Computing performance requirements for web service compositions

Antonio García-Domínguez[a], Francisco Palomo-Lozano[b,*], Inmaculada Medina-Bulo[b], Alfredo Ibias[c] and Manuel Núñez[d]

[a]Department of Computer Science, University of York, Deramore Lane, Heslington, York, YO10 5GH, UK
[b]Escuela Superior de Ingeniería, University of Cádiz, Avenida de la Universidad de Cádiz, 10, Puerto Real, Cádiz 11519, Spain
[c]Personalised Health Data Science research group, Sano – Centre for Computational Personalised Medicine, Krakow, Poland
[d]Design and Testing of Reliable Systems research group, Universidad Complutense de Madrid, C/ Profesor García Santesmases 9, Madrid 28040, Spain

ARTICLE INFO

Keywords:
Web Service Compositions
Workflow Models
Non-functional Requirements
Performance Requirements

Abstract

In order to produce service compositions, modern web applications now combine both in-house and third-party web services. Therefore, their performance depends on the performance of the services that they integrate. At early stages, it may be hard to quantify the performance demanded from the services to meet the requirements of the application, as some services may not be available or may not provide performance guarantees. The authors present several algorithms that compute the required performance for each service from a model of a service composition at an early stage of development. This is also helpful when testing service compositions and selecting candidate web services, enabling performance-driven recommendation systems for web services that could be integrated into service discovery. Domain experts can annotate the model to include partial knowledge on the expected performance of the services. We develop a throughput computation algorithm and two time limit computation algorithms operating on such a model: a baseline algorithm, based on linear programming, and an optimised graph-based algorithm. We conduct theoretical and empirical evaluations of their performance and capabilities on a large sample of models of several classes. Results show that the algorithms can provide an estimation of the performance required by each service, and that the throughput computation algorithm and the graph-based time limit computation algorithm show good performance even in models with many paths.

1. Introduction

The advent of Service-Oriented Architectures (SOAs) has given rise to the creation of specific technologies to combine external services, often web services, into new services named *service compositions*. Several special-purpose languages, such as the Web Services Business Process Execution Language (WS-BPEL 2.0) and the Business Process Model and Notation (BPMN 2.0), have been standardised [1, 2]. These languages allow software engineers to model service compositions concisely in a workflow-based notation, which is easier to understand by domain experts with no technical knowledge about their implementation.

Roughly speaking, domain experts just define the elements of the composition and how they relate to each other, while software engineers integrate the different parts required to make the service composition works. In order to quickly develop high-quality applications and reduce costs, software engineers normally reuse services from third parties or other parts of the organisation in their service compositions. Such a combination of internal and external services implies that the overall quality of service (QoS) cannot be fully understood during specification, as it depends on the QoS of the integrated services. Therefore, QoS is barely controlled at development time and may pose a threat in production.

Many different strategies have been proposed and combined in order to deal with dependencies in software systems [3]. One of the most common approaches is to sign Service Level Agreements (SLAs) with the external providers and watch the services for performance degradation. However, defining the parameters of the SLA or what are the conditions to detect a degradation in performance can be difficult: asking too much may be expensive for the service consumer, while asking too little may alienate its customers or users. QoS is usually assessed *dynamically*, by constantly monitoring the performance of services. Existing approaches have focused on computing the expected global QoS from the local QoS of each service, and using this information to select services among several candidates so that the global requirements are met [4, 5].

However, there are many cases in which either we do not know in advance the expected QoS for every service involved or better QoS comes at a higher cost. Perhaps, the data is not published by the service provider or we simply do not trust it. Alternatively, it may be the case that the service is not even implemented yet. Monitoring may be not possible in this context. Then, our only choice is to make an educated guess. Nevertheless, if we guess wrong, we will have to manually revise all the estimations, which is tedious and error-prone. Besides, we are likely to loose money, reputation, or both.

In this context, we propose using an abstract high-level model of the service composition to *statically* compute what QoS should be demanded from the services to be integrated,

*Corresponding author
a.garcia-dominguez@york.ac.uk (A. García-Domínguez); francisco.palomo@uca.es (F. Palomo-Lozano); inmaculada.medina@uca.es (I. Medina-Bulo); a.ibias@sanoscience.org (A. Ibias); mn@sip.ucm.es (M. Núñez)
ORCID(s): 0000-0002-4744-9150 (A. García-Domínguez); 0000-0002-6773-205X (F. Palomo-Lozano); 0000-0002-7543-2671 (I. Medina-Bulo); 0000-0002-3122-4272 (A. Ibias); 0000-0001-9808-6401 (M. Núñez)

so that their composition can meet the global QoS requirement of the service composition.

Needless to say, this is quite a challenging problem. First, a service composition can be executed concurrently by multiple users, which affect its workload. Second, we need to enrich the workflow-based model of the service composition with performance annotations to provide relevant information concerning performance requirements. Third, regarding the services that will integrate the composition, the QoS information can be known, unknown, or partially known. Fourth, when faced to a decision in the workflow, the domain expert should be able to provide an estimation of the probabilities with which each branch is taken. Fifth, QoS might refer to either throughput or time limits, the latter being considerably more challenging to compute.

The ultimate goal is to "fill the gaps", computing for each service the QoS that would make possible to meet the performance specification of the service composition. Should we develop a reliable method to estimate this missing information, we would be able to pick the services that better fit the task from the different web services and providers available. Aware of this information, service discovery could incorporate QoS negotiation too.

Recent work in web service recommendation systems takes into account different characteristics of web services, such as their history [6], geographic location [7], and isolated QoS [8]. However, to the best of our knowledge, there is a lack of approaches facing how to compute the performance of web service compositions from their constituent web services, particularly when complete information about the QoS of the web services is not available. This absence is probably due to two main causes. First, the automatic computation of the relevant quantities is a difficult task because it has to consider the interaction between the composition workflow and the different services involved in the composition. Strongly related to this first issue when the QoS is not known or it is just partially known, the combinatorial explosion underlying the possible interactions between services in a composition diminishes the capability of potential algorithms to compute a feasible solution in a reasonable amount of time. The need to work with incomplete information is a rather common situation in this context.

We present an algorithm to compute the throughput of the integrated services and two algorithms to compute their time limits. The first time limit computation algorithm transforms the model into several linear programming (LP) optimisation problems, which can be solved with standard LP tools. The second one is a graph algorithm that has been devised for this particular task, achieving better performance in the average case.

Many different notations and profiles have been defined to specify models featuring workflows and compositions, but, unfortunately, it is not always possible to faithfully translate to each other. Our algorithms work on models that are similar to UML activity diagrams, with some simplifications and extensions. The use of our own notation for models is not a restriction for people using a different notation. Our notation is defined by a metamodel that abstracts away details that are not necessary for the computations at hand, but it describes models that are high-level enough so that it is straightforward to translate models from different notations. For example, we will show how to transform a BPMN model into a model in our notation. We have chosen BPMN because it is becoming a *de facto* standard for developing service compositions (e.g. jBPM[1] and Bonita Platform[2] both support BPMN 2.0).

One might argue whether it would be better to stick to one formalism, either UML or BPMN. Although our primary focus is on UML, we think that our proposal has the potential to serve two communities: the Software Engineering community, using UML, and the Business-Process Modelling community, using BPMN. In fact, we would like to convey the idea that BPMN can be used for high-level modelling while UML can be used as a lower-level target to map other modelling paradigms (e.g. those based on OMG MARTE) through UML profiles. In our case, in order to increase reusability as much as possible, we have selected a very small subset of UML. This reduced set would allow users following our approach to automatically transform BPMN/BPEL models into UML by using transformation languages, such as ATL (the ATLAS Transformation Language) or ETL (the Epsilon Transformation Language) [9, 10].

Regarding the main contributions of this paper, they can be summarised as follows:

1. Three novel algorithms are presented to compute QoS performance characteristics (throughput and time limits) from a high-level annotated specification. The algorithms have been implemented and evaluated.
2. A metamodel inspired by UML activity diagrams with constraints, which is enriched with performance annotations, is presented. An Eclipse-based model editor can be used to graphically design new models, verify that they honour the metamodel, and apply the algorithms, as they are provided as Eclipse plugins.
3. Our models can take into account the number of concurrent users, which impact the workload of the service composition, the relative weight of different activities, and how many times they are repeated.
4. Our algorithms can work in a context where complete information concerning the performance of each service in isolation is not available.
5. The assignment of time limits is fair in a precise sense, so that slack time is distributed among activities according to their relative weight and how many times they are repeated.
6. Nested activities are included in the metamodel. Therefore, hierarchical models where a service composition uses another composition as one of its services can be defined. In particular, the lack of complete information about services included in nested compositions can be modelled too.

[1] https://www.jbpm.org

[2] https://www.bonitasoft.com

7. All the relevant source code and software artefacts are available in the GitHub repository https://github.com/agarciadom/sodmt under the Eclipse Public License 2.0.

The rest of the paper is structured as follows. Section 2 introduces our service composition metamodel, the performance annotations that we use, and a complete example of translation from a model specified in BPMN 2.0. Next, Section 3 introduces the general approach in an intuitive manner, explains our algorithms in detail, applies them to a simple running example, and explains the key optimisations used. Then, Section 4 discusses their implementation and analyses their theoretical and empirical performance. Section 5 identifies the limitations of our approach. Section 6 is devoted to discuss related work. Finally, Section 7 presents our conclusions and future work.

2. Service composition model for performance

This section presents how we model service compositions. The models that we use are performance-aware and can be used by our algorithms to compute relevant QoS information. These models resemble UML activity diagrams, but the available set of elements is restricted to those required by the algorithms. Besides, the executable nodes and control flow elements in a model can be annotated with optional performance annotations, in addition to a mandatory global performance constraint specified for the whole model.

After describing the core elements of the metamodel and their semantics, the available performance annotations are presented. This section concludes with a running example specified in BPMN 2.0 and its mapping to our notation.

2.1. Core elements and constraints

Figure 1 shows a UML class diagram with the elements of the metamodel. The models described by this metamodel can be regarded as a simplification of UML activity diagrams with some extensions allowing us to attach performance annotations. For the sake of readability, classes Activity and ActivityNode have been repeated in the diagram (please, notice the arrow mark in their upper-right corner).

An Activity contains a set of ActivityNodes connected by directed ActivityEdges modelling their control flow. It also includes one global PerformanceAnnotation, which establishes the *global constraint* of the model. Several kinds of ActivityNodes can be specified:

- An ExecutableNode encapsulates behaviour, which is described by its name. It may have a LocalPerformanceAnnotation describing the information known at the time by the modeller. There are two kinds of executable nodes:
  - The StructuredActivityNode, which simply acts as containers of zero or more activity nodes with their internal control flow.
  - The Action, which implements some atomic unit of behaviour.

  Together, they allow for a precise description of the structure of the workflow and its performance expectations at different levels of abstraction.
- An InitialNode is the single starting point of all execution paths in the model or in a certain StructuredActivityNode.
- A FinalNode terminates the current execution path, as flow final nodes do in UML. More than one final node is allowed in a workflow or a StructuredActivityNode.
- A DecisionNode selects the outgoing branch whose condition holds. Every outgoing edge from a DecisionNode should have a non-empty condition and probability of activation (a number between 0 and 1). The sum of all the probabilities across the outgoing edges of a DecisionNode should be equal to 1. Probabilities are set by the domain expert, as they depend on domain knowledge.
- A MergeNode brings together, at least, two mutually exclusive paths that diverged at a previous DecisionNode.
- A ForkNode continues execution through several concurrent paths.
- A JoinNode brings together, at least, two concurrent paths that diverged at a previous ForkNode, waiting for all of them to join.

Valid models have to meet additional constraints too:

1. ForkNodes and DecisionNodes must have two or more outgoing ActivityEdges. FinalNodes must not have outgoing edges. The rest must have exactly one outgoing edge.
2. JoinNodes must have two or more incoming ActivityEdges. InitialNodes must not have any incoming edges. The rest must have exactly one incoming edge.
3. There must be exactly one InitalNode outside all StructuredActivityNodes and one inside each StructuredActivityNode.
4. JoinNodes and MergeNodes must only join paths that diverged at the same ForkNode or DecisionNode, respectively. Please, notice that this guarantees that forks and decisions are "balanced".
5. Every execution path in the graph must start at an InitialNode and end at a FinalNode, and every node must belong to at least one execution path (otherwise, it would never be run and therefore should be removed).
6. Execution paths cannot cross a StructuredActivityNode to directly leave or enter any contained ActivityNode: the model has to be *well structured*.
7. The underlying directed graph must be acyclic. In order to model loops, the number of iterations can be specified by using local performance annotations.

