An Approach for Model-Driven Design and Generation of Performance Test Cases with UML and MARTE

Antonio García-Domínguez[1], Inmaculada Medina-Bulo[1], and Mariano Marcos-Bárcena[2]

[1] Department of Computer Languages and Systems, University of Cádiz, Cádiz, Spain
[2] Department of Mechanical Engineering and Industrial Design, University of Cádiz, Cádiz, Spain
{antonio.garciadominguez,inmaculada.medina,mariano.marcos}@uca.es
http://neptuno.uca.es/∼agarcia, http://neptuno.uca.es/∼imedina

Abstract. High-quality software needs to meet both functional and non-functional requirements. In some cases, software must accomplish specific performance requirements, but most of the time, only high-level performance requirements are available: it is up to the developer to decide what performance should be expected from each part of the system. In this work, we show several algorithms that infer the required throughput and time limits for each action in an UML activity diagram from a global constraint and some optional local annotations. After studying their theoretical and empirical performance, we propose an approach for generating performance test cases from the activity diagram after it has been implemented as code. Our approach decouples the performance analysis model from the implementation details of the code to be tested.

Keywords: Model-driven engineering, Performance testing, UML, MARTE, Non-functional requirements, Model weaving.

1 Introduction

In addition to functional requirements, software must meet non-functional requirements. Among them, performance plays a major role in shaping the user experience. In some cases, meeting specific performance requirements is critical. This is the case not only in soft and hard real-time systems, but also in service-oriented architectures [13], where Service Level Agreements (SLAs) may have been signed between the provider and the consumer of a service.

For these reasons, there has been considerable work in estimating and measuring the performance of software systems [28]. Estimating the performance of a prospective system usually requires building high-level execution and architecture models and deriving a formalism from them, as in [25,30], among many others. Measuring the performance of a system requires instrumenting it to produce the desired results, instead of building a model. These approaches complement each other: estimations can be performed before the actual system is implemented, while measurements are more accurate.

Measuring the performance of a system can be useful for many purposes: finding performance degradations over time, identifying load patterns over specific time periods

M.J. Escalona, J. Cordeiro, and B. Shishkov (Eds.): ICSOFT 2011, CCIS 303, pp. 136–150, 2013.

and checking if the system is meeting its performance requirements. Obviously, this last use case requires that the performance requirements have been previously defined. However, most of the time, detailed performance requirements are not provided [27]. Developers may have to meet high-level performance requirements without a clear view of what performance is required in each part of the system.

In this work we propose a model-driven approach to deriving the low-level performance requirements of a system from high-level performance requirements. The user creates UML models annotated with a small subset of the MARTE profile [23] and runs our inference algorithms to derive the low-level requirements. After the UML models have been implemented as code, the user can weave the analysis model with an implementation model to generate the concrete performance test cases.

The rest of this paper is structured as follows: in Section 2, we introduce the MARTE profile for UML, describe the subset used in our work and show our running example. Section 3 defines the inference algorithms and outlines some of the optimisations performed. Section 4 is dedicated to analysing the restrictions imposed upon the algorithms and evaluating their performance. Section 5 describes our proposed approach for generating the concrete performance test cases. Section 6 discusses related work. Finally, Section 7 condenses the main points of this paper and lists our future lines of work.

2 The MARTE Profile

UML has been widely adopted as a general purpose modelling language for describing software systems. However, UML itself does not include support for modelling scheduling, performance or time aspects, among other non-functional aspects.

For this reason, the Object Management Group proposed in 2005 the SPT (Schedulability, Performability and Time) profile [21], which extended UML with a set of stereotypes describing scenarios that various analysis techniques could take as inputs. In 2008, OMG proposed the QoS/FT (Quality of Service and Fault Tolerance Characteristics and Mechanisms) profile [22], with a broader scope than SPT and a more flexible approach: users formally defined their own quality of service vocabularies and used them to annotate their models.

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 [23], published in 2009. Like the QoS/FT profile, the MARTE profile defines a general framework for describing quality of service aspects. The MARTE profile uses this framework to define a set of pre-made UML stereotypes, as those in the SPT profile.

In this section, we will introduce the parts of the MARTE profile required for our algorithms and show an example model, using its predefined stereotypes.

2.1 Selected Subset

The MARTE specification provides support for model-based analysis and design of real-time and embedded systems. Among its sub-profiles, we are interested in a subset of the GQAM (Generic Quantitative Analysis Modelling) profile. The GQAM domain

model describes the concepts of the GQAM profile using the generic non-functional property modelling framework in MARTE. The stereotypes from the GQAM profile are prefixed with "Ga" (standing for "generic analysis"), and the non-functional property types from the normative MARTE model library are prefixed with "NFP".

