to understand, but its high computational cost made it impractical for general use. In this section, we will present

an algorithm which works around this issue by discarding uninteresting paths. The algorithm operates incrementally edge by edge, rather than iterating over every path from the source to a sink as the previous algorithm did. We will first offer a general outline, and then focus on some key details.

Let us recall from Section II that $P_S(v)$ denotes the set of all paths which start at $v$ and end at a sink, $o(v)$ are the outgoing edges of $v$, $g(e)$ is the target vertex of the edge $e$, $r(p)$ is the time received by the path $p$, $m(p)$ is the sum of the minimum time limits over the vertexes in $p$, $s(p) = r(p) - m(p)$ is the slack of the path $p$, $w(p)$ is the sum of the weights over $p$, and $S_w(p)$ is the slack per unit of weight of $p$. $S_w(p) = s(p)/w(p)$ when $w(p) > 0$, and $S_w(p) = 0$ when $w(p) = 0$.

We will define the algorithm as a recursive function which receives the current vertex $v$ and the available time, $T$. Initially, $v$ is equal to $s$, the source of the graph, and $T$ is set to $r(s) = L$, the global time limit. The algorithm follows these steps:

1) Select two paths among $p \in P_S(v)$:
   - $p_{ms}(v)$, the path among all $p \in P_S(v)$ with the minimum $S_w(p)$ when $T$ seconds are available. In case of a tie, pick the path with the highest $w(p)$.
   - $p_{Mm}(v)$, the path among all $p \in P_S(v)$ with the maximum value of $m(p)$.
2) If $s(p_{Mm}(v)) < 0$, the minimum time limits cannot be satisfied: abort.
3) If $s(p_{ms}(v)) = 0$ and $w(p_{ms}(v)) > 0$, no slack can be distributed in a path with a non-zero weight: abort.
4) Set the time limit of $v$, $t(v)$, to $m(v)+S_w(p_{ms}(v))w(v)$. The remaining time will be $T_R = T - t(v)$ seconds. Mark $v$ as visited.
5) Sort each edge $e \in o(v)$ in ascending order of $S_w(p_{ms}(g(e)))$ with $r(g(e)) = T_R$, the minimum slack per unit of weight when $T_R$ seconds are available of all the paths that start at the target of $e$.
6) Visit each edge in $o(v)$:
   a) If the target of $e$ has been visited before, check if the time which was sent to it, $T'_R$, is strictly less than $T_R$, the time which would have been sent through $e$. In that case, try to reuse the surplus $T_R - T'_R$ seconds on the source of $e$ and its ancestors, and send $T'_R$ seconds through $e$. We will detail the procedure later.
   b) If the target of $e$ has not been visited before, invoke this algorithm recursively, setting $v$ to the target of $e$ and $T$ to $T_R$.

At a first glance, the algorithm does not present an obvious performance improvement. Computing $p_{ms}(v)$ and $p_{Mm}(v)$ requires selecting elements from $P_S(v)$: $P_S(s)$ in particular will contain all the paths from the source of the graph to a sink, as in the exhaustive algorithm.

However, a closer look shows that the computation of $p_{ms}(v)$ and $p_{Mm}(v)$ can be considerably optimised:

- We do not need the sequence of vertexes that form $p_{ms}(v)$ and $p_{Mm}(v)$. For $p_{ms}(v)$, we only need its constraint, $c(p_{ms}(v))$, and for $p_{Mm}(v)$, we only need $m(p_{Mm}(v))$, its minimum time limit.
- $m(p_{Mm}(v))$ can be easily computed incrementally for each vertex, using the formula

$$m(p_{Mm}(v)) = m(v) + \max\{m(p_{Mm}(g(e))) \,|\, e \in o(v)\} \quad (2)$$

  We only have to traverse the graph from the sinks to the source: starting by the sinks, each vertex $v$ will be annotated with $m(p_{Mm}(v))$ after its direct children.
- We do not need to consider every path in $P_S(v)$ to compute $c(p_{ms}(v))$. Let $p_a, p_b \in P_S(v)$ be two paths starting at the same vertex. If $S_w(p_a) \geq S_w(p_b)$ (i.e. $p_a$ is *always less strict* than $p_b$) for any valid value of $r(v)$, we can safely ignore $p_a$ when computing $p_{ms}(v)$. We will denote the subset of $P_S(v)$ excluding these redundant paths as $P'_S(v)$. We will show below that $P'_S(v)$ can be computed incrementally in the same preprocessing step used to compute $m(p_{Mm}(v))$ for each vertex.