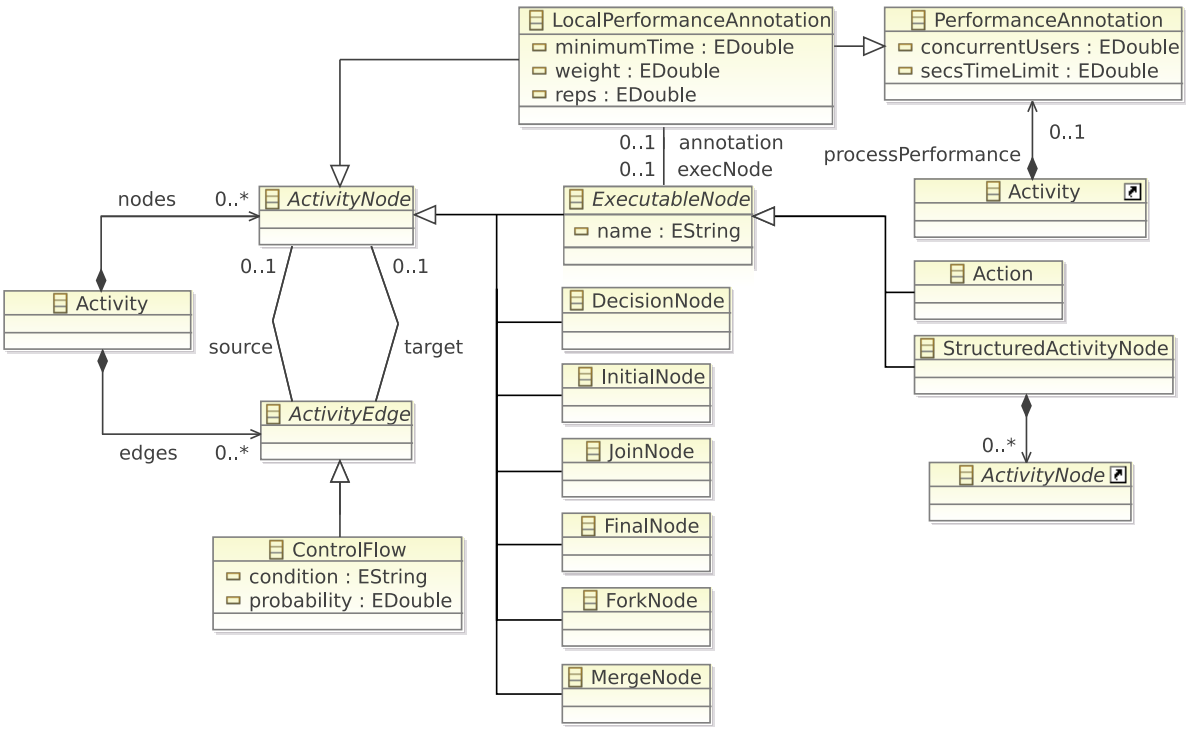


Figure 1: UML class diagram of our workflow metamodel (simplification of UML activity diagrams)

2.2. Performance annotations

There are two kinds of performance annotations:

1. The mandatory global PerformanceAnnotation. The modeller must set concurrentUsers to how many users per second are expected to start the workflow. Besides, secsTimeLimit must be set to the time in seconds in which all the paths in the workflow should have finished their execution under the specified workload.
2. The optional LocalPerformanceAnnotations for the executable nodes in the workflow. A relevant feature of our algorithms is that the local values for concurrentUsers and secsTimeLimit are automatically computed. In a local performance annotation, minimumTime is the minimum time in seconds that should be allotted to the node, while weight is a relative measure of how computationally intensive the node is. Having a weight of 3 roughly means that its execution may take up to three times longer than in nodes with weight 1, after considering all the minimum times. Finally, reps is the expected number of iterations (at least, one) the node will go through.

By default, attributes minimumTime, weight and reps are set to 0, 1, and 1, respectively. These values model the simplest case: a node with unknown minimum execution time and unit weight, whose execution is not repeated. We will formally define these concepts in Section 3.

2.3. Algorithm inputs and high-level behaviour

Our algorithms take as their input a valid model, with respect to the metamodel given in Figure 1 and the constraints described in Sections 2.1 and 2.2. Then, they process the model to extract all the necessary input variables.

1. Workload and time limit, as specified in the global PerformanceAnnotation.
2. Minimum time, weight and number of repetitions specified in each LocalPerformanceAnnotation.

The algorithms update the concurrentUsers and secsTimeLimit attributes of the LocalPerformanceAnnotation for each ExecutableNode with the computed values. These include local computations in the service composition, whose cost is usually negligible, and the performance requirements for each web service in the composition.

The values assigned to minimumTime and weight can model several common scenarios depending on what the modeller knows about the expected time limit of a node. Let $m \geq 0$ denote the minimum time and $w \geq 0$ denote the weight. The following mutually exclusive situations arise:

- $m = 0, w = 0$. In this case, the node execution costs nothing. This is not used with an ExecutableNode. Rather, these are the default values for every ActivityNode that is not an ExecutableNode. This makes them effectively invisible to the algorithms, except for how they branch and join the execution paths.
- $m > 0, w = 0$. In this situation, we have a node with a fixed time limit, as no extra time is allotted beyond the minimum time m. This usually means that there is a strict Service Level Agreement (SLA) for the web service or software component represented by the

node, which ensures that it will finish exactly within m seconds.

- $m = 0, w > 0$. If we have this combination of values, then time will be automatically allotted. There is no known SLA or estimate of how long it could take. Instead, the modeller must compare the cost of the node with the rest of nodes in the workflow.
- $m > 0, w > 0$. In this scenario, part of the allotted time is set manually and the rest is computed automatically. This can be useful if we have previous measurements that point to a certain minimum time, but still want to grant some of the remaining time.

Finally, let us briefly describe a key characteristic of our algorithms: they support nested activities. First, an initial pass is run on the outermost activities, computing their local performance requirements. These local requirements are then used in later passes as global requirements for the activities nested inside them. This allows the algorithms to descend recursively through the model, starting at the outermost layer and proceeding until only atomic actions are left.

2.4. Notation

Next, we introduce some concepts and notations that will be used to define our algorithms.

- $s(e)$ and $g(e)$ are the source and target ActivityNodes of ActivityEdge e, respectively.
- $i(n)$ and $o(n)$ are the incoming and outgoing edges of node n, respectively.
- $T(I)$ denotes the *throughput*, that is, the number of requests per second (henceforth, "req/s") entering the outermost InitialNode I.
- $S_w(p_A)$ denotes the additional time per unit of weight beyond the minimum time needed for the actions appearing in p_A. That is, this is the *slack* available per unit of weight.
- $L > 0$ is the global time limit of the model, in seconds.
- $C(L) = \{(m, w) \mid 0 \leq m \leq L \wedge w \geq 0\}$ is the set of all valid minimum time and weight constraints under a global time limit L.
- $c(n) = (m(n), w(n)) \in C(L)$ is the constraint associated to node n, where $m(n)$ is the minimum time limit of n and $w(n)$ is its weight.
- $r(n)$ is the number of times that node n is run (see Section 2.2). If n is inside a StructuredActivityNode, it is the number of times that n will be run each time its container is run.
- $c(p) = (m(p), w(p)) \in C(L)$ is the constraint associated to path p, where $m(p) = \sum_{n \in p} m(n)r(n)$ (the total minimum time through p) and $w(p) = \sum_{n \in p} w(n)r(n)$ (the total weight through p). Please, notice that the number of iterations corresponding to each node in p is taken into account in $m(p)$ and $w(p)$.
- $P_S(n)$ is the set of all paths starting at the node n.
- $d(n)$ is the *depth* of the node n. This value is defined as 0 when n does not belong to any StructuredActivityNode, and as $1 + d(n_s)$ when n belongs to the StructuredActivityNode n_s.
- We define *layer* D of the model as the set of nodes n such that $d(n) = D$. Layer 0 is the *global layer*, containing the topmost InitialNode.

2.5. Running example

In the previous sections we described the core elements that constitute our models, the information stored by our performance annotations, as well as the relevant parameters and high-level behaviour of our algorithms. Next, we show how to derive a non-trivial model in our notation from a web service composition modelled in BPMN 2.0, and explain the graphical notation used.

The original BPMN 2.0 model is shown in Figure 2. It is a collaboration between several partners: the client that invokes the web service composition implementing the model, the order processing system ("Orders") that executes the composition, the web services provided by the billing department ("Billing"), and the web services provided by the logistics department ("Logistics"). Each partner is shown in a different pool.

The current web service composition is contained in the "Orders" pool, which is the only executable pool. Its contents may be either natively executed by the execution engine or mapped first to a different language, like WS-BPEL 2.0.

This web service composition follows these steps:

1. Receive a message from the client with the order.
2. Evaluate the order using a set of internal business rules.
3. If the order is rejected, then close the order and notify the client.
4. If the order is accepted, then:
   (a) Divide the order into segments, to ensure that the customer receives each item as soon as possible.
   (b) For each segment, create the invoice and perform the payment at the same time the shipment order is sent. This is done by invoking the web services provided by the other departments.
   (c) Close the order and notify the client.

Mapping the previously described BPMN 2.0 model to the notation used by our algorithms is straightforward. The resulting model can be seen in Figure 3. We can study the performance needed by the different tasks and the invoked web services by executing our algorithms on this model. The model includes a global annotation with the global time limit L for all execution paths, and the throughput $T(I)$ specified as the number of requests per second entering the outermost InitialNode I. Every ExecutableNode includes an (m, w, r)

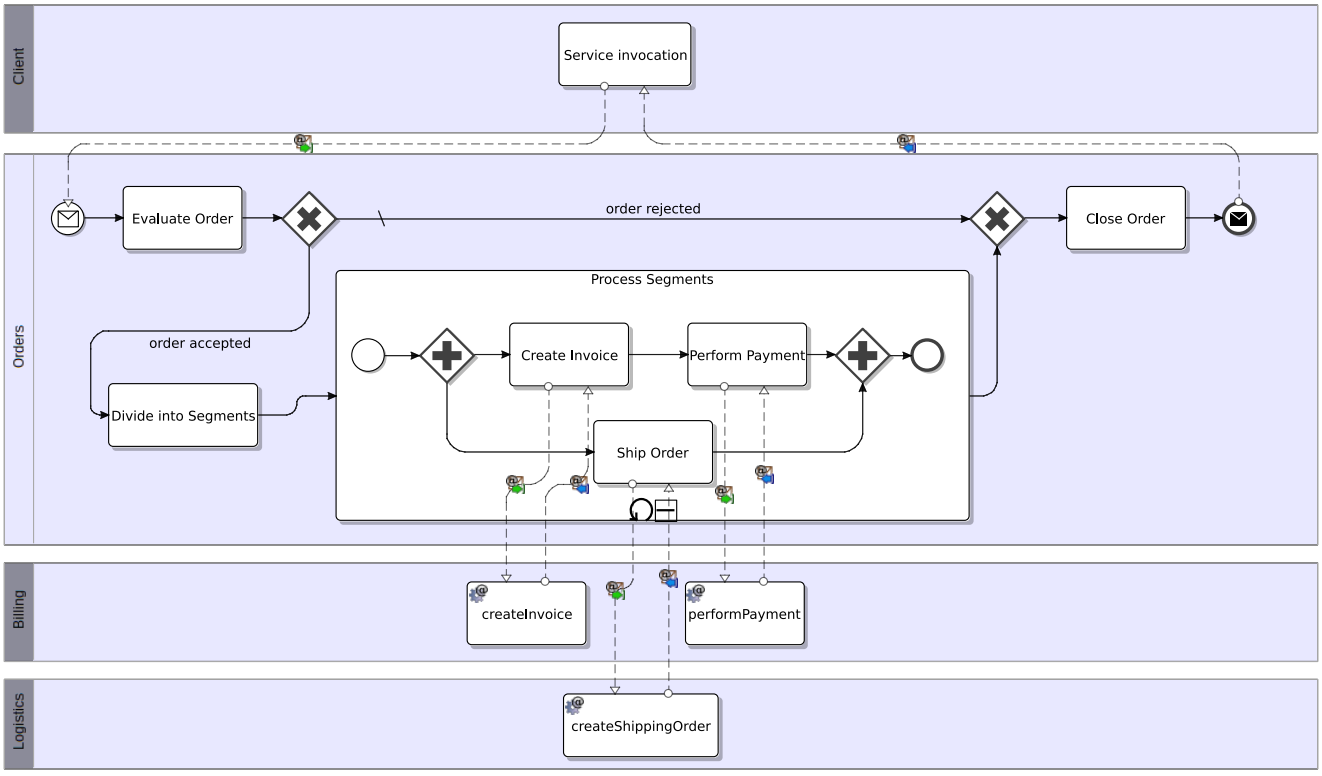


Figure 2: Running example with a BPMN collaboration between web services for processing an order

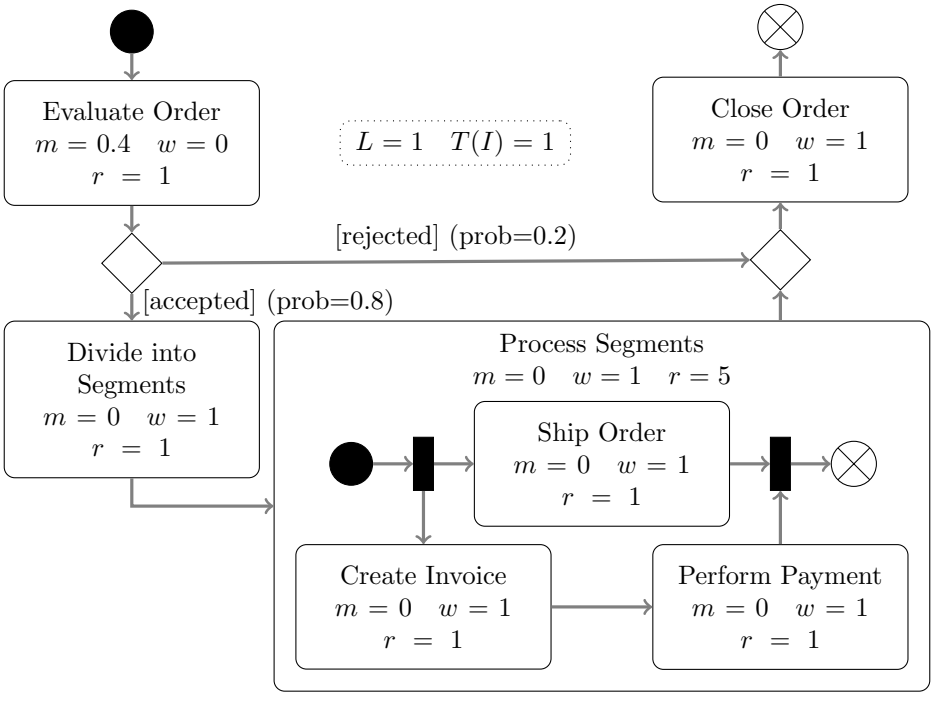


Figure 3: Annotated model for the running example, mapped from BPMN and defined using the metamodel in Figure 1.

tuple with the minimum time, weight and number of repetitions, corresponding to its local performance annotations. Decision branches include the estimated traversal probabilities.