The stereotype and attributes used by our algorithms are:

- «GaScenario»: *hostDemand* is used to model requirements on the CPU time to be used and *throughput* indicates how many requests should be handled per second. *respT* combines both, specifying the maximum response time when handling *throughput* requests per second.
- «GaStep»: *prob* is the probability of traversing a control flow, and *rep* is the number of times the annotated activity is repeated.
- «GaAnalysisContext»: *contextParams* contains a list of context parameters. These are variables which can be used to parametrise the annotations using VSL (Value Specification Language) expressions. VSL is a textual language defined in MARTE.

All the non-functional property types in the normative MARTE library share several traits, as they inherit from *NFP_CommonType*. Values can be specified as literals in the *value* attribute, or as VSL expressions in the *expr* attribute. The source of a requirement (estimated, measured, calculated or required) is described by the *source* attribute.

NFP_CommonType is a VSL tuple type. In this paper we will use the notation `(key1=value1,...,keyN=valueN)` for VSL tuples. For instance, a *NFP_Duration* of 5 milliseconds required by the client is written as `(value=5,unit=ms,source=req)`.

2.2 Usage

Activities must have the «GaScenario» and «GaAnalysisContext» stereotypes. «GaScenario» indicates the expected response time (*respT*) and throughput (*throughput*) for the entire activity. «GaAnalysisContext» only lists the context parameters (*contextParams*) which represent the slack per unit of weight assigned to each action in the activity.

Control flows leaving decision nodes are annotated with the «GaStep» stereotype, specifying the probability (*prob*) of traversing one of the conditional branches. The probabilities are estimated by the user.

Actions are annotated with the «GaStep» stereotype as well. The user must indicate their expected number of repetitions (*rep*) and how the available time is to be distributed among them. *hostDemand* must contain a tuple with a VSL expression matching `M+W*swI`: $M \geq 0$ is its minimum time limit, $W \geq 0$ is its weight and `swI` is its context parameter. The time limit inference algorithm will set `swI` to the slack per unit of weight assigned to that action.

After the algorithms are done, results are fed back into the activity diagram, replacing those from previous runs. Actions are annotated with the inferred time limits in *hostDemand*, and with the inferred throughputs in *throughput*. Context parameters are set to the slack per unit of weight assigned to their actions.

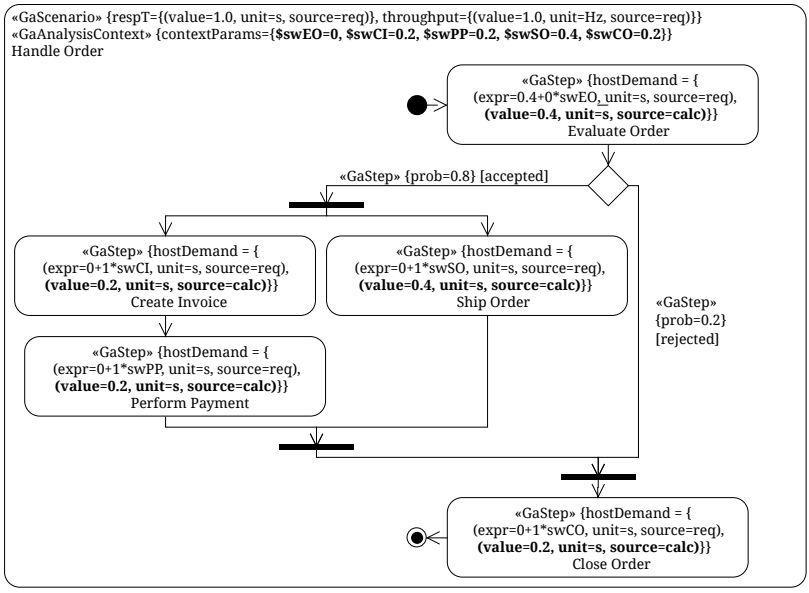


Fig. 1. Running example after inferring time limits

2.3 Running Example

Figure 1 shows the UML activity diagram which we will use as running example for the rest of this paper. Its activity, "Handle Order", describes how to process a specific order: first, the order is evaluated. If rejected, we simply close the order. If accepted, we fork into two execution branches: one creates the shipping order and sends it to the shipping partner, and the other creates the invoice, sends it to the customer and receives the payment. Once both branches are done, the order is closed and we are done.

According to the MARTE annotations, the activity should complete its execution in one second when receiving one request per second. Most of the actions have no minimum time limit and weight equal to 1, except for "Evaluate Order", whose CPU time is fixed by the modeller to 0.4s. All actions are run once, to simplify the discussion. The user has estimated that 80% of all orders are accepted. The annotations in bold have been inferred by our algorithms, and will be described more in depth in Section 3.2.

3 Inference Algorithms

In the previous section, we explained how we used the MARTE profile for our algorithms and described the running example for this paper (Figure 1). In this section we will outline the algorithms themselves. The first algorithm computes the expected throughput of each action, and the second algorithm computes the time limit for each action. They improve upon those in [16].