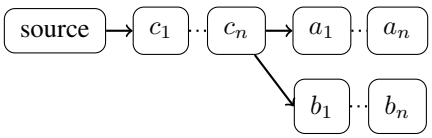



Fig. 4: Comparison of two paths $p_a$ and $p_b$, where $p_a = \langle a_1, \ldots, a_n \rangle$, $p_b = \langle b_1, \ldots, b_n \rangle$, and $p_c = \langle \text{source}, c_1, \ldots, c_n \rangle$ is the common prefix for their superpaths

By combining these optimisations, we can achieve better performance than the naive exhaustive algorithm. Over the following subsections, we will detail the optimisations and how surplus time is reused. Finally, we will show how the resulting algorithm is applied to the running example in Section III.

### A. Reducing exponential growth of the number of paths

$p_a$ is said to be *always less or just as strict* than $p_b$ ($c(p_a) \preceq_{s(L)} c(p_b)$) if $S_w(p_c + p_a) \geq S_w(p_c + p_b)$, for any $p_c$ and $T$ which ensure that:

- The time received by $p_a$ and $p_b$ is not negative and less than or equal to the global time limit: $0 \leq T \leq L$.
- The constraints of the paths are valid: $c(p_c + p_a) \in C(L)$ and $c(p_c + p_b) \in C(L)$.
- The available slack can be distributed in both paths, as they have non-zero weights: $w(p_c + p_a) > 0$ and $w(p_c + p_b) > 0$.
- The slack is not negative: $s(p_c + p_a) \geq 0$ and $s(p_c + p_b) \geq 0$.

Consider Figure 4. If $c(p_a) \preceq_{s(L)} c(p_b)$, then we can exclude $\langle c_n \rangle + p_a$ from $P_S(c_n)$, $\langle c_{n-1}, c_n \rangle + p_a$ from $P_S(c_{n-1})$, and so on. We only need to consider $p_a$ when we reach $a_1$.

Let $c(p_a) = (a, b)$, $c(p_b) = (c, d)$, and $c(p_c) = (x, y)$, where $p_c$ is the common prefix concatenated with $p_a$ and $p_b$

as in Figure 4. We define $\preceq_{s(L)}$ formally as follows:

$$\begin{aligned}(a,b) \preceq_{s(L)} (c,d) \equiv \\ \forall t \in [0,L]\ \forall x \in [0,L]\ \forall y \geq 0 \\ a+x \leq t \wedge c+x \leq t \wedge \\ b+y > 0 \wedge d+y > 0 \Rightarrow \\ \frac{t-(a+x)}{b+y} \geq \frac{t-(c+x)}{d+y}\end{aligned} \tag{3}$$

We can define a simplified version of this predicate:

$$a \leq c \wedge (b \leq d \vee a < L \wedge b > d \wedge (b-d)L \leq bc - ad) \tag{4}$$

It can be proved that this defines a partial order (a reflexive, antisymmetric, and transitive binary relation) on $C(L)$. The proof is omitted for the sake of brevity.

The simpler version in (4) has also the advantage that it defines a total order on constraints of the form $(L, x)$, where $L$ is the global limit and $x$ is a non-negative real number. The original version in (3) would consider all $(L, x)$ to be equal to each other.

Again, it is important to remark that $\preceq_{s(L)}$ is just a partial order: some pairs of constraints will be incomparable, forcing the algorithm to continue considering both paths. Therefore, the number of paths which are excluded from $P_S(v)$ to produce $P'_S(v)$ is key to the efficiency of the optimised algorithm.

### B. Computing the interesting paths incrementally

In the previous section, we showed how paths of the form $p_c + p_a$ could be ignored over those of the form $p_c + p_b$ if $c(p_a) \preceq_{s(L)} c(p_b)$. However, a naive approach which computed all paths in $P_S(v)$ and then excluded some with $\preceq_{s(L)}$ to compute $P'_S(v)$ would be still inefficient. In this section we will show how to integrate $\preceq_{s(L)}$ with the incremental computation of $P'_S(v)$, so the uninteresting paths are discarded as soon as possible and do not impact the performance of the algorithm.