Next, we describe the steps followed to obtain this model from the BPMN 2.0 model given in Figure 2:

1. The model was created from the contents of the only executable pool, "Orders". This is the actual process to be executed: the other pools only provide contex-

tual information.
2. Start and end events were mapped to InitialNodes and FinalNodes, respectively.
3. Atomic tasks were mapped to Action nodes, and tasks containing other tasks within them were mapped to StructuredActivityNodes.
4. The pair of exclusive gateways (marked with ×) were mapped to a DecisionNode and a MergeNode.
5. The pair of parallel gateways (marked with +) were mapped to a ForkNode and a JoinNode.
6. Local performance annotations were added to the ExecutableNodes. Most annotations had minimumTime set to 0 and weight set to 1, as we did not have any solid estimates of their execution times or computational cost. This is quite common at an early stage of development, when the actual services have not been implemented yet.

In addition, we set minimumTime to 0.4 s for "Evaluate Order", ensuring it receives at least 0.4 s. Since its weight is set to 0, it will not get any more time than that: it will be allotted exactly 0.4 s. We use this combination of values to represent the situation in which a strict SLA saying that it should never take longer than 0.4 s had been signed.

Finally, all Action nodes have reps set to 1. However, the StructuredActivityNode "Process Segments" was originally a loop, as indicated by its circular arrow symbol on the bottom edge in the BPMN 2.0 model. For that reason, we set reps to an estimate of the maximum number of segments that we would expect to see in an order (in this case, 5).

3. Algorithms

In the previous section, we described the formalism and metamodel, based on annotated UML activity diagrams, for our models. Next, we introduce the algorithms that will take one of these models and compute the desired figures.

First, we give an intuitive explanation to describe the general approach underlying the behaviour of our algorithms, together with some notation that will be used in their definitions. Then, we design and implement three algorithms. The first one computes the expected throughput of each ExecutableNode. The remaining two algorithms compute the time limit of each ExecutableNode, the latter being a more complex, but also more efficient, alternative to the former. A textual example is provided in Section 3.3.4. Additional graphical animations illustrating a step-by-step execution for each algorithm presented in Sections 3.2, 3.3.2 and 3.3.3 are available at https://agarciadom.github.io/sodmt/algorithms.

3.1. Intuitive approach

First, we will provide an intuitive description of what the algorithms try to achieve, based on the running example depicted in Figure 3.

3.1.1. Throughput computation

To begin with, we focus on the *global layer* (zero depth) and consider the global requirement associated with the number of requests per second. In our case, we have $T(I) = 1$ (see the dotted box in Figure 3), that is, 1 req/s will come through the InitialNode (depicted as a black circle) and reach "Evaluate Order". Therefore, its expected throughput will be 1 req/s too. Then, we observe that the domain expert has estimated that 80% of the requests will go through the DecisionNode to "Divide into Segments", while 20% of the requests will end up in "Close Order" (see value of "prob" associated to each branch in Figure 3). Therefore, the expected throughput of "Divide into Segments" will be 0.8 req/s. Obviously, this is also the amount corresponding to "Process Segments". Finally, all the incoming requests will merge into the MergeNode preceding "Close Order", so that its required throughput is again 1 req/s.

After the global layer is done, it is possible to obtain the throughput for the actions within "Process Segments" (depth one). Since we already know that 0.8 req/s will come through its nested InitialNode and both inner paths need to be concurrently executed, we can conclude that "Ship Order", "Create Invoice" and "Perform Payment" must all be able to handle 0.8 req/s.

Therefore, it can be observed that the process to compute the throughput from our models is quite simple and can be easily automated by traversing the graph in *topological order*. The algorithm for throughput computation will be given in Section 3.2.

3.1.2. Time limit computation

In order to compute time limits, every possible path starting from the InitialNode has to be taken into account. However, some paths may impose stricter constraints than others. Returning to our running example, there are two paths: one for rejected orders (that we call p_R) that skips over "Divide into Segments" and "Process Segments", and one for accepted orders (that we call p_A) that passes through them. Clearly, p_A is stricter, as it runs everything in p_R and more.

In order to decide the time limits for the actions appearing in p_A, we just need to sum their minimum times and weights, after multiplying them by the number of times that the corresponding action is repeated. The total minimum time through p_A is $m(p_A) = (0.4{\cdot}1{+}0{\cdot}1{+}0{\cdot}5{+}0{\cdot}1)\,\text{s} = 0.4\,\text{s}$. Therefore, we know that we have $L - m(p_A) = 0.6\,\text{s}$ left to distribute among the actions in p_A. Since the total weight through p_A is $w(p_A) = 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 5 + 1 \cdot 1 = 7$, we conclude that the slack available per unit of weight is given by $S_w(p_A) = 0.6/7 \approx 0.086\,\text{s}$.

The last step is using $S_w(p_A)$ to compute the time limits of each action in p_A. For "Evaluate Order", it is exactly 0.4 s, as it has zero weight. For "Divide into Segments" and "Close Order", it is 0.086 s, as their minimum time is 0. For "Process Segments", the minimum time is also 0, but each of the 5 repetitions gets 0.086 s for a total of 0.43 s.

This process can be now repeated within "Process Segments", using the 0.086 s time limit as its "global" constraint. There are only two paths, which do not share any activities. We will visit the bottom path first, as it has a higher total weight (2 instead of 1). The bottom path allocates $0.086/2 = 0.043\,\text{s}$ of slack per unit of weight to "Create Invoice" and

"Perform Payment", and the top path allocates 0.086 s per unit of weight to "Ship Order". The resulting time limits will be 0.043 s for "Create Invoice" and "Perform Payment" and 0.086 s for "Ship Order".

While this process is simple, it is somewhat complex to automate fully in an efficient manner. Our first approach (described in Section 3.3.1) is to produce an optimisation problem and solve it with standard linear programming machinery. Unfortunately, time can grow fast as the number of overlapping subpaths increases. In the worst scenario, the number of paths could grow exponentially with the size of the model, producing a combinatorial explosion. The second approach (described in Section 3.3.2) can discard uninteresting paths as soon as possible, keeping the size of the problem under control and avoiding the combinatorial explosion.

3.2. Throughput computation

The algorithm source code is publicly available at file `concurrent_users.eol` under repository `https://github.com/agarciadom/sodmt`. Next, we describe the algorithm in detail.

We will define T as a function which takes a node or edge and produces its expected throughput. The formula to be applied depends on the type of element passed to it. Formally:

- For an ActivityEdge e, $T(e) = \mathcal{P}(e) \cdot T(s(e))$, where $\mathcal{P}(e)$ is the probability of e being traversed.
- For the InitialNode I, $T(I)$ is equal to the throughput of the global performance annotation if I is not part of any StructuredActivityNode. Otherwise, $T(I)$ is equal to the throughput of the StructuredActivityNode it belongs to.
- For a JoinNode n, $T(n) = \min_{e \in i(n)} T(e)$, since requests in the least performing branch set the pace.
- For a MergeNode n, $T(n) = \sum_{e \in i(n)} T(e)$, as requests from mutually exclusive branches are reunited.
- For any other type of node n, $T(n) = T(e_1)$, where $e_1 \in i(n)$ is its only incoming edge.

The algorithm traverses the model in breadth-first order, starting from the InitialNode and continuing through its outgoing edges. This breadth-first order avoids computing the same values several times by annotating each edge e and node n with its value for $T(e)$ and $T(n)$, respectively. The concurrentUsers attribute of the local performance annotation of each ExecutableNode n is updated to $T(n)$.

For example, let us compute T(Create Invoice) for the model shown in Figure 3, which needs to handle $T(I) = 1$ request per second. The result can be computed as follows.

$$\begin{aligned} T(\text{Create Invoice}) &= T(\text{ForkNode in Process Segments}) \\ &= T(\text{InitialNode in Process Segments}) \\ &= T(\text{Process Segments}) \\ &= T(\text{Divide Segments}) \\ &= 0.8 \cdot T(\text{Evaluate Order}) \\ &= 0.8 \cdot T(I) \\ &= 0.8 \end{aligned}$$

As we will see in Section 4.2.1, complexity is linear in the size of the model (the number of nodes and edges) in the model graph, as the algorithm time is dominated by the topological sorting stage, which can be efficiently implemented by depth-first search, or obtained as a byproduct of Tarjan's algorithm for strongly-connected components [11].

3.3. Time limit computation

Computing the time limits of actions inside activities is considerably more complex than computing their required throughput. We will first devise an algorithm that produces time limits by solving a linear programming problem. This algorithm can be used as a test oracle and performance baseline for better algorithms. In particular, later in this section, an optimised graph-based algorithm will be defined. Finally, both algorithms will be applied to our running example.

3.3.1. Linear programming-based algorithm

The algorithm source code is publicly available at file `generate-glpk-input.egl` in repository `https://github.com/agarciadom/sodmt`. Next, we describe the algorithm in detail.

The simplest way to describe the computation of time limits is expressing the problem declaratively by translating its goal and constraints into an optimisation problem. Next, we will formulate time limit computation as a linear programming problem, which is quite convenient as efficient tools for solving linear programs are readily available. First, we present the problem if we only consider one layer and then we generalise to several layers.

Linear programming-based algorithm: one layer

For the sake of simplicity, let us assume for a moment that the model has only one layer, that is, that there are no StructuredActivityNodes and, thus, there is only one InitialNode, I, at zero depth. Let N be the set of all ExecutableNodes, $P = P_S(I)$ be the set of all the reachable paths from I, and $P_C(n) = \{p \mid p \in P \wedge n \in p\}$ be the set of all the paths in P that contain n.

In order to formulate the optimisation problem, we will need to associate a value, $S_w(n)$, to every $n \in N$. $S_w(n)$ denotes the *slack per unit of weight* of node n. In other words, this figure indicates the additional time beyond $m(n)$ that is assigned to n *per unit of weight*. Therefore, the resulting time limit *for a single execution* of node n is given by $m(n) + S_w(n) \cdot w(n)$ and the activity associated to the (executable) node n, which can be repeated $r(n)$ times, should terminate its execution in the following time interval:

$$[m(n) \cdot r(n), (m(n) + S_w(n) \cdot w(n)) \cdot r(n)]$$

Therefore, the time available can be maximised over all paths for $x_n = S_w(n)$, resulting in more lenient time limits whenever possible:

$$\arg\max \sum_{p \in P} \sum_{n \in p} (m(n) + x_n \cdot w(n)) \cdot r(n) \qquad (1)$$

However, there are two *validity constraints* to be taken into account:

R1. $S_w(n) \geq 0$, for all $n \in N$
R2. $\sum_{n \in p}(m(n) + S_w(n) \cdot w(n)) \cdot r(n) \leq L$, for all $p \in P$

The first validity constraint just denotes that the assigned times must not be negative. The second validity constraint guarantees that the assigned times cannot make a path to violate the global time limit.

However, these two constraints are not enough, because we could obtain unfair and undesired solutions. For example, it would be valid (but not fair) to assign all the slack time to the first node in a path, sharing no time with the rest of the nodes. In order to avoid this situation, we introduce *fairness constraints* ensuring that time values are evenly distributed according to the weight of each activity.

First, it is clear that if an executable node n belongs to a single path p, then an optimal value for $S_w(n)$ is:

$$\begin{cases} \dfrac{L - m(p)}{w(p)} & \text{if } w(p) > 0 \\ 0 & \text{if } w(p) = 0 \end{cases}$$

This would evenly distribute the remaining time after taking into account the minimum time limits, $L - m(p)$, in proportion to the relative weight of the nodes belonging to p over their sum, that is, $w(p)$. Please, notice that if $w(p) = 0$ then $S_w(n) = 0$, as no time can be distributed.

However, an executable node might appear in several paths. Fortunately, this scheme can be extended to nodes appearing in multiple paths by introducing two *fairness constraints* and taking into account the *strictest path* for the node under consideration:

R3. $S_w(n) \geq \min_{p \in P_C(n)} \{(L - m(p))/w(p) \mid w(p) > 0\}$, for all $n \in N$.
R4. $S_w(m) = S_w(n)$, for all $m, n \in N$ such that $P_C(m) = P_C(n)$.

The first fairness constraint guarantees that the slack per unit of weight is not shorter than the slack available on the strictest path that n belongs to. The second fairness constraint ensures that whenever two nodes appear in the same set of paths, the time remaining is distributed among them in proportion to their weights.

This formulation induces the linear programming problem in Equation 2, which can be readily solved for a single layer of the model graph. Please, notice that if the linear problem is unfeasible, then the model contains inconsistent requirements on the composition. For instance, if the global time limit is 10 s and one of our actions requires at least 15 s, no assignment of time will ever be possible. Therefore, our algorithms could be used to check the validity of a model from a semantic point of view, and developers can get reported of inconsistent or too demanding temporal requirements at an early stage of the development process.

$$\arg\max \sum_{p \in P} \sum_{n \in p} (m(n) + x_n \cdot w(n)) \cdot r(n)$$

subject to:

$$\begin{aligned} & x_n \geq 0, \quad \text{for all } n \in N \\ & \sum_{n \in p} (m(n) + x_n \cdot w(n)) \cdot r(n) \leq L, \quad \text{for all } p \in P \\ & x_n \geq \min_{p \in P_C(n)} \left\{ \frac{L - m(p)}{w(p)} \;\middle|\; w(p) > 0 \right\}, \quad \text{for all } n \in N \\ & x_m = x_n, \quad \text{for all } m, n \in N \text{ such that } P_C(m) = P_C(n) \end{aligned} \tag{2}$$

Linear programming-based algorithm: multiple layers

Generalising this approach to models with more than one layer of depth is simple: our algorithm first runs on layer 0, producing time limits for each topmost ExecutableNode (including any StructuredActivityNode). Then, it is run on the contents, lying in layer 1, of each StructuredActivityNode in layer 0, using the previously computed time limits as its "global" constraint. This process continues layer by layer, until the whole model has been annotated.