Both require that activities do not contain cycles, that they only have one initial node, and that all their actions are reachable from it. Let us define some terms:

- $s(e)$ and $g(e)$ are the source and target vertex of the edge e, respectively.
- $i(n)$ and $o(n)$ are the incoming and outgoing edges of the node n, respectively.
- $L > 0$ is the expected response time (the global time limit) of the activity.
- $c(n) = (m(n), w(n)) \in C(L)$ is the *constraint* of the node n, where $m(n)$ is the minimum time limit of n and $w(n)$ is its weight (see Section 2.2). The set of all valid constraints with L as global time limit is $C(L) = \{(m,w) \mid 0 \leq m \leq L, w \geq 0\}$.
- Each path p also has a constraint, $c(p) = (m(p), w(p)) \in C(L)$, with $m(p) = \sum_{n \in p} m(n)$ and $w(p) = \sum_{n \in p} w(n)$.
- A node n is run $R(n) \geq 1$ times (once by default).

3.1 Throughput Inference

We will define T as a function from a node or edge to its expected throughput. For a control flow e, $T(e) = P(e)T(s(e))$, where $P(e)$ is the probability of traversing e.

For a node n, the actual formula depends on its type. For an initial node, $T(n)$ is the expected throughput of the activity. For a join node, $T(n) = \min_{e \in i(n)} T(e)$, since requests in the least performing branch set the pace. For a merge node, $T(n) = \sum_{e \in i(n)} T(e)$, as requests from mutually exclusive branches are reunited. For any other type of node, $T(n) = T(e_1)$, where $e_1 \in i(n)$ is its only incoming edge.

Using these formulas, computing $T(\text{Create Invoice})$ for the example shown in Figure 1 requires walking back to the initial node, finding an edge with a probability of 0.8, no merge nodes and an initial node receiving 1 request/second. Therefore, it would be equal to $pL = 0.8$.

To compute these values efficiently, the expressions are evaluated in a topological traversal of the graph. For each action *a*, *throughput* will contain a single tuple of the form `(value=T(a),unit=Hz,source=calc)`.

3.2 Time Limit Inference

Inferring the time limits of each action inside an activity is considerably more complex than inferring their required throughputs. After more definitions, we will describe the algorithm, and then apply it to the running example in Figure 1.

Preliminaries. The algorithm adds a `(value=t(n), unit=s, source=calc)` tuple to the attribute *hostDemand* of each action node n, where $t(n)$ is its inferred time limit. The algorithm also updates the appropriate context parameter with the final slack per unit of weight distributed to n.

Let I be the initial node of the activity being annotated and let $P_S(n)$ contain all paths from the node n to a final node. $t(n)$ must meet the following constraints:

- For every action n, $t(n) \geq m(n)$: the assigned time limit must be greater or equal than the minimum set by the user.
- For every path p in $P_S(I)$, $\sum_{n \in p} R(n)t(n) \leq L$: the sums of the time limits over each path meet the global time limit.

The available time "flows" from the initial node. If a node n receives $0 \leq r(n) \leq L$ seconds, every path $p \in P_S(n)$ receives $r(p) = r(n)$ seconds to distribute among its nodes. $r(n)$ is not known *a priori* except for the initial node: $r(I) = L$.

If the «GaStep» and «GaScenario» annotations are consistent with each other, then $r(p) \geq m(p)$ for every path p: the minimum time constraints of all actions are always met. $s(p) = r(p) - m(p) \geq 0$ is known as the *slack* of the path p. $s(p)$ is distributed over p according to the weight of each node: the *slack per unit of weight* initially assigned to each node is $S_w(p) = s(p)/w(p)$. When $w(p) = 0$, we assume that $S_w(p) = 0$: all nodes in p have a zero weight, so no slack can be distributed.

The algorithms must ensure that $w(p) > 0 \Rightarrow s(p) > 0$, so every path p with a non-zero weight has some slack to distribute. If this condition is not met or the annotations are inconsistent, the user should be notified and every change should be rolled back.

Definition. The algorithm is a recursive function, taking a node n and the time it receives, $r(n)$. Initially, $n = I$ and $r(n) = L$, the global time limit. Its steps are as follows:

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

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

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 note as $m(p_{Mm}(n))$. We can compute it in advance using (1). As it is recursive, we can evaluate (1) incrementally, starting from the final nodes (for which $m(p_{Mm}(n)) = 0$) and going back to the initial node in reverse topological order:

$$m(p_{Mm}(n)) = R(n)m(n) + \max\{m(p_{Mm}(g(e))) \,|\, e \in o(n)\} \tag{1}$$

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, $r(n)$, which is not known *a priori*. Instead, we 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:

$$\begin{aligned}&(a,b) \preceq_{s(L)} (c,d) \equiv \\ &\quad \forall t \in [0,L]\ \forall x \in [0,L]\ \forall y \geq 0 \\ &\qquad a+x \leq t \wedge c+x \leq t \wedge b+y > 0 \wedge d+y > 0 \Rightarrow \frac{t-(a+x)}{b+y} \geq \frac{t-(c+x)}{d+y}\end{aligned} \tag{2}$$

We can simplify (2) into:

$$a \leq c \wedge (b \leq d \vee a < L \wedge b > d \wedge (b-d)L \leq bc-ad) \tag{3}$$

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.