We will preprocess the graph and annotate each edge $e$ with $c(e) = P'_S(g(e))$, the maximal constraints of the paths in $P_S(g(e))$ according to $\preceq_{s(L)}$. In other words, the constraints of the paths which are not always less or just as strict than any other path starting at the target of $e$ and ending at a sink.

Let $\max_{\preceq_{s(L)}} S$ be the set of the maximal elements of the set $S$ according to the partial order relation $\preceq_{s(L)}$. We define $c(e)$ as

$$\begin{aligned}c(e) = \max_{\preceq_{s(L)}} \{(m(g(e)) + M, w(g(e)) + W) \\ \mid e' \in o(g(e)), (M, W) \in c(e')\}\end{aligned} \tag{5}$$

As a special case, we will consider that $c(e) = \{c(g(e))\}$ when $o(g(e)) = \varnothing$, that is, when the target of $e$ has no outgoing edges (i.e. is a sink).

Equation (5) suggests that $c(e)$ can be computed incrementally by starting at the incoming edges of the sinks and going backwards through the graph, ensuring that the incoming edges of a vertex are only annotated when all its outgoing edges have been annotated. We are done when the outgoing edges of the source of the graph are annotated.

By using $c(e)$, we can easily retrieve the value of $P_S(v)$ for a vertex $v$ through any of its incoming edges. We will handle the source of the graph, $s$, as if it had a virtual incoming edge from a virtual vertex, so we can retrieve $P_S(s)$ in the same way.

Preprocessing the running example in Figure 2 would produce the annotations shown in Figure 5. In this simple case, each edge only has one constraint, as all constraints considered have been comparable. As an example, the edge from EO to CI is annotated with $(0, 3)$, which is the constraint of the only path which starts from CI and ends at a sink, $\langle \text{CI}, \text{PP}, \text{CO} \rangle$. The virtual edge to the source of the graph is drawn with a dashed line: $P'_S(s)$ is equal to $\{(0.4, 3)\}$. $P_S(s)$ would have consisted of $(0.4, 1)$, $(0.4, 2)$ and $(0.4, 3)$.

### C. Reusing surplus time

Reusing some surplus time $T$ back in the graph, starting from a vertex $v$, consists of these steps:

1) Create an empty collection of vertexes, $R$.
2) While $v$ has one outgoing edge:
   a) If $w(v) > 0$, add it to $R$.
   b) If $v$ has one incoming edge $e$, move to the source of $e$ and continue. Otherwise, exit the loop.
3) Increase the time limit of each $v \in R$ by $Tw(v)/w(R)$.

When reusing surplus time, it is important not to modify any vertex which could have been visited through a more restrictive path. This is why we only collect vertexes with one outgoing edge and why we only continue backing up if the vertex has one incoming edge.

### D. Example of invocation

Running the optimised algorithm on the running example in Figure 2 with $L = 1$ second yields different results depending on whether the surplus time is reused as described in Section V-C or not.

First, the preprocessing step described in Section V-B is run, producing the annotations shown in Figure 5, from which $P'_S(v)$ and $m(p_{Mm}(v))$ can be easily queried for each vertex $v$.

The strictest path starting at the source, EO, has the constraint $(0.4, 3)$, so the slack per unit of weight is $(1 - 0.4)/3 = 0.2$, $t(\text{EO}) = 0.4 + 0.2 \cdot 0 = 0.4$ and the remaining $1 - 0.4 = 0.6$ seconds are sent through its three outgoing edges. The edges are sorted using the minimum slack per unit of weight among their strongly strictest paths: first the edge to CI with $(0.6 - 0)/3 = 0.2$ seconds, then the edge to CSO with $(0.6 - 0)/2 = 0.3$ seconds and finally the edge to CO with $(0.6 - 0)/1 = 0.6$ seconds.

Therefore, CI is visited first. $t(\text{CI}) = 0 + (0.6 - 0)/3 = 0.2$, and the remaining 0.4 seconds are sent to PP, which takes another 0.2 seconds and sends the final 0.2 seconds to CO, a sink.