The above formulation can be implemented in any of the existing mathematical programming languages. In our case, we have selected GMPL (GNU MathProg Language), which is included in the GNU Linear Programming Kit[3]. GMPL is a very concise notation for linear programming and a subset of the AMPL language [12].

In GMPL, the problem can be split into two sections. The *model* section describes the available parameters, variables, constraints and objective function. The *data* section provides values for some of the parameters. This is useful for reusing the same problem with different data. The final result appears in Listing 1.

3.3.2. Graph-based algorithm

While the formulation based on linear programming is easy to understand and implement, it suffers from an exponential grow in the size of problem instances, since it needs to check every path from the InitialNode. In this section, we will introduce a graph-based algorithm that builds the set of paths under study incrementally, by culling uninteresting subpaths as soon as possible, therefore mitigating the impact of a potential combinatorial explosion.

The algorithm source code is publicly available at file `time_limits.eol` under `https://github.com/agarciadom/sodmt`. A detailed account of the algorithm follows.

Again, in order to present the algorithm, we will simplify the description by assuming that the model has only one layer. Multiple layers are handled in the same way as in the previous algorithm: once we have computed the values for all the layer i ExecutableNodes, we apply our algorithm to the contents of each StructuredActivityNode in layer i by using the previously computed time limits as its "global" requirement.

[3] `http://www.gnu.org/software/glpk`

Listing 1: GMPL model used by the LP algorithm.

```
set A; # All executable nodes
set P; # All paths
param G; # Global time limit
param m{a in A}, default 0; # Minimum times
param w{a in A}, default 1; # Weights
param r{a in A}, default 1; # Repetitions

param paths{a in A, p in P}, default 0, binary;
# Paths (1 if action belongs to path)

# Total minimum time and weight of each path
param mp{p in P} := sum{a in A} paths[a, p] * m[a] * r[a];
param tw{p in P} := sum{a in A} paths[a, p] * w[a] * r[a];

# Minimum slack per unit of weight by task
param msuw{a in A} := min{p in P: paths[a, p] == 1 && tw[p] > 0} (G - mp[p]) / tw[p];

# Slack per unit of weight for each action (must be positive)
var suw{a in A} >= 0;

maximize usage: sum{a in A, p in P} (paths[a, p] * (suw[a] * w[a] + m[a]) * r[a]);

subject to glimit {p in P}: sum{a in A} (paths[a, p] * (suw[a] * w[a] + m[a]) * r[a]) <= G;
subject to minslack {a in A}: suw[a] >= msuw[a];
subject to samepaths {a in A, b in A: a < b && forall {p in P} paths[a, p] == paths[b, p]}: suw[a] == suw[b];

solve;
```

We will first introduce some additional notation. First, in this section we say that the available time "flows" from the InitialNode. If a node n receives $0 \leq t(n) \leq L$ seconds, then every path $p \in P_S(n)$ receives $t(p) = t(n)$ seconds to distribute among its nodes. Initially, we only know that $t(I) = L$.

If the local and global annotations are consistent with each other, then $t(p) \geq m(p)$ for every path p: the minimum time constraints of all actions are always met. The value $s(p) = t(p) - m(p) \geq 0$ is known as the *slack* of the path p and it is distributed over p according to the weight of each node: the slack per unit of weight initially assigned to each node, denoted by $S_w(p)$, is given by

$$\begin{cases} \dfrac{s(p)}{w(p)} & \text{if } w(p) > 0 \\ 0 & \text{if } w(p) = 0 \end{cases}$$

Please, notice that $w(p) = 0$ implies $S_w(p) = 0$ because all nodes in p have a zero weight and, therefore, no slack can be distributed.

The algorithm must ensure that whenever $w(p) > 0$, we also have $s(p) > 0$, so that every path p with a non-zero weight has some slack to distribute. If this condition is not met or the annotations are inconsistent, then the user will be notified and the execution will abort.

With these definitions, we can describe the algorithm as a recursive function taking a node n and the time assigned to it, $t(n)$. Initially, $n = I$ and $t(n) = L$, the global time limit. The algorithm follows these steps:

1. Select two paths from $P_S(n)$ that, respectively, fulfil the following properties:
   - Let $p_{ms}(n)$ be the path with the minimum value of $S_w(p)$ when $t(n)$ seconds are available. In case of a tie, pick the path with maximum $w(p)$.
   - Let $p_{Mm}(n)$ be the path with the maximum $m(p)$.
2. If $s(p_{Mm}(n)) < 0$, then the minimum time limits cannot be satisfied: notify the user and abort the execution.
3. If $s(p_{ms}(n)) = 0$ and $w(p_{ms}(n)) > 0$, then there is no slack in a path with a non-zero weight: notify the user and abort the execution.
4. Set the time limit of n, that is, $l(n)$, to $m(n)+S_w(p_{ms}(n))\cdot w(n)$. The remaining time will be $T_R = t(n)-l(n)\cdot r(n)$ seconds. Mark n as visited.
5. Sort each edge outgoing from n, that is, all edges such that $e \in o(n)$, in increasing order of $S_w(p_{ms}(g(e)))$ with $t(g(e)) = T_R$. This ensures that we continue through the subpath starting at n that has the minimum slack per unit of weight when T_R seconds are available.
6. Visit each edge in $o(n)$, as previously sorted:

(a) If the target of e has been visited before, then check whether the time which was sent to it, that is, T'_R, is strictly less than T_R, the time which would have been sent through e. In that case, use the surplus $T_R - T'_R$ seconds on the source of e and its ancestors, and send T'_R seconds through e. Go back in the graph from the source of e, collecting nodes with non-zero weights into C until a node with more than one incoming or outgoing edge is found. Increase the time limit of each collected node by

$$\frac{(T_R - T'_R) \cdot w(n)}{w(C)}$$

where $w(C) = \sum_{n \in C} w(n) \cdot r(n)$.

(b) If the target of e has not been visited before, then invoke this algorithm recursively, setting n to the target of e and $t(n) = T_R$.

We will justify that the graph-based algorithm produces the same results as the algorithm based on linear programming. In order to do this, we will show that the graph-based algorithm ensures the validity and fairness constraints of the linear program (R1 to R4 in Section 3.3.1), and that the graph traversal order fulfils its maximisation objective.

The algorithm ensures R1 (slacks per weight are not negative) by checking, in Step 2, that the slack of the path with the highest minimum time is not negative. The corner case where there is no slack in a path with a non-zero weight is considered in Step 3.

The algorithm ensures R2 (time limits cannot make a path to violate the global time limit) by transferring the remaining time from node to node. That is, each node receives all the available time and passes unassigned time (the amount of time not assigned to it) to the next. When time limits are increased, see Step 6a, we take into account the time that was not used by the nodes reached after the currently analysed node.

Concerning R3 (slacks per weight are greater than or equal to the one corresponding to the most strictest path containing the corresponding node), the algorithm uses $m(n) + S_w(p_{ms}(n)) \cdot w(n)$, in Step 4, to compute the time limit. In addition, in Step 5, the algorithm traverses the outgoing edges in decreasing order of strictness. Therefore, node n is traversed for the first time when $p_{ms}(n)$ (the strictest path including n) is being analysed, and the assigned value is given by the minimum in R3.

R4 (slacks per weight of two nodes appearing in the same set of paths is the same) is also ensured by Step 4. Essentially, two nodes a and b will be in the same paths if they are consecutive nodes and there are no alternatives. In this situation, it can be proved that $S_w(p_{ms}(a)) = S_w(p_{ms}(b))$.

Finally, the maximisation of the available time computed in Equation 2 is achieved in Step 6a. Once the strictest path has been completely covered, the algorithm will consider pending edges, traversing less strict subpaths. Note that these paths might have nodes whose time limits where previously computed as part of stricter paths, leaving additional time available to the nodes that are traversed for the first time as part of the corresponding path. In compliance with R4, this extra time is distributed in Step 6a by using an ordering between edges so as to guarantee that the same slack per weight is assigned to nodes appearing in the same path.

This could be named the *exhaustive version* of the graph-based algorithm. However, an *incremental version* is also possible if we carefully apply some key optimisations. Next, we describe this incremental approach, which considerably improves the performance of the algorithm.

3.3.3. Key optimisations of the graph-based algorithm

The graph-based algorithm uses several optimisations to improve its performance. First of all, a path p is not represented by its sequence of nodes, but by its constraint $c(p) = (m(p), w(p))$, saving much memory.

Second, in order to select $p_{Mm}(n)$ at each node we need to know the maximum $m(p)$ for each path $p \in P_S(n)$, which we will denote by $m(p_{Mm}(n))$ or simply $M_m(n)$. We can compute it in advance using with the following equation:

$$\begin{aligned} M_m(n) &= m(p_{Mm}(n)) \\ &= m(n) \cdot r(n) + \max\left\{ M_m(g(e)) \mid e \in o(n) \right\} \end{aligned} \tag{3}$$

Since (3) is recursive, we can evaluate it incrementally, starting from the FinalNodes (for which $M_m(n) = 0$) and going back up to the InitialNode in reverse topological order.

Third, to select $p_{ms}(n)$ at each node we need to know the strictest path starting from it. We cannot compute it in advance, as it depends on the time received by the node, $t(n)$, which is not known *a priori*. Instead, we will remove redundant paths from $P_S(n)$. We will call this reduced set $P'_S(n)$. A path $p_a \in P_S(n)$ is removed when it is said to be *always less or just as strict* than some other path $p_b \in P_S(n)$, independently of the time received by n or the common ancestors of p_a and p_b. We denote this by $c(p_a) \preceq_{s(L)} c(p_b)$, and define it formally as follows:

$$\begin{aligned} &(a, b) \preceq_{s(L)} (c, d) \equiv \\ &\quad \forall t \in [0, L]\, \forall x \in [0, L]\, \forall y \geq 0 \\ &\qquad a + x \leq t \wedge c + x \leq t \wedge \\ &\qquad b + y > 0 \wedge d + y > 0 \Rightarrow \\ &\qquad \frac{t - (a + x)}{b + y} \geq \frac{t - (c + x)}{d + y} \end{aligned} \tag{4}$$

After a lengthy simplification, the right side of (4) can be replaced by:

$$a \leq c \wedge (b \leq d \vee (b - d) \cdot L \leq b \cdot c - a \cdot d) \tag{5}$$

Equations (4) and (5) would consider all pairs of the form (L, x) to be just as strict. We could further reduce the number of paths to be evaluated and still obtain the same results by considering the (L, x) pair with the highest value of x as the strictest one. This results in our revised and final definition of $\preceq_{s(L)}$:

$$a \leq c \wedge (b \leq d \vee (a < L \wedge (b - d) \cdot L \leq b \cdot c - a \cdot d)) \tag{6}$$

Listing 2: GMPL data for layer 0 of the example in Figure 3.

```
set A := "Evaluate Order", "Divide into Segments", "Close Order", "Process Segments";
set P := p_1 p_2;
param G := 1.0;
param m := "Evaluate Order" 0.4;
param w := "Evaluate Order" 0.0;
param r := "Process Segments" 5.0;
param paths :=
  [*, p_1] "Evaluate Order" 1 "Close Order" 1
  [*, p_2] "Evaluate Order" 1 "Divide into Segments" 1 "Process Segments" 1 "Close Order" 1;
end;
```

Listing 3: GMPL data for layer 1 of the example in Figure 3.

```
set A := "Perform Payment", "Create Invoice", "Ship Order";
set P := p_1 p_2;
param G := 0.08571428571428572;
param paths :=
  [*, p_1] "Create Invoice" 1 "Perform Payment" 1
  [*, p_2] "Ship Order" 1;
end;
```

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.

Finally, like $M_m(n)$, $P'_S(n)$ can also be computed incrementally by traversing the graph in reverse topological order. Let n_i be a child of n. Let p_a and p_b be two paths in $P_S(n_i)$, so that $c(p_a) \preceq_{s(L)} c(p_b)$. By definition, p_a is less or just as strict as p_b regardless of their common ancestors, so $\langle n \rangle + p_a$ will also be discarded from $P'_S(n)$ over $\langle n \rangle + p_b$. This means that instead of comparing every path in $P_S(n)$ for every node n, we can build $P'_S(n)$ by adding n at the beginning of the paths in $P'_S(n_i)$, for every child n_i of n, and then filtering the redundant paths using $\preceq_{s(L)}$.

Let $\max_{\preceq_{s(L)}} S$ select the paths in S which are not always less strict than any other (the maximal elements according to $\preceq_{s(L)}$). We define $P'_S(n)$ as:

$$P'_S(n) = \max_{\preceq_{s(L)}} \left\{ t \mid e \in o(n) \wedge (M, W) \in P'_S(g(e)) \right\} \quad (7)$$

where $t = (m(n) \cdot r(n) + M, w(n) \cdot r(n) + W)$.

Finally, please notice that $P'_S(f) = P_S(f) = \{(0, 0)\}$ when f is a final node.

The source code of the resulting algorithm, which includes all the optimisations, is publicly available at file `time_limits_new.eol` under `https://github.com/agarciadom/sodmt`.