Like $m(p_{Mm}(n))$, $P'_S(n)$ can also be computed incrementally by traversing the graph in reverse topological order. Let n_i be a child of n and p_a and p_b be two paths in $P_S(n_i)$, so $c(p_a) \preceq_{s(L)} c(p_b)$. 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 or just as strict than any other (maximal elements according to $\preceq_{s(L)}$). We define $P'_S(n)$ as:

$$P'_S(n) = \max_{\preceq_{s(L)}} \left\{ (R(n)m(n)+M, R(n)w(n)+W) \,|\, e \in o(n), (M,W) \in P'_S(g(e)) \right\} \tag{4}$$

Note that $P_S(f) = (0,0)$, where f is a final node.

Example. Previously, we defined the algorithm and described the key optimisations performed. We will now apply the algorithm to the example in Figure 1, producing the outputs highlighted in bold. To save space, we will shorten action names to their initials: "Evaluate Order" will be simply "EO".

First, $m(p_{Mm}(n))$ and $P'_S(n))$ are precomputed:

- $m(p_{Mm}(\mathrm{CO})) = 0$, $P'_S(\mathrm{CO}) = \{(0,1)\}$.
- $m(p_{Mm}(\mathrm{PP})) = 0$, $P'_S(\mathrm{PP}) = \{(0,2)\}$.
- $m(p_{Mm}(\mathrm{CI})) = 0$, $P'_S(\mathrm{CI}) = \{(0,3)\}$.
- $m(p_{Mm}(\mathrm{SO})) = 0$, $P'_S(\mathrm{SO}) = \{(0,2)\}$.
- $m(p_{Mm}(\mathrm{EO})) = 0.4$, $P'_S(\mathrm{EO}) = \{(0.4,3)\}$.

After that, the algorithm sends the available second ($L = 1s$) into the initial node and then into EO. EO takes 0.4s and sends the remaining 0.6 seconds through the decision

node. The next action in the strictest path is CI, which takes 0.2s and sends 0.4s into PP. PP takes another 0.2s and sends the remaining 0.2s to CO.

Once the strictest path is done, we back up and proceed with the next strictest path, sending 0.4s into SO. At first, SO takes only 0.3s, but since CO received only 0.2s before, we reuse the extra 0.1s into SO. The final time limit of SO is 0.4s. We back up and continue with the empty branch for rejected orders: we are done.

As for the context parameters: `swEO` is set to 0, as $w(\text{EO}) = 0$. `swCI`, `swPP` and `swCO` are set to 0.2. `swSO` is set to 0.4: note that the initial slack per unit of weight for SO was 0.3, but after reusing the extra 0.1 seconds, it changed to 0.4.

4 Evaluation

The algorithms have been implemented using the Epsilon Object Language (EOL) [19] and integrated into the Papyrus graphical UML editors [12]. Code is available at [15]. In this section we will analyse their restrictions and performance.

4.1 Restrictions

The inference algorithms are limited in several ways. The most important restriction is that the graph formed by the nodes of the activity must be acyclic, which hinders the modelling of repetitive structures. We have partially addressed this issue by using the attribute *rep* of «GaStep» to indicate the expected number of repetitions of an action.

At first glance, the algorithm still requires to annotate each action with some knowledge from the modeller, so it would appear not to save much effort. However, the information annotated by the user on each activity only depends on the action (minimum time and weight) or control flow (probability) themselves, instead of all the paths they are part of. In addition, any sufficiently advanced tool can add the missing annotations with the default values set by the user. The time limit inference algorithm also ensures that the annotations are consistent with each other.

The algorithms do not take into account the fact that the same behaviour might be reused in several places: each action is assumed to be different from the rest. A simple and conservative solution would be simply taking the strictest constraint over all the occurrences of that behaviour. Integrating the "same behaviour" constraint would be interesting, but it might considerably increase the cost of the algorithm.

4.2 Theoretical Performance

Let us consider an activity with n nodes and $e \in O(n^2)$ edges, with $O(n)$ incoming edges in each node. The throughput inference algorithm is easy to analyse: by going back from the final nodes to the initial nodes, each node and edge in the activity needs to be visited exactly once. The throughput for the $O(n)$ join and merge nodes requires evaluating an expression in constant time over their $O(n)$ incoming edges. However, throughputs for the rest of the $O(n+e)$ nodes and edges can be computed in constant time. Therefore, a conservative upper bound for the running time of the throughput inference algorithm is $O(n)O(n) + O(n+e)O(1) = O(n^2)$. The running time does not depend on the values of the annotations.