Back at EO, the edge to CSO is now traversed. CSO takes $0 + (0.6 - 0)/2 = 0.3$ seconds and tries to send 0.3 seconds

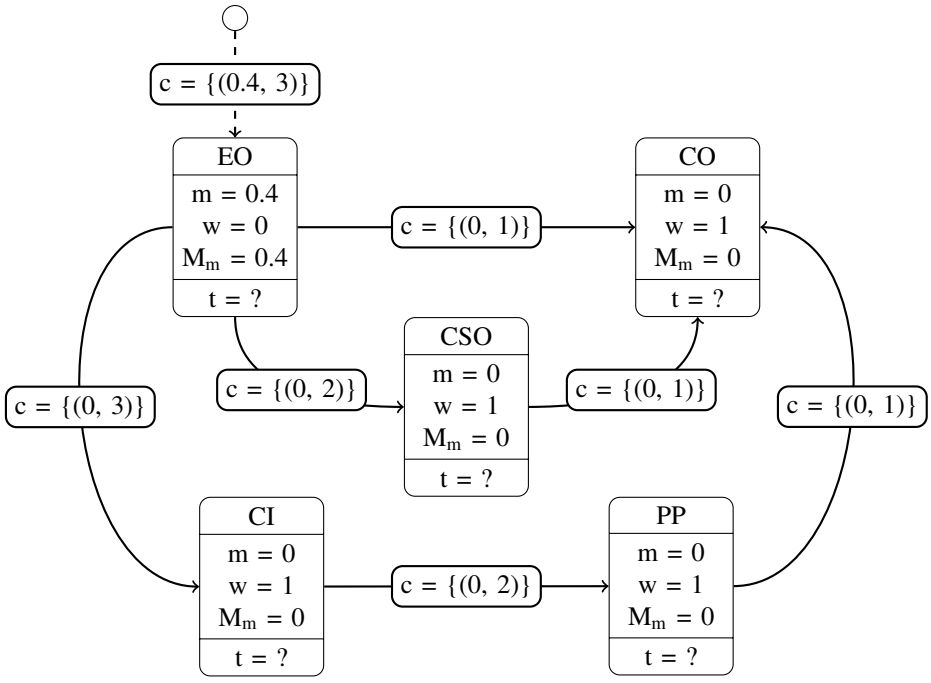



Fig. 5: Precomputed $M_m(v) = m(p_{Mm}(v))$ and $c(e) = P'_S(g(e))$ for the running example

to CO. However, CO already received 0.2 seconds: there is a surplus of 0.1 seconds. If we reuse those 0.1 seconds on CSO, we will obtain the same results than the exhaustive algorithm, as shown in Figure 6.

We are back at EO, and we try to send 0.6 seconds into EO, which already received 0.2 seconds. However, EO has more than one outgoing edge, so we do not attempt to reuse the 0.4 extra seconds: it would produce incorrect results in this case. We are done: all edges have been traversed.

## VI. Evaluation

In this section we will evaluate both algorithms and compare their performance after implementing them as part of a set of Eclipse plugins which work on simplified UML activity diagrams. These plugins are freely available at the SODM+T website [6]. The editors and the algorithms are implemented using several languages from the Epsilon project [7].

### A. Limitations

Both algorithms require that the graph has no cycles. This seems like an important restriction, as loops are common in most workflows, but some workflow languages can model iteration without cycles. This is the case of the *Loop Activities* in BPMN 2.0 [2]. A Loop Activity can contain a Sub-Process with the contents of each iteration. From the viewpoint of our algorithms, if we considered that the loop required 100 iterations, we could multiply the weight of the Loop Activity by 100, infer the time limit $T$ for the Loop Activity and then infer the time limits of its contents, using $T/100$ as the global time limit.

All vertexes need to be annotated with a minimum time limit and a weight. It could be argued that we just shifted the problem of defining the time limit of each task to defining its constraint. However, in the absence of other information, a tool can just annotate each vertex with $(0, 1)$, a simple constraint which requires no minimum time limit and distributes the available time uniformly.

The algorithms are not restricted to a particular workflow representation, as long as execution starts from a specific vertex, continues sequentially and forks into several branches which may or may not run concurrently until they reach some vertex. As a simple example, we started with UML activity diagrams. At this stage, our main priority was to obtain correct results and acceptable performance, so the annotations used do not follow any of the standard UML profiles for performance requirements. It could be interesting to adapt these annotations to the OMG SPTP [8] or the QFTP [9].