3.3.4. Examples

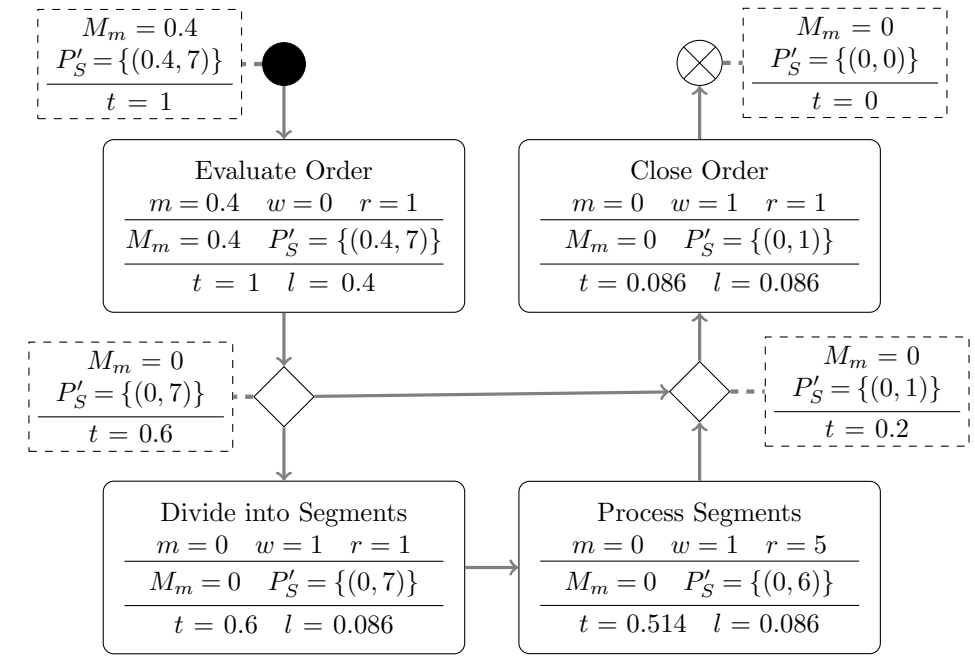
In this section we apply the two previous algorithms to our running example given in Figure 3. The global time limit will be set in both cases to $L = 1$ second. We will shorten action names to their initials when necessary. For example, "Evaluate Order" will be simply "EO".

Concerning the LP algorithm, we need to *encode* our model into GMPL for each layer of the model. This is straightforward as can be seen in Listing 2 (where layer 0 is defined). As for the time limits, the results for this layer were of 0.086 s for "Evaluate Order" (EO), "Divide into Segments" (DS), "Close Order" (CO) and each repetition of "Process Segments" (PS). Using these results, we produce the code given in Listing 3, where minimum times, weights and repetitions use the default values of 0, 1 and 1, respectively. The resulting time limits were of 0.043 s for "Perform Payment" (PP) and "Create Invoice" (CI) and 0.086 s for "Ship Order" (SO). Therefore, we have obtained a time limit for each of the six *basic* actions that conform our model (please, notice that the value for "Process Segments", computed at layer 0, is used to compute the values corresponding to the three *basic* actions processes at layer 1).

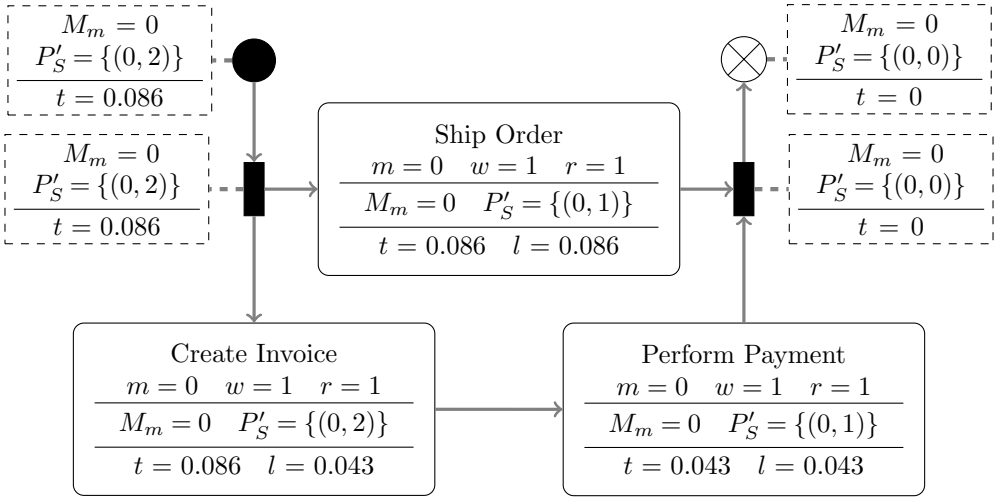
The execution trace of the graph-based algorithm over layers 0 and 1 of the running example is shown in Figure 4. In the upper part of Figure 4, showing layer 0, we consider $L = 1$ s. Every ExecutableNode is annotated with its minimum time limit m, its weight w and its number of repetitions r. The first pass will produce, from all subpaths starting at each node n, the maximum total minimum time limit $M_m(n)$ and the maximal constraints $P'_S(n)$.

First, $M_m(n)$ and $P'_S(n)$ are precomputed over layer 0:

- $M_m(\text{CO}) = 0$, $P'_S(\text{CO}) = \{(0, 1)\}$.
- $M_m(\text{PS}) = 0$, $P'_S(\text{PP}) = \{(0, 6)\}$ (since PS has $r = 5$).
- $M_m(\text{DS}) = 0$, $P'_S(\text{DS}) = \{(0, 7)\}$.
- $M_m(\text{EO}) = 0.4$, $P'_S(\text{EO}) = \{(0.4, 7)\}$.



(a) Layer 0



(b) Layer 1

Figure 4: Execution traces for the graph-based time limit algorithm on layers 0 and 1 of the running example in Figure 3.

Using the maximal constraint $(0.4, 7)$, we know that the slack per unit of weight on the strictest path will be equal to $\frac{1-0.4}{7} \approx 0.086$ seconds. After that, the algorithm sends the available time ($L = 1$ s) into the InitialNode and then into EO. EO takes 0.4 s and sends the remaining 0.6 s through the DecisionNode to DS, which takes 0.086 s and sends the remaining 0.514 s to PS. PS takes 0.086 s for each of its 5 repetitions and sends the remaining 0.086 s to CO through the JoinNode.

The algorithm then continues over the contents of "Process Segments", using its time limit as the new global time limit. In the lower part of Figure 4, we can see that layer 1 is comprised of the contents of "Process Segments" and its global time limit $L = 0.086$, as computed in layer 0. $M_m(n)$ and $P'_S(n)$ are computed in the same way as before. SO takes the full 0.086 s, being the only ExecutableNode in its path. CI and PP take half of L each: 0.043 s.

As expected, both algorithms return the same results. However, in this simple example we are not able to appreciate the exponential explosion underlying the LP algorithm. In the next section we will perform a thorough evaluation of the algorithms to analise their performance.

4. Evaluation of the algorithms

In the previous sections we have described the models used by the algorithms and the algorithms themselves. We will devote this section to evaluating their performance on our reference implementations.

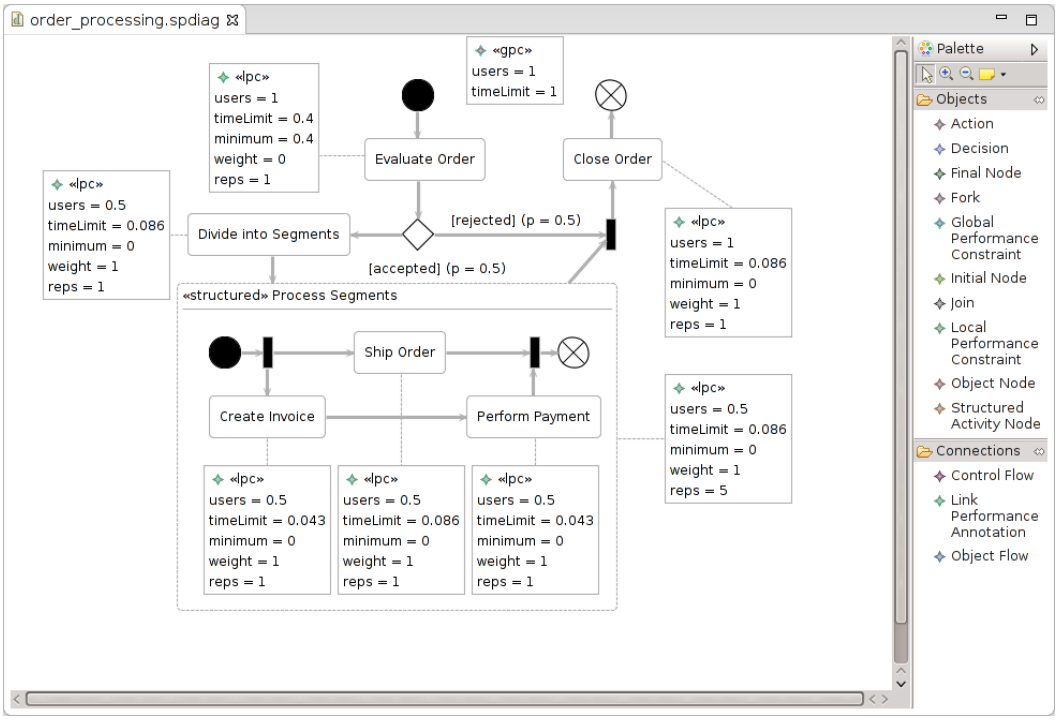


Figure 5: Screenshot of the Eclipse-based model editor.

4.1. Implementation

We have implemented the models and algorithms in this paper as a set of Eclipse plug-ins. The source code uses a mix of several task-specific model handling languages from the Epsilon family [13].

The models corresponding to our web service compositions can be created using a graph-oriented graphical editor. To ensure that the algorithms can be applied, our tool is able to validate the models automatically, providing error and warning markers and "quick fixes" to assist the user in correcting invalid models. The algorithms can be launched from the contextual menu of our graphical editor (see Figure 5).

The throughput computation algorithm and the graph-based time limit computation algorithm have been tested on a set of manually designed test cases, using the EUnit framework [14] included in Epsilon[4]. In addition, the graph-based time limit computation algorithm has been tested with a large set of automatically generated models, ensuring that its results were equivalent to those of the linear programming-based algorithm (barring negligible differences due to floating-point propagation errors).

4.2. Theoretical performance

Before evaluating the empirical performance of our algorithms, we will compute some upper bounds on their execution costs from their definitions. In this section we will also define several graph shapes for our models. We will use these shapes throughout the theoretical and empirical performance analyses of the time limit computation algorithms.

[4] http://www.eclipse.org/epsilon/doc/eunit

4.2.1. Throughput computation

The performance of this algorithm is quite simple to analyse. If we visit the nodes in topological order, then the algorithm will only need to visit each node once. For each node, the algorithm will compute a constant-time expression on every incoming edge (multiplications for conditional edges, scalar comparisons for JoinNodes and sums for MergeNodes). These operations are executed with finite precision and are, thus, in $\Theta(1)$.

Consider a model with n nodes and e edges. Since the algorithm visits each node and edge exactly once and spends a constant amount of time on each of them, the algorithm will require $O(n+e)$ operations. If the underlying graph is dense, then $e \in \Theta(n^2)$ and $O(n+e)$ becomes simply $O(n^2)$.

4.2.2. LP-based time limit computation

The time limit computation algorithms are harder to analyse than the throughput computation algorithm, as their performance depends on the structure of the underlying graph. For this reason, we will define three graph shapes for our models.

1. *Sequence* models consist of an InitialNode followed by one or more Action nodes in sequence, with a FinalNode at the end. Sequence models have 1 path each. A graphical representation is shown in Figure 6(a).
2. *Decision-merge* models have an InitialNode, followed by a sequence of f *levels*. Each level has a DecisionNode with two branches with a single Action, merged before the next level. The model has $2+4\cdot f$ nodes and $1+5\cdot f$ edges in total, and there are 2^f paths from the InitialNode to the FinalNode.

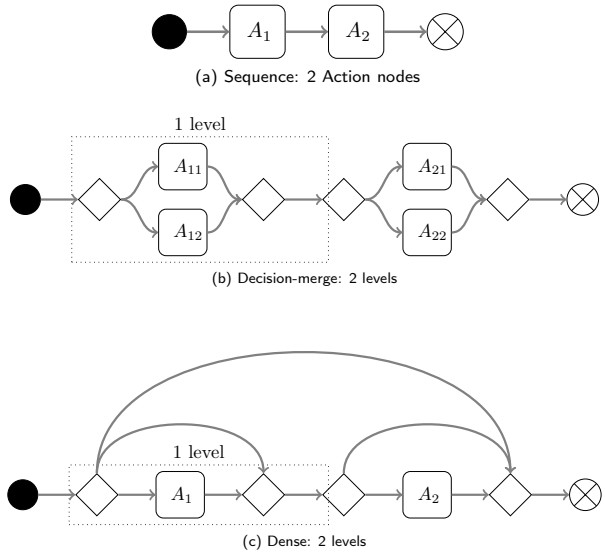


Figure 6: Graph shapes used in the performance analyses

Component	O(RC)	O(IGT)	O(TT)
Objective function		$n \cdot p$	$n \cdot p$
R1: $S_w(n) \geq 0$	n	1	n
R2: L per path	p	n	$n \cdot p$
R3: minimum S_w	n	$n \cdot p$	$n^2 \cdot p$
R4: same paths	n^2	p	$n^2 \cdot p$
Total	$n^2 + p$		$n^2 \cdot p$

Table 1
Restriction Counts, Individual Generation Time and Total Time for the LP-based algorithm, by component.