The time limit inference algorithm is harder to analyse. Its performance depends both on the structure of the graph and the values of the annotations. For this reason, we will use a specific kind of activity to frame the analysis, which we call a *fork-join activity*. As shown in Figure 2, it has an initial node, I, followed by a sequence of f "levels". Each level has a fork node with two branches with a single action, joined before the next level. The activity has $n = 2 + 4f \in \Theta(f)$ nodes and $e = 1 + 5f \in \Theta(f)$ edges in total, and there are 2^f paths from the initial node to the final node. These activities are inexpensive to generate, as the number of nodes and edges grows linearly. At the same time, they can represent the worst case of the algorithm, since the number of paths from the initial node to the final node grows exponentially.

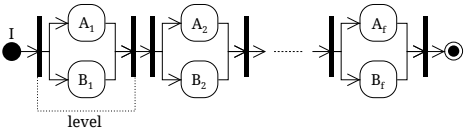


Fig. 2. Example fork-join activity with f levels

Having defined the structure of the activities, let us analyse the worst case by parts:

- Computing $m(p_{Mm}(n))$ in advance for each node always takes $O(1)O(n) = O(n)$ 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 $O(n4^f)$ in total.
- The last step depends on the number of elements of $P'_S(g(e))$ for each edge e in the graph: in the worst case, $|P'_S(g(e))| = |P_S(g(e))|$ for every node and $O(n2^f)$ operations are required.

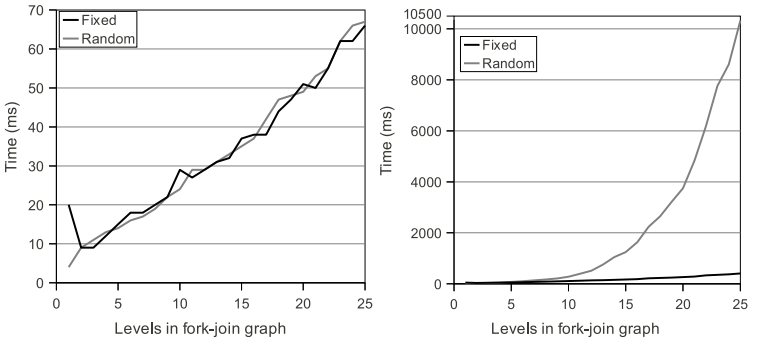
In total, we have $O(n4^f)$ operations in the worst case, which can be very expensive.

4.3 Empirical Performance

Previously, we concluded that the throughput algorithm had polynomial cost regardless of the annotations, and that the time limit inference algorithm could reach exponential cost, depending on the annotations. In this section we will study how close are the average times to this absolute worst case.

We first measured the performance of the algorithms using fork-join activities with 1 to 25 levels. We ran the algorithms on these activities requiring 1s response time when 1 request was received per second. The actions were annotated in two ways: either using a fixed minimum time limit and weight (0 and 1, respectively) or using uniformly distributed random values, so the minimum time limits were consistent and weights were between 0 and 1. To simplify the analysis, each action had *rep* set to 1.

The results are shown in Figures 3(a) and 3(b). Figure 3(a) confirms that the time required for the throughput inference algorithm grows linearly, regardless of the annotations. Figure 3(b) suggests that the average times for fixed and random annotations are quite far from the $O(n4^f)$ absolute worst case.



(a) Throughputs (10 runs fixed and random) (b) Time limits (10 runs fixed, 100 runs random)

Fig. 3. Average running times by number of levels and type of annotation

It is interesting to note that when the minimum time limit is equal to 0 in all actions, the partial order in (3) can be simplified to $a \leq c$, which is a total order. Therefore, these fixed annotations are instances of the best case of the time limit inference algorithm, in which all paths are comparable. As shown in Figure 3(b), the time limit inference algorithm required 400ms on average with a fork-join activity with fixed annotations and 25 levels.

On the other hand, using uniformly distributed random annotations resulted in much larger running times, with 10s required on average to annotate a fork-join activity with 25 levels. Nevertheless, Figure 3(b) does not grow as quickly as would be expected from the $O(n4^f)$ absolute worst case.

This suggests that removing redundant paths reduces the impact of the absolute worst case. However, its effectiveness depends on the relative magnitude of the minimum time limits and weights with regards to the global time limit L. The left operand of $(b-d)L < bc-ad$, part of (3), 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 $L = 0.5s$ and $L = 1.5s$ the space of all fork-join activities with 3 levels which contained a 2-level fork-join with 4

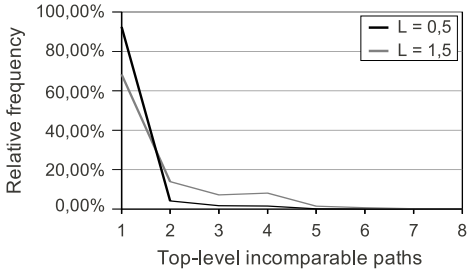


Fig. 4. Distribution of incomparable top-level paths over sampled 3-level fork-join activities, by global time limit

incomparable paths. Minimum time limits for the actions ranged from 0 to $\min\{L, 1\}$, in steps of $0.1s$. Weights ranged from 0 to 10, in steps of 1 unit. Inconsistent graphs were discarded. For each activity, we measured the number of incomparable paths at the initial node ("top-level paths"): in a 3-level fork-join activity, there can be between 1 and $2^3 = 8$ such paths.