The algorithms do not consider the fact that the same task may be used in several workflows, or several times in the same workflow. A simple solution to this problem would be taking the strictest time limit inferred among all its occurrences. Integrating the "same task" constraint into the algorithm would be useful, but it might considerably increase its cost. We will consider it in the future.

Finally, the algorithms do not generate code: they are focused on helping design the performance requirements for each task in a workflow. From these requirements, other elements can derive test cases for the concrete software artifact represented by the workflow task.

### B. Performance analysis

Having discussed the limitations of the algorithms, we will now compare their results and their performance. As it is difficult to consider every acceptable graph, we applied the

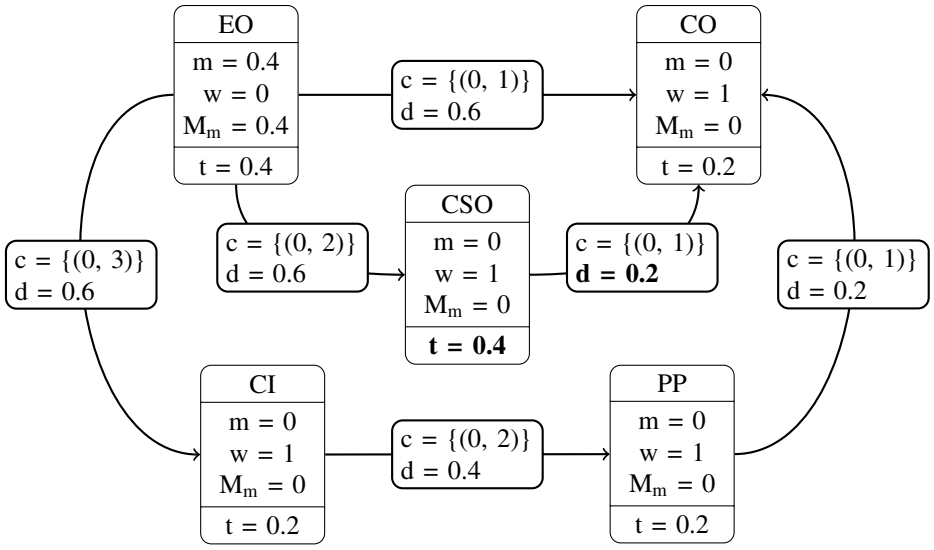



Fig. 6: Results of the incremental algorithm on the running example with $L = 1$ second

algorithms to three specific kinds of directed acyclic graphs: sequences, fork-join sequences and "dense" graphs. Figure 7 shows some examples of each kind. Among these, "dense" graphs are the most expensive: starting from a sequence of nodes, edges are added from each node to all the following nodes (excluding itself). These are the densest directed acyclic graphs we can build when limited to one edge per pair of nodes, and therefore the most expensive class for our algorithms.

In order to verify that both algorithms produced acceptable results, both algorithms were unit tested with representative examples from each kind of graph, checking that the inferred time limits for each vertex matched our expectations. In addition, for the incremental algorithm, we also verified that the $P'_S(v)$ derived in the preprocessing step were correct. We checked that the algorithms rejected some clearly inconsistent graphs.

For comparing their performance, we developed an Eclipse view that generated models of each kind automatically and plotted the times required by each algorithm. Figure 8 shows the results produced by the Eclipse view.

We can conclude that, except for small sequences, the incremental algorithm is a clear improvement over the exhaustive algorithm: it scales much better as the number of paths increase in the fork-join sequences and the dense graphs. The exhaustive algorithm is slower for sequences as it is forced to list all vertexes in the only path in the graph and scan that list several times, while the incremental algorithm only has to scan the sequence twice: backwards for the preprocessing step, and forwards for inferring the time limits.