Shape	O(N)	O(P)	O(RC)	O(TT)
Sequence, n nodes	n	1	n^2	n^2
Decision-merge, f levels	f	2^f	2^f	$n^2 \cdot 2^f$
Dense, f levels	f	$(f+1)!$	$(f+1)!$	$n^2 \cdot (f+1)!$

Table 2
Nodes, Paths, Restriction Counts and Total Time for the LP-based algorithm, by graph shape, using the results from Table 1.

3. *Dense* models have an InitialNode, followed by a sequence of f *levels*, like the decision-merge models. However, the structure of each level is different: a DecisionNode picks between running an Action or jumping to any of the following levels. The model has $2 + 3 \cdot f$ nodes and $1 + 3 \cdot f + \sum_{i=1}^{f} i$ edges. Finally, there are $(f+1)!$ paths from the InitialNode to the FinalNode. These models have many more edges and paths than the decision-merge models: in fact, they represent the densest graph that we can build with this combination of nodes.

With these shapes in mind, let us go back to analysing the performance of the linear programming-based time limit computation algorithm. Since there are many methods for solving LP problems (some of them are very efficient on the average case), we will focus instead on the size of the resulting LP problem. If we have a model with n nodes and p paths from the InitialNode to the FinalNode, then the LP problem will have n variables and will consist of an objective function which can be generated in $O(n \cdot p)$ time and the following restrictions:

- The slack per unit of weight must be positive (R1): one per node. Each restriction can be generated in $O(1)$ time.
- The global time limit must be honoured (R2): one per path, generated in $O(n)$ time by traversing every node in the path.
- The lower bound on the slack per unit of weight for every node (R3): one per node, generated in $O(n \cdot p)$ time by traversing every path and computing $m(p)$ and $w(p)$ for it.
- The slack per unit of weight must be equal for all (m, n) pairs of nodes in the same paths (R4): one for every such pair, generated or discarded in $O(p)$ time by comparing $P_C(m)$ and $P_C(n)$.

These results are aggregated in Table 1. We can conclude that we will generate $O(n^2 + p)$ restrictions in $O(n^2 \cdot p)$ operations. Table 2 applies these results to each of the three graph shapes in Figure 6. We can see that the rapidly increasing number of paths in the model is the main limiting factor for applying the algorithm to more complex models.

4.2.3. Graph-based time limit computation

Analysing the graph-based time limit computation algorithm is harder than analysing the LP-based algorithm. Actually, most models found in practice are sparse, not dense: the number of edges is linear in the number of nodes. Within these sparse models, the worst scenario arises when diamond-like structures (decision-merge or fork-join) chain together in long sequences. Then, the number of paths grows exponentially with the number of nodes. Regarding complexity, graph-based algorithms traversing paths to compute time limits are heavily impacted by an exponential increase in the number of paths. Thus, decision-merge models represent a worst case for sparse models.

Consequently, we will limit our analysis in this case to decision-merge models. Let us analyse the algorithm in the worst case by parts:

- Computing $M_m(n)$ in advance for each node always takes $O(1) \cdot O(n) = O(n)$ operations, as it requires evaluating an arithmetic expression over the $O(1)$ incoming edges of each of the n nodes.
- Computing $P'_S(n)$ in advance for each node is actually the most expensive part of the algorithm: in the worst case, $O(2^f)$ paths need to be considered at every node and selecting the strictest ones takes $O(4^f)$ operations per node and $O(n \cdot 4^f)$ in total.

- The last step depends on the number of elements of $P'_S(n)$ for each node n in the graph: in the worst case, $|P'_S(n)| = |P_S(n)|$ (no paths have been removed) for every node n and $O(n \cdot 2^f)$ operations are required.

Joining the three parts of the algorithm yields a time of $O(n \cdot 4^f)$ operations in the worst case for a decision-merge model. The absolute worst case is very expensive. However, it is also very rare, as we will see in Section 4.3.4.

4.3. Empirical performance

In the previous section we studied the definitions of the algorithms to derive several upper bounds for their execution times. We concluded that the throughput algorithm required $O(n^2)$ operations, the LP-based time limit computation algorithm required $O(n^2 \cdot 2^f)$ operations for decision-merge models with f levels and the graph-based time limit computation algorithm required $O(n \cdot 4^f)$ operations for the same decision-merge models.

However, we also concluded that these were very loose upper bounds, due to limitations in our analysis. For this reason, in this section we will discuss the results of several experiments based on the actual execution of the algorithms on a set of automatically generated models.[5] We will show that the graph-based algorithm requires much less time to run in practice than the LP-based algorithm, and that it does not show the exponential growth which would be expected from the previous upper bound. This is because the worst case becomes harder to find as models become more complex, as we show at the end of this section.

The performance tests were run in an inexpensive laptop computer, based on an Intel CPU at 1.73 GHz with 4 GiB DDR3 RAM, using Eclipse and Epsilon. Wall clock times were measured using the facilities provided by the Java Virtual Machine (JVM), ensuring other processes remained idle during the tests. The studies in Sections 4.3.1, 4.3.2 and 4.3.3 were conducted using an Eclipse plug-in that we built for this study. We ported parts of the graph-based time limit algorithm to C++ for the study in Section 4.3.4.

4.3.1. Throughput computation

Figure 7 shows the average execution times corresponding to the throughput computation algorithm for the three graph shapes described in Figure 6. The generated models had between 0 and 50 actions: decision-merge models had between 0 and 25 levels, and dense models had between 0 and 50 levels.

The algorithm shows the expected level of performance for decision-merge and sequence models. A model with 50 actions in a sequence model only takes 0.04 s, and a decision-merge model with 50 actions takes 0.07 s. A dense model with 50 actions takes somewhat longer, requiring 0.46 s. This confirms our previous $O(n + e)$ bound for the algorithm: sparse graphs exhibit linear growth in the number of nodes and edges, while dense graphs show quadratic growth in the number of nodes.

[5] The models were produced using purpose-built Java code that generated models with the desired shapes and added random performance annotations.

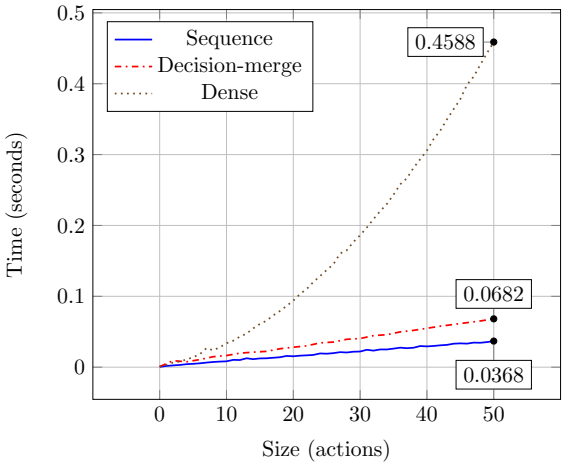


Figure 7: Average execution times over 10 runs for the throughput computation algorithm, by graph shape and size. X axis step was 1 for dense/sequence graphs, and 2 for decision-merge graphs.

4.3.2. Comparison between the time limit algorithms

Figure 8 compares the times required by the LP-based and graph-based time limit computation algorithms for the three graph shapes listed in Figure 6. All Action nodes were annotated with uniform random minimum times (between 0 and $0.5L$) and weights (up to 10). Execution times are represented in a base-10 logarithmic scale. In this case, because of the rapid increase in cost of the LP-based algorithm, we had to limit the maximum size of the models. Decision-merge models were limited to 20 actions (10 levels), and dense models were limited to 20 actions (20 levels).

From the results for the sequence models, it may seem that the graph-based algorithm may be better than the LP-based algorithm even for small models. However, it is important to remark that the LP-based algorithm needs to invoke an external program (the LP solver), while the graph-based algorithm runs entirely within the JVM. Using a Java-based LP solver could make the LP algorithm faster for small inputs, but it would not change the result for larger ones, as problem sizes grow very fast for the LP-based algorithm.

During these tests, we checked that the results produced by both algorithms were the same, except for a minimal error margin (0.1%) due to floating-point rounding and error propagation. More formally, if r_l and r_g were the results of the LP-based and graph-based algorithm for the same input model, we verified that $\frac{|r_l - r_g|}{\max\{r_l, r_g\}} \leq 0.001$.

4.3.3. Influence of annotations on the graph-based time limit algorithm

After concluding that the graph-based time limit algorithm was clearly superior to the LP-based algorithm, we decided to study the effect of the manual performance annotations on the graph-based time limit algorithm. Depending on

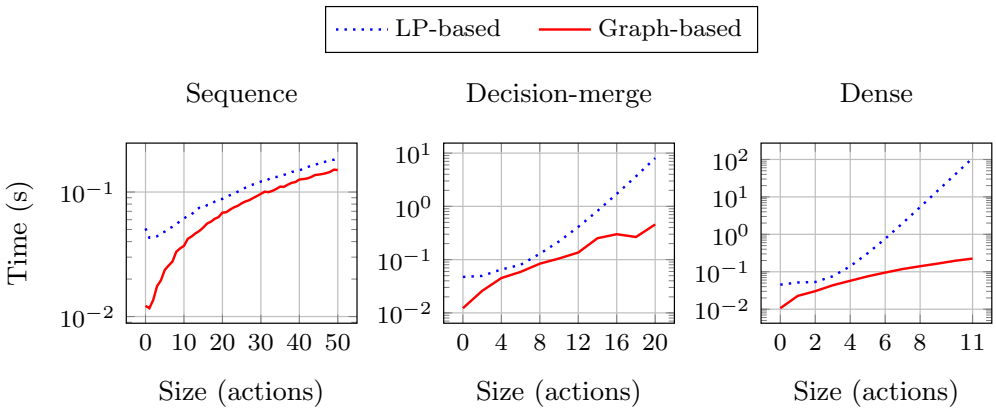


Figure 8: Average execution times over 10 runs for the time limit computation algorithms, by graph shape and size, with $L = 100\,\text{s}$. X axis step was 1 for dense/sequence graphs, and 2 for decision-merge graphs.

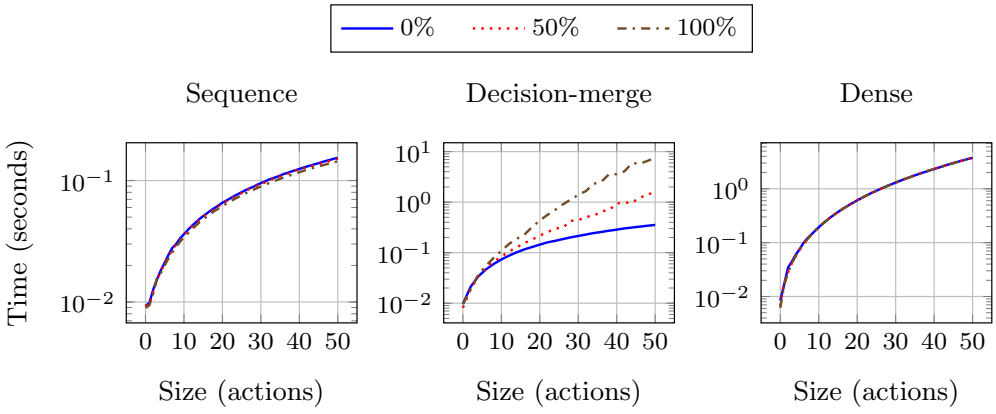


Figure 9: Average execution times over 100 runs for the graph-based time limit computation algorithm. X axis step was 1 for dense/sequence graphs, and 2 for decision-merge graphs.

the actual values used in this annotation, the algorithm may be unable to discard some paths, reducing the effectiveness of its optimisations over the LP-based algorithm.

To study the impact of this issue on performance, we measured the average time required by the graph-based time limit algorithm over 100 runs for each graph shape and size. We annotated either 0%, 50% or 100% of all ExecutableNodes with randomly generated performance annotations. Results are shown in Figure 9. We present the results by graph shape, graph size and percentage of Activity nodes with uniformly random performance annotations. We have set L to 100 s, minimum time limits ranged between 0 and $0.5L$ and weights ranged between 0 and 10. Finally, please notice that execution times are represented in a decimal logarithmic scale.

It is interesting to mention that only decision-merge models show notable differences between annotating 0%, 50% or 100% of all ExecutableNodes. This is obvious for sequence models, which only have one path, but it might be surprising for dense models, which have $f!$ paths for a model with f levels. This is because of Equation (6) and the structure of our dense models. If we need to choose between a subpath (m, w) that does not run a certain node with minimum time m_a and weight w_a, and a subpath $(m+m_a, w+w_a)$ that does, by Equation (6) we will discard (m, w) and only keep $(m+m_a, w+w_a)$. For this reason, at each DecisionNode we will only need to consider one subpath to find the strictest path from the InitialNode to the FinalNode. The observed faster-than-linear growth for dense models can be attributed to the need to traverse all $O(f^2)$ edges to precompute $M_m(n)$ for each node.

Going back to decision-merge models, we can see that annotating all ExecutableNodes with custom local performance annotations is more expensive than always using the default zero minimum time and unit weight. This can also be explained through Equation (6): when using the default performance annotations, it is always the case that $a = c = 0$ and Equation (6) can be simplified into $b \leq d$, which is a total order. In that case, we can remove many more paths and the optimisations are much more effective. Otherwise, some paths may not be comparable (as $\preceq_{s(L)}$ is a partial order) and execution costs will increase. Nevertheless, even when all ExecutableNodes are annotated, execution times do not show the exponential growth of the LP-based algorithm.