Evaluating 1.99×10^6 fork-join activities for $L = 0.5s$ and 7.16×10^9 for $L = 1.5s$ produced the results in Figure 4. It is interesting to note that for $L = 1.5s$, while 31.842% of all 1-level fork-join activities were in the worst case, only 2.492% 2-level fork-join activities were in the worst case. With 3 levels, no fork-join activities were in the worst case with $L = 0.5s$, and only 0.047% were in the worst case with $L = 1.5s$. This suggests that the absolute worst case becomes harder to find with more complex graphs, explaining why average times did not grow exponentially in Figure 3(b). It also indicates that the worst case is more common when L grows in relation to the values in the annotations.

5 Generation of Test Cases

In previous sections, we have shown how to create the performance analysis models and how to infer the missing constraints from the existing annotations. These performance analysis models can already be useful as a way to check if a certain performance level is feasible or not, and to negotiate SLAs. In this section, we will propose another use case for the performance analysis model: test case generation.

Generating test cases directly from an abstract model such as that in Figure 1 is unfeasible, as it lacks the implementation details that are required. At the same time, it is undesirable to pollute an abstract model with these implementation details and couple it to a specific technology. To solve this situation, we propose linking the performance analysis model to implementation model, using a third generic *weaving model*. Model weaving is a well-known model management technique and is readily implemented in tools such as AMW [11] or Epsilon ModeLink [18].

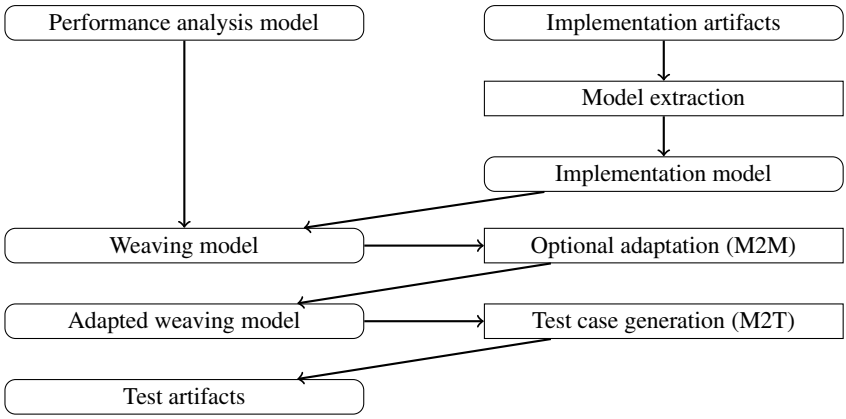


Fig. 5. Proposed approach for test case generation

The resulting approach is shown in Figure 5.

1. First, appropriate models of the implementation artifacts are extracted. Model extraction may involve parsing program code and creating an abstract syntax tree, using tools like MoDisco [8]. Alternatively, the model may already be available if the code was generated using a contract-first or model-driven approach. This is quite common for web services described using WSDL [31] documents: frameworks like Apache CXF [2] generate code from them.
2. Next, the user creates a weaving model that relates the implementation artifacts with the activities in the UML activity diagram. An activity could be linked to a JUnit test case to be reused as a performance test case, or as a web service described in a WSDL document.
3. The weaving model may not be usable as-is. In that case, an additional M2M (Model-to-Model) transformation will be required. Examples of M2T technologies include ATL [9] or ETL [19].
4. Finally, the weaving model will be used in an M2T (Model-to-Text) transformation to produce the test cases. In the case of a JUnit test case, it could be wrapped as a performance test case using a library such as JUnitPerf [10] to wrap the test case as a performance test case. For WSDL-based web services, a test plan for a performance testing tool such as Apache JMeter [14].

The main advantage to this approach is that it keeps the performance analysis model decoupled from both the methodology used to produce the implementation artifacts and the technologies used in them. Additionally, if the performance requiremens change, the algorithms can be run again and the performance test cases can be regenerated with no additional work. The main challenge is managing the additional complexity introduced by the steps in Figure 5, but it can be overcome with proper tooling.