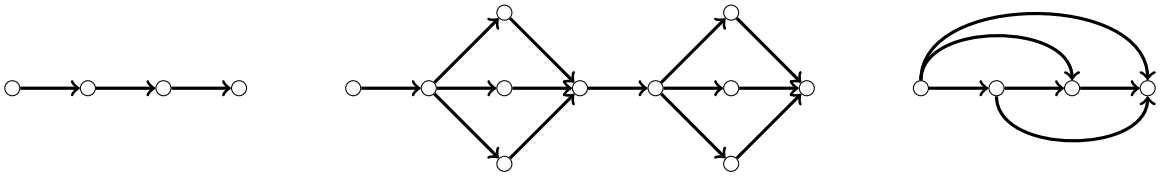
The best case for the incremental algorithm is when $P'_S(v)$ is left with only one element from $P_S(v)$, for every vertex $v$: all paths starting at each vertex were comparable. Conversely, the worst case is when $P'_S(v) = P_S(v)$ and $\preceq_{s(L)}$ has not excluded any paths. Randomly annotating 50% of the vertexes did not produce any example for the worst case. Further analysis looking for the exact conditions required for this worst case is required.

During performance comparison, we tested that both algorithms always produced equivalent results. The algorithms did not produce the exact same results bit by bit, however, due to floating point error propagation. Error propagation is more noticeable in the incremental algorithm, as the time limit of a vertex depends on the time limits of all its ancestors, rather than on the constraint of the strictest path from the source to a sink which contains it. This could be remedied by using integers (and a smaller unit of time) or exact rationals (as those in Lisp).

## VII. Related work

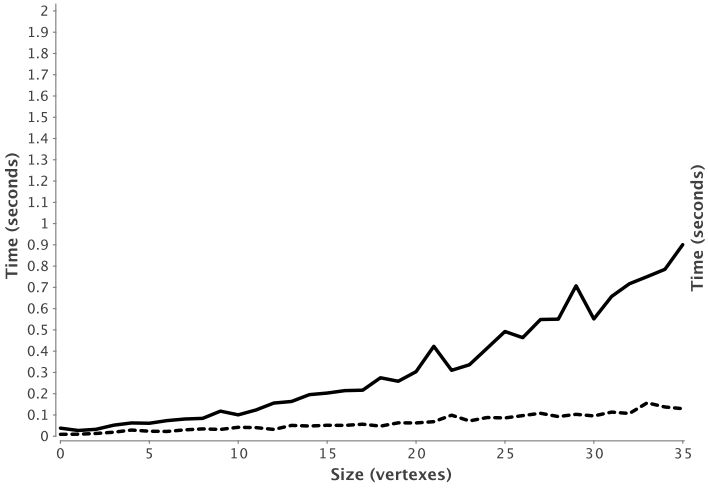
In order to obtain the desired performance level from a software system, several approaches are available, depending on whether we are at an early or a late stage of development. At an early stage, *performance engineering* attempts to predict the performance of the target system from a model, identifying problems as soon as possible [1]. Later on, when the system has been developed, *performance testing* uses representative workloads to exercise the system and collect valuable information to check its actual performance, to identify ways to improve it or to select the best environment for the system [10]. Some authors have proposed overloading the term "performance engineering" to point to both model- and measuring-based approaches [11].

Performance engineering usually requires building a model with information on each part of the system, from which the expected global performance is derived. The most common formalisms are layered queuing networks [12], stochastic Petri networks [13] and process algebra specifications [14]. Some works operate on workflows as we do: for instance, Silver et al. [15] annotated each task in a workflow with probability distributions for their running times, and collected data from 100 simulation runs to estimate the probability distribution

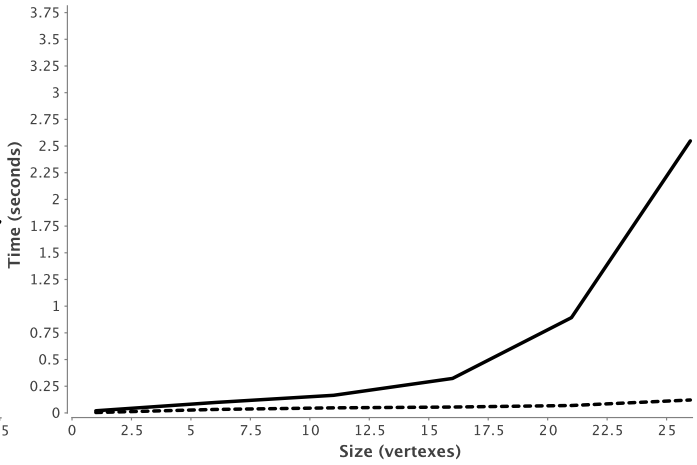


(a) Sequence: 4 vertexes (b) Fork-join sequence: 2 forks with 3 branches in each (c) Dense graph: 4 vertexes