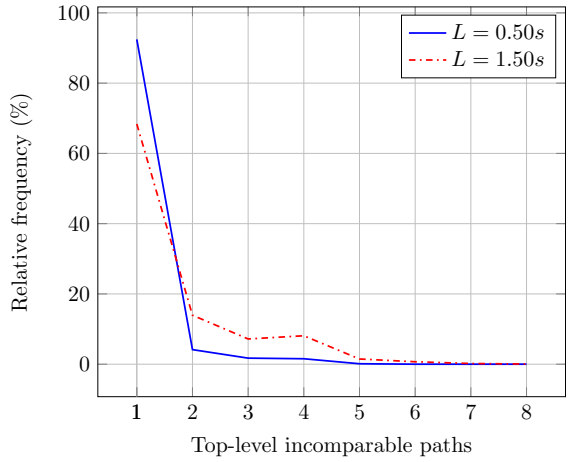


Figure 10: Percentages of sampled 3-level decision-merge models with a certain number of top-level incomparable paths, 4 incomparable paths at the second level and 2 incomparable paths at the last level, by values of the global time limit L.

4.3.4. Worst case of the graph-based time limit computation algorithm

So far, we have shown that removing redundant paths is effective in avoiding the exponential growth in cost that affected the LP-based time limit computation algorithm. However, its effectiveness depends on the values of the annotations used in the model. Taking a closer look at Equation (6), we can see that it depends on the relative magnitude of the minimum time limits and weights with regards to the global time limit L. The left operand of $(b-d)\cdot L < b\cdot c - a\cdot d$, part of Equation(6), grows as L increases and reduces the number of comparable pairs of paths.

We performed an additional study to clarify how common the absolute worst case was and study its relationship with L. We sampled with $L = 0.5\,\text{s}$ and $L = 1.5\,\text{s}$ the space of all decision-merge models with 3 levels which contained a 2-level decision-merge with 4 incomparable paths. Minimum times for the ExecutableNodes ranged from 0 to $\min\{L, 1\}$, in steps of 0.1 s. Weights ranged from 0 to 10, in steps of 1 unit. Inconsistent models were discarded. For each model, we measured the number of incomparable paths at the initial node ("top-level paths"): in a 3-level decision-merge model, there can be between 1 and $2^3 = 8$ such paths.

Evaluating 1.99×10^6 decision-merge activities for $L = 0.5\,\text{s}$ and 7.16×10^9 for $L = 1.5\,\text{s}$ produced the results in Figure 10. For $L = 0.5\,\text{s}$, less than 10% of these models had more than 1 incomparable path. With $L = 1.5\,\text{s}$, less than 20%s of the models had more than 2 incomparable paths.

Furthermore, it is interesting to remark that for $L = 1.5\,\text{s}$, while 31.8% of all 1-level decision-merge models were in the worst case, only 2.5% 2-level decision-merge models were in the worst case. With 3 levels, it was further reduced to 0.05%. This suggests that the absolute worst case becomes harder to find as models become more complex, explaining why average times did not grow exponentially in Figure 9. Additionally, it indicates that the worst case becomes more common as L grows in relation to the values used in the annotations.

5. Threats to the validity and usability of the framework

In this section we briefly review some potential issues that might hinder the applicability of our proposal. Please see Section 7 for future work on some of these issues.

First of all, our framework considers a static workflow. This allows us to accurately estimate some parameters from the model, e.g. bounds on the number of iterations that might be performed. However, if our framework is to be applied to dynamic compositions, then it should be expected that the accuracy will decrease, as there are some parameters that cannot be statically inferred. As a mitigation, our algorithms could be run in the background periodically and fed with information from the execution of services. Of course, this is a delicate balance and it is not easy to determine in advance how often the estimation algorithms should be executed, so that they do not drain too many resources from the execution environment and degrade the performance of the composition. That would certainly improve the results, but precise estimation in the presence of dynamic compositions would likely require a different approach.

Second, we currently rely on the user to provide the relative measure of the computational intensity of each node, that is, the *node weight*. This assumes that our user is a domain expert or, at least, she has enough knowledge of the services involved. Although weights are optional parameters in our framework, it would be desirable to have a (semi-)automatic process to set them to plausible values. In absence of service provider information on expected execution times, and given the likely variability of execution times for the same service on different inputs, a statistical or machine learning (ML) approach would probably be the most appropriate.

Third, in their current form, the algorithms do not take into account the fact that the same web service may be invoked several times from different points in a composition or in different composition. In fact, in BPMN, there can be *call activities* invoking a global process or a global task from different points and it would be useful that our framework could cope with these situations. One approach is changing the underlying model into labelled graphs, where the edges would be labelled to keep track of the inbound and outbound activity nodes during the traversal of an execution node. However, adapting the algorithms to this model may not be trivial, particularly the iterative version of the graph-based algorithm, which is our most efficient version developed so far. In this line, the LP-based time limit inference algorithm could be extended by including new constraints in the linear programs. However, the graph-based algorithm would not accommodate those additional constraints, but it could assign the strictest time limit inferred among all its occurrences. We discuss this in more detail in Section 7.

Fourth, we have considered dense decision-merge models up to 50 levels and this might be, in principle, a lim-

itation in the experimental justification of our framework. However, this number of levels allowed us to consistently obtain execution times in the order of seconds with mainstream computers. In our experience, models with less than this number of levels are quite common. In fact, we think that creating huge models instead of structuring them hierarchically may not be the best modelling practice. Over a certain size, models should be decomposed into manageable ones that can be easier to understand both by the modeller and the stakeholders. Moreover, if we couple the model size with the complexity of the corresponding algorithms and the experimental execution times obtained for the test data under consideration, we expect that scalability is guaranteed for models bigger than those present in our test data.

Finally, our framework relies on fairness to allocate slack and this might lead to potential inaccuracy in estimates. There are several techniques to cope with this, from sensitivity analysis to ML, that can be considered for future work. Also related to this, it is not easy to estimate the number of iterations in a process loop. In this case, we take into account that, usually, underestimation is more dangerous than overestimation. There are often two ways to get more precise estimates: expert reviews and sampling profiling. We discuss this in more detail in Section 7.

6. Related work

Obtaining the desired level of performance has been a regular concern since the development of the first computer systems, as shown by an early survey [15]. There are basically two approaches: evaluating a model of a system (known as *performance engineering*), or measuring the performance of an implemented system (*performance testing*). These approaches are complementary: using analytic models reduces the risk of implementing an inefficient software architecture, which is expensive to rework [16]. When the system is implemented, measuring its performance is more accurate, and can detect not only design issues, but also bad coding practices and unexpected workloads or platform issues [17]. Some authors have proposed overloading "performance engineering" to point to both model- and measuring-based approaches [18].

Throughout this section, we will briefly review some of the works that are more closely related to ours. We will first visit the existing notations, mainly focusing on well-established standards. Later on, we will discuss some of the performance analysis algorithms that are more closely related to ours. Finally, we will point to some works in service selection.

6.1. Notations

Widespread adoption of UML as a *de facto* standard notation has prompted researchers to derive their analytic models from UML models, first with custom annotations and later consolidating on standard extensions to UML, such as the Schedulability, Performability and Time (SPT) profile [19]. When UML 2.0 was published, OMG saw the need to update the SPT profile and harmonise it with other new concepts. This resulted in the MARTE (Modelling and Analysis of Real-Time and Embedded Systems) profile [20]. The MARTE profile defines a general framework for describing QoS aspects. Our work uses models based on UML activity diagrams with custom extensions to simplify the discussion away from the technical details of the MARTE notation and its VSL sublanguage.

Performance engineering usually involves building a simplified representation (a *model*) with information on each part of the system, from which the expected global performance is derived. There is a large number of works dealing with model-based testing, i.e. "the automatable derivation of concrete test cases from abstract formal models, and their execution" [21, 22]. Although most of the work considers functional testing, model-based testing can be used to analyse performance aspects in distributed environments such as the cloud [23]. Unfortunately, the notations and methods usually considered in model-based testing are very far from the focus of this article: we consider a UML-like notation to infer quantities of interest.

Queueing networks were among the first formalisms used for performance engineering. A classical example of this formalism is the PRIMA-UML methodology [24]. While our work is focused on producing local performance requirements, PRIMA-UML is oriented towards validating the early design of the system using the EQNM. Therefore, we believe that both works complement each other.

Currently, the most common formalisms in performance engineering are layered queuing networks [25], stochastic Petri Nets [26] and stochastic Process Algebras [27]. These formalisms are backed by in-depth research and the last two have solid mathematical foundations in Markov chain theory. However, they introduce an additional layer of complexity which might discourage some users from applying these techniques. This burden might be ameliorated by using recent work that shows progress in *learning* some of these models [28].

Recently, we have considered *provenance graphs* to log changes to a run-time model [29]. However, the notation is far from ours and the focus is on showing how provenance graphs can support the validation of systems.

6.2. Algorithms

Our algorithms compute local performance constraints for the various pieces of software that participate in a web service composition. Web Service compositions are modelled using UML activity diagrams, which define a workflow from an initial node to a set of final nodes.

There are many other works on performance estimation based on workflows. However, to the best of our knowledge, they focus on computing the global performance of the workflow from a set of local annotations. Our approach works in the opposite direction: it estimates the local performance which should be required of the composed web services from the global performance constraint set by the user.

The SWR algorithm [30] computes the expected QoS of the workflow given the QoS of the services involved. In a

sense, its goal is the opposite to ours: our algorithms compute performance requirements for the services from their compositions, while this work computes the expected QoS of the entire composition from the known QoS of its services by iteratively reducing its graph model to a single task.

Moving beyond workflows, and into entire software systems, MARTE was extended with the *Dependability and Analysis Modeling sub-profile* [31]. Our work also handles time limits, but our focus is different: we help the user "fill in the blanks" using the available partial information. It is possible to generate intermediate performance models from a set of UML diagrams annotated with the MARTE profile, describing a service-oriented architecture [32]. In this approach, activity diagrams model the workflows, component diagrams represent the architecture and sequence diagrams detail the behaviour of each action in the workflows. Our approach does not model the resources used by the system: we assume performance tests will be run in an environment that mimics the production environment.

Finally, Integer Linear Programming (ILP) has been previously used in the context of WS-BPEL compositions. In particular, we have used it to reduce the size of test suites [33]. ILP has been also used to select which services should be used in a web service composition [34]. Nevertheless, the aim of these approaches is quite different to ours: while they focus on selecting test cases/services from a pool of candidates, we intend to provide a first estimate of performance requirements during early analysis and design.

6.3. Service selection

Although in this paper we are mainly interested on performance, a related problem is to select services according to certain criteria [35, 36], which can be done using different approaches. For example, a global optimal selection strategy can be implemented by using dynamic programming [37] while other approaches consider algorithms such that QoS constraints guide the selection of services [38, 39]

7. Conclusions and future work

Modern software architecture design has seen a rise in the application of the SOA paradigm to the implementation of large scale software applications, in particular, web service compositions integrating both in-house and third-party services. One of the main issues when designing these compositions is the fair estimation of their required performance in a context where QoS information may be incomplete or even missing. Along these lines, this approach introduces dependencies on the performance of the integrated services, which instead determines the overall performance of the whole composition. Although QoS information of an in-house service may be available through its SLA or inferred from statistical data (historical data or data provided by monitoring), reliable performance-related information can be more difficult to obtain in the case of third-party services. Besides, this information is often unavailable when designing new service compositions from the ground up.

The expected performance of web services may be hard to assess and it is easy to wrongly estimate the related QoS information that is really required by the client without conducting proper stress testing, which is not always possible. In fact, when using third-party web services there may be not enough information available to make anything beyond an educated guess. If the guess is wrong, revising all the estimations and how they affect the web service composition integrating these services soon becomes a long, tedious, and error-prone process.

In this paper, we have presented three algorithms for computing the local performance requirements of services from the global performance requirements of the service composition that integrates them. The composition is modelled in a workflow-based graphical language, formally defined by a metamodel based on UML activity diagrams with performance annotations. This language is simple enough to allow for translations from other workflow-based languages like BPMN 2.0.

Each of the three algorithms receives a model of the service composition annotated with the required throughput, that is, the peak number of concurrent users or requests per second that should be processed, and the global time limit or maximum execution time. The first algorithm is based on topological sorting and computes the throughput (requests per second) corresponding to each service. The second algorithm is based on linear programming (LP) and computes the time limit for each service, which provides an upper bound of the time available to process each request. The time limits assigned are fair in a precise sense. The third algorithm is a time limit computation algorithm too and yields the same results that the second algorithm. This graph-based algorithm it is more efficient than its LP counterpart.

The time limit computation algorithms can combine the global performance annotation with optional local performance annotations that model the partial knowledge of the developer about the expected performance and computational cost of the services. The algorithms have been implemented using several languages from the Epsilon family [13], a set of high-level interpreted languages running on top of the JVM and specialised for working with models. The LP-based algorithm uses the GLPK linear programming solver under the hood.

Regarding performance, the first algorithm is $O(n + e)$, that is, linear in the size of the model graph. The performance of the second and third algorithms can be analysed for a decision-merge model with f levels. The parameter f is small in practice, as it represents the number of nested activities in a model. In the context of service compositions each nested activity is typically a service composition used as a service inside another composition. The generation of the linear program with the LP-based algorithm requires $O(n^2 \cdot 2^f)$ operations and the linear program has to be solved with an external LP solver. An initial analysis of the optimised graph-based algorithm reports a theoretical upper bound of $O(n \cdot 4^f)$ operations. When f is a small constant their fixed-parameter complexity is polynomial in the

worst-case. This is precisely the case with well-structured models. Our experiments show good performance for reasonably large and complex graphs representing service compositions.

Concerning future work, we envision several research lines and extensions to our proposal.