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 the early survey in [20]. There are basically two approaches: evaluating a model of a prospective system, or measuring the performance of an implemented system. These approaches are complementary: using analytic models reduces the risk of implementing an inefficient software architecture, which is expensive to rework [25], and can find potential bottlenecks before they happen [4]. When the system is implemented, measuring its performance is more accurate, and can detect not only design issues, but also bad coding practices, unexpected workloads or platform issues. Avritzer et al. describe in [5] an interesting case study in which a simulation model was used to find the cause for a performance regression found during regular monitoring of a configuration derived from regular performance testing. Our work adapts the MARTE profile, a standard notation used for modeling non-functional requirements and creating analytic models from them, to generate the performance requirements for testing each part of the system.

Using analytic models requires highly specialised knowledge and notations. Widespread adoption of UML as a *de facto* standard notation has prompted researchers to

derive their analytic models from UML models, first with *ad hoc* annotations and later consolidating on the standard extensions to UML, such as QoS/FT [22] or SPT [21]. The survey in [29] reviews many of the approaches before MARTE replaced SPT in 2009. Since then, MARTE has been used for many purposes, such as deriving process algebra specifications [26] and extended Petri networks [32] or detecting data races [24], among others. We selected MARTE as it is based on UML, it is being actively used and offers both pre-made annotations (like SPT) and a generic framework (like QoS/FT).

Bernardi et al. have defined the Dependability and Analysis Modeling sub-profile for MARTE [7]. It has been combined with the standard GQAM and PAM sub-profiles of MARTE to evaluate the risk that a soft real-time system does not meet its time limits [6]. Our work also handles time limits, but our focus is different: we help the tester "fill in the blanks" using the available partial information. We use a model of the system to generate some of the parameters of the performance test cases.

Alhaj and Petriu generated intermediate performance models from a set of UML diagrams annotated with the MARTE profile, describing a service-oriented architecture [1]: UML activity diagrams model the workflows, UML component diagrams represent the architecture and UML sequence diagrams detail the behaviour of each action in the workflows. In our previous work, we similarly modeled workflows in a service-oriented architecture using an *ad hoc* notation based on UML activity diagrams [16]. However, our approach does not model the resources used by the system: we assume tests are performed in an environment which mimics the production environment.

There are many other recent approaches that use UML activity diagrams for generating performance test cases, without using MARTE. Avritzer et al. describe in [3] an approach for generating performance test cases considering the most common states in a system, modelled as a Markov chain. Garousi [17] uses UML sequence and activity diagrams in combination with other models to generate network stress tests for a distributed system, using evolutionary algorithms to drive the process. In general, these approaches attempt to generate test cases that cover the entire system. In comparison, our approach focuses on obtaining test cases for each part of the system, based on global requirements for the entire system.

7 Conclusions and Future Work

Software needs to meet its performance requirements in addition to its functional requirements. To achieve this goal, several approaches can be combined: the expected performance can be estimated using an early model, or the actual performance of the system can be measured. Currently, the research community is converging on the UML MARTE profile [23] as a standard notation to drive early performance and scheduling analysis. On the other hand, performance testing requires expectations to be defined for each part of the system. However, these are usually only available for high-level components: developers need to manually translate these to lower-level requirements for the smaller subcomponents.

In this work, we have adapted and improved the algorithms in [16] to operate on MARTE-annotated UML activity diagrams, inferring performance requirements from a global annotation and some local ones. One algorithm infers throughputs and has polynomial cost in relation to the number of nodes of the activity. The other infers time

limits and its worst case has exponential cost, as it may need to enumerate all paths from the initial node to the final nodes. However, further analysis of the average case suggests that this worst case is very rare, and becomes even harder to find as graphs are more complex. This is because the time limit inference algorithm discards redundant subpaths using a partial order relation.

After describing and evaluating the inference algorithms, we have propose an approach for generating concrete performance test cases for each action in the UML activity diagram. To keep it decoupled from the implementation technology and methodology, we propose weaving it to an implementation model and generating the actual test cases from the weaving model. The implementation model may already exist if using a contract-first or model-driven methodology. Alternatively, the model may be extracted from the actual code. We have selected some target technologies to implement our approach for regular JUnit functional tests and WSDL-based web services. We plan to implement the proposed approach for some of these technologies in the near future.

As for the algorithms, we intend to handle nested activities in a later version, so the user can describe the system as a hierarchy of components and infer time limits and throughputs in a top-down approach. Handling actions which are repeated in several places would be interesting, but the cost of the algorithms might increase.

Acknowledgements. This work was partly funded by the research scholarship PU-EPIF-FPI-C 2010-065 of the University of Cádiz.