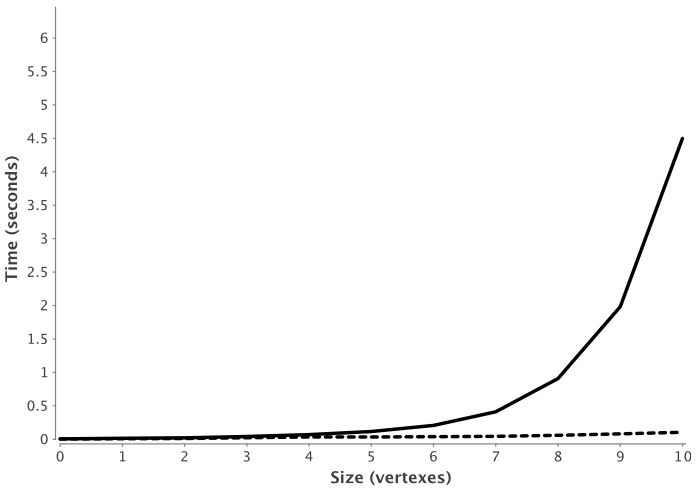
Fig. 7: Kinds of graphs used in the performance comparisons




(a) Sequence: from 0 to 35 vertexes




(b) Fork-join sequence: from 0 to 5 forks with 3 branches each




(c) Dense: from 0 to 10 vertexes

Fig. 8: Performance comparisons between the exhaustive algorithm (solid line) and the incremental algorithm (dashed line) for each model type: average time over 50 runs for each model size, global limit $L = 100$ seconds and annotating 50% of the vertexes with uniform random minimum times (between 0 and $L$) and weights (up to 10)

of the running time of the workflow. The SWR algorithm proposed by Cardoso et al. [16] computes workflow QoS by iteratively reducing its graph model to a single task. The SWR algorithm and its incremental nature is close to our work, but instead of inferring global information from local specifications, as it does, we infer local requirements from global specifications.

Unlike the previous works, our work does not attempt to infer the global performance of the system or measure it. We intend to assist analysts in specifying the performance requirements of the system. Most works in this aspect are dedicated to defining standard notations, such as the OMG SPTP [8] and QFTP [9] profiles, or on using model-driven approaches to generate the instrumentation code and test stubs [17]. Instead, we focus on helping the tester "fill in the blanks" using the partial information which is already known about the system: Service Level Agreements from business requirements on some workflows, historical data of legacy software, and so on.

## VIII. Summary and future work

Most of the time, developers lack concrete performance requirements on the components they use. Normally, only high-level performance requirements are available, due to the cost of maintaining the low-level performance requirements in sync as the structure of the system and the high-level requirements change.

Ideally, developers would write the known requirements of the system and let the tool fill in the blanks with requirements for each part. In this work, we have focused on the more manageable case of a workflow and its individual tasks. We continue our work in [5], which detailed how to infer the average number of concurrent requests per second of each task, and described a simple time limit inference algorithm which tried all paths in the graph.

In this work, we have presented two algorithms for inferring time limits for the tasks in a workflow: a generalised and extended version of the algorithm in [5], and a new incremental algorithm which does not need to always visit every path in the graph. To simplify the discussion, the running example was based on a small UML activity diagram and we used an *ad hoc* notation for the annotations. We have implemented these algorithms as Eclipse plug-ins, which are freely available from the SODM+T project website [6].

The exhaustive algorithm tries all paths from the source of the graph to each sink and is quite straightforward, but inefficient. The iterative algorithm obtains higher performance at the cost of increased complexity. The algorithms have been tested with manually and automatically generated models, producing equivalent and satisfactory results. We need to perform a more formal analysis to find the exact conditions for the worst case of the incremental algorithm: random search did not produce any concrete examples.

The algorithms do not take into account the fact that the same task may be reused in more than one workflow or more than once in the same workflow. There are simple (though suboptimal) ways to work around this. Checking if the algorithms can feasibly take advantage of forcing several tasks to have the same constraint would be interesting.