First, there are some intrinsic limitations to static methods when the dynamics of a service composition is taken into account. This is especially true for parameter estimation. That is why domain information is so valuable in this context. This information can be obtained from different sources: domain experts, historical data, statistical inference, machine learning (ML), etc. However, it is not always the case that these sources are available, particularly when the composition enters production for the first time. The same happens at specification or design time, where these static methods are also useful even when there is still nothing to execute. Sampling profiling is another technique that can be used to obtain precise estimates, but it requires low-level access to the execution environment, which is not always possible. We think that the following approach could be promising in the particular case of iteration bounds: an initial bound is guessed beforehand, and the composition is then monitored during its execution; if the bound is exceeded, this will trigger the execution of the algorithms to recompute the throughput and time limits.

Second, in our opinion, a similar hybrid technique combining static estimations and dynamic corrections could be applied to other parameters too. Besides, the possibility of implementing the algorithms in a natively compiled language could provide the efficiency boost to enable the execution of the algorithms computing the performance requirements on demand. This is important for highly dynamic compositions, where web services are hot-swapped between different providers, like in metasearch and global booking services.

Third, we are considering using ML techniques to estimate performance-related QoS information. There exists an initial step in this direction [40], where the reliability of web services is estimated by ML. As discussed previously, there are scenarios in which this approach does not apply, but it could prove a useful addition to our toolkit in others.

Fourth, we would also like to extend our framework so that we can reuse execution nodes, providing a functionality similar to *call activities* in BPMN models. For example, different call activities can invoke the same task, such as a web service. We think that a promising approach would imply a model transformation splitting paths corresponding to different invocations and replicating the task in each new path. Then, the graph-based algorithm has to be modified to keep track of these artificial replicas that, indeed, represent the original task. By construction, the graph-based algorithm assigns time limits to each node the first time the node is traversed, which is always done on the strictest path. When an execution node is assigned a time limit, this will be also assigned to each of its replicas, once and for all. All the remaining steps in the algorithm have now to be aware of the replicas and the time limits assigned to them. Of course, some of these changes are not trivial to implement and the impact on complexity is also to be assessed.

Fifth, as some of the parameters needed to execute the algorithms will necessarily come from estimations, business or domain-specific knowledge, it would be interesting to place the whole approach in the context of the current DevOps approach, where a dashboard is used to fine-tune the algorithm by controlling a few parameters, particularly the number of execution replicas of each service. Let us illustrate this with a simple example. Assume that our algorithm determines that, in order to meet a given total time limit, it is enough that a given service (with n replicas) finishes within a time limit that is less strict than the times it is currently incurring. In this case, we could reduce n, while checking average response times, so that we can reduce the bill charged by the provider. This saving could be used to increase the number of execution replicas of another service that it is requiring more time than initially thought and is therefore struggling to meet its local time limit.

Finally, we would like to adapt our algorithms to estimate the amount of time needed to test systems with distributed components whose specifications include temporal information [41, 42, 43].

Acknowledgements

We would like to thank the anonymous reviewers of this paper for their careful reading and useful suggestions that have improved the quality of the paper.

This work has been supported by the Spanish Ministry of Science and Innovation through the following projects: FAME (RTI2018-093608-B-C31 and RTI2018-093608-B-C33), AwESOMe (PID2021-122215NB-C31 and PID2021-122215NB-C33), and SEBASENet (RED2018-102472-T), which were co-funded by the European Regional Development Fund. Partial support was also provided through the Region of Madrid project FORTE-CM (S2018/TCS-4314), co-funded by EIE Funds of the European Union.

References

[1] 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, accessed on 3 May 2022 (2007).

[2] Object Management Group, Business Process Modeling Notation 2.0.2, http://www.omg.org/spec/BPMN/2.0.2, accessed on 3 May 2022 (2014).

[3] K. Falkner, C. Szabo, V. Chiprianov, G. Puddy, M. Rieckmann, D. Fraser, C. Aston, Model-driven performance prediction of systems of systems, Software and Systems Modeling 17 (2) (2018) 415–441. doi:10.1007/s10270-016-0547-8.

[4] A. Strunk, QoS-aware service composition: A survey, in: 8th IEEE European Conference on Web Services, ECOWS'10, IEEE Computer Society, 2010, pp. 67–74. doi:10.1109/ECOWS.2010.16.

[5] V. Hayyolalam, A. A. Pourhaji Kazem, A systematic literature review on QoS-aware service composition and selection in cloud environment, Journal of Network and Computer Applications 110 (2018) 52–74. doi:10.1016/j.jnca.2018.03.003.

[6] A. Sullivan, M. Z. Catur Candra, Web service recommendation system using history and quality of service, in: 6th Int. Conf. on Data

and Software Engineering, ICoDSE'19, IEEE, 2019, pp. 1–6. doi:10.1109/ICoDSE48700.2019.9092755.

[7] K. A. Botangen, J. Yu, Q. Z. Sheng, Y. Han, S. Yongchareon, Geographic-aware collaborative filtering for web service recommendation, Expert Systems with Applications 151 (2020) 113347. doi:10.1016/j.eswa.2020.113347.

[8] V. P. Singh, M. K. Pandey, P. S. Singh, S. Karthikeyan, Neural net time series forecasting framework for time-aware web services recommendation, in: 3rd Int. Conf. on Computing and Network Communications, CoCoNet'19, Procedia Computer Science 171, Elsevier, 2020, pp. 1313 – 1322. doi:10.1016/j.procs.2020.04.140.

[9] Eclipse Foundation, ATL – a model transformation technology, https://www.eclipse.org/atl, accessed on 3 May 2022 (2018).

[10] Eclipse Foundation, The Epsilon Transformation Language (ETL), https://www.eclipse.org/epsilon/doc/etl, accessed on 3 May 2022 (2022).

[11] R. Tarjan, Depth-first search and linear graph algorithms, SIAM Journal on Computing 1 (2) (1972) 146–160. doi:10.1137/0201010.

[12] R. Fourer, D. M. Gay, B. W. Kernighan, AMPL: a modeling language for mathematical programming, 2nd Edition, Cengage Learning, 2002.

[13] D. S. Kolovos, L. M. Rose, A. García-Domínguez, R. F. Paige, The Epsilon Book, http://www.eclipse.org/epsilon/doc/book/EpsilonBook.pdf, accessed on 15 March 2022 (2018).

[14] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, I. Medina-Bulo, EUnit: a unit testing framework for model management tasks, in: ACM/IEEE 14th Int. Conf. on Model Driven Engineering Languages and Systems, MODELS'11, LNCS 6981, Springer, 2011, pp. 395–409. doi:10.1007/978-3-642-24485-8_29.

[15] H. Lucas, Performance evaluation and monitoring, ACM Computing Surveys 3 (1971) 79–91. doi:10.1145/356589.356590.

[16] C. U. Smith, Introduction to software performance engineering: Origins and outstanding problems, in: M. Bernardo, J. Hillston (Eds.), 7th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM'07, Springer, 2007, pp. 395–428. doi:10.1007/978-3-540-72522-0_10.

[17] A. Avritzer, E. J. Weyuker, Deriving workloads for performance testing, Software: Practice and Experience 26 (6) (1996) 613–633. doi:10.1002/(SICI)1097-024X(199606)26:6<613::AID-SPE23>3.0.CO;2-5.

[18] M. Woodside, G. Franks, D. Petriu, The future of software performance engineering, in: Proceedings of Future of Software Engineering 2007, IEEE Computer Society, Los Alamitos, CA, USA, 2007, pp. 171–187. doi:10.1109/FOSE.2007.32.

[19] Object Management Group, UML Profile for Schedulability, Performance, and Time (SPTP) 1.1, http://www.omg.org/spec/SPTP/1.1, accessed on 3 May 2022 (2005).

[20] Object Management Group, UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) 1.2, http://www.omg.org/spec/MARTE/1.2, accessed on 3 May 2022 (2019).

[21] R. V. Binder, B. Legeard, A. Kramer, Model-based testing: where does it stand?, Communications of the ACM 58 (2) (2015) 52–56. doi:10.1145/2697399.

[22] A. R. Cavalli, T. Higashino, M. Núñez, A survey on formal active and passive testing with applications to the cloud, Annals of Telecommunications 70 (3-4) (2015) 85–93. doi:10.1007/s12243-015-0457-8.

[23] A. Núñez, P. C. Cañizares, M. Núñez, R. M. Hierons, TEA-Cloud: A formal framework for testing cloud computing systems, IEEE Transactions on Reliability 70 (1) (2021) 261–284. doi:10.1109/TR.2020.3011512.

[24] V. Cortellessa, R. Mirandola, PRIMA-UML: a performance validation incremental methodology on early UML diagrams, Science of Computer Programming 44 (1) (2002) 101–129. doi:10.1016/S0167-6423(02)00033-3.

[25] J. E. Neilson, C. M. Woodside, D. C. Petriu, S. Majumdar, Software bottlenecking in client-server systems and rendezvous networks, IEEE Transactions on Software Engineering 21 (9) (1995) 776–782. doi:doi:10.1109/32.464543.

[26] O. Diallo, J. J. P. C. Rodrigues, M. Sene, Performances evaluation and Petri nets, in: M. S. Obaidat, P. Nicopolitidis, F. Zarai (Eds.), Modeling and Simulation of Computer Networks and Systems, Morgan Kaufmann, 2015, Ch. 11, pp. 313–355. doi:10.1016/B978-0-12-800887-4.00011-0.

[27] J. Hillston, Stochastic Process Algebras and their Markovian semantics, ACM SIGLOG News 5 (2) (2018) 20–35. doi:10.1145/3212019.3212023.

[28] G. Garbi, E. Incerto, M. Tribastone, Learning queuing networks by recurrent neural networks, in: 11th ACM/SPEC International Conference on Performance Engineering, ICPE'20, ACM Press, 2020, pp. 56–66. doi:10.1145/3358960.3379134.

[29] O. Reynolds, A. García-Domínguez, N. Bencomo, Towards automated provenance collection for runtime models to record system history, in: 12th System Analysis and Modelling Conference, SAM'20, ACM Press, 2020, pp. 12–21. doi:10.1145/3419804.3420262.

[30] J. S. Cardoso, A. Sheth, J. Miller, J. Arnold, K. Kochut, Quality of service for workflows and web service processes, Journal of Web Semantics 1 (3) (2004) 281–308. doi:10.1016/j.websem.2004.03.001.

[31] S. Bernardi, J. Merseguer, D. C. Petriu, A dependability profile within MARTE, Software & Systems Modeling 10 (3) (2011) 313–336. doi:10.1007/s10270-009-0128-1.

[32] M. Alhaj, D. C. Petriu, Approach for generating performance models from UML models of SOA systems, in: 20th Conf. of the Center for Advanced Studies on Collaborative Research, CASCON'10, ACM Press, 2010, pp. 268–282. doi:10.1145/1923947.1923975.

[33] F. Palomo-Lozano, A. Estero-Botaro, I. Medina-Bulo, M. Núñez, Test suite minimization for mutation testing of WS-BPEL compositions, in: 20th Annual Conf. on Genetic and Evolutionary Computation, GECCO'18, ACM Press, 2018, pp. 1427–1434. doi:10.1145/3205455.3205533.

[34] D. Ardagna, B. Pernici, Adaptive service composition in flexible processes, IEEE Transactions on Software Engineering 33 (6) (2007) 369–384. doi:10.1109/TSE.2007.1011.

[35] Y.-J. Seo, H.-Y. Jeong, Y.-J. Song, Best web service selection based on the decision making between qos criteria of service, in: L. T. Yang, X. Zhou, W. Zhao, Z. Wu, Y. Zhu, M. Lin (Eds.), Embedded Software and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 408–419. doi:10.1007/11599555_39.

[36] W. Viriyasitavat, Z. Bi, Service selection and workflow composition in modern business processes, Journal of Industrial Information Integration 17 (2020) 100126. doi:https://doi.org/10.1016/j.jii.2020.100126.

[37] Y. Gao, J. Na, B. Zhang, L. Yang, Q. Gong, Optimal web services selection using dynamic programming, in: Proceedings of the 11th IEEE Symposium on Computers and Communications, ISCC'06, IEEE, 2006, pp. 365–370. doi:10.1109/ISCC.2006.116.

[38] T. Yu, Y. Zhang, K.-J. Lin, Efficient algorithms for web services selection with end-to-end QoS constraints, ACM Transactions on the Web 1 (1) (2007) article 6. doi:10.1145/1232722.1232728.

[39] C.-F. Lin, R.-K. Sheu, Y.-S. Chang, S.-M. Yuan, A relaxable service selection algorithm for QoS-based web service composition, Information & Software Technology 53 (12) (2011) 1370–1381. doi:10.1016/j.infsof.2011.06.010.

[40] Y. Song, Web service reliability prediction based on machine learning, Computer Standards & Interfaces 73 (2021) 103466. doi:https://doi.org/10.1016/j.csi.2020.103466.

[41] M. G. Merayo, R. M. Hierons, M. Núñez, Passive testing with asynchronous communications and timestamps, Distributed Computing 31 (5) (2018) 327–342. doi:10.1007/s00446-017-0308-0.

[42] R. Lefticaru, R. M. Hierons, M. Núñez, Implementation relations and testing for cyclic systems with refusals and discrete time, Journal of Systems and Software 170 (2020) 110738. doi:10.1016/j.jss.2020.110738.

[43] G. Ortiz, J. Boubeta-Puig, J. Criado, D. Corral-Plaza, A. G. de Prado, I. Medina-Bulo, L. Iribarne, A microservice architecture for real-time IoT data processing: A reusable web of things approach for smart ports, Computer Standards & Interfaces 81 (2022) 103604. doi:10.1016/j.csi.2021.103604.