References

1. Alhaj, M., Petriu, D.C.: Approach for generating performance models from UML models of SOA systems. In: Proc. of the 2010 Conference of the Center for Advanced Studies on Collaborative Research, CASCON 2010, pp. 268–282. ACM, New York (2010)
2. Apache Software Foundation: Apache CXF (November 2011), `https://cxf.apache.org/`
3. Avritzer, A., Vieira, M.E.R.: Generating performance tests from UML specifications using Markov chains. USPTO Patent Application 11/386,971 (November 2006)
4. Avritzer, A., Weyuker, E.J.: The automatic generation of load test suites and the assessment of the resulting software. IEEE Transactions on Software Engineering 21(9), 705–716 (1995)
5. Avritzer, A., Weyuker, E.J.: The role of modeling in the performance testing of e-commerce applications. IEEE Transactions on Software Engineering 30(12), 1072–1083 (2004)
6. Bernardi, S., Campos, J., Merseguer, J.: Timing-Failure risk assessment of UML design using time petri net bound techniques. IEEE Transactions on Industrial Informatics PP(99), 1 (2010)
7. Bernardi, S., Merseguer, J., Petriu, D.C.: A dependability profile within MARTE. Software & Systems Modeling 10(3), 313–336 (2009)
8. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, pp. 173–174 (September 2010)
9. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: The AMMA platform support for modeling in the large and modeling in the small. Research Report 04.09, LINA, University of Nantes, Nantes, France (February 2005)
10. Clark, M.: JUnitPerf (October 2009), `http://clarkware.com/software/JUnitPerf.html`

11. Del Fabro, M.D., Bézivin, J., Valduriez, P.: Weaving models with the eclipse AMW plugin. In: Proceedings of the 2006 Eclipse Modeling Symposium, Eclipse Summit Europe, Esslingen, Germany (October 2006)
12. Eclipse Foundation: Homepage of the Eclipse MDT Papyrus project (2011), http://www.eclipse.org/modeling/mdt/papyrus/
13. Erl, T.: SOA: Principles of Service Design. Prentice Hall, Indiana (2008)
14. Apache Software Foundation: Apache JMeter (November 2011), http://jakarta.apache.org/jmeter/
15. García-Domínguez, A.: Homepage of the SODM+T project (January 2011), https://neptuno.uca.es/redmine/projects/sodmt
16. García-Domínguez, A., Medina-Bulo, I., Marcos-Bárcena, M.: Inference of performance constraints in Web Service composition models. In: CEUR Workshop Proc. of the 2nd Int. Workshop on Model-Driven Service Engineering, vol. 608, pp. 55–66 (June 2010)
17. Garousi, V.: Traffic-aware Stress Testing of Distributed Real-Time Systems based on UML Models using Genetic Algorithms. PhD thesis, Carleton University, Ottawa, Canada (August 2006)
18. Kolovos, D.S.: Epsilon ModeLink (2011), http://eclipse.org/gmt/epsilon/doc/modelink/
19. Kolovos, D., Paige, R., Rose, L., Polack, F.: The Epsilon Book (2010), http://www.eclipse.org/gmt/epsilon
20. Lucas, H.: Performance evaluation and monitoring. ACM Computing Surveys 3(3), 79–91 (1971)
21. OMG: UML Profile for Schedulability, Performance, and Time (SPTP) 1.1 (January 2005), http://www.omg.org/spec/SPTP/1.1/
22. OMG: UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QFTP) 1.1 (April 2008), http://www.omg.org/spec/QFTP/1.1/
23. OMG: UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) 1.0 (November 2009), http://www.omg.org/spec/MARTE/1.0/
24. Shousha, M., Briand, L.C., Labiche, Y.: A UML/MARTE Model Analysis Method for Detection of Data Races in Concurrent Systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 47–61. Springer, Heidelberg (2009)
25. Smith, C.U., Williams, L.G.: Software performance engineering. In: Lavagno, L., Martin, G., Selic, B. (eds.) UML for Real: Design of Embedded Real-Time Systems, pp. 343–366. Kluwer, The Netherlands (2003)
26. Tribastone, M., Gilmore, S.: Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile. In: Proc. of the 7th Int. Workshop on Software and Performance, Princeton, NJ, USA, pp. 67–78. ACM (2008)
27. Weyuker, E.J., Vokolos, F.I.: Experience with performance testing of software systems: Issues, an approach, and case study. IEEE Transactions on Software Engineering 26, 1147–1156 (2000)
28. Woodside, M., Franks, G., Petriu, D.: The future of software performance engineering. In: Proc. of Future of Software Engineering 2007, pp. 171–187 (2007)
29. Woodside, M.: From Annotated Software Designs (UML SPT/MARTE) to Model Formalisms. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 429–467. Springer, Heidelberg (2007)
30. Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: Proc. of the 5th Int. Workshop on Software and Performance, Palma, Illes Balears, Spain, pp. 1–12. ACM (2005)
31. World Wide Web Consortium: WSDL 2.0 part 1: Core Language (June 2007), http://www.w3.org/TR/wsdl20
32. Yang, N., Yu, H., Sun, H., Qian, Z.: Modeling UML sequence diagrams using extended Petri nets. In: Proc. of the 2010 Int. Conference on Information Science and Applications, pp. 1–8 (2010)