After narrowing the conditions for the worst case of the incremental algorithm, our highest priority is improving the Eclipse plugins which implement the algorithms and the model editors. Right now, the Eclipse plugins only work on simplified UML activity diagrams: we would like to adapt them to notations which can be readily executed by process engines, such as WS-BPEL 2.0, BPMN 2.0 or jPDL. Once the plugins are ready, we will look into helping the user generate the actual test cases for performance testing.

## IX. Acknowledgements

This work was partly financed by the research scholarship PU-EPIF-FPI-C 2010-065 of the University of Cádiz. We would like to thank our colleague Francisco Palomo-Lozano for his insightful advice on the design and analysis of the algorithms.

## References

[1] C. U. Smith and L. G. Williams, "Software performance engineering," in *UML for Real: Design of Embedded Real-Time Systems*, L. Lavagno, G. Martin, and B. Selic, Eds. The Netherlands: Kluwer, May 2003, pp. 343–366.

[2] Object Management Group, "Business Process Modeling Notation 2.0 - Beta 2," Jun. 2010. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/Beta2/

[3] OASIS, "Web Service Business Process Execution Language (WS-BPEL) 2.0," http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, Apr. 2007.

[4] J. Community, "jBPM4 developers guide," http://docs.jboss.com/jbpm/v4/devguide/html_single/, Jul. 2010.

[5] A. García-Domínguez, I. Medina-Bulo, and M. Marcos-Bárcena, "Inference of performance constraints in Web Service composition models," *CEUR Workshop Proceedings of the 2nd International Workshop on Model-Driven Service Engineering*, vol. 608, pp. 55–66, Jun. 2010. [Online]. Available: http://ceur-ws.org/Vol-608/paper5.pdf

[6] A. García-Domínguez, "Homepage of the SODM+T project," https://neptuno.uca.es/redmine/projects/sodmt, Jan. 2011.

[7] D. Kolovos, R. Paige, L. Rose, and F. Polack, "The Epsilon Book," http://www.eclipse.org/gmt/epsilon, 2010.

[8] Object Management Group, "UML Profile for Schedulability, Performance, and Time (SPTP) 1.1," http://www.omg.org/spec/SPTP/1.1/, Jan. 2002.

[9] ——, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QFTP) 1.1," http://www.omg.org/spec/QFTP/1.1/, Apr. 2008.

[10] A. Avritzer and E. J. Weyuker, "Deriving workloads for performance testing," *Software: Practice and Experience*, vol. 26, no. 6, pp. 613–633, 1996. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6<613::AID-SPE23>3.0.CO;2-5

[11] M. Woodside, G. Franks, and D. Petriu, "The future of software performance engineering," in *Proceedings of Future of Software Engineering 2007*, 2007, pp. 171–187.

[12] D. C. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-based Derivation of LQN Models from UML Specifications," in *Proceedings of the 12th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS 2002)*, ser. Lecture Notes in Computer Science. London, UK: Springer Berlin, 2002, vol. 2324, pp. 159—177.

[13] J. P. López-Grao, J. Merseguer, and J. Campos, "From UML activity diagrams to Stochastic Petri nets: application to software performance engineering," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 25–36, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=974043.974048

[14] M. Tribastone and S. Gilmore, "Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile," in *Proceedings of the 7th International Workshop on Software and Performance*. Princeton, NJ, USA: ACM, 2008, pp. 67–78. [Online]. Available: http://portal.acm.org/citation.cfm?id=1383569

[15] G. A. Silver, A. Maduko, J. Rabia, J. Miller, and A. Sheth, "Modeling and simulation of quality of service for composite web services," in *Proceedings of 7th World Multiconference on Systemics, Cybernetics and Informatics*. International Institute of Informatics and Systems, Nov. 2003.

[16] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, "Quality of service for workflows and web service processes," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 3, pp. 281–308, Apr. 2004. [Online]. Available: http://www.sciencedirect.com/science/article/B758F-4C5HRYG-1/2/ff541da65f6a997acb0554e0ab2f5dd1

[17] K. Suzuki, T. Higashino, A. Ulrich, T. Hasegawa, A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "Model-based generation of testbeds for web services," in *Testing of Software and Communicating Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5047, pp. 266–282. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68524-1_19